تعلم البرمجة للمبتدئين

تأليف

Alan Gauld

ترجمة

أسامة دمراني

أكاديمية حسوب

تعلم البرمجة للمبتدئين

دليلك إلى تعلم برمجة الحواسيب والتخصص في هندسة البرمجيات

اسم الكتاب: تعلم البرمجة للمبتدئين

Book Title: Learning to Program

المؤلف: آلان غولد

Author: Alan Gauld

المترجم: أسامة دمراني

Translator: Osama Damarany

المحرر: باسمة بكلة – جميل بيلوني

Editor: Basema Bakleh – Jamil Bailony

تصميم الغلاف: سيرين ديرانية

Cover Design: Sirin Diraneyya

سنة النشر: 2023

Publication Year: 2023

رقم الإصدار: 1.0

Edition: 1.0

https://academy.hsoub.com

academy@hsoub.com

عن الناشـر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

تهدف أكاديمية حسوب إلى تعليم البرمجة باللغة العربية وإثراء المحتوى البرمجي العـربي عـبر توفـير دورات برمجة وكتب ودروس عالية الجودة من متخصصين في مجال البرمجة والمجالات التقنية الأخـرى، بالإضـافة إلى توفير قسم للأسئلة والأجوبة للإجابة على أي سؤال يواجه المتعلم خلال رحلته التعليمية لتكون معه وتؤهلـه حـتى دخول سوق العمل.

حسوب شركة تقنية في مهمة لتطوير العـالم العـربي. تبـني حسـوب منتجـات تركِّز على تحسـين مسـتقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العالم العربي، مستقل وخمسات ويعمل في فيها فريق شاب وشغوف من مختلف الدول العربية.

دورة علوم الحاسوب

مميزات الدورة

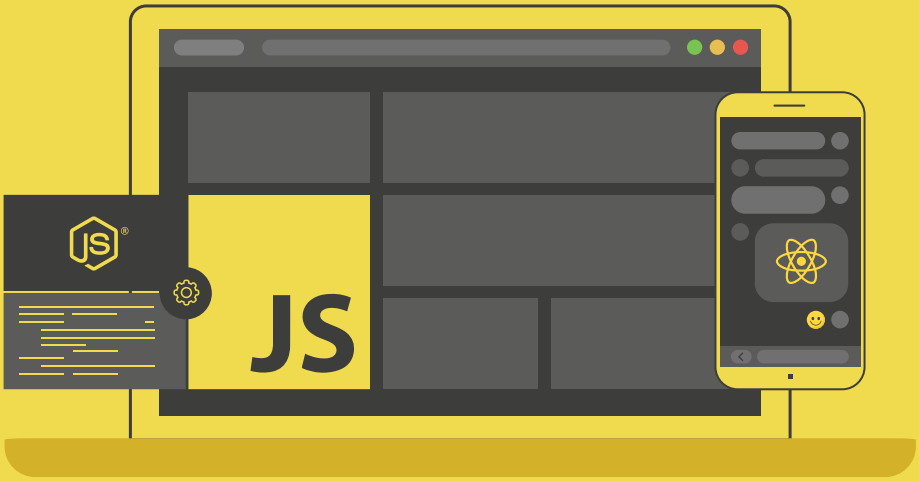
- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

المحتويات باختصار

جدول المحتويات

دورة تطوير التطبيقات
باستخدام لغة JavaScript



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

مقدمة

لا شك أن هذا الكتاب يُعد واحدًا من آلاف المصادر المتاحة لتعلم البرمجة هذه الأيام، فمنها الدورات المرئية والمساقات الجامعية والكتب والأدلة وغيرها، لكن تلك المصادر كلها لم تكن موجودة يوم وضعته وألفته، إذ أتاني صديقان لي قبل سنوات وطلبا مني مساعدتهما في تعلم البرمجة، فلما نظرت في المصادر المتاحة وقتها لم أر شيئًا يمكن إرشاد مبتدئ إليه، فكان أن كتبت هذا الكتاب.

أما الآن فقد تغير الوضع وصار لدينا كثير من المواقع الموجهة للمبتدئين، فيها دورات مرئية وكتب وغيرها، وسنشير إلى بعضها في نهاية هذا الفصل وفي مراجع الكتاب، لكن لا زلت أرى منظوري في تعليم البرمجة في هذا الكتاب فريدًا عما سواه ويناسب طائفة من الطلاب ممن لا يجدون بغيتهم في تلك المواقع، لذلك أبقيت عليه.

ولا يزال تعلم البرمجة نفسه مفيدًا لكثير من مستخدمي الحواسيب حتى لو كانوا لن يكتبوا برامج بأنفسهم، ذلك أن فهم كيفية تفكير الحواسيب معين على جعل التطبيقات أكثر منطقية وصديقة للمستخدم، فكثير من البرامج تسمح بتخصيصها أو تعديلها عن طريق كتابة بعض البريمجات الصغيرة التي تسمى بالشيفرات الجامعة أو الماكرو macro، وتلك يكتبها المستخدم حتى لو لم يكن مبرمجًا.

إضافة إلى أن لدينا الويب بما فيه من فرص لنشر موقعك الخاص الشخصي أو التجاري، وإن فعلت ستجد أنك محتاج في مرحلة ما إلى إضافة بعض الخصائص الديناميكية لصفحاتك، وهذا من البرمجة لا شك، ثم إن الإنترنت والويب يحفزان المرء على الاهتمام العام بالحواسيب، وسيقود ذلك الاهتمام لا محالة إلى الرغبة في التحكم والسيطرة على ما يراه المرء فيهما، وما ذلك مرة أخرى إلا برمجة!

وإنني -يقول المؤلف- مبرمج شبه متقاعد بخبرة تصل إلى أربعين عامًا في البرمجة، ولي خلفية في الهندسة الإلكترونية، ولا أنوي في هذا الكتاب تفضيل لغة برمجة على أختها أو التشجيع على استخدام نظام أو برنامج بعينه، رغم استخدامي عدة لغات برمجة في الشرح هنا.

محتوى الكتاب

هذا الكتـاب مـترجم عن الكتـاب Learning to Program لكاتبـه Alan Gauld والـذي يُعـد من أفضـل المراجع وأوضحها لتعلم البرمجة وقد اختارته أكاديمية حسوب بعناية لنقله للعربية.

سنتحدث في الكتـاب عن النظريـة الأساسـية الـتي بـنيت عليهـا برمجـة الحواسـيب، وبعض من تاريخهـا، والتقنيات الأساسية المطلوبة لحل المشاكل التي تواجهنا كمـبرمجين، لكن لن نتعمـق بحيث نشـرح تقنيـات أو تفاصيل خاصة بلغة برمجة بعينها، بل سنستخدم عدة لغات برمجة كي ترى بنفسك أن كل لغـة تمتـاز عن غيرهـا في مجال بعينه أو مهام بعينها، لكن مع هذا فأغلب محتوى الكتاب سيكون بلغة اسمها بايثون Python.

لمن هذا الكتاب

نتوقع أن يكون القارئ يعلم كيف يستخدم نظام التشغيل الذي لديه سواء كـان وينـدوز أو مـاك أو لينكس أو غيرها رغم أن غير المحترفين لتلك النظم سيتمكنون من التعلم أيضًـا وإن كـان بوتـيرة أبطـأ قليلًا، كمـا نتوقـع أن يفهم القـارئ بعض المفـاهيم الرياضـية الأساسـية مثـل حسـاب الأشـكال البسـيطة والإحـداثيات الهندسـية والمجموعات وبعض الجبر، أي في مستوى طالب الصف الثاني الثانوي مثلًا.

وتلك المفاهيم مهمة في بيئات البرمجة الحالية ويبنى عليهـا كثـير من المفـاهيم البرمجيـة، غـير أن العمـق المعرفي الذي نطلبه ليس كبيرًا، وإذا وجدت أن المفهوم الرياضي الذي نشرحه يصعب عليك فتخطاه إلى ما يليه من الفقرات وجرب الشيفرة كما هي، ونأمل حينئذ أن تكون قادرًا على فهم ما تنفذه الشـيفرة حـتى لـو لم تـدرك المفهوم الرياضي الذي خلفها.

كذلك يُتوقع أن يعلم القارئ كيفية تشغيل الأوامر من سطر الأوامر في نظام التشغيل، وسـيكون هـذا نافـذة دوس DOS في ويندوز، أو CMD كما يطلق عليها هذه الأيام، وهي نافذة سوداء بها محث نصي أبيض يقول لـك `C:\WINDOWS>`، وتستطيع الوصول إليه بكتابة `CMD` في نافذة تشغيل Run من قائمة ابدأ.

أما إذا كنت تستخدم لينكس فلا شـك أن تعلم مـا هي الطرفيـة Terminal، وكـذا إذا كنت تسـتخدم مـاك، فتسـتخدم برنـامج الطرفيـة أيضًـا -سـتجده في التطبيقـات Applications، ثم الأدوات Utilities-، وتوجـد اختصارات كثيرة توفر عليك وقت الكتابة إن شئت، ستجدها في قراءة ملفات المساعدة للمحث الخـاص بنظـام تشغيلك، ولن نذكرها هنا في هذا الكتاب لبعدها عن موضوعه، لكن إذا كنت تستخدم ويندوز فستجد هذا الدليل نافعًا لك، وكذلك هذا الدليل من موسوعة حسوب لمستخدمي الصدفة Shell في أشباه يونكس، لينكس وماك.

كما لن نغطي أمورًا مثل إنشاء ونسخ الملفات النصية أو تثبيت البرامج أو تنظيم الملفـات على الحاسـوب، فإذا كنت بحاجة إلى تعلم هذه الأمور فاعلم أنك لست مؤهلًا بعد لتعلم البرمجة بغض النظر عن رغبتك في ذلك، فابحث عن دليل استخدام للحاسوب أولًا، ثم عد إلى هذا الكتاب مرة أخرى حين تستطيع فهم مثل تلـك الأمـور،

واعلم أن نظمًا مثل ويندوز وماك بهما أدلة تعليمية مدمجة فيهما لتعليمك كيفية استخدامهما، أما لينكس فله كثير من المواد الشارحة له على الويب، واستعن في ذلك بمحركات البحث المختلفة إذا أردت البحث عن أمر ما.

لماذا بايثون؟

إن سبب اختيارنا للغة بايثون في شرح هذا الكتاب أنها سهلة التعلم، فبنيتها اللغوية بسيطة وبها مزايا قوية مدمجة فيها بنفس الوقت، كما تدعم كثيرًا من أنماط البرمجة، بداية من تلك البسيطة إلى البرمجة الكائنية والتقنيات الدالية Functional، إضافة إلى أنها تعمل على المنصات على اختلافها، من يونكس إلى ويندوز إلى ماكنتوش وغيرها، وأخيرًا فإن لها مجتمعًا لطيفًا ومستعد للمساعدة من مستخدميها، وكل ذلك مهم للمبتدئ الراغب في تعلم البرمجة.

لكن اللغة نفسها رغم ذلك ليست لغة للمبتدئين وحدهم، فسترى مع زيادة خبرتك أنك تستطيع استخدامها كلغة نهائية لبرنامجك أو كلغة نماذج أولية، وقل أن تجد أمرًا لا تناسبه بايثون، وإن وجد فهي أمور قليلة ونادرة.

وتأتي بايثون حاليًا بإصدارين أساسيين هما (2.7) و (3.x)، ويركز هذا الكتاب على الإصدار 3.6 رغم أن أكثر من 90 بالمائة منه سيعمل على أي نسخة بايثون بعد 3.2، وسيعمل أغلبه كذلك مع الإصدار 2.7 مع بعض التعديلات الطفيفة في تسمية الوحدات modules ومع إضافة السطر التالي في بداية ملف برنامجك أو جلستك التفاعلية:

```
from __future__ import print_function, division
```

سنذكر في الكتاب كذلك لغتي VBScript وجافاسكربت كلغات بديلة، وذلك لبيان أن التقنيات الأساسية ستعمل بغض النظر عن اللغة التي تستخدمها، وبمجرد أن تتقن الكتابة بلغة ما تستطيع الانتقال إلى غيرها بسهولة في بضعة أيام.

والسبب في اختيار هاتين اللغتين هو تعزيز إدراك المفهوم البرمجي الذي نشرحه بما أن أسلوبهما يختلف كلية عن بايثون، وإذا افترضنا أن أغلب مستخدمي الويب من المبتدئين يستخدمون حواسيب عليها نظام ويندوز، فإن هذا النظام فيه بيئة برمجية مدمجة به تسمى Windows Scripting Host أو WSH تدعم كلا من VBScript و JScript، وهي النكهة الخاصة بمايكروسوفت من جافاسكربت.

كذلك فإن أي شخص يستخدم متصفح الويب الخاص بمايكروسوفت يستخدم هاتين اللغتين داخل المتصفح، وسننظر أولًا في تشغيل VBScript و جافاسكربت داخل المتصفح، رغم إمكانية عمل جافاسكربت في أي متصفح أو نظام تشغيل، ثم نتعرض بعدها إلى WSH في أجزاء لاحقة من هذا الكتاب كزيادة.

مصادر أخرى

- توجـد مواقـع أخـرى في الـويب تحـاول تعليم البرمجـة بلغـات برمجيـة أخـرى، إضـافة إلى من يشـرحها باستخدام بايثون كما سنفعل هنا، كما توجد دورات كثيرة لأولئك الذين تعلموا البرمجة لكن يريدون تعلم لغة جديدة، لذا سأضع هنا روابط إلى بعض تلك المصادر التي أراها مفيدة ونافعة:
- كتاب البرمجة بلغة بايثون: وهو كتاب يشرح البرمجة بلغة بايثون للمبتدئين.
- توثيق لغة بايثون في موسوعة حسوب: وهو توثيق مترجم إلى العربية من التوثيق الرسمي للغة.
- الموقع الرسمي للغة بايثون، وفيه توثيق وروابط للتحميل وغيرها.
- الموقع الرسمي للغة Perl، وهي لغة منافسة لبايثون في إمكانياتها، لكنها أصعب قليلًا في التعلم.
- موسوعة حسوب ومؤسسة موزيلّا Mozilla، كمصدر للمعلومات عن لغة جافاسكربت.
- وأخيرًا، إذا كانت لديك خلفية رياضية جيدة فانظر موقع How to Design Programs المتاح أيضًا في صورة كتاب ورقي، وهو يشرح نكهة من لغة ليسـب Lisp اسـمها Scheme، وهي فعالـة جـدًا في بنـاء برامج بمنظور منهجي ونظامي.

إضافة إلى ما سبق، ستجد في قسم بايثون مئات المقالات لشروحات تركز أغلبها على تعليمك لغة بـايثون، دون التطرق إلى المصطلحات والمفاهيم التي حول البرمجة نفسها والحواسيب كما سنفعل في هذا الكتاب.

المساهمة

يرجى إرسال بريد إلكـتروني إلى academy@hsoub.com إذا كـان لـديك اقـتراح أو تصـحيح على النسـخة العربية من الكتاب أو أي ملاحظة حول أي مصطلح من المصطلحات المستعملة. يسهِّل علينا تضـمين جـزء من الجملة التي يظهر الخطأ فيها على الأقل عملية البحث، ويفضل إضافة أرقام الصفحات والأقسام أيضًا.

دورة تطوير التطبيقات
باستخدام لغة بايثون

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

الباب الأول: المفاهيم

1. ماذا تحتاج لتعلم البرمجة؟

يمكن القول إنك لن تحتاج لأكثر من اتصال بالإنترنت لتتعلم البرمجة من هذا الكتاب، ومع أن هذا يكفي من الناحية التقنية إلا أنه غير كافٍ لتعلم البرمجة، إذ يجب التنويه إلى أسلوب التفكير الذي ستتبعه في البرمجة، فلابد أن تتمتع بفضول للتعلم مع أسلوب منطقي لتفكير، وهما أمران لازمان لأي مبرمج ناجح.

تبرز أهمية الفضول في البحث عن إجابات للمشاكل، وفي قدرتك على التجربة والتنقيب في الملفات بحثًا عن الأفكار والمعلومات المطلوبة لتنفيذ مهمة معينة، كما أن التفكير المنطقي مهم لأن الحواسيب آلات غبية بطبيعتها، ولا تستطيع فعل أي شيء سوى إضافة أرقام مفردة إلى جانب بعضها البعض، وتحريك بايتات من مكان لآخر.

لقد كتب كثير من المبرمجين المهرة -لحسن الحظ- برمجيات تُخفي غباء الحواسيب هذا، لكنك بصفتك مبرمجًا ستتعرض لمواقف تواجه فيها ذلك الغباء الذي لم يعالجه أو يتعامل معه أحد قبلك، وهنا يجب أن تضع نفسك مكان الحاسوب وتفكر بدلًا عنه، وأن تعرف تحديدًا ما يجب عمله ببياناتك وزمن ذلك أيضًا. قد يكون هذا سردًا فلسفيًا لأمر يبدو تقنيًا وإلكترونيًا محضًا، غير أنك تحتاج إلى مثل تلك الفلسفة لتفهم هذا الشرح فهمًا جيدًا.

بعد ذلك يأتي الجزء الخاص بالتدريب العملي الذي ستحتاج فيه إلى كتابة الأمثلة بيدك، أو نسخها من الكتاب إلى المحرر النصي عندك، ثم تشغيلها ورؤية النتائج، وستحتاج هنا إلى تثبيت بايثون Python على حاسوبك، وإلى متصفح قادر على تشغيل لغتي VBScript وJScript، مع الإشارة إلى أن أي متصفح حديث قادر على تشغيل جافاسكربت.

1.1 لغة بايثون Python

يُعَد أحدث إصدار من لغة بايثون وقت كتابة هذه الكلمات -بلغتها الأصلية- هو 3.9، وحجم ملف تحميله من ActiveState يقارب 29 ميجا بايت لنسخة إصدار ويندوز (لا زالت ActiveState في الإصدار 3.8)، غير أنها تشمل كامل التوثيق والكثير من الأدوات التي سننظر في بعضها لاحقًا في الكتاب، لذا تأكد من اختيار النسخة الموافقة لنظام تشغيلك.

أما بالنسبة لنظام لينكس أو الأنظمة الشبيهة بيونكس عمومًا، فاطلب من مدير نظامك تثبيت النسخة المصدرية لبايثون على حاسوبك، لكن قد لا تحتاج إلى ذلك لأنها تأتي مدمجةً مسبقًا، ومثبتةً تلقائيًا في أغلب توزيعات لينكس، كما ستجدها في الإصدارات بتحزيمات موجهة للتوزيعات المشهورة مثل ريدهات Red Hat وديبيان Debian وأوبنتو Ubuntu وغيرها، بل قد تجد أن أغلب أدوات إدارة النظام في لينكس مكتوبة بلغة بايثون، ولا تقلق إذا كان إصدارك أقل من 3.6، فأي إصدار بعد 3.4 سيكون مناسبًا.

تستطيع النظر في نسخ التحميل المختلفة لاختيار ما يناسبك من موقع التحميل الرئيسي لبايثون، أما مستخدمو ويندوز وماك فقد يفضلون نسخة ActiveState.com التي تأتي غالبًا مع بعض الأدوات الإضافية المدمجة في البرنامج نفسه، وقد تكون ActiveState متأخرةً قليلًا في إصدار نسخها الجديدة بسبب برنامجهم الخاص في التحزيم والاختبار، غير أنها تستحق الانتظار.

1.2 لغة جافاسكربت ولغة VBScript

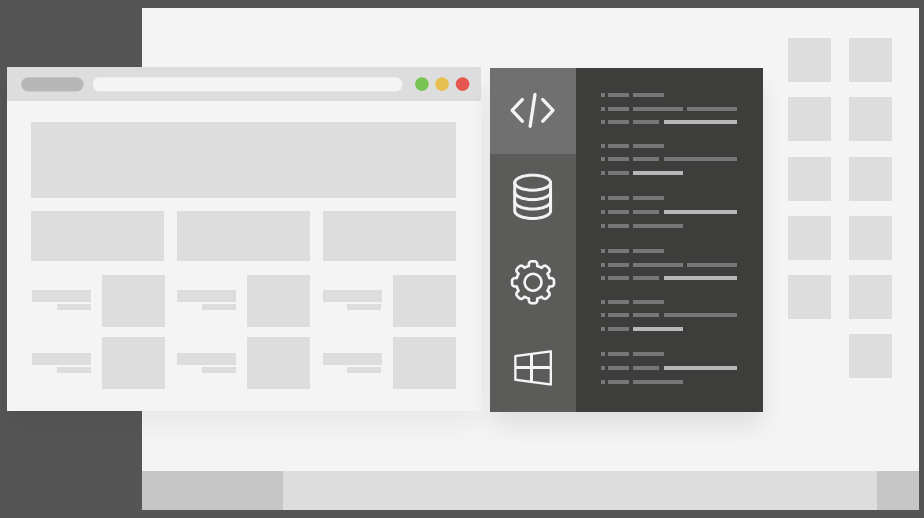
سبق وأن قلنا إن أغلب المتصفحات تستطيع تشغيل شيفرات لغة جافاسكربت دون مشاكل، لكن لغة VBScript عكس ذلك، فهي لن تعمل إلا في متصفح إنترنت إكسبلورر من مايكروسوفت، فإما أن تكونا لديك معًا إذا كنت تستخدم ويندوز، أو لن تكون لديك VBScript إذا كنت تستخدم ماك أو لينكس، والأمر الوحيد الذي عليك الانتباه إليه هنا هو أن بعض مديري الأنظمة القلقين بشأن الأمان قد يغلقون خاصية تشغيل السكربتات scripts في المتصفح لدواع أمنية، لكن هذا نادر الآن لأن أغلب المتصفحات تشغّل سكربتات جافاسكربت افتراضيًا التي أصبحت ضرورية لعمل صفحات الويب وخصوصًا تطبيقات الويب.

بخصوص لغة VBScript التي تعمل مع متصفح إكسبلورر (يُختصر إلى IE)، ضع في بالك أن هذا المتصفح قد انتهى دعمه رسميًا من مايكروسوف منتصف عام 2022 ولم يعد ضمن قائمة المتصفحات ولا القائمة المستهدفة في عملية تطوير الويب التي تستهدف متصفحات الويب في النهاية، وقد خلفه متصفح إج Microsoft Edge من مايكروسوفت مع ميزة فتح صفحات الويب ببيئة متصفح الويب القديم إنترنت إكسبلورر، أي كأن الصفحات تعمل في هذا المتصفح وسيستمر وضع التوافق هذا فترة من الزمن يُقدر حتى عام 2029 وستعلن مايكروسوفت قبل سنة من إزالة هذا الوضع رسميًا.

1.3 خاتمة

ذكرنا في هذا الفصل أنك تحتاج إلى التفكير المنطقي والفضول من أجل تعلم البرمجـة، ويمكن الرجـوع في هذا إلى مقـال الـدليل الشـامل في تعلم البرمجة من أكاديميـة حسـوب، ومقـال حـل المشـكلات وأهميتهـا في احتراف البرمجة من الأكاديمية أيضًا، كما يجب أن نـذكر أن كـل لغـات البرمجـة المـذكورة أعلاه متاحـة للتحميـل مجانًا، وبهذا لا يبقى سوى أن تأتي بذهن حاضر، مع قليل من حس الدعابة والمرح، لنبدأ البرمجة.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن

2. ما هي البرمجة؟

برمجة الحواسيب هي فن نجعل فيه الحاسوب ينفذ ما نرغب فيه بالضبط، وهي تتكون في أبسط صورها من سلسلة أوامر نعطيها للحاسوب لينفذها من أجل تحقيق هدف ما، وقد كان المستخدمون قديمًا أيام نظام دوس الخاص بويندوز، ينشئون ملفات نصيةً تحوي قوائم من تلك الأوامر، تُسمى ملفات الرُّقع أو باتش batch files (تدعى غالبًا سكربتات أو سكربت باتش)، وسبب تسميتها بذلك هو أنها تنفذ الأوامر مثل مجموعة أو رقعة واحدة، وكان امتدادها هو `.BAT`، لذا أُطلق عليها اسم ملفات بات BAT، ولا يزال بإمكاننا كتابة مثل تلك الملفات في بيئات ويندوز هذه الأيام رغم ندرة استخدامها، فمثلًا إذا كنت تكتب مستند HTML مكوَّن من ملفات كثيرة مثل الدليل الذي تقرؤه الآن، فسينشئ برنامج معالجة النصوص الذي تستخدمه نسخًا احتياطيةً من كل ملف كلما حفظ نسخةً جديدةً منه. فإذا رغبت في أن تضع النسخة الحالية من المستند -وهي الإصدارات الأخيرة من جميع ملفاته- في مجلد نسخ احتياطي `backup` في آخر كل يوم ثم حذف النسخ الاحتياطية التي لدى معالج النصوص؛ فيمكن كتابة ملف BAT بسيط لتنفيذ لك، وسيكون كما يلي:

```
COPY *.HTM BACKUP
DEL *.BAK
```

إذا كان اسم الملف هو `SAVE.BAT` فسنكتب `SAVE` في محث DOS (يطلق على المحث أيضًا موجه أوامر) في نهاية اليوم بعد انتهاء العمل، وستُحفظ الملفات وتُحذف النسخ الاحتياطية تلقائيًا، وهكذا تكون قد رأيت مثالًا عمليًا لبرنامج بسيط.

لاحظ أن مستخدمي نظام لينكس وغيره، بل حتى مستخدمي الإصدارات الحديثة من ويندوز، لديهم نسختهم الخاصة بهم من أمثلة تلك الملفات، والتي تُعرف عادةً باسم سكربتات الصدفة shell scripts، وهي أقوى بكثير من ملفات BAT الخاصة بنظام دوس، وتدعم أغلب التقنيات البرمجية التي سنتحدث عنها في هذا الفصل.

2.1 تعريف البرنامج مرة أخرى

لقد ذكرنا أعلاه مثالًا عن برنامج، لنبين بساطة مفهوم البرمجة في حد ذاته، لكن إذا خشيت عدم فهم البرمجة، فاعلم أن البرنامج ما هو إلا مجموعةً من التعليمات التي تخبر الحاسوب كيف ينفذ مهمةً ما، وهو يشبه وصفة الطعام التي تتكون من مجموعة من التعليمات التي تخبر الطاهي كيف يصنع طبقًا ما، فهي تصف مقادير المكونات (البيانات) وترتيب خطوات التنفيذ (العملية) المطلوبة لتحويل تلك المكونات إلى كعكة أو غيرها، فالبرنامج يشبهها كثيرًا في هذا.

2.2 نظرة تاريخية

نحن نستخدم لغةً للحديث مع الحواسيب، كما نستخدم لغات بيننا نحن البشر، غير أن اللغة الوحيدة التي يتحدثها الحاسوب تسمى اللغة الثنائية binary والتي توجد عدة نسخ منها، لهذا لا يعمل برنامج موجه لنظام ماك على ويندوز والعكس. ويُعَد تعلّم هذه اللغة صعبًا جدًا بالنسبة للبشر، سواءً قراءتها أو كتابة برامج بها، لذلك يجب أن نستخدم لغةً وسيطةً ثم نترجمها إلى اللغة الثنائية، تمامًا مثل شخصين من دولتين مختلفتين يتحدثان معًا وبينهما مترجم وسيط يفسر كلام كل منهما للآخر، فقد يتحدث أحدهما العربية والآخر الإنجليزية، فيترجم المترجم الكلام العربي إلى الإنجليزية وينقله إلى متحدث الإنجليزية، ثم يعكس العملية إذا تحدث الإنجليزي مخاطبًا الشخص العربي، وما سنستخدمه ليترجم لغتنا البرمجية إلى الحاسوب يسمى مترجمًا أو مفسرًا interpreter. وكما نحتاج مفسرًا لكل لغة من لغات البشر، فإننا نحتاج مفسرًا لكل لغة برمجية ليترجمها إلى لغة الحاسوب، فنحتاج مفسرًا من بايثون إلى لغة الآلة الثنائية binary، وآخر من VBScript إلى لغة الآلة كذلك، غير أن المبرمجين الأوائل كانوا مضطرين إلى كتابة الشيفرات الثنائية تلك بأنفسهم، أي بلغة الآلة التي ذكرناها قبل قليل وهي صعبة الكتابة والتعلم.

بعد ذلك أتت المرحلة التالية التي أُنشئ فيها مفسر يحول كلمات مكتوبةً بأحرف بشرية إلى ما يقابلها من اللغة الثنائية، فبدلًا من تذكر أن الرمز 001273 05 04 يعني جمع 4 إلى 5، نستطيع الآن أن نكتب شيئًا مثل ADD 5 4، وقد سهلت تلك النقلة البسيطة الكثير من عمل البرمجة والتطوير، حيث كانت تلك الأنظمة البسيطة من الشيفرات هي لغات البرمجة الأولى التي كانت تختلف باختلاف نوع الحاسوب، وتسمى باسم اللغات المجمّعة Assembler Languages، ولا زالت البرمجة بلغة التجميع مستخدَمةً هذه الأيام في بعض المهام المتخصصة، لكن هذا التطور كان بدائيًا في أسلوب إخبار الحاسوب بما يجب فعله، مثل نقل البيانات من هذا الموضع من الذاكرة إلى ذلك الموضع، وإضافة هذا البايت إلى ذلك البايت وغيرها من العمليات.

البايت Byte هو وحدة بيانات تتكون من ثمانية بتّات bits أصغر منه ومجموعة فيه، وتلك البتات تكون إما 1 أو 0، وقد استُخدم البايت في البداية لتمثيل محارف النصوص، بحيث يمثل كل بايت حرفًا واحدًا.

وبما أن هذا النمط من البرمجة لا زال صعبًا في تعلمه والعمل به لإنجاز أبسط المهام، فقد طور علماء الحوسبة مع الوقت لغات تسهل عملية البرمجة، وكأن ذلك جاء في وقته إذ تطورت المشاكل التي يواجهها

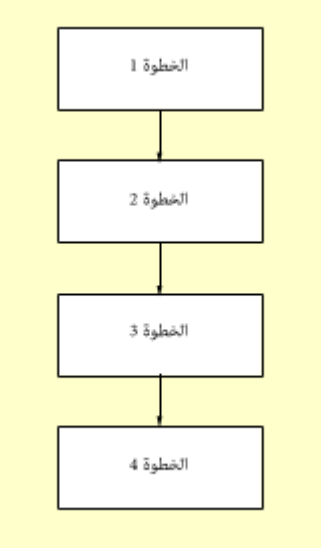
المستخدمون في ذلك الوقت بسبب تطور المهام في أعمـالهم اليوميـة، وهم يريـدون للحواسـيب أن تحـل لهم تلك المشاكل.

ولا زال هذا التنافس قائمًا، كما لا زالت لغات البرمجة الجديدة تظهر على الساحة. هـذا ومـع جعـل البرمجـة أمرًا مثيرًا يتغير كل يوم، إلا أنه يجعلك كونك مبرمجًا في حاجة إلى استيعاب المفاهيم البرمجيـة وطـرق تنفيـذها وتطبيقها في لغة واحدة بعينها، وسنناقش بعض تلك المفاهيم فيمـا يلي، لكن يجب أن تجعلهـا حاضـرةً عنـدك تعود إليها دائمًا أثناء قراءتك للكتاب.

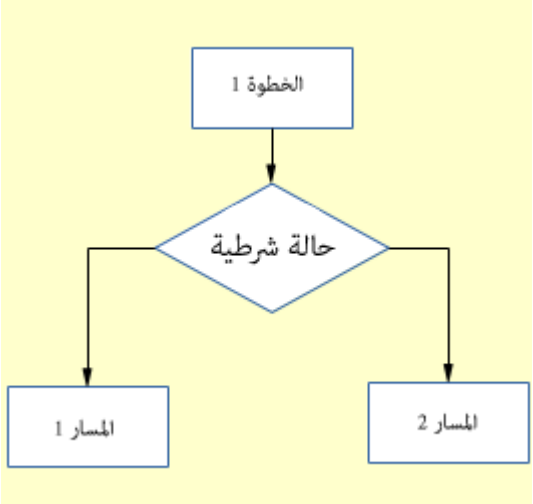
2.3 المزايا المشتركة لجميع البرامج

خـرج إدزجـر ديكسـترا Dijkstra Edsger قبـل زمن بمفهـوم اسـمه البرمجـة الهيكليـة Structured Programming، يتحدث فيه عن إمكانية هيكلة جميع البرامج بأربعة طرق، هي الآتية:

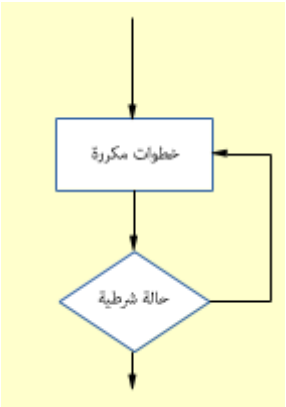
- **سلاسل من التعليمات**: يتحرك فيها البرنامج من خطوة لأخرى في تسلسل صارم.



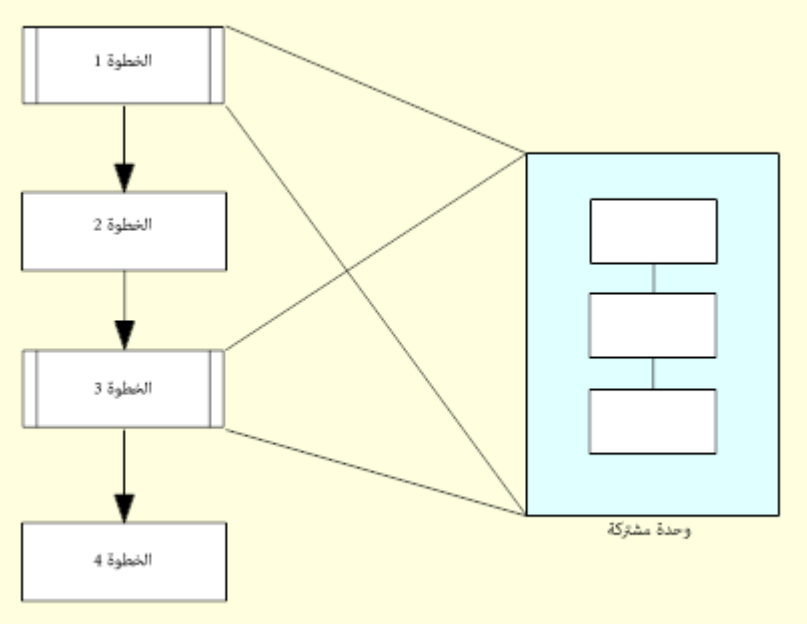
- **الفروع Branches**: هنا يصل البرنامج إلى نقطة اتخـاذ قـرار، فـإذا تحققت نتيجـة الاختبـار -أي كـانت القيمـة `true`-، فـإن البرنـامج سـينفذ التعليمـات الـتي في المسـار 1، وإذا كـانت `false` فسـينفذ الإجراءات التي في المسار 2، ويُعرف هذا بالبنية الشرطية لأن سير البرنـامج يتوقـف على نتيجـة اختبـار شرطي.



- **الحلقات التكرارية Loops**: تُكرَّر خطوات البرنامج في هذه الطريقة إلى أن نصل إلى اختبار شرطي مـا، ينتقل تحكم البرنامج بعدها من الحلقة التكرارية إلى الجزء التالي من منطق البرنامج.



- **الوحدات Modules**: هنـا ينفـذ البرنـامج تسلسـلًا متطابقًـا من الخطـوات عـدة مـرات، فتُجمـع تلـك الإجراءات في وحدة واحدة، وهي برنامج صغير الحجم يمكن تنفيـذه من داخـل البرنـامج الرئيسـي، وقـد تُسمى الوحدات بأسماء أخرى مثل الـدوال functions أو الإجـراءات procedures أو الـبرامج الفرعيـة sub-routines.



احتاجت البرامج إلى بعض المزايا الأخرى، إضافةً إلى ما سبق كي تكون مفيدة:

- **البيانات**: سننظر فيها بالتفصيل في الفصل الخامس: البيانات وأنواعها.
- **العمليات**: مثل الجمع والطرح والموازنة وغيرها، وسننظر فيها بالتفصيل كذلك في فصل البيانات وأنواعها.
- **إمكانية الإدخال والإخراج**: مثل عرض النتائج على شاشة مثلًا، وسننظر في كيفية قراءة البيانات في الفصل التاسع: قراءة المدخلات من المستخدم، والثاني عشر: التعامل مع الملفات.

وبمجرد أن تستطيع استيعاب المفاهيم أعلاه، والتعرُّف على كيفية تنفيذ اللغة البرمجية التي تستخدمها لها، فستكون قادرًا على كتابة برامج بتلك اللغة.

2.4 توضيح بعض المصطلحات

إذا قلنا إن البرمجة هي الفن الذي نجعل فيه الحواسيب تنفذ ما نريده منها، فما هو البرنامج إذًا؟

في الواقع لدينا مفهومان مختلفان عن البرنامج، أولهما من منظور المستخدم، وهو ملف تنفيذي يثبَّت على الحاسوب ثم يمكن تشغيله لتنفيذ مهمة ما، فمثلًا يقول المستخدم أنه "يشغِّل" برنامج معالجة النصوص؛ أما المفهوم الثاني فهو من منظور المبرمج، وهو هنا مجموعة من التعليمات النصية الموجهة إلى الحاسوب المكتوبة بلغة برمجة ما، ويمكن ترجمتها إلى ملف تنفيذي. لهذا يجب أن تكون مدركًا عن أي المفهومين تتحدث حين تستخدم كلمة برنامج.

يكتب المبرمجون برامجهم عادةً بلغة برمجة عالية المستوى تفسَّر إلى بايتات يفهمها الحاسوب، أي يكتب المبرمج شيفرةً مصدريةً source code؛ أما المفسر فيولد تعليمات مُصرَّفة object code، وقد يطلق على هذه

التعليمات الناتجة عن التصريف أسماء أخرى مثل ملف تنفيذي أو شيفرة تنفيذية أو محمولـة P-Code أو بـايت كود Byte Code أو شيفرة ثنائية binary code، أو شيفرة الآلة machine code.

كما يطلق على المترجم الذي يترجم لغة البرمجـة العاليـة إلى لغـة الآلـة عـدة أسـماء، فقـد يسـمى بالمفسـر interpreter أحيانًا، وبالمصرِّف compiler أحيانًا أخـرى، وتشـير هـذه المصـطلحات إلى نـوعين من التقنيـات المستخدمة في توليد الشيفرات الكائنية من الشيفرة المصدرية، فالمصرِّفات تنتج شيفرةً كائنيةً يمكن تشـغيلها مستقلةً بذاتها دون الحاجة إلى المصـرِّف في كـل مـرة يعمـل فيهـا البرنـامج، ويكـون البرنـامج في صـورة ملـف تنفيذي executable file؛ أما المفسِّرات فيجب أن تكون حاضرةً لتشغيل برامجها أثناء تنفيذها. لكن الفرق بين هذين المصطلحين صار ضبابيًا هذه الأيام بما أن بعض المصـرفات تحتـاج إلى مفسـرات تكـون حاضـرةً لتنفيـذ التحويل النهائي، كما تصرِّف بعض المفسرات شيفرتها المصـدرية إلى شـيفرة كائنيـة مؤقتـة وتنفـذها؛ أمـا من منظور المبرمجين فليس هناك فرق حقيقي، إذ تُكتب الشيفرة المصدرية وتُستخدَم أداة تسمح للحاسوب بقراءة الشيفرة وترجمتها وتنفيذها.

2.5 هيكل البرنامج

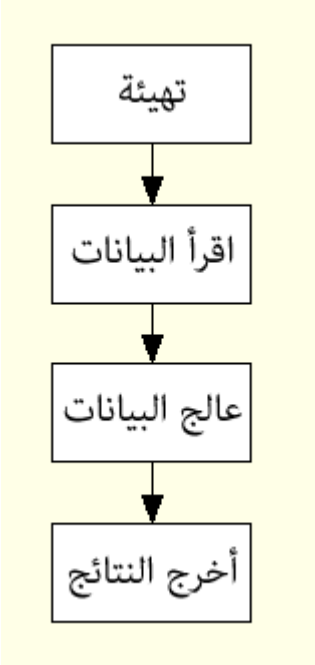
يعتمد الهيكل الدقيق للبرنامج على لغة البرمجة والبيئة التي يعمل فيها، لكن توجد بعض المبادئ الأساسية عمومًا:

- **المحمِّل Loader**: يحتاج كل برنامج أن يُحمّله نظام التشغيل إلى الذاكرة، وهذه وظيفـة المحمـل الـذي يُنشأ عادةً بواسطة المفسر.
- **تعريفـات البيانـات**: تعمـل أغلب الـبرامج بنـاءً على بيانـات، وسـنحتاج في مرحلـة مـا في شـيفرتنا المصدرية إلى تعريف نوع البيانات الذي نعمل معه تعريفًا دقيقًا، وهذا يختلف من لغة برمجة لأخرى.
- **التعليمات statements**: وهي صلب البرامج، وهي تعـدّل البيانـات الـتي نعرّفهـا وتنفـذ الحسـابات وتطبع الخرج لنا..

تتبع أغلب البرامج إحدى الهيكلين التاليين:

2.5.1 برامج باتش Batch Programs

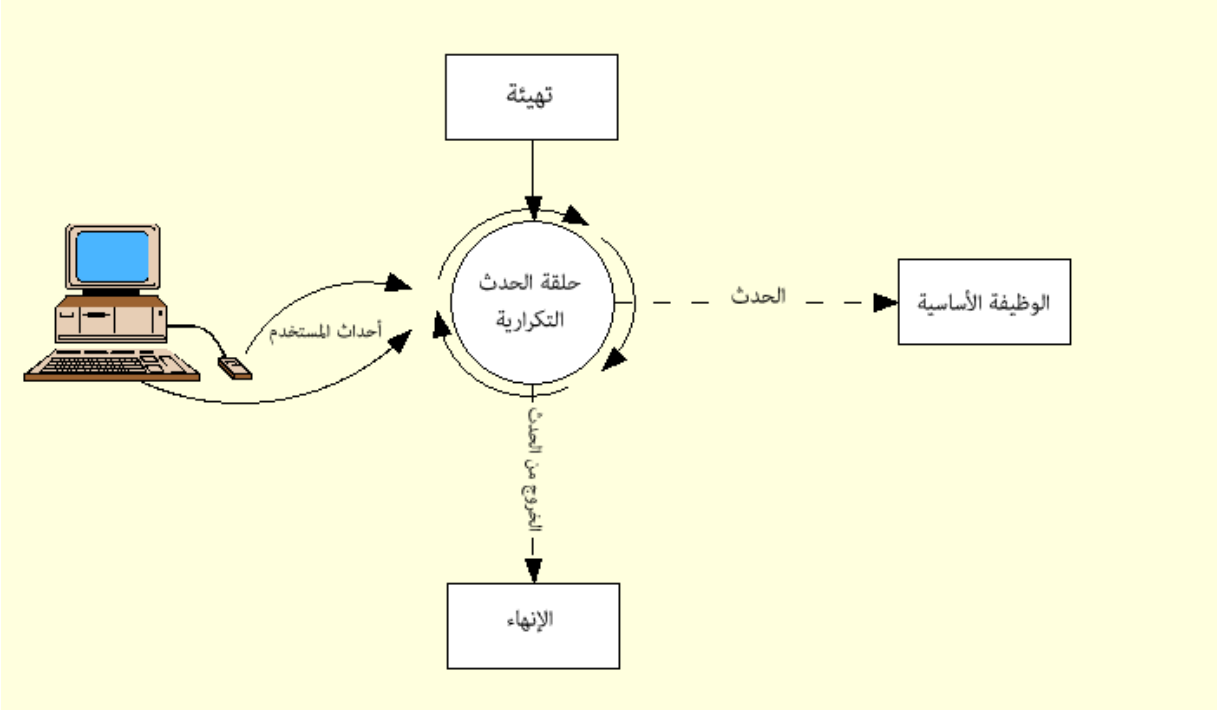
تُبدأ تلك البرامج عادةً من سطر الأوامر أو بواسطة أداة جدولة، وتتبع النمط التالي في الغالب.



يبدأ البرنامج بتعيين حالته الداخلية كأن يعين الإجماليات totals إلى صفر، ويفتح الملفات المطلوبة، وبمجرد أن يكون جاهزًا للعمل، فإنه يقرأ البيانات من المستخدم بعرض موجهات الأوامر أو المحثات على الشاشة أو من ملف بيانات، أو بالطريقتين معًا، حيث يعطي المستخدم اسم ملف البيانات ثم تُقرأ البيانات الفعلية من الملف، بعد ذلك يعالج البرنامج البيانات معالجةً تتضمن عمليات رياضيةً أو تحويلات للبيانات أو غير ذلك، أخيرًا، تُخرج النتائج إلى شاشة عرض أو تُكتب إلى ملف، وستكون جميع البرامج التي نكتبها في الأجزاء الأولى من الكتاب من هذا النوع، أي برامج باتش أو رُقع.

2.5.2 البرامج الحدثية Event driven programs

تُعَد أغلب الأنظمة ذات الواجهة الرسومية مثل وينـدوز أو أندرويـد، وأنظمـة التحكم المدمجـة Embedded control systems مثل جهاز المايكروويف لديك أو الكاميرا أو غيرهما، من البرامج الحدَثية، أي يتوقف سيرها على وقوع أحداث معينة، بحيث يرسِل نظام التشغيل أحداثًا إلى البرنامج فيستجيب لها وفقًا لوصولها إليه، وقد تكون تلك الأحداث أمورًا يفعلها المستخدم مثل نقر زر الفأرة أو ضغط زر ما، أو أمورًا يفعلها النظام نفسه مثل تعديل الساعة أو تحديث الشاشة، وتبدو البرامج الحدثية في الغالب كما يلي:



يبدأ البرنامج هنا بتعيين حالته الداخلية، ثم ينتقل التحكم إلى بيئة التشغيل (تسمى أحيانًا وقت التشغيل runtime)، ويوفر نظام التشغيل تلك البيئة في الغالب، كما ينتظر البرنامج حلقة الحدث لتلتقط تفاعل المستخدم الذي يترجَم إلى أحداث بعدها، ثم ترسَل تلك الأحداث إلى البرنامج كي يتعامل معها حدثًا حدثًا، وينفذ المستخدِم الإجراء النهائي الذي ينهي البرنامج، فيُنشَأ حدث خروج ويُرسَل إلى البرنامج. سننظر إلى حلقات الأحداث و البرمجة الحدثية في الفصل الثامن عشر: البرمجة المدفوعة بالأحداث.

2.6 خاتمة

تسمح لنا لغات البرمجة بالتحدث مع الحواسيب بلغة قريبة من لغات البشر المنطوقة، لكنها تختلف عن الطريقة التي تتحدث الحواسيب أو تفكر بها، وقد قلنا إن البرامج تعمل وفقًا للبيانات، وأنها إما برامج رُقع batch programs أو برامج حدثية Event driven تتصرف وفقًا للأحداث التي تُرسل إليها.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

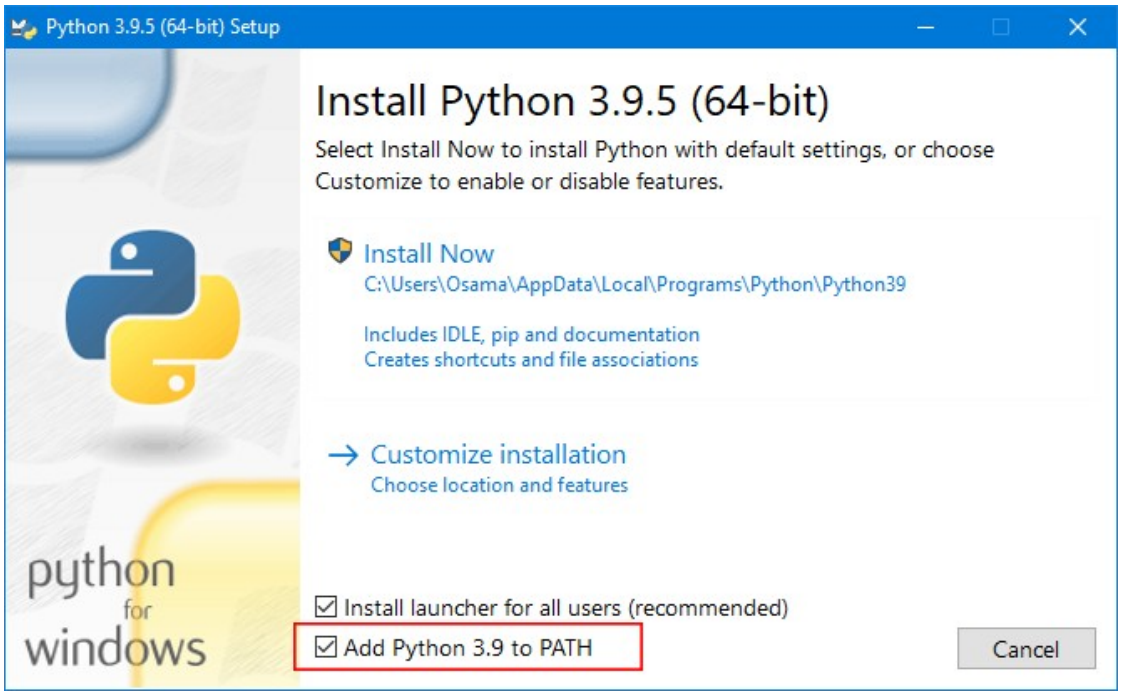
التحق بالدورة الآن

3. بداية رحلة تعلم البرمجة

رغم أننا نستخدم ثلاث لغات برمجية هنا إلا أنك لست مضطرًا إلى الكتابة بها جميعًا، فهذا سيشتتك ويؤثر سلبًا على قدرتك على التعلم، فربما تود تجربتها كلها من أجل التمرس فيها، لكن يجب أن تستقر على لغة واحدة فقط بعد الفصول الأولى، لذا اختر أيها شئت لتتمرن بها على الأمثلة التي في الكتاب (وإن كنا ننصحك ببايثون)، ثم إن مجرد قراءة الأمثلة المكتوبة باللغات الأخرى يُظهر أوجه الاختلاف والتشابه بين اللغات الثلاثة، كما يفيد في تعلم قراءة الشيفرات من ناحية أخرى، إلى جانب فهم الأفكار المشتركة بين لغات البرمجة المختلفة.

3.1 استخدام بايثون

سنفترض أنك قد ثبّتّ إحدى إصدارات بايثون على حاسوبك، فإذا لم تفعل فاذهب إلى موقع بايثون واجلب النسخة الأحدث منها واتبع إرشادات التثبيت على حاسوبك، لاحظ أن بايثون متاحة لأغلب أنواع الحواسيب المتوفرة في السوق، وستجد ملفات تثبيت لنسختي 64 بت و 32 بت من ويندوز، وكذلك نظام MacOS؛ أما لينكس فستجدها من خلال مدير الحزم في توزيعتك، فإذا لم تكن تعلم معمارية حاسوبك فاختر نسخة 32 بت، وانتبه عند التثبيت إلى خيار إضافة بايثون إلى `Path`، وهو متغير بيئة سنحتاج أن نضيف بايثون إليه كي يراها ويندوز في سطر الأوامر، بالشكل التالي:



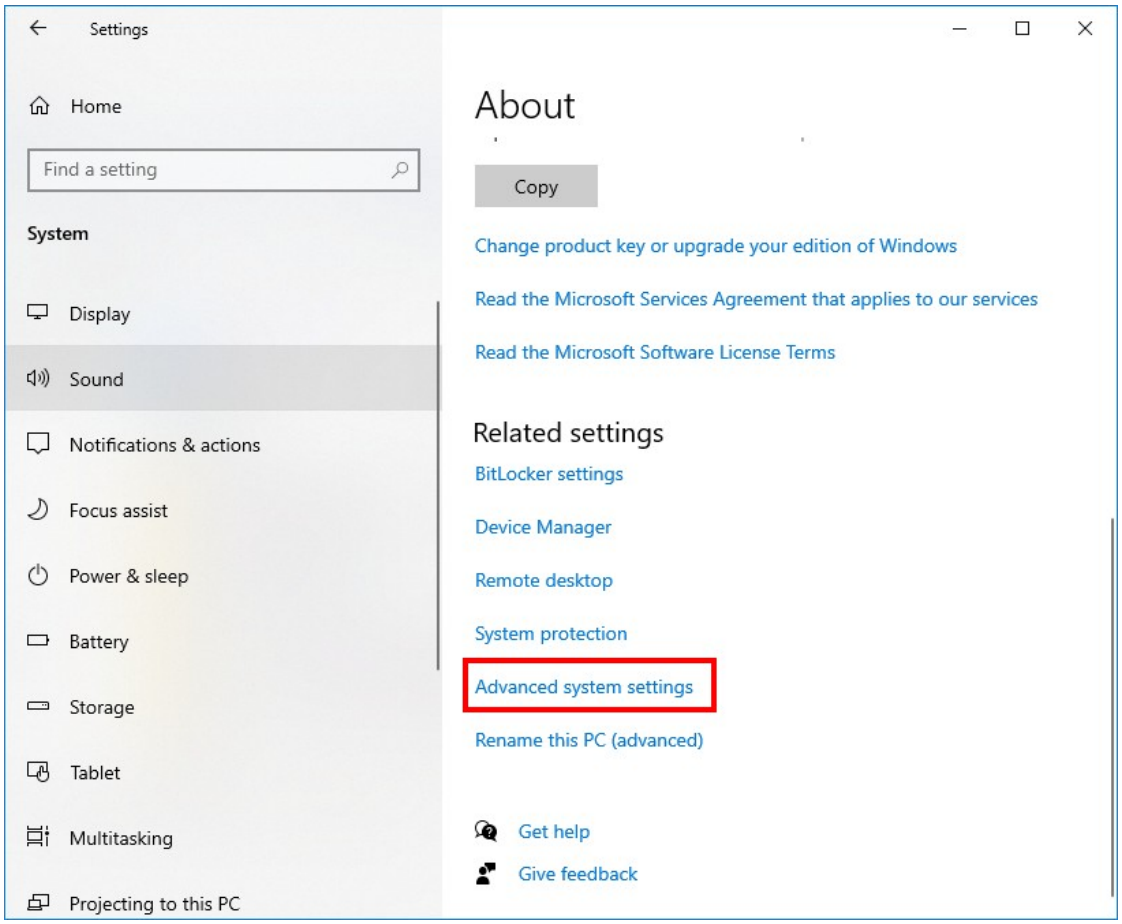
أما إذا لم تفعل ذلك أو نسيته أو لم تكن نسخة التثبيت تحتوي على ذلك الخيار، فاتبع الإرشادات التي نذكرها في الفقرة التالية.

3.2 سطر أوامر ويندوز

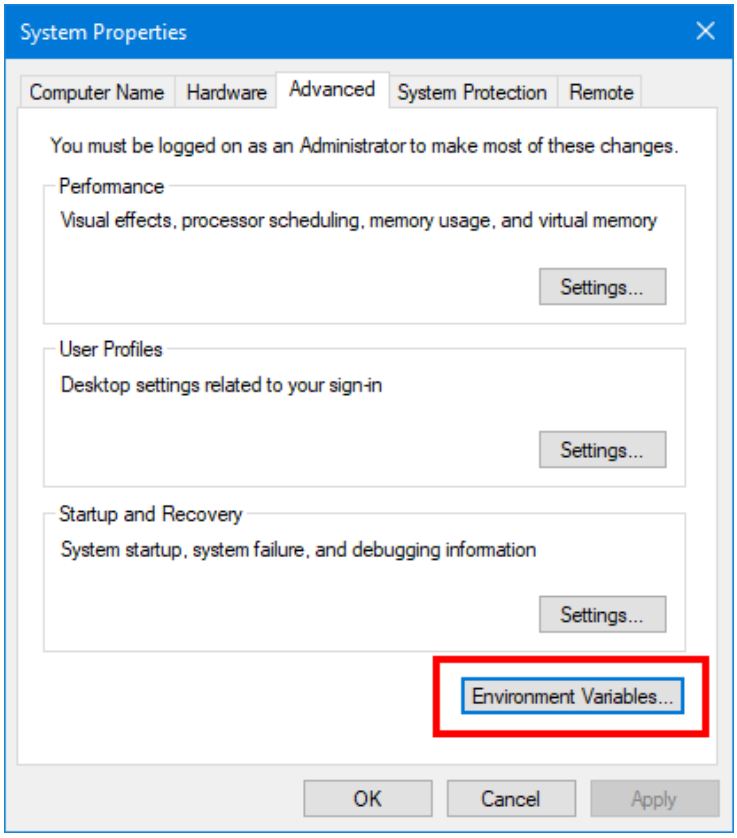
لقد بدا منذ كتابة هذا الكتاب -بنسخته الأجنبية لأصلية- أن العديد من مستخدمي ويندوز ليسوا معتادين على التعامل مع سطر أوامر MS Dos، لذا سنشرح كيف يمكن الوصول إلى تلك الأداة دون الاستغراق في التفاصيل غير الضرورية. توجد عدة طرق للوصول إلى سطر الأوامر أبسطها الضغط باستمرار على زر ويندوز وحرف R من أجل فتح نافذة Run، ثم كتابة `cmd` في مربع الحوار الذي سيظهر لك ثم الضغط على `OK`، وهنا يجب أن ترى نافذةً سوداء فيها نص أبيض مثل هذا:

```
C:\WINDOWS>
```

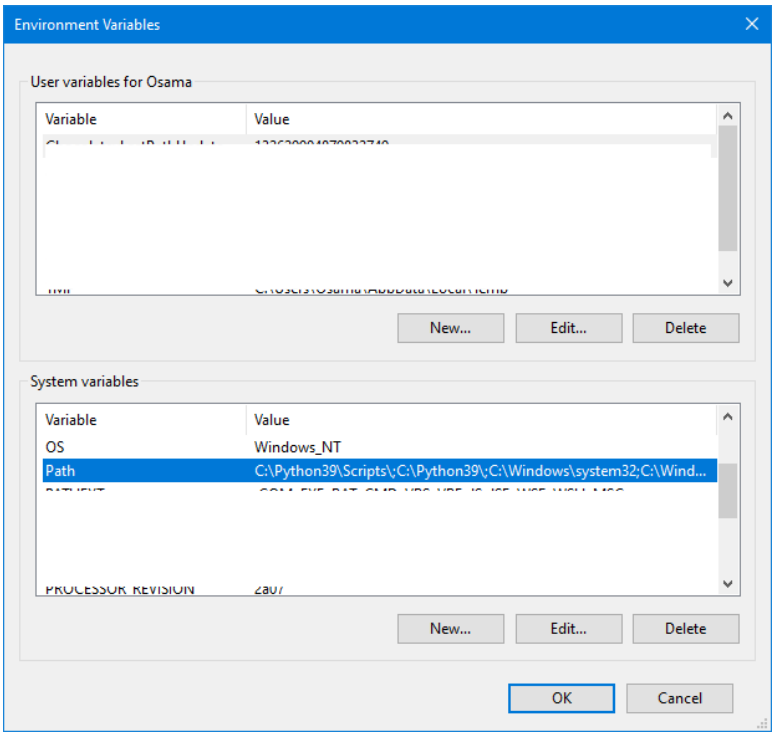
يشير السطر أعلاه إلى المجلد الذي نحن فيه، فإذا كتبنا `DIR` وضغطنا على زر الإدخال `Enter`؛ فستعرَض لنا قائمة بجميع الملفات التي في ذلك المجلد؛ أما إذا كتبنا `python` فيجب أن نرى المحث `>>>` الخاص ببايثون. قد لا تستطيع CMD العثور على بايثون بسبب بعض إصداراته الأخيرة التي لم تأتي بتلك الإعدادات، ولهذا فإنا لم تستطع إيجادها فستحتاج إلى ضبط متغير بيئة اسمه `Path`، وذلك بفتح مدير الملفات في ويندوز إما بالطريقة العادية، أو بضغط زر ويندوز مع حرف E (لفتح المتصفح)، ثم الذهاب إلى قسم This PC، والنقر بالزر الأيمن لاختيار Properties التي ستفتح نافذة About الخاصة بحاسوبك، والتي تحوي المعلومات الأساسية عنه.



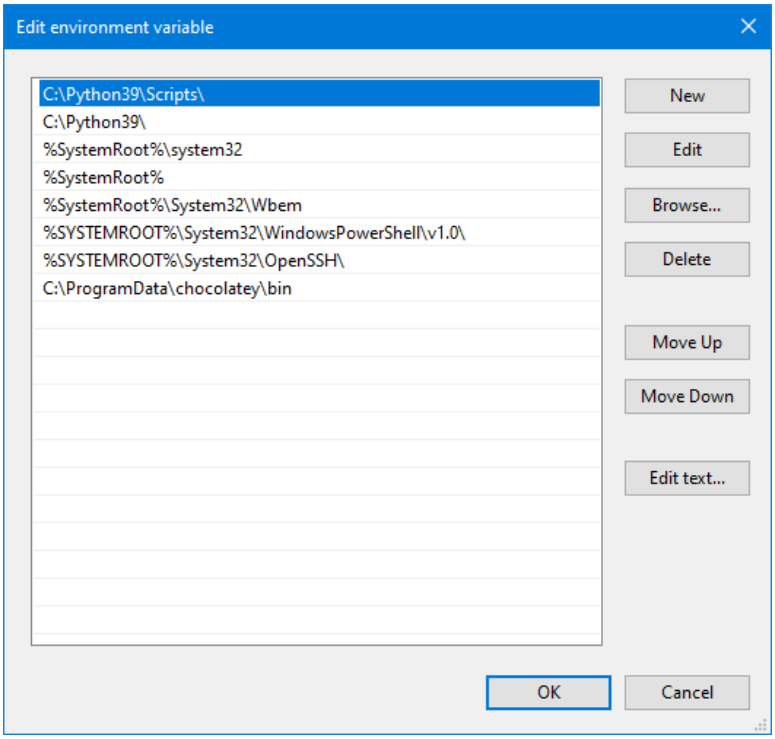
ستجد في الصفحة خيار Advaned System Settings اضغط عليه لفتح نافذة أخرى ثم اختر Advanced.



سترى زر Environment Variables في أسفل النافذة، اضغط عليه لتذهب إلى نافذة جديدة، وسترى أن Path معرَّف بالفعل كمتغير للنظام. اخترِه ثم انقر على زر Edit في يمين النافذة.



انتبه في هذه الخطوة لئلا تحذف المسار الموجود بالخطأ، فإذا مسحت شيئًا من غير قصد فاضغط زر Esc في لوحة المفاتيح، أو استخدم زر Cancel في النافذة المفتوحة لتخرج منها وتعيد المحاولة.



اذهب إلى نهاية الحقل Variable Value وأضف فاصلةً منقوطة ; ثم المسار الكامل لمجلد بايثون التنفيذي لديك، بعد ذلك اضغط زر الإدخال Enter لتكتمل العملية، فإذا لم تعرف المسار الكامل إلى بايثون فابحث في مدير الملفات عن الملف python3.exe، وستكون محتويات العمود In Folder الذي في شاشة البحث هي المسار الكامل، وقد تضطر إلى إعادة التشغيل كي يعتمد نظام التشغيل الإعدادات الجديدة.

اكتب python في سطر الأوامر لنظام تشغيلك من أي مجلد شئت وسترى المحث الثلاثي لبايثون، ونستطيع من هنا كتابة CD إلى المجلد الحامل للسكربت الخاص بنا لننتقل إليه -حيث تشير CD إلى Change Directory-، كما ستجد قائمةً من الأوامر المتاحة التي يمكن كتابتها في سطر الأوامر في نظام المساعدة المدمج في CMD نفسها، من خلال كتابة help في سطر الأوامر؛ أما إذا أردت معرفة المزيد من المعلومات عن أحد الأوامر، فاكتب اسم الأمر متبوعًا بـ /?، فإذا أردنا مثلًا عرض صفحة المساعدة الخاصة بالأمر DIR فسنكتب ما يلي:

```
C:\WINDOWS> DIR /?
```

أخيرًا يمكن إنشاء اختصار على سطح المكتب بالنقر بالزر الأيمن على سطح المكتب، ثم اختيار New جديد، ثم Short-cut اختصار، واكتب cmd في صندوق الموقع location داخل صندوق الحوار الذي يظهر لك، ثم انقر على Next التالي، ثم غير الاسم إلى شيء مثل Command Prompt أو نحوها. حين تنتهي انقر على Finish لتظهر أيقونة جديدة على سطح المكتب مثل اختصار إلى سطر الأوامر.

خذ وقتك في استعراض هذه الأداة، فرغم أنها قد تبدو بدائيةً إلا أنها أقوى مما تتخيل، وخاصةً في إنجاز المهام المتكررة، على عكس الأدوات ذات الواجهة الرسومية مثل مدير الملفات، حيث يمكنك قراءة المزيد عنها في Computer Hope للبدء في تعلمها.

3.3 عودة إلى بايثون

بما أن مستخدمي ويندوز قد ثبتوا بايثون باتباع الخطوات السابقة، فسأفترض أن مستخدمي لينكس متعودون على استخدام الطرفية، أما مستخدمي نظام ماك فيمكنهم تشغيل برنامج الطرفية بالنقر على الأيقونة كالمعتاد، وهنا يجب أن يظهر محث بايثون كما يلي:

```
Python 3.6.2 (default, Jul 29 2017, 00:00:00)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

وفي خيار بديل لسطر الأوامر ذاك؛ قد تجد اختصارًا إلى شيء اسمه IDLE، أو Python GUI في قائمة ابدأ لديك، وهو بيئة برمجة مخصصة لبايثون، وتوفر كثيرًا من الأدوات والأوامر المفيدة للمبرمجين، فإذا شغلته بدلًا من سطر الأوامر فستحصل على نافذة مستقلة لسطر الأوامر مع بعض الألوان المميزة للخطوط، وقد كتب

Danny Yoo دليلًا مفصلًا لهذه البيئة يمكنك قراءته والاطلاع عليه إذا أردت استخدامه بدلًا من سطر الأوامر العادي، وهو يكرر بعض المعلومات التي ذكرناها من قبل، لكن تكرار التعليم لا يضر، كما تستطيع الاطلاع على التوثيق الرسمي لبيئة IDLE كذلك إن شئت.

أما إذا كنت تفضل التدريب المرئي بالفيديو، فهناك الكثير من الفيديوهات المتعلقة بالبرمجة على يوتيوب، ويُرى أن تعلم المفاهيم واستخدام الأدوات من الفيديو أمر ممكن، شرط أن توقِف الفيديو مع كل سطر أوامر يُكتَب كي تتابع الشرح جيدًا، كما يوصى بكتابة الشيفرة وتشغيلها بنفسك لتتذكر ما تتعلمه، لذا فإن أردت مشاهدة الفيديو لفهم المبدأ فلا بأس، لكن عد إلى هنا مرةً أخرى واقرأ حول ذلك المفهوم واكتب الشيفرة وجربها بنفسك، ثم عدل فيها وجرب إلى أن تصل إلى توقع التغييرات التي ستحدث، وتلك هي الطريقة الوحيدة لتعلم البرمجة في رأيي.

المثير في بيئة IDLE أنها برنامج مكتوب ببايثون، فهي تبين لك قوة اللغة بمثال حي، وتوضح ما يمكن تنفيذه بها؛ أما إذا حصلت على بايثون من ActiveState أو كنت قد حمّلت نسخًا مخصصةً لويندوز -مثل حزمة PyWin32-؛ فيمكنك الوصول إلى بيئة برمجة رسومية أخرى تشبه IDLE لكنها أكثر أناقةً منها تسمى Pythonwin.

لا شك أن كلا البيئتين تسهلان البرمجة أكثر من سطر الأوامر العادي، لكننا نفضل استخدام الأدوات البسيطة في بداية التعلم لإدراك المفاهيم التي نشرحها إدراكًا أعمق.

3.4 كلمة حول رسائل الخطأ

سترى أثناء كتابة التعليمات البرمجية ببايثون رسائل خطأ لا محالةً، وستبدو مثل هذه:

```
>>> print( 'fred' + 7 )
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

لا تشغل بالك بالمعنى الدقيق لهذه الرسالة الآن، وإنما نريدك أن تنظر إلى هيكلها، فسطر `'>>> print ...'` مثلًا هو السطر الخاطئ، أما السطران التاليان فيصفان مكان الخطأ، وقد تتكون رسالة الخطأ من عدة أسطر في البرامج المعقدة، فقد تقترح كلمة `Traceback` مثلًا أن الرسالة تتضمّن أثرًا أو سجلًا لكل ما كان البرنامج يفعله عند وقوع الخطأ، وقد يكون هذا مربكًا للمبتدئين، لكن ثق أنك مع الخبرة ستسعد بوجود رسالة الخطأ تلك، وستعلم أن الأسلوب الأمثل لقراءتها حينئذ سيكون من الأسفل، وتصعد لأعلى بقدر حاجتك من الرسالة.

نعود إلى رسالة الخطأ السابقة حيث يشير `'line 1 in ?'` إلى السطر رقم 1 في التعليمة التي نكتبها، فلو كان برنامجًا أطول وكان مخزّنًا في ملف مصدري فسيحل اسم الملف الحقيقي محل `<input>`، بينما

يخبرنا السطر `'TypeError...'` بالخطأ الذي يراه المفسر، وقد يوجد أحيانًا محـرف إقحـام ^ يشـير إلى الجـزء الذي تراه بايثون خطأً، لكن هذا التقرير يكون غير صحيح في العادة، فقد يكون الخطـأ الحقيقي في مكـان سـابق للموضع الذي أشارت إليه بايثون، أو حتى قبلـه بسـطر أو سـطرين، فالحواسـيب آلات غبيـة بـأي حـال كمـا قلنا من قبل.

تُستخدم بيانات الأخطاء تلك لمعرفة ما يحدث في الشيفرة التي بين أيدينا، وقد يكون سبب الخطأ البرمجي بشريًا في الغالب، فرغم أن الحواسيب آلات غبية، إلا أنها آلات غاية في الدقة، فلعلنا أخطأنـا في كتابـة كلمـة أو تعليمة أو نسينا علامة اقتباس أو فاصلةً منقوطةً مثلًا، ولدينا عمومًا ثلاثة أنواع من الخطأ التي سنواجهها:

- الخطأ اللغوي syntax error، وهو يعني أن بايثون لا ترى أن ما أُدخـل إليهـا صـالح، وقـد يكـون سـببه نسيان علامة ترقيم أو خطأً في هجاء كلمة.
- خطأ وقت التشغيل runtime error، وذلك حين يكون البرنامج صالحًا لكن لا يمكن تنفيذه لسبب ما، كما في محاولة تنفيذ عملية غير مسموح بها أو غير صالحة، مثل طلب قراءة ملف غير موجود.
- خطأ دلالي semantic، حيث يعمل البرنامج ولا تظهر أي رسالة خطأ، لكنه يخـرج لنـا مخرجـات خاطئـةً غير التي يفترض به إخراجها، فإذا كتبنا مثلًا برنامجًا ليخبرنا بعدد كلمات الملف الذي تقرؤه الآن، وأخـرج لنا أنه يحوي خمس كلمات فقط، فستكون هذه النتيجة خاطئـةً قـولًا واحـدًا، فهنـا يكـون خطـأ البرنـامج دلاليًا.

ستكون أغلب الأخطاء التي تتعرض لها في البداية أخطاءً لغويةً أو أخطاء وقت تشغيل إلى أن تبدأ بتصـميم برامجك الخاصة، فحينها سترتكب أخطاء دلالية، وتسمى تلـك العمليـة حينئـذ بـالتنقيح debugging، وتُسـمى الأخطاء التي سترتكبها آنذاك بالعلل البرمجية bugs، لكن الترجمة الحرفية لاسمها الأجنبي bug هي عثـة، وهي حشرة صغيرة سميت الأخطاء البرمجيـة باسـمها لأسـباب تاريخيـة، فقـد كـانت أولى الأخطـاء الـتي حـدثت في الحواسيب الأولى بسبب عثة علقت داخل الحاسوب وتسببت في حرق بعض داراته الكهربائيـة، وعلى الـرغم أن الكلمة نفسها قد استُخدمت للدلالة على الأخطـاء البرمجيـة قبـل هـذا بعـدة عقـود، لكنهـا لم تثبت إلا مـع تلـك الحادثة، لكننا سنستخدم اصطلاح العلل البرمجية لقربها من المفهوم العربي للفعل ذاته.

وبالعودة إلى الخطأ الذي ارتكبناه في المثال السابق والـذي تسـبب في إخـراج رسـالة الخطـأ لنـا، فقـد كـان محاولة إضافة عدد إلى سلسلة محارف، وهـو خطـأ دلالي، لكنـه ينتج لنـا خطـأ وقت تشـغيل أيضًـا، فليس هـذا مسموحًا لك في بايثون، لذا كـانت رسـالة الخطـأ تمثـل اعتراضًـا من اللغـة على مـا فعلنـاه، فأخبرتنـا أن هنـاك `TypeError`، وسنتحدث عن هذه الأنواع في فصل البيانات وأنواعها.

وبغض النظر عن الأداة التي تريد استخدامها سواء كانت سطر الأوامـر أو بيئـة IDLE أو Pythonwin، فنحن الآن جمعنا عدتنا وجاهزون للبدء في إنشاء بعض البرامج البسيطة باستخدام بايثون.

3.5 جافاسكربت

إذا أردنا إنشاء برامج جافاسكربت في متصفح، فسنحتاج إلى مزيد من الإجراءات التي علينا تنفيـذها، حيث سنحتاج مثلًا إلى إنشاء ملف HTML نستطيع تحميله إلى متصفح، وهذا الملف لا يحوي إلا نصًا مجردًا نسـتطيع إنشاءه في محـرر نصـي بسـيط مثـل Notepad أو أي محـرر نصـي آخـر، ثم حفظـه بالامتـداد `.htm` أو `.html` وسيبدو بالشكل الآتي:

```
<html>
<body>

<script type="text/javascript">

document.write('Hello World\n');

</script>

</body>
</html>
```

سيكون الجزء الواقع بين بداية الوسم `<script>` ونهايته هو برنامجنا، ولن تعرض جميع وسـوم HTML في كل مرة أثناء شرحنا في هذا الكتاب، لذا عليك أن تنسخ هذا الملف في كل مرة مثل قالب ثم تسـتبدل الشـيفرة التي تريـد تجربتهـا بمـا هـو موجـود بين الوسـمين `<script>` و`</script>`، ثم افتح ملـف HTML ذاك في متصفح ويب عن طريق النقر عليه في مدير ملفاتك، يجب أن يشغّل نظامـك عندئـذ برنـامج المتصـفح ويحمّـل الملف الذي سيتسبب بالتبعية في تنفيذ برنامجنا.

3.6 VBScript

يمكن القول أن VBScript هي نفسها جافاسكربت لكن مع استبدال الاسـم "vbscript" الـذي في `type=` بالنص القديم الموجود "javascript"، كما يلي:

```
<html>
<body>

<script type="text/vbscript">

MsgBox "Hello World"
```

```
</script>

</body>
</html>
```

لاحظ أن متصفح إنترنت إكسبلورر الخاص بمايكروسوفت هو الوحيد القادر على تشغيل لغة VBScript، أما المتصفحات الأخرى فلا تدعم إلا جافاسكربت بما في ذلك متصفح Edge الخاص بويندوز، والذي استبدلت إنترنت إكسبلورر به، لذا يبدو أن أيام VBScript صارت معدودةً.

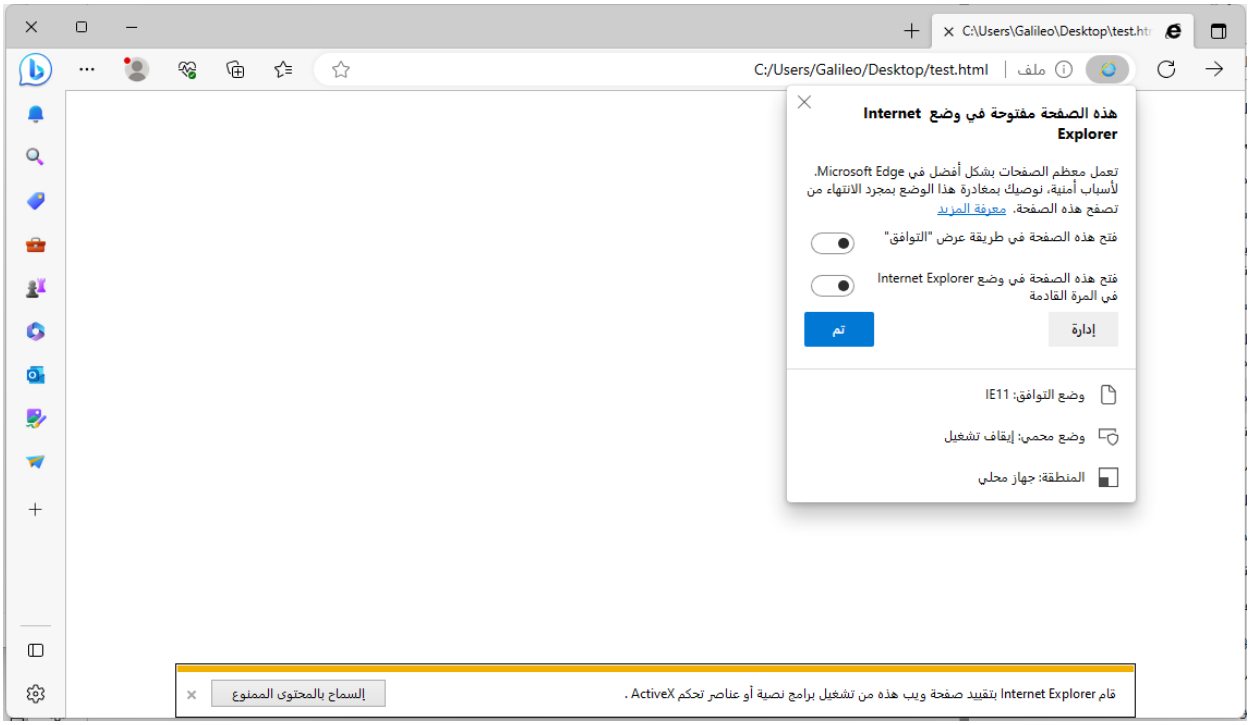
على أي حال، إن كنت تستعمل نظام التشغيل ويندوز وأردت تجريب يمكنك تفعيل وضع التوافق مع IE في متصفح Edge -كما أشرنا إليه سابقًا- بالطريقة التالية. اذهب إلى إعدادات المتصفح ثم حدد الخيار "المستعرض الافتراضي" أو Default Browser ثم اختر من قائمة "السماح بإعادة تحميل المواقع في وضع Internet Explorer (وضع IE)" الخيار "سماح":

بعد اختياره، سيُطلب منك إعادة تشغيل المتصفح، أعد تشغيله. احفظ الشيفرة التي تريد تجريبها ولتكن الشيفرة التالية في ملف باسم test.html لتجريبها:

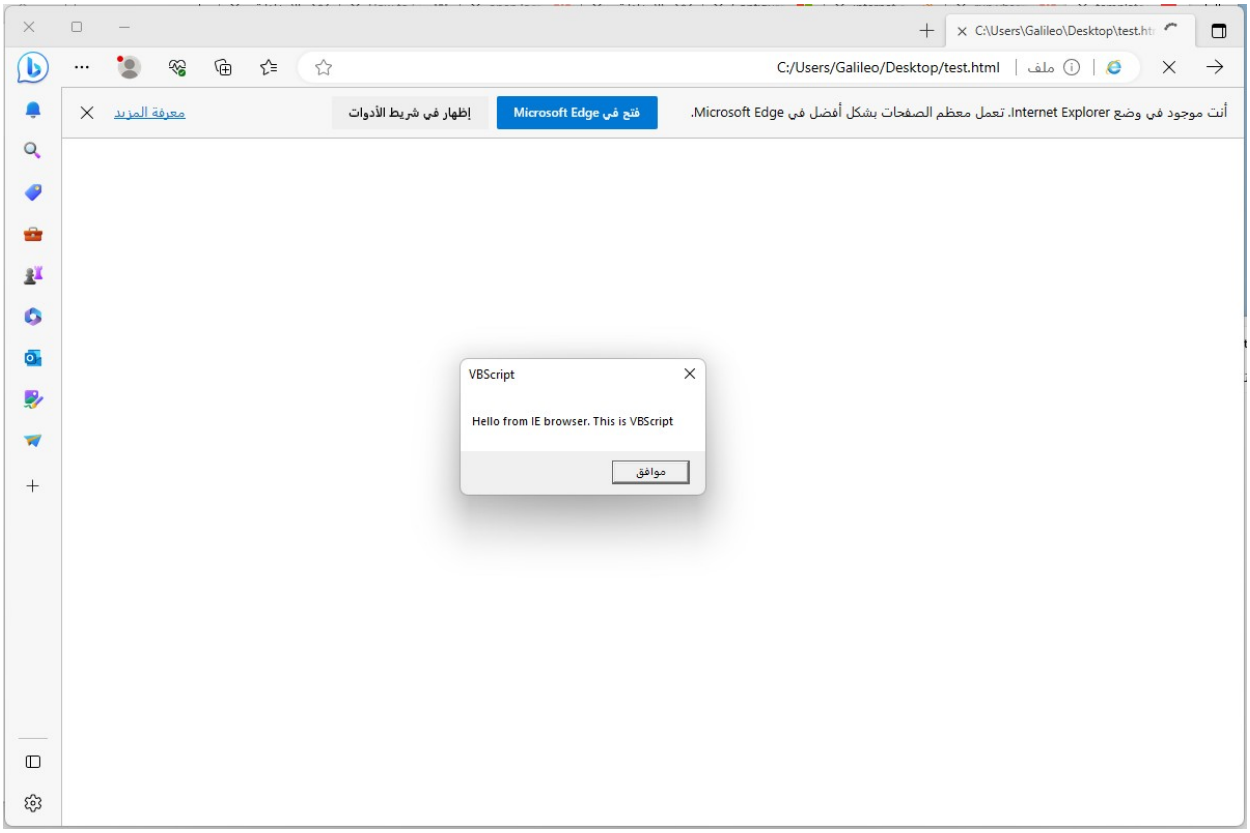
```
<script type="text/vbscript">
MsgBox "Hello from IE browser. This is VBScript"
</script>
```

افتحها في المتصفح Edge ثم افتح الخيارات واختر "إعادة التحميل في وضع Explorer":

ستظهر لك رسالة بمنع تشغيل أي برامج نصية أو سكربتات، فعل الوضع بالضغط على "السماح بـالمحتوى الممنوع":



وسيعمل السكربت وتظهر الرسالة التالية (أعد تحديث الصفحة إن لزم الأمر):



> أنصح بتفعيل الخيار "فتح هذه الصفحة في وضع Internet Explorer" في المرة القادمة (كما في الصورة قبل السابقة، إن أردت فتح الصفحة في هذا الوضع من جديد.

يمكنك إيقاف الوضع IE من خيارات ثم "إيقاف وضع Exploere على الإنترنت" وستعمل الصفحة بمتصفح Edge وفق المعتاد والذي يقبل شيفرات جافا سكريبت افتراضيًا ولا يُشغِّل شيفرات VBScript.

> ضع في بالك أن الغرض من هذا الكتاب تعلم البرمجة وليس تعلم لغة برمجة بذاتها، لذا سترى شيفرات بعدة لغات لكل فكرة أو مفهوم وهي بايثون وجافاسكربت وVBScript لذا ركز على المفاهيم واعتمد على لغة أو لغتين وطبق بهما، وكل المفاهيم التي ستتعلمها في هذا الكتاب يمكن تطبيقها على أي لغة برمجة أخرى.

3.6.1 أخطاء جافاسكربت وVBScript

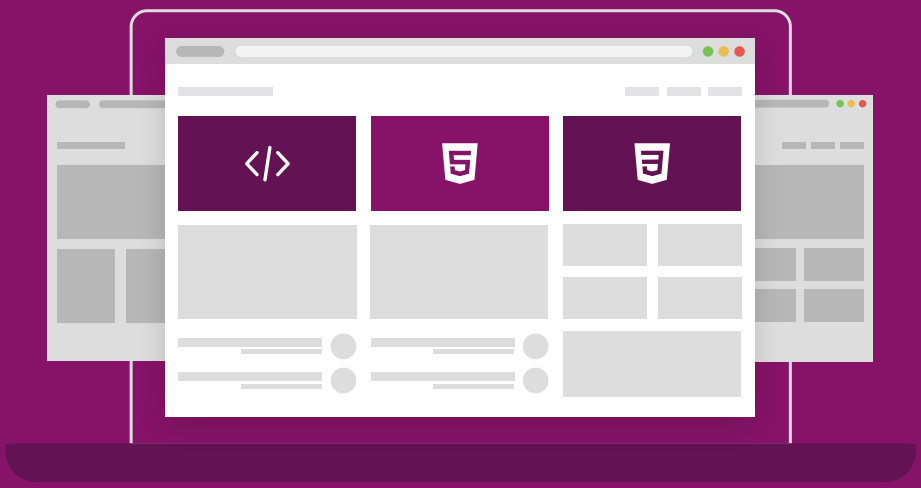
سنحصل على صندوق حوار في كل من جافاسكربت وVBScript يخبرنا برقم السطر الذي فيه الخطأ، كما سيكون لدينا عدة أخطاء غامضة غير معروفة، ويجب معاملة رقم السطر المذكور بأنه قد لا يكون دقيقًا في الغالب كما فعلنا مع بايثون، وسنحتاج بعد إصلاح الخطأ إلى إعادة تحميل الصفحة التي هو فيها.

توجد في المتصفحات أدوات تنقيح متقدمة وموجهة للمطورين المحترفين، لكنها مخبأة داخل المتصفحات ولا تناسب احتياجاتنا الحالية بعد.

3.7 خاتمة

بغض النظر عن اللغة التي ستستخدمها في متابعة الشرح في هذا الكتاب، فنرجو أن تخرج من هذا الفصل وأنت تعلم أنك تستطيع بدء بايثون بكتابة `python` في سطر الأوامر، وألا تخشى رسائل الخطأ، بل اقرأها بعناية، فهي تعطينا عادة المفاتيح التي نحتاج إليها لإصلاح الخطأ الواقع في البرنامج، غير أنها لا زالت مجرد مفاتيح، فإذا شككت في دقتها فانظر إلى الأسطر السابقة للسطر الذي تخبرك أن الخطأ فيه.

دورة تطوير واجهات المستخدم



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

الباب الثاني: الأساسيات

4. التسلسلات البسيطة

يتكون أبسط برنامج يمكن كتابته من سلسلة من بعض التعليمات statements المتتالية، وأصغرها هو الذي يحتوي تعليمةً برمجيةً واحدةً، حيث تُدخَل التعليمة عادةً في سطر واحد، كما يمكن أن تُقسم على عدة أسطر، وتعرَّف التعليمة بأنها مجموعة من الكلمات والرموز المكتوبة بلغة طبيعية، وسننظر الآن في بعضها، إذ تُظهر الأمثلة التالية ما يجب أن نكتبه في محث بايثون <<<، كما تُظهر النتيجة، ثم نشرح ما حدث.

4.1.1 عرض الخرج

يجب أن نتعلم أولًا كيف نجعل بايثون تعرض المعلومات التي نريدها، فإذا لم تعرض لنا اللغة شيئًا، فكيف نعرف ما فعله الحاسوب؟ وما فائدة الجهد الذي نبذله في البرمجة حينها إذا كنا لا نعرف هل أصبنا أم أخطأنا؟

```
>>> print('Hello there!')
Hello there!
```

انتبه إلى الملاحظة الأولى في المثال أعلاه، وهي أن المسافة التي بين المحث وبين أمر `print` ليست ضرورية، إذ ستضعها بايثون تلقائيًا إذا نسيناها، وما سنكتبه هو السطر الأول فقط؛ أما السطر الثاني فهو نتيجة البرنامج الذي كتبناه في السطر الأول، وقد عرضه لنا المفسر.

أما الملاحظة الثانية فهي أن بايثون حساسة لحالة الأحرف، حيث سنحصل على خطأ إذا كتبنا `Print` بدلًا من `print` لأن بايثون تراهما كلمتين مختلفتين، وكذلك الحال بالنسبة لجافاسكربت؛ أما لغة VBScript فلا تهتم لهذا، لكن يجب تعويد النفس على الاهتمام بحالة الأحرف كعادة برمجية حسنة لئلا تقع في أخطاء بسببها إذا انتقلت من لغة برمجية لأخرى.

وقد عرض لنا المفسر النص المكتوب بين علامتي الاقتباس لأن الدالة `print()` تخبر بايثون بعرض تسلسل المحارف `H,e,l,l,o, ,t,h,e,r,e,!` وهو ما يُعرَف في الأوساط البرمجية باسم السلسلة النصية

string، أو سلسلة المحارف النصية، حيث يجب أن تكون تلك المحارف داخل أقواس، وندل على السلسلة النصية بوضعها بين علامات الاقتباس، كما يمكن استخدام علامات الاقتباس المفردة أو المزدوجة في بايثون دون حرج، مما يسمح لنا بإدخال أحد نوعي الاقتباس في سلسلة نصية محاطة بالنوع الآخر، وهذا مفيد في حالة الفاصلة الإنجليزية العليا ':

```
>>> print("Monty Python's Flying Circus has a ' within it...")
Monty Python's Flying Circus has a ' within it...
```

أما جافاسكربت وVBScript فحساستان لنوع علامات التنصيص التي يجب استخدامها، لهذا يُنصح فيهما بالالتزام بالعلامات المزدوجة ما أمكن.

4.1.2 عرض النتائج الحسابية

```
>>> print(6 + 5)
11
```

يطبع لنا المثال أعلاه نتيجة العملية الحسابية التي في السطر الأول، والتي جمعنا فيها 5 إلى 6، وقد عرفت بايثون أن هذه المحارف أرقام وعاملتها على هذا الأساس، كما تعرفت على علامة الجمع التي بينهما، فجمعت العددين وأخرجت لنا الناتج، لذا يمكن القول أن بايثون مفيدة كحاسبة للجيب، وهو أحد استخداماتها الكثيرة جدًا، جرب الآن بعض الحسابات الأخرى واستخدم العوامل الحسابية التالية:

- الطرح (-)
- الضرب (*)
- القسمة (/)

يمكن دمج عدة تعبيرات حسابية كما يلي:

```
>>> print( ((6 * 5) + (7 - 5)) / (7 + 1) )
4.0
```

لاحظ الطريقة التي استخدمنا بها الأقواس لوضع الأعداد في مجموعات، لقد رأت بايثون تلك العملية السابقة كما يلي:

```
((6 * 5) + (7 - 5)) / (7 + 1)
=> (30 + 2) / 8
=> 32 / 8
=> 4
```

فماذا يحدث لو كتبنا نفس العمليات دون الأقواس؟

```
>>> print(6 * 5 + 7 - 5 / 7 + 1)
37.2857142857
```

حصلنا على هذه النتيجة لأن بايثون تنفذ عمليتي الضرب والقسمة قبل الجمع والطرح، لذا رأتها على النحو التالي:

```
(6*5) + 7 - (5/7)  + 1
=> 30 + 7 - 0.7143 + 1
=> 37 - 0.7143 + 1
=> 38 - 0.7143
=> 37.2857...
```

ومع أن ترتيب بايثون لمعالجة العمليات الحسابية هو نفسه الذي تقضيه قوانين الرياضيات، إلا أنه يختلف عما يجب عليك توقعه كونك مبرمجًا، لأن بعض لغات البرمجة لها قوانينها في ترتيب معالجة العمليات الحسابية، وهو ما يعرف بأسبقية العوامل operator precedence، لذا تأكد من النظر أولًا في توثيق لغة البرمجة التي تتعامل معها لترى أسلوبها الخاص؛ أما بايثون فقاعدتها العامة هي ما يقتضيه الحدس والمنطق، لكن هذا له استثناءاته، مما يجعل استخدام الأقواس خيارًا آمنًا لضمان حصولنا على النتيجة التي نريدها، خاصةً عند التعامل مع حسابات طويلة مثل التي رأيناها في المثال السابق، ومن الأمور التي يجب ملاحظتها أننا إذا استخدمنا عامل القسمة / فسنحصل على النتيجة الدقيقة للعملية، كما يلي:

```
>>> print(5/2)
2.5
```

فإذا أردنا الحفاظ على القسم الصحيح للناتج فقط فنستخدم عاملي قسمة متتاليين // لنحصل على العدد الكامل في النتيجة، وستطبع بايثون ناتج القسمة كما يلي:

```
>>> print(5//2)
2
```

أما إذا أردنا الحصول على باقي عملية القسمة فنستخدم محرف النسبة المئوية %:

```
>>> print(5%2)
1
>>> print(7//4)
1
>>> print(7%4)
```

```
3
```

يُعرَف العامل % باسم عامل الباقي modulo، وقد يكون في بعض لغات البرمجة على الصورة MOD أو نحوها، أما في بايثون فنستطيع الحصول على ناتج القسمة والباقي معًا باستخدام الدالة divmod():

```
>>> print( divmod(7,4) )
(1, 3)
```

هذه العمليات الحسابية للأعداد الصحيحة مهمة للغاية في البرمجة، فمنها نستطيع تحديد كون العدد زوجيًا أم فرديًا من خلال قسمته على 2 ورؤية الباقي (يكون زوجيًا إذا كان الباقي صفرًا)، كما يلي:

```
>>> print( 47 % 2 )
1
```

فنحن نعلم الآن أن العدد 47 هو عدد فردي، وقد يكون هذا أمرًا بدهيًا إذ يمكن معرفة هذا دون تكبد عناء لغة برمجية وانتظار خرج عملية فيها، لكن ماذا لو كنا نقرأ بيانات من ملف، أو كان المستخدم يدخل تلك البيانات، ثم يكون على برنامجنا معرفة هل العدد فردي أم زوجي من تلقاء نفسه، عندئذ لن نستطيع التدخل هنا لتقرير ذلك بالنظر في كل عدد، بل سنستخدم عامل الباقي كما فعلنا ليتحقق من المطلوب بدلًا منا، وتلك حالة واحدة فقط من الحالات التي تبرز فيها أهمية هذه العمليات، لذا تدرب عليها إلى أن تتقنها.

4.1.3 دمج السلاسل النصية والأعداد

```
>>> print( 'The total is: ', 23+45)
The total is:  68
```

لقد طبعنا سلاسل نصيةً أوأعدادًا فيما سبق، أما الآن فسندمج الاثنين في تعليمة واحدة، فاصلين بينهما بفاصلة إنجليزية , ويمكن توسيع تلك الميزة بجمعها مع إحدى خصائص بايثون في إخراج البيانات، وهي سلسلة التنسيق format string:

```
>>> print( "The sum of %d and %d is: %d" % (7,18,7+18))
The sum of 7 and 18 is: 25
```

تحتوي سلسلة التنسيق على علامات %، غير أنها تختلف عن عامل الباقي الذي تحدثنا عنه من قبل، بل يكون لها معنىً خاص حين تُستخدم في سلسلة كالتي بين أيدينا، وحالات الاستخدام المتعددة تلك تعني أن علينا قراءة المكتوب بانتباه لتحديد سياقه، ثم تحديد وظيفة % فيه، حيث يخبر الحرف d الذي بعد محرف % بايثون بوجوب وضع عدد عشري مكانه، والذي نحصل على قيمته -التي سنستخدمها لتحل محله- من القيم الموجودة داخل التعبير الذي بين الأقواس، والذي كتِب بعد علامة % المنفردة في نهاية العبارة، ومن المهم أن يكون عدد القيم التي في القوس النهائي مطابقًا لعدد علامات % الموجودة في السلسلة النصية.

تدريب

جرب التدرب على عدة صور من السطر السابق والبيانات التي تليه كي تفهم القاعدة فهمًا أفضل.

توجد حروف أخرى يمكن وضعها بعد علامة %، ويستعمَل كل منها لغرض مختلف، منها:

- %S للسلاسل النصية.
- %x للأعداد الست عشرية hexadecimal.
- %0.2f للأعداد الحقيقية التي لها منزلتان عشريتان كحد أقصى.
- %04d لإزاحة العدد بأصفار قبله ليكون من أربعة منازل عشرية.

انظر توثيق بايثون للمزيد من المعلومات.

ونستطيع طباعة أي كائن بايثون باستخدام دالة الطباعة، حتى لو كانت النتيجة على غير ما كنا نريد، فربما تكون وصفًا لنوع الكائن، لكننا نستطيع طباعة ذلك على أي حال.

4.1.4 أسلوب تنسيق السلسلة النصية

لقد غيرت بايثون أسلوب تنسيق السلاسل النصية في إصدارها الثالث عما ذكرناه في الفقرة السابقة إلى أسلوب قوي -على الرغم من تعقيده- باستخدام عملية format، مع استمرار استخدام الأسلوب القديم، وهو ما سنستخدمه في هذا الكتاب، يمكن النظر في التوثيق إذا أردت النظر في الأسلوب الجديد، ولكن عمومًا لا يختلف الأسلوبان عن بعضهما في الحالات البسيطة، فإذا نفذنا المثال السابق بالأسلوب الجديد فسيبدو بالشكل:

```
>>> print( "The sum of {:d} and {:d} is: {:d}".format(7,18,7+18)
The sum of 7 and 18 is: 25
```

كما ترى فقد استبدلنا القوسين المعقوصين {} بعلامات %، وقد نُسقت رموز أنواع البيانات تنسيقًا مختلفًا، كما يجب هنا استدعاء العملية format الخاصة بالسلسلة النصية التي تمرر القيم، تمامًا كما في المرة السابقة. يحتوي الأسلوب الجديد على بعض الخصائص الأخرى أيضًا لتحسين التخطيط العام وتمرير البيانات، لكننا لن نذكرها هنا وإنما يمكن الاطلاع عليها في توثيق بايثون، وننصحك بتأجيل النظر فيها إلى أن تتمرس في فصول هذا الكتاب كي تستوعب الشرح التقني لها.

4.1.5 زيادة إمكانيات اللغة

```
>>> import sys
```

إذا كتبنا السطر أعلاه في محث بايثون ثم ضغطنا على زر الإدخال، فلن نرى شيئًا على الشاشة، غير أن هذا لا يعني عدم حدوث شيء ما في الخلفية، ولشرح ما حدث بعد كتابة هذه الأمر سننظر أولًا إلى معمارية بايثون نفسها، وحتى لو لم تكن تستخدم بايثون فتابع الشرح، إذ ستكون ثمة آليات مشابهة في اللغة التي تستخدمها.

عند بدء لغة بايثون تتاح لنا عدة دوال وأوامر، وهذه الأوامر مدمجة في اللغة وتسمى بالمضمَّنات built-ins، لأنها مبنية داخل نسيج اللغة نفسها، غير أن بايثون تستطيع توسيع قائمة الدوال المتاحة بدمج وحدات توسيع extension modules فيها، وهو يشبه شراء المرء لأداة جديدة من متجر العُدد والآلات ليضيفها إلى مجموعته المنزلية، وفي مثالنا فإن الآلة هي `sys`، وقد وضعتها العملية `import` داخل صندوق الآلات الذي سنستخدمه؛ أما حقيقة ما يفعله هذا الأمر فهو إتاحة "أدوات" جديدة على شكل دوال للغة بايثون، تُعرَّف داخل وحدة اسمها `sys`، وتلك طريقة توسيع بايثون لتنفيذ أي شيء غير مدمج في النظام الأساسي، وستجد أكثر من مئة وحدة في المكتبة القياسية standard library التي تحصل عليها مع بايثون، كما يمكنك إنشاء وحداتك الخاصة واستيرادها واستخدامها مثل الوحدات التي وفرتها بايثون عند تثبيتها بالضبط، وسنعود إلى هذا الأمر لاحقًا. كذلك ستجد مزيدًا من الوحدات التي يمكن تحميلها من الإنترنت، فإذا بدأت مشروعًا لا تغطيه الوحدات التي في المكتبة القياسية، فانظر في الإنترنت أولًا فلعلك تجد شيئًا يساعدك.

4.1.6 الخروج السريع

لننظر الآن في كيفية استخدام تلك الأدوات التي أدخلناها، فإذا كتبنا الأمر التالي في محث بايثون؛ فسنجعل بايثون تنهي نفسها وتخرج، لأننا نكون قد نفّذنا الدالة `exit` المعرَّفة في وحدة `sys`.

```
>>> sys.exit()
```

لاحظ أننا نخرج من بايثون عادةً بكتابة محرف نهاية الملف End Of File واختصاره EOF في محث بايثون، وهو CTRL+Z على ويندوز أو CTRL+D على أنظمة يونكس Unix؛ أما إذا كنت تستخدم بيئة تطوير فتخرج من قائمة File ثم Exit، وإذا حاولنا تنفيذ هذا في أداة تطوير مثل IDLE؛ فإن الأداة تنتبه لمحاولة الخروج وتعرض رسالةً تقول شيئًا ما حول SystemExit، لكن لا تشغل بالك بها، فهذا يعني أن البرنامج يعمل وأن الأداة تحاول توفير الوقت عليك لئلا تبدأ من الصفر مرةً أخرى.

لاحظ أن `exit` لها أقواس حولها، وذلك لأنها دالة معرفة في وحدة `sys`، حيث سنحتاج إلى وضع هذه الأقواس عند استدعاء دالة بايثون حتى لو لم تحتوي الأقواس نفسها على أي شيء، فإذا كتبنا `sys.exit` دون الأقواس، فستستجيب بايثون لنا بإخبارنا أن `exit` دالةٌ، بدلًا من تنفيذ الدالة نفسها.

أخيرًا، لاحظ أن التعليمتين الأخيرتين مفيدتان عند استخدامهما معًا، أي أننا سنكتب ما يلي للخروج من بايثون بدلًا من EOF:

```
>>> import sys
>>> sys.exit()
```

هكذا نكون قد كتبنا تسلسلًا بسيطًا من تعليمتين، ونكون قد خطونا قليلًا نحو البرمجة الحقيقية.

4.1.7 استخدام جافاسكربت

لا توجد طريقة سهلة في جافاسكربت لنكتب الأوامر ونراها تنفَّذ مباشرةً كما في بايثون، لكن نستطيع كتابة جميع الأوامر البسيطة أعلاه في ملف HTML واحد ثم تحميلها إلى المتصفح، وسنرى كيف تبدو حينئذ في جافاسكربت:

```
<html><body>
<script type="text/javascript">
document.write('Hello there!<br />');
document.write("Monty Python\'s Flying Circus has a \' within it<br
/>");
document.write(6+5);
document.write("<br />");
document.write(  ((8 * 4) + (7 - 3)) / (2 + 4)  );
document.write("<br />");
document.write( 5/2 );
document.write("<br />");
document.write( 5 % 2 );
</script>
</body></html>
```

سيكون الخرج كما يلي:

```
Hello there!
Monty Python's Flying Circus has a ' within it
11
6
2.5
1
```

نستخدم `document.write` لإخراج النص إلى نافذة المتصفح، وهذا يكافئ تقريبًا دالة `print` في بايثون، وسنرى قريبًا طريقةً مختلفةً قليلًا في VBScript لعرض المخرجات في المتصفح في المثال التالي.

لاحظ كيف اضطررنا إلى كتابة `<br />` كي نجبر البرنامج أن يعرض الخرج التالي في سطر جديد، وذلك لأن جافاسكربت تكتب مخرجاتها في صورة HTML، والتي تعرض السطر بأقصى عرض تسمح به نافذة المتصفح لديك، فإذا أردنا قسرها على إنهاء السطر وبدء سطر جديد، فسنستخدم رمز HTML الخاص بالسطر الجديد، وهو `<br />`.

لاحظ كيف هرّبنا محارف الاقتباس المفردة بوضع شرطة مائلة خلفية `\` قبل الاقتباس، سنشرح ذلك بالتفصيل حين نتحدث عن السلاسل النصية في فصل البيانات وأنواعها.

VBScript 4.1.8

كما ذكرنا بخصوص جافاسكربت؛ فيجب أن ننشئ ملفًا لوضع أوامر VBScript فيها ثم نفتحه في المتصفح، وإذا كتبنا الأوامر السابقة باستخدام VBScript فستبدو كما يلي:

```
<html><body>
<script type="text/vbscript">
MsgBox "Hello There!"
MsgBox "Monty Python's Flying Circus has a ' in it"
MsgBox 6 + 5
MsgBox ((8 * 4) + (7 - 3)) / (2 + 4)
MsgBox 5/2
MsgBox 5 MOD 2
</script>
</body></html>
```

سنرى في الخرج كثيرًا من الصناديق الحوارية، يعرض كل منها خرجًا من أحد الأسطر التي في البرنامج، فإذا أردنا أن نجعل جافاسكربت تخرج لنا مثل تلك الصناديق الحوارية فسنستخدم `alert("Hello There!")` بدلًا من `document.write("Hello there!<br>")`، فقد استخدمنا `alert` هنا بدلًا من `MsgBox` التي في VBScript.

لاحظ أننا لا نستطيع بدء سلسلة نصية باستخدام علامة اقتباس مفردة في VBScript، لكن يمكننا إدراج اقتباسات مفردة داخل سلاسل نصية ذات علامات اقتباس مزدوجة، باستخدام الدالة `Chr` التي إذا أعطيناها رمزًا لمحرف أعادت لنا ذلك المحرف، وبما أن هذا المثال يبدو فوضويًا للغاية، لنلق نظرةً على المثال التالي الذي يوضح كيفية عملها:

```
<script type="text/vbscript">
Dim qt
qt = Chr(34)
MsgBox qt & "Go Away!" & qt & " he cried"
</script>
```

لمعرفة رمز أي محرف تريده؛ تستطيع الاسترشاد بهذا الموقع، وأخذ الرمز العشري decimal منه، أو بالنظر في خريطة المحارف التي يوفرها نظام التشغيل الخاص بك، إذ توفر أغلب نظم التشغيل المشهورة بريمجًا لذلك؛ أما إذا لم ترد استخدام هذا ولا ذاك لسبب ما؛ فاستخدم الجزء التالي من لغة جافاسكربت واستبدل المحرف الذي تريده بمحرف الاقتباس المزدوج الذي في المثال:

```
<script type="text/javascript">
var code, chr = '"'; // put the character of interest here
code = chr.charCodeAt(0);
document.write("<br />The character code of  " + chr + "  is " +
code);
</script>
```

لا تشغل بالك الآن بمعنى الأرقام التي في المثال، إذ سنعود إليها في الفصل التالي، وإنما سقناها الآن من أجل استخدامها إذا اضطررنا إلى إيجاد قيمة محرف ما.

4.2 خاتمة

تلك كانت نظرتنا الأولى على البرمجة، ولعلها كانت سهلة الفهم والإدراك، لكن سنحتاج إلى أن ننظر في أنواع البيانات التي سنقابلها في البرمجة وسنحتاج إليها قبل الشروع في البرمجة الحقيقية، كما سنرى الأمور والعمليات التي سنجريها على تلك البيانات.

وقد عرفنا في هذا الفصل أن البرنامج قد يكون صغيرًا للغاية، لذا فلا مانع من أن يكون مجرد أمر واحد، وأن أسلوب بايثون في إجراء العمليات الحسابية هو نفسه الأسلوب المتبع في الرياضيات، وإذا أردنا الحصول على نتيجة كسرية فيجب أن نستخدم أعدادًا كسريةً كذلك، وأننا نستطيع دمج النصوص والأرقام باستخدام عامل التنسيق %، وأخيرًا عرفنا أننا نخرج من بايثون بكتابة `import sys; sys.exit()`.

دورة تطوير تطبيقات الويب
باستخدام لغة Ruby

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

5. البيانات وأنواعها

يحتاج أي عمل إبداعي إلى ثلاثة مكونات أساسية هي: الأدوات التي يُنجَز بها، والخامات التي يتكـون منهـا، والأساليب والتقنيات التي تُستغل الأدوات والخامات بها لإنجاز ذلـك العمـل، فـأدوات الرسـم مثلًا هي الفـرش والأقلام ولوحات الرسم، كما أن من أساليب الرسم الخلط والرش وغيرها، ثم تأتي الخامـات الـتي هي الـدهانات والأوراق والماء ونحو ذلـك. وبصـورة مماثلـة نجـد أن أدوات البرمجـة هي لغـات البرمجـة ونظم التشـغيل وعتـاد الحواسيب التي ستعمل عليها الـبرامج؛ أمـا الأسـاليب والتقنيـات فهي هياكـل الـبرامج الـتي تحـدثنا عنهـا في الفصل السابق، والخامات هي البيانات التي نعدل فيها ونعالجها، وهي ما سننظر فيه في هذا الفصل.

هذا الفصل طويل بسبب طبيعة البيانـات نفسـها، لكننـا كتبنـاه بحيث لا تجب قراءتـه كلـه من أجـل إدراك الفهم الكامل للبيانات، بل تستطيع البدء فيه من البداية التي نمر فيها على الأنواع البسـيطة للبيانـات ثم كيفيـة معالجة تجميعات العناصر، وهنا تستطيع ترك الفصل والانتقال إلى ما يليـه، ثم العـودة إلى هنـا مـرةً أخـرى إلى القسم الذي نبحث فيه بعض الأمور المتقدمة.

5.1 البيانات Data

من المعلوم بالمشاهدة في عالم البرمجة أن البيانات هي أكثر المصطلحات المستخدمة حاليًـا، رغم قلـة من يفهمها على حقيقتها، وما يهمنا في هذا الكتاب في موضوع تعريف البيانات نفسها؛ هو أن تعلم أنها الحقائق أو الأرقام التي يمكن استخلاص النتائج والمعلومات المفيدة منها، ورغم أن هذا يكاد يكون تسـطيحًا لمفهومهـا، إلا أنه يعطينا نقطةً نبدأ منها على الأقل، وسنرى فيما يلي إن كنا نستطيع توضيح ذلك بـالنظر في كيفيـة اسـتخدام البيانات في البرمجة.

البيانات هي تلك المعلومات الخام التي يعدّل فيها البرنامج ويغيرها، فلا تستطيع البرامج تنفيـذ أي وظيفـة ولا مهمة ذات قيمة بغير تلك البيانات، وتتوقف أنواع التعـديلات الـتي تجريهـا الـبرامج على البيانـات على تلـك

الأنواع نفسها، فكل نوع له عدد من العمليات التي يمكن إجراؤها عليه، فقد رأينا مثلًا أننا نستطيع جمع الأعداد معًا، إذ تحدث عملية الجمع على النوع العددي من البيانات، مما يعني أن البيانات تأتي في عدة صور وأشكال، وسننظر في كل نوع من أشهر تلك الأنواع، إضافةً إلى العمليات المتاحة على ذلك النوع.

5.2 المتغيرات Variables

تخزَّن البيانات في ذاكرة الحواسيب، ويمكن تشبيه تلك الذاكرة بذاكرة مكاتب البريد القديمة، إذ توضع صناديق صغيرة على حوائط تلك المكاتب لتصنيف الرسائل البريدية، ويجب كتابة الوجهة التي تُرسَل إليها رسائل كل صندوق على الصندوق نفسه، إذ لا قيمة للرسائل من دون وجهة تُرسل إليها، وهنا نُسقط مثالنا على البرمجة، إذ إن المتغيرات هي العناوين التي تكتَب على الصناديق داخل ذاكرة الحاسوب.

بهذا نكون قد عرفنا نوع البيانات الآن، لكننا لا نستطيع تعديلها قبل أن تكون لنا صلاحية الوصول إليها، وهذه هي وظيفة المتغيرات، فنحن ننشئ نسخًا من أنواع البيانات ونسندها إلى المتغيرات، والنسخة instance هي مجرد جزء من البيانات وليكن 3 أو 5 مثلًا، وهي نُسخ لأعداد؛ أما المتغير فهو إشارة إلى منطقة محددة في ذاكرة الحاسوب تحتوي على البيانات.

قد تتطلب بعض لغات البرمجة أن يُصرَّح عن المتغير declared ليطابق نوع البيانات التي يشير إليها، فإذا حاولنا إسناد نوع خاطئ من البيانات إليه، فسنحصل على خطأ، ويفضل بعض المبرمجين ذلك النوع المسمى بالكتابة الثابتة static typing، لأنه يحول دون ارتكاب العلل البرمجية bugs الدقيقة التي يصعب اكتشافها، على أن جميع اللغات التي نستخدمها هنا مرنة إلى الحد الذي نستطيع معه إسناد أي نوع من البيانات إلى المتغيرات، وهو ما يُعرف بالكتابة الديناميكية dynamic typing.

تتبع أسماء المتغيرات قواعد محددةً تختلف باختلاف لغة البرمجة التي تستخدمها، فلكل لغة قواعدها للأحرف التي يُسمح أو لا يُسمح بها، كما تشترط بعض اللغات مثل جافاسكربت وبايثون أن ينتبه المبرمج إلى حالة الأحرف التي يكتبها، بحيث تختلف `python` عن `Python` مثلًا، كما توجد لغات أخرى مثل VBScript لا تهتم لهذا، لكن يفضل أن يُتبَّع أسلوب ثابت في تسمية المتغيرات لتجنب الأخطاء الشائعة عند الانتقال من لغة لأخرى، ومن الأساليب المستخدمة في تسمية المتغيرات هو بدء اسم المتغير بحرف صغير، ثم استخدام حرف كبير لبداية كل كلمة تالية في اسمه، كما يلي:

```
aVeryLongVariableNameWithCapitalisedStyle
```

لن نتحدث هنا عن القواعد التي تحكم المحارف المسموح بها في لغات البرمجة التي سنستخدمها، لكن إذا كنت تستخدم الأسلوب الذي في المثال؛ فستجنب نفسك كثيرًا من المشاكل في هذا الشأن. ومن الأساليب المستخدمة في تسمية المتغيرات أيضًا فصل الكلمات في اسم المتغير بشرطة سفلية، كما في `another_variable_name`؛ أما في لغة بايثون فيأخذ المتغير نوع البيانات المسندة إليه، ويحتفظ بهذا النوع ويحذرك إذا حاولت خلط البيانات بطرق تختلف عنه، كأن تحاول إضافة سلسلة نصية إلى عدد مثلًا، وهي

نفس حالة المثال الذي عرض لنا رسالة الخطأ في الفصل السابق، ويمكن تغيير نوع البيانـات الـتي يشـير إليهـا المتغير عن طريق إعادة الإسناد إلى المتغير reassigning.

```
>>> q = 7  # صار q عددًا الآن
>>> print( q )
7
>>> q = "Seven"  # أعد إسناد q ليكون سلسلة نصية
>>> print( q )
Seven
```

لاحظ أن المتغير q ضُبط ليشير إلى العدد 7 ابتداءً، وقـد حافـظ على تلـك القيمـة إلى أن جعلنـاه يشـير إلى سلسلة المحارف "Seven"، وعلى هذا نستنتج أن متغيرات بايثون تحافـظ على النـوع الـذي تشـير إليـه، لكننـا نستطيع تغيير ذلك عن طريق إعادة الإسناد إلى المتغير، وتلك هي الكتابة الديناميكية التي ذكرناها قبل قليل.

ويمكن التحقق من نوع المتغير باستخدام الدالة type() كما يلي:

```
>>> print( type(q) )
<class 'str'>
```

تُفقد البيانات الأصلية عند إعادة الإسناد، وتمسحها بايثون من الذاكرة ما لم يكن ثمة متغير آخر يشير إليهـا، وهذا المحو يُعرف باسم جمع المهملات Garbage Collection، والذي يمكن تشبيهه بموظف غرفة البريد الذي يأتي مرةً واحدةً كل فترة ويزيل أي طرود في صناديق البريد لا توجد عليها ملصقات، فإذا لم يسـتطع العثـور على مالك لها أو لم يوجد عنوان على الطرد؛ فسيرميها في المهملات.

لننظر الآن في بعض الأمثلة على أنواع البيانات.

5.2.1 متغيرات جافاسكربت و VBScript

تختلـف جافاسـكربت عن أختهـا VBScript قليلًا في طريقـة اسـتخدام المتغـيرات، فكلاهمـا تـرى وجـوب التصريح عن المتغير قبل استخدامه، وهي خاصية تشترك فيها اللغـات المصـرَّفة compiled مـع اللغـات ذات الكتابة الثابتة statically typed، بل حتى في اللغات الديناميكية مثل تلك التي نستخدمها، حيث نستفيد من التصريح عن المتغيرات في حالة حدوث خطأ إملائي عند اسـتخدام متغـير، إذ سـيدرك المـترجم أن متغـيرًا غـير معروف قد استُخدم، وسيرفع خطأً؛ أما عيب هذا الأسلوب فهو الحاجة إلى مزيد من الكتابة من قِبل المبرمج.

I. VBScript

يصرَّح عن المتغير في لغة VBScript باستخدام التعليمـة Dim، وهي اختصـار لكلمـة Dimension أو بُعـد، وهي إشارة إلى الجذور الأولى للغة VBScript في لغة BASIC ولغات التجميع من قبلهـا، حيث كـان يجب علينـا

إخبار المجمِّع بكمية الذاكرة التي سيستخدمها المتغير -أو أبعاده-، وظل الاختصار موجودًا حتى الآن. ويبدو التصريح عن المتغير في VBScript كما يلي:

```
Dim aVariable
```

نستطيع بمجرد التصريح عن المتغير أن نعين قيمًا له كما فعلنا في بايثون، ويمكن التصريح عن عدة متغيرات في تعليمة `Dim` واحدة بسردها جميعًا مع فصلها بفواصل إنجليزية `,`، كما يلي:

```
Dim aVariable, another, aThird
```

ويبدو الإسناد بعدها كما يلي:

```
aVariable = 42
another = "هذه جملة قصيرة."
aThird = 3.14159
```

قد نرى كلمةً مفتاحيةً أخرى هي `Let`، وهي من جذور لغة بيزك BASIC أيضًا، لكننا لن نراها إلا نادرًا لقلة الحاجة إليها، وهي تُستخدم كما يلي:

```
Let aVariable = 22
```

ولن نستخدمها على أي حال في هذا الكتاب.

ب. جافاسكربت

نستطيع التصريح عن المتغيرات مسبقًا في جافاسكربت باستخدام الكلمة `var`، كما يمكن التصريح عن عدة متغيرات كما في VBScript في تعليمة `var` واحدة:

```
var aVariable, another, aThird;
```

كما تسمح جافاسكربت بتهيئة -أو تعريف- المتغيرات مثل جزء من تعليمة `var`، كما يلي:

```
var aVariable = 42;
var another = "A short phrase", aThird = 3.14159;
```

يوفر هذا قليلًا من وقت الكتابة، لكنه لا يختلف عن أسلوب VBScript مع المتغيرات والمكون من خطوتين. يمكن التصريح عن متغيرات جافاسكربت وتهيئتها من غير استخدام `var`، بنفس الطريقة المتبعة في بايثون:

```
aVariable = 42;
```

غير أن المتعصبين لجافاسكربت يرون أن استخدام var أفضل، لذا سيكون هذا أسلوبنا في الكتاب أيضًا.

نأمل أن تكون تلك النبذة المختصر عن متغيرات جافاسكربت وVBScript قد أوضحت الفرق بين التصـريح عن المتغيرات وبين تعريفها، أما متغيرات بايثون، فتعريفها يعني ضمنيًا التصريح عنها كذلك.

5.2.2 أنواع الكتابة: صارمة أم متغيرة أم ثابتة

لقد أشرنا إلى الكتابة الثابتة والديناميكية فيما سبق من الفصل، وتوجد أنواع أخرى من الكتابة مثـل الكتابـة الصـارمة strict -أو القويـة أحيانًـا أخـرى-، والكتابـة الفضفاضـة loose أو الضـعيفة، وسـنحاول هنـا أن نوضـح الاختلافات بينها.

تُنشأ المتغيرات مثل أنواع بيانات مرنة أو ثابتة، ويقال أن النوع الثابت مكتوب كتابـةً ثابتـةً أي أنـه لا يتغـير أبدًا-، أمـا المتغـير الـذي قـد يشـير إلى عـدة أنـواع من البيانـات فيُكتب ديناميكيًـا، لـذا تشـير البيانـات الثابتـة والديناميكية إلى أنواع البيانات التي يمكن تخزينها داخل المتغير. ويمكن دمج المتغيرات في العمليات المختلفة عند البرمجة، حيث ستتطلب كـل عمليـة احتـواء متغيراتهـا على أنـواع بعينهـا، مثـل الأعـداد أو المحـارف، فـإذا رفضت اللغة السماح لعملية ما باستخدام أنواع خاطئة؛ فسيقال أن هذا النوع من الكتابة كتابةٌ صارمة.

تحاول بعض اللغات إجبار القيم على الأنواع الصالحة المسموح بها، بينما تسـمح لغـات أخـرى بـأي نـوع ثم تفشل العملية وتخرِج خطأً، وتلك هي الكتابة الفضفاضة، وعلى هذا تشير كل من الكتابـة الصـارمة والفضفاضـة إلى أسلوب التحقق من أنواع المتغيرات المستخدَمة في العمليات، وبالنسبة للغات التي سنستخدمها في هـذا الكتاب، فجميعها تستخدم كتابةً ديناميكيةً قويةً.

5.3 الأنواع الأساسية للبيانات

تسمى أنواع البيانات الأساسية primitive data types بهذا الاسم لأنها أبسط أنواع البيانات الـتي يمكن معالجتها والتعامل معها، أما الأنواع الأعقد من البيانات فما هي في الواقع إلا تجميعـات من الأنـواع الأساسـية، فهي اللبنات الأساسية التي تكوَّن منها جميع الأنواع الأخرى، وبناءً عليه تكون هي أساس الحوسبة كلها، وتشمل الأنواع الأساسية الأرقام والأحرف ونوعًا آخر اسمه النوع البولياني boolean type.

5.4 سلاسل المحارف Strings

لقد رأينا هذه السلاسل من قبل، وهي أي سلسلة من المحارف التي يمكن طباعتها على الشاشة، وقد توجـد فيها محارف تحكم لا تُطبع ولا تُرى على الشاشة.

تُعرض تلك السلاسل في صورة أحرف، لكن الحاسوب يخزنها في صورة تسلسلات من الأرقام، ويطلـق على ربط تلك الأرقام بالمحارف اسم الترميز encoding، وتوجد العديد من الترميزات المختلفة، رغم أن المنتشر حاليًا منها هو واحد من ثلاثة ترميزات معرَّفة بواسطة معيار الترميز الموحد Unicode Standard، وسننظر في الترميز

الموحد أو اليونيكود في فصل لاحق، لكن يجب أن تعلم أن غرضه توفير مخطط ترميز يمثل أي حرف من أي أبجدية في أي مكان في العالم، إضافةً إلى المحارف الخاصة التي تحتاجها الحواسيب مثل الأقواس والتحكم والتهريب وغيرها. لن نعير هذا الترميز كثير الانتباه أثناء العمل، لكن قد نحتاج إلى التحويل بين الرموز الرقمية والأحرف، كما في مشكلة الاقتباسات في مثال VBScript من الفصل السابق.

ويمكن تمثيل السلاسل النصية في بايثون بعدة طرق:

- باستخدام علامات اقتباس مفردة:

```
'هذه سلسلة نصية'
```

- أو باستخدام علامات اقتباس مزدوجة:

```
"وهذه سلسلة نصية تشبهها"
```

- أو بعلامات اقتباس ثلاثية مزدوجة:

```
""" أما هذه فسلسلة نصية أطول
    ونستطيع إطالتها وتقسيمها على ما نشاء من الأسطر
    وستحافظ بايثون على هذه الأسطر """
```

يُستخدم النوع الأخير في إنشاء توثيقات لدوال بايثون التي ننشئها بأنفسنا كما سنرى في فصل تال، ويمكن استخدام علامات اقتباس ثلاثية مفردة، لكننا لا ننصح بهذا، إذ قد يصعب تحديد ما إذا كانت العلامات ثلاثيةً مفردةً أم علامةً مزدوجةً داخل أخرى مفردة.

نستطيع الوصول إلى المحارف المنفردة في السلسلة النصية بمعاملتها مثل مصفوفة من المحارف -انظر المصفوفات أدناه-، كما توجد بعض العمليات التي توفرها لغة البرمجة لتساعدنا في التعامل مع السلاسل، مثل العثور على سلسلة فرعية أو ربط سلسلتين معًا، ونسخ واحدة إلى أخرى، وهكذا.

تجدر الإشارة إلى أن بعض اللغات فيها نوع منفصل للمحارف، أي لحرف واحد فقط، وفي تلك الحالة تكون السلاسل مجرد مجموعات من قيم تلك الأحرف، لكن بايثون -على نقيض هذا- إذ تستخدم سلسلةً طولها وحدة واحدة لتخزين الحرف الواحد، ولا يلزم وجود صياغة خاصة.

5.4.1 العوامل النصية String Operators

توجد عدة عمليات يمكن إجراؤها على السلاسل النصية، وبعضها مضمّن مسبقًا في بايثون، لكن يمكن توفير عمليات إضافية باستخدام الوحدات المستورَدة كما فعلنا مع وحدة `sys` في فصل التسلسلات البسيطة.

العامل	الوصف
S1 + S2	ضم كل من S1 و S2
S1 * N	تكرار قدره N من S1

يمكننا أن نرى ذلك في الأمثلة التالية:

```
>>> print( 'Again and ' + 'again' )     # ضم السلسلة
Again and again
>>> print( 'Repeat ' * 3 )                    # تكرار السلسلة
Repeat Repeat Repeat
>>> print( 'Again ' + ('and again ' * 3) )    # جمع '+' و '*'
Again and again and again and again
```

كما يمكن إسناد سلاسل أحرف إلى متغيرات:

```
>>> s1 = 'Again '
>>> s2 = 'and again '
>>> print( s1 + (s2 * 3) )
Again and again and again and again
```

لاحظ أن المثالين الأخيرين أنتجا نفس الخرج.

هناك الكثير من الأمور التي يمكن تنفيذها باستخدام السلاسل النصية، وسننظر فيها بتفصيل في فصول تالية بعد أن نكتسب خبرةً جيدةً في المواضيع التي تسبقها.

ربما تجب ملاحظة أن السلاسل النصية في بايثون لا يمكن تعديلها، أي أننا نستطيع إنشاء سلسلة نصية جديدة مع تغيير بعض الأحرف، لكن لا يمكن تعديل أي حرف داخل سلسلة تعديلًا مباشرًا، ويُعرف نوع البيانات ذاك الذي لا يمكن تغييره باسم النوع غير القابل للتغير immutable.

5.4.2 متغيرات السلاسل النصية في VBScript

تسمى المتغيرات في VBScript باسم المتغايرات variants، لأنها قد تحتوي على أي نوع من البيانات، وتحاول VBScript أن تحولها إلى النوع المناسب وفق الحاجة، وعليه فقد نسند عددًا إلى متغير ما، لكن إذا استخدمناه مثل سلسلة نصية فستحاول VBScript أن تحوله نيابةً عنا، وهذا يشبه -من الناحية العملية- سلوك أمر `print` في بايثون، غير أنه هنا يمتد إلى أي أمر VBSCript. ويمكن إعطاء VBScript لإشارة على أننا نريد قيمةً عدديةً في صورة سلسلة نصية من خلال تغليفها بعلامتي اقتباس مزدوجتين:

```
<script type="text/vbscript">
MyString = "42"
```

```
MsgBox MyString
</script>
```

يمكن دمج سلاسل VBScript معًا في عملية تسمى بالضم concatenation، باستخدام العامل &:

```
<script type="text/vbscript">
MyString = "Hello" & "World"
MsgBox MyString
</script>
```

5.4.3 سلاسل جافاسكربت النصية

تحاط سلاسل جافاسكربت النصية بعلامات اقتباس مفردة أو مزدوجة، ويجب أن نصرّح عن المتغيرات قبل استخدامها، وذلك بكلمة `var` المفتاحية. في المثال التالي سنصرح عن متغيري سلاسل نصية ونعرّفهما:

```
<script type="text/javascript">
var aString, another;
aString = "Hello ";
another = "World";
document.write(aString + another);
</script>
```

كما تسمح لنا جافاسكربت بإنشاء كائنات سلاسل نصية، ويمكن القول بأنها سلاسل نصية مع بعض المزايا الإضافية، والاختلاف الرئيسي بينهما هو طريقة إنشائها، والتي هي بالشكل:

```
<script type="text/javascript">
var aStringObj, anotherObj;
aStringObj = String("Hello ");
anotherObj = String("World");
document.write(aStringObj + anotherObj);
</script>
```

قد تكون تلك كتابةً كثيرةً قياسًا على المثال السابق مع أنهما ينجزان العمل نفسه -وهذا صحيح نوعًا ما-، غير أن كائنات السلاسل تعطينا مزايا في بعض المواقف تجعلنا نفضل استخدامها عما سواها، كما سنرى لاحقًا.

5.5 الأعداد الصحيحة Integers

الأعداد الصحيحة هي الأعداد الكاملة غير الكسرية، من أكبر قيمة سالبة إلى أكبر قيمة موجبة، وهذا التحديد لمجال القيم مهم في البرمجة على عكس الرياضيات، إذ لا نعير اهتمامًا لقيمة العدد غالبًا طالما ينتمي إلى

مجموعة أساسية أو المجموعة التي نعمل عليها، أو لم يكن يؤثر في الناتج؛ أما في البرمجة فهناك حدود عظمى وحدود دنيا، ويعرَف الحد الأعلى باسم MAXINT، كما يعتمد على عدد البتات bits المستخدَمة في حاسوبك لتمثيل عدد ما. يكون في أغلب الحواسيب ولغات البرمجة الحالية العدد 64-بت في المعماريات الحديثة، لذا فإن قيمة الحد الأعلى هنا MAXINT تساوي 10 مرفوعة إلى الأس 19، وتستطيع بايثون إخبارنا بقيمة maxint باستخدام وحدة sys، فإذا كانت معمارية النظام 64-بت مثلًا فستكون قيمة sys.maxsize هي 9223372036854775807، وهو رقم كبير؛ أما في VBScript فتكون تلك القيمة محدودةً بـ (+/- 32000) لأسباب تاريخية.

تُعرَف الأعداد التي تحمل إشارةً موجبةً أو سالبةً بالأعداد الصحيحة ذات الإشارة signed integers، ومن الممكن استخدام أعداد صحيحة لا إشارة لها، غير أن هذا مقصور على الأعداد الموجبة، بما في ذلك الصفر، وهذا يعني أن الأعداد الصحيحة التي لا إشارة لها ضعف عدد ذوات الإشارة، بما أننا سنستغل المساحة التي كانت مخصصةً للأعداد السالبة لضمها إلى الأعداد التي لا إشارة لها، مما يعني زيادة قيمة الحد الأقصى إلى الضعف أيضًا "2 * MAXINT"، ونظرًا لأن حجم الأعداد الصحيحة مقيد بقيمة MAXINT، فإذا جمعنا عددين صحيحين وكان ناتج الجمع أكبر من الحد الأقصى؛ فسنحصل على خطأ، لكن قد تعيد بعض الأنظمة واللغات تلك القيمة الخاطئة كما هي مع رفع راية سرية secret flag نستطيع التحقق من وجودها إذا كنا نشك في الإجابة، وفي مثل هذه الحالة يكون التحقق عادةً هو برفع حالة خطأ، فإما أن يكون البرنامج حينها قادرًا على معالجة ذلك الخطأ فيعالجه، أو ينهي البرنامج نفسه إذا لم يستطع.

تحوّل كل من جافاسكربت وVBScript العدد إلى صيغة تستطيعان التعامل معها، ولا تهتمان بالدقة هنا، إذ قد تختلف الصيغة الجديدة اختلافًا طفيفًا؛ أما لغة بايثون فتستخدم شيئًا اسمه العدد الصحيح الطويل Long Integer، وهي ميزة خاصة لها تسمح بأعداد صحيحة غير محدودة الحجم.

```
>>> import sys
>>> x = sys.maxsize * 1000000
>>> print(x)
9223372036854775807000000
>>> print (type(x))
<class 'int'>
```

كما نرى فالنتيجة أكبر بكثير من القيمة التي تتوقعها من الحاسوب في العادة، رغم احتسابها int، أما إذا استخدمنا شيفرةً مكافئةً لهذا المثال في جافاسكربت وVBScript؛ فسيخرج لنا العدد بصيغة تختلف عن العدد الصحيح الذي نتوقعه، وسنتعلم المزيد عن هذا في قسم الأعداد الحقيقية.

```
<script type="text/vbscript">
Dim x
```

```
x = 123456700 * 34567338738999
MsgBox CStr(x)
</script>
```

5.5.1 العمليات الحسابية

رأينا معظم العمليات الحسابية التي نحتاج إليها في فصل التسلسلات البسيطة، ولكن للتلخيص نعيدها مرةً أخرى:

مثال العامل	الوصف
M + N	جمع M و N
M - N	طرح N من M
M * N	ضرب M في N
M / N	قسمة M على N، سيكون الناتج عددًا حقيقيًا كما سنبين لاحقًا
M // N	القسمة الصحيحة لـ M على N، ستكون النتيجة عددًا صحيحًا
M % N	باقي القسمة: أوجد ما تبقى من قسمة M على N
M**N	الأس: M مرفوعًا إلى الأس N

بما أننا لم نرى المتغير الأخير من قبل، فلننظر في مثال على إنشاء بعض متغيرات الأعداد الصحيحة ونستخدم عامل الأس:

```
>>> i1 = 2     # أنشئ عددًا صحيحًا وأسنده إلى i1
>>> i2 = 4
>>> i3 = i1**i2  # أسند نتيجة 2 إلى أس 4 إلى i3
>>> print( i3 )
16
>>> print( 2**4 )  # أكد النتيجة
16
```

تحتوي الوحدة `math` في بايثون على بعض الدوال الرياضية الشائعة مثل `sin` و `cos` وغيرهما.

5.5.2 عوامل الاختصار

إحدى العمليات الشائعة في البرمجة هي زيادة قيمة المتغير التدريجية، فإذا كان لدينا المتغير `x` الذي قيمته `42`، ونريد زيادة تلك القيمة إلى `43`، فسيكون ذلك كما يلي:

```
>>> x = 42
>>> print( x )
>>> x = x + 1
>>> print( x )
```

لاحظ السطر التالي:

```
x = x + 1
```

لا معنىً لهذا السطر من حيث المنطق الرياضي، لكنه في البرمجة يعني أن x الجديدة تساوي قيمة x السابقة مضافًا إليها 1، فإذا كنت معتادًا على المنطق الرياضي؛ فستستغرق بعض الوقت لتعتاد على هذا النمط الجديد، لكن الفكرة هنا أن علامة = تُقرأ "**ستكون**" وليست "**تساوي**"، وعليه فإن السطر يعني أن x **ستصبح** قيمتها الجديدة هي x+1، وهذا النوع من العمليات شائع جدًا في الممارسة العملية إلى حد أن بايثون وجافاسكربت توفران عاملًا مختصرًا له من أجل توفير الكتابة على المبرمج:

```
>>> x += 1
>>> print( x )
```

وهذان العاملان مساويان تمامًا لتعليمة الإسناد السابقة لكنهما أقصر، توجد عوامل اختصار لباقي العمليات الحسابية بنفس المنطق السابق، نبينها في الجدول التالي:

مثال العامل	الوصف
M += N	M = M + N
M -= N	M = M - N
M *= N	M = M * N
M /= N	M = M / N
M //= N	M = M // N
M %= N	M = M % N

5.5.3 الأعداد الصحيحة في VBScript

لقد قلنا إن الأعداد الصحيحة في VBScript محدودة بقيمة عظمى أقل من بايثون وجافاسكربت، وهي تقابل 16-بت على خلاف 64-بت التي رأيناها من قبل، أي حوالي "+/- 32000"، فإذا احتجنا إلى عدد صحيح أكبر من هذا؛ فنستخدم العدد `long` الذي يساوي حجمه حجم العدد الصحيح من معمارية 32-بت (أي 2 مليار)، كذلك لدينا النوع `byte` الذي هو عدد ثماني البتات، وحجمه الأكبر يساوي 255، وسيتضح بالممارسة العملية أن النوع القياسي للعدد الصحيح كافٍ لأغلب العمليات، فإذا كانت نتيجة العملية أكبر من MAXINT؛ فستحول VBSCript النتيجة تلقائيًا إلى عدد ذي فاصلة عائمة أو حقيقي، كما سنبين لاحقًا.

ورغم دعم VBScript لجميع العمليات الحسابية المعتادة، إلا أن عملية الباقي modulo تمثَّل تمثيلًا مختلفًا، باستخدام العامل MOD، وقد رأينا ذلك في فصل التسلسلات البسيطة، كذلك فإن تمثيل الأسس مختلف، إذ يُستخدم محرف الإقحام ^ بدلًا من ** في بايثون.

5.5.4 أعداد جافاسكربت

تحتوي جافاسكربت على نوع عددي في صورة كائن كما سنصفه فيما بعد، وهي تسميه Number، لكن أحيانًا قد يكون عدد جافاسكربت "ليس عددًا" أو Not a Number، ويشار إليه بـ NaN اختصارًا، وهو ما نحصل عليه أحيانًا كنتيجة لبعض العمليات المستحيلة رياضيًا، والهدف منه هو السماح لنا بالتحقق من أنواع أخطاء بعينها دون تعطيل البرنامج، كما تحوي لغة بايثون كذلك النوع NaN أيضًا.

تحتوي جافاسكربت على نسخ عددية خاصة لتمثيل اللانهاية الموجبة والسالبة، وهي ميزة نادرة نسبيًا في لغات البرمجة، وقد تكون كائنات جافاسكربت العددية إما أعدادًا صحيحةً أو حقيقيةً كما سنرى لاحقًا، وتَستخدم جافاسكربت نفس العوامل التي تستخدمها بايثون، باستثناء الإجراءات الأسية التي تستخدم كائنًا خاصًا بها اسمه Math.

5.6 الأعداد الحقيقية Real Numbers

الأعداد الحقيقية هي نفسها الأعداد الصحيحة لكن مع إضافة الكسور، وقد تمثل أعدادًا أكبر من الحد الأقصى MAXINT لكن مع دقة أقل، وهذا يعني أننا قد نرى عددين حقيقيين على أنهما متطابقين، لكن الحاسوب يراهما غير متطابقين عندما يوازنهما معًا، لأنه ينظر في أدق التفاصيل بينهما، حيث يمكن تمثيل العدد 5.0 في الحاسوب بإحدى الصورتين 4.9999999 أو (5.000000...01) مثلًا، وهذه التقاربات مناسبة لأغلب الاستخدامات، لكن قد نحتاج أحيانًا إلى تلك الدقة، فتذكر هذا الأمر إذا حصلت على نتيجة غريبة لا تتوقعها عند استخدام الأعداد الحقيقية.

كما تُعرَف الأعداد الحقيقية أيضًا باسم أعداد الفاصلة العائمة floating point numbers، ولها نفس العمليات الحسابية التي تطبّق على الأعداد الصحيحة، مع زيادة هنا وهي قدرتها على اقتطاع العدد ليكون قيمة صحيحة.

تدعم لغات البرمجة الثلاث بايثون وجافاسكربت وVBScript الأعداد الحقيقية، وننشئها في بايثون بتخصيص عدد بعلامة عشرية فيه كما رأينا في فصل التسلسلات البسيطة؛ أما جافاسكربت وVBScript فلا يوجد تمييز واضح بين الأعداد الصحيحة والحقيقية، فما عليك إلا استخدامها وستتعامل اللغة مع ما تدخله إليها وفق المناسب.

5.7 الأعداد المركبة أو التخيلية Complex numbers

إذا كانت لديك خلفية رياضية فقد تتساءل أين الأعداد المركبة complex في هذا الشرح، والواقع أننا لا نحتاج إليها غالبًا في البرمجة، لكن على أي حال فبعض اللغات مثل بايثون توفر دعمًا مضمّنا فيها للأعداد المركبة، بينما توفر بعض اللغات الأخرى مكتبةً من الدوال التي تستطيع التعامل مع الأعداد المركبة، ونفس الأمر ينطبق على المصفوفات.

يمثَّل العدد المركب في بايثون كما يلي:

```
(real+imaginaryj)
```

وبناءً عليه تبدو عملية جمع بسيطة لعدد مركب كما يلي:

```
>>> M = (2+4j)
>>> N = (7+6j)
>>> print( M + N )
(9+10j)
```

تنطبق جميع عمليات الأعداد الصحيحة على الأعداد المركبة كذلك، كما توجد وحدة `cmath` التي تحتوي بعض الدوال الرياضية الشائعة التي تعمل على قيم الأعداد المركبة، ولا توفِّر جافاسكربت وVBScript دعمًا للأعداد المركبة.

5.8 القيم البوليانية True و False

سُمي هذا النوع من القيم بهذا الاسم نسبةً إلى عالم الرياضيات George Boole الذي كان يدرس المنطق في القرن التاسع عشر، ولا يحوي هذا النوع من البيانات إلا قيمتين فقط، هما `True` عند التحقق أو الصحة، و`false` عند عدم التحقق أو الخطأ، وتدعم بعض اللغات هذا النوع دعمًا مباشرًا، بينما تستخدم بعض اللغات الأخرى اصطلاحًا تمثل فيه قيمةً عدديةً القيمة false، وتكون تلك القيمة غالبًا هي الصفر؛ بينما تمثَّل القيمة true بالعدد (1) أو (-1)، وقد كان هذا سلوك بايثون حتى الإصدار 2.2، لكنها صارت تدعم القيم البوليانية مباشرةً بعد ذلك باستخدام القيمتين `True` و `False`، وهما كما ترى تبدآن بحرف كبير، فتذكر أن بايثون حساسة لحالة الأحرف.

تعرَف القيم البوليانية أحيانًا باسم قيم الحقيقة لأنها تُستخدم للتحقق مما إذا كان شيء ما صحيحًا أم لا، فإذا كتبنا برنامجًا يأخذ نسخةً احتياطيةً من جميع الملفات في مجلد ما مثلًا، فنستطيع نسخ كل ملف على حدة ثم نطلب اسم الملف التالي من النظام، وإذا لم تتبق ملفات لنسخها، فسيعيد النظام سلسلةً نصيةً فارغةً، حيث سنتحقق هنا مما إذا كان الاسم هو سلسلةً فارغةً أم لا، ونخزن النتيجة مثل قيمة بوليانية تكون قيمتها

`True` إذا كان الاسم سلسلةً فارغةً، و `False` إذا لم يكن كذلك، وسنرى كيفية استخدام تلك النتيجة لاحقًا في الكتاب.

5.8.1 العوامل البوليانية أو المنطقية

العامل	الوصف	الأثر
A and B	الإضافة أو AND	صحيح إذا كان كل من A و B صحيحًا أو متحققًا، وخطأ في الحالات الأخرى
A or B	الاختيار أو OR	صحيح إذا كان أحدهما أو كلاهما صحيحًا، وخطأ إذا كان كل من A و B خاطئًا
A == B	التساوي	صحيح إذا كان A يساوي B
A != B	عدم التساوي	صحيح إذا كانت A لا تساوي B
not B	النفي	صحيح إذا لم يكن B صحيحًا

> لاحظ أن العامل الأخير يعمل على قيمة واحدة، بينما توازن بقية العوامل بين قيمتين.

تدعم VBScript القيم البوليانية كما تفعل بايثون على الصورة `True` و `False`، أما جافاسكربت فتدعمها لكن على الصورة `true` و `false` -لاحظ حالة الأحرف الصغيرة هنا-. كما ستجد أسماءً مختلفةً لنوع البيانات البولياني أو المنطقي في كل لغة، ففي بايثون ستجدها باسم `bool`، أما في جافاسكربت وVBScript فاسمه `Boolean`، ولا تشغل بالك بهذا لأننا لن ننشئ متغيرات من هذا النوع، لكننا سنستخدم نتائجه في الاختبارات.

5.9 التجميعات Collections

لقد شكلت علوم الحاسوب نظامًا كاملًا يدرس التجميعات وسلوكياتها المختلفة، والتي تسمى بالحاويات أحيانًا أو التسلسلات، وسننظر أولًا في التجميعات المدعومة في بايثون وجافاسكربت وVBScript، ثم نختم بملخص موجز لبعض أنواع التجميعات الأخرى التي قد نراها في لغات أخرى.

5.9.1 القوائم Lists

لا شك في أن القوائم غنية عن تعريفها، إذ نستخدمها في حياتنا كل يوم، فهي مجرد سلسلة من العناصر، حيث نستطيع إضافة عناصر إلى تلك السلسلة أو حذفها، وعادةً ما نضيف العناصر في نهاية القائمة إذا كانت مكتوبةً على الورق، فلا نستطيع إدخال عنصر في المنتصف مثلًا، على عكس القوائم الإلكترونية في برنامج معالج النصوص مثلًا، إذ نستطيع وضع العنصر الجديد في أي مكان، كما نستطيع البحث في القائمة لنتحقق من وجود شيء بعينه فيها، لكن يجب التنقل في القائمة من أعلاها لأسفلها عنصرًا عنصرًا متحققين من كل

عنصر لنرى إذا كان ما نريده موجودًا أم لا، وتُعَد القوائم أحد أنواع التجميعات التي لا غـنى عنهـا في أغلب لغـات البرمجة الحديثة.

لا تحتوي جافاسكربت ولا VBScript على قوائم مضمّنة فيها، لكن نستطيع محاكـاة مزايـا القـوائم بواسـطة المصفوفات في جافاسكربت، وبتجميعات بيانـات أخـرى في VBScript كمـا سنشـرح لاحقًـا؛ أمـا لغـة بـايثون فالقوائم مضمّنة تلقائيًا فيها، وتستطيع تنفيذ العمليات الأساسية التي تحـدثنا عنهـا جميعًـا، إضـافةً إلى القـدرة على فهرسة العناصر فيها، والفهرسة هي الإشارة إلى عنصـر في القائمـة بـرقم تسلسـله، على فـرض أن العنصـر الأول يبدأ من الصفر، كما توفر بايثون عدة عمليات أخرى على التجميعات، وتنطبق كلها تقريبًا على القوائم، كمـا تنطبق مجموعة فرعية منها على أنواع أخرى من التجميعات بما في ذلك السلاسل النصية الـتي هي مجـرد نـوع خاص من القوائم، إذ هي قوائم من المحارف.

تُستخدم الأقواس المربعة لإنشاء القوائم والوصول إليها في بـايثون، ويمكن إنشـاء قائمـة فارغـة باسـتخدام زوج من تلك الأقواس ليس بينهما شيء، أو إنشاء قائمة تحتوي على قيم تفصل بينها فاصلة إنجليزية ,:

```
>>> aList = []
>>> another = [1,2,3]
>>> print( another )
[1, 2, 3]
```

نستطيع الوصول إلى العناصـر المنفـردة باسـتخدام رقم الفهـرس، حيث يحمـل العنصـر الأول رقم 0 داخـل الأقواس المربعة، فإذا أردنا الوصول إلى العنصر الثالث مثلًا -والذي سيحمل الترتيب 2 داخل الأقـواس بمـا أننـا نبدأ من الصفر-، فإننا نصل إليه كما يلي:

```
>>> print( another[2] )
3
```

كما نستطيع تغيير قيم العناصر في القائمة بطريقة مماثلة:

```
>>> another[2] = 7
>>> print( another )
[1, 2, 7]
```

لاحظ كيف تغير العنصر الثالث -الذي فهرسه 2- من 3 إلى 7.

يمكن استخدام أرقام الفهرس السالبة للوصول إلى عناصر القائمة من نهايتها، فإذا أردنـا العنصـر الأخـير من القائمة نستخدم 1-:

```
>>> print( another[-1] )
```

```
7
```

وتضاف العناصر الجديدة ملحقةً إلى نهاية القائمة باستخدام العملية append():

```
>>> aList.append(42)
>>> print( aList )
[42]
```

كما يمكن الاحتفاظ بقائمة داخل قائمة أخرى، فإذا ألحقنا القائمة الثانية بالقائمة الأولى فستكون على النحو التالي:

```
>>> aList.append(another)
>>> print( aList )
[42, [1, 2, 7]]
```

لاحظ كيف تكون النتيجة قائمةً من عنصرين، لكن العنصر الثاني عبارة عن قائمة بحد ذاته كما نرى من القوسين المحيطين به، ونستطيع الآن الوصول إلى العنصر 7 باستخدام فهرس مزدوج:

```
>>> print( aList[1][2] )
7
```

حيث يستخرج الفهرس 1 -الفهرس الأول- العنصر الثاني من القائمة الأولى -الذي هو نفسه القائمة الثانية-، ثم يستخرج الفهرس 2 -الفهرس الثاني- العنصر الثالث في القائمة الثانية، وهو العنصر 7.

هذا التداخل وإن بدا معقدًا إلا أنه مفيد للغاية، حيث يسمح ببناء جداول من البيانات بفعالية، كما يلي:

```
>>> row1 = [1,2,3]
>>> row2 = ['a','b','c']
>>> table = [row1, row2]
>>> print( table )
[ [1,2,3], ['a','b','c'] ]
>>> element2 = table[0][1]
>>> print( element2 )
2
```

يمكن استخدام هذه الطريقة في إنشاء دفتر عناوين يكون كل مدخل فيه عبارةً عن قائمة من الاسم والعنوان، ويوضح المثال التالي دفتر عناوين فيه مدخلان:

```
>>> addressBook = [
... ['Samy', '9 Some St',' Anytown', '0123456789'],
```

```
... ['Warda', '11 Nother St', 'SomePlace', '0987654321']
... ]
>>>
```

لاحظ كيف أنه رغم إدخالنا لأربعة أسطر من النص، إلا أن بايثون تعاملت معها على أنها سطر واحد، كما نستنتج من محثات `...`، وهذا لأن بايثون ترى أن عدد الأقواس الافتتاحية لا يطابق عدد أقواس الإغلاق، ولهذا تستمر في قراءة المدخلات إلى أن يحدث ذلك التطابق. قد تكون هذه طريقةً فعالةً للغاية في بناء هياكل بيانات معقدة بسرعة مع جعل الهيكل العام -أي قائمة القوائم- واضحًا للقارئ.

> إذا كنت تستخدم أداة IDLE فلن ترى محث `...`، وإنما مجرد سطر فارغ.

تدريب

استخرج رقم هاتف سامي -العنصر 3 من الصف الأول-، وتذكر أن الفهارس تبدأ من الصفر وليس الواحد، ثم أضف بعض المدخلات من عندك باستخدام عملية `append()`.

لاحظ أننا نفقد بياناتنا عند الخروج من بايثون، لكن سنرى كيفية الاحتفاظ بها عند شرح الملفات في هذا الكتاب.

يُستخدم الأمر `del` لحذف العناصر من القائمة:

```
>>> del aList[1]
>>> print( aList )
[42]
```

لاحظ أن `del` لا يحتاج إلى أقواس حول القيمة، على عكس دالة الطباعة، لأنه أمر وليس دالةً، قد يكون هذا الفرق طفيفًا، لكن إذا وضعت أقواسًا حول القيمة فلا بأس إذ ستظل صالحةً.

إذا أردنا ضم قائمتين معًا لجعلهما قائمةً واحدةً، فسنستخدم عامل الضم `+` الذي رأيناه في ضم السلاسل النصية من قبل:

```
>>> newList = aList + another
>>> print( newList )
[42, 1, 2, 7]
```

لاحظ اختلاف هذا المثال عن سابقه الذي ألحقنا فيها القائمتين معًا، فقد كان لدينا عنصران، وكان العنصر الثاني منهما عبارةً عن قائمة؛ أما هنا فلدينا أربعة عناصر لأن عناصر القائمة الثانية أضيفت عنصرًا عنصرًا إلى `newList`، فإذا أردنا الوصول إلى العنصر 1 هذه المرة فسنحصل على 1، وهو العنصر الثاني هنا، بدلًا من الحصول على قائمة فرعية.

```
>>> print( newList[1] )
1
```

يمكن تطبيق إشارة عملية الضرب هنا مثل عامل تكرار لملء القائمة بمضاعفات نفس القيمة:

```
>>> zeroList = [0] * 5
>>> print( zeroList )
[0, 0, 0, 0, 0]
```

كما يمكن إيجاد فهرس عنصر ما في قائمة باستخدام العملية `index()` كما يلي:

```
>>> print( [1,3,5,7].index(5) )
2
>>> print( [1,3,5,7].index(9) )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

لاحظ أن محاولة العثور على فهرس لشيء غير موجود في القائمة ينتج خطأً، فإذا أردنا التحقق من وجود شيء في تجميعة ما فسنستخدم العامل `in` كما يلي:

```
>>> print( 5 in [1,3,5,7] )
True
>>> print( 9 in [1,3,5,7] )
False
```

النتائج قيم بوليانية كما نرى، إما true أو false، وسنرى كيف نستخدم تلك النتائج في فصل لاحق.

أخيرًا، يمكن معرفة طول القائمة باستخدام الدالة المضمَّنة `len()`:

```
>>> print( len(aList) )
1
>>> print( len(newList) )
4
>>> print( len(zeroList) )
5
```

لا تدعم جافاسكربت ولا VBScript نوع القائمة مباشرةً، وسنرى أن لديهما نوع بيانات يحاكي وظيفة القائمة هنا، وهو نوع المصفوفات arrays.

5.9.2 الصف Tuple

توفر بعض لغات البرمجة بنية بيانات اسمها صف tuple، وما هي إلا تجميعة عشوائية من القيم لكنها تعامَل مثل وحدة واحدة، وهي تشبه القوائم في نواحٍ كثيرة، لكن مع فرق أن وحدات tuple غير قابلة للتغيير، مما يعني أننا لا نستطيع التعديل عليها أو إلحاق شيء إليها بعد إنشائها أول مرة، تمثَّل وحدات tuple في بايثون بقائمة من القيم المفصول بينها بفواصل:

```
>>> aTuple = 1,3,5
>>> print( aTuple[1] ) # استخدم الفهرسة مثل القوائم
3
>> aTuple[2] = 7   # خطأ، لا يمكن تعديل tuple
Traceback (innermost last):
  File "<pyshell#20>", line 1, in ?
        aTuple[2] = 7
TypeError: object doesn't support item assignment
```

من المعتاد إحاطة التسلسل بأقواس، وهذا يضمن عدم إخراج القيم من السياق، إضافةً إلى التذكير المرئي بأنها وحدة واحدة، أي وحدة صف، توضح البيانات التالية هذا الأمر:

```
>>> t1 = 1,2,3 * 3
>>> t2 = (1,2,3) * 3
>>> print(t1)
(1, 2, 9)
>>> print( t2 )
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

يطبَّق الضرب في الحالة الأولى على العنصر الأخير فقط في وحدة Tuple، أما في الحالة الثانية فتضاعَف المجموعة بالكامل، ولتجنب مثل هذا الغموض، سنستخدم الأقواس في كل مرة نصف فيها وحدات tuple.

لعل أحد أهم الأمور التي يجب تذكرها هو أن الأقواس المربعة تستخدم لفهرسة وحدات tuple، بينما تُستخدم الأقواس العادية لتعريفها، ولا يمكن تغييرها بعد إنشائها؛ أما في سوى هذه الأمور فما ينطبق من العمليات على القوائم ينطبق أيضًا على الصفوف، ورغم أننا لا نستطيع تعديل وحدة الصف بعد إنشائها؛ إلا أننا نستطيع إضافة أعضاء جديدة إليها باستخدام عامل الإضافة، فهذا سينشئ صفًا جديدًا، كما يلي:

```
>>> tup1 = (1,2,3)
>>> tup2 = tup1 + (4,)
# الفاصلة تجعل هذا وحدة tuple وليس عددًا صحيحًا
>>> print( tup2 )
(1,2,3,4)
```

إذا لم نستخدم الفاصلة اللاحقة بعد العدد 4 فستفسره بايثون على أنه العدد الصحيح 4 داخل أقواس وليس على أنه صف، لكن سنحصل على خطأ بأي حال بما أنه لا يمكن إضافة أعداد صحيحة إلى الصفوف، لذا سنضيف الفاصلة لنخبر بايثون بأن تعامل الأقواس على أنها صف، فبإضافة الفاصلة الزائدة نقنع بايثون بأن وحدة الصف التي فيها مدخل واحد هي صف فعلًا؛ أما بالنسبة للغتي جافاسكربت وVBScript فليس لديهما أي مفهوم للصف tuple.

5.9.3 القاموس Dictionary أو Hash

يحتوي هذا النوع من البيانات على قيمة ترتبط بمفتاح، كما يربط القاموس اللغوي المعنى بكلمة، وقد تكون تلك القيمة سلسلةً نصيةً أو لا، وتُجلب تلك القيمة بفهرسة القاموس بالمفتاح، ولا يلزم أن يكون المفتاح سلسلةً من الأحرف -رغم أنه غالبًا كذلك-، وهذا على خلاف القاموس اللغوي، فقد يكون أي نوع غير قابل للتغير، بما في ذلك الأعداد والصفوف، كما قد تحتوي القيم المرتبطة بالمفاتيح على أي نوع من البيانات.

تُستخدم القواميس داخليًا بواسطة تقنية برمجة متقدمة تسمى بجداول Hash، لهذا يشار أحيانًا إلى القاموس باسم Hash، ونظرًا لأننا نتوصَّل إلى قيم القاموس عبر المفتاح، فلا توضع إلا عناصر ذات مفاتيح فريدة، رغم أن المفتاح قد يشير إلى قائمة من القيم.

تُعَد القواميس مفيدةً للغاية في هيئة هياكل للبيانات، ونجدها في بايثون مثل نوع بيانات مضمَّن في اللغة نفسها، لكن قد نحتاج في بعض اللغات الأخرى إلى استخدام وحدة module، أو بناء واحدة بأنفسنا إذا أردنا استخدام القواميس، وتُستخدم القواميس بعدة طرق سنراها كثيرًا في الأمثلة التالية من الكتاب؛ أما الآن فسنرى أولًا كيف ننشئ قاموسًا في بايثون، ثم نملؤه ببعض المدخلات ونقرؤها.

```
>>> dct = {}
>>> dct['boolean'] = "A value which is either true or false"
>>> dct['integer'] = "A whole number"
>>> print( dct['boolean'] )
A value which is either true or false
```

لاحظ أننا نبدأ القاموس بأقواس معقوصة ثم نستخدم الأقواس المربعة لإسناد القيم وقراءتها، كما تُستخدم دالة النوع `dict()` لإعادة قاموس فارغ، ويمكن بدء القاموس أثناء إنشائه كما فعلنا في حالة القوائم، باستخدام الصيغة التالية:

```
>>> addressBook = {
... 'Samy' : ['Samy', '9 Some St',' Anytown', '0123456789'],
... 'Warda' : ['Warda', '11 Nother St', 'SomePlace', '0987654321']
... }
>>>
```

يُفصل المفتاح والقيمة بنقطتين رأسيتين، وتُفصل الأزواج بفواصل إنجليزية , ، كما يمكن تخصيص القاموس باستخدام صيغة مختلفة -انظر أدناه-، ويمكنك اختيار إحدى الطريقتين لاستخدامها.

```
>>> book = dict(Samy=['Samy', '9 Some St',' Anytown', '0123456789'],
...             Warda=['Warda', '11 Nother St', 'SomePlace',
'0987654321'])
>>> print( book['Samy'][3] )
0123456789
```

لاحظ أننا لا نحتاج إلى علامات اقتباس حول المفتاح في التعريف، لأن بايثون تفترض أنها سلسلة نصية، لكننا نحتاج إليها بأي حال من أجل استخراج القيم، وهذا الأسلوب يَحُد من عملية الطريقة الثانية، لهذا يفضِّل المبرمجون استخدام الطريقة الأولى التي يستخدمون الأقواس المعقوصة فيها، وبهذا نكون قد أنشأنا دفتر عناوين من قاموس مفاتيحه هي الأسماء، ويخزن قوائمنا مثل قيم، ونريد أن نستخدم الاسم وحده لجلب كل المعلومات بدلًا من حساب الفهرس العددي، كما يلي:

```
>>> print( addressBook['Warda'] )
['Warda', '11 Nother St', 'SomePlace', '0987654321']
>>> print( addressBook['Samy'][3] )
0123456789
```

في الحالة الثانية فهرسنا القيم المعادة من أجل الحصول على رقم هاتف فقط، لكن يمكن جعل هذا أسهل عبر إنشاء بعض المتغيرات وإسناد قيم الفهرسة المناسبة، كما يلي:

```
>>> name = 0
>>> street = 1
>>> town = 2
>>> tel = 3
```

نستطيع الآن استخدام تلك المتغيرات لنعرف اسم مدينة Warda:

```
>>> print( addressBook )
SomePlace
```

لاحظ أن town ليست داخل علامات اقتباس لأنها اسم متغير، وستحوله بايثون إلى قيمة الفهرس التي أسندناها -وهي 2-، على عكس 'Warda' المحاطة بعلامات اقتباس لأن المفتاح هو سلسلة نصية، وهنا بدأ دفتر العناوين يتحول إلى ما يشبه التطبيق القابل للاستخدام بسبب مزايا القواميس، حيث لن نحتاج إلى كثير من العمل الإضافي لحفظ البيانات واستعادتها وإضافة محث استعلام ليسمح لنا بتحديد البيانات التي نريدها.

يمكن استخدام قاموس لتخزين البيانات، وسيكون دفتر العناوين حينها قاموسًا مفاتيحه الأسماء، وستكون القيم قواميس مفاتيحها هي حقول الأسماء، كما يلي:

```
addressBook = {
... 'Samy' : {'name':      'Samy',
...           'street':    '9 Some St',
...           'town':      'Anytown',
...           'tel':       '0123456789'},
... 'Warda' : {'name':      'Warda',
...           'street':    '11 Nother St',
...           'town':      'SomePlace',
...           'tel':       '0987654321'}
... }
```

لاحظ أن هذا التنسيق سهل القراءة رغم أنه يحتاج إلى الكثير من الكتابة، ويُشار إلى مثل هذه البيانات التي تخزَّن بتنسيق يجمع بين المعنى والمحتوى في تنسيق مقروء للبشر باسم البيانات الموثقة ذاتيًا.

حين ندخِل هيكل بيانات داخل هيكل آخر مطابق له، كما في حالة إدخال قاموس داخل قاموس آخر هنا، فإنه يُسمى بالتداخل أو التشعب nesting، حيث يكون القاموس الداخلي قاموسًا متشعبًا من الخارجي، ويمكن الوصول إلى تلك البيانات بطريقة تشبه أسلوب القوائم ذات الفهارس المسماة:

```
>>> print( addressBook['Warda']['town'] )
SomePlace
```

وليس ثمة اختلاف إلا في الاقتباس الإضافي حول كلمة town التي هي ليست موجودةً في حالة القوائم، وميزة هذا الأسلوب أننا نستطيع إدخال حقول جديدة دون تعطيل الشيفرة الحالية، بينما نضطر في حالة الفهارس المسماة إلى العودة لتغيير جميع قيم الفهارس، وتبرز أهمية هذا الأسلوب عند استخدام نفس البيانات في عدة برامج، فيكون بذل قليل من الجهد هنا موفرًا لكثير من العمل لاحقًا في المستقبل.

لا تدعم القواميس كثيرًا من عوامل التجميعات التي رأيناها من قبل، فلا تدعم عامل الضم ولا التكرار ولا الإلحاق، رغم إمكانية إسناد أزواج من المفاتيح والقيم مباشرةً كما رأينا في بداية هذا القسم.

وتُستخدم العملية `keys()` لتسهيل الوصول إلى مفاتيح القاموس، إذ تسمح لنا بالحصـول على قائمـة من جميع المفاتيح الموجودة في قاموس ما، فإذا أردنا الحصول على كل الأسماء الموجودة في دفتر العنـاوين الـذي أنشأناه من قبل؛ فسيكون ذلك كما يلي:

```
>>> print( list(addressBook.keys()) )
['Samy','Warda']
```

لاحظ أننا استخدمنا `list()` للحصول على قيم المفاتيح الحقيقية، أما إذا حذفنا `list()` فسنحصـل على نتيجة غريبة بعض الشيء لن نذكرها الآن.

لاحظ أن القواميس لا تخزِّن مفاتيحها بنفس ترتيب إدخالها، فقد نرى أن المفاتيح تظهر بـترتيب غـريب عن ترتيبها الأول، بل قد يتغير الترتيب بتغير الوقت، لكن هذا لا يهم بما أننا نستطيع استخدام المفاتيح للوصول إلى البيانات المرتبطة بها، إذ سنحصل على القيم الصحيحة في كل مرة.

يمكن الحصول على قائمة بجميع القيم أيضًا باستخدام العملية `values()`، فجرب ذلك على دفتر العناوين وانظر إن اسـتطعت تنفيـذها بنجـاح، اسـتخدم مثـال `keys()` السـابق للاسترشـاد بـه، ويمكن بصـورة مماثلـة استخدام العامل `in` على القاموس ليخبرنا هل توجد قيمة ما في صورة مفتاح للقاموس أم لا.

5.9.4 قواميس VBScript

توفر VBScript كائن قاموس يقدم تسهيلات تماثل قاموس بايثون، لكن استخدامه مختلـف قليلًا عن حالـة بايثون، حيث ننشئ القاموس هنا بالتصريح عن متغير يحمل الكائن، ثم ننشئ ذلك الكـائن، وبعـد ذلـك نضـيف المدخلات إلى القاموس الجديد كما يلي:

```
Dim dict     ' Create a variable.
Set dict = CreateObject("Scripting.Dictionary")
dict.Add "a", "Athens" ' Add some keys and items.
dict.Add "b", "Belgrade"
dict.Add "c", "Cairo"
```

لاحـظ أن دالـة `CreateObject` توضـح أننـا ننشـئ كـائن `"Scripting.Dictionary"`، وهـو كـائن `Dictionary` من وحـدة `Scripting` الخاصـة بلغـة VBScript، وسـندرس ذلـك بالتفصـيل حين نـأتي على الكائنـات، لكننـا نأمـل بـذكرها هنـا أن تتـذكر مفهـوم اسـتخدام كـائن من وحـدة مـا كمـا ذكرنـا في فصل التسلسلات البسيطة.

يجب كذلك استخدام كلمة `Set` المفتاحية عند إسناد كائن إلى متغير في VBScript. نستطيع الآن أن نصل إلى البيانات كما يلي:

```
item = dict.Item("c") ' Get the item.
dict.Item("c") = "Casablanca" ' Change the item
```

توجد عمليات أخرى لإزالة عنصر ما، والتحقق من وجود مفتاح مـا، والحصـول على قائمـة بجميـع المفـاتيح وغير ذلك. كما يمكن تخزين أي نوع من الكائنات في القاموس إذ أن الأمر ليس مقصورًا على السلاسـل النصـية فقط، وهذا يعني أننا نستطيع إنشاء قواميس متشعبة بما أن القاموس نفسه ما هو إلا كائن.

فيما يلي نسخة كاملة مبسطة عن مثال دفتر العناوين السابق، ولكن في VBScript:

```
<script type="text/VBScript">
Dim addressBook
Set addressBook = CreateObject("Scripting.Dictionary")
addressBook.Add "Samy", "Samy, 9 Some St, Anytown, 0123456789"
addressBook.Add "Warda", "Warda, 11 Nother St, SomePlace, 0987654321"

MsgBox addressBook.Item("Warda")
</script>
```

ما فعلناه هنا هو تخزين جميع البيانات مثل سلسلة نصية واحـدة بـدلًا من اسـتخدام قائمـة -وهـذا يصـعّب استخراج الحقول المنفردة على عكس القائمة والقاموس-، ثم نسـتطيع الوصـول إلى تفاصـيل `Warda` ونطبعهـا في صندوق رسالة.

تستطيع هنا الانتقال إلى الفصـل التـالي إذا كنت قـد شـعرت بالملـل من طـول هـذا الفصـل وتريـد كتابـة شيفرات وتجربتها، لكن تذكر أن تعود وتنهي هذا الفصل حين تتعرض لأنواع بيانات لم نذكرها حتى الآن.

5.9.5 المصفوفات أو المتجهات Arrays

المصفوفة هي أحد أنواع التجميعات التي يمتد تاريخهـا موازيًـا لتـاريخ الحوسـبة، ومـا هي إلا مجموعـة من العناصر المفهرسة بحيث يسهل جلبها بسرعة، والغالب أننا يجب أن نحدد عدد العناصر الـتي نريـد تخزينهـا في المصفوفة مسبقًا، إذ يُخزَّن نوع واحد أيضًا من البيانات في المصفوفة عادةً، وهاتان الخاصيتان للمصفوفة همـا اللتان تميزانها عن القوائم مثلًا. لاحظ أننا قد قلنا "عادة"، و"الغالب" للدلالة على غالب الحالات، ذلـك أن لغـات البرمجة المختلفة لها أساليبها الخاصة التي تصف المصفوفات، لذا يصعب وضع وصف يغطي جميع الحالات.

تدعم لغة بايثون المصفوفات من خلال وحدة تسمى `array`، لكننا لا نحتاج إليها إلا نادرًا مع استثناء حالات الحساب الرياضي عالي الأداء؛ أمـا غـير ذلـك فنسـتخدم القـوائم، وفي لغـتي جافاسـكربت وVBScript؛ تتكـون المصفوفات مثل نوع بيانات، لذا سننظر في كيفية استخدامها فيهما.

5.9.6 مصفوفات VBScript

تكون المصفوفة في VBScript تجميعة بيانات ذات طول ثابت، حيث يمكن التوصَل إليها بواسطة فهـرس عددي، ويصرَّح عنها ويوصَل إليها كما يلي:

```
Dim AnArray(42)     ' A 43! element array
AnArray(0) = 27     ' index starts at 0
AnArray(1) = 49
AnArray(42) = 66    ' assign to last element
myVariable = AnArray(1)  ' read the value
```

تُستخدم الكلمة المفتاحية Dim لتحديد أبعاد المتغير، وهي الطريقة الـتي نخـبر بهـا VBScript عن المتغـير، فهي تتوقع إذا بدأنا الشيفرة بـ OPTION EXPLICIT؛ أن نحدد أبعاد أي متغير نستخدمه من خلال Dim، وهـذا سلوك محمود يؤدي إلى برامج ثابتة قوية الأداء، كما أننا حددنا آخر فهرس صالح 42، وهـذا يعـني أن المصـفوفة فيها 43 عنصرًا بما أنها تبدأ من الصفر.

تُستخدم الأقواس في VBScript لحساب أبعاد المصـفوفة وفهرسـتها، وهـذا على خلاف الأقـواس المربعـة المستخدمة في بايثون وجافاسكربت، حيث لا يوجد في VBScript نوع بيانات إلا النوع Variant، والذي يخزِّن أي نوع من أنواع القيم، وبناءً عليه فلا تخزِّن المصفوفات في هذه اللغة إلا المتغيرات بمـا أن عناصـر المصـفوفة تكون من نوع واحد، لكن في نفس الوقت، فإن المتغير هنا قد يحمـل أي نـوع من أنـواع القيم، ممـا يعـني مـرةً أخرى أن المصفوفات في VBScript تخزن أي شيء، وهذا أمر مربك عند التفكير فيه.

نستطيع أيضًا الإسناد إلى أي موضع في المصفوفة بما في ذلك الموضع الأخـير، من غـير أن نملأ المواضـع التي قبله، وسيُضبط أي عنصر غير مهيأ إلى القيمة الخاصة Null.

كما يمكن التصريح عن عدة مصفوفات متعـددة الأبعـاد لنمذجـة جـداول بيانـات كمـا في قـوائم بـايثون إذا استخدمنا تدريب دفتر العناوين مثلًا.

```
Dim MyTable(2,3)  ' 3 rows, 4 columns
MyTable(0,0) = "Samy"  ' Populate Samy's entry
MyTable(0,1) = "9 Some Street"
MyTable(0,2) = "Anytown"
MyTable(0,3) = "0123456789"
MyTable(1,0) = "Warda"  ' And now Warda...
...and so on...
```

لكن لا توجد طريقة سهلة لملء البيانات دفعةً واحدةً كما فعلنا في قوائم بايثون، بل يجب أن نملأ كل حقل على حدة، وعلى الرغم من وجود الدالة Array في VBScript والتي تستطيع ملء مصفوفة دفعةً واحدةً -مما يعني الحاجة إلى جعلها متشعبةً-؛ فستصبح معقدةً وصعبة القراءة.

إذا دمجنا قواميس VBScript مع هذه الخاصية لمصفوفاتها، فسنحصل على نفس النتيجة التي حصلنا عليها في بايثون، كما يلي:

```
<script type="text/VBScript">
Dim addressBook
Set addressBook = CreateObject("Scripting.Dictionary")
Dim Samy(3)
Samy(0) = "Samy"
Samy(1) = "9 Some St"
Samy(2) = "Anytown"
Samy(3) = "0123456789"
addressBook.Add "Samy", Samy

MsgBox addressBook.Item("Samy")(3) ' Print the Phone Number
</script>
```

وآخر ملاحظة متعلق بمصفوفات VBScript هو أنها لا تحتاج إلى أن تكون ثابتة الحجم، على أن هذا لا يعني استمرار إضافة العناصر إليها كما في القوائم، بل يجب أن نغير حجم المصفوفة صراحةً، بالتصريح عن مصفوفة ديناميكية، وذلك بإهمال ذكر الحجم ببساطة، كما يلي:

```
Dim DynArray()  ' no size specified
```

ثم إذا أردنا تغيير حجمها نفعل ذلك كما يلي:

```
<script type="text/vbscript">
Dim DynArray()
ReDim DynArray(5)  ' Initial size = 5
DynArray(0) = 42
DynArray(4) = 26
MsgBox "Before: " & DynArray(4)  ' prove that it worked
' Resize to 21 elements keeping the data we already stored
ReDim Preserve DynArray(20)
DynArray(15) = 73
```

```
MsgBox "After Preserve: " & DynArray(4) & " " & DynArray(15)' Old and
new still there
' Resize to 51 items but lose all data
Redim DynArray(50)
MsgBox "Without Preserve: " & DynArray(4) & " Oops, Where did it go?"
</script>
```

لاحظ أن السلوك الافتراضي هو حذف كل محتويات المصفوفة عند تغيير حجمها، فإذا أردنا الحفاظ على محتوياتها؛ فيجب أن نخبر VBScript أن تحفظ المحتويات باستخدام `Preserve`.

هذا الأسلوب غير مريح موازنةً بالقوائم التي تغير حجمها تلقائيًا، لكنه يعطي المبرمج تحكمًا أفضل في كيفية تصرف البرنامج، مما يحسن الأمان -ضمن أمور أخرى- نظرًا لأن بعض الفيروسات قد تستغل مخازن البيانات القابلة لتغيير حجمها ديناميكيًا.

5.9.7 مصفوفات جافاسكربت

رغم أنه يطلَق على هذا النوع من البيانات في جافاسكربت مصفوفة؛ إلا أنها تسمية خاطئة، فهو مزيج غريب يجمع بين مزايا القوائم والقواميس والمصفوفات العادية، حيث نستطيع مثلًا التصريح عن مصفوفة من عشرة عناصر من نوع ما كما يلي:

```
var items = new Array(10);
```

لاحظ استخدام كلمة `new` المفتاحية، إذ تشبه دالة `CreateObject()` التي استخدمناها في VBScript لإنشاء قاموس، كذلك فقد استخدمنا الأقواس لتحديد حجم المصفوفة، نستطيع الآن أن نملأ المصفوفة ونصل إليها كما يلي:

```
items[4] = 42;
items[7] = 21;
var aValue = items[4];
```

وهنا نستخدم الأقواس المربعة للوصول إلى عناصر المصفوفة، وتبدأ الفهارس فيها من الصفر، لكن رغم هذا كله فمصفوفات جافاسكربت ليست مقيدة بتخزين نوع واحد من البيانات، بل يمكن إسناد أي شيء إلى عناصرها:

```
items[9] = "A short string";
var msg = items[9];
```

كما يمكن إنشاء مصفوفات من خلال توفير قائمة بالعناصر:

```
var moreItems = new Array("one","two","three",4,5,6);
aValue = moreItems[3];  // -> 4
msg = moreItems[0];   // -> "one"
```

كذلك يمكن تحديد طول المصفوفة باستخدام خاصية تدعى length، التي تستخدَم كما يلي:

```
var size = items.length;
```

وعلى غرار أسلوب بايثون في استدعاء الدوال في وحداتها، فإن الصياغة name.property هنا هي نفسها، لكن دون أقواس لاحقة.

ورغم أن مصفوفات جافاسكربت تبدأ بالصفر كما ذكرنا؛ إلا أن فهارسها ليست مقصورةً على الأرقام وحدها، ولهذا يمكن استخدام السلاسل النصية أيضًا -والتي تكون هنا أشبه بالقواميس-، كما يمكن توسيع مصفوفة بإسناد قيمة إلى فهرس تكون أكثر من الحد الأقصى الحالي، مما يعني أننا لا نحتاج إلى تحديد الحجم عند إنشاء المصفوفة، رغم كون ذلك سلوكًا مستحسنًا في البرمجة، ويوضح المثال التالي خصائص مصفوفات جافاسكربت التي ذكرناها:

```
<script type="text/javascript">
var items = new Array(10);
var moreItems = new Array(1);
items[42] = 7;
moreItems["foo"] = 42;
msg = moreItems["foo"];
document.write("msg = " + msg + " and items[42] = " + items[42] );
</script>
```

إذا شغّلنا تلك الشيفرة في المتصفح فسنرى القيم مكتوبةً على الشاشة. تحقق إن كنت تستطيع متابعة الإسنادات في الشيفرة لتتأكد من سبب ظهور القيم التي تراها على الشاشة.

لننظر أخيرًا في مثال دفتر العناوين مرةً أخرى، باستخدام مصفوفات جافاسكربت هذه المرة:

```
<script type="text/javascript">
var addressBook = new Array();
addressBook["Samy"] = new Array("Samy", "9 Some St", "Anytown",
"0123456789");
addressBook["Warda"] = new Array("Warda", "11 Nother St", "SomePlace",
"0987654321");


document.write(addressBook.Warda);
```

```
</script>
```

يمكن الوصول إلى المفتاح كما لو كان خاصيةً مثل `length`، ويتبين مما سبق أن مصفوفات جافاسكربت ما هي إلا هياكل بيانات مرنة للغاية.

5.9.8 المكدس Stack

يكدِّس العاملون في المطاعم الأطباق النظيفة فوق بعضها، ثم تؤخَذ واحدًا تلو الآخر للعملاء بدءًا من الأعلى، فيكون الطبق الأخير هو آخر طبق يُستخدم، وفي نفس الوقت أقل طبق يُستخدم، يشبه هذا المثال مكدسات البيانات بالضبط، إذ نضع العنصر في قمة المكدس أو نأخذ عنصرًا منه، ويكون العنصر المأخوذ هو آخر عنصر مضاف إلى المكدس، ويطلَق على هذه الخاصية للمكدسات "آخرها دخولًا أولها خروجًا" أو LIFO اختصارًا إلى Last In First Out. من الخصائص المفيدة في المكدسات أننا نستطيع عكس قائمة العناصر عن طريق دفع القائمة في المكدس ثم إخراجها مرةً أخرى، فتكون النتيجة عكس القائمة الأولى.

لا تأتي المكدسات مدمجةً في بايثون ولا VBScript ولا جافاسكربت، بل يجب كتابة بعض التعليمات البرمجية في البرنامج من أجل الحصول على سلوك المكدس، والقوائم هي أفضل نقطة بداية في الغالب للمكدسات بما أنه يمكن زيادة حجمها عند الحاجة.

تدريب

اكتب مكدسًا باستخدام قائمة في بايثون، وتذكر أننا نلحق العناصر في نهاية القائمة باستخدام `append()`، ونحذفها باستخدام فهارسها عن طريق `del()`، كما تستطيع استخدام `-1` مثل فهرس يشير إلى آخر عنصر في القائمة.

يجب أن تكون قادرًا على كتابة برنامج -مستفيدًا من هذه الإرشادات- يدفع 4 محارف إلى قائمة ثم يخرجها مرةً أخرى، ثم اطبعها بعد ذلك. راقب الترتيب الذي تستدعي به `print` و `del`، وإذا نجحت في هذا، فجرب طباعتها بعكس الترتيب الذي أدخلتها به، استخدم العملية `pop()` الخاصة بقوائم بايثون والتي تعيد العنصر الأخير وتحذفه في خطوة واحدة، من أجل تسهيل التدريب.

5.9.9 الحقيبة Bag

في سياق الحديث عن البيانات؛ تُعَد الحقيبة تجميعةً من العناصر التي ليس لها ترتيب محدد، ويمكن أن تحتوي على عناصر مكررة، وتكون لها عادةً عوامل تمكننا من إضافة تلك العناصر وحذفها وإيجادها، فما هي إلا قوائم في اللغات الثلاث التي نستخدمها في الكتاب.

5.9.10 المجموعات Sets

تخزن المجموعة نوعًا واحدًا من العناصر، ونستطيع التحقق مما إذا كان العنصر موجـودًا في المجموعـة أم لا -أي عضوًا فيها-، إلى جانب إضافة تلك العناصر وحذفها، بالإضافة إلى دمج مجموعتين معًا بطرق شـتى تتوافـق مع نظرية المجموعات في الرياضيات، مثل الاتحاد والتقاطع وغيرها، لكن على هذا فالمجموعات لا تكـون مرتبـةً ولا تهتم للترتيب.

لا تسـتخدم جافاسـكربت ولا VBScript المجموعـات مباشـرةً، لكن يمكن تنفيـذ سـلوك قـريب باسـتخدام القواميس التي لا تحتوي إلا على مفاتيح بقيم فارغة؛ أما بايثون فتدعم المجموعات مثل نوع من أنـواع البيانـات فيها، وأحد الاستخدامات البسيطة لها ما يلي:

```
>>> A = set()  # أنشئ مجموعة فارغة
>>> B = set([1,2,3]) # مجموعة من قائمة
>>> C = {3,4,5} # تهيئة مثل [] في القوائم
>>> D = {6,7,8}
>>> # جرب الآن بعض عمليات القوائم
>>> print( B.union(C) )
{1, 2, 3, 4, 5}
>>> print( B.intersection(C) )
{3}
>>> print( B.issuperset({2}) )
True
>>> print( {3}.issubset(C) )
True
>>> print( C.intersection(D) == A )
True
```

لاحظ أننا نستطيع تهيئة المجموعة باستخدام الأقواس المعقوصة، لكن هذا قد يتشابه مـع القـواميس، لـذا نفضـل اسـتخدام الصـيغة `set([...])` رغم أنهـا تتطلب كتابـةً أكـثر، فيمـا يلي اختصـارات لعمليـات الاتحاد والتقاطع:

```
>>> print( B & C ) # كما في B.intersection(C)
>>> print( B | C ) # كما في B.union(C)
```

نستطيع أخيرًا التحقق من وجود عنصر ما في مجموعة باستخدام العامل `in`:

```
>>> print( 2 in B )
```

```
True
```

هذه العمليات كافية إلى الآن لما نشرحه، رغم وجود عمليات أخرى على المجموعات.

5.9.11 الطابور Queue

يشبه الطابور أو قائمة الانتظار؛ المكدس، باستثناء أن العنصر الأول فيه الذي يخرج أولًا أيضًا، ويسمى هذا السلوك "الأول دخولًا الأول خروجًا" أو FIFO اختصار للعبارة First In First Out، ويُنشَأ الطابور باستخدام قائمة أو مصفوفة.

تدريب

جرب كتابة طابور باستخدام قائمة، وتذكر أنك تستطيع الإضافة إلى القائمة باستخدام `append()`، وأن تحذف من موضع ما باستخدام `del()`. أضف أربعة محارف إلى الطابور ثم استخرجها واطبعها، غذ يجب أن تُطبع المحارف بنفس الترتيب الذي دخلت به.

توجد أنواع أخرى من تجميعات البيانات، لكن ما ذكرناه هنا يغطي أغلب الحالات التي ستراها أثناء البرمجة، بل لن نستخدم إلا قليلًا مما ذكرناه هنا في الشرح، لكن قد ترى بقية الأنواع في المقالات أو مجموعات النقاشات البرمجية.

5.10 الملفات Files

ينبغي أن يكون مستخدم الحاسوب معتادًا على التعامل مع الملفات بما أنها تمثل أساس كل شيء نفعله على الحواسيب، ولا غرو إذًا أن نكتشف أن أغلب لغات البرمجة توفر نوع file خاص من البيانات، لكن أهمية الملفات ومعالجتها ستجعلنا نرجئ الحديث عنها وشرحها إلى فصل لاحق خاص بها.

5.11 الوقت والتاريخ

يُعطى التاريخ والوقت في العادة أنواعًا خاصةً بهما في البرمجة، لكن قد يمثَّلان أحيانًا في صورة عدد كبير (عدد الثواني منذ وقت أو تاريخ قديم، مثل الوقت الذي كُتب فيه نظام التشغيل)، لكن قد يكون نوع البيانات في أحيان أخرى هو ما يُعرف بالنوع المعقد الذي سنذكره في الفصل التالي، وهو يسهل استخراج الشهر واليوم والساعة.

سننظر في استخدام بايثون لوحدة `time` في فصل تالٍ، أما جافاسكربت وVBScript فلديهما آليات خاصة للتعامل مع الوقت، لكننا لن ندرسها.

5.12 النوع الذي يعرفه المستخدم User Defined Type

قد لا تكفي الأنواع التي ذكرناها سابقًا لاحتياجاتنا البرمجية حتى لو دمجناها معًا، فقد نريد جمع عدة أجزاء من البيانات معًا ثم نعاملها مثل عنصر واحد، كما في وصف العنوان مثلًا، حيث يكون على هيئة "رقم المنزل والشارع والمدينة ، ثم الرمز البريدي". تسمح أغلب اللغات بجمع مثل هذه المعلومات معًا في سجل أو هيكل بيانات أو في هيئة صنف Class، وهذا الأخير في البرمجة كائنية التوجه Object Oriented.

5.12.1 VBScript

تبدو تعريفات السجلات في VBScript كما يلي:

```
Class Address
     Public HsNumber
     Public Street
     Public Town
     Public ZipCode
End Class
```

تعني الكلمة المفتاحية `Public` أن البرنامج يستطيع الوصول إلى البيانات، وعليه توجد كلمة `Private` التي يمكن استخدامها لجعل البيانات خاصةً ومقصورةً على أجزاء بعينها من البرنامج، كما سنرى لاحقا.

5.12.2 بايثون

يختلف الأمر قليلًا في بايثون:

```
>>> class Address:
...   def __init__(self, Hs, St, Town, Zip):
...     self.HsNumber = Hs
...     self.Street = St
...     self.Town = Town
...     self.ZipCode = Zip
...
```

سنترك شرح معنى كل من `self` و `def __init__(...)` إلى حين الحديث عن البرمجة كائنية التوجه، أما الآن فنريد ملاحظة أن هناك شرطتين سفليتين حول `init`، وهذا أسلوب متبع في بايثون. لاحظ كذلك ضرورة استخدام المسافات الموضحة أعلاه، ذلك أن بايثون دقيقة في تحديد المسافات. قد يواجه البعض مشاكل أثناء كتابة ما يتشابه مع هذا المثال في محث بايثون، وستجد في نهاية هذا الفصل شرحًا مختصرًا لهذه

المشكلة، ثم سنشرحها بالتفصيل في جزء تالٍ من الكتاب، فإذا أردت كتابة المثال أعلاه في محث بايثون؛ فتأكد من نسخ المسافات البادئة لكل سطر.

والأمر الذي نريد التوكيد عليه هنا هو أننا جمعنا عدة أجزاء من بيانات مرتبطة ببعضها بعضًا ثم وضعناها في هيكل واحد هو `Address`، كما فعلنا في VBScript قبل قليل.

5.12.3 جافاسكربت

توفر جافاسكربت اسمًا غريبًا لهيكل بيانات العنوان في حالتنا، وهو `function`، إذ ترتبط الدوال عادةً بالعمليات وليس تجميعات البيانات، غير أن دوال جافاسكربت تستطيع التعامل مع كليهما، فإذا أردنا إنشاء كائن العنوان في جافاسكربت؛ فسنكتب الآتي:

```
function Address(Hs,St,Town,Zip)
{
   this.HsNum = Hs;
   this.Street = St;
   this.Town = Town;
   this.ZipCode = Zip;
}
```

نريد مرةً أخرى تجاهل الصياغة واستخدام الكلمة `this` هنا، والنظر إلى النتيجة التي ستكون مجموعةً من عناصر البيانات التي نستطيع القول بأنها تصف عنوانًا، ويمكننا معاملته على أنه وحدة واحدة.

والآن، كيف نصل إلى تلك البيانات بعد إنشاء هياكلها؟

5.13 الوصول إلى الأنواع التي يعرفها المستخدم

من الممكن إسناد نوع بيانات معقد إلى متغير، لكن لا يمكن الوصول إلى الحقول المنفردة دون استخدام بعض آليات الوصول الخاصة التي تحددها اللغة، وتكون نقطة `.` في الغالب.

5.13.1 VBScript

إذا استخدمنا حالة صنف العنوان الذي عرّفناه أعلاه، فسنكتب ما يلي:

```
Dim Addr
Set Addr = New Address

Addr.HsNumber = 7
Addr.Street = "High St"
```

```
Addr.Town = "Anytown"
Addr.ZipCode = "123 456"

MsgBox Addr.HsNumber & " " & Addr.Street & " " & Addr.Town
```

نحدد هنا بُعد متغير جديد هو Addr باستخدام Dim، ثم نستخدم الكلمتين Set و New لإنشاء نسخة جديدة من صنف Address، ثم نسند القيم إلى حقول نسخة العنوان الجديدة، ثم نطبع العنوان في صندوق رسالة.

5.13.2 بايثون

على فرض أننا كتبنا تعريف الصنف أعلاه:

```
>>> Addr = Address(7,"High St","Anytown","123 456")
>>> print( Addr.HsNumber, Addr.Street, Addr.Town )
High St Anytown
```

ينشئ هذا نسخةً من نوع Address ويسند إليه المتغير Addr، ونستطيع تمرير قيم الحقل إلى الكائن الجديد عند إنشائه في بايثون، ثم نطبع حقول HsNumber و Street للنسخة المنشأة حديثًا باستخدام عامل النقطة . .

كما يمكن إنشاء نسخ عنوان متعددة لكل منها قيمه المستقلة لأرقام البيوت والشوارع وغيرها. جرب هذا بنفسك لتتدرب عليه، فهل تستطيع معرفة كيفية استخدام هذا في مثال دفتر العناوين؟

5.13.3 جافاسكربت

تشبه آلية جافاسكربت هنا ما سبق ذكره لكن مع بعض التعديل، غير أن الآلية الأساسية بسيطة ومباشرة:

```
var addr = new Address(7, "High St", "Anytown", "123 456");
document.write(addr.HsNum + " " + addr.Street + " " + addr.Town);
```

إحدى الآليات التي يمكن استخدامها في جافاسكربت أيضًا هي معاملة الكائن مثل قاموس واستخدام اسم الحقل كمفتاح:

```
document.write( addr['HsNum'] + " " + addr['Street'] + " " +
addr['Town']);
```

وما لم نُعط اسم الحقل في هيئة سلسلة نصية كما في حالة قراءة ملف أو إدخال مستخدم للبرنامج، فلا ننصح باستخدام هذا النموذج.

5.14 العوامل التي يعرِّفها المستخدم

يمكن للأنواع التي يعرِّفها المستخدم أن تكون لها عمليات معرفة لها كذلك، وذلك أساس ما يعرف بالبرمجة كائنية التوجه. الكائن هو عبارة عن تجميعة من عناصر البيانات والعمليات المرتبطة بتلك البيانـات، وهي مغلفـة جميعًا في وحدة واحدة، حيث تستخدم بايثون الكائنات على نطاق واسع في مكتبتهـا القياسـية للوحـدات، كمـا تسمح لنا كوننا مبرمجين بإنشاء أنواع الكائنات الخاصة بنا.

يحصل الوصول إلى عمليات الكائنات بنفس الطريقـة الـتي يمكن الوصـول بهـا إلى أعضـاء البيانـات لنـوع معرَّف من قِبل المستخدم، أي من خلال عامل النقطة؛ أما غير هذا فتكون أشـبه بالـدوال، وتسـمى تلـك الـدوال الخاصة بـالتوابع methods، وقـد رأيناهـا في حالـة العمليـة append() في القـوائم، أينمـا تـذكر أنـه من أجـل استخدامها يجب أن نربط استدعاء الدالة باسم المتغير.

```
>>> listObject = []    # قائمة فارغة
>>> listObject.append(42) # استدعاء تابع لكائن القائمة
>>> print( listObject )
[42]
```

يجب استيراد الوحدة عند توفير نوع كائن - يُسمى صنفًا- في وحدة بايثون كمـا فعلنـا في sys من قبـل، ثم كتابة اسم الوحدة قبل اسم نوع الكائن عند إنشاء نسخة يمكن تخزينها في متغـير مـع اسـتخدام الأقـواس طبعًـا، بعد ذلك نستطيع استخدام المتغـير دون اسـتخدام اسـم الوحـدة، ولتوضـيح هـذا لنَعُـد إلى وحـدة array الـتي ذكرناها من قبل، فهي توفر الصنف array، ولنستورد هذه الوحـدة ثم ننشـئ نسـخةً من array، ونسـند اسـم myArray إليها، ثم نستخدم myArray فقط للوصول إلى عملياتها وبياناتها كما يلي:

```
>>> import array
>>> myArray = array.array('d')  # array of reals(d), use module name
>>> myArray.append(42)         # use array operation
>>> print( myArray[0]] )  # access array content
42.0
>>> print( myArray.typecode )  # access array attribute
'd'
```

نستورد الوحدة array في السطر الأول إلى البرنامج، ثم نسـتخدم تلـك الوحـدة في السـطر الثـاني لإنشـاء نسخة من صنف array باستدعائها مثل دالة، ونحتاج هنا إلى توفير سلسلة الرموز النوعية typecode string التي توضح نوع البيانات التي يجب تخزينها. تذكر أن المصفوفات الخالصة تخزن نوعًا واحدًا من البيانـات، حيث سنصـل في السـطر الثـالث إلى إحـدى عمليـات صـنف المصـفوفة، وهي append() الـتي تعامـل الكـائن myArray كأنه وحدة وكأن العملية داخل الوحدة، ثم نستخدم الفهرسة لجلب البيانات الـتي أضـفناها، نسـتطيع

في الأخير الوصول إلى بعض البيانات الداخلية، إذ يخزن `typecode` النوع الذي مررناه عند إنشاء المصفوفة، من داخل كائن `myArray` صياغةً تشبه صياغة الوحدة.

لا يوجد فرق كبير بين استخدام الكائنات التي توفرها الوحدات، وبين الدوال الموجودة في الوحدات، باستثناء الحاجة إلى إنشاء نسخة instance، ويمكن النظر إلى اسم الكائن على أنه عنوان يحفظ الدوال والمتغيرات المرتبطة به مجموعةً معًا.

إحدى الطرق الأخرى التي ننظر بها للأمر هي أن الكائنات تمثل أشياء من العالم الحقيقي ونحن نستخدمها أو نفعل بها أمورًا تنفعنا، وهذا المنظور على بساطته إلا أنه الفكرة التي نشأت بسببها الكائنات في البرمجة، فنحن نكتب -برمجيًا- محاكاةً لمواقف في العالم الحقيقي.

تستطيع كل من جافاسكربت وVBScript أن تتعامل مع الكائنات، وقد كان هذا ما استخدمناه في كل أمثلة العناوين أعلاه، فقد عرفّنا صنفًا ثم أنشأنا نسخةً أسندناها إلى متغير كي نستطيع الوصول إلى خصائص تلك النسخة، ولهذا ارجع إن شئت إلى ما سبق من الشرح عن الأصناف والكائنات، وانظر كيف توفر الأصناف آليات لإنشاء أنواع جديدة من البيانات في برامجنا من خلال الربط بين بيانات النوع الجديد وعملياته معًا.

5.14.1 العوامل الخاصة بپايثون

إنّ الهدف من هذا الكتاب هو تعليم البرمجة للمبتدئين، ورغم أننا نستخدم پايثون في الشرح؛ إلا أنك تستطيع قراءة لغة برمجة أخرى واستخدامها بدلًا منها، بل إن هذا هو عين ما نتوقعه منك، إذ أنه لا توجد لغة برمجة بما فيها پايثون، تستطيع تنفيذ جميع المهام وحل كل المشاكل التي تواجهك برمجيًا، وبسبب ذات الهدف لن نشرح جميع الخصائص الموجودة في پايثون، بل سنركز على تلك التي يمكن إيجادها في لغات البرمجة الأخرى كذلك، ولهذا سنهمل شرح بعض الخصائص بالغة القوة في پايثون، والتي تميزها عما سواها، مثل العوامل الخاصة special operators، ذلك أن أغلب لغات البرمجة تدعم عمليات بعينها لا تدعمها لغات أخرى، وتلك العوامل الفريدة هي التي تسبب ظهور لغات برمجة جديدة، وهي من العوامل المهمة في تحديد شهرة تلك اللغات من حيث الاستخدام فيما بعد.

حيث تدعم پايثون مثلًا عمليات غير شائعة نسبيًا في لغات البرمجة الأخرى، مثل تشريح القائمة `spam[X:Y]` (إلى شرائح) list slicing من أجل استخراج شريحة من وسط القائمة أو السلسلة النصية أو الصف tuple، كما تدعم إسناد الصف Tuple الذي يسمح لنا بإسناد عدة قيم متغيرات مرةً واحدةً: `X, Y = (12, 34)`. كما تسهل تنفيذ العمليات على كل عضو من أعضاء تجميعة ما باستخدام الدالة `map()` التي سنشرحها في فصل البرمجة الدالية أو الوظيفية، إضافةً إلى الكثير من الخصائص التي جعلت پايثون تشتهر بأنها تأتي "مع البطارية" إشارةً إلى كثرة الملحقات والمزايا التي تأتي معها. ارجع إلى توثيق پايثون للمزيد عن تلك المزايا.

تجدر أخيرًا الإشارة إلى أنه رغم قولنا بأن تلك العوامل خاصة ببايثون؛ إلا أن هذا لا يعني أنها لا توجد في لغات برمجة أخرى، بل أنها لن توجد مجتمعةً في كل لغة، فالعوامل التي نشرحها في الكتاب تكون متاحةً بشكل أو بآخر في أغلب لغات البرمجة الحديثة.

يلخص هذا نظرتنا على المواد الخام للبرمجة، وسننتقل الآن إلى موضوع آخر نرى فيه كيف نستغل تلك المواد الخام في البرمجة الحقيقية.

5.15 شرح لمثال العناوين

رغم قولنا أننا سنشرح تفاصيل هذا المثال لاحقًا، إلا أننا رأينا صعوبة تنفيذ مثال بايثون من بعض القراء، ولهذا سيشرح هذا القسم شيفرة بايثون سطرًا سطرًا.

ستبدو شيفرة المثال كاملة كما يلي:

```
>>> class Address:
...   def __init__(self, Hs, St, Town, Zip):
...     self.HsNumber = Hs
...     self.Street = St
...     self.Town = Town
...     self.Zip_Code = Zip
...
>>> addr = Address(7,"High St","Anytown","123 456")
>>> print( addr.HsNumber, addr.Street )
```

وسنشرح الآن سطرًا سطرًا:

```
>>> class Address:
```

تخبر التعليمة `class` بايثون أننا على وشك تعريف نوع جديد اسمه `Address` في هذه الحالة، وتشير النقطتان الرأسيتان إلى أن أي أسطر مزاحة تالية ستكون جزءًا من تعريف الصنف، وينتهي التعريف عند أول سطر غير مزاح، فإذا كنت تستخدم IDLE؛ فسترى أن المحرر قد أزاح السطر التالي تلقائيًا، أما إذا كنت تعمل من محث بايثون في سطر أوامر DOS مثلًا، فسيكون عليك إزاحة الأسطر كما هو موضح أعلاه، ولا تهتم بايثون بمقدار الإزاحة طالما أنها نفسها لكل سطر.

أما السطر الثاني:

```
...   def __init__(self, Hs, St, Town, Zip):
```

يطلق على العنصر الأول في الصنف الخاص بنا "تعريف التابع method definition"، ويجب أن يكون حول الاسم شرطتان سفليتان على كل ناحية منه، وهي طريقة بايثون للأسماء التي لها أهمية خاصة، ويسمى ذلك التابع `__init__`، وهو عملية خاصة تنفذها بايثون حين ننشئ نسخةً من الصنف الجديد الخاص بنا كما سنرى بعد قليل؛ أما النقطتان الرأسيتان فتخبران بايثون أن مجموعة الأسطر المزاحة التالية ستكون هي التعريف الحقيقي للتابع.

والسطر الثالث:

```
...     self.HsNumber = Hs
```

يُسنِد هذا السطر -والأسطر الثلاثة التالية- قيمًا إلى الحقول الداخلية للكائن الخاص بنا، وهذه الأسطر مزاحة من تعليمة `def` لتخبر بايثون أنها تشكل التعريف الحقيقي لعملية `__init__`، أما السطر الفارغ، فيخبر مفسر بايثون بأن تعريف الصنف قد انتهى، لنعود مرةً أخرى إلى محث بايثون المعتاد `>>>`.

```
>>> addr = Address(7,"High St","Anytown","123 456")
```

ينشئ هذا نسخةً جديدةً من النوع `Address`، وتستخدم بايثون عملية `__init__` المعرَّفة أعلاه لإسناد القيم التي أعطيناها إلى الحقول الداخلية، وتسنَد النسخة إلى متغير `addr` كما تسنَد أي نسخة لأي نوع بيانات آخر.

```
>>> print( addr.HsNumber, addr.Street )
```

نطبع الآن قيم حقلين من الحقول الداخلية مستخدمين عامل النقطة للوصول إليهما.

5.16 خاتمة

يُعَد هذا الفصل دسمًا وغنيًا بالمعلومات النظرية التي تؤسس لتعلم البرمجة على أسس سليمة، وسنعود إلى ما فيه مرات كثيرة أثناء الشرح، بما أن البيانات وأنواعها هي اللبنات التي تُصنع منها البرامج، لكن نريد التذكير هنا بأن بايثون تسمح لنا بإنشاء أنواع البيانات التي نريدها بأنفسنا ونستخدمها مثل أي نوع مضمّن فيها، وأن المتغيرات تشير إلى بيانات، وقد نحتاج إلى التصريح عنها قبل تعريفها، وهي تأتي في أنواع شتى، ويتوقف نجاح العملية التي نجريها على نوع البيانات الذي نستخدمه.

ومن أنواع البيانات البسيطة سلاسل المحارف والأعداد والقيم البوليانية، كما تشمل أنوع البيانات المعقدة التجميعات والملفات والبيانات وأنواع البيانات المعرّفة من قِبل المستخدم.

توجد أيضًا العديد من العوامل في كل لغة برمجة، ويُعَد تعلّم تلك العوامل جزءًا من التعود على أنواع بياناتها وعلى العمليات المتاحة لتلك الأنواع، وقد يكون العامل الواحد متاحًا لعدة أنواع، غير أن النتيجة قد لا تكون هي نفسها في كل مرة، بل ربما تختلف اختلافًا كبيرًا.

6. المزيد من التسلسلات وأمور أخرى

إذا قرأت الفصول السابقة من الكتاب فأنت على دراية بأننا تعلمنا فيها كيف نكتب أوامر بسيطةً وحيدة المدخلات في بايثون، وعرفنا أهمية البيانات وما نفعله بها، ثم كتبنا تسلسلات أطول قليلًا (5-10 أسطر). وها نحن نقترب من كتابة برامج مفيدة، لكن لا زالت لدينا عقبة هنا، وهي خسارة البيانات والبرامج كلما خرجنا من بايثون على الحاسوب، وإذا كنت تستخدم جافاسكربت أو VBScript فسترى أنك قد خزنت تلك الأمثلة في ملفات تستطيع تشغيلها وقتما أردت، ونريد أن نفعل ذلك مع بايثون أيضًا، وقد ذكرنا من قبل أن هذا ممكن باستخدام أي محرر نصي مثل notepad أو pico ، وذلك بحفظ ملف بامتداد `.py` ثم تشغيل الملف من محث أوامر نظام التشغيل مع كتابة اسم السكربت مسبوقًا بكلمة `python`، يستخدم المثال التالي تسلسلًا من أوامر بايثون، فيه كل النقاط التي شرحناها من قبل:

```
#  File: firstprogram.py
print( "hello world" )
print( "Here are the ten numbers from 0 to 9\n0 1 2 3 4 5 6 7 8 9" )
print( "I'm done!" )
# نهاية الملف
```

لاحظ أن السطرين الأول والأخير ليسا ضروريين، وهما تعليقان سنشرحها في هذا الفصل، وقد أضفناهما لنوضح ما يحدث في الملف.

يمكن استخدام notepad أو أي محرر نصي لإنشاء الملف طالما سنحفظه بصيغة نصية مجردة، ولا نستخدم معالج نصوص متقدمًا مثل MS Word، لأنه يحفظ الملفات بصيغة ثنائية خاصة لا تستطيع بايثون فهمها، ولا نستخدم صيغة html التي تحوي كثيرًا من النصوص المنسقة التي لا تعني شيئًا لبايثون كذلك.

لتشغيل هذا البرنامج افتح سطر أوامر نظام التشغيل الآن، يمكنك مراجعة فصل البداية في تعلم البرمجة لمعرفة سطر أوامر النظام، وانتقل إلى المجلد الذي حفظت فيه ملف بايثون لديك، ثم نفّذه بكتابة python قبله كما ذكرنا:

```
D:\PROJECTS\Python> python firstprogram.py
hello world
Here are the ten numbers from 0 to 9
1 2 3 4 5 6 7 8 9
I'm done!

D:\PROJECTS\Python>
```

في السطر الأول محث أوامر ويندوز إضافةً إلى الأمر الذي كتبناه، ثم خرج البرنامج معروضًا قبل أن يظهر محث ويندوز مرةً أخرى، لكن توجد طريقة أسهل، فقد يكون لدينا أكثر من إصدار لبايثون على نفس الحاسوب إذا كان النظام نفسه يستخدم الإصدار الثاني منها لبعض المهام مثلًا، وإذا كتبنا python في تلك الحالة فقد نحصل على أخطاء لم نتوقعها، ولتجنب مثل تلك الحالة نلحق رقم إصدار بايثون الذي نريد استخدامه إلى الاسم، فإذا كنا نريد الإصدار الثالث نكتب:

```
D:\PROJECTS\Python> python3 firstprogram.py
```

لاحظ أنها صارت الآن python3 بدلًا من python فقط، وبذلك سنتجنب كثيرًا من الأخطاء على أغلب نظم التشغيل.

6.1 مزايا استخدام البيئة المتكاملة

يثبَّت مع بايثون تلقائيًا عند تثبيتها على الحاسوب تطبيق مفيد، وهو برنامج مكتوب بلغة بايثون أيضًا، ويسمى IDLE اختصارًا لبيئة التطوير المتكاملة Integrated Development Environment، وهو يعني أن فيه أدوات عديدةً مدمجةً فيه لمساعدة المبرمجين، ولن نتعمق فيه هنا، لكن نريد تسليط الضوء على مزيتين فيه، أولاهما أنه يوفر نسخةً محسنةً من محث بايثون >>> فيها تظليل للكلمات، حيث تُعرَض خصائص اللغة بألوان مختلفة لإبرازها، إضافةً إلى محرر نصي خاص ببايثون، يسمح لك بتشغيل ملفات برنامجك مباشرةً من داخل IDLE، وننصحك بتجربة IDLE إذا لم تكن قد فعلت، وإذا أردت شرحًا للبرنامج فانظر دليل داني يو Danny Yoo عنه.

أما في حالة استخدام ويندوز فثمة خيار آخر هو PythonWin الذي يمكن تحميله ضمن حزمة PyWin32، والذي يسمح لنا بالوصول إلى جميع دوال البرمجة الدنيا لمكتبة MFC الخاصة بويندوز، وهو بديل جيدًا جدًا لبيئة

IDLE، لكنه لا يعمل إلا على ويندوز، وستحصل عليه تلقائيًا إذا حمّلت بايثون من ActiveState، فنسختهم فيها جميع مزايا PyWin32 بما فيها Pythonwin افتراضيًا.

لكن من الناحية الأخرى فإن IDLE يأتي افتراضيًا في حزمة بايثون القياسية، وهو يعمل على أغلب أنظمة التشغيل، لذا يُستخدم أكثر، ولا تهم الأداة التي تستخدمها بأي حال طالما ليست notepad من ويندوز إذ الخيارات المتقدمة أفضل من مجرد محرر نصي بسيط.

أخيرًا، إذا اخترت طريق المحررات النصية فستجد نفسك مع الوقت تعمل أمام ثلاث نوافذ في نفس الوقت، هي نافذة المحرر التي تكتب فيها شيفرتك المصدرية وتحفظها، وجلسة بايثون التي تجرب فيها ما تكتب من خلال محث بايثون قبل إضافتها إلى برنامجك في المحرر، وسطر أوامر نظام التشغيل الذي تستخدمه لتشغيل البرنامج من أجل اختباره.

أما أغلب المبتدئين فيفضلون أسلوب بيئات التطوير المتكاملة التي تأتي كلها في نافذة واحدة مثل IDLE، والأمر كله اختياري فيه سعة ولهذا اختر ما يناسبك، وإذا كنت تستخدم جافاسكربت أو VBScript؛ فنقترح استخدام أحد المحررات النصية المذكورة أعلاه، ومتصفح ويب حديث لجافاسكربت مثل كروم أو فاير فوكس؛ أما بالنسبة للغة VBScript فهذا يعني إنترنت إكسبلورر حصرًا، وليس متصفح Edge حتى، ويكون المتصفح مفتوحًا على الملف الذي تعمل عليه، فإذا أردت تجربة تغيير أو تعديل ما، فاضغط زر إعادة تحميل الصفحة.

6.2 التعليقات السريعة

إحدى أهم الأدوات البرمجية التي لا يشعر المبتدئون بأهميتها هي التعليقات، وهي أسطر داخل البرنامج تصف ما يحدث فيه، وليس لها أي تأثير على أداء البرنامج أو سير عملياته، فهي أسطر وصفية لما يحدث فقط، إلا أن دورها في غاية الأهمية، إذ تخبر المبرمج ما يفعله البرنامج في كل كتلة برمجية أو كل سطر، ولماذا يتصرف على هذا النحو، وتظهر أهمية التعليقات هنا إذا كان البرنامج الذي يقرؤه المبرمج قد كتبه شخص غيره، أو قد كتبه هو بنفسه لكن منذ مدة طويلة فنسي سبب اختياره لخاصية أو دالة بعينها.

وسيشعر المبرمج بأهمية التعليقات حسنة الوصف مع الوقت، وقد أضفنا تعليقات إلى بعض الشيفرات التي استخدمناها في شرح الكتاب إلى الآن، فهي تلك الأسطر المسبوقة بعلامة # في بايثون، أو بعلامة ' في VBScript، وسنبدأ تدريجيًا من الآن بكتابة شرح الشيفرة في التعليقات، إلى أن يقل ما نكتبه من الشرح خارج الشيفرة ثم يختفي.

لكل لغة طريقة في ترميز التعليقات، ففي VBScript مثلًا تكون `REM` مثل اختصار إلى كلمة Remark أو ملاحظة، لكن يشيع استخدام العلامة ' لتؤدي نفس الوظيفة بحيث يتجاهل البرنامج أي شيء مكتوب بعدها.

```
REM لن يُعرض هذا السطر
' ولا هذا
```

```
msgBox "أما هذا السطر فسيُعرض"
```

لاحظ أن استخدام علامة اقتباس مفردةً ' مثل محدد للتعليقات في VBScript هو سبب منع بدء السلسلة النصية بها، لئلا تظن اللغة أنها تعليق، أما بايثون فتستخدم رمز # مثل حدد للتعليقات فيها، وتتجاهل أي شيء يتبعه:

```
v = 12      # أعط v القيمة 12
x = v*v     # x تساوي مربع v
```

غير أن هذا الأسلوب في التعليقات سيء للغاية، إذ لا يجب أن تصف التعليقات ما تفعله الشيفرة فحسب، فنحن نستطيع رؤية ذلك بأنفسنا، لكنها يجب أن تصف سبب اختيار المبرمج لهذه الدالة أو تلك القيمة:

```
v = 3600    # 3600 عدد الثواني في الساعة الواحدة
s = t*3600  # t تحتوي الوقت المار بالساعة
              # لذا نحولها إلى ثوانٍ.
```

هذه النسخة من التعليقات أكثر فائدةً إذ تشرح سبب ضرب t في 3600.

تستخدم جافاسكربت شرطتين مائلتين // مثل محدد للتعليقات، ثم تتجاهل أي شيء بعدهما، كما تسمح بعض اللغات بتعليقات متعددة الأسطر بين زوجين من الرموز، لكن هذا قد يؤدي إلى بعض الأخطاء الخفية إذا لم يوضع رمز الإغلاق، وتسمح جافاسكربت بهذا النوع من التعليقات باستخدام الرمز /* متبوعًا برمز الإغلاق */ بعد نهاية التعليقات:

```
<script type="text/javascript">
document.write("هذا السطر يُطبع\n");
// تعليق من سطر واحد

/* هنا تعليق نقسمه على عدة أسطر
فيحتوي على هذا السطر
وهذا السطر
وذا أيضًا
ولن يظهر أي من هذا في خرج السكربت
فإذا أردنا إنهاء التعليق نعكس رمز البدء ليكون
كما يلي: */

document.write("وهذا أيضًا");
</script>
```

تبرز أهمية التعليقات كما ذكرنا في أنها تشرح الشيفرة لأي شخص يحاول قراءتها، حيث يجب الحرص على توضيح الأقسام الغامضة مثل القيم العشوائية المستخدمة، أو المعادلات الحسابية المعقدة. تذكر أن ذلك القارئ المحتار في فهم شيفرتك قد يكون أنت نفسك بعد بضعة أسابيع أو أشهر.

6.3 التسلسلات باستخدام المتغيرات

لقد شرحنا مفهوم المتغيرات في فصل المواد الخام، وقلنا إنها عناوين نعلّم بها بياناتنا من أجل الإشارة إليها في المستقبل، ورأينا بعض الأمثلة عن استخدام المتغيرات في عدة أمثلة لقوائم ودفاتر عناوين، غير أن المتغيرات بالغة الأهمية في البرمجة إلى الحد الذي يجعلنا نعيد مراجعة كيفية استخدامها مرةً أخرى قبل أن ننتقل إلى جزء جديد.

اكتب ما يلي في محث بايثون، سواءً في صدفة IDLE أو في نافذة أوامر دوس أو يونكس:

```
>>> v = 7
>>> w = 18
>>> x = v + w    # استخدم متغيرنا في عملية حسابية
>>> print( x )
```

ما حدث في الشيفرة أعلاه هو أننا أنشأنا المتغيرات `v` و `w` و`x` وعدّلناها، وهذا أشبه بزرّ M على حاسبة الجيب القديمة التي تخزن نتيجةً لاستخدامها لاحقًا، يمكن تحسين تلك الشيفرة باستخدام سلسلة تنسيق لطباعة النتيجة:

```
>>> print( "The sum of %d and %d is: %d" % (v,w,x) )
```

إحدى مزايا سلاسل التنسيق هنا أننا نستطيع تخزينها في متغيرات أيضًا:

```
>>> s = "The sum of %d and %d is: %d"
>>> print( s % (v,w,x) )   # هذا مفيد عند طباعة نفس الخرج بقيم مختلفة
```

هذا يقصّر كثيرًا من طول تعليمة الطباعة، خاصةً حين تحتوي على قيم كثيرة، لكنه يجعلها أكثر غموضًا، وعندها تقرر بنفسك أيهما الأفضل من حيث القراءة: الأسطر الطويلة أم القصير؟ فإذا أبقينا سلسلة التنسيق هنا إلى جانب تعليمة الطباعة كما فعلنا فهذا حسن.

يفضل تسمية المتغير بطريقة تشرح الغرض من وجوده، فبدلًا من تسمية سلسلة التنسيق باسم `s`، يفضل تسميتها `sumFormat` فتبدو الشيفرة كما يلي:

```
>>> sumFormat = "The sum of %d and %d is: %d"
>>> print( sumFormat % (v,w,x) )   # هذا مفيد عند طباعة نفس الخرج بقيم مختلفة
```

يمكن معرفة أي تنسيق تمت طباعته بمجرد النظر إليه في هذا البرنامج لأنه لا يحتوي على سلاسل تنسيق كثيرة، لكن لا تزال فكرة التسمية المنطقية للمتغيرات ساريةً هنا، وسنحاول قدر الإمكان استخدام أسماء ذات معنى في ما يلي من الشرح؛ أما المتغيرات التي ذكرناها إلى الآن فلم يكن لها معنىً يستحق التسمية.

6.4 أهمية الترتيب

قد يظن المرء أن بنية التسلسل تلك مبالغ فيها لبداهتها، ولعل هذا صحيح بما أنها بديهية فعلًا، لكن الأمر ليس بتلك البساطة التي يبدو عليها، فقد تكون هناك مشاكل خفية كما في حالة الرغبة في ترقية جميع الترويسات في ملف HTML مثلًا، بمقدار رتبة واحدة، انظر المثال التالي، حيث تُميَّز الترويسات في HTML بإحاطة النص بوسم يشير إلى أنها ترويسة مع رتبتها:

```
<h1> نص </h1> لترويسة المستوى الأول
<h2> نص </h2> لترويسة المستوى الثاني
<h3> نص </h3> لترويسة المستوى الثالث
```

والمشكلة هنا أننا حين نصل إلى المستوى الخامس يصير نص الترويسة أصغر من نص المتن العادي نفسه، مما يبدو غريبًا على القارئ إذ يرى نص العنوان أصغر من متنه، وعلى ذلك نريد ترقية جميع الترويسات دفعةً واحدةً، يسهُل هذا بعملية بحث واستبدال في محرر نصي بسيط بأن نستبدل h2 بـ h1 مثلًا، لكن ماذا لو بدأنا بالأرقام الأكبر، مثل h4 إلى h3، ثم h3 إلى h2 وهكذا؛ سنجد في النهاية أن جميع الترويسات صارت من المستوى الأول h1، وهنا تبرز أهمية ترتيب الأحداث والإجراءات، وذلك ينطبق في حالة كتابتنا لبرنامج ينفذ عملية الاستبدال، وهو ما سنفعله على الأغلب بما أن ترقية الترويسات عملية واردة الحدوث، وقد رأينا أمثلةً أخرى نستخدم فيها المتغيرات والتسلسلات في فصل المواد الخام، خاصةً أمثلة دفتر العناوين. جرب التفكير في أمثلة مشابهة لحلها، وإذا أتممت هذا فسننتقل إلى دراسة حالة جديدة نطورها بالتدريج كلما تقدمنا في الكتاب، لنحسنها مع كل تقنية نتعلمها.

6.5 جدول الضرب

سنشرح الآن تدريبًا برمجيًا نطوره معنا أثناء دراسة الفصول التالية، وستتطور حلوله تدريجيًا مع تعلم تقنيات جديدة، ذكرنا أننا نستطيع كتابة سلاسل نصية طويلة بتغليفها بعلامات اقتباس ثلاثية، لنستخدم هذا الأسلوب لبناء جدول ضرب:

```
>>> s = """
x 12 = %d
x 12 = %d
x 12 = %d
x 12 = %d
```

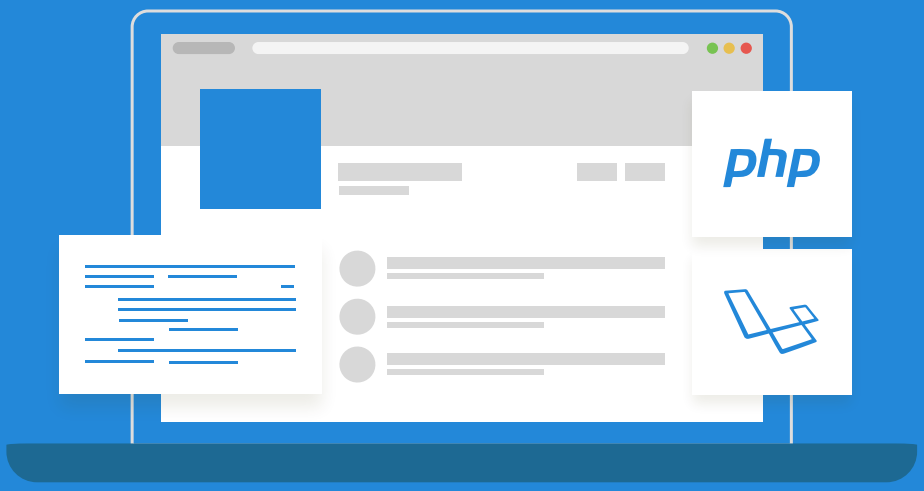
```
"""   # احذر وضع تعليقات في السلاسل النصية
>>>   # إذ ستكون جزءًا من السلسلة نفسها
>>> print( s % (12, 2*12, 3*12, 4*12) )
```

إذا وسعنا الشيفرة السابقة فسنتمكن من طباعة جدول ضرب العـدد 12 كـاملًا من 1 إلى 12، لكن هـل من طريقة أفضل؟ بالتأكيد، وهي ما سنراها في الفصل التالي.

6.6 خاتمة

عرفنا في هذا الفصل أن IDLE هي بيئة تطوير خاصة بكتابة برامج بايثون وتطويرها، وأن التعليقـات تسـهل قراءة البرامج وفهمها، ولا تأثير لها على سير العمليات في البرنامج، وأن المتغيرات تستطيع تخـزين نتـائج بينيـة من أجل استخدامها لاحقًا.

دورة تطوير تطبيقات الويب
باستخدام لغة PHP



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

7. الحلقات التكرارية Loops

رأينا في التدريب الأخير من الفصل السابق كيف طبعنا جزءًا من جدول ضرب العـدد 12، لكن ذلـك تطلب منا كثيرًا من الكتابة، وسيستغرق توسيعه وقتًا طويلًا، ولحسن الحـظ توجـد طريقـة أسـهل سـترينا بدايـة المزايـا القوية التي توفرها لنا لغات البرمجة في تنفيذ المهام وإنجازها.

7.1 حلقات for

تتيح لغة البرمجة تكرار تنفيذ أمر أو عدة أوامر لعـدد محـدَّد من المـرات، في مـا يسـمى حلقـة تكـرار، حيث يمكن استعمال متغير تزيد قيمته مع كل دورة للحلقة، وستبدو هذه العملية في بايثون كما يلي:

```
>>> for n in range(1,13):
...     print( "%d x 12 = %d" % (n, n*12) )
...
x 12 = 12
x 12 = 24
x 12 = 36
x 12 = 48
x 12 = 60
x 12 = 72
x 12 = 84
x 12 = 96
x 12 = 108
x 12 = 120
x 12 = 132
```

```
x 12 = 144
```

لندرس الشيفرة السابقة ونستخرج بعض الملاحظات منها:

- انتهى سطر for بنقطتين رأسيتين (:)، وهذا أمر مهم، إذ يخبر بايثون أن ما سيكتَب بعد هاتين النقطتين هو ما سيُكرَّر.
- كتبنا مجال الأعداد الذي نريده في دالة range() بالشكل range(1,13)، رغم أننا نريد تكرار عملية الضرب إلى العدد 12 فقط، لأن دالة range() تولد الأعداد من العدد الأول فيها إلى العدد الذي يسبق العدد الثاني، 13 في مثالنا، وهذا له أسبابه قطعًا وستعتاده مع الوقت، لكن تجدر الإشارة إلى أن هذه الدالة تستطيع توليد مجموعات معقدة من الأعداد، لكننا اخترنا هذا المثال البسيط لأنه يحقق الغرض.
- العامل for في بايثون هو عامل foreach في الواقع، إذ يطبق تسلسل الشيفرة التالي على كل عنصر في التجميعة، والتي هي في حالتنا قائمة الأعداد التي تولدها الدالة range()، ونستطيع استخدام قائمة أعداد صراحةً عوضًا عن الدالة range()، بالشكل التالي:

```
>>> for n in [1,2,3,4,5,6,7,8,9,10,11,12]:
...     print( "%d x 12 = %d" % (n, n*12) )
```

- أُزيح سطر print موازنةً بسطر حلقة for الذي فوقه، وهذا مهم جدًا لأن هذه الإزاحة تخبر بايثون أن عليها تكرار القسم الخاص بـ print، ومن الممكن إزاحة أكثر من سطرK وستكرر بايثون هذه الأسطر المزاحة جميعًا لكل عنصر في التجميعة، ولا يهم مقدار الإزاحة المستخدمة طالما هي نفسها في كل الأسطر المزاحة، وستخبرنا بايثون إذا وجدت اختلافًا في الإزاحة.
- نحتاج إلى ضغط زر الإدخال Enter مرتين في المفسر التفاعلي لتشغيل البرنامج، لأن مفسر بايثون يضيف سطرًا جديدًا إلى شيفرة الحلقة التكرارية عندما نضغط مرةً واحدةً؛ أما مع الضغطة الثانية فستفترض بايثون أننا أنهينا إدخال الشيفرة، فتشغّل البرنامج.
- وبما أننا قد رأينا الآن هيكل حلقة for، لننظر في كيفية عملها خطوةً خطوةً على النحو الآتي:
- أولًا: تستخدم بايثون دالة range() لتوليد قائمة أعداد من 1 إلى 12، وفقًا لآلية خاصة تسمى بالمولد generator، وتشبه القائمة التي تزيد من عناصرها عنصرًا واحدًا في كل مرة حسب الطلب، مما يوفر الذاكرة عندما تكون القائمة كبيرةً، وهو نفس سبب توليد القائمة صراحةً باستخدام list() عندما طبعنا range() أعلاه، وهذا شبيه بما فعلناه في مثال مفاتيح القاموس keys() في الفصل الخامس: البيانات وأنواعها.
- ثانيًا: تجعل بايثون قيمة n مساويةً للقيمة الأولى في القائمة، وهي 1 في هذه الحالة، ثم تنفذ الشيفرة المزاحة باستخدام القيمة n = 1:

```
    print( "%d x 12 = %d" % (1, 1*12) )
```

ثم تعود إلى سطر for وتضبط n على القيمة التالية في القائمة، وهي 2 في هذه الحالة، ثم تنفذ الشيفرة المزاحة مع القيمة n = 2:

```
    print( "%d x 12 = %d" % (2, 2*12) )
```

وتستمر في تكرار هذا التسلسل حتى تمر n على جميع القيم في القائمة، ثم تنتقل إلى الأمر التالي غير المزاح، في حالتنا لا توجد أي أوامر، لذا سيتوقف البرنامج.

7.1.1 الحلقة التكرارية في VBScript

تُعَد For...Next أبسط حلقة تكرارية في VBScript، وتُستخدم بالشكل التالي:

```
<script type="text/vbscript">
For N = 1 To 12
    MsgBox N & " x 12 = " & N*12
Next
</script>
```

يُعَد أسلوب VBScript أوضح وأسهل في فهم ما تفعله الشيفرة، إذ تتغير قيمة N من 1 إلى 12، وتنفَّذ الشيفرة التي توجد قبل الكلمة المفتاحية Next، وفي حالتنا تطبع الشيفرة النتيجة في صندوق حواري؛ أما إزاحة السطر هنا فاختيارية وليست شرطًا، لكنها تجعل الشيفرة أسهل في القراءة، وقد يحتوي متن الحلقة التكرارية على أكثر من سطر ينفَّذ على عناصر القائمة، كما في حالة بايثون التي رأيناها قبل قليل، ورغم أن VBScript تبدو أوضح للوهلة الأولى؛ إلا أن بايثون أكثر مرونةً كما سنرى بعد قليل.

7.1.2 الحلقة التكرارية في جافاسكربت

تستخدم جافاسكربت البُنية for الشائعة في كثير من لغات البرمجة الشبيهة بلغة C، وستبدو الحلقة كما يلي:

```
<script type="text/javascript">
for (n=1; n <= 12; n++){
    document.write(n + " x 12 = " + n*12 + "<BR>");
    };
</script>
```

تبدو هذه الشيفرة معقدةً للوهلة الأولى، وهي تتكون من ثلاثة أجزاء بين القوسين ( ) هي:

- جزء البداية: n = 1 الذي يُنفَّذ مرةً واحدةً قبل أي شيء آخر.

- جزء الاختبار: `n <= 12` الذي يُنفَّذ قبل كل تكرار.
- جزء الزيادة: `n++` وهو اختصار "زِد n بمقدار 1"، وينفَّذ بعد كل تكرار.

لاحظ أن جافاسكربت تضع الشيفرة المكرَّرة، أي متن الحلقة التكرارية، بين قوسين معقوصين `{}`، وهذا كافٍ لتكون الحلقة صالحةً للتنفيذ، إلا أنه يفضَّل إزاحة الشيفرة التي داخل الأقواس المعقوصة لتحسين قابلية قراءة الشيفرة.

لن ينفَّذ متن الحلقة loop body إلا إذا تحقق جزء الاختبار، أي كان true. وقد تحتوي هذه الأجزاء على شيفرات عشوائية، إلا أن الجزء الثاني الخاص بالاختبار يجب أن يعطي قيمةً بوليانيةً.

7.1.3 مزيد من المعلومات حول حلقة for في بايثون

تتكرّر حلقة `for` في بايثون على تسلسل، تذكر أن التسلسل الذي شرحناه في الفصول السابقة يشمل أشياء، مثل السلاسل النصية والقوائم والصفوف tuples، كما تستطيع بايثون أن تكرِّر عدة أنواع أخرى لكننا سنؤجل ذلك إلى وقت لاحق، وبناءً على ذلك نستطيع كتابة حلقات `for` تتعامل مع أي نوع من التسلسلات، ولنجرب مثلًا طباعة حروف كلمة حرفًا حرفًا باستخدام حلقة `for` مع سلسلة نصية:

```
>>> for c in 'word': print( c )
...
w
o
r
d
```

لاحظ أننا طبعنا كل حرف على سطر منفصل، وأنه يمكننا إضافة متن الحلقة، عندما يكون سطرًا واحدًا، إلى نفس سطر الحلقة بوضعه بعد نقطتين رأسيتين `:`، إذ تخبران بايثون بوجود كتلة برمجية تليهما.

ويمكن التكرار على صف tuple:

```
>>> for word in ('one','word', 'after', 'another'): print (word)
...
```

جعلنا في هذا المثال كل كلمة في سطر منفصل على خلاف المثال السابق، ويمكن وضعها جميعًا في سطر واحد باستخدام ميزة خاصة بدالة `print()`، إذ نستطيع إضافة وسيط argument بعد العنصر المراد طباعته كما يلي:

```
>>> for word in ('one', 'word', 'after', 'another'): print( word,
end='' )
```

```
...
```

لاحظ كيف ستظهر الكلمات الآن في سطر واحد، لأن الوسيط `end=''` يخبر بايثون أن تستخدم سلسلةً نصيةً فارغةً `''` لنهاية السطر عوضًا عن محرف السطر الجديد الذي تستخدمه افتراضيًا، لنجرب حلقة `for` مرةً أخرى مع القوائم:

```
>>> for item in ['one', 2, 'three']: print( item )
...
```

ستظهر هنا مشكلة عند استخدام الحلقات التكرارية التي على نمط `foreach`، وهي أن الحلقة تعطينا نسخةً مما كان في التجميعة، دون أن نستطيع تعديل محتوياتها مباشرةً، فإذا احتجنا إلى تعديلها؛ فعلينا استخدام برنامج غريب الشكل نستخدم فيه فهرس التجميعة، كما يلي:

```
myList = [1,2,3,4]
for index in range(len(myList)):
    print( myList[index] )
    myList[index] += 1
print( myList )
```

سيزيد هذا البرنامج كل فهرس في `myList`، ولو أننا لم نستخدم طريقة الفهرس تلك، لكنا زدنا العناصر المنسوخة فقط دون تغيير القائمة الأصلية، وقد صار لدينا متن متعدد الأسطر في حلقتنا التكرارية.

7.1.4 دالة التعداد enumerate function

إن المثال الذي أوردناه أعلاه شائع جدًا لدرجة أن بايثون توفر دالةً سهلة الاستخدام لتجنب هذه المشكلة، أو تخفيفها على الأقل، وهي دالة `enumerate` التي تعيد عنصرين في كل مرة نستخدمها على تجميعة ما، هذان العنصران هما قيمة التجميعة ورقم فهرسها، مما يسمح لنا بالوصول إلى القيمة كما فعلنا في حلقة `for` الأصلية، وتعديلها بواسطة الفهرس كما فعلنا في المثال أعلاه، وتبدو دالة `enumerate` كما يلي:

```
myList = [1,2,3,4]
for index, value in enumerate(myList):
    print( value )
    myList[index] += 1
print( myList )
```

لاحظ أننا لم نستخدم محث بايثون التفاعلي `>>>` في هذا المثال، لذا عليك كتابته في ملف كما شرحنا في الفصل السادس: المزيد من التسلسلات وأمور أخرى؛ أما إذا حاولنا كتابته في محث بايثون، فسنحتاج إلى أسطر فارغة إضافية لنخبر بايثون بانتهاء كتلة برمجية، وذلك بعد السطر `myList[index] += 1` مثلًا، وهذه

الطريقة مفيدة لتعلّم بداية الكتل البرمجية ونهايتها، بكتابة الشيفرة وتخمين أماكن الحاجة إلى السطر الإضافي، إذ يجب أن يكون عند الموضع الذي تتغير فيه الإزاحة.

لا بد من ملاحظة أمر آخر في حلقات `foreach`، وهو أننا لا نستطيع حذف العناصر من التجميعة التي نمر عليها، لأن ذلك يربك الحلقة نفسها، فهو يشبه قطع فرع من شجرة أثناء جلوسك عليه، والحل هو استخدام نوع مختلف من الحلقات التكرارية، كما سنرى في فصل تالٍ، إذ سنشرح كيفية حذف العناصر دون تعطيل الشيفرة في الفصل العاشر: مقدمة في البرمجة الشرطية.

ربما تجب الإشارة إلى أن كلًا من جافاسكربت وVBScript لديهما بنىً للتكرار على العناصر الموجودة في تجميعة ما، ولن نتحدث عنها بتفصيل هنا، لكن إذا أردت أن تبحث في الأمر بنفسك فانظر صفحات المساعدة لبنية `for each...in` في لغة VBScript، وبنية `for...in` في جافاسكربت.

7.2 حلقات While التكرارية

تتطلب حلقات `for` أن يعلم المبرمج عدد مرات التكرار التي يريد تنفيذها قبل بدء الحلقة نفسها، أو أن يستطيع حسابها على الأقل، لكن ماذا لو أردنا تنفيذ مهمة ما حتى وقوع حدث معين لا نعرف موعده؟ فقد نرغب في قراءة بيانات من المستخدم ومعالجتها مثلًا إلى أن يخبرنا المستخدم نفسه بالتوقف عن ذلك، كما لن نعرف عدد عناصر البيانات التي سنعالجها إلى أن يكتفي المستخدم ويخبرنا بذلك أيضًا. يُعَد تنفيذ مثل هذه المهام باستخدام حلقة `for` ممكنًا نظريًا لكنه معقد، لهذا لدينا نوع آخر من الحلقات التي تحل مثل هذه المشاكل، وهي حلقة `while`، وستبدو في بايثون بالشكل التالي:

```
>>> j = 1
>>> while j <= 12:
...     print( "%d x 12 = %d" % (j, j*12) )
...     j = j + 1
```

لنحلل هذه الشيفرة:

1. نهيئ أولًا `j` لتأخذ القيمة `1`، وهذه الخطوة مهمة للغاية، أي تهيئة متغير التحكم في حلقة `while`، لأن نسيان تهيئته يتسبب في أخطاء كثيرة.
2. ننفذ تعليمة `while` نفسها، التي تختبر تعبيرًا بوليانيًا هو `j<=12` في حالتنا.
3. ننفذ الكتلة المزاحة التي تتبعها إذا كانت النتيجة `True`، وهو محقق في حالتنا بما أن قيمة `j` أقل من `12`، لذا سننتقل إلى الكتلة.
4. ننفذ تعليمة الطباعة لإخراج أول سطر من جدولنا.

5. يزيد السطر التالي في الكتلة متغير التحكم `j`، وهو السطر الأخير في حالتنا، ليشير إلى نهاية كتلة `while`.
6. نعود إلى تعليمة `while` ونكرر الخطوات من 4 إلى 6 بقيمة `j` الجديدة.
7. نكرر هذا التسلسل إلى أن تصل قيمة `j` إلى `13`.
8. هنا يعيد اختبار `while` القيمة `False`، فنتخطى الكتلة المزاحة إلى السطر الجديد الذي يوازي تعليمة `while` في إزاحتها.
9. يتوقف البرنامج في مثالنا لأنه لا توجد أسطر أخرى.

تعليمة while واضحة وبسيطة كما رأينا، لكننا نريد الإشارة إلى النقطتين الرأسيتين `( : )` اللتين في نهاية سطر تعليمة while -وتعليمة for أيضًا-، إذ تخبر هاتان النقطتان بايثون أن ما يليهما كتلة برمجية أو مجموعة مترابطة من التعليمات، وكل لغة لها طريقتها في إخبار المفسر بجمع عدة أسطر معًا كما سنرى، ففي حالة بايثون مثلًا، سنستخدم النقطتين الرأسيتين وإزاحة الأسطر.

7.2.1 حلقة While في VBScript

فيما يلي حلقة while في لغة VBScript:

```
<script type="text/vbscript">
DIM J
J = 1
Do While J <= 12
    MsgBox J & " x 12 = " & J*12
    J = J + 1
Loop
</script>
```

يعطينا هذا المثال نفس النتيجة التي حصلنا عليها من مثال بايثون السابق، مع ملاحظة انتهاء كتلة الحلقة التكرارية بالكلمة المفتاحية `Loop`. هذه البنية التي تستخدمها VBSCript، أي `Do...Loop` هي بنية متعددة الاستخدامات، ومنها نستطيع تشكيل عدة بنىً مختلفة تعطينا تأثيرًا مختلفًا في كل مرة، ونختار ما يناسبنا وفق حالة التطبيق الذي نعمل عليه، وهذه البنى هي:

- `Do...Loop`: وهي البنية الأبسط التي تكرر إلى ما لا نهاية، لكن يمكن الخروج منها بطريقة سنراها في الفصل العاشر.
- `Do While ...Loop`: وهي البنية التي استخدمناها في المثال أعلاه.

- `Do Until ...Loop`: عكس منطق البنية السابقة.
- `Do...Loop While`: تشبه حلقة while لكنها تنفَّذ مرةً واحدةً على الأقل، لأنها تختبر الشرط في نهاية متن الحلقة.
- `Do...Loop Until`: تشبه البنية السابقة لكن مع عكس منطق الاختبار.

7.2.2 حلقة While في جافاسكربت

```
<script type="text/javascript">
j = 1;
while (j <= 12){
   document.write(j + " x 12 = "+ j*12 + "<BR>");
   j++;
   }
</script>
```

تتشابه هذه البنية مع ما سبق، لكن مع بعض الأقواس المعقوصة بدلًا من كلمة `Loop` التي في VBScript، لاحظ أن `J++` تعني زيادة قيمة `j` بمقدار 1، كما ذكرنا في فصل سابق، وأن جافاسكربت وVBScript لا تشترطان إزاحة الأسطر، على عكس لغة بايثون، لكننا أزحنا الأسطر لتسهيل قراءة الشيفرة فقط.

لنراجع الآن حلقة `for` في جافاسكربت:

```
for (j=1; j<=12; j++){....}
```

نلاحظ أنها تشبه حلقة `while` تمامًا، لكنها مكتوبة في سطر واحد بحيث يمكن رؤية المهيئ initializer وشرط الاختبار ومعدِّل الحلقة معًا، وبهذا يتبين أن حلقة `for` في جافاسكربت ما هي إلا حلقة `while` ولكن بصورة مضغوطة، وسنستنتج أن بعض اللغات قد تستغني عن حلقة `for` نهائيًا، وهو ما يحدث حقًا.

7.3 حلقات تكرارية مرنة

إذا عدنا إلى مثال جدول ضرب العدد 12 الذي ذكرناه في بداية هذا الفصل، فسنجد أن الحلقة التي أنشأناها ستطبع جدول هذا العدد بكفاءة، لكن هل يمكننا تعديل الحلقة لجعلها تطبع جدول العدد 7 مثلًا؟ ينبغي أن تبدو الحلقة كما يلي:

```
>>> for j in range(1,13):
...     print( "%d x 7 = %d" % (j,j*7) )
```

وهذا يعني أن علينا تغيير 12 إلى 7 مرتين، وإذا أردنا قيمةً أخرى غيرهما فسنغير في موضعين من جديد، ولكن ألا توجد طريقة أفضل لإدخال مضاعف الضرب ذاك؟، يمكن ذلك باستخدام متغير آخر للقيم التي في سلسلة الطباعة، ثم ضبط ذلك المتغير قبل تشغيل الحلقة التكرارية:

```
>>> multiplier = 12
>>> for j in range(1,13):
...     print( "%d x %d = %d" % (j, multiplier, j*multiplier) )
```

وهذا جدول العدد 12 السابق، وإذا أردنا تغييره إلى العدد 7 مثلًا، فلا نغير إلا قيمة `multiplier`، جرب كتابة هذا البرنامج في ملف سكربت بايثون وتشغيله من محث سطر الأوامر، ثم عدِّل قيمة المضاعف `multiplier` إلى أعداد أخرى، لاحظ أننا جمعنا هنا بين التسلسل والحلقات التكرارية، فقد كتبنا أمرًا وحيدًا في البداية هو `multiplier = 12`، متبوعًا بحلقة `for`.

7.4 تكرار الحلقة نفسها

لنطور مثالنا السابق قليلًا، ولنفترض أننا نريد طباعة جميع جداول الضرب للأعداد من 2 حتى 12، سيكون كل ما نحتاجه هو ضبط متغير المضاعف ليكون جزءًا من الحلقة التكرارية، ونفعل بذلك بالشكل التالي:

```
>>> for multiplier in range(2,13):
...     for j in range(1,13):
...         print( "%d x %d = %d" % (j,multiplier,j*multiplier) )
```

لاحظ أن الجزء المزاح داخل حلقة `for` الأولى هو نفس الحلقة التي بدأنا بها، وستنفَّذ الحلقة كما يلي:

1. نضبط multiplier أولًا على القيمة الأولى `2` ثم ننتقل إلى الحلقة الثانية الداخلية.
2. نعيد ضبط multiplier على القيمة التالية `3` ثم ننتقل إلى الحلقة الداخلية مرةً أخرى.
3. نكرر هذا حتى نمر على جميع الأعداد.

يُعرف هذا الأسلوب باسم الحلقات المتشعبة nesting loops، لكن من مساوئه أن جميع الجداول ستكون مدمجةً معًا، ونستطيع إصلاح هذه المشكلة بطباعة سطر فاصل في نهاية الحلقة الأولى، كما يلي:

```
>>> for multiplier in range(2,13):
...     for j in range(1,13):
...         print( "%d x %d = %d" % (j,multiplier,j*multiplier) )
...     print( "------------------- " )
```

لاحظ أن تعليمة الطباعة الثانية لها نفس إزاحة تعليمة `for` الثانية، وهي ثاني تعليمة في تسلسل التكرار، فتسلسل الإزاحة مهم جدًا في بايثون كما ذكرنا.

دعنا نرى الآن كيفية تنفيذ هذا التدريب في جافاسكربت، لرؤية الفرق بينهما فقط:

```
<script type="text/javascript">
for (multiplier=2; multiplier < 13; multiplier++){
    for (j=1; j <= 12 ; j++){
        document.write(j, " x ", multiplier, " = ", j*multiplier,
"<BR>");
        }
    document.write("---------------<BR>");
    }
</script>
```

حاول أن تجعل السطر الفاصل يشير إلى الجدول التابع له، يمكنك استخدام متغير المضاعف وسلسلة التنسيق التي في بايثون لفعل ذلك.

7.5 حلقات أخرى

توفر بعض اللغات الأخرى بنىً مختلفةً للتكرار، كما رأينا في مثال VBScript أعلاه، غير أنها لا تخلو من صورة ما لحلقتي `for` و`while`، ولا تحوي بعض اللغات، مثل Modula 2 وOberon، إلا حلقة while فقط، بما أننا نستطيع تمثيل سلوك for منها كما رأينا قبل قليل.

ومن الحلقات التكرارية الموجودة في اللغات الأخرى:

- `do-while`: هذه الحلقة هي نفسها حلقة while لكن مع وجود الاختبار في نهايتها، بحيث تُنفَّذ الحلقة مرةً واحدةً على الأقل.
- `repeat-until`: شبيهة بالسابقة أيضًا لكن مع عكس المنطق.
- `GOTO` و `JUMP` و `LOOP`: هذه الحلقات التكرارية موجودة في اللغات القديمة، وهي تعين علامةً في الشيفرة ثم تقفز إليها مباشرةً.

7.6 خاتمة

لقد رأينا أن حلقات `for` تكرر مجموعةً من الأوامر لعدد محدد وثابت من المرات، وأن `while` تكرر تلك الأوامر إلى أن يتحقق شرط محدد في الحلقة، ولا تنفذ متن الحلقة أصلًا إذا كان شرط إنهاء الحلقة غير متحقق من البداية، أي أعطى اختباره النتيجة `false`، وعلمنا أنه توجد أنواع أخرى من الحلقات التكرارية؛ لكن اللغة التي تحتوي عليها، ستحتوي على حلقتي for وwhile أو إحداهما على الأقل، وأن حلقات for في بايثون ما هي إلا حلقات foreach حقيقةً، إذ تعمل على قائمة من العناصر، كما تعلمنا الحلقات التكرارية المتشعبة، بإدخال حلقة فرعية داخل أخرى أكبر منها.

8. أسلوب كتابة الشيفرات البرمجية وتحقيق سهولة قراءتها

من الضروري أن تكـون الـبرامج الـتي نكتبهـا سـهلة القـراءة بمجـرد النظـر إليهـا، لأن أخلاط الرمـوز الغريبـة المتجاورة والأقواس المتتالية والكلمات المبتورة من اللغـة الإنجليزيـة؛ تكفي ليظن النـاظر إلى الشـيفرة أن طفلًا صغيرًا عبث بيده على لوحة المفاتيح، فإذا اخترنا أسماءً مفهومةً للمتغـيرات تـدل على وظائفهـا، ورتبنـا الأسـطر فجعلنا كل مجموعة منها تنفذ مهمةً ما بإزاحة خاصة بها، وكتبنا تعليقات توضح سـبب اختيـار الأرقـام أو الرمـوز التي لا يبدو لها سبب منطقي، وقبل كل ذلك افتتحنا الملف ببيانـات تعريفيـة لـه نعـرف بهـا من كتبـه وإصـدار البرنامج وغير ذلك، وإذا اتبعنا مثل هذه السلوكيات -وهي المتبعة في البرمجة بالفعل-؛ فسنوفر كثيرًا من الجهـد والوقت اللذين سيهدرَان في معرفة غرض كل سطر من البرنامج ومشـاكله المسـتقبلية، ويسـهل تحديثـه لاحقًـا وإضافة ميزات جديدة للبرنامج.

8.1 التعليقات

تحـدثنا عن التعليقـات في الفصـل السـادس، وسـنلقي في هـذا الفصـل الضـوء على بعض تطبيقاتهـا واستخداماتها.

8.1.1 سجل الإصدارات

يجب على المبرمج أن يسهّل على من يقرأ برامجه معرفة التفاصيل الأساسية لكل ملف، مثل تـاريخ إنشـائه واسم مؤلفه وبيانات آخر تعديل فيه وإصداره ووصف عام لمحتوياته، وبقيـة المعلومـات الـتي تـدخل في سـجل تغييراته، وهذه الكتلة النصية تُكتب في تعليق، بالشكل التالي:

```
##############################
# Module:   Spam.py
```

```
# Author:   A.J.Gauld
# Version:  Draft 0.4
'''
This module provides a Spam class which can be
combined with any other type of Food object to create
interesting meal combinations.
'''
################################
# Log:
# 2015/09/01    AJG - File created
# 2015/09/02    AJG - Fixed bug in pricing strategy
# 2015/09/02    AJG - Did it right this time!
# 2016/01/03    AJG - Added broiling method(cf Change Req #1234)
################################
import sys, string, food
...
```

نرى في هذا المثال ملخصًا بسيطًا لما يحتويه الملف بمجرد فتحه، فنعرف ما الذي تغير فيه من ما قبله، ومن الذي نفذ هذه التغييرات ووقت إجرائها، وتبرز أهمية هذه المعلومات عند العمل على المشروع ضمن فريق، وذلك حين نحتاج إلى معرفة المسؤول عن التغيير الذي بين أيدينا أو تصميم البرنامج الذي تعمل عليه، وقد وضعنا الوصف بين زوج من علامات الاقتباس الثلاثية، وهذا أسلوب تتبعه بايثون يسمى سلسلة التوثيق، ويُستخدم لإضافة هذا التوثيق والوصف إلى الدالة `help()` المضمَّنة فيها، كما سنرى بعد قليل.

من المهم ملاحظة وجود أدوات تعمل مثل مستودعات للشيفرات البرمجية، مثل Git وSubversion وCVS وRCS، إذ تستطيع الاحتفاظ بمعلومات تلقائيًا، مثل اسم المؤلف واسم الملف وتفاصيل سجل الإصدار وننصح بالنظر في خصائص التوثيق في تلك المستودعات وتعلمها، لأنها توفر الكثير من الوقت الضائع في التسجيل اليدوي لهذه التعليقات.

8.1.2 تعطيل الشيفرات الفاسدة

تُستخدم التعليقات لعزل جزء فاسد أو غير صالح من الشيفرة عن باقي البرنامج، كأن يكون لدينا برنامج يقرأ بعض البيانات ويعالجها، ويطبع الخرج ثم يحفظ النتائج في ملف البيانات مرةً أخرى، ويجب منع حفظ البيانات الخاطئة أو الفاسدة في الملف وإفساده؛ إذا كانت النتائج على غير ما نريد أو نتوقع. وقد يقال أن الحل الأبسط هو حذف الشيفرة الفاسدة، وهذا صحيح؛ لكننا نستطيع عزلها وتعطيل تنفيذها بتحويلها إلى تعليق، لإصلاحها فيما بعد، وذلك كما يلي:

```
data = readData(datafile)
for item in data:
    results.append(calculateResult(item))
printResults(results)
######################
# تُعزل الشيفرة أدناه إلى حين إصلاح
# calculateResult الزلة التي في
# for item in results:
#     dataFile.save(item)
######################
print 'Program terminated'
```

ثم نحذف علامات التعليق إذا أصلحنا ذلك الخلل، لجعل الشيفرة نشطةً مرةً أخرى.

تحتوي العديد من المحررات البرمجية مثل IDLE، خاصيةً نستطيع فيها اختيار جزء من الشيفرة وتحويله إلى تعليق تلقائيًا، ثم إلغاء ذلك التحديد عند الحاجة، وسنجد هذه الخاصية في IDLE في القائمة Format ثم الخيار Comment Out Region.

8.1.3 سلاسل التوثيق

تسمح جميع اللغات البرمجية بإنشاء تعليقات لتسجيل ما تفعله الدالة أو الوحدة التي نكتبها، لكن قليلًا منها يزيد على ذلك بأن يسمح لنا بتوثيق الدالة بطريقة تستطيع اللغة أو البيئة نفسها أن تستفيد منها لتوفر لنا مساعدةً تفاعليةً أثناء البرمجة، ومن تلك اللغات: جافا وبايثون وSmalltalk، نستخدم هذه الخاصية في بايثون من خلال علامات الاقتباس الثلاثية التي على نمط """documentation""":

```
class Spam:
    """A meat for combining with other foods

    It can be used with other foods to make interesting meals.
    It comes with lots of nutrients and can be cooked using many
    different techniques"""

    def __init__(self):
       pass   # ie. it does nothing!

help(Spam)
```

لاحظ أننا نستطيع الوصـول إلى سلسـلة التوثيـق باسـتخدام الدالـة help()، وأن سلاسـل التوثيـق تصـلح للوحدات والدوال والأصناف والتوابع، لننظر إلى الشيفرة التالية:

```
>>> import sys
>>> help (sys.exit)
Help on built-in function exit:

exit(...)
    exit([status])

    Exit the interpreter by raising SystemExit(status).
    If the status is omitted or None, it defaults to zero (i.e.,
success).
    If the status is numeric, it will be used as the system exit
status.
    If it is another kind of object, it will be printed and the system
    exit status will be one (i.e., failure).
(END)
```

إذا أردنا الخروج من وضع المساعدة الذي في المثال السابق عند رؤية END؛ التي تشـير إلى نهايـة التوثيـق، فسنضغط على مفتاح Q في لوحة المفاتيح، وهـو اختصـار كلمـة Quit؛ أمـا إذا كـان التوثيـق أكـثر من صـفحة، فسنضغط على مفتاح المسافة لتصفحه، ومن الجدير بالذكر أن كلمة END التي في نهاية التوثيق قد لا تظهر في بعض الطرفيات.

توجد دالة مساعِدة أخرى هي dir()، والتي تعرض جميع المزايا التي تعرفها بايثون عن كائن ما، فإذا أردنـا معرفة الدوال أو المتغيرات الموجودة في وحدة sys مثلًا، فسنكتب ما يلي:

```
>>> import sys
>>> dir(sys)
[..... 'argv', 'builtin_module_names', 'byteorder', .... 'copyright',
.... 'exit', ..... 'stderr', 'stdin', 'stdout', 'subversion',
'version', 'version_info', 'warnoptions', 'winver']
```

بعد ذلك سنختار ما نريد منها، ونستخدم دالـة help() للحصـول على مزيـد من المعلومـات، وقـد أهملنـا العديد من المدخلات في المثال أعلاه لتوفير المسـاحة، كمـا نسـتفيد من وظيفـة الدالـة dir() عنـد اسـتخدام وحدة ليس لها توثيق جيد أو كافٍ، فقد نتعامل مع وحدات ليس لها توثيق خارجي أصلًا.

8.2 إزاحة الكتل

ذكرنا عدة مرات أن إزاحة الأسطر مهمة في بايثون، وتتوقف عليها الإجراءات التالية للأسطر المزاحة، مـع أن أكثر لغات البرمجة لا تهتم بإزاحة الأسطر، لذا يزيح المبرمجون الأسطر في برامجهم وشـيفراتهم لتسـهيل قراءتهـا وتمييز أجزائها لا أكثر، فقد صار لكل مبرمج تقريبًا أسلوبه الخاص في الإزاحة وفق ما يراه مناسبًا، وقد ولّدت كثرة الأهواء والآراء في هذا الشأن جدالات حامية الوطيس في البرمجة منذ زمن، بلغت حد إجـراء بدراسـات للوقـوف على الجوانب التي تؤثر فيها إزاحة الأسطر سوى المظهر الجمالي، كأن تساعد في فهم الشيفرة البرمجية بطريقـة أسر ع وأسهل، لننظر في المثال التالي من VBScript مثلًا:

```
<script type="text/vbscript">
For I = 1 TO 10
    MsgBox I
Next
</script>
```

ثم لننظر إلى النسخة التالية منه، والتي لا تحوي إزاحات للأسطر:

```
<script type="text/vbscript">
For I = 1 TO 10
MsgBox I
Next
</script>
```

يكاد المثالان يتطابقان في نظرنا نحن البشر، والفـرق بينهمـا هـو أن النسـخة الأولى أسـهل في القـراءة في نظرنا، أما في نظر مفسر VBScript فهما متطابقتان تمامًا ولا فرق بينهما، والشـرط الحـاكم للإزاحـة هنـا هـو أن تعكس الهيكل المنطقي للشيفرة، فتتبع تدفق البرنامج في هيئتها المرئية، وقد أظهرت الدراسات الـتي أجـريت على الشيفرات المزاحة أن الشيفرات المقسَّمة بصريًا إلى كتل تعكس البنية الكتلية المنطقيـة للبرنـامج؛ أسـهل في الفهم، وعلى ذلك تكون البنية التالية:

```
XXXXXXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXX
```

أسهل في فهم مرادها ووظيفتها من هذه البنية:

```
XXXXXXXXXXXXXXXXXXXX
  XXXXX
    XXXXXXXXXXXX
    XXXXXXXXXXXX
    XXXXXXXXXXXX
  XXXXX
```

قد لا تظهر أهمية الإزاحة في الأمثلة البسيطة التي أوردناها إلى الآن، لكن الفرق يتضح عند التعامل مع برامج تحتوي على مئات الآلاف من الأسطر البرمجية.

يتبقى الآن أن ننظر في مقدار تلك الإزاحة، وهو أمر طال الخلاف حوله في مجال البرمجة، ويكمن الاختلاف بين استخدام المسافات spacebar للإزاحة أو استخدام زر الجدول tab. ومع أن استخدام زر الجدول أسهل لأنه يؤدي الغرض بنقرة واحدة؛ غير أن هذه النقرة الواحدة تساوي 8 مسافات، وهي إزاحة كبيرة، كما يصعب تحديد آلية الإزاحة الموجودة في شيفرة ما، فهل نتجت عن استخدام زر المسافة أم زر الجدول، لأننا لا نستطيع استخدامهما معًا، فإما هذا أو ذاك، وتظهر هذه المشكلة لمن يتعامل مع برنامج كتبه مبرمج غيره، ولتجنب هذه المعضلة من بداية تعلمنا للبرمجة، يُفضّل تجنب استخدام زر الجدول في الإزاحة، لأن بعض الدراسات أظهرت أن أفضل صورة للشيفرة -من حيث سهولة القراءة- تتحقق إذا كانت الإزاحة محصورةً بين مسافتين وخمس مسافات؛ أما مستخدمو بايثون فقد اصطلحوا على استخدام أربع مسافات في الإزاحة، على الرغم من صلاحية الأمثلة التي استخدمناها إلى الآن، والتي كانت إزاحاتها أقل من ذلك، لأنها توفر بعض المساحة على الشاشة.

8.3 أسماء المتغيرات

لم نستخدم أسماءً لها معانٍ للمتغيرات في الأمثلة التي أوردناها في ما سبق، لأننا لم نقصد إلا شرح بعض التقنيات البرمجية، لكن يُفضل أن تعكس أسماء المتغيرات ما تمثله في الشيفرة، من حيث الوظيفة التي تنفذها، فقد استخدمنا المتغير `multiplier` في تدريب جدول الضرب مثلًا؛ والذي سبق أن عرضناه مثل متغير يوضح الجدول الذي نريد طباعته. علمًا أن هذه التسمية أفضل من اختيار اسم عشوائي أو رمزي مثل `m` الذي لن يسبب مشاكل في تنفيذ الشيفرة، بل سيريحنا في الكتابة بما أنه حرف واحد، لكنه بلا معنىً، وسيتسبب في مشاكل مستقبلية إذا أردنا معرفة وظيفة هذا المتغير، أو أراد مبرمج آخر ذلك.

يُفضل في تسمية المتغيرات أن تكون مفهومةً على أن تكون سهلة الكتابة، وأفضل حل لأسماء المتغيرات هو أن تكون قصيرةً وذات معنىً في نفس الوقت، فالأسماء الطويلة مربكة وصعبة الكتابة، فمثلًا: نستطيع استخدام `the_table_we_are_printing` عوضًا عن `multiplier`، غير أن هذا الاسم طويل ويتجاوز المعقول في الكتابة البرمجية.

8.4 حفظ البرامج

ذكرنا سابقًا أننا نخسر كل شيء كتبناه في محث بايثون <<< بمجرد الخروج منه، فهو وإن كان ممتازًا في تجربة الأفكار بسرعة؛ إلا أنه غير مفيد على المدى البعيد، إذ نريد أن نكتب البرامج ونشغلها مرةً بعد مرة وقتما نشاء، فهذا هو المنطق، ولفعل ذلك في بايثون؛ ننشئ ملفًا نصيًا للبرنامج ونجعله بالامتداد .py، وهو اصطلاح متبع، رغم أننا نستطيع استخدام أي امتداد نريده هنا، لكن يُفضل اتباع هذا الاصطلاح، ثم نستطيع تشغيل البرنامج من محث سطر أوامر النظام بكتابة ما يلي:

```
C:\WINDOWS> python spam.py
```

فيكون اسم برنامج بايثون هنا هو spam.py، ومحث نظام التشغيل هو C:\WINDOWS>.

إحدى مزايا استخدام هذه الطريقة في برامج بايثون أننا نستطيع تشغيل البرنامج بمجرد النقر عليه في مدير الملفات، مثل أي ملف تنفيذي يمثل برنامجًا بما أن ويندوز يربط امتداد .py بمفسر بايثون، كما أن استخدام الملفات لتخزين البرامج يسمح بتصحيح أخطائها دون الحاجة إلى إعادة كتابة الجزء الخاطئ من البرنامج مرةً أخرى في محث بايثون، أو التحرك بالمؤشر في IDLE حتى نتجاوز الخطأ كي نعيد تحديد الشيفرة، لأن IDLE تدعم فتح الملف لتعديله وتشغيله من القائمة Run، ثم خيار Run module، أو باستخدام مفتاح F5 على لوحة المفاتيح.

لن نعرض محث بايثون <<< مرةً أخرى في الأمثلة البرمجية التي نشرحها، إذ سنفترض أنك تنشئ البرامج في ملف منفصل وتشغلها إما داخل IDLE أو من محث سطر الأوامر.

8.4.1 ملاحظة لمستخدمي ويندوز

يمكن إعداد ربط الملفات التي تنتهي بامتداد .py داخل مدير الملفات، مما يسمح بتشغيل برامج بايثون بالنقر المزدوج على أيقونة الملف، والمفترض أن مثبت بايثون قد نفذ هذا الإجراء بالفعل، وللتحقق من ذلك؛ ابحث عن ملف بامتداد بايثون وجرب تشغيله، فإذا نجحت يكون الربط جاهزًا على حاسوبك، حتى لو أخرج لك رسالة خطأ.

المشكلة التي قد تواجهها هنا هي أن الملفات التي أيقونتها شعار بايثون نفسه؛ إذ ستشَغل في صندوق DOS ثم تغلَق بسرعة بحيث لا تكاد تراها، ويمكن حل هذا بكتابة السطر التالي في نهاية البرنامج:

```
input("Hit ENTER to quit")
```

سيعرض هذا السطر رسالةً تخبرك أن تضغط زر ENTER للخروج، وينتظر حتى يحدث ذلك؛ أما في إصدارات بايثون الأخيرة فهناك أداة إضافية في ويندوز ستساعدك على تشغيل بايثون دون مشاكل، وهي py.exe، فإذا كتبنا:

```
C:\WINDOWS> py spam.py
```

فستبحث هذه الأداة عن بايثون، وتشغل الشيفرة تشغيلًا صحيحًا، كما تستفيد من أي سطر كامل المسار shebang line داخل الملف لتعرف أي إصدار تستخدمه من بايثون -إذا وُجد أكثر من إصدار مثبت على حاسوبك-.

8.4.2 ملاحظة لمستخدمي يونكس

يجب أن يحتوي أول سطر في ملف سكربت بايثون على التسلسل `#!` متبوعًا بالمسار الكامل لبايثون على حاسوبك، ويُعرف سطر المسار الكامل هذا أحيانًا باسم شيبانك shebang باللغة الإنجليزية، وهو السطر الذي يجعل شيفرة بايثون قابلةً للتنفيذ عبر مفسر بايثون المحدد بالمسار، والذي نعثر عليه بكتابة السطر التالي في محث الصدفة shell prompt:

```
$ which python
```

وقد يبدو ذلك المسار كما يلي:

```
#!/usr/local/bin/python
```

مما يسمح لنا بتشغيل الملف دون استدعاء بايثون، بعد إعدادها لتكون قابلةً للتشغيل من خلال chmod كما تعلم:

```
$ spam.py
```

يمكن استخدام `/usr/bin/env python` بدل معلومات المسار، مثل أسلوب أفضل يمكن تنفيذه على أغلب أنظمة يونكس، بما فيها جميع توزيعات لينكس:

```
#!/usr/bin/env python
```

سيبحث هذا الأمر عن بايثون في مسارك تلقائيًا، غير أن عيبه يظهر عندما توجد عدة إصدارات من بايثون، ولا يعمل السكربت إلا مع إصدار واحد منها فقط عند استخدام ميزة جديدة في بايثون مثلًا، وهنا يُفضَّل استخدام أسلوب المسار الكامل، أو على الأقل تحديد إصدار بايثون المناسب، سواء كان `python2` أو `python3`.

أما سطر `#!` فلا يسبب مشاكل في نظامي ويندوز أو ماك لأنه سيبدو تعليقًا، لذا لا بأس بكتابته عند استخدام هذين النظامين، وذلك تحسبًا لاحتمال تشغيل الشيفرة على حاسوب يونكس يومًا ما، أو استخدام مطلِق `py.exe` الذي لا يتجاهل سطر المسار الكامل shebang.

8.4.3 جافاسكربت وVBScript

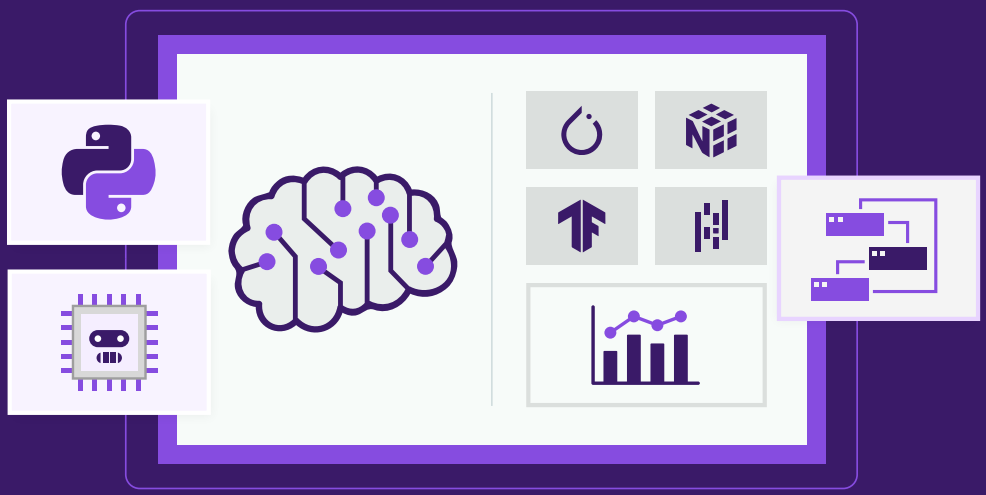
لا يحتاج من يبرمج مستخدمًا جافاسكربت وVBScript إلى الملاحظات التي ذكرناها أعلاه، لأن هـذه الـبرامج تُحفظ في ملفات.

8.5 خاتمة

تعلمنا في هذا الفصل ما يلي:

- استخدام التعليقـات لعـزل جـزء من الشـيفرة مؤقتًـا عنـد الاختبـارات أو تنقيح الشـيفرة debugging، وكذلك استخدامها لتوفير ترويسة توضيحية يُذكر فيها سجل إصدارات الملف.
- كيفية استخدام سلاسـل التوثيـق في توفـير معلومـات فوريـة runtime-information عن وحـدة مـا، والكائنات التي داخلها.
- فائدة إزاحة كتل الشيفرات في توضيح الترتيب المنطقي لهيكل الشيفرة البرمجية، وتسـهيل تتبـع عين القارئ لذلك التسلسل المنطقي.
- حفظ برامج بايثون في ملفات لإعادة تشغيلها في أي وقت، بدلًا من كتابتهـا في محث بـايثون `>>>` ثم ضياعها عند الخروج منه، وذلـك بكتابـة `$ python progname.py` في سـطر الأوامـر، أو بتشـغيل الملف في مدير الملفات بالنقر عليه.

دورة الذكاء الاصطناعي



تعلم الذكاء الاصطناعي وتعلم الآلة والتعلم العميق
وتحليل البيانات، وأضفها إلى تطبيقاتك

التحق بالدورة الآن

9. قراءة المدخلات من المستخدم

تعاملت البرامج التي كتبناها في الفصول السابقة مع بيانات ثابتة، نستطيع فحصها -عند الحاجة- قبل أن يستخدمها البرنامج، ونصمم البرنامج ليناسب تلك البيانات، لكن الواقع يقول إن أغلب البرامج تتصرف وفقًا لمدخلات المستخدم، حيث يخبر المستخدم البرنامج بالملف التي يجب عليه فتحه أو تعديله مثلًا، وقد يطلب البرنامج من المستخدم بيانات ضروريةً، ويشار إلى مثل هذا المنظور البرمجي باسم واجهة المستخدم، وتصميم وبناء هذه الواجهة في السوق البرمجي وظيفة متخصصين في التفاعل بين الآلة والبشر، وفي كيفية تصميم بيئات العمل؛ أما المبرمج العادي فليس لديه تلك الرفاهية، فهو يتصرف بما لديه من منطق، ويفكر مليًا في كيفية تفاعل المستخدمين مع برنامجه أثناء استخدامه. وأبسط ميزة لواجهة المستخدم هي عرض المخرجات، وقد شرحنا ذلك من قبل بطرق بدائية وبسيطة، باستخدام الدالة `print` في بايثون، والدالة `write()` في جافاسكربت، وصندوق `MsgBox` الحواري في VBScript.

تتمثل الخطوة التالية في تصميم واجهة المستخدم في أخذ المدخلات من المستخدم مباشرةً، وأبسط طريقة لفعل ذلك هي جعل البرنامج يطلب الدخل في وقت التشغيل، أو جعل المستخدم يمرر البيانات عندما يشغِّل البرنامج، أو -في الحالة المتقدمة- تكون لدينا واجهة رسومية فيها صناديق إدخال للنصوص مثلًا، وسنشرح في هذا الفصل الطريقتين الأولى والثانية فقط، ونترك الواجهة الرسومية لوقت لاحق في الكتاب لأنها أكثر تعقيدًا، غير أن هناك وحدةً تسمح لنا بتنفيذ صناديق حوارية رسومية بسيطة، وسندرِجها في هذا الفصل أيضًا.

سنرى الآن كيف نستطيع الحصول على البيانات من المستخدم في جلسة بايثون تفاعلية عادية تعمل في IDLE، أو في طرفية نظام التشغيل، ثم نحاول الحصول على نفس البيانات داخل برنامج.

9.1 دخل المستخدم في بايثون

نحصل على ما يدخله المستخدم في بايثون بالشكل التالي:

```
>>>> print( input("Type something: ") )
```

تعـرض `input()` المحث المعطى، وهـو "Type something" في حالتنـا، وتلتقـط أي شـيء يكتبـه المستخدم، ثم تعرض `print()` تلك الإجابة، ونستطيع إسناد ما يدخله المستخدم إلى متغير:

```
>>> resp = input("What's your name? ")
```

يمكننا طباعة أي قيمة يلتقطها ذلك المتغير:

```
>>> print( "Hi " + resp + ", nice to meet you" )
```

لاحظ أننا لم نستخدم هذه المرة عامل تنسيق السلسلة النصية لعـرض القيمـة المخزَّنـة في المتغـير `resp`، واكتفينا بإدراج القيمة بين سلسلتين نصيتين، ودمجنا السلاسل الثلاث باستخدام عامل إضافة السلاسل النصية `+`، وقيمة المتغير `resp` هي التي التُقطت من المستخدم بواسطة `input()`.

لاحظ كيف استخدمنا المسافات داخل السلاسل النصية التي في المحث المعطى إلى `input()` وسلسـلة الخرج، وانظر الجزء الثالث من سلسلة الخرج الذي يبدأ بالفاصلة المتبوعة بمسافة، إذ يخطئ الكثيرون في أماكن تلك المسافات عند إنتاج خرج كهذا، لذا تفحص مثل هذه المواضع جيدًا عند اختبارك للبرامج.

قرأ المثال السابق السلاسل النصـية، لكن مـاذا عن أنـواع البيانـات الأخـرى؟ نجيب على هـذا السـؤال بـأن بايثون تأتي بمجموعة كاملة من دوال تحويل البيانات، التي تستطيع تحويل سلسلة نصية إلى نـوع بيانـات آخـر، مع وجوب أن تكون البيانات التي في السلسلة النصية متوافقةً مع ذلك النوع وإلا فسنحصل على خطأ، ولنعـدل مثلًا جدول الضرب الذي ذكرناه من قبل، ليقرأ قيمة المضاعف من المستخدم:

```
>>> multiplier = input("Which multiplier do you want? Pick a number ")
>>> multiplier = int(multiplier)
>>> for j in range(1,13):
...   print( "%d x %d = %d" % (j, multiplier, j * multiplier) )
```

نقرأ في هذا المثال القيمة من المستخدم، ثم نحولها إلى عدد صحيح باستخدام دالة التحويـل `int()`، كمـا يمكننا تحويلها إلى عدد ذي فاصلة عائمة باستخدام `float()` عنـد الحاجـة إلى ذلـك، ونفـذنا التحويـل هنـا في سطر منفصل لنفصّل الخطوات لتبسيط الشرح، أما في البرمجة في سوق العمـل من الشـائع تغليـف اسـتدعاء `input()` داخل التحويل كما يلي:

```
>>> multiplier = int( input("Which multiplier do you want? Pick a
number ") )
>>> for j in range(1,13):
...   print( "%d x %d = %d" % (j, multiplier, j * multiplier) )
```

غلفنا استدعاء `input()` داخل استدعاء `int()`، لنجرب تطبيق هذا في برنامج حقيقي، وسنستخدم برنامج دليل جهات الاتصال الذي أنشأناه باستخدام قاموس في الفصل الخامس: البيانات وأنواعها، بما أننا نستطيع كتابة الحلقات التكرارية وقراءة بيانات الإدخال:

```
# أنشئ قاموس فارغًا لدليل جهات اتصال
addressBook = {}

# اقرأ المدخلات إلى أن تصل إلى سلسلة فارغة
print()  # print a blank line
name = input("Type the Name(leave blank to finish): ")
while name != "":
   entry = input("Type the Street, Town, Phone.(Leave blank to
finish): ")
   addressBook = entry
   name = input("Type the Name(leave blank to finish): ")

# والآن، اطلب عرض واحد منها
name = input("Which name to display?(leave blank to finish): ")
while name != "":
   print( name, addressBook )
   name = input("Which name to display?(leave blank to finish): ")
```

لعل هذا البرنامج هو أكبر برنامج كتبناه حتى الآن، إذ يشمل تسلسلات وحلقتين تكراريتين، وعلى الرغم من أن تصميم واجهة المستخدم فيه ليس مثاليًا؛ إلا أنه يؤدي الغرض المطلوب منه، وسنرى كيفية تطوير هذه الواجهة في فصول تالية؛ أما الآن فنريد الإشارة إلى استخدام الاختبار البولياني في حلقات `while` لتحديد رغبة المستخدم في التوقف، لاحظ أيضًا أننا مع استخدامنا لقائمة في الفصل الخامس لتخزين البيانات في حقول منفصلة، إلا أننا خزنّاها هنا في سلسلة نصية واحدة، لأننا لم نشرح كيفية تقسيم السلسلة النصية إلى حقول منفصلة بعد.

9.2 الإدخال في لغة VBScript

تقرأ تعليمات InputBox في لغة VBScript دخل المستخدم كما يلي:

```
<script type="text/vbscript">
Dim Input
Input = InputBox("Enter your name")
MsgBox ("You entered: " & Input)
</script>
```

تمثل دالة `InputBox` صندوقًا حواريًا مع محث وحقل إدخال، وتعيد محتويات حقل الإدخال، وهناك عدة قيم يمكن تمريرها إليها، مثل سلسلة عنوان الصندوق الحواري، إضافةً إلى المحث، فإذا ضغط المستخدم زر الإلغاء Cancel، فستعيد الدالة سلسلةً فارغةً بغض النظر عن محتويات حقل الإدخال.

سيبدو مثال دليل الاستخدام في لغة VBScript كما يلي:

```
<script type="text/vbscript">
Dim dict,name,entry  ' Create some variables.
Set dict = CreateObject("Scripting.Dictionary")
name = InputBox("Enter a name", "Address Book Entry")
Do While name <> ""
   entry = InputBox("Enter Details - Street, Town, Phone number",
                    "Address Book Entry")
   dict.Add name, entry ' Add key and details.
   name = InputBox("Enter a name","Address Book Entry")
Loop

' Now read back the values
name = InputBox("Enter a name","Address Book Lookup")
Do While name <> ""
   MsgBox(name & " - " & dict.Item(name))
   name = InputBox("Enter a name","Address Book Lookup")
Loop
</script>
```

لاحظ أن الهيكل الأساسي هنا مطابق تمامًا لبرنامج بايثون، مع أن بعض الأسطر أطول هنا بسبب الحاجة إلى التصريح المسبق عن المتغيرات باستخدام `Dim` في VBScript، وبسبب الحاجة إلى تعليمة `Loop` لإنهاء كل حلقة.

9.3 قراءة المدخلات في جافاسكربت

تمثل جافاسكربت تحديًا في هذا المجال لأنها تعمل أساسًا داخل المتصفحات، ولقراءة المدخلات يمكننا استخدام صندوق إدخال بسيط كما في VBScript باستخدام دالة `prompt()`، أو قراءة المدخلات من عنصر استمارة في HTML، أو استخدام تقنية البرمجة النصية النشطة Active Scripting الخاصة بمايكروسوفت -وذلك في إنترنت إكسبلورر فقط- لتوليد صندوق `InputBox` كما في VBScript، ولتنويع الأمثلة سنشرح كيفية استخدام تقنية عنصر الاستمارة في HTML، فإذا لم تكن تعرضت للغة HTML من قبل فانظر توثيقها في موسوعة حسوب، أو انسخ ما سنكتبه هنا إذا شئت.

```
<form id='entry' name='entry'>
<p>Type a value then click outside the field with your mouse</p>
<input type='text' name='data'
          onChange='alert("We got a value of " +
document.forms["entry"].data.value);'>
</form>
```

تتكون شيفرة HTML أعلاه من عنصر استمارة `form` الذي يحتوي على فقرة `<p>` من سطر واحد، حيث يمثل رسالةً إلى المستخدم، وحقل إدخال نصي `input`، يحتوي على شيفرة من سطر واحد مرتبطة به، تنفَّذ في كل مرة تتغير فيها القيمة المدخلة، ووظيفة هذه الشيفرة ببساطة، أن تخرِج صندوق رسالة `alert` يشبه ذاك الذي في VBScript، ويحتوي على القيمة التي في الحقل النصي.

كما نرى في أول سطر في الشيفرة، فإن للاستمارة خاصيتين هما `id` و `name`، وتحتويان على القيمة `entry`، وتُخزَّن الاستمارات في سياق المستند `document` داخل مصفوفة مفهرسة بالاسم، لأنه يمكن استخدام مصفوفات جافاسكربت مثل القواميس كما ذكرنا من قبل، ويحتوي حقل `input` على الخاصية `name` التي تحمل القيمة `data` داخل سياق الاستمارة، ومع ذلك نستطيع الإشارة إلى قيمة `value` حقل ما داخل برنامج جافاسكربت كما يلي:

```
document.forms["entry"].data.value
```

لن نشرح هنا مثال دليل جهات الاتصال في جافاسكربت، لأن HTML ستتعقَّد وسنحتاج إلى شرح الدوال، لذا سنؤجل هذا إلى فصل تال.

9.4 مجاري الدخل والخرج القياسية

ننظر الآن في أحد المفاهيم الأساسية في حوسبة سطر الأوامر، وهو مفهوم مجاري البيانات data streams، فمصطلح `stdin` هو أحد المصطلحات الحاسوبية التي تشير إلى جهاز الدخل القياسي standard input device، وهو عادةً لوحة المفاتيح، وبالمثل يشير `stdout` إلى جهاز الخرج القياسي، وهو

الشاشة عادةً، وسنرى إشارات كثيرةً إلى هذين المصطلحين عند الحديث عن البرمجة، كما يوجد مصطلح ثالث لا يُستخدم كثيرًا، وهو `stderr` الذي يشير إلى المكان الذي ترسَل إليه جميع أخطاء الطرفية، ويظهر عادةً في نفس مكان stdout، ويطلق على هذه المصطلحات اسم مجاري البيانات، لأن البيانات تظهر في صورة مجارٍ من البايتات التي تتدفق إلى الأجهزة، وقد أُعدَّ كل من stdin وstdout ليشبها الملفات، ليتوافقا مع شيفرة معالجة الملفات.

وتوجد هذه المصطلحات في لغة بايثون داخل الوحدة `sys`، وتسميان `sys.stdin` و`sys.stdout`، وتستخدم الدالة `input()` تلقائيًا stdin، بينما تستخدم `print()` الخرج القياسي stdout.

نستطيع أن نقرأ من الدخل القياسي stdin، وأن نكتب مباشرةً في الخرج القياسي stdout، مما يسمح لنا بتحكم دقيق في الدخل والخرج، لننظر في المثال التالي إلى القراءة من الدخل القياسي:

```
import sys
print( "Type a value: ", end='')  # تمنع السطر الجديد
value = sys.stdin.readline()  # صراحة stdin استخدم
print( value )
```

تطابق الشيفرة أعلاه تقريبًا الشيفرة التالية:

```
print( input("Type a value: ") )
```

إن ميزة النسخة الصريحة هنا أننا نستطيع جعل الدخل القياسي يشير إلى ملف حقيقي، ليقرأ البرنامج الدخل الخاص به من الملف عوضًا عن الطرفية، وهذه الطريقة مفيدة في حالة جلسات الاختبار الطويلة التي يقرأ البرنامج فيها مدخلاته من ملف بدلًا من الجلوس وكتابة الدخل يدويًا كلما طلبه البرنامج، مما يضمن تكرار تشغيل الاختبار مع ثبات الدخل في كل مرة، وكذلك الخرج، وتسمى هذه التقنية -التي تكرر الاختبارات السابقة لضمان عدم تعطل شيء في الشيفرة- باسم الاختبار الارتدادي regression testing.

وأخيرًا لدينا مثال عن خرج مباشر إلى `sys.stdout`، ويمكن إعادة توجيهه إلى ملف كذلك، وتكافئ الدالة `print` ما يلي:

```
sys.stdout.write("Hello world\n") # \n= newline
```

لا شك أننا نستطيع تحقيق نفس النتيجة باستخدام سلاسل التنسيق النصية، إذا كنا نعرف الشكل الذي ستكون البيانات عليه، لكن إذا لم نعرف شكلها إلا وقت التشغيل، فمن الأسهل أن نرسلها إلى الخرج القياسي، بدلًا من محاولة بناء سلسلة تنسيق معقدة وقت التشغيل.

9.4.1 إعادة توجيه الدخل والخرج القياسيين

يمكن توجيه الدخل والخرج القياسيين إلى ملفات من داخل برنامجنا مباشرةً، باستخدام تقنيات بايثون المعتادة للتعامل مع الملفات، والتي سنشرحها فيما يلي، لكن الطريقة الأسهل هي من خلال نظام التشغيل، تتصرف أوامر النظام عند استخدام إعادة التوجيه في سطر الأوامر كما يلي:

```
C:> dir
C:> dir > dir.txt
```

يطبع الأمر الأول قائمةً من المجلدات على الشاشة -والتي تمثل الخرج القياسي-، بينما يطبع الأمر الثاني تلك القائمة إلى ملف، فقد أخبرنا البرنامج أن يوجه الخرج القياسي إلى الملف `dir.txt`، ويمكن فعل نفس الشيء مع برنامج بايثون كما يلي:

```
$ python myprogram.py > result.txt
```

ستشغّل الشيفرة السابقة البرنامج `myprogram.py`، وستكتب الخرج إلى الملف `result.txt` بدلًا من كتابته على الشاشة، ونستطيع رؤية الخرج لاحقًا باستخدام محرر نصي، لاحظ أن محث علامة الدولار `$` هو المحث القياسي لمستخدمي لينكس وماك.

> يمكن إعادة توجيه خرج الدالة `print` باستخدام الوسيط الاختياري `file="result.txt"`، وتتيح هذه الطريقة التحكم في كل تعليمة، لكن يصعب كتابتها في البرنامج، وهي مناسبة لطباعة مجموعة فرعية بعينها من خرج البرنامج، ولا يفضل استخدامها في حفظ الخرج إلى ملف، وحتى في حالة المجموعة الفرعية يفضَّل توجيه الخرج إلى ملف، لذا لا يُستخدم خيار `file=` كثيرًا.

نستخدم علامة `>` بدلًا من `<` لتوجيه الدخل القياسي إلى ملف، ننشئ في المثال التالي ملفًا باسم `echoinput.py` يحتوي على الشيفرة:

```
import sys
inp = input()
while inp != '':
   print( inp )
   inp = input()
```

نستطيع الآن تشغيل الملف من سطر الأوامر كما يلي:

```
$ python echoinput.py
```

يجب أن تكون النتيجة برنامجًا يعيد طباعة أي شيء تكتبه إلى أن تكتب سطرًا فارغًا، والآن أنشئ ملفًا نصيًا بسيطًا باسم `input.txt` يحتوي بعض الأسطر النصية، وشغّل البرنامج الأخير مرةً أخرى معيدًا توجيه الدخل من `input.txt`:

```
$ python echoinput.py < input.txt
```

ستعيد بايثون طباعة الأسطر النصية الموجودة في الملف.

نستطيع اختبار عدة سيناريوهات على برامجنا بسهولة إذا استخدمنا تلك التقنية على عدة ملفات، مثل قيم البيانات المعطوبة أو أنواع البيانات الخاطئة، وننفذ ذلك بأسلوب موثوق قابل للتكرار، كما نستطيع استخدامها لمعالجة البيانات كبيرة الحجم من ملف، مع السماح بالإدخال اليدوي للبيانات صغيرة الحجم باستخدام نفس البرنامج، وهكذا نرى أن الدخل والخرج القياسيين مفيدان جدًا للمبرمجين.

9.5 معاملات سطر الأوامر

لدينا نوع آخر من الإدخال، وهو الإدخال من سطر الأوامر، كما في حالة تشغيل المحرر النصي من سطر أوامر النظام:

```
$ edit Foo.txt
```

يستدعي نظام التشغيل هنا البرنامج الذي يحمل الاسم `edit`، ليمرر إليه اسم الملف الذي نريد تعديله، وهو `Foo.txt` هنا. لكن كيف يقرأ المحرر اسم الملف؟، يوفر نظام التشغيل في أغلب لغات البرمجة مصفوفةً أو قائمةً من سلاسل نصية تحتوي كلمات سطر الأوامر، وبناءً عليه سيحتوي العنصر الأول على الأمر نفسه، ثم يحتوي العنصر التالي على الوسيط الأول وهكذا، وقد توجد بعض المتغيرات السحرية هنا والتي يُطلق عليها argc، وهي اختصار argument count، وتحمل عدد العناصر الموجودة في القائمة، وتحتفظ الوحدة `sys` في بايثون بهذه القائمة وتسميها `argv`، وهي اختصار argument values، وأول عنصر فيها `argv[0]` هو اسم ملف السكربت الذي ينفَّذ، ولا تحتاج بايثون قيمةً من النوع `argc` لأننا نستطيع استخدام التابع `len()` لإيجاد طول السلسلة، بل لا نحتاج إلى ذلك أصلًا في أغلب الحالات، لأننا نمر على القائمة باستخدام حلقة `for` الخاصة ببايثون:

```
import sys
for item in sys.argv:
    print( item )

print( "The first argument was:", sys.argv[1] )
```

لاحظ أن هذه الشيفرة لا تعمل إلا إذا وُضعت في ملف مثل `args.py`، ونُفذت من محث نظـام التشـغيل كما يلي:

```
C:\PYTHON\PROJECTS> python args.py 1 23 fred
args.py
1
23
fred
The first argument was: 1
C:\PYTHON\PROJECTS>
```

يجب إحاطة اسم الوسيط بعلامات اقتباس إذا احتوى على مسافات، كما يلي:

```
C:\PYTHON\PROJECTS> python args.py "Alan Gauld" fred
args.py
Alan Gauld
fred
The first argument was: Alan Gauld
C:\PYTHON\PROJECTS>
```

9.5.1 جافاسكربت VBscriptg

لا نرى مفهوم وسائط سطر الأوامر في هـاتين اللغـتين لأنهمـا موجهتـان للعمـل داخـل المتصـفحات، فـإذا استخدمناهما داخل بيئة Windows Script Host الخاصة بمايكروسوفت، فستوفر بيئة WSH آليةً لاسـتخراج مثل تلك الوسائط من كائن `WshArguments` الذي تملؤه WSH في وقت التشغيل.

9.6 خاتمة

لن نذهب أبعد من هذا الحد في شأن مدخلات المستخدم في هذا الكتاب، وهو الحد الذي يمكننا من كتابـة برامج مفيدة، وقد كان ذلك كل ما يستطيعه المبرمج في الأيام الأولى لأنظمـة يـونكس أو لمـبرمجي الحواسـيب الشخصية، ولا شك أن البرامج الرسومية تقرأ المدخلات من المستخدم، لكن التقنيات التي تسـتخدمها مختلفـة كليًا، لهذا سنؤجلها إلى فصول لاحقة من هذا الكتاب.

تعلمنا في هذا الفصل استخدام الدالة `input()` لقراءة السلاسل النصية، وعرفنا أنها تعرض سلسلةً تطلب إدخالًا من المستخدم، كما تعرفنا إلى الدخل والخرج القياسيين لمجاري البيانات، وكيفية إعادة توجيههمـا ليكونـا في صورة ملفات، وعرفنا أن `input()` تعمل مع stdin، وأن `print()` تعمل مع stdout.

ويمكن الحصول على وسطاء سطر الأوامر من قائمة `argv` المستوردة من وحدة `sys` في بايثون، حيث يكون العنصر الأول هو اسم البرنامج، ورأينا أن جافاسكربت وVBScript تستطيعان قراءة المدخلات من استمارات الويب `forms` أو من خلال الصناديق الحوارية، لكن ليس لهما وصول إلى `stdin`، كما تستطيعان عرض الخرج بكتابته إلى المستند `document` أو من خلال الصناديق الحوارية، وليس لهما وصول إلى `stdout`.

10. مقدمة في البرمجة الشرطية

تصف التعليمات الشرطية في البرمجة قدرتنا على تنفيذ تسلسل واحد من بين عدة تسلسلات محتملة من الشيفرات -التي تمثل الفروع- وفقًا لوقوع حدث ما، وقد كانت أبسط صورة لتلك التعليمات الشرطية في الأيام الأولى لاستخدام لغة التجميع Assembly هي تعليمة `JUMP` التي تجعل البرنامج يقفز إلى عنوان ما في الذاكرة، غالبًا عندما يكون ناتج التعليمة السابقة صفرًا، وقد استُخدمت هذه التعليمة في برامج بالغة التعقيد، دون استخدام أي صورة شرطية أخرى وقتها، ليؤكد هذا قول ديكسترا Dijkstra عن الحد الأدنى المطلوب للبرمجة، ثم ظهرت نسخة جديدة من تعليمة `JUMP` مع تطور لغات البرمجة عالية المستوى، وهي `GOTO`، التي لا تزال متاحةً حتى الآن في لغة QBASIC، والتي يمكن تحميلها واستخدامها من www.qbasic.net، لننظر إلى الشيفرة التالية مثلًا -بعد تثبيت QBASIC-:

```
PRINT "Starting at line 10"
J = 5
IF J < 10 GOTO 50
PRINT "This line is not printed"
STOP
```

لاحظ كيف تحتاج إلى بضع ثوان لتعرف الخطوة التالية رغم صغر البرنامج، إذ لا يوجد هيكل للشيفرة، بل عليك أن تكتشف ما يفعله البرنامج بينما تقرؤه بنفسك، وهذا الأمر مستحيل في البرامج كبيرة الحجم، لهذا لا تحتوي أغلب لغات البرمجة الحديثة بما فيها لغة بايثون وجافاسكربت وVBScript، على تعليمة JUMP أو GOTO بحيث يمكن استخدام أي منهما مباشرةً، وإن وجدت فستحثك اللغة على عدم استخدامها، فما العمل إذًا في حالة البرمجة الشرطية؟.

10.1 تعليمة if الشرطية

إن أول ما يرد إلى الذهن عند التفكير في حالات شرطية هي البنية if..then..else، والتي تعني "إذا حدث س، فسيكون ص، وإلا سيكون ع"، فهي تتبع منطقًا لغويًا بحيث إذا تحقق شرط بولياني ما -أي كان true-، فستُنفَّذ كتلة من التعليمات، وإلا ستنفَّذ كتلة أخرى.

10.1.1 بايثون

إذا أردنا كتابة مثال GOTO السابق بلغة بايثون، فسيبدو كما يلي:

```
import sys  # لنستطيع استخدام exit فقط
print( "Starting here" )
j = 5
if j > 10:
    print( "This is never printed" )
else:
    sys.exit()
```

هذه الصورة أسهل من سابقتها في القراءة وفهم المراد منها، ونستطيع وضع أي شرط اختباري نريده بعد تعليمة if، طالما أنه يقيَّم إلى True أو False، جرب تغيير علامتي > و < وانظر ما سيحدث.

لاحظ النقطتين الرأسيتين : في نهاية سطري التعليمتين if وelse، إذ تخبرنا هاتان النقطتان أن ما يليهما كتلة مستقلة من الشيفرة، بينما تخبرنا إزاحة تلك الكتلة ببدايتها ونهايتها.

10.1.2 VBScript

سيبدو المثال السابق في VBScript كما يلي:

```
<script type="text/vbscript">
MsgBox "Starting Here"
DIM J
J = 5
If J > 10 Then
    MsgBox "This is never printed"
Else
    MsgBox "End of Program"
End If
</script>
```

يكاد يكون هذا المثال مطابقًا لمثال بايثون، مع فرق أن كلمة Then مطلوبة للكتلة الأولى، وأننا نستخدم End If لنعلن نهاية بنية if/then، ولعلك تذكر أننا استخدمنا كلمة Loop لإنهاء حلقة Do While في مثال VBScript من فصل سابق، ذلك أن VBScript تستخدم في اصطلاحها محددًا لنهاية التعليمة.

10.1.3 جافاسكربت

يبدو مثال if في جافاسكربت كما يلي:

```
<script type="text/javascript">
var j;
j = 5;
if ( j > 10 ){
    document.write("This is never printed");
    }
else {
    document.write("End of program");
    }
</script>
```

لاحظ أن جافاسكربت تستخدم أقواسًا معقوصةً لتحديد كتل الشيفرات الموجودة داخل الجزء الخاص بتعليمة if والجزء الخاص بتعليمة else، وأن الاختبار البولياني يوجد بين قوسين، وليس هناك استخدام صريح لكلمة then.

من المنظور الجمالي للتنسيق، يمكن وضع الأقواس المعقوصة في أي مكان، لكننا اخترنا صفها كما في المثال لإبراز بنية الكتلة البرمجية، ويمكن إهمال الأقواس المعقوصة كليًا إذا كان لدينا سطر واحد فقط داخل الكتلة كما في حالتنا هذه، لكن يفضل بعض المبرمجين وضعها للحفاظ على اتساق الكتابة، ولاحتمال إضافة سطر آخر مستقبلًا، فوضعها الآن أفضل من نسيانها لاحقًا.

10.2 التعابير البوليانية

لعلك تذكر أننا ذكرنا في الفصل الخامس: البيانات وأنواعها، نوعًا من البيانات اسمه النوع البولياني، وقلنا إن هذا النوع لا يحوي إلا قيمتين فقط، هما True وFalse، ورغم أننا لا ننشئ متغيرًا بوليانيًا إلا نادرًا، إلا أننا نحتاج إلى إنشاء قيم بوليانية مؤقتة باستخدام التعابير، والتعبير هو تجميعة من المتغيرات و/أو القيم، تجمعها معًا عوامل لإخراج قيمة ناتجة، لننظر إلى المثال التالي:

```
if x < 5:
  print( x )
```

التعبير هنا هو `x < 5`، والنتيجة ستكون `True` إذا كانت x أقل من 5، و`False` إذا كانت x أكبر من (أو تساوي) 5.

قد تكون التعابير معقدةً بما أنها تقيَّم إلى قيمة نهائية وحيدة، إذ يجب أن تكون تلك القيمة `True` أو `False` في حالة الفرع، لكن يختلف تعريف هاتين القيمتين من لغة لأخرى، فأغلب اللغات تساوي بين القيمة `False` وبين الصفر أو القيمة غير الموجودة أو غير المعرفة والتي تدعى `NULL` أو `Nil` أو `None`، وبناءً على ذلك تقيَّم القائمة الفارغة أو السلسلة النصية الفارغة إلى false في السياق البولياني وتعتمد بايثون هذا السلوك، مما يعني أننا نستطيع استخدام حلقة `while` لمعالجة القائمة إلى أن تصبح فارغةً، بالشكل التالي:

```
while aList:
   # افعل شيئًا هنا
```

أو يمكن استخدام تعليمة `if` لننظر هل القائمة فارغة أم لا، دون اللجوء إلى دالة `len()` كما يلي:

```
if aList:
   # افعل شيئًا هنا
```

يمكن جمع عدة تعابير بولبانية باستخدام عوامل بولبانية كذلك، والتي تقلل من عدد تعليمات `if` التي علينا كتابتها، كما في المثال التالي:

```
if value > maximum:
   print( "Value is out of range!" )
else if value < minimum:
   print( "Value is out of range!" )
```

لاحظ أن كتلة الشيفرة التي يجب تنفيذها متطابقة في شرطي المثال أعلاه، ويمكن أن نوفر على أنفسنا بعض الكتابة وعلى الحاسوب بعض العمل، بأن نجمع الاختبارين في اختبار واحد كما يلي:

```
if (value < minimum) or (value > maximum):
   print( "Value is out of range!" )
```

لاحظ كيف جمعنا الاختبارين باستخدام عامل `or` البولياني، وبما أن بايثون تقيِّم مجموعة الاختبارات المجمعة تلك إلى نتيجة واحدة؛ فنستطيع القول أن هذين التعبيرين إنما هما تعبير واحد، ونستطيع استيعاب هذا الأسلوب إذا عرفنا أننا نقيّم المجموعة الأولى من الأقواس أولًا، ثم نقيّم المجموعة الثانية، ثم نجمع القيمتين الناتجتين لنشكل قيمةً نهائيةً وحيدةً تكون إما True أو False. تستخدم بايثون أسلوبًا أكثر كفاءة من هذا يُعرف بالتقييم المقصور أو تقييم الدارة المقصورة short-circuit evaluation، وإذا تأملنا في هذه الاختبارات، فسنجد أننا نفكر فيها بالمنطق اللغوي البشري الذي يستخدم حروف العطف مثل ("و" "and") و("أو" "or") و("النفي" "not")، مما يتيح لنا كتابة اختبار واحد مجمَّع بدلًا من عدة اختبارات منفصلة، كما سنرى فيما يلي.

10.3 تعليمات if المتسلسلة

يمكن سَلسَلة تعليمات "if..then..else" معًا بجعلها متشعبةً، بحيث تتداخل كل واحدة مع الأخـرى، لننظر مثالًا على ذلك في بايثون:

```
# افترض أن السعر قد حُدد مسبقًا...
price = int(input("What price? "))
if price == 100:
    print( "I'll take it!" )
else:
    if price > 500:
        print( "No way!" )
    else:
        if price > 200:
            print( "How about throwing in a free mouse mat?" )
        else:
            print( "price is an unexpected value!" )
```

لاحظ أننا استخدمنا == المزدوجة لاختبار التساوي في تعليمة if الأولى، بينمـا اسـتخدمنا علامـة = مفـردةً لإسناد القيم إلى المتغيرات، وهذا الخطأ شائع جدًا في الكتابة بلغة بايثون، لأنه يعارض المنطـق الرياضـي الـذي تعودنا عليه، وبايثون تحذرك من أن هذا خطأ لغوي syntax error، وعليك النظر والبحث لإيجاده.

ننفـذ اختبـار "أكـبر من" بـدءًا من القيمـة العظمى إلى الصـغرى، فـإذا عكسـنا هـذا المنطـق، أي بـدأنا من price > 200 فستتحقق التعليمة دومًـا ولن ننتقـل إلى اختبـار 500 >، وبالمثـل يجب أن يبـدأ اسـتخدام تسلسل اختبارات "أقل من" من القيمة الصغرى ثم ننتقل إلى العظمى، وهـذا أيضًـا أحـد الأخطـاء الشـائعة في البرمجة ببايثون.

10.3.1 جافاسكربت VBScript

يمكن أن نسلسـل تعليمـات if في جافاسـكربت VBScript أيضًـا، لكننـا لن نشـرح إلا VBScript لأن المثال واضح:

```
<script type="text/vbscript">
DIM Price
price = InputBox("What's the price?")
price = CInt(price)
If price = 100 Then
```

```
    MsgBox "I'll take it!"
Else
    If price > 500 Then
        MsgBox "No way Jose!"
    Else
        If price > 200 Then
            MsgBox "How about throwing in a free mouse mat too?"
        Else
            MsgBox "price is an unexpected value!"
        End If
    End If
End If
</script>
```

يجب ملاحظة وجود تعليمة `End If` مطابقة لكل تعليمة `If`، واستخدامنا لدالة التحويل `CInt` الخاصة بلغة VBScript لتحويل الدخل من سلسلة نصية إلى عدد صحيح.

10.4 تعليمات الحالة Switch/Case

تتسبب إزاحة الشيفرات بملء الصفحة بسرعة، وهو أحد مساوئ سلسلة تعليمات `if/else` أو تشعبها، لكن شيوع هذا التسلسل المتشعب في البرمجة جعل كثيرًا من اللغات توفر نوعًا آخر من التفريع خاصًا بها، ويشار إليه عادةً باسم تعليمة `Case` أو `Switch`، التي تبدو في جافاسكربت كما يلي:

```
<script type="text/javascript">
function doArea(){
    var shape, breadth, length, area;
    shape   = document.forms["area"].shape.value;
    breadth = parseInt(document.forms["area"].breadth.value);
    len     = parseInt(document.forms["area"].len.value);
    switch (shape){
        case 'Square':
            area = len * len;
            alert("Area of " + shape + " = " + area);
            break;
        case 'Rectangle':
            area = len * breadth;
            alert("Area of " + shape + " = " + area);
```

```
            break;
        case 'Triangle':
            area = len * breadth / 2;
            alert("Area of " + shape + " = " + area);
            break;
        default: alert("No shape matching: " + shape)
        };
}
</script>

<form name="area">
<label>Length:  <input type="text" name="len"></label&lgt;
<label>Breadth: <input type="text" name="breadth"></label>
<label>Shape:   <select name="shape" size=1 onChange="doArea()">
            <option value="Square">Square
            <option value="Rectangle">Rectangle
            <option value="Triangle">Triangle
          </select>
          </label>
</form>
```

لا تقلق من حجم الشيفرة أعلاه، فما هي إلا امتداد لما شرحناه في الفصول السابقة وإن زاد حجمه قليلًا.

تحتوي استمارة HTML التي في آخر الشيفرة على حقلين نصيين لإدخال الطول length و العرض breadth، أما حقل الإدخال الثالث فهو قائمة منسدلة من القيم التي ألحقناها بسمة onChange التي تستدعي دالةً ما، وكل تلك الحقول لديها عناوين labels مرتبطة.

تسمح شيفرة الاستمارة في HTML بالتقاط التفاصيل والبيانات، ثم تستدعي دالة جافاسكربت حين يختار المستخدم أحد الأشكال، وبما أننا لم نشرح الدوال بعد، فيكفي أن تعلم الآن أن الدالة هي برنامج صغير تستدعيه البرامج الأخرى، وفي حالتنا هذه عرَّفنا تلك الدالة في أول الشيفرة.

تنشئ الأسطر الأولى بعض المتغيرات المحلية، وتحوِّل سلاسل الإدخال النصية إلى أعداد صحيحة عند الحاجة، وما نريده هو تعليمة switch، التي تختار الإجراء المناسب وفقًا لقيمة الشكل.

لاحظ أن الأقواس التي حول shape ضرورية، ولا تُحدَّد كتل الشيفرة التي داخل بنية case باستخدام الأقواس المعقوصة كما هو متوقع، فإذا أردنا إنهاءها، فسنستخدم تعليمة break؛ أما مجموعة تعليمات case الخاصة بـ switch فهي مرتبطة معًا مثل كتلة واحدة بين قوسين معقوصين.

يلتقط الشرط الأخير `default` أي شيء لم تلتقطه تعليمات `case` السابقة.

جرب توسيع المثال السابق ليشمل الدوائر، وتذكر أن تضيف خيارًا جديدًا إلى القائمة المنسدلة في استمارة HTML وتعليمة `case` جديدة إلى `switch`.

10.4.1 بنية Case الاختيارية في VBScript

تحتوي لغة VBscript على بنية case كما يلي:

```
<script type="text/vbscript">
Dim shape, length, breadth, SQUARE, RECTANGLE, TRIANGLE
SQUARE = 0
RECTANGLE = 1
TRIANGLE = 2
shape  = CInt(InputBox("Square(0),Rectangle(1) or Triangle(2)?"))
length = CDbl(InputBox("Length?"))
breadth  = CDbl(InputBox("Breadth?"))
Select Case shape
 Case SQUARE
   area = length * length
   MsgBox "Area = "  & area
 Case RECTANGLE
   area = length * breadth
   MsgBox "Area = " & area
 Case TRIANGLE
   area = length * breadth / 2
   MsgBox "Area = " & area
 Case Else
   MsgBox "Shape not recognized"
End Select
</script>
```

تجمع الأسطر الأولى البيانات من المستخدم وتحولها إلى النوع المناسب، تمامًا مثلما يحدث في جافاسكربت، وتظهر تعليمة `select` بنية `case` الخاصة بلغة VBScript، حيث تنهي كل تعليمة `Case` الكتلة التي سبقتها، كذلك لدينا فقرة `Case Else` التي تلتقط أي شيء لم تلتقطه تعليمات `Case` التي سبقتها، كما في حالة `default` في جافاسكربت.

وكما تعودنا في أسلوب VBScript في التنسيق، فإن الشيفرة تغلَق بتعليمة `End select`.

ربما تجب الإشارة إلى استخدام الثوابت الرمزية بدلًا من الأعداد، أي استخدام المتغيرات SQUAR وRECTANGLE وTRIANGLE، لأنها تجعل الشيفرة أسهل في القراءة، كما أن استخدام الأحرف الكبيرة هو اصطلاح لبيان كونها قيمًا ثابتةً وليست متغيرات، مع أن VBScript تسمح بأي أسماء نريدها للمتغيرات.

10.4.2 الاختيار المتعدد في بايثون

لا توفر بايثون بنية case صراحةً، وإنما تعوضنا عن ذلك بتوفير صيغة مضغوطة من if/else-if/else:

```
menu = """
Pick a shape(1-3):
Square
Rectangle
Triangle
"""
shape = int(input(menu))
if shape == 1:
   length = float(input("Length: "))
   print( "Area of square = ", length ** 2 )
elif shape == 2:
   length = float(input("Length: "))
   width = float(input("Width: "))
   print( "Area of rectangle = ", length * width )
elif shape == 3:
   length = float(input("Length: "))
   width = float(input("Width: "))
   print( "Area of triangle = ", length * width/2 )
else:
   print( "Not a valid shape, try again" )
```

لاحظ استخدام elif، وعدم تغير الإزاحة على عكس مثال تعليمة if المتشعبة، وتجدر الإشارة إلى أنه لا فرق بين استخدام if المتشعبة أو elif، فكلاهما صالح، كما أنهما تؤديان نفس الغرض، غير أن elif أيسر في القراءة إذا كان لدينا عدة اختبارات.

أما الشرط الأخير فهو else التي تلتقط أي شيء لم تلتقطه الاختبارات السابقة، كما في default في جافاسكربت وCase Else في VBScript.

توفر VBScript نسخةً معقدةً من هذه التقنية هي `ElseIf...Then` الـتي تُسـتخدم بنفس طريقـة `elif` في بايثون، لكنها غير شائعة بما أن `Select Case` أسهل استخدامًا.

10.5 إيقاف الحلقة التكرارية

يغطي هذا الجزء استخدامًا خاصًا للتفريع كنـا نريـد شـرحه في فصـل الحلقـات التكرارية، لكن اضـطررنا إلى الانتظار حتى شرح تعليمات if، لذا نعود الآن إلى موضوع الحلقات التكرارية لـنرى كيـف يمكن أن يحـل التفريـع مشكلةً شائعةً فيها، فقد نريد الخروج من حلقة تكراريـة قبـل أن تنتهي. تـوفر أغلب اللغـات آليـةً لإيقـاف تنفيـذ الحلقة، وفي بايثون تكون تلك الآلية هي كلمة `break`، ويمكن اسـتخدامها في حلقـتي `for` و`while`. تُسـتخدم هذه الكلمة غالبًا عند البحث في قائمة عن قيمة ما وإيجادها، عندها لا نريد متابعة البحث إلى نهاية الحلقـة، كمـا تُستخدم إذا اكتشفنا خللًا في البيانات، أو إذا قابلنا شرط خطأ error condition؛ أما أكثر الاستخدامات شـيوعًا خاصةً في بايثون التي اعتُمد فيها هذا السلوك لكثرة استخدامه، فهو عند قراءة الدخل باستخدام حلقة `while`.

في مثالنا التـالي سنسـتخدم كلمـة `break` للخـروج من الحلقـة عنـدما نقابـل أول قيمـة صـفرية في قائمة من الأعداد:

```
nums = [1,4,7,3,0,5,8]
for num in nums:
   print( num )
   if num == 0:
      break  # اخرج من الحلقة فورًا
print("We finished the loop")
```

في المثال السابق نستطيع رؤية كيفية اختبار شرط الخروج في تعليمة `if`، ثم استخدام `break` للقفز خارج الحلقة. تحتوي جافاسكربت على الكلمة المفتاحية `break` لإيقاف الحلقات التكرارية، وتُسـتخدم بنفس الطريقـة المستخدمة في بـايثون؛ أمـا VBScript، فتسـتخدم الكلمـات `Exit For` أو `Exit Do` للخـروج من حلقاتهـا التكرارية.

> توجد كلمة قريبة في سلوكها من `break`، وهي `continue` غير أنها أقل استخدامًا منها، ووظيفتها الخروج من متن الحلقة فقط، بدلًا من إنهاء الحلقة بالكلية، حيث تقفز إلى بداية الحلقة التكرارية وتبدأ التكرار التالي، قد يكون هذا مفيدًا إذا أردنا معالجة أنواع بعينها من البيانات مثلًا، لكن يمكن الحصول على نفس النتيجة بشرط `if` داخل الكتلة البرمجية، تحتوي بايثون وجافاسكربت على كلمة `continue`، على عكس VBScript.

10.6 الجمع بين الشروط والحلقات التكرارية

بما أن الأمثلة التي أوردناها سابقًا كانت نظريةً وتجريديةً، فسننظر الآن في مثال يستخدم كل ما تعلمناه سابقًا لنشرح تقنيةً شائعةً في البرمجة، وهي عرض القوائم للتحكم في مدخلات المستخدم، لنر الشيفرة أولًا ثم نشرحها:

```
menu = """
Pick a shape(1-3):
Square
Rectangle
Triangle

Quit
"""
shape = int(input(menu))
while shape != 4:
   if shape == 1:
      length = float(input("Length: "))
      print( "Area of square = ", length ** 2 )
   elif shape == 2:
      length = float(input("Length: "))
      width = float(input("Width: "))
      print( "Area of rectangle = ", length * width )
   elif shape == 3:
      length = float(input("Length: "))
      width = float(input("Width: "))
      print( "Area of triangle = ", length * width / 2 )
   else:
      print( "Not a valid shape, try again" )
   shape = int(input(menu))
```

لقد زدنا ثلاثة أسطر على مثال بايثون السابق، وهي:

```
Quit

while shape != 4:
```

```
shape = int(input(menu))
```

لكن هذه الأسطر الثلاثة جعلت البرنامج أكثر سهولةً، فقد مكنّا المستخدم من استمرار حساب أحجام الأشكال المختلفة إلى أن يجمع كل المعلومات التي يريدها، وذلك بإضافة الخيار `Quit` إلى القائمة وحلقة `while`، كما لا نحتاج الآن إلى إعادة تشغيل البرنامج يدويًا في كل مرة.

أما السطر الثالث الذي أضفناه فقد كان لتكرار عملية اختيار الشكل `input(menu)` ليتمكن المستخدم من تغيير الشكل والخروج من البرنامج إذا أراد.

يوهم هذا البرنامج المستخدم بأنه يعرف ما يرغب به وينفذه له، ويختلف سلوك البرنامج في كل مرة باختلاف مدخلات المستخدم، مما يجعل المستخدم يبدو وكأنه هو المتحكم بينما يكون المتحكم الحقيقي هو المبرمج، لأنه توقع جميع المدخلات الصالحة وجعل البرنامج يتفاعل معها، فتكون العبقرية الحقيقية هنا للمبرمج وليس للبرنامج أو الحاسوب، فالحواسيب غبية كما ذكرنا من قبل.

تتضح الآن سهولة توسعة وظائف برنامجنا بمجرد إضافة بعض الأسطر ودمج تسلسلات -الكتل البرمجية التي تحسب مساحات الأشكال-، والحلقات التكرارية -حلقة `while`-، والتعليمات الشرطية -بنية `if` / `elif`-، وبهذا نكون قد أنهينا الوحدات الثلاث الأساسية التي تتكون منها البرمجة وفقًا لديكسترا، ونستطيع الآن برمجة أي شيء، من الناحية النظرية، بما أننا تعلمنا هذه الوحدات، غير أننا لا زلنا بحاجة إلى تقنيات أخرى لنتعرف على كيفية تسهيل كتابة البرامج على أنفسنا.

10.6.1 مبدأ DRY: لا تكرر نفسك

إحدى المزايا التي نريد الإشارة إليها في هذا البرنامج هو أننا اضطررنا إلى تكرار سطر `input()` مرةً قبل الحلقة ومرةً داخلها؟ ولا يفضَّل تكرار الشيفرة البرمجية نفسها أكثر من مرة، لأننا سنضطر إلى تذكر تغيير كلا السطرين إذا احتجنا إلى تعديلها أو تغييرها، مما يفسح مجالًا للخطأ والنسيان، وهنا يمكن استخدام حيلة مبنية على خاصية `break` التي تحدثنا عنها في فصل الحلقات التكرارية، حيث نجعل في هذه الحيلة حلقة `while` حلقةً لا نهائية، ثم نختبر شرط الخروج، ونستخدم `break` للخروج منها، انظر إلى المثال التالي:

```
menu = """
Pick a shape(1-3):
Square
Rectangle
Triangle

Quit
"""
while True:
```

```
    shape = int(input(menu))
    if shape == 4: break

    if shape == 1:
       length = float(input("Length: "))
       print( "Area of square = ", length ** 2 )
    elif shape == 2:
       length = float(input("Length: "))
       width = float(input("Width: "))
       print( "Area of rectangle = ", length * width )
    elif shape == 3:
       length = float(input("Length: "))
       width = float(input("Width: "))
       print( "Area of triangle = ", length * width / 2 )
    else:
       print( "Not a valid shape, try again" )
```

يسمى تقليل التكرار هذا، والذي يبدو بدهيًا في أوساط المبرمجين، باسم مبدأ DRY، وهـو اختصـار للعبـارة: "لا تكرر نفسك Dont Repeat Yourself".

10.7 التعابير الشرطية

توجد صورة أخرى من التفريع شائعة جدًا في البرمجة، عندما نريد إسناد قيمة مختلفة إلى متغير وفقًا لشرط ما، ويمكن فعل هذا بسهولة باستخدام شرط `if/else` كما يلي:

```
if someCondition:
   value = 'foo'
else:
   value = 'bar'
```

إلا أن لغات البرمجة توفر اختصارًا لذلك، وهو ما يسمى ببنية التعبير الشـرطي، ويبـدو في بـايثون كمـا يلي، وهو مطابق تمامًا في وظيفته للشيفرة أعلاه:

```
value = 'foo' if شرط ما else 'bar'
```

لا تحتـوي VBScript على مثـل هـذه البنيـة، وتـوفر جافاسـكربت شـيئًا شـبيهًا بهـا، لكن البنيـة اللغويـة له مبهمة قليلًا:

```
<script type="text/javascript">
var someCondition = true;
var s;

s = (someCondition ? "foo" : "bar");
document.write(s);
</script>
```

لاحظ البنية الغريبة للشرط بين القوسين ()، الذي له نفس وظيفة الأمثلة السابقة في بايثون، إلا أنه يستخدم مجموعةً مختصرةً من الرموز، وهو يقول: "إن تحقق التعبير الذي يسبق علامة الاستفهام وكان true، فأعد القيمة التي بعد علامة الاستفهام، وإلا فأعد القيمة التي بعد النقطتين الرأسيتين". لا يُعَد القوسان ضروريين في هذا المثال رغم استخدامنا لهما، لأنهما يوضحان للقارئ ما يحدث في الشيفرة، ويُنصح باستخدامهما حتى في التعبيرات الشرطية في بايثون.

إن مثل هذه الاختصارات تبدو جماليةً مريحةً في استخدامها، لكن كثيرًا من المبرمجين ينأون عنها ويرونها غير عملية، لذا يجب استخدمها متى كان وجودها ضروريًا ومتى دعت الحاجة إليها، ويُفضل تجنبها إذا جعلت الشيفرة تبدو معقدةً أكثر من اللازم.

10.8 تعديل التجميعات من داخل الحلقات التكرارية

لقد ذكرنا في فصل الحلقات التكرارية أن تعديل التجميعة collection من داخل حلقة تكرارية أمر صعب، لكن لم نشرح كيفية تعديلها وقتها، إذ أردنا الانتظار حتى نشرح التفريع أولًا، وقد أتى وقت بيان ذلك، يمكن استخدام حلقة while لتنفيذ التعديلات أثناء التكرار على التجميعة نفسها، وهذا ممكن لأننا نستطيع التحكم الصريح في الفهرس داخل بنية while على عكس حلقة for التي يحدَّث الفهرس فيها تلقائيًا. لننظر الآن كيف نحذف جميع الأصفار من قائمة ما:

```
myList = [1,2,3,0,4,5,0]
index = 0
while index < len(myList):
   if myList[index] == 0:
      del(myList[index])
   else:
      index += 1
print( myList )
```

تجدر الإشارة هنا إلى أننا لا نزيد الفهرس عند حذف عنصر، بل تحرك عملية الحذف باقي العناصر للأعلى كي يشير الفهرس القديم إلى العنصر التالي في التجميعة.

سنستخدم فرع `if/else` لنتحكم في وقت زيادة الفهرس، ويسهل هنا ارتكاب خطأ في تنفيذ مثل هذا التعديل، لذا احرص على اختبار شيفرتك جيدًا، هناك مجموعة أخرى من دوال بايثون مصممة خصيصًا لتعديل محتويات القائمة، وسننظر فيها في فصل البرمجة الوظيفية Functional Programming لاحقًا.

10.9 خاتمة

في نهاية هذا الفصل نريد التذكير بما يلي:

- استخدم `if/else` لتفريع مسار البرنامج.
- استخدام `else` أمر اختياري يعود إليك.
- يمكن تمثيل القرارات المتعددة باستخدام بنية `Case` أو `if/elif`.
- تعيد التعابير البوليانية القيمة `True` أو `False`، ويمكن دمجها باستخدام `and` أو `or`.
- دمج القوائم باستخدام البنية `Case` يسمح لنا ببناء تطبيقات يتحكم فيها المستخدم.

خمسات

لبيع وشراء الخدمات المصغرة

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من $5 فقط

تصفح الخدمات

11. البرمجة باستخدام الوحدات

إن العنصر الرابع من عناصر البرمجة هو الوحدات، ومع أنك تستطيع كتابة بعض البرامج الرائعة بما تعلمته من بداية هذه الدروس إلى الآن دون تعلم الوحدات، إلا أن تتبع ما يحدث في البرامج -مع زيادة حجمها- يصبح صعبًا جدًا، وسنحتاج إلى طريقة لاستخلاص بعض التفاصيل لنستطيع التفكير في المشكلات التي تواجهنا ونريد حلها، بدلًا من صرف الجهد والوقت في التفاصيل الدقيقة المتعلقة بكيفية عمل الحاسوب. توفر بايثون وجافاسكربت وVBScript ذلك إلى حد ما بما فيها من إمكانيات مضمّنة تلقائيًا، فهي تجنبنا التعامل مع عتاد الحواسيب والنظر في كيفية قراءة كل مفتاح على لوحة المفاتيح على حدة، وغير ذلك من الأمور التي تستنزف موارد المبرمجين.

وتعتمد فكرة البرمجة باستخدام الوحدات على السماح للمبرمج بتوسيع الإمكانيات الموجودة في لغة البرمجة، عبر تجميع أجزاء من البرنامج في وحدات يمكن توصيلها ببرامجنا، وقد كانت الصورة الأولى للوحدات هي البرامج الفرعية subroutines التي تمثل كتلةً من الشيفرة نستطيع القفز إليها، بدلًا من استخدام GOTO التي ذكرناها في الفصل السابق، لكنها تقفز عند تمام الكتلة عائدةً إلى المكان الذي استدعيت منه، ويُعرف هذا الأسلوب من الوحدات بالإجراء procedure أو الدالة function، بل إن مفهوم الوحدة module نفسه يتغير في بعض اللغات مثل بايثون، ليأخذ معنًى أكثر دقة.

11.1 استخدام الدوال

لننظر أولًا في كيفية استخدام الدوال functions الكثيرة التي تأتي في أي لغة برمجة، حيث تسمى مجموعة الدوال القياسية المضمّنة في اللغة باسم المكتبة القياسية، وقد رأينا استخدام بعض الدوال، وذكرناه في قسم العوامل في الفصل الخامس: البيانات وأنواعها؛ أما الآن فسننظر في الأمور المشتركة بينها، وكيف يمكن أن نستخدمها في برامجنا.

للدالة الهيكل الأساسي التالي:

```
aValue = someFunction ( anArgument, another, etc... )
```

يأخذ المتغير aValue القيمة التي نحصل عليها باستدعاء الدالة someFunction، والتي تقبل عدة وسطاء arguments بين القوسين -وقد لا يوجد أي وسيط-، وتعاملها على أنها متغيرات داخلية، وتستطيع الدوال أن تستدعي دوالًا أخرى داخلها.

تفرض بعض اللغات وضع القوسين للدالة عند استدعائها حتى لو لم يكن لها وسطاء، وقد ذكرنا أنه يفضل تعويد النفس على استخدام الأقواس حتى في لغات مثل VBScript التي لا تشترط استخدامها حتى مع وجود الوسطاء.

لننظر الآن في بعض الأمثلة في لغاتنا الثلاث، ولنرى الدوال عمليًا:

11.1.1 الدالة Mid في VBScript

تعيد الدالة Mid(aString, start, length) مجموعة المحارف من aString بدءًا من start وبعدد محارف length، حيث ستعرض الشيفرة التالية "Good EVENING" مثلًا.

```
<script type="text/vbscript">
Dim time
time = "MORNING EVENING AFTERNOON"
MsgBox "Good" & Mid(time, 8, 8)
</script>
```

نلاحظ أن VBScript لا تتطلب أقواسًا لجمع وسطاء الدالة بل تكتفي بالمسافات كما كنا نفعل مع MsgBox، لكن عند جمعنا دالتين -كما فعلنا هنا- فيجب أن تستخدم الدالة الداخلية أقواسًا، والنصيحة العامة هي استخدام الأقواس عندما نشك هل يجب استخدامها أم لا.

11.1.2 الدالة Date في VBScript

تعيد الشيفرة التالية التاريخ الحالي للنظام:

```
<script type="text/vbscript">
MsgBox Date
</script>
```

بما أن المثال بسيط فلا يوجد ما يمكن شرحه هنا، لكن تجدر الإشارة إلى وجود مجموعة كاملة من دوال التاريخ الأخرى التي تستخرج اليوم والأسبوع والساعة.

11.1.3 الدالة startString.replace في جافاسكربت

تعيد دالة `startString.replace(searchString, newString)` سلسلةً نصيةً جديدةً مع وضع `newString` بدلًا من `searchString` في `startString`.

```
<script type="text/javascript">
var r,s = "A long and winding road";
document.write("Original = " + s + "<BR>");
r = s.replace("long", "short");
document.write("Result  = " + r);
</script>
```

لاحظ أن الدوال في جافاسكربت ما هي إلا مثال لنوع خاص يسمى بالتابع method، وهو دالة ترتبط بكائن، كما شرحنا في الفصل الخامس: البيانات وأنواعها، وكما سنشرح لاحقًا. وما نريد قوله، هو أن الدالة ترتبط بالسلسلة النصية `s` بواسطة عامل النقطة `.`، وهذا يعني أن `s` هي السلسلة التي سننفذ عليها الاستبدال، وهذا أمر عرضناه من قبل باستخدام التابع `write()` الخاص بالكائن `document` الذي استخدمناه لعرض الخرج من برامج جافاسكربت باستخدام `document.write()`، لكننا لم نشرح السبب وراء صيغة الاسم المزدوجة حتى الآن.

11.1.4 الدالة Math.pow في جافاسكربت

نستخدم الدالة `pow(x,y)` في وحدة Math -التي ذكرناها في الفصل الخامس: البيانات وأنواعها- لرفع x إلى الأس y:

```
<script type="text/javascript">
document.write( Math.pow(2,3) );
</script>
```

11.1.5 الدالة pow في بايثون

تستخدم بايثون دالة `pow(x,y)` لرفع x إلى الأس y:

```
>>> x = 2   #  سنستخدم 2 كرقم قاعدة
>>> for y in range(0,11):
...     print( pow(x,y) )    # y ارفع 2 إلى الأس
                                  # 10-0 أي من
```

نولد هنا قيم y من 0 إلى 10، ونستدعي الدالة المضمنة `pow()` لتمرير وسيطين هما x وy، ونستبدل القيم الحالية لكل منهما في استدعاء `pow()` في كل مرة، ثم نطبع النتيجة.

تجدر الإشارة إلى أن العامل الأسي ** في بايثون يكافئ الدالة pow( ).

11.1.6 الدالة dir في بايثون

إحدى الدوال المفيدة المضمنة في بايثون هي دالة dir، والتي تعرض جميع الأسماء المصدَّرة داخل وحدة ما إذا مررنا اسم الوحدة إليها، بما في ذلك جميع المتغيرات والدوال التي يمكن استخدامها، وعلى الرغم من أننا لم نشرح إلا بضع وحدات في بايثون، إلا أنها تأتي بوحدات كثيرة. تعيد الدالة dir قائمةً من الأسماء الصالحة في الوحدة، والتي تكون دوالًا في الغالب. لنجرب هذه الدالة على بعض الدوال المضمنة:

```
>>> print( dir(__builtins__) )
```

لاحظ أن builtins هي إحدى الكلمات السحرية في بايثون، وعلى ذلك يجب إحاطتها بزوج من شرطتين سفليتين على كل جانب. وإذا أردنا استخدام dir() على أي وحدة، فسنحتاج إلى استيرادها أولًا باستخدام import، وإلا فستخبرنا بايثون أنها لا تتعرف عليها.

```
>>> import sys
>>> dir(sys)
```

لقد استوردنا وحدة sys في المثال أعلاه، وهي الوحدة التي ذكرناها أول مرة في الفصل الرابع: التسلسلات البسيطة، ونستطيع أن نرى كلًا من exit و argv و stdin و stdout في الخرج الأخير لدالة dir، بين الأشياء الأخرى الموجودة في وحدة sys.

لننظر الآن في وحدات بايثون بقليل من التفصيل قبل أن ننتقل إلى ما بعدها.

11.2 استخدام الوحدات في بايثون

تتميز لغة بايثون بقابليتها الكبيرة للتوسع، حيث نستطيع إضافة وحدات جديدة باستيرادها بواسطة import، وفيما يلي بعض الوحدات التي تأتي افتراضيًا مع بايثون.

11.2.1 الوحدة sys

رأينا sys سابقًا عند الخروج من بايثون، وهي تحتوي على مجموعة من الدوال المفيدة كدالة dir التي سبق شرحها، ويجب أن نستورد وحدة sys أولًا لنصل إلى تلك الدوال:

```
import sys          # نجعل الدوال متاحة
print( sys.path ) # نرى أين تبحث بايثون عن الدوال
sys.exit()          # أسبقها بـ 'sys'
```

وإذا علمنا أننا سنستخدم الدوال بكثرة وأنها لن تحمل نفس أسماء الدوال التي استوردناها أو أنشأناها؛ فعندئذ نستطيع فعل ما يلي:

```
from sys import *  # استورد جميع الأسماء في sys
print( path )      # يمكن استخدامها دون تحديد السابقة 'sys'
exit()
```

يتمثل خطر هذا النهج في إمكانية وجود دوال بنفس الاسم في وحدتين مختلفتين، وبالتالي لن نستطيع الوصول إلا إلى الدوال التي استوردناها ثانيًا، لأننا بهذه الطريقة سنلغي الدالة التي استوردناها أولًا، فإذا أردنا أن نستخدم بعض العناصر فقط، فيفضل أن ننفذ ذلك كما يلي:

```
from sys import path, exit  # استورد الدوال التي تحتاجها فقط
exit()                      # استخدم دون تحديد السابقة 'sys'
```

لاحظ أن الأسماء التي نحددها لا تحتوي على أقواس تليها، ففي تلك الحالة سنحاول تنفيذ الدوال بدلًا من استيرادها، ولا يُعطى هنا إلا اسم الدالة فقط.

نريد الآن أن نشرح طريقةً مختصرةً توفر علينا القليل من الكتابة، وهي أننا نستطيع إعادة تسمية الوحدة عند استيرادها إذا كان لها اسم طويل جدًا، مثلًا:

```
import SimpleXMLRPCServer as s
s.SimpleXMLRPCRequestHandler()
```

لاحظ كيف أخبرنا بايثون أن تعُد `s` اختصارًا لـ `SimpleXMLRPCServer`، ولن نحتاج بعد ذلك إلا إلى كتابة `s` كي نستخدم دوال الوحدة، مما قلل من الكتابة.

11.2.2 وحدات بايثون الأخرى

يمكن استيراد أي وحدة من وحدات بايثون واستخدامها بالطريقة السابقة، وهذا يشمل الوحدات التي ننشئها بأنفسنا، لنلق الآن نظرةً على بعض الوحدات القياسية في بايثون، وما توفره من وظائف:

- وحدة `sys`: تسمح بالتفاعل مع نظام بايثون:
  - `exit()`: للخروج.
  - `argv`: للوصول إلى وسطاء سطر الأوامر.
  - `path`: للوصول إلى مسار البحث الخاص بوحدة النظام.
  - `ps1`: لتغيير محث بايثون `>>>`.
- وحدة `os`: تسمح بالتفاعل مع نظام التشغيل:

- `name`: تعطينا اسم نظام التشغيل الحالي، وهي مفيدة في البرامج المحمولة التي لا تحتاج إلى تثبيت.
- `system()`: تنفذ أمرًا من أوامر نظام التشغيل.
- `mkdir()`: تنشئ مجلدًا.
- `getcwd()`: تعطينا مجلد العمل الحالي.

- وحدة `re`: تسمح هذه الوحدة بتعديل السلاسل النصية باستخدام التعابير النمطية regular expressions لنظام يونكس:
  - `search()`: تبحث عن النمط في أي موضع في السلسلة النصية.
  - `match()`: تبحث في البداية فقط.
  - `findall()`: تبحث عن جميع مرات الورود في سلسلة نصية.
  - `split()`: تقسيم السلسلة النصية إلى حقول مفصولة بالنمط.
  - `sub()` و`subn()`: استبدال السلاسل النصية.
- وحدة `math`: تسمح بالوصول إلى العديد من الدوال الرياضية.
  - `sin()` و`cos()`: دوال حساب المثلثات.
  - `Log()` و`log10()`: اللوغاريتمات الطبيعية والعشرية.
  - `ceil()` و`floor()`: دالتا الجزء الصحيح floor والمتمم الصحيح الأعلى ceiling.
  - `pi` و`e`: الثوابت الطبيعية.
- وحدة `time`: دوال التاريخ والوقت.
  - `time()`: تحصل على الوقت الحالي بالثواني.
  - `gmtime()`: تحول الوقت بالثواني إلى التوقيت العالمي UTC (أو GMT).
  - `localtime()`: التحويل إلى التوقيت المحلي.
  - `mktime()`: معكوس التوقيت المحلي.
  - `strftime()`: تنسيق السلسلة النصية للوقت، مثل YYYYMMDD أو DDMMYYYY.
  - `sleep()`: إيقاف البرنامج مؤقتًا لمدة n ثانية.

- وحدة `random`: تولد أرقامًا عشوائية، وهي مفيدة في برمجة الألعاب.
  - `randint()`: تولد عددًا صحيحًا عشوائيًا بين نقطتين تضمَّنان في التوليد.
  - `sample()`: تولد قائمةً فرعيةً عشوائيًا من قائمة أكبر.
  - `seed()`: إعادة ضبط مفتاح توليد الأعداد.

هذه الوحدات على كثرتها ليست إلا شيئًا يسيرًا من الوحدات التي توفرها بايثون، والتي تزيد على مئة واحدة وأكثر يمكن تحميلها. تذكر أننا نستطيع استخدام `dir()` و`help()` للحصول على معلومات عن كيفية استخدام الدوال المختلفة، وأن المكتبة القياسية موثقة جيدًا في فهرس وحدات بايثون، ومن المصادر الجيدة للوحدات الإضافية فهرس حزم بايثون، الذي ضم عند كتابتنا لهذا المقال أكثر من مئة ألف حزمة، وكذلك مشروع SciPy الذي يستضيف مئات وحدات المعالجة العلمية والعددية، ولا غنى عنه لمن يعمل على مشاريع تحليل علمية، وأفضل وسيلة للوصول إلى تلك الحزم هو تثبيت إحدى إصدارات بايثون التي تحتوي على SciPy كإضافة قياسية فيها، مثل موقع anaconda.org، كما يحتوي sourceforge ومواقع التطوير مفتوح المصدر الأخرى على عدة مشاريع لبايثون فيها وحدات نافعة، ويمكنك الاستعانة بمحرك البحث نظرًا لتعدد المصادر، وتستطيع تصفحها وغيرها إذا أضفت كلمة python في بحثك، ولا تنس قراءة توثيق بايثون للمزيد من المعلومات عن البرمجة للإنترنت والرسوميات وبناء قواعد البيانات وغيرها.

والمهم هنا إدراك أن أغلب لغات البرمجة تحوي هذه الدوال، والوظائف الأساسية إما مضمّنة فيها أو تكون جزءًا من مكتبتها القياسية، فابحث أولًا في توثيقك قبل كتابة دالة ما، فربما تكون موجودةً ولا تحتاج إلى كتابتها.

11.3 تعريف الدوال الخاصة بنا

يمكن إنشاء دوال خاصة بنا من خلال تعريفها، أي بكتابة تعليمة تخبر المفسر أننا نعرِّف كتلةً من الشيفرة، ويجب عليه تنفيذها عندما نطلبها في أي مكان في البرنامج.

11.3.1 VBScript

سننشئ الآن دالةً تطبع مثال جدول الضرب الخاص بنا لأي قيمة نعطيها إليها كوسيط، ;ستبدو هذه الدالة في VBScript كما يلي:

```
<script type="text/vbscript">
Sub Times(N)
Dim I
For I = 1 To 12
    MsgBox I & " x " & N & " = " & I * N
Next
```

```
End Sub
</script>
```

sنبدأ هنا في هذا المثال بالكلمة المفتاحية Sub -وهي اختصار برنامج فرعي Subroutine-، التي تلي محدد VBScript لكتل الشيفرات مباشرةً في السطر الثاني، ثم نعطيها قائمةً من المعامِلات بين قوسين.

أما الشيفرة التي داخل الكتلة المعرَّفة فهي شيفرة VBScript عادية مع استثناء أنها تعامل المعامِلات على أنها متغيرات محلية، لذا تسمى الدالة في المثال أعلاه Times، وتأخذ معامِلًا واحدًا هو N، كما تعرِّف المتغير المحلي I، ثم تنفذ حلقةً تكراريةً تعرض جدول الضرب الخاص بالعدد N باستخدام المتغيرين N و I، نستدعي هذه الدالة الجديدة كما يلي:

```
<script type="text/vbscript">
MsgBox "Here is the 7 times table..."
Times 7
</script>
```

لاحظ أننا عرّفنا معاملًا اسمه N ومررنا إليه الوسيط 7، وقد أخذ المعامل أو المتغير المحلي N داخل الدالة القيمة 7 عندما استدعيناه، ويمكن تعريف أي عدد نريده من المعامِلات في تعريف الدالة، وعلى البرامج المستدعية أن توفر قيمةً لكل معامِل. تسمح بعض لغات البرمجة بتعريف قيم افتراضية للمعامل في حالة عدم توفير قيمة، لتستخدم الدالة تلك القيمة الافتراضية.

كذلك فقد غلفنا المعامل N بقوسين أثناء تعريف الدالة، لكن هذا غير ضروري في VBScript عند استدعاء الدالة، كما قلنا من قبل.

لا تعيد هذه الدالة أي قيمة، وهو ما نسميه بالإجراء في البرمجة، وما هو إلا دالة لا تعيد قيمةً، وتفرّق لغة VBScript بين الدوال والإجراءات باستخدام أسماء مختلفة لتعريفاتهما، فمثلًا: تعيد الدالة التالية في VBScript جدول الضرب الخاص بنا في سلسلة نصية طويلة واحدة:

```
<script type="text/vbscript">
Function TimesTable (N)
  Dim I, S
  S = N & " times table" & vbNewLine
  For I = 1 to 12
    S = S & I & " x " & N & " = " & I*N & vbNewLine
  Next
  TimesTable = S
End Function
```

```
Dim Multiplier
Multiplier = InputBox("Which table would you like?")
MsgBox TimesTable (Multiplier)
</script>
```

تكاد صيغة هذه الشيفرة أن تطابق صيغة Sub، مع استثناء أننا استخدمنا الكلمة Function بـدلًا من Sub السابقة، ويجب أن نسند النتيجة إلى اسم الدالة داخل التعريـف، لهـذا سـتعيد الدالـة أي قيمـة يحتويهـا اسـمها عند خروجها:

```
  ...
  TimesTable = S
End Function
```

فإذا لم نسند قيمةً لاسم الدالة بشكل صريح، فستعيد الدالـة القيمـة الافتراضـية، والـتي تكـون في الغـالب صفرًا أو سلسلةً نصيةً فارغةً.

لاحظ أننا وضعنا أقواسًا حول الوسيط في سطر MsgBox، لأن MsgBox لن يعرف -لولا هذه الأقـواس- هـل يجب أن يُطبـع Multiplier أم يُمـرّره إلى الوسـيط الأول والـذي هـو TimesTable، فلمـا وضـعناه داخـل الأقواس فهم المفسر أن القيمة هي وسيط للدالة TimesTable بدلًا من MsgBox.

11.3.2 بايثون

ستكون دالة جدول الضرب في بايثون كما يلي:

```
def times(n):
    for i in range(1,13):
        print( "%d x %d = %d" % (i, n, i*n) )
```

وتُستدعى كما يلي:

```
print( "Here is the 9 times table..." )
times(9)
```

لاحظ أن الإجراءات في بايثون لا تميَّز عن الدوال، ويُستخدم def لتعريفهمـا، والفـرق الوحيـد هـو أن الدالـة التي تعيد قيمةً تستخدم تعليمة return، كما يلي:

```
def timesTable(n):
   s = ""
```

```
    for i in range(1,13):
        s = s + "%d x %d = %d\n" % (i,n,n*i)
    return s
```

الأمر بسيط، إذ تعيد الدالة النتيجة باستخدام تعليمة return، أما إذا لم يكن لدينا تعليمة return صريحة فستعيد بايثون تلقائيًا قيمةً افتراضيةً تسمى None، والتي نتجاهلها في العادة. نستطيع الآن طباعة نتيجة الدالة كما يلي:

```
print( timesTable(7) )
```

يفضل عدم وضع تعليمات print داخل الدوال، وإنما جعل الدالة تعيد النتيجة باستخدام return، ثم طباعتها من خارج الدالة، ورغم أننا لم نتبع هذا الأسلوب في أمثلتنا، إلا أن هذا يجعل الدوال قابلةً لإعادة الاستخدام في نطاق أوسع من الحالات.

يبقى أمر مهم للغاية يجب أن نذكره حول تعليمة return، وهو أنها لا تعيد قيمةً من الدالة فحسب، بل تعيد التحكم إلى الشيفرة التي استدعت الدالة، وتكمن أهمية ذلك في أن الإعادة لا يجب أن تكون آخر سطر في الدالة، فقد تكون هناك تعليمات أكثر، وقد لا تنفَّذ أبدًا، كما يمكن أن يحتوي متن الدالة الواحدة على عدة تعليمات return. تُنهي التعليمة التي نصل إليها أولًا الدالة وتعيد القيمة إلى الشيفرة المستدعية، سنعرض مثالًا عن دالة لها عدة تعليمات return، وهي تعيد أول عدد زوجي تجده في القائمة المزودة بها أو None إذا لم تجد أي عدد زوجي:

```
def firstEven(aList):
    for num in aList:
        if num % 2 == 0: # تحقق من كونه زوجيًا
            return num  # اخرج من الدالة فورًا
    return None    # لا نصل إليها إلا إذا لم نجد عددًا زوجيًا
```

11.3.3 ملاحظة بشأن المعاملات

قد يصعب على المبتدئين فهم دور المعامِلات في تعريف الدالة، فهل يجب تعريف الدالة كما يلي:

```
def f(x): # يمكن استخدام x داخل الدالة...
```

أم تعريفها بالشكل:

```
x = 42
def f(): # يمكن استخدام x داخل الدالة...
```

يعرِّف المثال الأول هنا المعامِل x ويستخدمه داخل الدالة، بينما يستخدم المثال الثاني متغيرًا معرَّفًا من خارج الدالة مباشرةً، وبما أن المثال الثاني صالح ويعمل دون مشاكل، فلم نتكبد عناء تعريف المعامل؟، والجواب هو أن المعامِلات تتصرف مثل متغيرات محلية، أي مثل المتغيرات التي لا تُستخدم إلا داخل الدالة، وقلنا إن مستخدم الدالة يستطيع أن يمرر وسطاءً إلى هذه المعامِلات، وعلى ذلك تتصرف قائمة المعامِلات مثل بوابة للبيانات التي تتحرك بين البرنامج الرئيسي والدالة.

تستطيع الدالة أن ترى بعض البيانات خارجها -انظر الفصل الخامس عشر: فضاءات الأسماء-، لكن إذا أردنا للدالة أن تكون قابلةً لإعادة الاستخدام في أكثر من برنامج، فيجب أن نقلل اعتمادها على البيانات الخارجية، بل يجب أن تُمرَّر البيانات المطلوبة لأداء وظيفة الدالة بكفاءة إليها من خلال معامِلاتها، فإذا عُرِّفت الدالة داخل ملف وحدة، فيجب أن تكون لها صلاحية قراءة البيانات المعرَّفة داخل نفس الوحدة، لكن هذه الخاصية ستقلل من مرونة الدالة التي نعرِّفها. نريد أن نقلل عدد المعامِلات المطلوبة لتشغيل الدالة إلى حد يمكن السيطرة عليه وإدارته، وهذا يراعى في حالة البيانات الكثيرة التي تحتاج إلى معامِلات أكثر، وذلك من خلال استخدام تجميعات البيانات مثل القوائم وصفوف tuples والقواميس وغيرها، كما نستطيع تقليل عدد قيم المعامِلات الفعلية التي يجب أن نوفرها، باستخدام ما يسمى بالوسيط الافتراضي.

11.3.4 الوسيط الافتراضي

يشير هذا المصطلح إلى طريقة لتعريف معامِلات الدالة التي تأخذ قيمًا افتراضيةً إذا لم تمرَّر كوسطاء صراحةً، وأحد استخدامات هذا الوسيط المنطقية في دالة تعيد يوم الأسبوع، فإذا استدعيناها بدون قيمة فيكون قصدنا اليوم الحالي، وإلا فإننا نوفر رقم اليوم كوسيط، كما يلي:

```
import time

# None قيمة اليوم هي
def dayOfWeek(DayNum = None):
    # طابق ترتيب اليوم مع قيم إعادة بايثون
    days = ['Saturday','Sunday',
            'Monday','Tuesday',
            'Wednesday', 'Thursday', 'Friday']

    # تحقق من القيمة الافتراضية
    if DayNum == None:
        theTime = time.localtime(time.time())
        DayNum = theTime[6] # استخرج قيمة اليوم
    return days[DayNum]
```

لا نحتاج إلى استخدام وحدة الوقت إلا إذا كانت قيمة المعامِل الافتراضي مطلوبةً، وبناءً عليه نستطيع تأجيل عملية الاستيراد إلى أن نحتاج إليها، وسيحسن هذا من الأداء تحسينًا طفيفًا إذا لم نضطر إلى استخدام خاصية القيمة الافتراضية للدالة، لكن هذا التحسن يُعَد طفيفًا كما ذكرنا، ويلغي اصطلاح الاستيراد الذي ذكرناه أعلاه، لذا لا يستحق هذا التشوش.

نستطيع الآن أن نستدعي ذلك كما يلي:

```
print( "Today is: %s" % dayOfWeek() )
Saturday.
print( "The third day is %s" % dayOfWeek(2) )
```

تذكر أننا نبدأ العد من الصفر في عالم الحواسيب، وأننا افترضنا أن اليوم الأول في الأسبوع هو السبت.

11.3.5 عد الكلمات

أحد الأمثلة الأخرى على دالة تعيد قيمةً، هو الدالة التي تَعُد الكلمات في سلسلة نصية، ويمكن استخدامها لحساب الكلمات في ملف ما بجمع كلمات كل سطر، وتكون شيفرة هذه الدالة كما يلي:

```
def numwords(s):
    s = s.strip() # أزل المحارف الزائدة
    list = s.split() # قائمة كل عنصر فيها يمثل كلمة
    return len(list) # عدد كلمات القائمة هو عدد كلمات s
```

لقد عرَّفنا الدالة في المثال أعلاه مستفيدين من بعض توابع السلاسل النصية المضمَّنة التي ذكرناها في الفصل الخامس: البيانات وأنواعها، ويمكن استخدام الدالة الآن كما يلي:

```
for line in file:
    total = total + numwords(line) # راكم مجموع كل سطر
print( "File had %d words" % total )
```

إذا حاولنا كتابة ذلك فلن تنجح الشيفرة، لأن مثل هذه الشيفرة تُعرف باسم الشيفرة الوهمية pseudocode، وهي تقنية تصميم شائعة للشيفرات لتوضيح الفكرة العامة لها، وليست مثالًا لشيفرة حقيقية صحيحة التركيب، وتُعرف أحيانًا باسم لغة وصف البرامج Program Description Language.

يتضح لنا سبب أفضلية إعادة قيمة من دالة وطباعة النتيجة خارج الدالة بدلًا من داخلها، فإذا طبعت الدالة الطول بدلًا من إعادته فلم نكن لنستخدمها في عدّ إجمالي الكلمات في الملف، بل كنا سنحصل على قائمة طويلة فيها طول كل سطر، لكننا بإعادة القيمة نستطيع الاختيار بين استخدام القيمة كما هي، أو تخزينها في متغير بهدف المعالجة اللاحقة كما فعلنا هنا من أجل أخذ العدد الإجمالي، وهذه نقطة في غاية الأهمية من ناحية التصميم لفصل عرض البيانات من خلال الطباعة عن معالجها داخل الدالة.

هناك ميزة أخرى وهي أننا إذا طبعنا الخرج فلن يكون مفيدًا إلا في بيئة سطر أوامر، أما بإعادة القيمة فنستطيع عرضها في صفحة ويب أو واجهة رسومية، فلفصل عرض البيانات عن معالجتها فائدة كبيرة، لذا اجتهد في إعادة القيم من الدوال بدلًا من طباعتها ما استطعت، وليس ثمة استثناء لهذه القاعدة إلا عند إنشاء دالة مخصصة لطباعة بعض البيانات، ففي تلك الحالة يجب توضيح هذا باستخدام كلمة print أو عرض اسم الدالة.

11.3.6 دوال جافاسكربت

يمكن إنشاء دوال داخل جافاسكربت باستخدام أمر `function`، كما يلي:

```
<script type="text/javascript">
var i, values;

function times(m) {
    var results = new Array();
    for (i = 1; i <= 12; i++) {
        results[i] = i * m;
        }
    return results;
}

// استخدم الدالة
values = times(8);

for (i=1;i<=12;i++){
    document.write(values[i] + "<br />");
    }
</script>
```

لن نستطيع الاستفادة كثيرًا من هذه الدالة بتلك الحالة، لكن هذا المثال يوضح كيف يشبه الهيكل الأساسي لإنشاء الدالة هنا تعريفات الدوال في بايثون وVBScript، وسننظر في دوال أكثر تعقيدًا في جافاسكربت لاحقًا، لأنها تستخدم الدوال لتعريف الكائنات والدوال أيضًا، وهو الأمر الذي يبدو مربكًا للقارئ أو لمستخدم اللغة.

> تنبع قوة الدوال من سماحها بتوسيع نطاق الوظائف والمهام التي تستطيع اللغة تنفيذها، كما أنها تتيح لنا إمكانية تغيير اللغة من خلال تعريف معنىً جديد لدالة موجودة مسبقًا -لا يُسمح بهذا في بعض اللغات-، ولكن هذا أمر غير محمود العاقبة، إلا إذا تحكمنا فيه بحذر شديد، لأن تغيير السلوك القياسي للغة يجعل قراءة الشيفرة وفهمها يُعَد صعبًا للغاية على غيرك، بل عليك أنت نفسك فيما بعد، وبما أن القارئ يتوقع من الدالة أن تنفذ سلوكًا معينًا

لكنك غيرت هذا السلوك إلى شيء آخر، لذا يفضل عدم تغيير السلوك الافتراضي للدوال المضمّنة في اللغة. يمكن التغلب على هذا التقييد للإبقاء على السلوك المضمّن للغة مع الاستمرار في استخدام أسماء ذات معنىً لدوالنا، من خلال وضع الدوال داخل كائن أو وحدة توفر سياقها المحلي، وسننظر في المنظور الكائني في الفصل السابع عشر: البرمجة كائنية التوجه، أما الآن فننظر في إنشاء وحداتنا الخاصة.

11.4 إنشاء الوحدات الخاصة

رأينا كيف ننشئ دوالًا خاصةً بنا وكيف نستدعيها من أجزاء أخرى من البرنامج، وهذا أمر جيد لأنه يوفر علينا كثيرًا من الكتابة المتكررة، ويجعل برامجنا سهلة الفهم، لأننا ننسى بعض التفاصيل بعد إنشاء دالة تخفيها، وهو مبدأ متبع في البرمجة عند الحاجة إلى إخفاء بعض التفاصيل، ويسمى إخفاء المعلومات، حيث تغلَّف المعلومات والتفاصيل في دالة ننشئها لها، لكن كيف نستخدم هذه الدوال في البرامج الأخرى؟ الإجابة على هذا هي إنشاء وحدة module لهذا الغرض.

11.4.1 وحدات بايثون

الوحدة في بايثون ما هي إلا ملف نصي بسيط فيه تعليمات برمجية مكتوبة بلغة بايثون، وتكون تلك التعليمات عادةً تعريفات دوال، فمثلًا عند كتابة:

```
import sys
```

فإننا نخبر مفسر بايثون أن يقرأ هذه الوحدة وينفذ الشيفرة الموجودة فيها، ويتيح لنا الأسماء التي تولدها في ملفنا، ويبدو هذا كأننا ننشئ نسخةً من محتويات `sys.py` في برنامجنا، على أن بعض الوحدات مثل `sys` في البرمجة العملية ليس لها ملف `sys.py` أصلًا، لكننا سنتجاهل هذا الآن، وتوجد لغات برمجة مثل C و++C، التي ينسخ فيها المترجم أو المصرِّف ملفات الوحدة إلى البرنامج الحالي حسب الطلب.

ننشئ الوحدة بإنشاء ملف بايثون يحتوي الدوال التي نريد إعادة استخدامها في برامج أخرى، ثم نستورد الوحدة كما نستورد الوحدات القياسية، ولتنفيذ هذا عمليًا، انسخ الدالة التالية إلى ملف، واحفظه باسم `timestab.py`. يمكنك فعل هذا باستخدام IDLE أو Notepad، أو أي محرر آخر يحفظ الملفات النصية العادية، لكن لا تستخدم برامج معالجة نصوص مثل مايكروسوفت وورد، لأن تلك البرامج تدخل شيفرات تنسيق كثيرةً لن تفهمها بايثون.

```
def print_table(multiplier):
    print( "--- Printing the %d times table ---" % multiplier )
    for n in range(1,13):
        print( "%d x %d = %d" % (n, multiplier, n*multiplier) )
```

ثم اكتب في محث بايثون:

```
>>> import timestab
>>> timestab.print_table(12)
```

وهكذا تكون قد أنشأت وحدةً واستوردتها واستخدمت الدالة المعرَّفة فيها.

لاحظ أنك إن لم تبدأ بايثون من نفس المجلد الذي خزنت فيه ملف `timestab.py`، فلن تستطيع بايثون أن تجد الملف وستعطيك خطأً، وعندها يمكن إنشاء متغير بيئة اسمه PYTHONPATH يحمل قائمةً من المجلدات الصالحة للبحث فيها عن وحدات، إضافةً إلى الوحدات القياسية التي تأتي مع بايثون، ويفضل تعريف مجلد داخل PYTHONPATH وتخزين جميع ملفات الوحدات القابلة لإعادة الاستخدام داخله، ولا تنسى اختبار الوحدات جيدًا قبل نقلها إليه.

يجب التأكد من عدم استخدام اسم تحمله وحدة قياسية في بايثون، لئلا تجعل بايثون تستورد ملفك أنت بدلًا من القياسي، مما سينتج عنه سلوك غريب جدًا كما ذكرنا من قبل في شأن التلاعب في لغة البرمجة. ولا تستخدم اسم إحدى الوحدات التي تحاول استيرادها إلى نفس الملف، فهذا سيؤدي أيضًا إلى حدوث مشاكل.

> تختلف عملية إنشاء متغيرات البيئة من منصة لأخرى، وهذا أمر يُفترض أن يعلمه القارئ أو على الأقل يعرف كيف يبحث عن كيفية تنفيذه، حيث يستطيع مستخدمو ويندوز مثلًا أن يبحثوا عن متغيرات البيئة Environment Variables في قائمة ابدأ، ثم المساعدة والدعم، ويتعلموا بأنفسهم كيفية إنشائها.

11.4.2 الوحدات في جافاسكربت و VBScript

يُعَد إنشاء الوحدات في VBScript أكثر تعقيدًا من بايثون، فلم يكن مفهوم الوحدات موجودًا في هذه اللغة ولا في اللغات التي بنفس عمرها، بل كانت تعتمد على إنشاء الكائنات لإعادة استخدام الشيفرة بين المشاريع، وسننظر الآن في كيفية النسخ من المشاريع السابقة واللصق في مشروعنا الحالي باستخدام المحرر النصي.

> لاحظ أن لغة Visual Basic -وهي الأخت الكبرى للغة VBScript-، فيها ذلك المفهوم الخاص بالوحدات، ويمكن تحميل وحدة من خلال بيئة تطوير متكاملة IDE من قائمة File، ثم خيار Open Module، وهناك بعض القيود لما يمكن فعله داخل وحدة Visual Basic، لكننا لن نتعرض لها بما أنها خارج نطاق حديثنا. توفر مايكروسوفت نسخةً مجانيةً من أحدث إصدار للغة VB Express، لكن يجب أن تسجل لديهم قبل أن تستخدمها. انظر في صفحتها إذا أردت التجربة بنفسك.

أما جافاسكربت فتوفر آليةً لإنشاء وحدات من ملفات الشيفرة القابلة لإعادة الاستخدام، لكنها آلية معقدة وتستخدم صيغةً غامضةً تخرج عن نطاق عملنا في هذه المرحلة، غير أن لدينا حلًا أسهل، وهو استخدام وحدات كتبها أشخاص آخرون، باستخدام الصيغة التالية:

```
<script type=text/JavaScript src="mymodule.js"></script>
```

نستطيع الوصـول إلى جميـع تعريفـات الـدوال الموجـودة في ملـف mymodule.js بمجـرد إدراج السـطر السابق داخل قسم <head> في صفحة الويب الخاصة بنـا، وتوجـد وحـدات عديـدة من الطـرف الثـالث متاحـة لمبرمجي الويب ويمكن اسـتيرادها بهـذه الطريقـة، ولعـل أشـهرها هي وحـدة jQuery؛ أمـا في المواضـع الـتي تُستخدم فيها جافاسكربت خارج المتصفحات -انظر قسم Windows Script Host اللاحق- فهناك آليات أخرى متاحة، ويُرجع في ذلك إلى التوثيق.

11.4.3 Windows Script Host

نظرنا حتى الآن إلى جافاسكربت وVBScript على أنهما لغات للبرمجة داخل الويب، ولكن توجد طريقة أخرى لاستخدامهما داخل بيئة ويندوز، وهو تقنية مضيف سكربت ويندوز Windows Script Host أو WSH اختصارًا، وهي تقنية أتاحتها مايكروسوفت لتمكين المستخدمين من برمجة حواسيبهم بنفس الطريقة الـتي اسـتخدم بهـا مستخدمو نظام DOS قديمًا ملفات باتش Batch Files، فهي توفر آليات لقراءة الملفات والسجل والوصول إلى الحواسيب والطابعات التي في الشبكة وغيرها.

في الإصـدار الثـاني من WSH إمكانيـة تضـمين ملـف WSH آخـر، ومن ثم توفـير وحـدات قابلـة لإعـادة الاستخدام، وذلك بإنشاء ملف وحدة أولًا اسمه SomeModule.vbs يحتوي على ما يلي:

```
Function SubtractTwo(N)
   SubtractTwo = N - 2
End function
```

ننشئ الآن ملف سكربت WSH اسمه testModule.wsf مثلًا، كما يلي:

```
<?xml version="1.0" encoding="UTF-8"?>

<job>
  <script type="text/vbscript" src="SomeModule.vbs" />
  <script type="text/vbscript">
      Dim value, result
      WScript.Echo "Type a number"
      value = WScript.StdIn.ReadLine
      result = SubtractTwo(CInt(value))

      WScript.Echo "The result was " &amp; CStr(result)
  </script>
</job>
```

يمكن تشغيل هذا في ويندوز ببدء جلسة DOS وكتابة ما يلي:

```
C:\> cscript testModule.wsf
```

تسمى الطريقة التي تمت هيكلة ملف (.wsf) بها باسم XML، ويتواجد البرنامج داخل زوج من وسوم `<job></job>` بدلًا من وسم `<html></html>` الذي رأيناه في لغة HTML من قبل.

يشير أول وسم سكربت في الداخل إلى ملف وحدة اسمه `SomeModule.vbs`، أما وسم السكربت الثاني فيحتوي على برنامجنا الذي يصل إلى `SubtractTwo` داخل ملف `SomeModule.vbs`؛ بينما ملف `.vbs` فيحتوي على شيفرة VBScript عادية ليس فيها وسوم XML أو HTML.

لاحظ أن علينا تهريب محرف `&` من أجل ضم السلاسل النصية لتعليمة `WScript.Echo`، لأن التعليمة جزء من ملف XML، وهذا المحرف مستخدم في لغة XML كرمز محدِّد.

نستخدم `WScript.Stdin` لقراءة مدخلات المستخدم، وهو تطبيق لما تحدثنا عنه في الفصل التاسع: قراءة المدخلات من المستخدم.

تصلح هذه التقنية مع جافاسكربت أيضًا، أو لنكون أدق، مع نسخة مايكروسوفت من جافاسكربت التي تسمى JScript، وذلك بتغيير سمة `type=`، بل يمكن دمج اللغات في WSH بأن نستورد وحدةً مكتوبةً بجافاسكربت ونستخدمها في شيفرة VBScript أو العكس.

فيما يلي سكربت WSH مكافئ لما سبق، حيث تُستخدم جافاسكربت للوصول إلى وحدة من VBScript:

```
<?xml version="1.0" encoding="UTF-8"?>

<job>
  <script type="text/vbscript" src="SomeModule.vbs" />
  <script type="text/javascript">
      var value, result;
      WScript.Echo("Type a number");
      value = WScript.StdIn.ReadLine();
      result = SubtractTwo(parseInt(value));

      WScript.Echo("The result was " + result);
  </script>
</job>
```

نستطيع رؤية مدى تقارب هاتين النسختين، فإذا استثنينا بعض الأقواس الزائدة فسيمكننا القول أنهما متشابهتان للغاية؛ أما أغلب الأمور الفنية فتجري من خلال كائنات WScript.

لن نستخدم WSH كثيرًا، لكن قد ننظر فيها بين الحين والآخر إذا رأينا أنها توفر لنا مزايا لا يمكن شرحها باستخدام بيئة المتصفح الأكثر تقييدًا، فعلى سبيل المثال، سنستخدم WSH في الفصل التالي لبيان كيف يمكن تعديل الملفات باستخدام جافاسكربت وVBScript، وتوجد بعض الكتب التي تتحدث عن WSH إذا كنت مهتمًا بتعلم المزيد عنها، ويحتوي موقع مايكروسوفت على قسم خاص بها مع أمثلة لبرامج وأدوات تطوير وغير ذلك.

11.5 خاتمة

نأمل أن تخرج من هذا الفصل وقد تعلمت:

- الدوال التي هي شكل من أشكال الوحدات.
- تعيد الدوال قيمًا، أما الإجراءات فلا تعيد شيئًا.
- تتكون الوحدات في بايثون في العادة من تعريفات للدوال داخل ملف.
- يمكن إنشاء دوال جديدة في بايثون باستخدام الكلمة المفتاحية `def`.
- استخدام `Sub` أو`Function` في VBScript، و`function` في جافاسكربت.

دورة علوم الحاسوب

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

12. التعامل مع الملفات

لا تختلف الملفات من منظور برمجي عن الملفات التي نستخدمها في برامج معالجة الكلمات أو أي برامج أخرى، والتي نفتحها ونكتب فيها بعض المهام أو التعليمات ثم نغلقها مرةً أخرى، لكن أحد الفروق الرئيسية هنا هو أننا نقرأ الملف تتابعيًا، أي نقرأ سطرًا واحدًا في كل مرة من بدايته، ورغم أن برامج معالجة الكلمات تنتهج هذا النهج أيضًا، إلا أنها تحتفظ بالملف كله في الذاكرة أثناء عملنا عليه، ثم تنفذ عملية الكتابة كلها عند إغلاقها، وهناك فرق آخر بين الملفات البرمجية والعادية، وهو أننا حين نبرمج فإننا نفتح الملف للقراءة فقط أو الكتابة فقط، وننفذ الكتابة بإنشاء ملف جديد من الصفر أو إلغاء ملف موجود والكتابة فوقه، أو بإلحاق الملف الجديد إلى ملف موجود من قبل، كما يمكن العودة في الملفات البرمجية إلى بداية الملف أثناء معالجته.

12.1 الدخل والخرج بالملفات

لنطبق مثالًا عمليًا، ولنفترض وجود ملف اسمه `menu.txt`، يحتوي على قائمة من الوجبات:

```
steak & eggs
steak & chips
steak & steak
```

لنكتب برنامجًا يقرأ الملف ويعرض الخرج كما يفعل أمر `cat` في صدفة يونكس، وأمر `type` في صدفة ويندوز.

```
# (r) افتح الملف للقراءة
inp = open("menu.txt","r")
# اقرأ الملف سطرًا سطرًا
for line in inp:
```

```
    print( line )
# أغلق الملف مرة أخرى
inp.close()
```

تأخذ `open()` وسيطين، الأول هو اسم الملف الذي يمكن أن يكون متغيرًا أو سلسلةً نصيةً مجردةً literal string كما فعلنا هنا، والوسيط الثاني هو الوضع الذي نفتح به الملف، حيث يمكن فتحه للقراءة فقط `r` أو للكتابة `w`، كما نحدد هل هو للاستخدام النصي أم للاستخدام الثنائي binary، وذلك بإضافة محرف `b` إلى `r` أو `w` في حالة الاستخدام الثنائي، كما يلي:

```
open(fn,"rb")
```

لاحظ أننا نقرأ الملف في حلقة `for`، التي تتصرف في بايثون مثل حلقة foreach في تجميعة، إذ تعيد كل عنصر في التجميعة، ويمكن النظر إلى الملف النصي على أنه تجميعة من الأسطر، وهكذا تقرأ الحلقة كل سطر على حدة.

ثم نغلق الملف باستخدام دالة مسبوقة بمتغير ملف، وتُعرف هذه الصياغة باستدعاء التابع method invocation كما شرحنا في الفصل السابق، ويمكن تقريب مفهوم متغير الملف بأنه وحدة تحتوي الدوال التي تعمل على الملفات، ونستوردها في كل مرة ننشئ متغيرًا من نوع ملف.

تُغلَق الملفات تلقائيًا في بايثون في نهاية البرنامج، لكن من الأفضل أن نتعود على إغلاق الملفات إغلاقًا صريحًا، لأن نظام التشغيل قد لا يكتب البيانات إلى الملف إلا عند إغلاقه لأسباب تتعلق بتسريع أدائه، مما يعني أن بياناتنا قد لا تكتَب إلى الملف إذا خرج البرنامج فجأةً، لذا يفضَّل التعود على إغلاق الملف بمجرد الانتهاء من الكتابة فيه.

كما أننا لم نحدد المسار الكامل للملف الموجود في الشيفرة أعلاه، لذا سنتعامل مع الملف على أنه موجود في المجلد الحالي، لكن يمكن تمرير الاسم الكامل للمسار إلى `open()` بدلًا من اسم الملف مجردًا. قد نواجه مشكلةً بسيطةً في هذا الشأن في نظام ويندوز، إذ يُستخدم المحرف `\` لفصل المجلدات في مسارات ويندوز، لكن نفس المحرف له معنىً خاص آخر في سلاسل بايثون، لذلك يفضل استخدام المحرف `/` عند تحديد المسارات في بايثون لضمان عملها على أي نظام تشغيل بما فيها ويندوز.

عند التعامل مع ملف طويل لن نستطيع عرضه كاملًا على الشاشة مرةً واحدةً، وعلينا التوقف بعد كل شاشة كاملة، لذا نستخدم المتغير `line_count` ونزيده بعد قراءة كل سطر، ثم نتحقق هل يساوي 25 -إذا كانت الشاشة تحتوي على 25 سطرًا- أم لا، ثم سنطلب من المستخدم أن يضغط على زر ما عندما يساوي 25، وليكن زر الإدخال مثلًا، قبل إعادة ضبط عداد `line_count` إلى الصفر والمتابعة بعد ذلك.

توجد طريقة أخرى لقراءة الملفات باستخدام حلقة `while` وتابع لكائن الملف اسمه `readline()`، وتمتاز هذه الطريقة بأننا نستطيع التوقف عن معالجة الملف بمجرد العثور على البيانات التي نريدها، مما يسرع الأداء كثيرًا إذا كنا نتعامل مع ملفات طويلة، لكنها أعقد، لذا سننظر في مثالنا السابق باستخدام هذه الطريقة لنشرحها:

```
# افتح الملف للقراءة
inp = open("menu.txt","r")
# اقرأ الملف واطبع كل سطر
while True:
    line = inp.readline()
    if not line: break
    print( line )
# أغلق الملف
inp.close()
```

لاحظ أننا استخدمنا تقنية `break` التي ذكرناها في الفصل العاشر: مقدمة في البرمجة الشرطية، وذلك للخروج من الحلقة إذا كان السطر فارغًا، إذ يَعُد السطر الفارغ بالاصطلاح البولياني قيمة `false`، ثم طبعنا كل سطر وكررنا الحلقة مرةً أخرى، ثم أغلقنا الملف بعد الخروج من حلقة `while`. وإذا رغبنا في التوقف عند نقطة ما في الملف، فسيكون ذلك بشرط فرع branch condition داخل حلقة `while`، فإذا اكتشفنا شرط الإيقاف فسنستدعي `break` أيضًا لتنتهي الحلقة.

هذا كل ما يتعلق بفتح الملف وقراءته والتعامل معه بالطريقة التي تريدها مما سبق، لكن في المثال السابق ملاحظة صغيرة، فالأسطر المقروءة من الملف لها محرف سطر جديد في نهايتها، لذا سيكون لدينا أسطر فارغة إذا استخدمنا `print()` التي تضيف محرف السطر الجديد الخاص بها، ولتجنب هذا الأمر وفّرت بايثون تابع سلسلة نصية اسمه `strip()` يحذف المسافات البيضاء أو المحارف التي لا تُطبع من كلا طرفي السلسلة النصية، كما توجد توابع مثل `rstrip` و`lstrip` اللذين يُحذفان المسافات من جانب واحد فقط، وبناءً عليه نستطيع إصلاح مشكلة المسافات إذا استبدلنا السطر التالي بسطر `print()` السابق:

```
    print( line.rstrip() )  # احذف الجانب الأيمن فقط
    end
```

إذا أردنا إنشاء أمر نسخ في بايثون، فإننا نفتح ملفًا جديدًا في وضع الكتابة ثم نكتب الأسطر إليه بدلًا من طباعتها، سننشئ ملفًا اسمه `MENU.BAK` يكافئ الملف `MENU.TXT`:

```
# افتح الملفين للقراءة والكتابة على الترتيب
inp = open("menu.txt","r")
outp = open("menu.bak","w")
```

```
# اقرأ الملف ناسخًا كل سطر إلى الملف الجديد
for line in inp:
    outp.write(line)

print( "1 file copied..." )

# أغلق الملفات
inp.close()
outp.close()
```

لاحظ أننا أضفنا تعليمة `print()` في النهاية كي نطمئن المستخدم بحدوث شيء ما، لأن التغذية الراجعة للمستخدم تنفع ولا تضر، ولم نتعرض هنا لمشاكل مع محارف الأسطر الجديدة لأننا كتبنا نفس السطر `line` الذي قرأناه، لكن إذا كنا نكتب سلاسل نصيةً ننشئها بأنفسنا، أو سلاسل حذفنا منها محارف الأسطر الجديدة من قبل، فسيكون علينا إضافة سطر جديد إلى نهاية سلسلة الخرج، كما يلي:

```
outp.write(line + '\n') # \n => سطر جديد
```

لننظر في كيفية تطبيق هذا في برنامج النسخ. سنضيف تاريخ اليوم في الأعلى بدلًا من مجرد نسخ القائمة، وبهذا نستطيع إنشاء قائمة يومية من ملف نصي للوجبات يسهل تعديله، وكل ما علينا فعله هو كتابة بعض الأسطر في بداية الملف الجديد قبل نسخ ملف menu.txt، كما يلي:

```
import time
# أنشئ القائمة اليومية وفقًا لـ MENU.TXT
# افتح الملفات للقراءة والكتابة على الترتيب
inp = open("menu.txt","r")
outp = open("menu.prn","w")

# أنشئ سلسلة تاريخ اليوم
today = time.localtime(time.time())
theDate = time.strftime("%A %B %d", today)

# أضف نص الإعلان وسطرًا فارغًا
outp.write("Menu for %s\n\n" % theDate)

# انسخ كل سطر من menu.txt إلى ملف جديد
for line in inp:
    outp.write(line)
```

```
print( "Menu created for %s..." % theDate )

# أغلق الملفات
inp.close()
outp.close()
```

لاحظ أننا نستخدم وحدة time للحصول على تاريخ اليوم time.time()، وتحويله إلى صف tuple من القيم (time.localtime()) التي تُستخدم لاحقًا بواسطة time.strftime() -انظر توثيق الوقت والتاريخ في بايثون من موسوعة حسوب- لإنتاج سلسلة نصية تبدو بالشكل التالي عندما ندخلها في رسالة عنوان باستخدام تنسيق السلاسل النصية:

```
Menu for Sunday September 19

Spam & Eggs
Spam &...
```

لم يُطبع إلا سطر واحد فارغ رغم إضافتنا لمحرفين \n في نهاية السلسلة النصية، وذلك لأن أحدهما كان السطر الجديد عند نهاية العنوان نفسه، وهذا يظهر جانبًا مزعجًا في إدارة عملية إنشاء محارف الأسطر الجديدة وحذفها.

12.1.1 إلحاق البيانات

ثمة أمر آخر في معالجة الملفات، إذ قد نرغب في إلحاق بيانات بنهاية ملف موجود، ويمكن فعل هذا بفتح الملف للإدخال، وقراءة البيانات إلى قائمة، ثم إلحاق البيانات إليها، ثم كتابة القائمة كلها إلى إصدار جديد من الملف القديم، ولن يُحدث هذا مشكلةً إذا كان الملف قصيرًا؛ أما إذا كان كبيرًا -أكبر من 100 ميجا بايت مثلًا-، فستنفد الذاكرة التي يجب أن تحتفظ بالقائمة، وسيستغرق الأمر زمنًا طويلًا. لحسن الحظ لدينا وضع يسمى "a" والذي نستطيع تمريره إلى open()، فيسمح لنا بإلحاق البيانات مباشرةً إلى ملف موجود بمجرد كتابتها، وإذا لم يكن الملف موجودًا، فسيُفتح ملف جديد كما لو كنا قد حددنا الوضع "w".

لنفترض أن لدينا ملف سجل نستخدمه لالتقاط رسائل الخطأ، ولا نريد أن نحذف رسائل الخطأ الموجودة كلما جاءت رسالة جديدة، لذا يمكن إلحاق الخطأ في الملف كما يلي:

```
def logError(msg):
    err = open("Errors.log","a")
    err.write(msg)
    err.close()
```

قد نرغب -من الناحية العملية- بتقييد حجم الملف بشكل ما، والتقنية الشائعة لذلك هي إنشاء اسم ملف مبني على التاريخ، فإذا تغير التاريخ فسننشئ ملفًا جديدًا، مما يسهل على مشرفي النظام أن يجدوا أخطاء يوم بعينه، وأرشفة ملفات الأخطاء القديمة إذا لم نعد بحاجة إليها. تذكر أن وحدة `time` من مثال القائمة أعلاه يمكن استخدامها لإيجاد التاريخ الحالي.

12.1.2 بنية With في بايثون

قدم الإصدار الثالث من بايثون طريقةً جديدةً سهلةً للعمل مع الملفات، لا سيما عند التكرار على محتوياتها، وتستخدم هذه الطريقة بنيةً جديدةً اسمها `with`، بالشكل التالي:

```
with open('Errors.log',"r") as inp:
    for line in inp:
        print( line )
```

لاحظ أننا لم نستخدم `close()`، إذ تضمن `with` إغلاق الملف في نهايتها، وهكذا يكون التعامل مع الملفات أكثر موثوقيةً، وهذه الطريقة هي التي يُنصح بها لفتح الملفات في الإصدار الثالث من بايثون، وقد اخترنا استخدام الأسلوب القديم `open/close` لأن أغلب لغات البرمجة تستخدمه، وهو أكثر وضوحًا وصراحةً من أسلوب `with`، لكن إذا أردت أن تستخدم بايثون خاصةً فمن الأفضل استخدام `with`.

12.2 بعض العثرات في أنظمة التشغيل

تتعامل أنظمة التشغيل مع الملفات بطرق مختلفة، مما قد يسبب مشاكل في برامجنا إذا أردناها أن تعمل على عدة أنظمة تشغيل، ونخص بالذكر منها مشكلتين اثنتين:

12.2.1 الأسطر الجديدة

تشكل الملفات النصية والأسطر الجديدة منطقةً غامضةً تختلف فيها أنظمة التشغيل من حيث كيفية تنفيذها، وتمتد جذور تلك الاختلافات إلى الأيام الأولى لتواصل البيانات والتحكم في الآلات الكاتبة البرقية teleprinters الميكانيكية، وملخص الأمر أن لدينا ثلاثة طرق مختلفة للإشارة إلى السطر الجديد:

1. `\r`: محرف إعادة لأول السطر (CR: Carriage Return).
2. `\n`: محرف تغذية بداية السطر (LF: Line Feed).
3. `\r\n` زوج CR/LF.

تُستخدم التقنيات الثلاث في أنظمة تشغيل مختلفة، فنظام MS DOS مثلًا -وويندوز بالتبعية- يستخدم التقنية الثالثة، أما يونكس -بما في ذلك لينكس- فيستخدم الطريقة الثانية، بينما تستخدم أبل الطريقة الأولى في نظام ماك أو إس القديم، وتستخدم الطريقة الثانية في نظام MacOS X وما بعده بما أن هذا النظام ما هو إلا

يونكس. وعلى المبرمج أن يجري اختبارات كثيرةً ويتخذ إجراءات مختلفةً لكل نظام تشغيل، ليتعامل مع هذا التعدد لنهايات الأسطر. لكن اللغات الحديثة -بما في ذلك بايثون- توفر تسهيلات للتعامل مع هذه الفوضى بالنيابة عنا، وتحل بايثون هذه المشكلة في وحدة `os` التي تعرّف متغيرًا اسمه `linesep` يُضبط على محرف السطر الجديد الذي يستخدمه نظام التشغيل، مما يسهل عملية إضافة الأسطر الجديدة، كما ينتبه التابع `rstrip()` إلى نظام التشغيل عندما يحذف هذه المحارف، وهكذا نستخدم هذا التابع لنريح أنفسنا من عناء التفكير في الأسطر الجديدة التي تُحذف من الأسطر المقروءة من الملف، كما نضيف `os.linesep` إلى السلاسل النصية التي تُكتب إلى الملف.

فإذا أنشأنا ملفًا على نظام تشغيل ثم عالجناه على نظام تشغيل آخر غير متوافق مع الأول، فلا نستطيع إلا أن نوازن نهاية السطر مع `os.linesep` لنعرف الفرق.

12.2.2 تحديد المسارات

هذه المشكلة تخص مستخدمي ويندوز أكثر من غيرهم، فقد ذكرنا سابقًا أن كل نظام تشغيل يحدد مسارات الملفات باستخدام محارف مختلفة لفصل الأقراص والمجلدات والملفات عن بعضها، وأن الحل العام لهذا هو استخدام وحدة `os` التي توفر متغير `os.sep` لتعريف محرف فصل المسار الخاص بالمنصة الحالية، ولن نحتاج إلى ذلك كثيرًا في المواقف العملية، لأن المسار سيختلف لكل حاسوب على أي حال، لذا سنُدخِل المسار الكامل مباشرةً في سلسلة نصية -ربما سلسلة لكل نظام تشغيل نعمل عليه-، لكن لهذا عقبة كبيرة بالنسبة لمستخدمي ويندوز، فقد رأينا في القسم السابق أن بايثون تتعامل مع السلسلة `'\n'` على أنها محرف السطر الجديد، أي أنها تأخذ محرفين وتعاملهما مثل محرف واحد، ولدينا كثير من مثل هذه التسلسلات الخاصة من المحارف التي تبدأ بشرطة مائلة خلفية `\`، ومنها:

- `\n`: سطر جديد.
- `\r`: إعادة العربة.
- `\t`: جدول أفقي.
- `\v`: جدول رأسي، يعني أحيانًا صفحةً جديدة.
- `\b`: زر backspace.
- `\0nn`: أي شيفرة ثمانية عشوائية، مثل `\033` التي تشير إلى زر الهروب Esc.

فإذا كان لدينا ملف اسمه `test.dat` ونريد فتحه في بايثون من خلال تحديد مسار ويندوز كامل، فستكون الشيفرة:

```
>>> f = open('C:\test.dat')
```

لكن ما يحدث هو أن بايثون سترى الزوج `\t` على أنه محرف جدول وستخبرنا أنها لا تستطيع إيجاد ملف باسم `C:  est.dat`، ولحل هذه المشكلة لدينا ثلاث طرق، هي:

1. نضع `r` أمام السلسلة النصية، لنخبر بايثون أن تتجاهل أي شرطة مائلة خلفية، وتعاملها على أنها سلسلة نصية خام.
2. نستخدم شرطات مائلة أمامية `/` بدلًا من الخلفية، وستتوافق بايثون مع ويندوز ليخرجا لنا المسار، وهذا الحل يجعل الشيفرة قابلةً للعمل على أنظمة التشغيل الأخرى.
3. استخدام شرطتين خلفيتين `\\` بما أن بايثون ترى محرفي الشرطتين المزدوجتين على أنهما شرطة خلفية واحدة.

وعلى ذلك سيفتح أي سطر مما يلي ملف البيانات الخاص بنا بشكل سليم:

```
>>> f = open(r'C:\test.dat')
>>> f = open('C:/test.dat')
>>> f = open('C:\\test.dat')
```

لاحظ أن هذه المشكلة مقصورة على السلاسل المجردة التي نكتبها في شيفرة برنامجنا، أما إذا قرئت سلاسل المسار من ملف أو من المستخدم، فستفسر بايثون محارف `\` وتستخدمها كما هي دون مشاكل.

12.3 دليل جهات الاتصال

لقد كتبنا دليل جهات اتصال في الفصل الخامس: البيانات وأنواعها، ثم زدنا عليه وطورناه في الفصل التاسع: قراءة المدخلات من المستخدم، وسنجعله في هذا الفصل تطبيقًا مفيدًا بحفظه في ملف، إلى جانب قراءة ذلك الملف عند بدء التشغيل، وبما أننا سنفعل ذلك من خلال كتابة بعض الدوال، فسنستخدم بعضًا مما شرحناه في الفصول السابقة. سيحتاج التصميم الأولي دالةً لقراءة الملف عند بدء التشغيل، ودالةً أخرى عند نهاية البرنامج، كما سننشئ دالةً تزود المستخدم بقائمة من الخيارات، ودالةً مستقلة لكل خيار في تلك القائمة، ستسمح القائمة للمستخدم بما يلي:

- إضافة مدخل في دليل جهات الاتصال.
- حذف مدخل منه.
- البحث عن مدخل موجود من قبل وعرضه.
- الخروج من البرنامج.

12.3.1 تحميل دليل جهات الاتصال

```
import os
filename = "addbook.dat"

def readBook(book):
    if os.path.exists(filename):
       with open(filename,'r') as store:
          for line in store:
             name = line.rstrip()
             entry = next(store).rstrip()
             book = entry
```

نلاحظ هنا أننا نستورد الوحدة `os` التي نستخدمها للتحقق من أن مسار الملف موجـود قبـل فتحـه، ونعـرِّف اسم الملف مثل متغير مستوى وحدة module level variable لنستخدمه في تحميل البيانات وحفظها.

سنستخدم أيضًا `rstrip()` لحذف محرف السطر الجديـد من نهايـة السـطر، ودالـة `next()` الـتي تجلب السطر التالي من الملف إلى داخل الحلقة، وهذا يعني أننا نقرأ سطرين في نفس الوقت أثناء عمل الحلقة، ودالة `next` هي جزء من خاصية في بـايثون تسـمى بـالمكرر، وهي خاصـية لن نشـرحها في الكتـاب بمـا أنهـا خاصـة ببايثون كلغة برمجة، لكننا سنقول أن كل تجميعات بايثون وملفاتها وبعض الأمـور الأخـرى يُنظـر إليهـا على أنهـا مكرَّرات أو أنواع قابلة للتكرار، ويمكن معرفة المزيد عن هذه الخاصية في توثيق بايثون.

12.3.2 حفظ دليل جهات الاتصال

```
def saveBook(book):
    with open(filename, 'w') as store:
       for name,entry in book.items():
         store.write(name + '\n')
         store.write(entry + '\n')
```

لاحظ أننا نحتاج إلى إضافة محرف السطر الجديـد `'\n'` عنـدما نكتب البيانـات، وأننـا نكتب سـطرين لكـل إدخال، فهذا يعكس حقيقة أننا عالجنا سطرين عند قراءة الملف.

12.3.3 الحصول على مدخلات المستخدم

```
def getChoice(menu, length):
    print( menu )
    prompt = "Select a choice(1-%d): "  % length
    choice = int( input(prompt) )
    return choice
```

نلاحظ أننا نستقبل معامِل طول يخبرنا عدد المداخل الموجودة، مما يسمح لنا بإنشاء محث يحدد نطاق الأعداد المناسب.

12.3.4 إضافة مدخل

```
def addEntry(book):
    name =  input("Enter a name: ")
    entry = input("Enter street, town and phone number: ")
    book = entry
```

12.3.5 حذف مدخل

```
def removeEntry(book):
    name = input("Enter a name: ")
    del(book)
```

12.3.6 العثور على مدخل

```
def findEntry(book):
    name = input("Enter a name: ")
    if name in book:
       print( name, book )
    else: print( "Sorry, no entry for: ", name )
```

12.3.7 الخروج من البرنامج

لن نكتب دالةً مستقلةً للخروج من البرنامج، بل سنجعل خيار الإنهاء اختبارًا في حلقة `while` الخاصة بالقائمة، لذا سيكون البرنامج الرئيسي كما يلي:

```
def main():
    theMenu = '''
```

```
إضافة مدخل
حذف مدخل
العثور على مدخل
الخروج والحفظ
    '''
    theBook = {}
    readBook(theBook)
    while True:
        choice = getChoice(theMenu, 4)
        if choice == 4: break

        if choice == 1:
            addEntry(theBook)
        elif choice == 2:
            removeEntry(theBook)
        elif choice == 3:
            findEntry(theBook)
        else: print( "Invalid choice, try again" )
    saveBook(theBook)
```

لم يبق الآن إلا استدعاء `main()` عند تشغيل البرنامج، وهنا سنستخدم القليل من سحر بايثون:

```
if __name__ == "__main__":
    main()
```

تسمح لنا هذه الشيفرة الغامضة باستخدام أي ملف بايثون مثل وحدة، وذلك باستيراده `import` أو مثل برنامج عبر تشغيله، والفرق هنا هو أن بايثون تضبط المتغير الداخلي `__name__` عند استيراد البرنامج على اسم الوحدة، لكن إذا شغّلنا الملف مثل برنامج، فستُضبط قيمة `__name__` على `"__main__"`، هذا يعني أن دالة `main()` لا تُستدعى إلا إذا شغلنا الملف مثل برنامج وليس عند استيراده.

إذا كتبنا هذه الشيفرة في ملف نصي وحفظناه باسم addressbook.py، فيجب أن يكون قابلًا للتشغيل في أي محث لنظام تشغيل بكتابة ما يلي:

```
C:\PROJECTS> python addressbook.py
```

أو بالنقر المزدوج عليه مثل أي ملف في مدير الملفات في ويندوز، ليشغَّل في نافذة CMD خاصة به، تُغلَق عند تحديد خيار الإغلاق، أو باستخدام ما يلي في لينكس:

```
$ python addressbook.py
```

يُعَد هذا البرنامج المكون من ستين سطرًا نموذجًا لما يجب أن تكون قادرًا على كتابته الآن بنفسك، وهو أداة مفيدة على حالته تلك، رغم أننا سنضيف إليه بعض الأشياء التي ستحسّنه في الفصل التالي.

12.4 جافاسكربت وVBScript

ليس لدى جافاسكربت وVBScript القدرة على معالجة الملفات، وذلك لمنع أي أحد من قراءة ملفاتك عند تحميلك لصفحة ويب مثلًا، لكن هذا يقيد فائدة اللغة نفسها ونطاق استخدامها من ناحية أخرى. توجد طريقة لجعلهما تعالجان الملفات باستخدام WSH كما فعلنا في الوحدات القابلة لإعادة الاستخدام من قبل، إذ توفر WSH كائن `FileSystem` يسمح لأي لغة WSH بقراءة الملفات. سننظر في مثال جافاسكربت بالتفصيل، ثم نعرض شيفرةً مشابهةً من لغة VBScript من أجل الموازنة بينهما، وكما رأينا من قبل فإن العناصر الأساسية ما هي إلا استدعاءات لكائنات WScript.

لربما يجب أن نشرح نموذج الكائن `FileSystem` قبل أن ننظر في الشيفرة، فنموذج الكائن هو مجموعة من الكائنات المرتبطة ببعضها، والتي يمكن للمبرمج استخدامها. يتكون نموذج كائن `FileSystem` من كائن `FSO` وعدد من كائنات `File`، بما في ذلك كائن `TextFile` الذي سنستخدمه، كما توجد بعض الكائنات المساعدة مثل `TextStream`.

ما سنفعله هنا هو إنشاء نسخة من كائن `FSO` ثم نستخدمها لإنشاء كائنات `TextFile`، ثم ننشئ كائنات `TextStream` كي نستطيع قراءة النصوص فيها وكتابتها أيضًا، كذلك فكائنات `TextStream` نفسها هي التي نقرؤها من الملفات أو نكتبها.

اكتب الشيفرة أدناه في ملف باسم `testFiles.js` وشغله باستخدام `cscript` كما ذكرنا في قسم WSH من الفصل السابق.

12.4.1 فتح ملف

يجب أن ننشئ كائن FSO أولًا، ثم ننشئ كائن TextFile منه كي نتمكن من فتح ملف في WSH:

```
var fileName, fso, txtFile, outFile, line;

// احصل على اسم الملف
fso = new ActiveXObject("Scripting.FileSystemObject");
WScript.Echo("What file name? ");
fileName = WScript.StdIn.Readline();
```

```
// افتح inFile للقراءة
// افتح outFile للكتابة
inFile = fso.OpenTextFile(fileName, 1); // mode 1 = Read
fileName = fileName + ".BAK"
outFile = fso.CreateTextFile(fileName);
```

12.4.2 إغلاق الملفات

```
inFile.close();
outFile.close();
```

12.4.3 شرح المثال في VBScript

احفظ ما يلي في `testFiles.ws` وشغله باستخدام:

```
cscript testfiles.ws
```

أو ضع الجزء الذي بين وسوم `script` في ملف باسم `testFile.vbs` وشغله، حيث تسمح صـيغة `.ws` بدمج شيفرة جافاسكربت وVBScript في نفس الملف باستخدام عدة وسوم `script`:

```
<?xml version="1.0"?>

<job>
  <script type="text/vbscript">
      Dim fso, inFile, outFile, inFileName, outFileName
      Set fso = CreateObject("Scripting.FileSystemObject")

      WScript.Echo "Type a filename to backup"
      inFileName = WScript.StdIn.ReadLine
      outFileName = inFileName &amp;amp; ".BAK"

      ' open the files
      Set inFile = fso.OpenTextFile(inFileName, 1)
      Set outFile = fso.CreateTextFile(outFileName)

      ' read the file and write to the backup copy
      Do While not inFile.AtEndOfStream
         line = inFile.ReadLine
```

```
            outFile.WriteLine(line)
        Loop

        ' close both files
        inFile.Close
        outFile.Close

        WScript.Echo inFileName &amp; " backed up to " &amp; outFileName
    </script>
</job>
```

12.5 التعامل مع الملفات غير النصية

يُعَد التعامل مع الملفات النصية أكثر ما يفعله المبرمج، غير أننا قد نحتاج إلى معالجة بيانات ثنائيـة أحيانًـا، وسنشرح هذا الجزء في بايثون فقط لندرة حدوثه في جافاسكربت وVBScript.

12.5.1 الترميز الثنائي للبيانات

يجب أن ننظر في كيفية تمثيل البيانات وتخزينها على الحاسوب قبل أن نتحـدث عن كيفيـة الوصـول إليهـا داخل ملف ثنائي، حيث تُخزَّن البيانـات في هيئـة تسلسـلات من الأرقـام الثنائيـة أو البتـات، والـتي تُجمـع في مجموعات من 8 بتات تسمى البايت، أو 16بتًا تسمى الكلمة، في حين أن المجموعـة الـتي تتكـون من 4 بتـات تسمى أحيانًا بالحلمة. قد يكون للبايت الواحد نمط من 256 نمط للبتـات، وتعطى هـذه الأنمـاط قيمًـا من 0 إلى 255، كما يجب أن تُحوَّل المعلومات التي نعالجها في برامجنا من سلاسل نصية وأعداد وغيرها إلى سلاسـل من تلك البايتات، لذا يخصَّص نمط بايت معين لكل محرف نستخدمه في السلاسل النصـية، ورغم وجـود عـدة نظم ترميز للبيانات، إلا أن أشهرها هو ترميز آسكي ASCII، لكن هذا الترميز لا يحوي إلا 128 محرفًا فقط، وهذا بالكـاد يكفي للغة الإنجليزية فقط، لذا طوِّر معيار ترمـيز جديـد سـمي بـالترميز الموحـد أو اليونيكـود Unicode، والـذي يَستخدم كلمات البيانات لتمثيل المحارف بدلًا من البايتات، ويشتمل على أكثر من مليون محرف، ثم يمكن بعـد ذلك ترميز تلك المحارف في مجرى بيانات مضغوط أكثر.

إن أكثر ترميزات اليونيكود شيوعًا هو UTF-8، ويتوافق توافقًا كبيرًا مع آسـكي، بحيث أن أي ملـف متوافـق مع آسكي يتوافق أيضًا مع UTF-8، رغم أن العكس قد لا يكون صحيحًا. يوفر اليونيكود عددًا من الترميزات الـتي يعرِّف كل منها البايت الذي يمثل قيمةً عدديةً من اليونيكود، أو "نقطـةً رمزيـةً" وفـق اصـطلاح اليونيكـود. وهـذا النظام المعقد هو الثمن الذي كان يجب دفعه من أجل بناء شبكة حواسيب عالمية تعمل بمختلف اللغـات الـتي يستخدمها البشر، لكن المتحدثين بالإنجليزية لم يكن عليهم القلـق بشـأن اليونيكـود إلا عنـد قـراءة البيانـات من

ملف ثنائي، على الرغم من أن الإصدار الثالث لبايثون يَعُد السلاسل النصية سلاسل يونيكـود، لـذا يجب أن نعلم الترميز المستخدم من أجل تفسير البيانات الثنائية تفسيرًا صحيحًا.

تدعم بايثون نصوص اليونيكود دعمًا كاملًا، فتنظر إلى سلسلة المحارف المرمَّزة على أنها سلسلة بايتات لهـا النوع `bytes`، بينما تكون السلسلة غير المرمزة من النوع `str`، ويكـون ترميزهـا الافتراضـي هـو UTF-8، ولهـذه القاعدة بعض الاسـتثناءات -نظريًـا على الأقـل-، ولن نشـرح اسـتخدام المحـارف الـتي ليسـت UTF-8 في هـذا الكتاب، لكن يمكن مراجعة مستند How-To في موقع بايثون.

ما نريد الإشارة إليـه من كـل هـذا هـو أن المجـرى الثنـائي لنص اليونيكـود المرمَّز يعامَـل مثـل سلسـلة من البايتات، وتوفر بايثون دوالًا لتحويل (أو فك ترميز) قيم `bytes` لتكون قيم `str`، بالمثل يجب أن تحـوَّل الأعـداد إلى ترميزات ثنائية أيضًا، فعلى الرغم من أن قيم البايتات تكفي في حالة الأعـداد الصـغيرة، إلا أن الأعـداد الأكـبر من 255 أو الأعداد السالبة أو الكسور تحتاج إلى عمل آخر، وقد ظهرت عدة ترميزات معياريـة للبيانـات العدديـة، والـتي تسـتخدمها أغلب لغـات البرمجـة ونظم التشـغيل، فمثلًا: يعـرِّف المعهـد الأمـريكي للهندسـة الكهربيـة والإلكترونية IEEE عدة ترميزات للأعداد ذات الفاصلة العائمة floating point numbers، وتهدف هذه الجهود إلى حل مشكلة التفسير الصحيح للبيانات الثنائية، فمن المهم للغاية أن نحولها إلى النوع الصحيح عنـد قراءتهـا، بما أنه يتعين علينا تفسير أنماط البتات الخام إلى النوع الصحيح المناسب للبرنامج الذي نعمل عليه، وعلى ذلـك من الممكن أن نفسـر مجـرى بيانـات كُتب أصـلا على أنـه سلسـلة نصـية في صـورة مجموعـة من الأعـداد ذات الفاصلة العائمة، ورغم أن المعنى الأصلي له سيُفقد، إلا أن أنماط البتات يمكن أن تمثل أي واحد منهما.

12.5.2 فتح الملفات الثنائية وإغلاقها

يتمثل الفرق الجوهري بين الملفات النصية والثنائية في أن الملفـات النصـية تتكـون من ثمانيـات octets -جمع ثُماني، وهو الشيء المكون من ثمانية أجزاء-، لكن الاسم الأشهر لها هـو بايتـات، ويمثـل كـل بـايت حرفًـا. تُحدَّد نهاية الملف بنمط بايت خاص يُعرف باسم EOF، وهو اختصار لعبارة نهايـة الملـف End Of File؛ بينمـا يحتوي الملف الثنائي على بيانات ثنائية عشوائية، ومن ثم لا يمكن استخدام قيمة بعينها لتحديـد نهايـة الملـف، لذا نحتاج إلى وضع عمليات مختلف لقراءة تلك الملفات، فعند فتح ملف ثنائي في بايثون أو في أي لغة أخـرى، فيجب أن نحدد أنه يُفتَح في الوضع الثنائي، أو المخاطرة بقطع البيانات التي نقرؤها عند أول محـرف `eof` تجـده بـايثون في تلـك البيانـات. يمكن تنفيـذ ذلـك في بـايثون بإضـافة `b` إلى معامِـل الوضـع mode parameter كما يلي:

```
binfile = open("aBinaryFile.bin","rb")
```

لا يختلف هذا عن فتح ملف نصي إلا في قيمة الوضع `"rb"`، ونسـتطيع اسـتخدام أي وضـع آخـر، لكن مـع إضـافة حـرف `b`، فنسـتخدم `"wb"` للكتابـة، و`"ab"` للإلحـاق، كمـا لا يختلـف إغلاق الملـف الثنـائي عن الملـف النصي، فنستدعي التابع `close()` لكائن الملف المفتوح:

```
binfile.close()
```

وبما أننا فتحنا الملف في الوضع الثنائي، فلا داعي لإعطاء بايثون أي معلومات إضافية، فهي تعرف كيف تغلق الملف.

12.5.3 الوحدة struct

توفّر بايثون وحدةً اسمها `struct` لترميز البيانات الثنائية وفك ترميزها، وهي تتصرف مثل سلاسل التنسيق التي استخدمناها لطباعة البيانات المختلطة، إذ توفر سلسلةً تمثل البيانات التي نقرؤها وتطبقها على مجرى البايتات الذي نحاول تفسيره، ومن الممكن استخدام هذه الوحدة لتحويل مجموعة من البيانات إلى مجرى البايت للكتابة، إما إلى ملف ثنائي أو حتى خط اتصالات communications line.

هناك الكثير من رموز تنسيق التحويل المختلفة، لكننا لن نستخدم إلا رموز الأعداد الصحيحة integers والسلاسل النصية strings هنا؛ أما البقية فيمكن قراءة المزيد عنها في توثيق بايثون الرسمي. إن رمز تنسيق التحويل للأعداد الصحيحة هو `i`، ورمز السلاسل النصية `s`، وتتكون سلاسل تنسيق الوحدة `struct` من سلاسل من الرموز فيها أرقام مسبقة التعليق pre-pended، حيث توضح عدد العناصر التي نحتاج إليها، مع استثناء لرمز `s` الذي يشير العدد مسبق التعليق فيه إلى طول السلسلة النصية، حيث تعني `4s` مثلًا سلسلةً من أربعة محارف (أربعة محارف وليس أربعة سلاسل نصية).

لنفترض أننا نريد كتابة تفاصيل العنوان، في برنامج دليل جهات الاتصال الذي شرحناه أعلاه، في صورة بيانات ثنائية، حيث يكون رقم الشارع فيها عددًا صحيحًا والبقية سلسلةً نصيةً. رغم أن هذا الاختيار سيء لأن أرقام الشوارع تتضمن حروفًا أحيانًا، إلا أن سلسلة التنسيق ستكون كما يلي:

```
'i34s' # نفترض وجود 34 محرف في العنوان
```

ولكي نعالج مسألة الأطوال المختلفة للعناوين، فسنكتب دالةً تنشئ السلسلة الثنائية كما يلي:

```
def formatAddress(address):
    fields = address.split()
    number = int(fields[0])
    rest = bytes(' '.join(fields[1:],'utf8') # أنشئ سلسلة بايت
    format = "i%ds" % len(rest) # أنشئ سلسلة التنسيق
    return struct.pack(format, number, rest)
```

لقد استخدمنا تابع السلسلة `split()` أعلاه لتقسيم سلسلة العنوان النصية إلى أجزائها، وقسمناها في حالتنا إلى قائمة من الكلمات، ثم استخرجنا الجزء الأول ليكون رقم الشارع، بعد ذلك استخدمنا تابع سلسلةٍ نصية آخر هو `join` لدمج الحقول المتبقية معًا مع الفصل بينها بمسافة. كذلك نحتاج إلى تحويل السلسلة النصية إلى مصفوفة `bytes` لأن هذا هو ما تستخدمه وحدة `srtuct`، ويكون طول تلك السلسلة هو ما نحتاج

إليه في سلسلة تنسيق `struct`، لذا نستخدم دالة `len()` بالتوازي مع سلسلة تنسيق عادية لبناء سلسلة تنسيق `struct`.

ستعيد `formatAddress()` تسلسلًا من البايتات يحتوي على التمثيل الثنائي لعناوننا، وبما أن لدينا البيانات الثنائية فسنرى كيف نستطيع كتابتها إلى ملف ثنائي، ثم قراءتها مرةً أخرى.

12.5.4 القراءة والكتابة باستخدام struct

لننشئ ملفًا ثنائيًا يحتوي على سطر عنوان واحد باستخدام الدالة `formatAddress()` المعرَّفة أعلاه، وهنا سنحتاج إلى فتح الملف للكتابة في وضع `'wb'`، وترميز البيانات وكتابتها إلى الملف، ثم إغلاق الملف.

```
import struct
f = open('address.bin','wb')
data = "10 Some St, Anytown, 0171 234 8765"
bindata = formatAddress(data)
print( "Binary data before saving: ", repr(bindata) )
f.write(bindata)
f.close()
```

نستطيع التحقق من أن البيانات في صورة ثنائية بفتح `address.bin` في محرر نصي، وسنرى حينها أن المحارف لا زالت قابلةً للقراءة لكن الرقم اختفى، فإذا فتحنا الملف في محرر نصي يدعم الملفات الثنائية مثل ViM أو Emacs، فسنجد أن بداية الملف فيها 4 بايت، حيث سيبدو الأول منها مثل محرف سطر جديد، أما البقية فتبدو أصفارًا، وذلك لأن القيمة العددية للسطر الجديد هي 10، كما نرى هنا باستخدام بايثون:

```
>>> ord('\n')
10
```

تعيد الدالة `ord()` في هذا المثال القيمة العددية لأي محرف، لذا فإن أول أربعة بايتات هي `10,0,0,0` في النظام العشري، أو `0xA,0x0,0x0,0x0` في النظام الست عشري، وهو النظام المستخدم عادةً في عرض البيانات الثنائية بما أنه أكثر اختصارًا من استخدام الأرقام الثنائية الخالصة.

يُؤخذ العدد الصحيح في حاسوب ذي معمارية 32 بت أربعة بايتات، لذا فإن القيمة العددية "10" قد حُولت بواسطة وحدة `struct` إلى تسلسل من 4 بايتات هو `10,0,0,0`. وتضع معالجات إنتل البايت الأقل أهميةً في البداية، وهنا سنحصل على القيمة الثنائية الحقيقية من خلال قراءة التسلسل عكسيًا `0,0,0,10`، وهي القيمة الصحيحة 10 ممثلةً بأربعة بايتات عشرية؛ أما بقية البيانات فهي السلسلة النصية الأصلية، لذا تظهر بصيغتها المحرفية العادية.

يجب الانتباه إلى عدم حفظ الملف من داخل محرر Notepad، فرغم أنه يستطيع تحميل بعض الملفات الثنائية، إلا أنه لا يستطيع حفظ الملفات الثنائية، لذا سيحول القيم الثنائية إلى نصوص، مما سيؤدي إلى تخريب البيانات الحقيقية.

لم يكن امتداد الملف الذي استخدمناه `.bin` إلا للشرح، إذ ليس له تأثير حقيقي في كون الملف نصيًا أم ثنائيًا، وتستخدم بعض أنظمة التشغيل هذه الامتدادات لتحديد البرنامج الذي يفتح الملفات، لكن يمكن تغيير الامتداد بإعادة تسمية الملف ببساطة، ولن يتغير المحتوى الذي فيه، إذ سيظل ثنائيًا أو نصيًا كما كان، فإذا أعدنا تسمية ملف نصي بحيث نجعل امتداده `.exe` مثلًا، فسيعامله ويندوز على أنه ملف ثنائي تنفيذي، لكننا سنحصل على خطأ عند محاولة تشغيله مثل ملف تنفيذي، وذلك لأن النص الموجود فيه ليس شيفرةً ثنائيةً تنفيذية، فإذا أرجعناه إلى التسمية الأصلية له ثم فتحناه في Notepad، فسنجده كما كان قبل التسمية بالضبط، بل لو فتحناه في Notepad أثناء تسميته بامتداد `.exe` لفتحه المحرر مثل ملف نصي أيضًا.

نحتاج إلى فتح الملف في وضع `rb` لقراءة بياناتنا الثنائية مرةً أخرى، فنقرأ البيانات في تسلسل من البايتات ثم نغلق الملف ونفك ترميز البيانات باستخدام سلسلة تنسيق `struct`، سيبقى السؤال هنا حول كيفية معرفة شكل سلسلة التنسيق، والإجابة هنا هي أننا نحتاج إلى إيجاد السلسلة الثنائية من تعريف الملف، وتوفر عدة مواقع على الويب هذه المعلومات، فشركة أدوبي مثلًا تنشر تعريف تنسيقها الثنائي لصيغة PDF الخاصة بها، وفي حالتنا هذه سنعرف أنها يجب أن تكون السلسلة التي أنشأناها في `formatAddress()`، وهي `'iNs'`، حيث N رقم متغير.

توفر وحدة `struct` بعض الدوال المساعدة التي تعيد حجم كل نوع من أنواع البيانات، لذلك فإن شغلنا محث بايثون وأجرينا بعض التجارب، فسنستطيع معرفة عدد بايتات البيانات التي سنحصل عليها لكل نوع بيانات، وهكذا نستطيع تحديد قيمة N:

```
>>> import struct
>>> print struct.calcsize('i')
4
>>> print struct.calcsize('s')
1
```

وبما أننا نعرف أن بياناتنا ستترك 4 بايتات للعدد وبايتًا واحدًا لكل محرف، فستكون N هي إجمالي طول البيانات ناقصًا منه 4، لنجرب استخدام هذا لقراءة ملفنا:

```
import struct
f = open('address.bin','rb')
data = f.read()
f.close()
```

```
fmtString = "i%ds" % (len(data) - 4)
number, rest = struct.unpack(fmtString, data)
rest = rest.decode('utf8') #convert bytes to string
address = ' '.join((str(number),rest))

print( "Address after restoring data:", address )
```

لاحظ أننا اضطررنا إلى تحويل `rest` إلى سلسلة نصية باستخدام دالة `decode()` لأن بايثون رأتها من النوع `bytes` الذي لن يعمل مع `join()`.

وهكذا نكون قد شرحنا ملفات البيانات الثنائية بما يكفي في هذا الفصل، وقد بينا أنها تأتي بعدة تعقيدات، ولا ننصح باستخدامها ما لم يكن ثمة سبب مقنع، لكن هذا الشرح كاف لتستطيع قراءة الملفات الثنائية عند الحاجة إلى ذلك، طالما عرفت ما هي البيانات الممثلة في تلك الملفات ابتداءً.

12.6 الوصول العشوائي إلى الملفات

يشير الوصول العشوائي إلى التحرك مباشرةً إلى جزء بعينه من الملف، دون قراءة البيانات التي بين نقطة البدء والبيانات المطلوبة، وتوفر بعض لغات البرمجة نوع ملف مفهرس خاص يستطيع فعل ذلك بسرعة عالية، لكنه مبني في أغلب اللغات على الوصول المتتابع للملفات، الذي كنا نستخدمه منذ بداية الكتاب إلى الآن.

يقوم الوصول العشوائي على مفهوم مؤشر يشير إلى الموضع الحالي في الملف، أي عدد البايتات التي تفصلنا عن البداية حرفيًا، ونستطيع تحريك هذا المؤشر بالنسبة إلى موضعه الحالي أو نسبةً إلى بداية الملف، كما نستطيع أن نطلب من الملف أن يخبرنا بالمكان الحالي للمؤشر.

ونستخدم طول سطر ثابت -ربما بحشو سلاسل البيانات الخاصة بنا بمسافات أو بعض المحارف الأخرى عند الحاجة- كي نقفز إلى بداية سطر ما، من خلال ضرب طول السطر بعدد الأسطر، وهذا ما يوحي بالوصول العشوائي للبيانات في الملف.

12.6.1 أين أنا؟

لتحديد مكاننا في ملف ما نستخدم التابع `tell()` الخاص بالملف، فإذا فتحنا ملفًا وقرأنا ثلاثة أسطر، فعندها سنستطيع أن نسأل الملف حينها كم قرأنا من الملف حتى الآن.

لننظر في مثال ليتضح المعنى، حيث سننشئ ملفًا بخمسة أسطر نصية لها نفس الطول -وتساوي الطول هنا ليس ضروريًا وإنما هو لتوضيح المثال-، بعدها سنقرأ ثلاثة أسطر ونسأل أين نحن، ثم نعود إلى البداية ونقرأ سطرًا واحدًا ونقفز إلى السطر الثالث ونطبعه، ثم نعود إلى السطر الثاني، كما يلي:

```
# (+ \n) أنشئ 5 أسطر من عشرين محرفًا
testfile = open('testfile.txt','w')
for i in range(5):
    testfile.write(str(i) * 20 + '\n')
testfile.close()

# اقرأ ثلاثة أسطر واسأل أين نحن
testfile = open('testfile.txt','r')
for line in range(3):
    print( testfile.readline().strip() )
position = testfile.tell()
print( "At position: ", position, "bytes" )

# عد إلى البداية
testfile.seek(0)
print( testfile.readline().strip() ) # كرر السطر الأول
lineLength = testfile.tell()
testfile.seek(2*lineLength)  # اذهب إلى نهاية السطر 2
print( testfile.readline().strip() ) # السطر الثالث
testfile.close()
```

لقد استخدمنا الدالة `seek()` لتحريك المؤشر، والعملية الافتراضية هنا هي تحريكه إلى رقم البايت المحدد كما رأينا هنا، لكن يمكن إضافة وسطاء آخرين لتغيير تابع الفهرسة المستخدم.

لاحظ أيضًا أن القيمة التي طبعتها `tell()` الأولى تعتمد على طول السطر الجديد على نظامك، فنظام ويندوز 10 مثلًا يطبع 66 ليشير إلى أن تسلسل السطر الجديد طوله 2 بايت، لكن بما أن هذه القيمة تتوقف على نظام التشغيل ونحن نريد أن نجعل الشيفرة محمولةً قدر الإمكان؛ فقد استخدمنا `tell()` مرةً ثانيةً بعد قراءة سطر واحد لحساب طول كل سطر، وسترى أن مثل تلك الحيل ضرورية عند التعامل مع مشاكل المنصات المختلفة.

12.7 خاتمة

في نهاية هذا الفصل نرجو أن تكون تعلمت ما يلي:

- ضرورة فتح الملفات قبل استخدامها.
- قراءة الملفات أو الكتابة فيها، إذ لا يمكن تنفيذ العمليتين معًا.

- الدالــة `readlines()` في بــايثون الــتي تقــرأ جميــع الأســطر الموجــودة في ملــف مــا، والدالــة `readline()` التي تقرأ سطرًا واحدًا في كل مرة، وهذا يوفر في الذاكرة.
- عدم الحاجة إلى اسـتخدام أي من الـدالتين السـابقتين بمـا أن دالـة `open` الخاصـة ببـايثون تعمـل مـع حلقات `for`.
- ضرورة غلق الملفات بعد استخدامها.
- ضرورة إضافة `b` إلى نهاية رايـة الوضـع mode flag الخاصـة بالملفـات الثنائيـة، وتحتـاج البيانـات إلى تفسير بعد قراءتها، وذلك بواسطة وحدة `struct` عادةً.
- تفعّل كل من `tell()` و`seek()` الوصول شبه العشوائي pseudo-random إلى الملفات متسلسـلة الوصول sequential files.

13. التعامل مع النصوص

لا شك أن التعامل مع النصوص هو أكثر ما يفعله المبرمج في عمله لأنه يبرمج بلغات تتكون من كلمات ورموز، وهي نصوص في النهاية، لذلك توجد أدوات كثيرة جدًا في أغلب لغات البرمجة لتسهيل التعامل مع هذه النصوص، وسننظر في كيفية استخدام تلك الأدوات في تنفيذ المهام البرمجية المعتادة التي تشمل ما يلي:

- تقسيم الأسطر إلى مجموعات من المحارف.
- البحث عن سلاسل نصية داخل سلاسل أخرى.
- استبدال سلسلة نصية بأخرى.
- تغيير حالة الأحرف.

سننظر في كيفية تنفيذ كل مهمة من هذه المهام باستخدام بايثون، ثم نمر عليها سريعًا في جافاسكربت وVBScript، وتُستخدم توابع السلاسل النصية في بايثون للتعامل مع السلاسل النصية، فلعلك تذكر من الفصل الخامس: البيانات وأنواعها أن التوابع تشبه الدوال المرتبطة ببيانات، ونستطيع الوصول إلى التوابع باستخدام نفس الترميز النقطي dot notation الذي نستخدمه للوصول إلى الدوال في الوحدات modules، لكننا سنستخدم البيانات نفسها بدلًا من استخدام اسم الوحدة. لننظر الآن في ذلك.

13.1 تقسيم السلاسل النصية

أولًا سنقسم سلسلةً نصيةً إلى عدة أجزاء، والذي نحتاج إليه عند معالجة الملفات لأننا نقرأ الملف سطرًا سطرًا، لكن قد يحتوي جزء من السطر فقط على البيانات المطلوبة. من أمثلة ذلك برنامج دليل جهات الاتصال الذي نعود إليه كل حين، فقد نرغب في الوصول إلى حقول بعينها من مدخل ما، دون الحاجة لطباعة المدخل كله، وسنستخدم لهذا الغرض التابع `split()` في بايثون كما يلي:

```
>>> aString = "Here is a (short) String"
>>> print( aString.split() )
['Here', 'is', 'a', '(short)', 'String']
```

لاحظ كيف حصلنا على قائمة تحتوي الكلمات التي في `aString` مع حذف جميع المسافات، لأن الفاصل الافتراضي للدالة `''.split()` هو المسافة البيضاء، سواءً كانت سطرًا جديدًا، أم مسافةً عادية، أم مسافة جدول tab. لنجرب الآن استخدامه مرةً أخرى بجعل الفاصل قوسًا افتتاحيًا:

```
>>> print( aString.split('(') )
['Here is a ', 'short) String']
```

يكمن الفرق هنا في أننا حصلنا على عنصرين فقط في القائمة، وقد حُذف القوس الافتتاحي من بداية `'short)'`، وهذه ملاحظة مهمة حول `''.split()`، وهي حذفه للمحارف الفاصلة، الذي نريده غالبًا مع استثناءات قليلة.

كذلك لدينا التابع `''.join()` الذي يأخذ قائمةً -أو أي نوع آخر- من التسلسلات النصية ويدمجها معًا، لكن له خاصية قد تسبب حيرةً عند استخدامه، وهو استخدامه السلسلة التي نستدعي التابع عليها محرفًا للدمج، كما يلي:

```
>>> lst = ['here','is','a','list','of','words']
>>> print( '-+-'.join(lst) )
here-+-is-+-a-+-list-+-of-+-words
>>> print( ' '.join(lst) )
here is a list of words
```

رغم منطقية هذا السلوك، إلا أنه يبدو غريبًا عند رؤيته لأول مرة، كما أنه سلوك مناقض لما هو موجود في جافاسكربت التي تحوي تابع مصفوفة اسمه `join`، وتكون السلسلة الدامجة فيه معامِلًا.

13.1.1 عد الكلمات

سنعيد النظر مرةً أخرى في برنامج عد الكلمات الذي أوردناه في الفصل الحادي عشر: البرمجة باستخدام الوحدات، الذي كانت الشيفرة الوهمية فيه كما يلي:

```
def numwords(aString):
    list = split(aString) # قائمة بكل عنصر وكلمة
    return len(list) # تعيد عددًا من العناصر في القائمة

for line in file:
```

```
    total = total + numwords(line) # accumulate totals for each line

print( "File had %d words" % total )
```

لننظر إلى متن دالة `numwords()` بما أننا شرحنا كيفية جلب الأسطر من الملف، حيث نريد أن ننشئ قائمةً من الكلمات في سطر، وذلك باستخدام التابع `''.split()` الافتراضي. وإذا نظرنا في توثيق بايثون فسنجد أن دالة `len()` المضمَّنة تعيد عدد العناصر في قائمة ما، ويجب أن يكون ذلك العدد في حالتنا عدد الكلمات في السلسلة النصية، وهو ما نريده بالضبط، وعلى ذلك تبدو الشيفرة النهائية كما يلي:

```
def numwords(aString):
    lst = aString.split() # split() aString هو تابع كائن السلسلة
    return len(lst)       # أعد عدد العناصر في القائمة

with open("menu.txt","r") as inp:
   total = 0  # initialize to zero; also creates variable
   for line in inp:
      total += numwords(line)  # راكم إجمالي كل سطر

print( "File had %d words" % total )
```

لكن هذه الشيفرة تحسب محارفًا مثل & على أنها كلمات، وهذا ليس صحيحًا، كما أنه لا يمكن استخدامها إلا على ملف واحد فقط هو menu.txt، رغم إمكانية تحويلها لتقرأ اسم الملف من سطر الأوامر `argv[1]`، أو عبر `input()` كما رأينا في الفصل التاسع: قراءة البيانات من المستخدم، وسنترك هذا تدريبًا للقارئ.

13.2 البحث في النصوص

ستكون العملية التالية التي ننظر فيها هي البحث عن سلسلة فرعية داخل سلسلة أكبر منها، وتدعم بايثون هذا من خلال تابع السلسلة `''.find()` الخاص بها، وأبسط استخدامات هذا التابع تزويده بسلسلة للبحث، وتعيد بايثون فهرس أول محرف من السلسلة الفرعية إذا وجدتها داخل السلسلة الرئيسية، أما إذا لم تجدها فستعيد 1-:

```
>>> aString = "here is a long string with a substring inside it"
>>> print( aString.find('long') )
10
>>> print( aString.find('oxen') )
-1
>>> print( aString.find('string') )
```

```
15
```

لقد كان المثالان الأولان واضحين ومباشرين، فالأول يعيد فهرس بداية `'long'`، أما الثاني فيعيد `-1`، وذلك لأن `'oxen'` غير موجودة داخل `aString`؛ بينما المثال الثالث ففيه أمر مثير للاهتمام، إذ لا تحدد `find` إلا الورود الأول لسلسلة البحث فقط، لكن ماذا لو كانت سلسلة البحث مكررةً أكثر من مرة في السلسلة الأصلية؟ من الممكن هنا أن نستخدم فهرس المرة الأولى لورود سلسلة البحث لنقسم السلسلة الأصلية إلى جزأين ونبحث مرةً أخرى، ونكرر ذلك إلى أن نحصل على النتيجة `-1`، كما يلي:

```
aString = "Bow wow says the dog, how many ow's are in this string?"
temp = aString[:] # استخدم التقسيم لصنع نسخة
count = 0
index = temp.find('ow')
while index != -1:
    count += 1
    temp = temp[index + 1:]  # استخدم التقسيم هنا.
    index = temp.find('ow')

print( "We found %d occurrences of 'ow' in %s" % (count, aString) )
```

وهنا نكون قد عددنا مرات الحدوث فقط، لكن كان بإمكاننا جمع نتائج الفهرس في قائمة من أجل المعالجة لاحقًا.

يستطيع التابع `find()` أن يسرّع من هذه العملية قليلًا باستخدام أحد المعامِلات الاختيارية الخاصة به، وهو موضع البداية في السلسلة الأصلية:

```
aString = "Bow wow says the dog, how many ow's are in this string?"
count = 0
index = aString.find('ow')  # استخدم البدء الافتراضي
while index != -1:
    count += 1
    index = aString.find('ow', index+1)  # اضبط بدءًا جديدًا

print( "We found %d occurrences of 'ow' in %s" % (count, aString) )
```

يلغي هذا الحل الحاجة إلى إنشاء سلسلة نصية جديدة في كل مرة، وهو الأمر الذي قد يبطئ العملية إذا كانت السلسلة طويلةً، وإذا علمنا أن السلسلة الفرعية ستكون في بداية السلسلة الأصلية، أو لم نهتم لمرات الحدوث الأخرى، فمن الممكن أن نحدد قيمتي بداية ونهاية البحث، كما يلي:

```
>>>  # قصر البحث على أول 20 محرف
>>> aString = "Bow wow says the dog, how many ow's are in the string?"
>>> print( aString.find('the',0,20) )
```

توفر بايثون عدة توابع أخرى لمواقف البحث الشائعة، مثل `''.startswith()` و`''.endswith()`، ونستطيع أن نخمن وظائف هذه التوابع بمجرد قراءة أسمائها، وهي تعيد إما `True` أو `False` بناءً على بدء السلسلة بسلسلة نصية معطاة أو انتهائها بها، كما يلي:

```
>>> print( "Python rocks!".startswith("Perl") )
False
>>> print( "Python rocks!".startswith('Python') )
True
>>> print( "Python rocks!".endswith('sucks!') )
False
>>> print( "Python rocks!".endswith('cks!') )
True
```

لاحظ أنها تعطينا نتائج بوليانيةً، حيث سنعلم أين نبحث إذا كانت الإجابة `True`. كذلك لاحظ أن سلسلة البحث لا يجب أن تكون كلمةً كاملة، بل تكفي سلسلة نصية فرعية. يمكن تزويد موضعي البدء `start` والانتهاء `stop` داخل السلسلة النصية، تمامًا مثل `''.find()`، لنتحقق من وجود سلسلة نصية فرعية في أي موضع معطىً داخل السلسلة النصية، ولا تستخدَم هذه الخاصية الأخيرة في البرمجة العملية كثيرًا. كما يمكن استخدام عامل `in` الخاص ببايثون لإجراء اختبار بسيط للتحقق من وجود سلسلة نصية فرعية في أي مكان داخل سلسلة أخرى:

```
>>> if 'foo' in 'foobar': print( 'True' )
True
>>> if 'baz' in 'foobar': print( 'True' )
>>> if 'bar' in 'foobar': print( 'True' )
True
```

13.3 استبدال النصوص

بما أننا عثرنا على النص الذي نريده، فلنغيره الآن إلى شيء آخر. توفر توابع السلاسل النصية في بايثون لهذا حلًا متمثلًا في التابع `''.replace()` الذي يأخذ وسيطين، هما سلسلتا البحث والاستبدال، وتكون القيمة المعادة هي السلسلة الجديدة الناتجة عن الاستبدال.

```
>>> aString = "Mary had a little lamb, its fleece was dirty!"
>>> print( aString.replace('dirty','white') )
"Mary had a little lamb, its fleece was white!"
```

إن الفرق الأساسي بين `''.find()` و`''.replace` هو أن الأخير يستبدل جميع مرات الحدوث في سلسلة البحث، وليس المرة الأولى فقط، ويُستخدم الوسيط الاختياري `count` لتقييد عدد مرات الاستبدال، كما يلي:

```
>>> aString = "Bow wow wow said the little dog"
>>> print( aString.replace('ow','ark') )
Bark wark wark said the little dog
>>> print( aString.replace('ow','ark',1) ) # واحد فقط
Bark wow wow said the little dog
```

من الممكن إجراء عمليات بحث واستبدال معقدة باستخدام ما يسمى بالتعبير النمطي regular expression، لكنها معقدة وسنتحدث عنها في فصل كامل لاحقًا.

13.4 تغيير حالة الأحرف

مما يجب مراعاته في النصوص هو كيفية التحويل من الحالة الصغرى للحروف إلى الحالة الكبرى، وإن كان هذا غير شائع، إلا أن بايثون توفر بعض التوابع المساعدة لذلك على أي حال:

```
>>> print( "MIXed Case".lower() )
mixed case
>>> print( "MIXed Case".upper() )
MIXED CASE
>>> print( "MIXed Case".swapcase() )
mixED cASE
>>> print( "MIXed Case".capitalize() )
Mixed case
>>> print( 'MIXed Case'.title() )
Mixed Case
>>> print( "TEST".isupper() )
True
>>> print( "TEST".islower() )
False
```

لاحظ أن التابع `''.capitalize()` يغير حالة الأحرف إلى الحالة الكبرى للسلسلة النصية كلها، وليس لكل كلمة فيها؛ أما تغيير حالة كل كلمة فينفذها التابع `title()`.

انتبه أيضًا إلى سلوك دالتي الاختبار `''.isupper()` و`''.islower()`، إذ توفر بايثون دوالًا توكيديةً مثل هذه لاختبار السلاسل النصية، منها: `''.isdigit()` و`''.isalpha()` و`''.isspace()`، وتتحقق الدالة الأخيرة من جميع أنواع المسافات البيضاء، لا محارف المسافة العادية فقط.

وسنستخدم عدة توابع من هذا النمط، خاصةً في الفصل الثاني والعشرين: دراسة حالة لبرنامج عد الكلمات.

13.5 التعامل مع النصوص في VBScript

تنحدر VBScript من لغة BASIC القديمة، لذا تحتوي على الكثير من دوال معالجة النصوص المدمجة فيها، وقد تجد 20 دالةً أو تابعًا في توثيقها، بالإضافة إلى الدوال الموجودة لمعالجة محارف اليونيكود، مما يعني أننا نستطيع فعل كل ما فعلناه في بايثون باستخدام VBScript، وسنمر عليها سريعًا فيما يلي:

13.5.1 تقسيم النصوص

نبدأ بدالة `split`:

```
<script type="text/vbscript">
Dim s
Dim lst
s = "Here is a string of words"
lst = Split(s) ' returns an array
MsgBox lst(1)
</script>
```

كما في بايثون، نستطيع إضافة قيمة فاصلة إذا أردنا، غير فاصل المسافة البيضاء العادي، كذلك لدينا دالة `Join` لعكس هذه العملية.

13.5.2 البحث عن النصوص واستبدالها

نبحث باستخدام `InStr`، وهو اختصار لـ "In String":

```
<script type="text/vbscript">
Dim s,n
s = "Here is a long string of text"
n = InStr(s, "long")
MsgBox "long is found at position: " & CStr(n)
```

```
</script>
```

القيمة المعادة هنا هي الموضع الذي تبدأ فيه السلسلة الفرعية داخل السلسلة الأصلية، فإذا لم توجد السلسلة الفرعية فستعيد صفرًا، وهذه ليست مشكلةً بما أن VBScript تبدأ فهارسها بواحد وليس صفر، لذا لا يُعَد الصفر هنا فهرسًا صالحًا؛ أما إذا كانت إحدى السلسلتين `Null`، فستعيد القيمة `Null`، مما يجعل عملية اختبار شروط الخطأ أكثر صعوبةً وتحتاج إلى عدة اختبارات معًا.

يمكن تحديد نطاق فرعي من السلسلة الأصلية للبحث فيه باستخدام قيمة بادئة، كما فعلنا في بايثون، كما يلي:

```
<script type="text/vbscript">
Dim s,n
s = "Here is a long string of text"
n = InStr(6, s, "long") ' start at position 6
If n = 0 or n = Null Then ' check for errors
   MsgBox "long was not found"
Else
   MsgBox "long is found at position: " & CStr(n)
End If
</script>
```

ونستطيع تحديد كون البحث حساسًا لحالة الأحرف أم لا، وهذا غير موجود في بايثون، والوضع الافتراضي للبحث هو أن يكون حساسًا لحالة الأحرف.

تستبدَل النصوص باستخدام الدالة `Replace` كما يلي:

```
<script type="text/vbscript">
Dim s
s = "The quick yellow fox jumped over the log"
MsgBox Replace(s, "yellow", "brown")
</script>
```

ونستطيع توفير وسيط اختياري ليحدد عدد مرات الحدوث في سلسلة البحث التي يجب استبدالها، والوضع الافتراضي هو استبدالها جميعًا، لكن يمكن تحديد موضع بدء كما في `InStr` أعلاه.

13.5.3 تغيير الحالة

تغيّر الحالة في VBScript بواسطة `UCase` و`LCase`، لكن لا يوجد فيها ما يكافئ التابعين `capitalize` و`title` الموجودين في بايثون:

```
<script type="text/vbscript">
Dim s
s = "MIXed Case"
MsgBox LCase(s)
MsgBox UCase(s)
</script>
```

وهـذا مـا سـنغطيه من معالجـة النصـوص في VBScript، فـإذا أردت المزيـد فـارجع إلى ملـف مسـاعدة VBScript لترى القائمة الكاملة للدوال.

13.6 التعامل مع النصوص في جافاسكربت

تُعَد جافاسكربت أقل اللغات الثلاث تجهيزًا للتعامل مع النصوص، لكن العمليات الأساسية موجودة إلى حد ما، رغم افتقارها إلى العدد الكبير من التوابع والدوال الموجودة في بـايثون وVBScript للتعامـل مـع النصـوص، وهي تعوض ذلك بدعم قوي للتعابير النمطيـة، حيث تعـوض بدائيـة الـدوال الموجـودة كثـيرًا، لكن على حسـاب تعقيد التعابير النمطية، وهي تأخذ المنظور كائني التوجه، مثل بايثون في التعامل مـع السلاسـل النصـية، حيث تُنفَّذ المهام باستخدام توابع الصنف `String`.

13.6.1 تقسيم النصوص

تقسَّم النصوص في جافاسكربت باستخدام التابع `split`:

```
<script type="text/javascript">
var aList, aString = "Here is a short string";
aList = aString.split(" ");
document.write(aList[1]);
</script>
```

لاحظ أن جافاسكربت تتطلب تزويدها بمحرف الفصل، فليس لديها قيمة افتراضـية لـه، والفاصـل هنـا هـو تعبير نمطي، مما يجعل عمليات الفصل المعقدة ممكنةً. وقد ذكرنـا سـابقًا أن النصـوص تُـدمج باسـتخدام تـابع المصفوفة `Join`، لذا فإن عكس عملية التقسيم أعلاه سيكون كما يلي:

```
aList.join(" ")  // ادمج عناصر مفصولة بمسافات
```

13.6.2 البحث في النصوص

نبحث في النصوص في جافاسكربت باستخدام التابع search():

```
<script type="text/javascript">
var aString = "Round and Round the ragged rock ran a rascal";
document.write( "ragged is at position: " + aString.search("ragged"));
</script>
```

وهنا يكون وسيط سلسلة البحث تعبيرًا نمطيًا أيضًا كي نتمكن من إجراء عمليات بحث معقدة، لكن انتبه إلى عدم وجود طريقة لتقييد نطاق السلسلة الأصلية الذي يُبحث فيه، وذلك من خلال تمرير موضع بدء، رغم إمكانية محاكاة ذلك باستخدام التعابير النمطية على أي حال.

> توفر جافاسكربت عملية بحث أخرى لها سلوك مختلف تسمى match()، لكننا لن نشرح استخدامها هنا.

13.6.3 استبدال النصوص

يُستخدم التابع replace() لاستبدال النصوص كما يلي:

```
<script type="text/javascript">
var aString = "Humpty Dumpty sat on a cat";
document.write(aString.replace("cat","wall"));
</script>
```

يمكن أن تكون سلسلة البحث تعبيرًا نمطيًا بحيث نرى هنا السلوك المتوقع، إذ تستبدل عملية الاستبدال جميع نسخ سلسلة البحث، ونستطيع القول أنه لا توجد طريقة تقيد عملية الاستبدال في جزء من السلسلة أو مرة حدوث واحدة دون تقسيم السلسلة أولًا ثم إعادة دمجها مرةً أخرى.

13.6.4 تغيير الحالة

تغيِّر حالة الأحرف في جافاسكربت باستخدام دالتين هما toLowerCase() وtoUpperCase():

```
<script type="text/javascript">
var aString = "This string has Mixed Case";
document.write(aString.toLowerCase()+ "<BR>");
document.write(aString.toUpperCase()+ "<BR>");
</script>
```

تغني بساطة هذين التـابعين عن شـرحهما، والجـدير فقـط بالـذكر هنـا أن جافاسـكربت على عكس اللغـات الأخرى، توفر دوالًا كثيرةً لمعالجة HTML، لأنها لغة برمجـة ويب بالأسـاس، ويمكن الرجـوع إلى تلـك الـدوال في توثيق اللغة، إذ هي خارج مجال حديثنا.

13.7 خاتمة

مررنا على الدوال والتوابع المستخدمة للتعامل مع النصوص التي قد تراها في مشاريعك، ونود الإشـارة هنـا إلى وجوب النظر في التوثيق الرسمي للغة التي تعمل بها عند معالجة النصوص، إذ توجد أدوات قوية لمثل هـذه العمليات والمهام الضرورية في البرمجة، وننصـح بـالبحث أولًا في التوثيقـات المتـوفرة في موسـوعة حسـوب، وهي توثيقات عربية مترجمة من التوثيقات الرسمية للغات أو من أمهات الكتب فيهـا، ونـود أن تخـرج من هـذا الفصل بما يلي:

- معالجة النصوص عملية شائعة لها دعم قوي مضمَّن في أغلب اللغات.
- أكثر المهام شيوعًا هي تقسيم النصوص والبحث فيها، واستبدالها، وتغيير حالة الأحرف فيها.
- توفر كل لغة مستويات مختلفة من الدعم، لكن العمليات الثلاث الأساسية متاحة دومًا.

14. التعامل مع الأخطاء البرمجية

نقصد بمعالجة الأخطاء عملية التقاط الأخطاء التي تولدها برامجنا وإخفائها عن المستخدم، فرسائل الأخطـاء لا تخيف المبرمجين بل يمكن توقعها أحيانًا، إلا أن المستخدم لا يتوقع رؤيتها، وهي تربكه وتسبب له حيرةً، فـإن كان سيرى رسالة خطأ للضرورة، فلتكن رسالةً سهلة الفهم، وحتى في هذه الحالة سيرغب المستخدم في أن يحل المبرمج المشكلة، وهنا يأتي دور التعامل مع الأخطاء ومعالجتها، حيث توفر كل لغة تقريبًا آليةً لالتقـاط الأخطـاء عند حدوثها لمعرفة الأجزاء التي تعطلت، واتخاذ الإجـراء المناسـب لإصـلاح المشـكلة، وقـد تطـورت عـدة طـرق لمعالجة هذه الأخطاء، والتي سننظر فيها هنا متبعين تطورها التاريخي مع تطـور التقنيـة لنعـرف الأسـباب الـتي أدت إلى ظهور منظور جديد رغم وجودها سابقًا، ونرجو أن تكون قادرًا في نهاية الفصل على كتابة برامج لا تسـمح بظهور رسائل خطأ للمستخدم.

تجدر الإشارة إلى أن لغة VBScript هي أكثر لغة غريبة من اللغات الثلاثة التي ندرسها في معالجـة الأخطـاء، لأنها بُنيت على لغة BASIC، التي إحدى أوائل لغات البرمجـة الـتي خـرجت في عـام 1963، وسـنرى كيـف ألقى تراثها بظلاله على VBScript فيما يتعلق بمعالجـة الأخطـاء، لكن هـذا لا يـؤثر على سـياق شـرحنا، بـل سـيعطينا الفرصة لشرح سبب سلوك VBScript بتعقب تاريخ معالجة الأخطاء، بدءًا من لغـة BASIC، مـرورًا بلغـة Visual Basic حتى VBSCript، ثم ننظر بعدها إلى منظور أحدث، ونرى أمثلةً له في بايثون وجافاسكربت.

لقد كُتبَت البرامج في لغة BASIC مع أرقام للأسـطر لتمييزهـا، حيث يُنقـل التحكم بـالقفز إلى سـطر بعينـه باستخدام تعليمة GOTO التي رأينا مثالًا لها في الفصل العاشر: مقدمة في البرمجة الشـرطية، وقـد كـانت هـذه صورة التحكم الوحيدة المتاحة وقتها، حيث كان الأسلوب الشائع لمعالجة الأخطاء حينئذ هو التصـريح عن متغـير `errorcode` الذي يخزن قيمةً عدديةً، وكلما حدث خطأ في البرنـامج، سيُضـبط المتغـير `errorcode` ليعكس المشكلة. فإما أن يخبرنا أنه لم يستطع فتح الملف، أو أن النوع غير متطابق، أو حـدث طفح للعوامـل operator overflow، أو غير ذلك، وقد أدى هذا إلى شيفرة تشبه المثال التالي من برنامج وهمي:

```
LET DATA = INPUT FILE
CALL DATA_PROCESSING_FUNCTION
IF NOT ERRORCODE = 0 GOTO 5000
CALL ANOTHER_FUNCTION
IF NOT ERRORCODE = 0 GOTO 5000
REM CONTINUE PROCESSING LIKE THIS
...
IF ERRORCODE = 1 GOTO 5100
IF ERRORCODE = 2 GOTO 5200
REM MORE IF STATEMENTS
...
REM HANDLE ERROR CODE 1 HERE
...
REM HANDLE ERROR CODE 2 HERE
```

يفحص نصف البرنامج الرئيسي وجود خطأ، لكن ظهرت مع الـوقت آليـة أفضـل، حيث تـولى مفسـر اللغـة عملية التقاط الأخطاء ومعالجتها؛ ولو جزئيًا، كما يلي:

```
LET DATA = INPUTFILE
ON ERROR GOTO 5000
CALL DATA_PROCESSING_FUNCTION
CALL ANOTHER_FUNCTION
...
IF ERRORCODE = 1 GOTO 5100
IF ERRORCODE = 2 GOTO 5200
```

سمح هذا بالإشارة إلى المكان الذي توجد فيه شيفرة الخطأ بواسطة سطر واحد، ورغم أننـا لا زلنـا بحاجـة إلى الدوال التي اكتشفت الخطأ لضبط قيمة ERRORCODE، إلا أنها جعلت كتابة الشيفرة وقراءتها أسهل بكثـير، لكن كيف يتأثر المبرمجون بهذا الأمر؟ توفر Viusal Basic إلى الآن هـذا النـوع من معالجـة الأخطـاء -على الـرغم من استخدامنا حاليًا طريقةً أفضل من أرقام الأسطر-، وبما أن VBScript تنحدر من Visual Basic، فإنها توفر نسخةً مختصرةً للغاية من هذه الطريقة، وهي تخيّرنا بين معالجة الأخطاء محليًـا أو تجاهلهـا تمامًـا، ونسـتخدم الشـيفرة التالية لتجاهل الأخطاء:

```
On Error Goto 0   ' 0 implies go nowhere
SomeFunction()
SomeOtherFunction()
```

```
....
```

أما لمعالجتها محليًا فنستخدم ما يلي:

```
On Error Resume Next
SomeFunction()
If Err.Number = 42 Then
   ' handle the error here
SomeOtherFunction()
...
```

يبدو هذا المنطق معكوسًا، لكنه يوضح العملية تاريخيًا كما أوضحنا أعلاه، فالسلوك الافتراضي للمفسر هو توليد رسالة إلى المستخدم، وإيقاف تنفيذ البرنامج إذا اكتشف خطأً ما، وهذا ما يحدث مع معالجة خطأ `GoTo 0`، فما هي إلا طريقة لإيقاف التحكم المحلي والسماح للمفسر بالعمل كالمعتاد.

تسمح لنا تعليمة `Resume Next` بالتظاهر وكأن الخطأ لم يحدث، أو أن التحقق من كائن الخطأ -الذي يسمى `Err`-، وسمة العدد -مثل تقنية errorcode الأولى-، كما أن للكائن `Err` أجزاء معلومات أخرى قد تفيدنا في التعامل مع الموقف بطريقة أفضل من مجرد إيقاف البرنامج، بحيث نستطيع معرفة مصدر الخطأ مثلًا، سواءً كان كائنًا أم دالةً أم غير ذلك، كما نستطيع الحصول على وصف نصي نستخدمه في تعبئة رسالة تخبر المستخدم بما يحدث، أو كتابة ملاحظة في ملف السجل.

كما يمكن تغيير نوع الخطأ باستخدام التابع `Raise` للكائن `Err`، إلى جانب استخدامنا له لتوليد أخطائنا من داخل دوالنا، لننظر في مثال حالة القسمة على الصفر؛ وهي حالة شائعة، لنرى معالجة الأخطاء في VBScript:

```
<script type="text/vbscript">
Dim x,y,Result
x = Cint(InputBox("Enter the number to be divided"))
y = CINt(InputBox("Enter the number to divide by"))
On Error Resume Next
Result = x/y
If Err.Number = 11 Then ' Divide by zero
   Result = Null
End If
On Error GoTo 0 ' turn error handling off again
If VarType(Result) = vbNull Then
   MsgBox "ERROR: Could not perform operation"
Else
```

```
    MsgBox CStr(x) & " divided by " & CStr(y) & " is " & CStr(Result)
End If
</script>
```

هذا الأسلوب غير مثالي، ورغم أن تقدير التراث البرمجي هنا جميل ولطيف، إلا أن لغات البرمجة الحديثة بما فيها بايثون وجافاسكربت- لديها طرق أفضل لمعالجة الأخطاء، كما سنشرح في فقرتنا التالية.

14.1 معالجة الأخطاء في بايثون

سنعرض فيما يلي آليات التعامل مع الأخطاء والاستثناءات التي تحصل أثناء تنفيذ شيفرة البرنامج وكيفية معالجتها في بايثون.

14.1.1 التعامل مع الاستثناءات

تتعامل لغات البرمجة الحديثة مع الاستثناءات exceptions وتعالجها بجعل الدوال ترفع الاستثناء raise أو تلقيه throw، ثم يفرض النظام قفزةً إلى خارج كتلة التعليمات البرمجية الحالية إلى أقرب كتلة معالجة استثناءات، ويوفر النظام معالجًا افتراضيًا يلتقط جميع الاستثناءات التي لم تعالَج في مكان آخر، كما يطبع رسالة خطأ ثم يخرج. انظر الفصل الثالث: بداية رحلة تعلم البرمجة لمراجعة كيفية قراءة رسائل الخطأ في بايثون وتفسيرها، حيث تتمثل إحدى مزايا هذا النمط من معالجة الأخطاء في سهولة رؤية الوظيفة الأساسية للبرنامج، لأنها غير مختلطة بشيفرة معالجة الأخطاء، إذ نستطيع قراءة الكتلة الرئيسية دون الحاجة إلى النظر إلى شيفرة الخطأ مطلقًا. لننظر في كيفية عمل هذا النمط عمليًا:

14.1.2 استثناءات Try/Except

تُكتب كتلة معالجة الاستثناءات على شكل كتلة `if ...then...else`:

```
try:
    # منطق البرنامج هنا
except ExceptionType:
    # معالجة الاستثناءات للاستثناء المسمى هنا
except AnotherType:
    # معالجة الاستثناءات لاستثناءات أخرى هنا
else:
    # هنا نقوم بالترتيب إذا لم تُرفع استثناءات
```

تحاول بايثون أن تنفذ التعليمات بين `try` وأول تعليمة `except`، فإذا واجهت خطأً ما، فستوقف تنفيذ شيفرة `block` وتقفز إلى تعليمات `except` حتى تجد واحدةً تطابق نوع الخطأ أو الاستثناء، فإذا وجدت مطابقةً، فستنفذ الشيفرة التي في الكتلة التي بعد هذا الاستثناء مباشرةً؛ أما إذا لم توجد تعليمة `except` مطابِقة،

فسيُنشر الخطأ إلى المستوى التالي للبرنامج، إلى أن توجد مطابقة، أو أن يكتشف مفسر المستوى الأعلى في بايثون هذا الخطأ ويعرض رسالة خطأ ويوقف تنفيذ البرنامج، وهو ما رأيناه في برامجنا إلى الآن.

أما إذا لم يوجد خطأ في كتلة `try`، فستنفَّذ كتلة `else` الأخيرة، رغم أن هذه الخاصية لا تُستخدم إلا نادرًا. لاحظ أن تعليمة `except` التي ليس فيها نوع خطأ محدد، إذ ستلتقط كل أنواع الأخطاء التي لم تعالَج بعد، وهذا سيئ إلا في حالة المستوى الأعلى في برنامجك حين تريد تجنب عرض رسائل بايثون التقنية إلى المستخدمين، حيث يمكن استخدام تعليمة استثناء عامة لالتقاط أي أخطاء غير ملتقطة، وعرض رسالة "إغلاق" مناسبة للمستخدم، ويجب الانتباه إلى تسجيل بيانات الخطأ في ملف السجل للتحليل في المستقبل.

توفر لغة بايثون وحدة `traceback` التي تمكنك من استخراج أجزاء من المعلومات من مصدر الخطأ، وقد يكون هذا مفيدًا في إنشاء ملفات السجلات وما شابهها، لكننا لن نشرح هذه الوحدة، فإذا احتجت إليها فسيوفر توثيق الوحدات القياسي قائمةً كاملةً من المزايا والخصائص المتوفرة لها.

لننظر الآن في مثال حقيقي لتوضيح الشرح:

```
value = input("Type a divisor: ")
try:
   value = int(value)
   print( "42 / %d = %d" % (value, 42/value) )
except ValueError:
   print( "I can't convert the value to an integer" )
except ZeroDivisionError:
   print( "Your value should not be zero" )
except:
   print( "Something unexpected happened" )
else: print( "Program completed successfully" )
```

إذا شغلنا هذا البرنامج وأدخلنا قيمةً ليست برقم مثل إدخال سلسلة نصية في المحث، فسنحصل على رسالة `ValueError`؛ أما إذا أدخلنا `0` فنحصل على رسالة `ZeroDivisionError`، وإذا ضغطنا Ctrl+C فسنرفع استثناء `KeyboardInterrupt` ونرى رسالةً تقول "Something unexpected happened"؛ أما إذا كتبنا عددًا صالحًا فسنحصل على النتيجة مع رسالة "Program Completed successfully".

14.1.3 استثناءات Try/Finally

ثمة نوع آخر من كتل الاستثناءات التي تسمح لنا بالترتيب بعد حدوث خطأ ما، وتسمى `try...finally`، كما تُستخدم لإغلاق الملفات واتصالات الشبكات أو قواعد البيانات، وتنفَّذ كتلة `finally` في النهاية بغض النظر عما يحدث في قسم `try`:

```
try:
    # المنطق المعتاد للبرنامج
finally:
    # try هنا نرتب بغض النظر عن نجاح كتلة
    # أو فشلها
```

تصبح الكتلة قويةً للغاية إذا جمعناها مع `try/except`:

```
print( "Program starting" )
try:
    data = open("data.dat")
    print( "data file opened" )
    value = int(data.readline().split()[2])
    print( "The calculated value is %s" % (value/(42-value)) )
except ZeroDivisionError:
    print( "Value read was 42" )
finally:
    data.close()
    print( "data file closed" )

print( "Program completed" )
```

لاحظ أن ملف البيانات يجب أن يحتوي على سطر مع رقم في الحقل الثالث، كما يلي:

```
Foo bar 42
```

هنا يُغلق ملف البيانات دومًا بغض النظر عن رفع الاستثناء في كتلة `try/except` أو لا. لاحظ أن هذا السلوك مختلف عن شرط `else` لكتلة `try/except`، لأنه يُستدعى فقط عند عدم رفع استثناءات، كما يعني وضع الشيفرة خارج كتلة `try/except` أن الملف لم يغلَق إذا كان الاستثناء شيئًا غير `ZeroDivisionError`، ولا نضمن أن الملف مغلق إلا بإضافة كتلة `finally`. كذلك وضعنا تعليمة `open()` داخل كتلة `try/except`، فإذا أردنا التقاط خطأ فتح ملف، فسنحتاج إلى إضافة كتلة `except` أخرى لـ `IOError`. جرب هذا بنفسك ثم افتح ملفًا غير موجود لترى ذلك عمليًا.

14.1.4 توليد الأخطاء

إذا أردنا توليد استثناءات ليلتقطها غيرنا في وحدة ما، فنستخدم الكلمة المفتاحية `raise` في بايثون:

```
numerator = 42
denominator = int( input("What value will I divide 42 by?") )
if denominator == 0:
   raise ZeroDivisionError
```

يرفع هذا استثناء `ZeroDivisionError` الذي يمكن التقاطه بواسطة كتلة `try/except`، أما بالنسبة لبقية البرنامج فسيبدو كما لو أن بايثون ولّدت هذا الخطأ داخليًا.

يمكن استخدام كلمة `raise` في توليد خطأ لمستوى أعلى في البرنامج من داخل كتلة الاستثناء، فقد نرغب في أخذ إجراء محلي، مثل تسجيل خطأ في ملف، ثم نسمح للمستوى الأعلى من البرنامج أن يقرر الإجراء النهائي، كما يلي:

```
def div127by(datum):
    try:
      return 127/(42-datum)
    except ZeroDivisionError:
      logfile = open("errorlog.txt","a")
      logfile.write("datum was 42\n")
      logfile.close()
      raise

try:
   div127by(42)
except ZeroDivisionError:
   print( "You can't divide by zero, try another value" )
```

لاحظ كيف تلتقط الدالة `div127by()` الخطأ، وتسجل رسالةً في ملف الخطأ، ثم تمرر الاستثناء مرةً أخرى إلى كتلة `try/except` الخارجية لتتعامل معه باستدعاء `raise` دون كائن خطأ محدد. لنجمع هذين الجزأين معًا في برنامج واحد يوضح معالجة الأخطاء عمليًا:

```
def div127by(datum):
    try:
      return 127/(42-datum)
    except ZeroDivisionError:
```

```
        logfile = open("errorlog.txt","a")
        logfile.write("datum was 42\n")
        logfile.close()
        raise

try:
   divisor = int( input("What value will I divide by?") )
   if divisor == 0:
      raise ZeroDivisionError
   print( "The result is: ", div127by(divisor) )
except ZeroDivisionError:
   print( "You can't divide by zero, try another value" )
```

فإذا أدخل المستخدم `42` أو `0` فسينتِج `ZeroDivisionError`، مع أن 0 قيمة آمنة في هـذه الحالـة؛ أمـا غير ذلك فنطبع نتيجة القسمة ونسجل قيمة الدخل في الملف `errorlog.txt`.

14.1.5 الاستثناءات المعرفة من قبل المستخدم

توفر بايثون نطاقًا واسعًا من أنواع الأخطاء القياسية، ويجب أن نعيد استخدام هذه الأخطاء ما أمكن، لكن قد لا نجد خطأً يناسب احتياجنا، عندئذ نستطيع تعريف أنواع الاسـتثناءات الخاصـة بنـا للتحكم في برامجنـا تحكمًـا دقيقًا، وقد مررنا على تعريف الأصناف في الفصل الخامس: البيانات وأنواعها مرورًا سـريعًا، وسـنعود إليهـا مـرةً أخرى في الفصل السابع عشر: البرمجة كائنية التوجه.

لا يحتـوي صـنف الاسـتثناء عـادةً على محتـوىً خـاص بـه، وإنمـا نعـرف صـنفًا فرعيًـا من `Exception`، ونستخدمه مثل نوع من الملصقات الذكيـة الـتي يمكن التقاطهـا بواسـطة تعليمـات `except`. لننظـر في هـذا المثال القصير:

```
>>> class BrokenError(Exception): pass
...
>>> try:
...   raise BrokenError
... except BrokenError:
...   print( "We found a Broken Error" )
...
```

لاحظ أننا نستخدم اصطلاح تسمية نضيف فيه `Error` إلى نهاية اسم الصنف، وأننا نكتسب سلوك صنف `Exception` العام بإدراجه في أقواس بعد الاسم، كما سنتعلم الاكتساب أو الوراثة inheritance في فصل البرمجة كائنية التوجه.

يجب ملاحظة نقطة أخيرة في رفع الأخطاء، وهي أننا كنا ننهي برامجنا باستيراد `sys` واستدعاء الدالة `exit()`، لكن يمكن استخدام أسلوب آخر يحقق نفس النتيجة، عن طريق رفع خطأ `SystemExit()` كما يلي:

```
>>> raise SystemExit
```

وميزة هذا الأسلوب أننا لا نحتاج إلى `import sys` في البداية.

14.2 جافاسكربت

تعالج جافاسكربت الأخطاء بطريقة تشبه طريقة بايثون، باستخدام الكلمات المفتاحية `try` و`catch` و`throw` مقابل كلمات بايثون `try` و`except` و`raise`، وسننظر الآن في بعض الأمثلة، كما سنرى استخدام المبادئ نفسها التي كانت في بايثون، وقد أدخلت الإصدارات الأخيرة من جافاسكربت بنية `finally`، كما يمكن استخدام شرط `finally` في جافاسكربت مع كتلة `try/catch` في بنية واحدة. انظر توثيق جافاسكربت لمزيد من التفاصيل.

14.2.1 التقاط الأخطاء

تُلتقَط الأخطاء باستخدام كتلة `try` مع مجموعة من تعليمات `catch` تكاد تكون مطابقةً لما رأيناه في بايثون:

```
<script type="text/javascript">
try{
   var x = NonExistentFunction();
   document.write(x);
}
catch(err){
   document.write("We got an error in the code");
}
</script>
```

يكمن الاختلاف الأساسي في أننا نستخدم تعليمة `catch` واحدةً فقط لكل بنية `try`، وعلينا أن نفحص الخطأ الممرَّر داخل كتلة `catch` لنرى نوعه، وهذا فوضوي موازنةً بأسلوب `except` المتعدد في بايثون والمبني على نوع الاستثناء، وسنرى مثالًا بسيطًا لاختبار قيم الأخطاء في الشيفرة التالية.

14.2.2 رفع الأخطاء

يمكن رفع الأخطاء باستخدام الكلمة `throw` كما استخدمنا كلمة `raise` في بايثون، كما نستطيع إنشاء أنواع الخطأ الخاصة بنا في جافاسكربت كما فعلنا في بايثون، لكن الأسهل هو استخدام سلسلة نصية:

```
<script type="text/javascript">
try{
   throw("New Error");
}
catch(e){
   if (e == "New Error")
      document.write("We caught a new error");
   else
      document.write("An unexpected error found");
}
</script>
```

هذا كل ما سنشرحه حول معالجة الأخطاء، وسنرى أمثلةً عمليةً لها في الفصول التالية، كما سنرى بعض المفاهيم الأساسية التي تحدثنا عنها من قبل، مثل التسلسلات والحلقات التكرارية والفروع، مما يعني أن لديك جميع الأدوات اللازمة لإنشاء برامج قوية. يفضل الآن أن تأخذ وقتًا تحاول فيه إنشاء بعض البرامج بنفسك -ربما بضعة برامج فقط-، لتثبت تلك المفاهيم في رأسك قبل الانتقال إلى مجموعة الفصول التالية، وتستطيع البدء بالبرامج التالية:

- لعبة بسيطة مثل OXO.
- قاعدة بيانات بسيطة، ربما مبنية على دليل جهات الاتصال الخاص بنا، ولكن لتخزين مجموعة أقراصك أو مقاطع الفيديو.
- أداة تسجيل يوميات تتيح لك إمكانية تخزين الأحداث أو التواريخ المهمة، وربما تخرج لك إشعارًا تذكيريًا.

ستحتاج إلى استخدام جميع الخصائص التي شرحناها من قبل، وربما بعض وحدات اللغات كذلك، وهنا تذكر أن تعود إلى التوثيق كل فترة، إذ ستجد أدوات أخرى تعينك على إنشاء مثل هذه البرامج، ولا تنسى أيضًا قوة محث بايثون. جرب برامجك هناك إلى أن تفهم كيفية عملها، ثم انقل ذلك إلى البرنامج الخاص بك.

14.3 خاتمة

نرجو في نهاية الفصل أن تكون تعلمت ما يلي:

- التحقق من شيفرات أخطاء VBScript باستخدام تعليمة `if`.
- التقاط الاستثناءات بشرط `except` في بايثون أو `catch` في جافاسكربت.
- توليد الاستثناءات باستخدام كلمة `raise` المفتاحية في بايثون أو `throw` في جافاسكربت.
- أنه يمكن أن تكون أنواع الأخطاء أصنافًا في بايثون أو سلسلةً بسيطةً في جافاسكربت.

دورة تطوير التطبيقات
باستخدام لغة بايثون

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

الباب الثالث: مواضيع متقدمة

15. فضاءات الأسماء

فضاء الاسم Namespace هو المساحة أو المنطقة داخل البرنامج التي يكون الاسم صالحًا فيها، سواءً كان ذلك الاسم متغيرًا، أو دالةً، أو صنفًا، أو غير ذلك. يُستخدم هذا المبدأ في الحياة العملية كل يوم، فلو كنت تعمل في شركة كبيرة مثلًا ولك زميل اسمه عماد، ويوجد عماد آخر في قسم المحاسبة، فإنك تشير إلى عماد زميلك في القسم باسمه المجرد "عماد"؛ أما عماد الآخر فستقول "عماد الذي في قسم المحاسبة"، أي أنك تستخدم أسماء أقسام الشركة معرِّفات للعاملين فيها، وهذا هو مبدأ فضاء الاسم في البرمجة، فهو يخبر المبرمج والمترجم بالاسم المقصود فيما لو وجد أكثر من اسم.

لقد ظهرت فضاءات الأسماء لأن لغات البرمجة القديمة -مثل BASIC- لم تكن فيها إلا متغيرات عامة Global Variables، أي متغيرات يمكن رؤيتها في البرنامج كله وداخل الدوال أيضًا، لكن هذا جعل متابعة أداء البرامج الكبيرة وصيانتها صعبًا، لأن التعديل على متغير في جزء ما من البرنامج سيتسبب في تغير وظيفة جزء آخر دون أن يدرك المبرمج ذلك، وهو ما يسمى بالآثار الجانبية، وقد قدمت اللغات التالية -بما فيها النسخ الأحدث من BASIC- مبدأ فضاء الأسماء إلى البرمجة، بل إن لغةً مثل C++ تسمح للمبرمج بإنشاء فضاء اسم خاص به في أي مكان داخل البرنامج، وهذا مفيد لمنشئي المكتبات الذين يرغبون في الاحتفاظ بأسماء دوالهم فريدةً عند اختلاطها مع مكتبات أخرى.

يشار أحيانًا إلى فضاء الاسم بالنطاق Scope، ونطاق الاسم هو المجال الذي يمكن استخدام الاسم فيه، كأن يكون داخل دالة أو وحدة module، وفضاء الاسم والنطاق هما وجهان لعملة واحدة باستثناء فروق طفيفة بين المصطلحين، ولن يناقش فيها إلا عالم حاسوب يحب الجدل؛ أما بالنسبة لمستوى الشرح الذي نريده فهما متطابقان.

سيشرح هذا الفصل:

- مفهوم فضاء الاسم أو النطاق وأهميته.

- كيفية عمل فضاء الاسم في بايثون.
- مفهوم فضاء الاسم في لغتي جافاسكربت وVBScript.

15.1 فضاء الاسم في بايثون

تنشئ كل وحدة في بايثون فضاء الاسم الخاص بها، وللوصول إلى تلك الأسماء لا بد أن نسبقها باسم الوحدة، أو أن نستورد الأسماء التي نريد استخدامها إلى فضاء الاسم الخاص بالوحدة، وقد كنا نفعل هذا مع وحدتي `sys` و`time` في الفصول السابقة، كما أن تعريف الصنف Class ينشئ فضاء اسم خاص به. وبناءً عليه، فإذا أردنا الوصول إلى تابع أو خاصية في صنف ما، فسنحتاج إلى استخدام اسم متغير النسخة أو اسم الصنف أولًا، وسنتحدث عن هذا بالتفصيل في الفصل السابع عشر: البرمجة كائنية التوجه.

تتيح بايثون خمسة فضاءات أسماء -أو نطاقات- هي:

1. النطاق المضمَّن: الأسماء المعرفة داخل بايثون نفسها، وهي متاحة دائمًا من أي مكان في البرنامج.
2. نطاق الوحدة: وهي أسماء معرَّفة ومرئية داخل ملف أو وحدة، لكن هذا النطاق يشار إليه في بايثون باسم النطاق العام global scope، في حين أن المعنى الذي يتبادر للذهن عند سماع الاسم هو أن النطاق العام يمكن رؤيته في أي جزء من البرنامج.
3. النطاق المحلي: وهي الأسماء المعرَّفة داخل دالة أو تابع صنف، بما في ذلك المعامِلات.
4. نطاق الصنف: الأسماء المعرفة داخل الأصناف، وسننظر فيها في فصل البرمجة كائنية التوجه.
5. النطاق المتشعب nested Scope: هذا موضوع معقد قليلًا تستطيع تجاهله حاليًا.

لننظر الآن في الشيفرة التالية التي تحتوي على أمثلة لأول ثلاثة نطاقات:

```
def square(x):
   return x*x

data = int(input('Type a number to be squared: '))
print( data, 'squared is: ', square(data) )
```

يسرد الجدول التالي كلًا من الاسم والنطاق الذي ينتمي إليه:

الاسم	فضاء الاسم
square	الوحدة-عام
x	محلي للنطاق square
data	الوحدة-عام

الاسم	فضاء الاسم
int	مضمَّن
input	مضمَّن
print	مضمَّن

لاحظ أننا لا نَعُد `def` و`return` من الأسماء، وذلك لأنهما كلمتان مفتاحيتان أو جزء من تعريف اللغة نفسها، وسنحصل على خطأ إذا استخدمنا كلمةً مفتاحيةً اسمًا لمتغير.

لنطرح سؤالًا جديدًا الآن، ماذا يحدث عندما يكون للمتغيرات التي في فضاءات أسماء مختلفة نفس الاسم؟ وماذا يحدث عند الإشارة إلى اسم ليس موجودًا في فضاء الاسم الحالي؟

15.1.1 الوصول إلى الأسماء التي خارج النطاق الحالي

تحدد بايثون الأسماء حتى لو لم تكن في فضاء الاسم الحالي بالنظر إلى:

1. فضاء الاسم المحلي -الدالة الحالية- أو الدالة المغلِّفة أو الصنف المغلِّف إذا كانت دالةً متشعبةً أو تابعًا متشعبًا.
2. نطاق الوحدة، أي الملف الحالي.
3. النطاق المضمَّن.

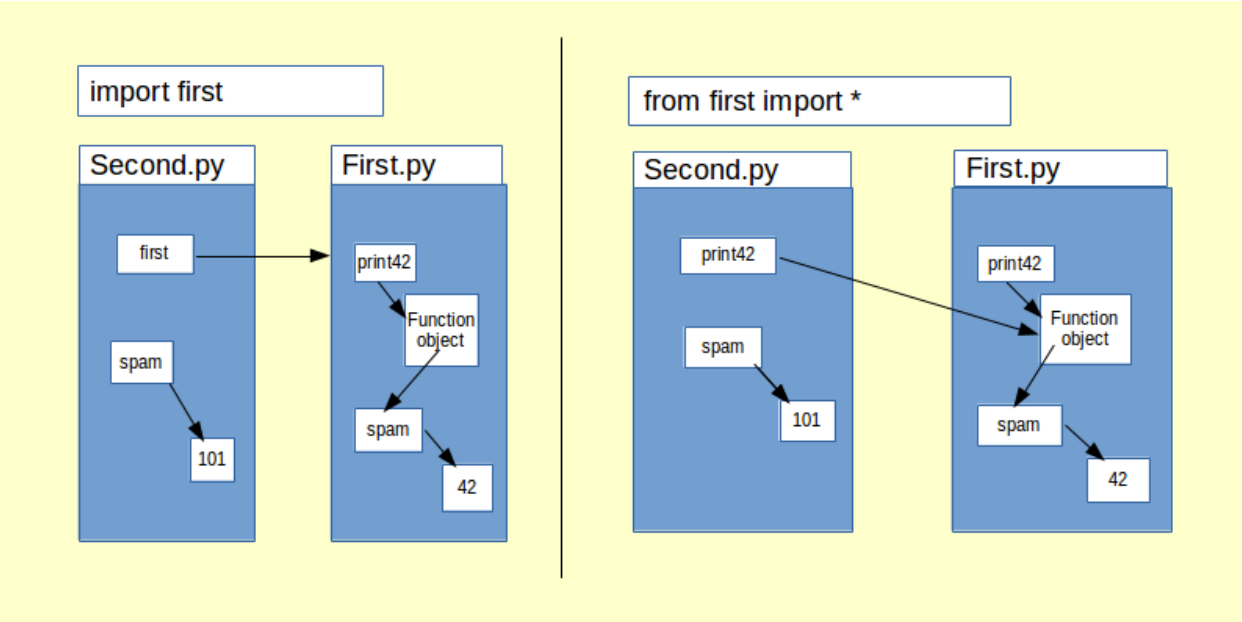
فإذا كان الاسم في وحدة أخرى، فسنستورد الوحدة باستخدام `import` كما رأينا في الفصول السابقة، وعند استيرادها سيكون اسم الوحدة مرئيًا في نطاق تلك الوحدة، وسنستطيع حينئذ أن نستخدم اسم الوحدة للوصول إلى أسماء المتغيرات فيها باستخدام نمط `module.name` المعتاد، ويتبين من هذا أن استيراد جميع الأسماء من وحدة إلى الملف الحالي ليس صحيحًا، إذ قد يتطابق اسم وحدة ما مع اسم أحد متغيراتنا، مما سيتسبب في سلوك غريب للبرنامج لأن أحد الاسمين سيغطي على الآخر.

لنعرِّف وحدتين تستورد الثانية فيهما الأولى على سبيل المثال:

```
##### module first.py #########
spam = 42
def print42(): print( spam )
###############################
##### module second.py ########
from first import *  # استورد جميع الأسماء من الأولى

spam = 101    # أنشئ متغير spam لتخفي النسخة الأولى
print42()     # ما الذي سيُطبع، هل 42 أم 101
```

```
################################
```

إذا ظننت أن هذا المثال سيطبع 101 فستكون مخطئًا، لأنه سيطبع 42 بسبب تعريف المتغير في بـايثون، فكما شرحنا -في الفصل الخامس: البيانات وأنواعها-؛ الاسم هو عنوان يُستخدم للإشارة إلى كائن، وقد كان الاسم `print42` في الوحدة الأولى يشير إلى كائن الدالة المعرَّف في الوحدة، (وسننظر في ذلك بالتفصيل حين نشـرح تعبير لامدا في الفصل الحادي والعشرين: البرمجة الوظيفية)، ومـع أننـا اسـتوردنا الاسـم إلى وحـدتنا، إلا أننـا لم نستورد الدالة التي لا تزال تشير إلى نسخة الوحدة الخاصة بها من `spam`، وعلى ذلـك فقـد أنشـأنا متغـير spam جديد ليس لديه تأثير على الدالة المشار إليها بالاسـم `print42`. يشـرح المخطـط التـالي هـذا لكلا النـوعين من الاستيراد، ويمكنك ملاحظة كيف يكرر الاستيراد الثاني الاسم `print42` من `first.py` إلى `second.py`:



ينبغي أن يكون هذا اللبس قد شرح سهولة الوصول إلى الأسماء في الوحدات المسـتوردة باسـتخدام ترمـيز النقطة dot notation. رغم أننا سنكتب شيفرةً أكثر، لكن توجد وحدات قليلة -مثل `Tkinter` الذي سـنتعرض لها لاحقًا-، تُستخدم لاستيراد جميع الأسماء، لكنها تُكتب بطريقة تقلل خطر تعارض الأسماء رغم وجـود الخطـورة على أي حال، كما قد تنشأ عنها علل برمجية bugs يصعب العثور عليها، وعلى أي حـال توجـد طريقـة أكـثر أمانًـا لاستيراد اسم وحيد من وحدة ما، كما يلي:

```
from sys import exit
```

وهنا نأتي بدالة `exit` فقط إلى فضاء الاسم المحلي، لكن لا نسـتطيع اسـتخدام أسـماء أخـرى من `sys`، ولا حتى `sys` نفسها.

15.1.2 تجنب تعارض الأسماء

إذا كانت الدالة تشير إلى متغير اسمه X، مع وجود X آخر في الدالة أي في النطاق المحلي، فسيكون هذا الأخير هو الذي ستراه بايثون وتستخدمه، وهنا يقع على عاتق المبرمج تجنب تعارض الأسماء، بحيث إذا كان لدينا متغيران أحدهما محلي والآخر متغير وحدة لهما الاسم نفسه، فلا نطلبهما في نفس الدالة، إذ سيغطي المتغير المحلى على اسم الوحدة.

ولن نواجه مشكلةً إذا أردنا قراءة متغير عام داخل دالة، إذ ستبحث بايثون عن الاسم محليًا، فإذا لم تجده فستبحث في النطاق العام، بل وفي النطاق المضمَّن كذلك إذا دعت الحاجة، وإنما ستظهر المشكلة عندما نريد إسناد قيمة إلى متغير عام، إذ سينشئ هذا متغيرًا محليًا جديدًا داخل الدالة، فكيف نسند القيمة إلى متغير عام دون إنشاء متغير محلي بنفس الاسم إذًا؟ يمكن تنفيذ هذا باستخدام الكلمة المفتاحية `global`:

```
var = 42
def modGlobal():
   global var  # تمنع إنشاء var محلي
   var = var - 21

def modLocal():
   var = 101

print( var )   # تطبع 42
modGlobal()
print( var )  # تطبع 21
modLocal()
print( var )  # تطبع 21
```

نرى هنا كيف غيرت الدالة `modGlobal` من المتغير العام، على عكس الدالة `modLocal` التي لم تغيره، لأنها تنشئ متغيرها الداخلي الخاص بها وتسند قيمةً إليه، ثم يُجمع هذا المتغير في نهاية الدالة ليُحذف، ويكون غير مرئي في مستوى الوحدة، لكن عمومًا ينبغي أن نقلل من استخدام تعليمات `global`، فمن الأفضل أن نمرر المتغير معامِلًا للدالة، ثم نعيد المتغير المعدَّل. يعيد المثال التالي كتابة دالة `modGlobal` دون استخدام تعليمة `global`:

```
var = 42
def modGlobal(aVariable):
    return aVariable - 21
```

```
print( var )
var = modGlobal(var)
print( var )
```

في هذه الحالة نسند القيمة التي أعادتها الدالة إلى المتغير الأصلي في نفس الوقت الذي نمررها فيه وسيطًا، وتكون النتيجة نفسها، لكن لا تعتمد الدالة الآن على أي شيفرة خارجها، مما يسهل إعادة استخدامها في برامج أخرى، كما يسهل رؤية كيف تتغير القيمة العامة global value، حيث نستطيع أن نرى حدوث الإسناد الصريح هنا، ويطبق المثال التالي كل ذلك عمليًا، إذ يوضح النقاط التي شرحناها إلى الآن، لهذا ادرسه جيدًا حتى تعرف استخدام الأسماء والقيم في كل خطوة فيه:

```
# متغيرات نطاقها الوحدة
W = 5
Y = 3

# المعامِلات تشبه متغيرات الدوال
# وعليه يكون لـ X نطاق محلي
def spam(X):

    # أخبر الدالة أن تنظر في مستوى الوحدة
    # وألا تنشئ وحدة W خاصة بها
    global W

    Z = X*2 # أنشأنا متغير Z الذي له نطاق محلي
    W = X+5 # استخدم الوحدة W كما شرحنا أعلاه

    if Z > W:
        # pow هو اسم نطاق مضمَّن
        print( pow(Z,W) )
        return Z
    else:
        return Y # لا يوجد Y محلي
                 # لذا نستخدم نسخة الوحدة
print("W,Y = ", W, Y )
for n in [2,4,6]:
    print( "Spam(%d) returned: " % n, spam(n) )
    print( "W,Y = ", W, Y )
```

15.1.3 فضاء الاسم في VBScript

إذا صرحنا عن متغير خارج الدالة أو البرنامج الفرعي في VBScript، فسيكون عامًا globally أما إذا صرحنا عنه داخل الدالة أو البرنامج الفرعي، فسيكون محليًا في الوحدة وسيخفي أي متغير عام له نفس الاسم، كما سيكون المبرمج هنا هو المسؤول عن إدارة التعارض بين هذه الأسماء، وبما أن متغيرات VBScript تُنشأ باستخدام تعليمة `Dim` فلن يحدث غموض أو لبس حول المتغير المقصود على عكس بايثون، وبهذا نرى أن منظور VBScript لقواعد النطاقات أبسط وأوضح من بايثون، لكن هناك بعض الأمور الخاصة بصفحات الويب، حيث ستكون المتغيرات العامة مرئيةً في كامل الملف، وليس داخل حدود الوسم `script` الذي عُرِّفت فيه فقط، وتوضح الشيفرة التالية ذلك:

```
<script type="text/vbscript">
Dim aVariable
Dim another
aVariable = "This is global in scope"
another = "A Global can be visible from a function"
</script>

<script type="text/vbscript">
Sub aSubroutine
   Dim aVariable
   aVariable = "Defined within a subroutine"
   MsgBox aVariable
   MsgBox another  ' uses global name
End Sub
</script>

<script type="text/vbscript">
MsgBox aVariable
aSubroutine
MsgBox aVariable
</script>
```

توجد بعض مزايا النطاقات في VBScript، والتي نتيح بها إمكانية الوصول إلى المتغيرات بين الملفات المختلفة في صفحة ويب -من الفهرس مثلًا إلى المحتوى والعكس-، لكن لن نتحدث عن هذا المستوى من برمجة صفحات الويب هنا، لذا سنكتفي بالإشارة إلى وجود كلمات مفتاحية مثل `Public` و`Private`.

15.1.4 فضاء الاسم في جافاسكربت

تتبع جافاسكربت نفس القواعد تقريبًا، إذ تكون المتغيرات المصرح عنها داخل الدوال مرئيةً داخل تلك الدوال فقط؛ أما المتغيرات التي خارج الدوال فيمكن أن تراها الشيفرة التي خارج الدوال، إضافةً إلى إمكانية رؤيتها داخل الدوال أيضًا. وكما هو الحال في VBScript؛ ليس هناك تعارض أو غموض بشأن المتغير المقصود، لأن المتغيرات تُنشأ صراحةً باستخدام تعليمة var، والمثال التالي شبيه بمثال VBScript السابق لكنه مكتوب بلغة جافاسكربت:

```
<script type="text/javascript">
var aVariable, another;  // متغيرات عامة
aVariable = "This is Global in scope<BR>";
another = "A global variable can be seen inside a function<BR>";
function aSubroutine(){
   var aVariable;        // متغير محلي
   aVariable = "Defined within a function<BR>";
   document.write(aVariable);
   document.write(another);  // يستخدم متغيرًا عامًا
}
document.write(aVariable);
aSubroutine();
document.write(aVariable);
</script>
```

ولا أظننا بحاجة إلى تكرار شرح المثال مرةً أخرى هنا.

15.2 خاتمة

نرجو في نهاية هذا الفصل أن تكون تعلمت ما يلي:

- النطاقات وفضاءات الأسماء وجهان لعملة واحدة، ويشيران إلى نفس الشيء.
- المفاهيم واحدة بين اللغات المختلفة، لكن الذي يختلف هو التطبيق الدقيق لها وفق قواعد كل لغة.
- تحتوي بايثون على خمسة نطاقات هي: النطاق المضمَّن built in، ونطاق الصنف class، والنطاق المتشعب nested، ونطاق الملف أو النطاق العام global، ونطاق الدالة أو النطاق المحلي function، وهذه النطاقات الثلاثة الأخيرة هي أهم نطاقات فيها من حيث كثرة الاستخدام في البرمجة.
- تحتوي كل من جافاسكربت وVBScript على نطاقين لكل واحدة منهما، هما نطاق الملف أو النطاق العام file، ونطاق الدالة أو النطاق المحلي function.

16. التعابير النمطية في البرمجة

تعرَّف التعابير النمطية بأنها مجموعات من المحارف التي تصف مجموعـةً أخـرى من المحـارف أكـبر منهـا، وهي تصف نمط المحارف الـذي نسـتطيع البحث عنـه في متن نص مـا، وهي تشـبه مفهـوم المحـارف البديلـة wildcards المستخدمة في تسمية الملفات على أغلب نظم التشغيل، حيث يمكن استخدام محـرف النجمـة * لتمثيل أي تسلسل من المحارف في اسم ملف، ولهذا فإن `*.py` تعني أي ملـف ينتهي بالامتـداد `.py`، بـل إن المحارف البديلة ما هي إلا مجموعة فرعية صغيرة من التعابير النمطية.

وتدعم أغلب لغات البرمجة الحديثة التعابير النمطية ضمنيًا، أو لديها مكتبات أو وحـدات متاحـة للاسـتخدام في البحث عن النصوص واستبدالها وفقًا لتعابير نمطية، وذلك بسـبب الإمكانيـات الكبـيرة لهـذه التعـابير، لكن شرحها المفصل هو خارج نطاق حديثنا، وستجد مصادر أخرى تتحدث عنها بتوسع شديد، وننصحك هنا بمراجعة كتب مثل كتاب أورايلي في التعابير النمطية وهو باللغة الإنجليزية، إضافةً إلى المقالات الموجـودة في أكاديميـة حسوب.

ولعل إحدى الخصائص المميزة للتعابير النمطية هي أنها تُظهـر أوجـه التشـابه مـع الـبرامج في البنيـة، فهي أنماط مبنية من وحدات أصغر منها، وتلك الوحدات هي:

- محارف منفردة.
- محارف بديلة.
- نطاقات أو مجموعات من المحارف، أو مجموعات محاطة بأقواس.

وبما أن المجموعة نفسها ما هي إلا وحدة، فيمكن أن تكون لدينا مجموعـات من المجموعـات إلى أن نصـل إلى مستوى تعقيد كبير، ونستطيع جمع تلك الوحدات بطرق تشبه استخدام التسلسلات أو التكرارات أو العوامل الشرطية في لغات البرمجة، وسننظر في كل منها في حينه، فإضافةً إلى شرح مفهوم التعابير النمطية، سـنعرف

كيف نستخدمها في برامج بايثون، وننظر كيف تدعمها لغتا VBScript وجافاسكربت، ولنستطيع تجربة الأمثلة هنا يجب أن نستورد وحدة `re` ونستخدم التوابع الخاصة بها، وسنفترض أنك استوردتها تلقائيًا دون ذكر ذلك في كل مرة.

16.1 التسلسلات

16.1.1 تسلسلات المحارف

لا شك أن أبسط بنية برمجية يمكن تصورها هي تسلسل من المحارف، كما أن أبسط تعبير نمطي ما هو إلا تسلسل من المحارف كذلك:

```
red
```

وهذا سيطابق أو يبحث في سلسلة نصية عن أي حدوث لهذه الأحرف الثلاثة التي تتكون منها كلمة red على الترتيب، وبناءً على ذلك سيجد كلمات مثل red وlettered وcredible، لأنها تحتوي على كلمة red ضمنها. ولنتحكم أكثر في خرج المطابقات، فإننا نوفر بعض المحارف الخاصة التي تُعرف باسم المحارف الوصفية metacharacters للحد من نطاق البحث:

التعبير	المعنى	مثال
`^red`	في بداية السطر فقط	**red** ribbons are good
`red$`	في نهاية السطر فقط	I love **red**
`\Wred`	في بداية الكلمة فقط	it's **red**irected by post
`red\W`	في نهاية الكلمة فقط	you cove**red** it already

يُطلق على هذه المحارف اسم المرابط anchors لأنها تثبت موضع التعبير النمطي في جملة أو كلمة ما، وهناك عدة مرابط أخرى معرَّفة في توثيق وحدة `re` يمكنك الاطلاع عليها.

16.1.2 المحارف البديلة

قد تحتوي التسلسلات على محارف بديلة Wildcard Characters تحل محل أي محرف، والمحرف البديل هو نقطة `.` جرِّب الشيفرة التالية مثلًا:

```
>>> import re
>>> re.match('be.t', 'best')
<_sre.SRE_Match object at 0x01365AA0>
>>> re.match('be.t', 'bess')
```

تخبرنا الرسالة التي في الأقواس السهمية أن التعبير النمطي 'be.t' -الممرَّر وسيطًا أول- يطابق السلسلة 'best' الممرَّرة وسيطًا ثانيًا، كما يطابق 'beat' و'bent' و'belt' وغيرها، لكن المثال الثاني لا يطابق لأن 'bess' لا تنتهي بحرف t، لذا لا يُنشئ MatchObject. يمكنك تجريب عدة مطابقات أخرى لتفهم كيفية عملها، ولاحظ أن match() لا تطابق إلا في بداية السلسلة النصية، أما لمنتصفها فنستخدم search() كما سنرى لاحقًا.

16.1.3 المجالات أو الفئات

يتكون المجال range (أو الفئة set) من تجميعة من المحارف المغلَّفة في أقواس مربعة، ويبحث التعبير النمطي عن أي محرف يكون داخل هذه الأقواس:

```
>>> re.match('s[pwl]am', 'spam')
<_sre.SRE_Match object at 0x01365AD8>
```

فهذا سيطابق swam أو slam، لكن لن يطابق sham لأن h غير موجودة في فئة التعبير النمطي.

أما إذا وضعنا محرف الإقحام ^ أول عنصر في المجموعة، فكأننا نقول أننا نريد البحث عن أي محرف عدا المحارف الموجودة في المجموعة:

```
>>> re.match('ool', 'cool')
<_sre.SRE_Match object at 0x01365AA0>
>>> re.match('ool','fool')
```

وبناءً على ذلك نستطيع مطابقة cool وpool، لكننا لن نطابق fool لأننا نبحث عن أي محرف عدا f في بداية النمط.

16.1.4 المجموعات

نستطيع جمع تسلسلات من المحارف أو الوحدات الأخرى معًا من خلال تغليفها بأقواس، وهي لا تقوم بدور العزل هنا، بل تفيدنا عند دمجها مع خصائص التكرار والشرطيات التي سنشرحها لاحقًا.

16.2 التكرار

يمكن إنشاء تعابير نمطية تطابق تسلسلات مكررةً من المحارف باستخدام بعض المحارف الوصفية الخاصة، ونستطيع البحث بها عن تكرار محرف واحد أو مجموعة محارف:

التعبير	المعنى	مثال
'?'	محرف واحد على الأكثر من المحارف السابقة -أي عدد صفر أو واحد من المحارف-، انتبه إلى الجزء الصفري	python!?y يطابق: **pythony** **pythonly**

التعبير	المعنى	مثال
	هنا لأنه قد يربكك.	
'*'	يبحث عن صفر محرف سابق أو أكثر.	pythonl*y يطابق ما ذكر في السطر السابق بالإضافة إلى : **pythonlly** **pythonllly** ...إلخ.
'+'	يبحث عن محرف واحد أو أكثر من المحارف السابقة.	pythonl+y يطابق: **pythonly** **pythonlly pythonllly** ...إلخ.
{n,m}	يبحث عن نطاق من التكرارات من n إلى m من المحارف السابقة.	fo{1,2 } يطابق : **fo** أو **foo**

يمكن تطبيق جميع محارف التكرار هذه على مجموعات أخرى من المحارف، وبناءً عليه:

```
>>> re.match('(.an){1,2}s', 'cans')
<_sre.SRE_Match object at 0x013667E0>
```

يكون النمط هنا `(.an){1,2}s` الذي يقول إن لدينا مجموعةً تتكون من أي محـرف متبـوع بـالحرفين an، ونريد أن نجد مجموعةً أو اثنتين متبوعتين بالحرف s، وسيطابق هذا النمط: cancans وpans وcanpans، لكن لن يطابق bananas لانعدام وجود حرف قبل مجموعة an الثانية فيها. جرب تعديل البحث ليطابق bananas.

> انظر في محددات التكرار الأخرى، ولا تنس a الإضافية التي في آخر bananas.

هناك مشكلة واحدة مع نمط التكرار `{m,n}`، وهـو أنـه لا يحـد المطابقـة لعـدد n من الوحـدات، وعلى ذلـك سيطابق مثال `fo{1,2}` الذي في الجدول السابق `fooo` لأنه يطابق `foo` في بداية `fooo`، وعليه فإذا أردنا الحد من عدد المحارف المطابَقة، فسنحتاج إلى أن نُتبع تعبير الضرب بمرساة أو نطاق نفي negated range.

وفي حالتنا، فإن `fo{1,2}[^o]` سيمنع `fooo` من المطابقة بما أنه يقول طـابق حـرف o واحـد أو حـرفين متبوعين بأي شيء غير o، لكن يجب أن يكون متبوعًا بشيء ما، لذا فإن `foo` لم تَعُـد مطابقـةً الآن. يوضـح هـذا الطبيعة المتقلبة للتعابير النمطية، فقد يصعب ضبطها للحصول على الخرج الذي نريده، ويجب أن نضعها تحت اختبارات فاحصة لتجنب الأخطاء.

أمـــا النمـــط الفعلي المطلـــوب للســـماح بمطابقـــة `foo` و`foobar` مـــع اســـتثناء `fooo`، فهو `'fo{1,2}[^o]*$'`، وهذا يعني `fo` أو `foo` المتبوعين بعدد صفر أو أكـثر من حـروف o ونهايـة السـطر، وفي الواقع حتى هذا النمط ليس تامًا وخاليًا من الأخطاء -جرب `fooboo` مثلًا-، لكن نحتـاج أن نشـرح بعض العناصـر الأخرى قبل أن نحسّنه ونعدل فيه.

16.2.1 التعابير الجشعة

يقال إن التعابير النمطية جشعة، أي أن دوال البحث والمطابقة ستطابق كل ما تستطيعه من السلسلة النصية، بدلًا من التوقف عند أول مطابقة تامة، وهذا لا يهم غالبًا، لكننا سنحصل على مطابقات أكثر من المطلوب عند جمع المحارف البديلة مع عوامل التكرار. لننظر في المثال التالي: إذا كان لدينا تعبير نمطي مثل a.*b الذي يقول إننا نريد إيجاد a متبوعًا بأي عدد من المحارف إلى أن نصل إلى حرف b، فستبحث دالة المطابقة من أول a إلى آخر b، مما يعني أنه إذا احتوت سلسلة البحث على أكثر من b فستُضمَّن جميعًا في الجزء .* من التعبير عدا آخر واحدة، وعليه:

```
re.match('a.*b','abracadabra')
```

فقد طابق MatchObject في هذا المثال كل abracadab وليس أول ab فقط، وسلوك المطابقة الجشع هذا هو أكثر الأخطاء التي يرتكبها المبرمج في بداية استخدامه للتعابير النمطية، ولمنع هذا السلوك الجشع نضيف '?' بعد محرف التكرار، كما يلي:

```
re.match('a.*?b','abracadabra')
```

مما سيطابق ab فقط.

16.3 الشرطيات

يتبقى لدينا الآن أن نجعل التعبير النمطي يبحث في عناصر اختيارية أو يختار نمطًا من بين عدة أنماط، وسننظر في كل منها على حدة.

16.3.1 العناصر الاختيارية

يمكن تحديد محرف ما ليكون اختياريًا باستخدام عدد صفر أو أكثر من محارف التكرار الوصفية:

```
>>> re.match('computer?d?', 'computer')
<re.MatchObject instance at 864890>
```

هذا سيطابق compute وcomputer وcomputed، كما سيطابق computerd، لكننا لا نريد هذه الأخيرة، لذا سنضيق النطاق الذي نريده كما يلي:

```
>>> re.match('compute[rd]$','computer')
<re.MatchObject instance at 874390>
```

وهذا سيختار computer وcomputed فقط، ويرفض computerd، وإذا أضفنا ? بعد هذا النطاق فسنسمح باختيار compute مع تجنب computerd أيضًا.

16.3.2 التعابير الاختيارية

بالإضافة إلى خيارات المطابقة من قائمة محارف السابقة الذكر، يمكن المطابقة بناءً على اختيار من تعابير فرعية، فقد ذكرنا سابقًا أننا نستطيع جمع تسلسلات من المحارف في أقواس، لكن الواقع أننا نستطيع جمع أي تعبير نمطي عشوائي بين أقواس ومعاملته مثل وحدة، وسنستخدم الصيغة `(RE)` أثناء شرح التركيب اللغوي هنا للإشارة إلى أي تجميع لتعابير نمطية، والحالة التي نريد دراستها هنا هي مطابقة تعبير نمطي يحتوي على `(RE)xxxx` أو `(RE)yyyy` حيث تكون `xxxx` و`yyyy` أنماطًا مختلفةً، وبناءً عليه فإذا أردنا مطابقة `premature` و`preventative` فسنفعل هذا بواسطة محرف الاختيار الوصفي `|`:

```
>>> regexp = 'pre(mature|ventative)'
>>> re.match(regexp,'premature')
<re.MatchObject instance at 864890>
>>> re.match(regexp,'preventative')
<re.MatchObject instance at 864890>
>>> re.match(regexp,'prelude')
```

نلاحظ أنه عند تعريف التعبير النمطي تعين علينا أن ندرج النص الكامل لكلا الخيارين داخل أقواس بدلًا من `(e|v)`، ولو لم نفعل لاقتصر الخيار على `prematureentative` و`prematurventative` فقط، أي كان الحرفان فقط هما اللذان سيمثلان الخيارات المتاحة، وليس المجموعات كلها.

نستطيع الآن باستخدام هذه التقنية أن نعود إلى المثال أعلاه الذي أردنا فيه التقاط `fo` أو `foo` وتجنب التقاط `fooo` إضافةً إلى أي شيء يأتي بعدها، وقد تركناها مع تعبير نمطي يتكون من `fo{1,2}[^o]*$`، والمشكلة هنا أن التطابق سيفشل إذا احتوت السلسلة النصية التالية لـ `fo` أو `foo` على `o`، لكن يمكن الالتفاف على ذلك باستخدام عدة خيارات من التعبيرات، ونريد أن ينجح التطابق سواء كان النمط في نهاية السطر أو متبوعًا بأي حرف سوى `o`، وسيبدو ذلك كما يلي:

```
fo{1,2}($|[^o])
```

والذي سيعطينا أخيرًا ما نريده، تجدر الإشارة إلى أنه يجب تنفيذ اختبارات كافية عند استخدام التعابير النمطية، لضمان عدم التقاط أي شيء غير مرغوب فيه، وأننا نلتقط كل ما نريد التقاطه.

16.4 المزيد من الملاحظات حول التعابير النمطية

تحتوي وحدة `re` على مزايا كثيرة لم نذكرها هنا، يجدر النظر فيها ودراستها من توثيق الوحدة نفسها، لكننا نريد تسليط الضوء على مجموعة من الرايات flags التي يمكن استخدامها عند تصريف التعبيرات مع دالة `re.compile()`، والتي تتحكم في أمور مثل مطابقة النمط في الأسطر المختلفة، أو تجاهله لحالة الأحرف، أو غير ذلك.

ومن المهم استخدام أداة تختبر التعابير النمطية للتحقق من نتائجها، وتوجد أدوات عديدة منها على الـويب، لكن نخص بالذكر منها أداة regex101.com، حيث نكتب فيها تعبـيرًا نمطيًـا وسلسـلة اختبـار، ثم نـرى الأجـزاء التي طابقها التعبير من السلسلة النصية، وهذه الأداة تحديدًا تعطينا وصـفًا مفيـدًا عمـا يفعلـه التعبـير النمطي، وتسمح لنا باختيار أصناف فرعية للمطابَقات الناتجة، وغيرها من المزايا المفيدة.

16.5 استخدام التعابير النمطية في بايثون

رأينا في السطور السابقة شيئًا يسيرًا من التعابير النمطية، ونريد أن نطبق ذلك في بـايثون، حيث نسـتطيع استخدامها أداة بحث قويةً جدًا في النصوص، إذ يمكن البحث عن صور كثيرة مختلفة لسلاسل نصية في عمليـة واحدة، بل يمكن البحث عن المحارف التي لا تُطبـع مثـل الأسـطر الفارغـة باسـتخدام بعض المحـارف الوصـفية المتاحة، كما يمكن اسـتبدال هـذه الأنمـاط باسـتخدام التوابـع والـدوال الخاصـة بوحـدة `re`، كمـا رأينـا في دالـة `match()` أعلاه، وفيما يلي بعض الدوال الأخرى المتاحة:

الدالة - التابع	التأثير
`match(RE,string)`	يعيد كائن مطابقة إذا طابق التعبير النمطي بداية السلسلة.
`search(RE,string)`	يعيد كائن مطابقة إذا وُجد تعبير نمطي في أي مكان في السلسلة النصية.
`split(RE, string)`	مثل `string.split()` لكن يستخدم تعبيرًا نمطيًا مثل فاصل.
`sub(RE, replace, string)`	تعيد سلسلةً نصيةً أُنتجت عن طريق استبدال لـ re في أول مطابقة للتعبير النمطي، وهذه الدالة لها مزايا أخرى إضافية، انظر توثيقها للمزيد.
`findall(RE, string)`	تبحث عن جميع مرات حدوث التعبير النمطي في سلسلة نصية، وتعيد سلسلةً من كائنات المطابقة.
`compile(RE)`	تنتج كائن تعبير نمطي يمكن إعادة استخدامه لعدة عمليات مع نفس التعبير النمطي، ويكون للكائن جميع التوابع أعلاه لكن مع re مضمَّنة، ويكون أكثر كفاءةً من النسخ الخاصة بالدالة.

لا شك أن هذه القائمـة لا تحتـوي جميـع توابـع re ودوالهـا، كمـا أن التوابـع الـتي ذكرناهـا في الجـدول لهـا معامِلات اختيارية لتوسيع استخدامها، واخترناها لأنها أكثر العمليات استخدامًا، ومناسبةً لاحتياجاتنا.

16.5.1 مثال عملي على التعابير النمطية

لننشئ برنامجًا يبحث في ملف HTML عن وسم IMG ليس له قسـم ALT، فـإذا وجـدنا واحـدًا فسنضـيف رسالةً إلى المالك ليكتب ملفات HTML أفضل في المستقبل.

```
import re
# التقط IMG أو img لنسمح بصفر مسافة أو أكثر بين
# I و >
```

```
img = '< * '

# alt أو ALT السماح بأي عدد من المحارف حتى
before >
alt = img + '.*[aA][lL][tT].*>'

# افتح الملف واقرأه في قائمة
filename = input('Enter a filename to search ')
inf = open(filename,'r')
lines = inf.readlines()

# ALT بدون IMG إذا احتوى السطر على وسم
# HTML فأضف رسالتنا كتعليق
for index,line in enumerate(lines):
  if ( re.search(img,line) and not
       re.search(alt,line) ):
        lines[index] += '<!-- PROVIDE ALT TAGS ON IMAGES! -->\n'

# والآن اكتب الملف المعدَّل.
inf.close()
outf = open(filename,'w')
outf.writelines(lines)
outf.close()
```

لـدينا ملاحظتـان على الشـيفرة أعلاه نريـد الإشـارة إليهمـا، الأولى أننـا اسـتخدمنا `re.search` بـدلًا من `re.match`، لأن الأولى تبحث عن الأنماط في أي مكان داخل السلسلة النصـية، بينمـا تبحث الثانيـة في بدايـة السلسلة فقط، أما الملاحظة الثانية فهي أننا وضعنا زوجًا خارجيًـا من الأقـواس حـول الاختبـارين، وهـو أمـر غـير ضروري لكنه يسـمح لنـا بتقسـيم الاختبـار إلى سـطرين، ممـا يجعلـه أسـهل في القـراءة خاصـةً إذا كنـا سـندمج تعبيرات كثيرةً.

وهذه الشيفرة ليست مثاليةً لأنها لا تأخذ في الحسبان الحالة التي يكون وسـم `img` فيهـا مقسـمًا على عـدة أسطر، لكنها تكفي لشرح التقنية عمومًا، على أنه يُفضل تجنب هذا "التخريب" الذي قمنا بـه في ملـف HTML، لكن الذي ينسى وسوم `alt` يستحق جزاءه.

نأتي لأمر أخير، وهو حدود كفاءة التعابير النمطية، فلهياكل البيانات المعرَّفـة بوضـوح -مثـل HTML- أدوات أخـرى غـير التعـابير النمطيـة، تُعـرف باسـم المحلِّلات تكـون أكـثر كفـاءةً وأسـهل في الاسـتخدام دون أخطـاء، وسنستخدم محلل HTML في الفصل التاسع والعشرين: برمجة عملاء الويب، وتتجلى فائدة التعابير النمطية في

عمليات البحث المعقدة في النصوص الحرة إذ تحل لنا مشاكل كثيرة، مع التأكيد مرةً أخرى على الاختبار المفصل لها، كما لا يجب استخدامها إلا عند الحاجة الضرورية إليها، أما إذا كنا نريد البحث عن سلسلة بسيطة فنستخدم التابع `find`، لتجنب مشاكل التعابير النمطية.

وسنعود مرةً أخرى إلى التعابير النمطية في دراسة الحالة لعدّاد القواعد النحوية، لذا جرب استخدامها حتى ذلك الحين، وتفقد التوابع الأخرى الموجودة في وحدة re، فلم نشرح حتى الآن إلا قشور هذه الأدوات بالغة القوة في معالجة النصوص.

16.6 التعابير النمطية في جافاسكربت

تدعم جافاسكربت التعابير النمطية ضمنيًا وبقوة، بل إن عمليات البحث في السلاسل النصية التي استخدمناها من قبل ما هي إلا بحوث تعابير نمطية، فقد استخدمنا أبسط صورة لها "تسلسل بسيط من المحارف"، وتنطبق جميع القواعد التي ذكرناها في بايثون على جافاسكربت، عدا أن التعابير النمطية هنا تكون محاطةً بشرطة مائلة / بدلًا من علامات الاقتباس:

```
<script type="text/javascript">
var str = "A lovely bunch of bananas";
document.write(str + "<BR>");
if (str.match(/^A/)) {
   document.write("Found string beginning with A<BR>");
}
if (str.match(/b[au]/)) {
   document.write("Found substring with either ba or bu<BR>");
}
if (!str.match(/zzz/)) {
   document.write("Didn't find substring zzz!<BR>");
}
</script>
```

ينجح التعبيران الأولان ويفشل الثالث، لذا حصلنا على الاختبار السلبي، لاحظ علامة التعجب في البداية.

16.7 التعابير النمطية في VBScript

لا تحتوي VBScript على دعم مضمّن للتعابير النمطية كما في جافاسكربت، لكن فيها كائن تعبير نمطي يمكن بدؤه واستخدامه للبحث وعمليات الاستبدال وغيرها، كما يمكن التحكم فيه لتجاهل حالة الأحرف وللبحث في جميع النسخ أو نسخة واحدة فقط:

```
<script type="text/vbscript">
Dim regex, matches
Set regex = New RegExp

regex.Global = True
regex.Pattern = "b[au]"

Set matches = regex.Execute("A lovely bunch of bananas")
If matches.Count > 0 Then
   MsgBox "Found " & matches.Count & " substrings"
End If
</script>
```

نكون بهذا قد وصلنا إلى نهاية هذا الفصل مكتفين بما ذكرناه فيه، لكن نعيد التأكيد على أن التعابير النمطية غنية بالتعقيدات الدقيقة التي لا يمكن تغطيتها في هذا الفصل القصير، ويمكن الرجوع إلى المصـادر الموجـودة في الويب لمزيد من المعلومات عن استخدامها، إضافةً إلى كتاب أورايلي الذي أوردناه في بداية الفصل.

16.8 خاتمة

بنهاية هذا الفصل نود أن تكون قد تعلمت:

- التعابير النمطية هي أنماط نصية تستطيع تطوير قوة وكفاءة عمليات البحث النصية.
- يصعب التحكم بالتعابير النمطية، وقد تتسبب في علل برمجية غريبة، لذا يجب التعامل معها بحرص.
- التعابير النمطية ليست الحل السهل لكل مشكلة، بل قد يكون الحـل في منظـور أكـثر تعقيـدًا، فـإذا لم ينجح استخدام التعابير النمطية في حل مشكلة لثلاث محاولات متتالية، فيجب البحث عن حل آخر.

17. البرمجة كائنية التوجه

رغم أن الأفكار التي كانت وراء البرمجة الكائنية التوجه طُورت في ستينيات القرن الماضي إلا أنها لم تشتهر في الوسط البرمجي إلا بعد ذلك بعقدين، أي في الثمانينيات، بعد إطلاق Smalltalk-80 ومجموعة متنوعة من تطبيقات لغة Lisp، ولم تكن وقتها اتجاهًا سائدًا في البرمجة وإنما كانت تثير الفضول فقط، ثم تغير ذلك عندما انتشرت الواجهات الرسومية في الحواسيب الشخصية على حواسيب أبل أولًا، ثم على الحواسيب العاملة بنظام ويندوز ونظام نوافذ X في يونكس، إلى أن اقتربنا من نهاية الألفية السابقة، حيث صارت البرمجة الكائنية التوجه Object Oriented Programming -والتي تعرف اختصارًا OOP- التقنية الأبرز لتطوير البرمجيات.

وتتجسد مفاهيم البرمجة الكائنية التوجه في لغات مثل جافا وC++ وبايثون بحيث لا تكاد تفعل شيئًا فيها إلا وتقابل كائنًا في مكان ما، ونريد في هذا الفصل أن نتعرف على هذه التقنية وننظر في المفاهيم الأساسية لها، مثل تعريف الكائن والصنف وتعددية الأشكال polymorphism والوراثة inheritance، وكيفية إنشاء الكائنات وتخزينها واستخدامها، والبرمجة الكائنية التوجه موضوع كبير قد كُتبت فيه كتب، فإذا أردت التعمق فيه أكثر مما هو مذكور في هذا الفصل فانظر هذه الكتب باللغة الإنجليزية:

- كتاب Object Oriented Analysis لبيتر كود Peter Coad وإد يوردون Ed Yourdon.
- كتاب Object Oriented Analysis and Design with Applications لجريدي بوش Grady Booch (الطبعة الأولى أو الثالثة).
- كتاب Object Oriented Software Construction لبرتراند ماير Berterand Meyer (الطبعة الثانية).

تختلف هذه الكتب عن بعضها في العمق والحجم والدقة الأكاديمية بالترتيب، فالكتاب الأول مناسب للأغراض العامة التي هي خارج نطاق العمل البرمجي، والحق أنها كلها ليست كتبًا في البرمجة، بل كتب تحليل

وتصميم برمجي، لأن أفضل تطبيق للبرمجة الكائنية التوجه يكـون بتطـبيق المبـادئ خلال دورة حيـاة المشـروع، وهذه طريقة مختلفة لحل المشاكل عن الطريقة العادية للبرمجة.

أما في هذا الفصل فسنتحدث باختصار عن مفاهيم البرمجة الكائنية التوجه التي ذكرناها، ولا مشـكلة إذا لم تستوعبها في البداية، فلا تزال تستطيع استخدام الكائنـات بـدون أن تسـتوعب المفهـوم الـذي بُـنيت عليـه، ثم ستتضح الأمور بالتدريج مع التدرب والوقت،كما يمكن استخدام تصميم كـائني التوجـه في لغـة غـير كائنيـة من خلال الاصطلاحات البرمجية، لكن لا يُنصح بهذا الأسلوب، وإنما يستخدَم ملاذًا أخيرًا عندما لا نجد حلًا آخـر، حيث تُستخدم التقنيات الكائنية التوجه إذا كانت المشكلة توافقها وتُحل بها، ويُفضل أن تُسـتخدم لغـة كائنيـة التوجـه أيضًـا عنـدها، وتـدعم أغلب اللغـات الحديثـة البرمجـة الكائنيـة التوجـه جيـدًا، بمـا فيهـا اللغـات الثلاثـة الـتي نتـدرب عليهـا، لكننـا سنسـتخدم بـايثون في الأمثلـة الـواردة هنـا، ثم سـنعرض المفـاهيم الأساسـية فقـط في جافاسكربت و VBScript.

17.1 جمع البيانات والدوال

الكائنات هي تجميعات من البيانات والدوال الـتي تنفـذ مهامًـا على تلـك البيانـات، وتوضـعان معًـا بحيث يمكن تمرير كائن من جزء ما في البرنامج كي نحصل تلقائيًا على وصول إلى العمليات المتاحة وسمات البيانات، وهذا الجمع بين البيانات والدوال هو أصل البرمجة الكائنية التوجه، ويُعرف باسم التغليف encapsulation، لأن بعض لغات البرمجة تخفي البيانـات عن مسـتخدمي الكـائن، وبنـاءً عليـه تتطلب توابـع الكـائن للوصـول إليهـا، وتسمى هذه التقنية باسم إخفاء البيانات، ويطلق عليها التغليف أحيانًا، وكمثـال على التغليـف، قـد يخـزن كـائن سلسلة نصية سلسلة محارف، لكنه يوفر كذلك توابع للعمل على هذه السلسلة؛ لتنفيذ أمور مثـل البحث وتغيـير حالة الأحرف وحساب الطول وغير ذلك، وتستخدم الكائنـات مجـازًا تمريـر الرسـالة message passing، حيث يمرر كائنٌ رسالةً إلى كائن آخر، ويرد الكائن المستقبل بتنفيذ تابع -وهو إحـدى عملياتـه-، وهكـذا يُسـتدعى التـابع عند استقبال الرسالة الموافقة له بواسطة الكائن المالـك، ويمكن تمثيـل ذلـك بصـيغ مختلفـة، وأكثرهـا شـيوعًا الصيغة النقطية . التي تحاكي الوصول إلى العناصر التي في الوحدات، فبالنسبة إلى صنف widget وهمي:

```
w = Widget() # أنشئ نسخة w جديدة من widget
w.paint()  # أرسل إليها الرسالة paint
```

ستستدعي هذه التعليمات التابع paint الخاص بكائن widget.

17.2 تعريف الأصناف

يمكن للكائنات أن تحتوي على أنواع مختلفة، كما أن للبيانات أنواعًـا مختلفـةً، وتُعـرَّف تجميعـات الكائنـات التي لها صفات متطابقة باسم الأصناف classes، ونستطيع تعريـف هـذه الأصـناف وإنشـاء نسـخ منهـا، حيث تكون تلك النسخ هي الكائنات الفعلية، كما يمكن تخزين مراجع إلى تلك الكائنات في المتغيرات داخل برامجنـا،

لننظر الآن في مثال حقيقي لنرى إن كنا نستطيع تفسيره وشرحه، حيث سننشئ صنف رسالة يحتوي على سلسلة نصية -تمثل نص الرسالة- وتابع لطباعة الرسالة.

```
class Message:
    def __init__(self, aString):
        self.text = aString
    def printIt(self):
        print( self.text )
```

الملاحظة الأولى: يسمى أحد توابع هذا الصنف باسم `__init__`، وهو تابع خاص يسمى الباني constructor، وسبب هذا الاسم أنه يُستدعى عند إنشاء أو بناء نسخة جديدة من كائن ما، وستكون المتغيرات المسندة داخل هذا التابع -والتي أنشئت داخل بايثون- متغيرات فريدة للنسخة الجديدة، وتوجد عدة توابع خاصة مثل هذا التابع في بايثون، وجميعها مميزة بصيغة التسمية التي فيها شرطتان سفليتان عن يمينها وشمالها `__xyz__`، ويطلق عليها مستخدمو بايثون أحيانًا اسم التوابع السحرية magic methods أو التوابع المحاطة بشرطين سفليتين dunder methods (إذ dunder اختصار إلى double under)، أما وقت استدعاء الباني الدقيق فيختلف بين اللغات، حيث يُستدعى التابع `init` في بايثون بعد إنشاء النسخة في الذاكرة، لكنه في لغات أخرى يعيد النسخة نفسها، والفرق في هذا بين اللغات طفيف ولا يستحق الانتباه له.

الملاحظة الثانية: يحتوي كلا التابعين المعرفين على معامل أول هو `self`، والاسم مجرد اصطلاح يشير إلى نسخة الكائن، وسنرى قريبًا أن هذا المعامِل لا يملؤه المبرمج، بل يُملأ بواسطة المفسر في وقت التشغيل، وعلى هذا يُستدعى `printIt` على نسخة للصنف -انظر أدناه-، بدون وسطاء بالشكل `m.printIt()`.

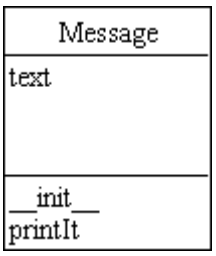
الملاحظة الثالثة: لقد استدعينا الصنف `Message` بحرف M كبير كما نرى، وهذا للسهولة فقط، لكن هذا الاصطلاح يُستخدم بكثرة في لغات البرمجة الكائنية التوجه، وليس في بايثون وحدها، ويوجد اصطلاح قريب من هذا يقتضي أن تبدأ أسماء التوابع بحرف صغير ثم تبدأ الكلمات التالية في الاسم بحرف كبير، فإذا كان لدينا تابع اسمه "calculate current balance" فسيُكتب بالشكل `calculateCurrentBalance`.

ننصحك عند هذه النقطة بالعودة إلى الفصل الخامس: البيانات وأنواعها، لقراءة قسم الأنواع المعرَّفة من قِبل المستخدم، حيث ستفهم مثال دليل جهات الاتصال في بايثون فهمًا أفضل بعد هذا الشرح هنا، فالنوع الوحيد الذي يعرّفه المستخدم في بايثون هو الصنف، والصنف الذي له سمات attributes وليس له توابع -ما عدا `__init__`- يكافئ الباني المسمى `record` أو `struct` في بعض لغات البرمجة.

17.3 الصيغة الرسومية

تبنّى مجتمع هندسة البرمجيات صيغةً مرئيةً لوصف الأصناف والكائنات وعلاقاتها بعضها ببعض، وتسمى هذه الصيغة باسم لغة النمذجة الموحدة Unified Modelling Language أو UML اختصارًا، وهي أداة تصميم

قوية وتحتوي على العديد من المخططات والأيقونات، وسننظر في بعضها هنا بما يعيننا على فهم المبادئ الـتي نريد شرحها فقط، وأول أيقونة سنقابلها في UML هي وصف الصنف، وهي أهم الأيقونات، وتتكون من صندوق من ثلاثة أجزاء، يحتوي الجزء العلوي على اسم الصنف، والأوسط على سماته أو البيانات فيه، أما الجـزء السـفلي فيحتوي على توابع الصنف أو دواله، وبناءً عليه سيبدو صنف Message المعرَّف أعلاه كما يلي:



سنرى في هذا الفصل أيقونات UML أخرى، ونتعرض لمفاهيم جديدة تدعمها هذه الصيغة.

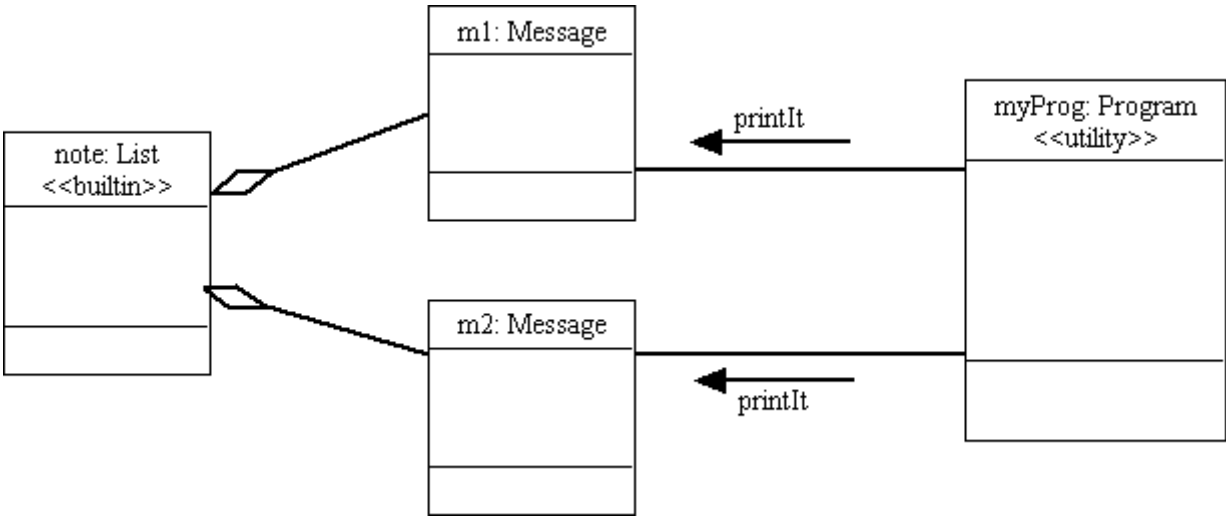
17.4 استخدام الأصناف

بما أننا عرّفنا الصنف Message فنستطيع إنشاء نسخ منه الآن والعمل عليها:

```
m1 = Message("Hello world")
m2 = Message("So long, it was short but sweet")

notes = [m1, m2] # ضع الكائنات في قائمة
for msg in notes:
    msg.printIt() # اطبع الرسائل متتابعة
```

وبهذا نعامل الصنف كما لو كان نوع بيانات قياسيًا في بايثون، وهو الغرض من التدريب ابتداءً.

كما توجد أيقونة للكائن أو النسخة في UML، وهي مثل أيقونة الصنف إلا أننا نـترك الجـزئين السـفليين في الصندوق فارغين، ويتكون الاسم من اسم الكائن أو النسخة متبوعًا بنقطـتين رأسـيتين ثم اسـم الصـنف، وعليـه فإن `m1:Message` تخبرنا أن `m1` ما هي إلا نسخة من الصنف `Message`، ويمكن رسم مثال الرسالة الخاص بنا الآن كما يلي:



نلاحظ أن الصنف `List` يمثل نوع القائمة القياسي في بايثون، كما هو موضح من وضع كلمة `builtin` بين أقواس حادة، وهي بنية معروفة في UML باسم القالب النمطي stereotype، وتشير الخطوط ذوات الرؤوس الماسية في الصورة إلى القائمة التي تحتوي على كائنات `Message`، وبالمثل فإن كائن `MyProg` يُنمَّط stereotyped على أنه صنف مساعد utility class، مما يعني في هذه الحالة أنه غير موجود مثل صنف داخل البرنامج، لكنه منتج من منتجات البيئة نفسها، وتُظهِر أدوات نظام التشغيل عادةً بهذه الطريقة، مثل مكتبات لدوال.

أما الخطوط المستقيمة التي تخرج من `myProg` إلى `Message` فتوضح أن الكائن `myProg` يرتبط بكائنات `Message` أو يشير إليها، وتشير الأسهم المرافقة لتلك الخطوط أن كائن `myProg` يرسل رسالة `printIt` إلى كل كائن من كائنات `Message`، وتُنقل رسائل الكائنات من خلال ارتباطات associations.

17.4.1 المعامل self

يطرح من يبدأ حديثًا في البرمجة الكائنية التوجه ببايثون سؤالًا هو: ما هو المعامل self؟ لأن تعريف أي تابع في صنف ما في بايثون يبدأ به، ويجب أن نبين أن الاسم نفسه مجرد اصطلاح، ولم يتغير إلى الآن لأن الثبات أمر محمود في الاصطلاحات البرمجية، فلغة جافاسكربت مثلًا لديها مفهوم مشابه لكنها تستخدم اسم `this` بدلًا من self.

أما self نفسه فهو مرجع إلى النسخة الحالية، فحين ننشئ نسخةً من الصنف فإنها تحتوي على بياناتها الخاصة كما أنشأها الباني، لكنها لا تحوي التوابع، لذا عندما نرسل رسالةً إلى نسخة وتستدعي التابع الموافق؛ فإنها تستدعيه إلى الصنف من خلال مرجع داخلي، فتمرر مرجعًا إلى نفسها self إلى التابع لتعرف شيفرة الصنف أي نسخة يجب أن تستخدمها.

لننظر الآن في مثال مألوف، فإذا كان لدينا تطبيق رسومي فيه كائنات أزرار كثيرة، فسيُفعَّل التابع المرتبط بكل زر عندما يضغطه المستخدم، ويعرف تابع الزر أي زر ضغِط بالإشارة إلى قيمة self التي ستكون مرجعًا إلى

نسخة الزر الحقيقي الذي ضُغط، وسنرى ذلك عمليًا حين نصل إلى الفصل التاسع عشر: برمجة الواجهات الرسومية.

وعند إرسال رسالة إلى كائن يحدث ما يلي:

- تستدعي شيفرة العميل النسخة، أي ترسل الرسالة إذا أردنا الحديث باصطلاح البرمجة الكائنية التوجه.
- تستدعي النسخة تابع الصنف، وتمرر مرجعًا إلى نفسها self.
- يستخدم تابع الصنف بعدها المرجع الممرَّر ليأخذ بيانات النسخة للكائن المستقبِل.

يمكن رؤية تلك النقاط عمليًا في تسلسل الشيفرة التالي، ونلاحظ أننا نستطيع استدعاء تابع الصنف صراحةً كما فعلنا في السطر الأخير:

```
>>> class C:
...   def __init__(self, val): self.val = val
...   def f(self): print ("hello, my value is:", self.val)
...
>>> # create two instances
>>> a = C(27)
>>> b = C(42)
>>> # first try sending messages to the instances
>>> a.f()
hello, my value is 27
>>> b.f()
hello, my value is 42
>>> # now call the method explicitly via the class
>>> C.f(a)
hello, my value is 27
```

نستطيع استدعاء التوابع كما نرى في المثال أعلاه من خلال النسخة، وتملأ بايثون معامِل self في تلك الحالة لنا، أو من خلال الصنف صراحةً، وفي تلك الحالة نحتاج إلى تمرير قيمة self صراحةً.

لدينا سؤال يطرح نفسه الآن، فإذا كانت بايثون تستطيع توفير مرجع بين النسخة وصنفها، ألا تستطيع أن تملأ self بنفسها أيضًا؟ ربما يكون هذا سؤالًا منطقيًا لكن الإجابة عليه هي أن Guido Van Rossum -منشئ اللغة- صممها هكذا! لكن مع هذا فإن العديد من لغات البرمجة الكائنية التوجه تخفي معامِل self، لكن بايثون تعتمد الصراحة explicity وتفضلها على الضمنية implicity، ويتعود المبرمج على هذا المبدأ مع كثرة العمل.

17.5 تعددية الأشكال polymorphism

لدينا الآن القدرة على تعريف أصنافنا الخاصة وإنشاء نسخ منها وإسنادها إلى متغيرات، ثم تمرير رسائل إلى تلك الكائنات تؤدي إلى تشغيل التوابع التي عرفناها، لكن هناك عنصر أخير يتعلق بالبرمجة الكائنية التوجه هنا، وهو الأهم فيها من نواحٍ عدة، فإذا كان لدينا كائنان من صنفين مختلفين لكنهما يدعمان نفس مجموعة الرسائل، لكن مع التوابع الموافقة لها، فنستطيع عندئذ أن نجمع تلك الكائنات معًا ونعاملها معاملةً واحدةً في برنامجنا، لكنها ستتصرف تصرفًا مختلفًا، وتُعرف تلك القدرة على التصرف المختلف لنفس رسائل الدخل باسم تعددية الأشكال، تُستخدم هذه الخاصية عادةً لجعل عدد من الكائنات الرسومية المختلفة ترسم نفسها عند استلام رسالة paint، فترسم الدائرة شكلًا مختلفًا تمامًا عن المثلث، لكن بما أن لهما نفس تابع paint فنستطيع -نحن المبرمجين- أن نتجاهل الاختلافات ونراهما على أنهما مجرد أشكال، لننظر الآن في مثال نحسب فيه مساحة الأشكال بدلًا من رسمها:

ننشئ أولًا الصنفين `Square` و `Circle`:

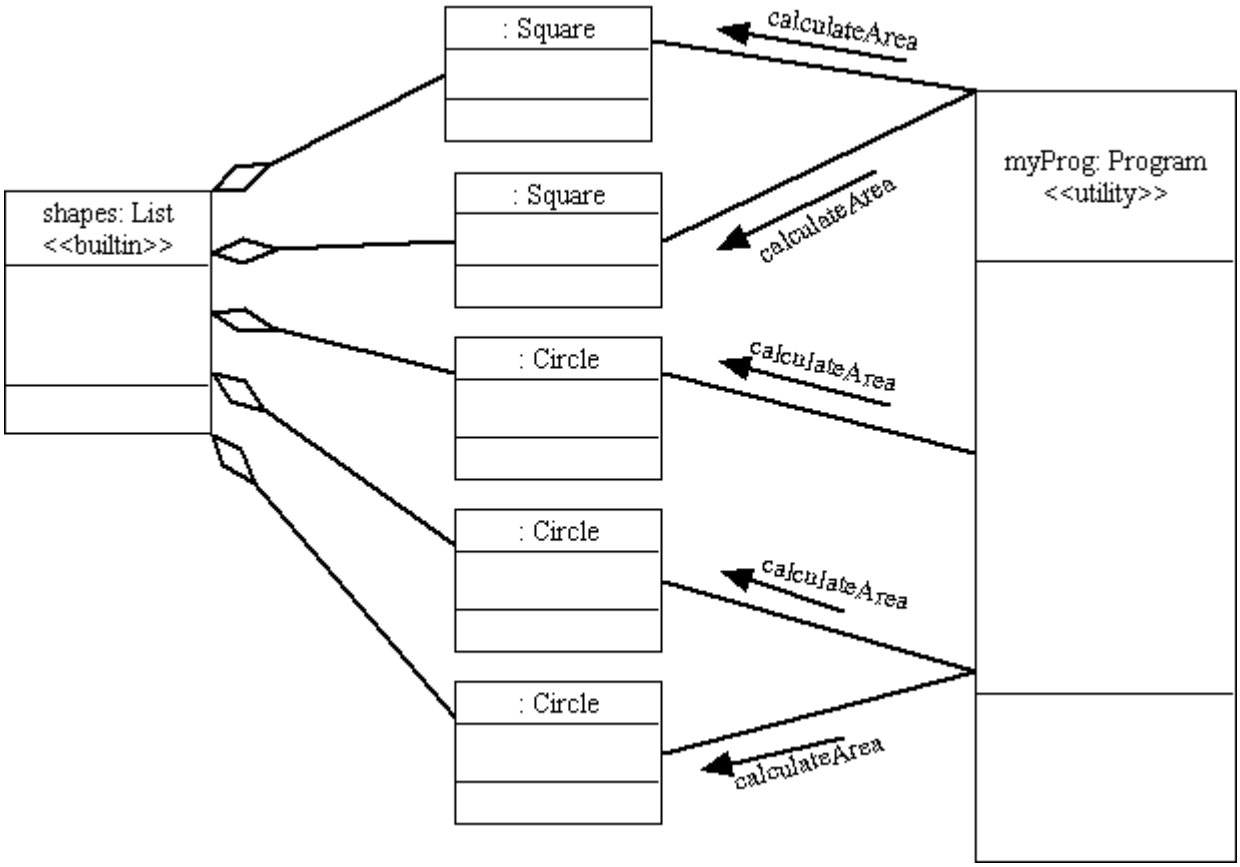
```
class Square:
    def __init__(self, side):
        self.side = side
    def calculateArea(self):
        return self.side**2

class Circle:
    def __init__(self, radius):
        self.radius = radius
    def calculateArea(self):
        import math
        return math.pi*(self.radius**2)
```

نستطيع الآن أن ننشئ قائمةً من الأشكال -دوائر أو مربعات- ثم نطبع مساحاتها:

```
shapes = [Circle(5),Circle(7),Square(9),Circle(3),Square(12)]

for item in shapes:
    print "The area is: ", item.calculateArea()
```

إذا جمعنا هذه الأفكار مع وحدات modules فسنحصل على آلية بالغة القوة لإعادة استخدام الشيفرة، حيث نضع تعريفات الصنف في وحدة ولتكن `shapes.py` مثلًا، ثم نستورد تلك الوحدة حين نرغب في التعديل على

الأشكال، وهذا بالضبط ما حصل مع العديد من وحدات بايثون القياسية، وهو السبب الذي يجعـل الوصـول إلى توابع كائن ما يشبه استخدام الدوال في وحدة.



نرى في الصورة أعلاه مخطط كائن أكثر تعقيدًا، ونلاحظ أن الكائنات التي داخل القائمة في هذه الحالـة ليس لها أسماء لأننا لم ننشئ متغيرات لها صراحةً، ففي تلك الحالـة نعـرض مسـافةً فارغـةً قبـل النقطين الرأسـيتين واسم الصنف، لكن هذا يجعل المخطط مزدحمًا، لذا لا نرسم مخططات الكائنات إلا عند الضرورة لتوضـيح بعض المزايا غير المألوفة للتصميم، أما في الأحوال العادية فنستخدم خصائص معقـدةً أكـثر من مخططـات الأصـناف لعرض العلاقات التي لدينا، كما سنرى في الأمثلة التالية.

17.6 الوراثة inheritence

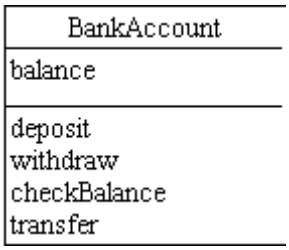
تُستخدم الوراثة (أو الاكتساب) عادةً لتنفيذ تعددية الأشكال واستخدامها، وقد تكون هي الآلية الوحيدة لذلك في العديد من لغات البرمجـة الكائنيـة، ويمكن للصـنف أن يـرث السـمات والعمليـات من صـنف أب parent class أو صنف رئيسي super class، وهذا يعني أن الصنف الجديد المطابق لصـنف آخـر في أغلب جوانبـه لا يجب أن يعيد تنفيذ جميع التوابع التي في الصنف الأول، بل يمكن أن يرث تلـك الإمكانيـات ثم يغيرهـا لتنفيـذ أمور مختلفة، كما في تابع `calculateArea` أعلاه، وسنستخدم للتوضيح مثـالًا فيـه هرميـة أصـناف حسـابات بنكية، حيث نستطيع إيداع المال والحصـول على الرصـيد والقيـام بعمليـات سـحب، ولبعض الحسـابات نسـبة ربوية (فائدة) سنفترض أنها تُحسب عند كل إيداع، إضافةً إلى بعض الرسوم الأخرى لعمليات السحب.

17.6.1 الصنف BankAccount

لنر الآن كيف سيبدو هذا المثال، سننظر في السمات والعمليات الخاصة بالحساب البنكي في أكثر مستوىً عام له، ومن الأفضل هنا أن ننظر في العمليات أولًا، ثم نوفر السمات حسب الحاجة لدعم تلـك العمليـات، فمـع الحساب المصرفي نستطيع القيام بما يلي:

- إيداع المال.
- سحب المال.
- التحقق من الرصيد الحالي.
- تحويل الأموال إلى حساب آخر.

وسنحتاج إلى معرِّف الحساب المصرفي ID للحساب الآخر والرصيد الحالي، بالنسبة للمعـرِّف فسنسـتخدم المتغير الذي نسند الكائن إليه، لكن إذا كنا في مشروع حقيقي فيجب إنشاء سمة خاصة بالمعرِّف تخـزن مرجعًـا فريدًا، كما سنحتاج إلى تخزين الرصيد، وعند تمثيل ذلك بلغة النمذجة الموحدة UML فسيبدو كما يلي:



نستطيع الآن أن ننشئ صنفًا يدعم ذلك:

```
# ننشئ صنف اعتراض مخصص Exception
class BalanceError(Exception):
      value = "Sorry you only have $%6.2f in your account"

class BankAccount:
    def __init__(self, initialAmount):
       self.balance = initialAmount
       print( "Account created with balance %5.2f" % self.balance )

    def deposit(self, amount):
       self.balance = self.balance + amount

    def withdraw(self, amount):
```

```
        if self.balance >= amount:
            self.balance = self.balance - amount
        else:
            raise BalanceError()

    def checkBalance(self):
        return self.balance

    def transfer(self, amount, account):
        try:
            self.withdraw(amount)
            account.deposit(amount)
        except BalanceError:
            print( BalanceError.value % self.balance )
```

الملاحظة الأولى: نتحقق من الرصيد قبل السحب، ونستخدم اعتراضًا لمعالجة الأخطاء، وبما أنه لا يوجد خطأ من النوع `BalanceError` في بايثون فسنحتاج إلى إنشاء واحد، وهو صنف فرعي من الصنف `Exception` مع قيمة نصية، وتُعرَّف قيمة السلسلة `value` سمةً لصنف الاعتراض لمجرد الاصطلاح فقط، وهي تضمن أننا نولد رسائل خطأ في كل مرة نرفع فيها اعتراضًا، ونلاحظ هنا أننا لم نستخدم `self` عند تعريف القيمة في `BalanceError` لأن `value` سمة مشتركة بين كل النسخ، وهي معرَّفة على مستوى الصنف وتُعرف بمتغير الصنف، ونصل إليها باستخدام اسم الصنف متبوعًا بنقطة `BalanceError.value` كما رأينا أعلاه، فعندما يولّد خطأ التعقب العكسي traceback -أي مسار مكان وقوع الخطأ ورجوعًا ضمن سلسلة الاستدعاءات- فسينتهي بطباعة سلسلة الخطأ المصاغة مع عرض الرصيد الحالي.

الملاحظة الثانية: يستخدم التابع `transfer` الدالة التابعة `withdraw/deposit` الخاصة بالصنف `BankAccount` أو توابعه لتنفيذ عملية التحويل، وهذا أمر شائع في البرمجة الكائنية التوجه ويُعرف بالمراسلة الذاتية self messaging، ويعني أن الأصناف المشتقة تستطيع تنفيذ نسخها الخاصة من `deposit/withdraw` لكن يظل التابع `transfer` كما هو لجميع أنواع الحسابات.

بما أننا عرّفنا `BankAccount` صنفًا قاعديًا فنستطيع أن نعود إلى الوراثة التي كنا نشرحها، ولننظر في أول صنف فرعي لنا فيما يلي.

17.6.2 الصنف InterestAccount

نستخدم الوراثة الآن لتوفير حساب يضيف نسبة ربوية -سنفترض أنها 3%- عند كل عملية إيداع، وستكون مطابقةً لصنف BankAccount القياسي عدا تابع الإيداع وبدء معدل النسبة، لذا نعيد كتابة تنفيذ هذه التوابع كما يلي:

```
class InterestAccount(BankAccount):
    def __init__(self, initialAmount, interest=0.03):
        super().__init__(initialAmount)
        self.interest = interest
    def deposit(self, amount):
        super().deposit(amount)
        self.balance = self.balance * (1 + self.interest)
```

الملاحظة الأولى: نمرر الصنف الرئيسي (أو الأب) معامِلًا في تعريف الصنف، ويكون هذا الصنف الأب في حالتنا هو `BankAccount`.

الملاحظة الثانية: نستدعي `super().__init__()` في بداية التابع `__init__()`، والدالة `super()` هي دالة خاصة وظيفتها معرفة الصنف الرئيسي، ويفيدنا ذلك عند وراثة أكثر من صنف رئيسي واحد فيما يسمى بالوراثة المتعددة، حيث نتجنب بعض المشاكل الغريبة التي قد تظهر إذا حاولنا استدعاء الصنف الرئيسي باسمه، لذلك يُفضل استخدام `super()`.

ونبدأ الصنف الموروث باستدعاء التابع `__init__` الخاص بالصنف الرئيسي، ولا نحتاج هنا إلا إلى بدء السمة `interest` التي قدمناها هنا، وبالمثل في استخدام `super()` في تابع الإيداع، إذ يستدعي التابع `deposit` الخاص بالصنف الأب فلا نحتاج إلا إلى إضافة المزايا الجديدة للصنف `InterestAccount`.

وهكذا نرى قوة البرمجة الكائنية التوجه وإمكانياتها، فبما أننا وضعنا `BankAccount` داخل الأقواس بعد اسم الصنف فقد صارت جميع التوابع موروثةً من `BankAccount`، ونلاحظ أن `deposit` تستدعي التابع `deposit` الخاص بالصنف الرئيسي بدلًا من نسخ الشيفرة، وسيحصل الصنف الفرعي على تلك التعديلات تلقائيًا إذا عدّلنا `deposit` الخاص بالصنف `BankAccount` بحيث يحوي تحققًا من بعض الأخطاء.

17.6.3 الصنف ChargingAccount

هذا الصنف مطابق للصنف `BankAccount` عدا أنه يطلب رسومًا افتراضية مقدارها 3$ لكل عملية سحب، وبالنسبة لـ `InterestAccount` فيمكن إنشاء صنف يرث من `BankAccount` ويعدّل التابعين `init` و `withdraw`:

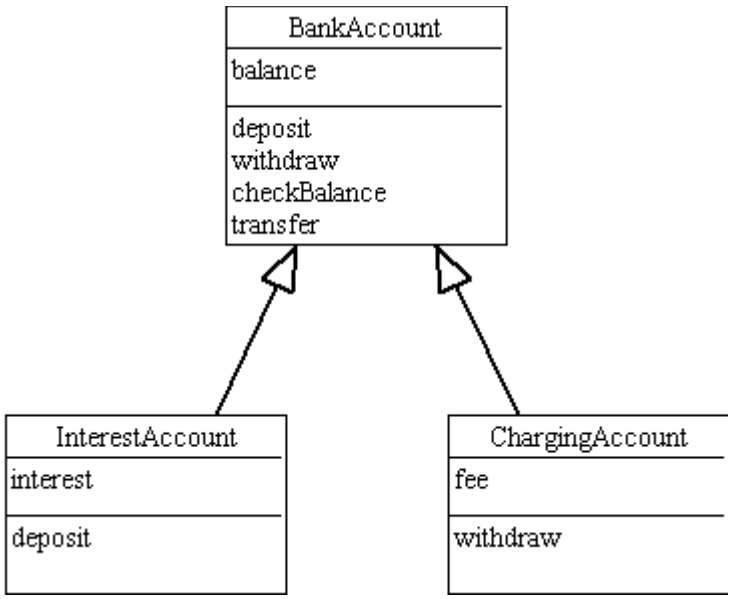
```
class ChargingAccount(BankAccount):
    def __init__(self, initialAmount, fee=3):
        super().__init__(initialAmount)
        self.fee = fee

    def withdraw(self, amount):
        super().withdraw(amount+self.fee)
```

الملاحظة الأولى: نخزن الرسوم مثل متغير نسخة intance variable لنستطيع تغييره لاحقًا عند الحاجة، ونلاحظ أننا نستدعي `__init__` مرةً أخرى مثل أي تابع آخر.

الملاحظة الثانية: نضيف الرسوم إلى عملية السحب المطلوبة في استدعاء التابع الموروث `withdraw` الذي ينجز العمل الفعلي.

الملاحظة الثالثة: نضيف أثرًا جانبيًا هنا حيث تُفرض رسوم تلقائيًا على عمليات التحويل، لكن هذا مطلوب على الأرجح لذا لا بأس به، وتجدر الإشارة هنا إلى أن إعادة الاستخدام هذه تحمل في طياتها احتمالية الآثار الجانبية غير المتوقعة التي يجب الحذر منها.

ونمثل هذه الوراثة في UML بسهم مصمت من الصنف الفرعي إلى الصنف الرئيسي، فتُمثَّل هرمية الحساب البنكي الآن كما يلي:



نلاحظ أننا سردنا التوابع والسمات التي تغيرت فقط أو أضيفت إلى الأصناف الفرعية.

17.6.4 اختبار النظام

للتحقق من عمل الهرمية السابقة بكفاءة، جـرب تنفيـذ الشـيفرة التاليـة في محث بـايثون أو بإنشـاء ملـف اختبار منفصل:

```
from bankaccount import *
# الحساب البنكي العادي
a = BankAccount(500)
b = BankAccount(200)
a.withdraw(100)
# a.withdraw(1000)
a.transfer(100,b)
print( "A = ", a.checkBalance() )
print( "B = ", b.checkBalance() )

# حساب للنسبة الربوية
c = InterestAccount(1000)
c.deposit(100)
print( "C = ", c.checkBalance() )

# حساب للرسوم المفروضة
d = ChargingAccount(300)
d.deposit(200)
print( "D = ", d.checkBalance() )
d.withdraw(50)
print( "D = ", d.checkBalance() )
d.transfer(100,a)
print( "A = ", a.checkBalance() )
print( "D = ", d.checkBalance() )

# حوّل من حساب الرسوم إلى حساب النسبة الربوية
# حساب الرسوم سيطلب رسومًا، وحساب النسبة الربوية
# يضيف نسبة ربوية
print( "C = ", c.checkBalance() )
print( "D = ", d.checkBalance() )
d.transfer(20,c)
print( "C = ", c.checkBalance() )
```

```
print( "D = ", d.checkBalance() )
```

أزل علامة التعليق الآن من السطر `a.withdraw(1000)` لترى الاعتراض عمليًا.

وبهذا نكون أتممنا مثالًا بسيطًا يظهر كيفية استخدام الوراثة لتوسيع إطار بسيط وإضافة مزايا قوية إليه، وقد رأينا كيف يبنى المثال على مراحل ويوضع برنامج اختبار للتحقق من نجاحه، مع أن اختباراتنا لم تكن كاملةً لأننا لم نغطِّ كل الحالات الممكنة، وكان من الممكن إدراج المزيد من الاختبارات، كما في حالة إنشاء حساب برصيد سالب.

ا. التطوير الموجه بالاختبارات

يستخدم العديد من المبرمجين العاملين تقنيةً تسمى التطوير الموجَّه بالاختبار test driven development، حيث يكتبون اختباراتهم قبل كتابة الشيفرة البرمجية نفسها، مما يسمح لهم باختبار شيفرتهم مرةً بعد مرة أثناء تطويرها، وينتقلون تدريجيًا من فشل الاختبارات المتكرر إلى نجاحها، وإذا نجحت جميع الاختبارات فإن هذا يعني أن البرنامج يعمل بكفاءة.

وهذا النهج شائع جدًا في البرمجة لدرجة تطوير أدوات خاصة لتساعد فيه، ولدى بايثون أدوات مثل هذه موجودة في وحدة `unittest` في المكتبة القياسية، وهذا النهج -وإن كان مفيدًا في كتابة شيفرة حقيقية- لن يفيدنا في شرحنا كثيرًا لأنه يخفي الشيفرة الأساسية وحالات اختبار كثيرة نحاول دراستها، لذا لن نستخدمه هنا، لكن قد ننظر فيه في فصل لاحق.

17.7 تجميعات الكائنات

إحدى المشاكل التي قد تواجهها هي كيفية التعامل مع كائنات كثيرة، أو كيفية التعامل مع كائنات تنشئها في وقت التشغيل، فمن السهل إنشاء حسابات بنكية ثابتة كما فعلنا أعلاه:

```
acc1 = BankAccount(...)
acc2 = BankAccount(...)
acc3 = BankAccount(...)
etc...
```

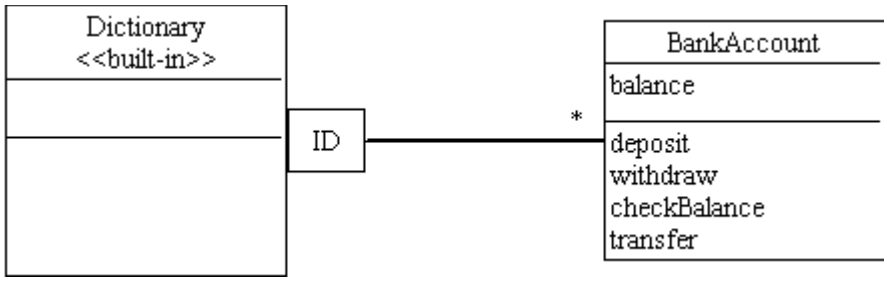
لكن في العالم الحقيقي لا تكون لدينا بيانات عن عدد الحسابات التي يجب إنشاؤها، فكيف نحل هذه المشكلة؟ لننظر فيها بتفصيل أكبر:

نريد شكلًا ما من قواعد البيانات التي تسمح لنا بإيجاد أي حساب بنكي باسم مالكه أو رقم حسابه -بما أنه قد يكون للشخص الواحد عدة حسابات-، والعكس صحيح، ولكن ألا يشبه البحث عن شيء له معرّف خاص به القاموس؟ لنجرب استخدام قاموس في بايثون للاحتفاظ بكائنات منشأة ديناميكيًا:

```
from bankaccount import BankAccount
import time

# أنشئ دالة جديدة لتوليد أرقام معرّفات فريدة
def getNextID():
    ok = input("Create account? ")
    if ok[0] in 'yY':  # check valid input
       id = time.time() # use current time as basis of ID
       id = int(id) % 10000 # حول إلى عدد صحيح وقلله إلى 4 أرقام
    else: id = -1  # وذلك سيوقف الحلقة التكرارية
    return id

# أنشئ بعض الحسابات وخزنها في القاموس
accountData = {}  # قاموس جديد
while True:           # كرر حلقيًا بلا نهاية
   id = getNextID()
   if id == -1:
      break       # تخرج إجباريًا من الحلقة التكرارية
   bal = float(input("Opening Balance? "))  #  حول السلسلة إلى عدد ذي فاصلة عائمة
   accountData[id] = BankAccount(bal) #  استخدم المعرِّف لإنشاء إدخال جديد في القاموس
   print( "New account created, Number: %04d, Balance %0.2f" % (id, bal) )

# دعنا نصل الآن إلى بعض الحسابات
for id in accountData.keys():
    print( "%04d\t%0.2f" % (id, accountData[id].checkBalance()) )

# ونبحث عن واحد فيها
# أدخل محرفًا غير رقمي لرفع اعتراض وإنهاء البرنامج
while True:
   id = int(input("Which account number? "))
   if id in accountData:
      print( "Balance = %0.2d" % accountData[id].checkBalance() )
   else: print( "Invalid ID" )
```

لا شك أن المفتاح الذي نستخدمه للقاموس قد يكون أي شيء يعرّف الكائن تعريفًا فريدًا، فقد يكون أحد سماته -مثل الرصيد `balance`- لكن الرصيد قد يتشابه بين الحسابات، ويتغير بالزيادة والنقص، وهكذا نفكر في الخيارات المتاحة إلى أن نصل إلى معرّف لا يتكرر، وهنا يمكن الرجوع إلى الفصل الخامس: البيانات وأنواعها، لقراءة القسم الخاص بالقاموس مرةً أخرى.

يمثَّل هذا مرئيًا في UML باستخدام مخطط الصنف، ويُعرض القاموس مثل صنف له علاقة مع العديد من الحسابات البنكية، ونرى ذلك موضحًا بمحرف النجمة على الخط الموصل بين الأصناف، ونستخدم محرف النجمة هنا لأنه الرمز المستخدم في التعابير النمطية للإشارة إلى عدد من العناصر مقداره صفر أو أكثر، وهذا يُعرف بعدد عناصر العلاقة cardinality of the relationship، ويمكن رؤيته بعد طرق، لكن المجالات العددية للتعابير النمطية هي المستخدمة بكثرة لثرائها ومرونتها.



نلاحظ استخدام القالب النمطي stereotype على القاموس `Dictionary` لإظهار أنه صنف مضمَّن، كما نلاحظ وجود الصندوق الملحق بالارتباط، والذي يوضح أن المفتاح هو قيمة المعرّف `ID`، فإذا كنا نستخدم قائمةً بسيطةً فلن يكون لدينا الصندوق ولكان الخط وصل بين الصنفين مباشرةً، وبهذا يتضح أننا نتجنب الحاجة إلى مخططات الكائنات الكبيرة والمعقدة باستخدام علاقات الأصناف وعدد العناصر في المجموعة cardinality، حيث نركز على العلاقات المجردة بين الأصناف بدلًا من التعامل مع عدد كبير من العلاقات الحقيقية بين النسخ المفردة.

17.8 حفظ الكائنات

إن فقد البيانات عند انتهاء المخطط هو أحد مشاكل السلوك السابق، لذا نريد طريقةً لحفظ الكائنات، ويمكن فعل ذلك باستخدام قواعد البيانات لكنه أسلوب متقدم، أما الآن فسنستخدم ملفًا نصيًا بسيطًا لحفظ الكائنات واسترجاعها، ورغم أن بايثون تحتوي على وحدتين لتنفيذ ذلك بكفاءة، وهما `pickle` و`shelve`، إلا أننا سنشرح الطريقة العامة التي تصلح لأي لغة، والطريقة العامة التي نقصدها هي إنشاء التابعين `save` و`restore` في الكائن ذي المستوى الأعلى، ثم إعادة كتابتهما في كل صنف ليستدعيا النسخة الموروثة ثم يضيفا السمات المعرفة محليًا، وبالمناسبة يُستخدم مصطلح الثبات Persistence للإشارة إلى القدرة على حفظ الأشياء واستعادتها.

```
class A:
   def __init__(self,x,y):
     self.x = x
     self.y = y
     self.f = None

   def save(self,fn):
     f = open(fn,"w")
     f.write(str(self.x)+ '\n') # حول إلى سلسلة نصية وأضف سطرًا جديدًا
     f.write(str(self.y)+'\n')
     return f              # من أجل أن يستخدمه الكائنات الأبناء

   def restore(self, fn):
     f = open(fn)
     self.x = int(f.readline()) # إعادته إلى النوع الأصلي
     self.y = int(f.readline())
     return f

class B(A):
   def __init__(self,x,y,z):
     super().__init__(x,y)
     self.z = z

   def save(self,fn):
     f = super().save(fn)  # استدعاء save من الأب
     f.write(str(self.z)+'\n')
     return f          # إن وجدات كائنات أبناء

   def restore(self, fn):
     f = super().restore(fn)
     self.z = int(f.readline())
     return f

# أنشئ النُسخ
a = A(1,2)
b = B(3,4,5)
```

```
# احفظ النُسخ
a.save('a.txt').close() # تذكر أن تغلق الملف
b.save('b.txt').close()

# اجلب النسخ
newA = A(5,6)
newA.restore('a.txt').close() # تذكر أن تغلق الملف
newB = B(7,8,9)
newB.restore('b.txt').close()
print( "A: ",newA.x,newA.y )
print( "B: ",newB.x,newB.y,newB.z )
```

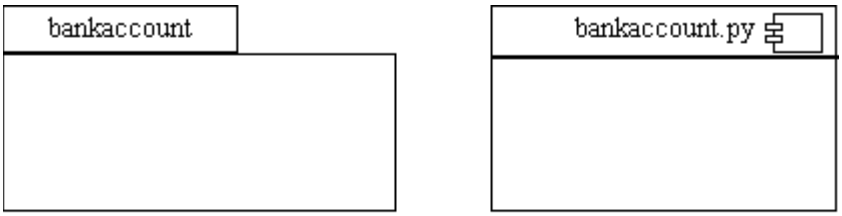
نلاحظ أن القيم المطبوعة هي القيم المسترجعة، وليست القيم التي استخدمناها لإنشاء النسخ.

والمهم هنا هو إعادة كتابة التابع save والتابع restore في كل صنف واستدعاء التابع الرئيسي في خطوة أولى، ثم لا نتعامل في الصنف الفرعي إلا مع سمات ذلك الفرعي فقط، ومن البديهي أن تحويل السمة إلى سلسلة نصية وحفظها أمر متروك لك كونك مبرمجًا، لكن يجب إخراجها على سطر واحد، أما عند الاستعادة فببساطة يمكن عكس عملية التخزين، لكن تظهر هنا عقبة كبيرة في هذا الأسلوب، وهي أننا نحتاج إلى إنشاء ملف منفصل لكل كائن، وهذا قد يعني آلاف الملفات الصغيرة إذا كنا في بيئة برمجية لسوق حقيقي، حيث سيتعقد العمل بوتيرة متسارعة، وسنحتاج إلى استخدام قاعدة بيانات لحفظ الكائنات، وسنبحث في ذلك في فصل لاحق، أما الآن فيكفي أن تعلم أن المفاهيم الأساسية ستظل كما هي.

17.9 دمج الأصناف والوحدات

توفر الأصناف والوحدات آليات للتحكم في تعقيد البرنامج، ومن المنطقي مع ازدياد حجم البرنامج ازدياد الحاجة إلى دمج المزايا بوضع أصناف داخل وحدات، وتنصح بعض الهيئات بوضع كل صنف في وحدة منفصلة، لكن هذا يؤدي إلى وحدات كثيرة جدًا ويزيد التعقيد بدلًا من تقليله، والبديل الذي لدينا هو تجميع الأصناف معًا، ووضع المجموعة في وحدة، وإذا عدنا إلى مثالنا أعلاه فسنضع كل تعريفات أصناف الحساب المصرفي في وحدة واحدة هي bankaccount مثلًا، ثم ننشئ وحدةً منفصلةً لشيفرة التطبيق التي تستخدم الوحدة.

يمكن تمثيل ذلك مرئيًا بواسطة UML بطريقتين، حيث يمكن تمثيل الجمع المنطقي للأصناف باستخدام حزمة، أو نستطيع تمثيل الملف الحقيقي مثل مكوّن:



A Package A Component

والهدف هنا أن تبدو أيقونة الحزمة مثل مجلد في أي برنامج مـدير ملفـات، أمـا الأيقونـة الصـغيرة الـتي في أعلى اليمين في أيقونة المكون فهي رمز المكون القديم في UML، وبما أن رسمها صـعب في المخططـات عنـد رسم الخطوط التي تظهر العلاقات بين المكونات فقد صغِّر شكلها في UML 2.0.

هذا ما سنغطيه حول UML في هذا الكتاب، ويمكن الرجوع لمحركات البحث والويب للاستزادة من المراجع والتدريبات وأدوات رسم UML، رغم أن الأشكال سهلة الرسم في أي برنامج رسم متجهي.

إذا أردنا تمثيلًا بسيطًا لذلك التصميم فسيكون كما يلي:

```
# File: bankaccount.py
#
# Implements a set of bank account classes
####################

class BankAccount: ....

class InterestAccount: ...

class ChargingAccount: ...
```

ثم إذا أردنا استخدامه:

```
import bankaccount

newAccount = bankaccount.BankAccount(50)
newChrgAcct = bankaccount.ChargingAccount(200)



# هنا يمكن تنفيذ المهام التي تريدها
```

لكن ماذا لو أراد صنفان في وحدتين الوصول إلى بيانات بعضهما بعضًا؟ إن أبسط حل لهذا هو استيراد كلتـا الوحدتين وإنشاء نُسخ للأصناف الـتي نريـدها، وتمريـر نُسـخ أحـد الصـنفين إلى توابـع النسـخة الأخـرى، وتمريـر

الكائنات كاملةً من مكان لآخر ما هو إلا برمجة كائنية التوجه، وهنا ندرك أحد أسباب التسمية لهذا النوع من البرمجة، فلا نحتاج إلى استخراج سمات كائن وتمريرها إلى كائن آخر، بل نمرر الكائن كله، وإذا كان الكائن يستخدم رسالةً متعددة الأشكال للوصول إلى المعلومات التي يحتاج إليها فسيصلح التابع مع أي نوع من الكائنات التي تدعم الرسالة.

لننظر في مثال واقعي لتوضيح ذلك، حيث ننشئ وحدةً قصيرةً نسميها `logger` تحتوي صنفين، هما `Logger` الذي يسجل النشاط داخل ملف، ويحتوي هذا الصنف على تابع واحد هو `log()` الذي يأخذ معاملًا كائنًا قابلًا للتسجيل، أما الصنف الآخر فهو `Loggable` الذي تستطيع الأصناف الأخرى أن ترثه لتعمل مع logger:

```
# File: logger.py
#
# Create Loggable and Logger classes for logging activities
# of objects
############

class Loggable:
   def activity(self):
       return "This needs to be overridden locally"

class Logger:
   def __init__(self, logfilename = "logger.dat"):
       self._log = open(logfilename,"a")

   def log(self, loggedObj):
       self._log.write(loggedObj.activity() + '\n')

   def __del__(self):
       self._log.close()
```

نلاحظ أننا وفرنا تابع تدمير destructor هو `__del__` لإغلاق الملف عند حذف كائن التسجيل أو كنسه garbage collected، وهو أحد التوابع "السحرية" الموجودة في بايثون كما نرى من الشرطتين السفليتين حوله، واللتين تشبهان `__init__()`، مع فرق أن `init` يُستدعى عند إنشاء نسخة ما، أما `del` فيستدعى عندما يحذف كانس المخلفات النسخة، وقد لا يُستدعى إذا خرجت بايثون خروجًا غير متوقع، حيث سيكون لدينا في هذه الحالة مشاكل أكبر من استدعاء `del` أو عدم استدعائه.

كما استدعينا سمة السجل _log مع شرطة سفلية قبلها، وهو اصطلاح للتسمية في بايثون، كما في استخدام الكلمات ذات الأحرف الكبيرة لأسماء الأصناف، فالشرطة السفلية المفردة تعني أن السمة ليست مصممةً لنصل إليها مباشرةً، بل من خلال توابع الصنف.

سننشئ الآن وحدةً جديدةً تعرِّف النسخ القابلة للتسجيل لأصناف حساباتنا المصرفية السابقة، لنستطيع استخدام وحدتنا:

```
# File: loggablebankaccount.py
#
# Extend Bank account classes to work with logger module.
###############################

import bankaccount, logger

class LoggableBankAccount(bankaccount.BankAccount, logger.Loggable):
    def activity(self):
        return "Account balance = %d" % self.checkBalance()

class LoggableInterestAccount(bankaccount.InterestAccount,
                              logger.Loggable):
    def activity(self):
        return "Account balance = %d" % self.checkBalance()

class LoggableChargingAccount(bankaccount.ChargingAccount,
                              logger.Loggable):
    def activity(self):
        return "Account balance = %d" % self.checkBalance()
```

استخدمنا الوراثة المتعددة في المثال أعلاه، حيث ورثنا صنفين رئيسيين وليس صنفًا واحدًا، وهذا غير ضروري في بايثون بما أننا نستطيع إضافة التابع activity() إلى أصنافنا الأصلية ونحقق نفس الأثر، أما في لغات برمجة كائنية التوجه ثابتة مثل جافا أو C++ فسيكون هذا ضروريًا، لذا سنشرح التقنية هنا، قد تلاحظ أن التابع activity() متطابق في الأصناف الثلاثة، وهذا يعني أننا نستطيع توفير بعض الكتابة على أنفسنا بإنشاء نوع وسيط من صنف حساب قابل للتسجيل يرث Loggable وله تابع activity فقط، ثم ننشئ ثلاثة أنواع حسابات قابلة للتسجيل بوراثتها من ذلك الصنف الجديد ومن صنف Loggable، كما يلي:

```
class LoggableAccount(logger.Loggable):
     def activity(self):
```

```
            return "Account balance = %d" % self.checkBalance()

class LoggableBankAccount(bankaccount.BankAccount, LoggableAccount):
     pass

class LoggableInterestAccount(bankaccount.InterestAccount,
LoggableAccount):
     pass

class LoggableChargingAccount(bankaccount.ChargingAccount,
LoggableAccount):
     pass
```

لا يوفر هذا علينا الكثير من الكتابة، لكنه يعني أن علينا اختبار تعريف تابع واحد فقط والاحتفاظ بـه بـدلًا من ثلاثة توابع متطابقة، وهذا النوع من البرمجة الذي يكون فيـه لصـنف رئيسـي وظيفـةً مشـتركةً يسـمى بالبرمجـة الخليطة mixin programming، ويسمى الصنف الأدنى بالصنف الخليط mixin class، والناتج المتوقـع من ذلك الأسلوب أن يكون لتعريفات الصنف النهائي متن صغير أو ليس لها متن أصلًا، لكنهـا تحـوي قائمـةً طويلـةً من الأصناف الموروثة كما رأينا هنا.

ومن الشائع أيضًا أن الأصناف المختلطة لا ترث من أي شيء بنفسها، رغم أننا فعلنـا هـذا هنـا، فمـا هي إلا طريقة لإضافة تابع مشـترك أو مجموعـة من التوابـع إلى صـنف أو مجموعـة أصـناف باسـتخدام الوراثـة، ويـأتي مصطلح "المختلطة mixin" من مجال صناعة المثلجات، حيث تضاف نكهات مختلفة إلى الفانيليا لإنتاج نكهـة جديدة، وكانت أول لغة تدعم هذا الأسلوب هي لغة Flavors، والتي كانت إحدى لغات Lisp المشهورة وقتها.

نأتي الآن إلى النقطة التي نُظهـر فيهـا شـيفرة التطـبيق الخـاص بنـا بإنشـاء كـائن مسـجل logger وبعض الحسابات المصرفية، ثم تمرير الحسابات إلى المسجل، رغم أنها معرّفة في وحدات مختلفة:

```
# Test logging and loggable bank accounts.
#############

import logger
import loggablebankaccount as lba

log = logger.Logger()

ba = lba.LoggableBankAccount(100)
ba.deposit(700)
```

```
log.log(ba)

intacc = lba.LoggableInterestAccount(200)
intacc.deposit(500)
log.log(intacc)
```

نلاحظ هنا كلمة `as` المفتاحية التي تُستخدم لإنشاء اسم مختصر عند استدعاء `loggablebankaccount`.

لا نحتاج إلى استخدام سابقة الوحدة module prefix بعد إنشاء النسخ المحلية، وبما أنه لا يوجد وصول مباشر من كائن إلى آخر بل من خلال الرسائل، فلا حاجة أن تشير وحدتا تعريف الصنف إلى بعضهما البعض مباشرةً، كما يعمل المسجل مع نسخ كل من `LoggableBankAccount` و `LoggableInterestAccount` لأنهما يدعمان واجهة `Loggable`، فالتوافق بين واجهات الكائنات من خلال تعددية الأشكال هو الأساس الذي تبنى عليه جميع البرامج الكائنية التوجه.

يوجد نظام تسجيل أكثر تعقيدًا في وحدة المكتبة القياسية `logging`، لإظهار بعض التقنيات فقط، ونذكره هنا لتكون هذه المكتبة أول خيار نبحث فيه عند الحاجة إلى أدوات تسجيل في البرامج التي نكتبها.

نأمل أن يكون هذا الشرح كافيًا لفهم البرمجة كائنية التوجه، مع الاستزادة من المصادر الموجودة في الويب أو قراءة أحد الكتب المذكورة في بداية الفصل، أما الآن فسننظر في كيفية تنفيذ البرمجة كائنية التوجه في جافاسكربت و VBScript.

17.10 البرمجة الكائنية في VBScript

تدعم VBScript مفهوم الكائنات وتسمح بتعريف الأصناف وإنشاء نسخ منها، لكنها لا تدعم مفهوم الوراثة ولا تعددية الأشكال، وعلى هذا تكون لغة VBScript لغةً كائنية الأساس object based وليست كائنية التوجه، لكن مفهوم دمج البيانات والدوال في كائن واحد لا يزال صالحًا هنا، ويمكن تنفيذ صورة محدودة من الوراثة باستخدام تقنية تسمى التفويض delegation.

17.10.1 تعريف الأصناف

يُعرَّف الصنف في VBScript باستخدام تعليمة `Class` كما يلي:

```
<script type=text/VBScript>
Class MyClass
   Private anAttribute
   Public Sub aMethodWithNoReturnValue()
```

```
        MsgBox "MyClass.aMethodWithNoReturnValue"
    End Sub
    Public Function aMethodWithReturnValue()
        MsgBox "MyClass.aMethodWithReturnValue"
        aMethodWithReturnValue = 42
    End Function
End Class
</script>
```

وهذا يعرِّف صنفًا جديدًا اسمه `MyClass` مع سمة تسمى `anAttribute` تكون مرئيةً فقـط للتوابـع الـتي في الصنف، كما نرى من كلمة `Private` المفتاحية، ومن المتعارف عليه أن نصـرح عن سـمات البيانـات بأنهـا `Private`، وعن أغلب التوابع بأنها `Public`، ويُعرف هذا باسم إخفاء البيانات، وميزته أنه يسمح لنا بالتحكم في الوصول إلى البيانات بإجبار استخدام التوابـع الـتي تتحقـق من جـودة البيانـات على القيم الـتي تمـرَّر إلى داخـل وخارج الكائن، وتوفر بايثون آليةً خاصةً بها لتنفيذ هذا لكنها خارج نطاق شرحنا.

17.10.2 إنشاء النسخ

ننشئ النسخ في VBScript بدمج الكلمتين المفتاحيتين `Set` و`New`، ويجب أن يكون المتغير الذي أُسندت إليه النسخة الجديدة قد صرِّح عنه باستخدام كلمة `Dim` كما هو متبع في VBScript:

```
<script type=text/VBScript>
Dim anInstance
Set anInstance = New MyClass
</script>
```

يؤدي هذا إلى إنشاء نسخة من الصنف مصرح عنها في القسم السابق وإسنادها إلى متغير `anInstance`، لاحظ أننا يجب أن نسبق اسم المتغير بـ `Set`، وأننا نستخدم كلمة `New` لإنشاء الكائن.

17.10.3 إرسال الرسائل

تُرسَل الرسائل إلى النسخ باستخدام نفس الصيغة النقطية `.` التي تستخدمها بايثون:

```
<script type=text/VBScript>
Dim aValue
anInstance.aMethodWithNoReturnValue()
aValue = anInstance.aMethodWithReturnValue()
MsgBox "aValue = " & aValue
</script>
```

يُستدعى التابعان المصرح عنهما في تعريف الصنف، ولا توجد في الحالة الأولى قيمة معادة، أما في الحالة الثانية فسنسند القيمة المعادة إلى المتغير aValue، ولا يوجد شيء غير اعتيادي هنا باستثناء أن البرنامج الفرعي subroutine والدالة مسبوقان باسم النسخة.

17.10.4 الوراثة وتعددية الأشكال

لا تدعم VBScript أي آلية للوراثة أو تعددية الأشكال، لكن نستطيع محاكاة ذلك إلى حد ما باستخدام تقنية تسمى التفويض، وهذا يعني أننا نعرف سمةً للصنف الفرعي ليكون نسخةً من الصنف الرئيسي المفترض، ثم نعرف تابعًا لجميع التوابع الموروثة التي تستدعي تابع النسخة الرئيسية أو تفوض إليه، لننشئ صنفًا فرعيًا من MyClass كما هو معرَّف أعلاه:

```
<script type=text/VBScript>
Class SubClass
   Private parent
   Private Sub Class_Initialize()
      Set parent = New MyClass
   End Sub
   Public Sub aMethodWithNoReturnValue()
      parent.aMethodWithNoREturnVAlue
   End Sub
   Public Function aMethodWithReturnValue()
      aMethodWithReturnValue = parent.aMethodWithReturnValue
   End Function
   Public Sub aNewMethod
      MsgBox "This is unique to the sub class"
   End Sub
End Class

Dim inst,aValue
Set inst = New SubClass
inst.aMethodWithNoReturnVAlue
aValue = inst.aMethodWithReturnValue
inst.aNewMethod
MsgBox "aValue = " & CStr(aValue)
</script>
```

نلاحظ هنا استخدام السمة الخاصة parent والتابع الخاص المميز Class_Initialise، فالأولى هي سمة تفويض الصنف الأب، والثاني هو المكافئ للتابع __init__ في بايثون لبدء النسخ عند إنشائها، أي أنه الباني في لغة VBScript.

17.11 البرمجة الكائنية في جافاسكربت

تدعم جافاسكربت الكائنات باستخدام تقنية تسمى النمذجة الأولية prototyping، وهذا يعني عدم وجود بنية صنف صريحة في جافاسكربت، بل نستطيع تعريف الصنف بمجموعة من الدوال أو مثل مفهوم شبيه بالقاموس يُعرف بالمهيئ initializer.

17.11.1 تعريف الأصناف

من أكثر الطرق شيوعًا لتعريف الأصناف في جافاسكربت هي إنشاء دالة بنفس اسم الصنف، ويكون هو الباني، لكنه لا يُحتوى داخل أي بنية أخرى:

```
<script type=text/JavaScript>
function MyClass(theAttribute)
{
   this.anAttribute = theAttribute;
};
</script>
```

ربما تلاحظ كلمة this التي تُستخدم بنفس طريقة استخدام self في بايثون مثل مرجع نائب placeholder reference إلى النسخة الحالية، رغم أننا لا نحتاج في الغالب إلى إدراج this صراحةً في قائمة المعامِلات لتوابع الصنف، ونستطيع إضافة سمات جديدة إلى الصنف لاحقًا باستخدام السمة prototype المضمَّنة كما يلي:

```
<script type=text/JavaScript>
MyClass.prototype.newAttribute = null;
</script>
```

ويعرِّف هذا سمةً جديدةً لـ MyClass اسمها newAttribute، وتضاف التوابع من خلال تعريف دالة عادية؛ ثم إسناد اسم الدالة إلى سمة جديدة مع اسم التابع، ويكون للتابع والدالة نفس الاسم عادةً، لكن لا مانع من تغيير اسم التابع إلى شيء مختلف، كما يلي:

```
<script type=text/JavaScript>
function oneMethod(){
```

```
    return this.anAttribute;
}
MyClass.prototype.getAttribute = oneMethod;
function printIt(){
    document.write(this.anAttribute + "<BR>");
};
MyClass.prototype.printIt = printIt;
</script>
```

ولا شك أن الأسهل تعريف الدوال ثم الباني، ثم إسناد التوابع داخل الباني، وهذا هـو الأسـلوب الافتراضـي، لذا سيبدو تعريف الصنف كاملًا كما يلي:

```
<script type=text/JavaScript>
function oneMethod(){
    return this.anAttribute;
};

function printIt(){
    document.write(this.anAttribute + "<BR>");
};

function MyClass(theAttribute)
{
   this.anAttribute = theAttribute;
   this.getAttribute = oneMethod;
   this.printIt = printIt;
};
</script>
```

لكن ثمة طريقة أخرى في جافاسكربت، باسـتخدام صـيغة مختلفـة قليلًا لإنشـاء دوال التـابع، حيث تسـمح جافاسكربت بتعريف الدالة كما يلي:

```
square = function(x){ return x*x;}
```

ثم نستدعي ذلك كما يلي:

```
document.write("The square of 5 is: " + square(5))
```

فإذا طبقناه على تعريف الصنف الخاص بنا نحصل على:

```
<script type=text/JavaScript>
function MyClass(theAttribute)
{
   this.anAttribute = theAttribute;
   this.getAttribute = function(){
                       return this.anAttribute;
                       };
   this.printIt = function printIt(){
                  document.write(this.anAttribute + "<BR>");
                  };
};
</script>
```

يفضل بعض المبرمجين هذا الأسلوب لأنه يبقي تعريفات التابع داخل الصنف دون تلويث فضاء الاسم الخارجي، بينما يرى آخرون أن هذا أسلوب فوضوي وأصعب في القراءة، وأنت حر في اختيار أي الأسلوبين تفضل.

17.11.2 إنشاء النسخ

تُنشأ نسخ الأصناف باستخدام كلمة new المفتاحية كما يلي:

```
<script type=text/JavaScript>
var anInstance = new MyClass(42);
</script>
```

مما ينشئ نسخةً جديدةً اسمها anInstance.

17.11.3 إرسال الرسائل

لا يختلف إرسال الرسائل في جافاسكربت عن اللغات الأخرى، إذ نستخدم الصيغة النقطية:

```
<script type=text/JavaScript>
document.write("The attribute of anInstance is: <BR>");
anInstance.printIt();
</script>
```

17.11.4 الوراثة وتعددية الأشكال

يمكن استخدام آلية النمذجة الأولية في جافاسكربت للوراثة من صنف آخر -على عكس VBScript-، مع أنها أعقد من تقنية بايثون لكن يمكن التعامل معها وضبطها، لكنها ليست منتشرةً بين مبرمجي جافاسكربت، إن أساس الوراثة في جافاسكربت هو الكلمة المفتاحية `prototype` التي استخدمناها في الشيفرة أعلاه، حيث يمكن إضافة مزايا إلى كائن ما بعد تعريفه، كما يلي:

```
<script type="text/javascript">
function Message(text){
   this.text = text;
   this.say = function(){
        document.write(this.text + '<br>');
        };
};

msg1 = new Message('This is the first');
msg1.say();

Message.prototype.shout = function(){
    alert(this.text);
    };

msg2 = new Message('This gets the new feature');
msg2.shout();

/* msg1 وبالمثل بالنسبة لـ ...*/
msg1.shout();
</script>
```

الملاحظة الأولى: أضفنا تابع `alert` جديد باستخدام `prototype` بعد إنشاء نسخة `msg1` للصنف، غير أن الخاصية كانت متاحةً للنسخة الحالية ونسخة `msg2` التي أنشئت بعد الإضافة، أي أن الخاصية الجديدة تضاف إلى جميع نسخ `Message` الحالية والجديدة، وتؤدي خاصية النمذجة الأولية هذه إلى إمكانية تغيير سلوك كائنات جافاسكربت المضمَّنة، بإضافة مزايا جديدة أو تغيير الطريقة التي تتصرف بها المزايا الحالية، لذا استخدمها بحرص إذا لم ترد أن تضيع وقتك مع علل برمجية يصعب تعقبها. ومع ذلك لاستخدام `prototype` آليةً لإضافة وظائف إلى الأصناف الحالية عيوبه، التي منها تغيير سلوك النسخة الحالية، وتغيير تعريف الصنف الأصلي. ويوجد أسلوب تقليدي أكثر من الوراثة يمكن استخدامه، كما يلي:

```
<script type="text/javascript">
function Parent(){
   this.name = 'Parent';
   this.basemethod = function(){
       alert('This is the parent');
       };
};

function Child(){
   this.parent = Parent;
   this.parent();
   this.submethod = function(){
       alert('This from the child');
       };
};

var aParent = new Parent();
var aChild = new Child();

aParent.basemethod();
aChild.submethod();
aChild.basemethod();
</script>
```

يجب أن نلاحظ أن كائن `child` هنا لـه وصـول إلى `basemethod`، دون أن يُعطى ذلـك الوصـول صـراحةً، وإنما يرثه من الصنف الرئيسي بحكم أسطر الإسناد/الاستدعاء داخل تعريف الصنف `Child`:

```
   this.parent = Parent;
   this.parent();
```

وبهذا نكون ورثنا `basemethod` من الصنف الرئيسي `Parent`.

17.11.5 الخلاف حول جافاسكربت

لجافاسكربت سمعة سيئة وسط المبرمجين لأسباب كثيرة، أحدها إمكانية إنشاء كائنات دون استخدام الكلمة المفتاحية new، ودون إرسال أي خطأ، لذا ستكون النتيجة غير التي نريدها، وسنواجه علل برمجيةً يصعب إيجادها، أحد أسبابها أن جافاسكربت ليس لديها مفهوم الصنف نفسه، وإنما تحاكيه محاكاةً فقط من خلال آلية النموذج الأولي، وليتغلب مبرمجو جافاسكربت على هذه المشكلة طوروا بعض المصطلحات التي تنتج شيفرات أكثر موثوقيةً، لكن تفسيرها يتطلب بعض المفاهيم المتقدمة من علوم الحاسوب التي يقصر عنها شرحنا، فإذا أردت استخدام جافاسكربت للبرمجة الكائنية التوجه ننصحك بقراءة الكتاب القصير Javascript the Good Parts الذي كتبه Douglas Crockford، والذي يشرح السبب الذي يجعل التقنيات المشروحة أعلاه غير مثالية، وكيفية تنفيذها بشكل أفضل، وننصحك عمومًا بقراءة كتاب دليل JavaScript الشامل من أكاديمية حسوب فهو يشرح لغة جافاسكربت بالكامل.

يمكن استخدام نفس حيلة التفويض التي استخدمناها في VBScript، كما يلي:

```
<script type=text/JavaScript>
function noReturn(){
   this.parent.printIt();
};

function returnValue(){
      return this.parent.getAttribute();
};

function newMethod(){
      document.write("This is unique to the sub class<BR>");
};

function SubClass(){
   this.parent = new MyClass(27);
   this.aMethodWithNoReturnValue = noReturn;
   this.aMethodWithReturnValue = returnValue;
   this.aNewMethod = newMethod;
};

var inst, aValue;
inst = new SubClass(); // عرِّف الصنف الرئيسي
document.write("The sub class value is:<BR>");
```

```
inst.aMethodWithNoReturnValue();
aValue = inst.aMethodWithReturnValue();
inst.aNewMethod();
document.write("aValue = " + aValue);
</script>
```

سنرى استخدام الكائنات والأصناف في دراسات الحالات والفصول التالية، ورغم أنه من الصعب أن يرى المبرمج المبتدئ كيف أن هذه البنية المعقدة تسهل كتابة وفهم البرامج، إلا أننا نأمل أن يتضح هذا المفهوم إذا رأيت استخدام الأصناف في البرامج الحقيقية، على أن ذلك ليس له فائدة كبيرة في البرامج الصغيرة، وإنما سيجعلها أعقد وأطول، لكن كلما زاد حجم البرنامج -أكثر من 100 سطر مثلًا-، فستعين الأصناف والكائنات على تنظيمه وتقليل كمية الشيفرات المكتوبة، ومن المهم أن تعلم أنه من الممكن تعلم البرمجة وكتابة برامج دون الحاجة إلى مفهوم البرمجة الكائنية التوجه أصلًا، فكم من مبرمج كتب برامج دون إنشاء صنف واحد طيلة حياته، لكن إذا استطعت استيعابها فإن فيها مزايا وتقنيات قويةً ومفيدةً.

17.12 خاتمة

نأمل في نهاية الفصل أن تكون تعلمت ما يلي:

- تغلِّف الأصناف البيانات والدوال في كيان واحد.
- تشبه الأصناف قاطعات البسكويت، حيث تُستخدم لإنشاء النسخ أو الكائنات.
- تتواصل الكائنات من خلال إرسال رسائل إلى بعضها البعض.
- عندما يستقبل كائن ما رسالةً فإنه ينفذ تابعًا موافقًا لها.
- التوابع هي دوال تُخزَّن مثل سمات للصنف.
- تستطيع الأصناف أن ترث التوابع والبيانات من أصناف أخرى، مما يسهل توسيع إمكانيات الصنف دون تغيير الصنف الأصلي.
- تعددية الأشكال هي القدرة على إرسال نفس الرسالة إلى عدة أنواع مختلفة من الكائنات، وسيستجيب كل منها بطريقته الخاصة.
- التغليف وتعددية الأشكال والوراثة كلها خصائص للغات البرمجة كائنية التوجه.
- تسمى لغة VBScript لغةً كائنية الأساس، لأنها لا تدعم الوراثة وتعددية الأشكال دعمًا كاملًا، رغم دعمها للتغليف.

دورة إدارة تطوير المنتجات

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

18. مفهوم البرمجة المدفوعة بالأحداث

درسنا حتى الآن البرامج جزئية التوجه batch oriented programs، والتي تستدعى بإحدى طريقتين: جزئية التوجه batch oriented، حيث تبدأ البرامج بالعمل ثم تنفذ شيئًا ما ثم تتوقف، أو مدفوعة بالأحداث أو حدثية التوجه event driven، أي تبدأ البرامج وتنتظر وقوع أحداث بعينها؛ ولا تتوقف إلا حين يأمرها حدث آخر بالتوقف.

كيف ننشئ برنامجًا مقودًا بالأحداث؟ سنفعل ذلك بطريقتين، حيث سنحاكي أولًا بيئةً حدثيةً، ثم سننشئ برنامجًا رسوميًا بسيطًا يستخدم نظام التشغيل والبيئة لتوليد أحداث.

وسنغطي في هذا الفصل النقاط التالية:

- أوجه اختلاف البرامج المدفوعة بالأحداث عن البرامج جزئية التوجه.
- كيفية كتابة حلقة حدث تكرارية.
- كيفية استخدام إطار عمل أحداث مثل Tkinter.

18.1 محاكاة حلقة أحداث تكرارية

يحتوي البرنامج المدفوع بالأحداث أو البرنامج الحدثي event driven program على حلقة تلتقط الأحداث المستَلمة وتعالجها، وقد تولد بيئة التشغيل الأحداث -كما يحدث في جميع البرامج الرسومية تقريبًا-، أو يبحث البرنامج عن الأحداث -كما يحدث في أنظمة التحكم المدمجة التي في الكاميرات وغيرها-، وسنكتب برنامجًا يبحث عن نوع واحد من الأحداث، وهو مدخلات لوحة المفاتيح، ويعالج النتائج إلى أن يستلم حدث خروج quit، والذي سيكون حالتنا مفتاح المسافة space على لوحة المفاتيح، وسنعالج الأحداث المدخلة بطريقة سهلة، إذ سنطبع ترميز آسكي ASCII الخاص بالمفتاح الذي ضغطه المستخدم، وسنستخدم بايثون لأنها تحوي دالة

getch() سهلة الاستخدام، وسنقرأ المفاتيح مفتاحًا تلو الآخر، وتأتي هذه الدالة في صورتين وفقًا لنظام التشغيل الذي نستخدمه، فنجدها في لينكس في وحدة curses، أما في ويندوز فستكون في وحدة msvcrt، وسنستخدم نسخة ويندوز أولًا ثم نناقش خيار لينكس بالتفصيل، ويجب أن نشغل هذه البرامج من سطر أوامر النظام، لأن بيئات التطوير -مثل IDLE- ستلتقط ضربات لوحة المفاتيح التقاطًا مختلفًا.

وسنطبق دوال معالجة الأحداث أولًا، والتي تُستدعى عند التقاط ضغطة أحد المفاتيح، ثم متن البرنامج الرئيسي الذي يبدأ حلقة جمع الأحداث؛ ويستدعي دالة معالجة الحدث المناسبة عند التقاط حدث صالح.

18.1.1 تطبيق المثال في ويندوز

إليك التطبيق التالي:

```
import msvcrt
import sys

# معالجات الأحداث أولًا
def doKeyEvent(key):
    if key == '\x00' or key == '\xe0':
       key = msvcrt.getch()
    print ( ord(key), ' ', end='')
    sys.stdout.flush() # make sure it appears on screen

def doQuit(key):
    print() # أدخل سطرًا جديدًا
    raise SystemExit

# أخل مساحة على الشاشة أولًا
lines = 25
for n in range(lines): print()

# والآن حلقة الحدث الأساسية
while True:
    ky = msvcrt.getch()
    if len(str(ky)) != 0:
        # لدينا حدث حقيقي
        if " " in str(ky):
            doQuit(ky)
```

```
        else:
            doKeyEvent(ky)
```

لاحظ أن ما نفعله مع الأحداث لا علاقة له بالمتن الرئيسي، الذي يجمع الأحداث ويمررها إلى معالجات الأحداث وحسب، وهذه الاستقلالية في التقاط الأحداث معالجتها هي الميزة الأساسية في البرمجة الحدثية.

كما نلاحظ أن `getch()` تعيد بايتات، لذا نحتاج إلى تحويلها إلى سلسلة نصية، وإلى استخدام اختبار `in` للتحقق متى يمكن الخروج بما أن السلسلة الناتجة لن تكون حرفًا.

وفي الحالة التي لا يكون فيها المفتاح محرف آسكي، كأن يكون مفتاحًا وظيفيًا مثلًا، وهي المفاتيح التي تحمل حرف F في أولها أعلى لوحة المفاتيح، فسنحتاج إلى جلب محرف ثانٍ من لوحة المفاتيح، لأن هذه المفاتيح الخاصة تولد أزواجًا من البايتات، أما `getch()` فلا تجلب إلى بايتًا واحدًا في كل مرة، وتكون القيمة المهمة التي نريدها هي البايت الثاني فعليًا.

18.1.2 تطبيق المثال في لينكس وماك

لا يستطيع المبرمجون الذين يستخدمون أنظمة تشغيل لينكس وماك استخدام مكتبة `msvcrt`، لذا يستخدمون وحدةً أخرى تسمى `curses`، وتكون الشيفرة الناتجة شبيهةً بشيفرة ويندوز، مع بعض التعديلات التي يجب إجراؤها، كما يلي:

```
import curses as c

def doKeyEvent(key):
    if key == '\x00' or key == '\xe0': # ASCII ليس من محارف
       key = screen.getch()      # اجلب المحرف الثاني
    screen.addstr(str(key)+' ') # استخدام متغير عام

def doQuitEvent(key):
    c.resetty() # اضبط إعدادات الطرفية
    c.endwin()  # أوقف جلسة المؤشر
    raise SystemExit

# امسح الشاشة واحفظ الإعدادات الحالية
# وأوقف الطباعة الآلية للمحارف على الشاشة
# ثم أخبر المستخدم ما يفعله للخروج
screen = c.initscr()
c.savetty()
```

```
c.noecho()
screen.addstr("Hit space to end...\n")

# والآن تعمل الحلقة الأساسية بلا نهاية
while True:
      ky = screen.getch()
      if ky != -1:
        # أطلق حدثًا لدالة معالجة الأحداث
        if ky == ord(" "): # التحقق من إنهاء الحدث
          doQuitEvent(ky)
        else:
          doKeyEvent(ky)

c.endwin()
```

لا تعمل أوامر الطباعة العادية في وحدة `curses`، بل يجب أن نستخدم دوال معالجة الشاشة الخاصة بوحدة `curses`، كما تعيد `getch` هنا `-1` إذا لم يُضغط على أي مفتاح -بدلًا من سلسلة فارغة-، ويطابق منطق البرنامج نسخة ويندوز السابقة فيما عدا ذلك.

ويجب أن تستعيد `curses.endwin()` شاشتنا إلى الحالة العادية، لكنها قد لا تعمل أحيانًا، فإذا اختفى المؤشر أو لم نحصل على محرف إرجاع لبداية السطر أو غير ذلك، فسنصلح المشكلة بالخروج من بايثون باستخدام Ctrl+D واستخدام الأمر التالي:

```
$ stty echo -nl
```

تشير `nl` إلى "سطر جديد"، وينبغي أن يصلح هذا السطر المشكلة أعلاه حال حدوثها.

إذا كنا ننشئ هذا مثل إطار عمل لاستخدامه في مشاريع عدة، فسنضيف استدعاءً إلى دالة تهيئة initialization function في البداية ودالة تنظيف في النهاية، ثم يستطيع المبرمج عندئذ استخدام الجزء الخاص بالحلقة وتوفير دواله الملائمة للتهيئة والمعالجة والتنظيف، وهذا ما تفعله البيئات الرسومية تحديدًا، حيث يكون الجزء الخاص بالحلقة مضمَّنًا في بيئة التشغيل أو إطار العميل، ويكون على البرامج أن توفر دوال معالجة الأحداث الخاصة بها وتربطها بحلقة الأحداث بشكل ما، لنر ذلك عمليًا أثناء دراسة مكتبة Tkinter الرسومية.

18.2 برنامج رسومي

سنستخدم صندوق أدوات Tkinter من بايثون في هذه التدريب، وهو مغلف بايثون لصندوق أدوات Tk، والذي كُتب في البداية امتدادًا للغة Tcl، كما أنه متاح للغتي Perl وروبي أيضًا، ونسخة بايثون منه هي إطار عمل كائني التوجه، العمل فيه أسهل من نسخة Tk الأصلية، وسننظر بتفصيل أكثر في مبادئ برمجة الواجهات الرسومية فيما بعد في الفصل التالي: برمجة الواجهات الرسومية، لذا لن نسهب كثيرًا في الحديث عن مبادئ الواجهات الرسومية في هذا الفصل، لأننا نريد التركيز على نمط البرمجة نفسه، وهو استخدام Tkinter لمعالجة حلقة الأحداث، ونترك المبرمج ينشئ الواجهة الرسومية الابتدائية ثم يعالج الأحداث حال وصولها.

وفي مثالنا هنا ننشئ صنف تطبيق application class اسمه `KeysApp` ينشئ الواجهة الرسومية في التابع `__init__`، ويربط مفتاح المسافة بالتابع `doQuitEvent`، كما يعرّف الصنف تابع `doQuitEvent` المطلوب، أما الواجهة الرسومية نفسها فتتكون ببساطة من ودجِت widget (تطبيق مُصغَّر) لإدخال النصوص؛ سلوكها الافتراضي هو طباعة المحارف المدخَلة على الشاشة.

إن إنشاء صنف تطبيق في البيئات الحدثية كائنية التوجه أمر شائع، بسبب الارتباط بين مفاهيم الأحداث المرسَلة إلى برنامج ما والرسائل المرسَلة إلى كائن، إذ يرتبطان ببعضهما بسهولة كبيرة، وبذلك تصبح دالة معالجة الحدث تابعًا لصنف التطبيق، وبعد أن عرّفنا الصنف سننشئ نسخةً منه ونرسل إليها رسالة `mainloop`، وستبدو الشيفرة كما يلي:

```
# استخدم from X import * للحفظ، بما أن علينا تقديم كل شيء
# tkinter.xxx كـ
from tkinter import *
import sys


# أنشئ صنف التطبيق الذي يعرف الواجهة الرسومية وتوابع
# معالجة الأحداث
class KeysApp(Frame):
    def __init__(self): # use constructor to build GUI
        super().__init__()
        self.txtBox = Text(self)
        self.txtBox.bind("<space>", self.doQuitEvent)
        self.txtBox.pack()
        self.pack()

    def doQuitEvent(self,event):
```

```
        sys.exit()

# والآن أنشئ نسخة وابدأ تشغيل حلقة الحدث
myApp = KeysApp()
myApp.mainloop()
```

نلاحظ أن البرنامج لن يغلق إغلاقًا صحيحًا عند تشغيل هذه الشيفرة من داخل IDLE، بل سيطبع رسالة إغلاق في نافذة الصدفة، وهو أسلوب IDLE حين يريد أن يساعدنا! أما إذا شغلناها في سطر الأوامر فسيعمل كل شيء بسلاسة.

كما نلاحظ أننا لا نطبق معالج أحداث المفاتيح، لأن السلوك الافتراضي لودجِت أو تطبيق النص هو طباعة المفاتيح المضغوطة، لكن هذا يعني أن برامجنا ليست متكافئةً وظيفيًا، فقد طبعنا رموز آسكي في الطرفية بدلًا من طباعة النسخة الأبجدية من المفاتيح القابلة للطباعة كما فعلنا هنا، فليس ثمة شيء يمنعنا من التقاط جميع ضغطات المفاتيح وفعل نفس الشيء، فإذا أردنا تنفيذ هذا فسنضيف السطر التالي إلى تابع `__init__`:

```
self.txtBox.bind("<Key>", self.doKeyEvent)
```

كما سنضيف التابع التالي لمعالجة الحدث:

```
def doKeyEvent(self,event):
    str = "%d\n" % event.keycode
    self.txtBox.insert(END, str)
    return "break"
```

نلاحظ تخزين قيمة المفتاح في حقل `keycode` للحدث، وقد اضطررت إلى العودة إلى الشيفرة الأساسية للملف Tkinter.py لمعرفة ذلك، تذكر أن الفضول هو الصفة المميزة للمبرمج!

كما نلاحظ أن `return "break"` هي إشارة سحرية تخبر Tkinter بألا يستدعي معالجة الأحداث الافتراضية لهذه الودجِت، وبدون هذا السطر سيعرض الصندوق النصي رمز آسكي متبوعًا بالمحرف الحقيقي الذي ضُغط، وليس هذا ما نريده.

إلى هنا يكفي الحديث عن Tkinter، فلم نكن ننو شرحه هنا وإنما في فصل تالٍ.

18.3 البرمجة الحدثية في VBScript وجافاسكربت

نستطيع تطبيق البرمجة الحدثية في كل من جافاسكربت و VBScript عند برمجة متصفح، فعادةً إذا حُمِّلَت صفحة ويب تحتوي على سكربت فإن السكربت ينفَّذ جزءًا جزءًا مع تحميل الصفحة، لكن إذا لم يحوِ السكربت إلا تعريفات الدوال فلن يفعل التنفيذ شيئًا إلا تعريف الدوال على أنها جاهزة للاستخدام، أما الدوال نفسها فلن

تُستدعى هنا ابتداءً، بل ستكون مقيدةً إلى عناصر HTML في الجزء الخاص بشيفرة HTML في الصفحة -داخل عنصر Form غالبًا-، بحيث تُستدعى هذه الدوال عند وقوع الأحداث، وقد رأينا هذا في مثال جافاسكربت الخاص بالحصول على مدخلات المستخدم عندما نقرأ المدخلات من استمارة HTML، لننظر في هذا المثال مرةً أخرى؛ ونر كيف أنه تطبيق عملي على برمجة حدَثية في صفحة ويب:

```
<form name='entry'>
<p>Type value then click outside the field with your mouse</p>
<input Type='text'
          Name='data'
          onChange='alert("We got a value of " +
document.entry.data.value);'/>
</form>
```

نلاحظ عدم وجود تعريف لدالة جافاسكربت، بل مجرد استدعاء لـ alert المرتبطة بسمة onChange لعنصر input، وهذه السمة هي إحدى الأحداث التي تستطيع عناصر HTML توليدها، ونستطيع ربط أي شيفرة جافاسكربت عشوائية لتنفيذها في كل مرة تقع فيها هذه الأحداث، كما يمكن إنشاء دالة واستدعاؤها بدلًا من استدعاء alert كما يلي:

```
<script type="text/javascript">
function echoValue(){
   alert("We got a value of " + document.entry.data.value);
}
</script>

<form name='entry'>
<p>Type value then click outside the field with your mouse</p>
<input Type='text' Name='data' onChange='echoValue()'/>
</form>
```

يعرّف الجزء الخاص بالسكربت دالة جافاسكربت هي echoValue تحاكي استدعاء alert الذي كان معنا من قبل، ويحتوي عنصر input الآن على دالة مسندة على أنها معالج الأحداث للسمة onChange، ثم تنفَّذ الدالة عند تغير قيمة الدخل، وتكون حلقة الحدث التي تلتقط الأحداث مضمنةً داخل المتصفح.

ورغم أن دالتنا هذه استدعت alert إلا أنها تستطيع أكثر من ذلك، بل من الممكن جعلها برنامجًا معقدًا بذاتها، وذلك وفقًا لما نضعه في متنها.

يمكن استخدام VBScript بنفس الطريقة، عدا أن تعريفات الدوال ستكون بـ VBscript بدلًا من جافاسكربت، كما يلي:

```
<script type="text/vbscript">
Sub EchoInput()
   MsgBox "We got a value of " & Document.entry2.data.value
End Sub
</script>

<form name='entry2'>
<p>Type value then click outside the field with your mouse</p>
<input Type='text' Name='data' onChange='EchoInput()'/>
</form>
```

وبهذا نرى أن الشيفرة الموجهة للمتصفحات يمكن كتابتها في صورة أجزاء batches أو في صورة حدَثية event driven، أو بهما معًا، وفق متطلبات كل حالة.

18.4 خاتمة

نأمل في هذا الفصل أن تكون تعلمت ما يلي:

- لا تبالي حلقات الأحداث بالأحداث التي تلتقطها.
- تعالج معالجات الأحداث حدثًا واحدًا في كل مرة.
- توفر أطر العمل -مثل Tkinter- حلقة أحداث، وبعض معالجات الأحداث الافتراضية أحيانًا.
- يمكن كتابة الشيفرة لمتصفحات الويب بأسلوب مدفوع بالأحداث، أو بالأجزاء، أو بهما معًا.

19. برمجة الواجهات الرسومية باستخدام Tkinter

سنلقي في هذا الفصل نظـرةً عامـةً على كيفيـة تجميـع برنـامج رسـومي بشـرح المفـاهيم الأساسـية لبنـاء الواجهات الرسومية، ثم كيفية تنفيـذه باسـتخدام صـندوق أدوات Tkinter الرسـومي الخـاص ببـايثون، لكن لن يكون هذا الشرح مرجعًا لصندوق Tkinter بحال من الأحوال، إذ يحتوي موقع بايثون على شرح مفصـل لـه، مـع ملاحظة أن ذلك الشرح يستخدم الإصدار الثاني من بايثون الذي يحتوي على أسماء مختلفة للوحـدات، لـذا يجب التحقق من قائمة وحدات بايثون للحصول على الأسماء الصحيحة للوحدات المستوردة، أمـا شـرحنا فيسـتعرض أساسيات برمجة الواجهات الرسومية، ويشرح بعض المكونات الأساسية للواجهات وكيفية استخدامها، كما ينظر في كيفية استخدام البرمجة الكائنيـة التوجـه في تنظيم تطـبيق رسـومي، وبهـذا تكـون العناصـر الأساسـية لهـذا الفصل هي:

- مفاهيم بناء الواجهات الرسومية البسيطة.
- الوِدجات البسيطة أو عناصر واجهة المستخدم.
- هيكل برنامج Tkinter بسيط.
- الواجهات الرسومية والبرمجة الكائنية التوجه، تطابق مثالي.
- wxPython بديل Tkinter.

19.1 مبادئ الواجهات الرسومية

لن نتعرض لمفاهيم برمجية جديدة هنا، فبرمجة الواجهات الرسومية تشبه أي نوع آخر من البرمجة، إذ تحوي تسلسلات sequences، ووحدات modules، وفروعًا branches، وحلقات تكراريةً loops، أما الأمـر الإضـافي فيها فهو استخدام صندوق أدوات Toolkit، واتباعنا نمطًا معينًا في تصميم البرمجيات يحدده من كتب صندوق

الأدوات ذاك، لذا يكون لكل صندوق أدوات مجموعته الخاصة من الوحـدات والأصـناف والـدوال، والـتي تعـرف باسم واجهة برمجة التطبيقات Application Programming Interface، واختصارًا API، كما يحتوي صندوق الأدوات على مجموعة من قواعد التصميم، وعلينا -نحن المبرمجون- أن نتعلم واجهة برمجـة التطبيقـات وقواعـد التصميم معًا، ولهذا يحاول أغلبنا اعتمـاد صـناديق أدوات قليلـةً تكـون متاحـةً في عـدة لغـات برمجـة، لأن تعلم استخدام صندوق الأدوات أصعب من تعلم لغة البرمجة نفسها.

19.1.1 صناديق أدوات الواجهات الرسومية

تأتي أغلب لغات برمجة نظام التشغيل ويندوز مع صندوق أدوات مضمن فيها، ويكـون طبقـةً خفيفـةً فـوق صندوق الأدوات البـدائي المضـمن في نظـام النوافـذ نفسـه، ويوجـد في لغـات مثـل Visual Basic وDelphi وVisual C++/.NET، أما جافا فتختلف عن لغات ويندوز في أنها تحتـوي على صـندوق أدوات الرسـوم الخـاص بها، بل أكثر من صندوق واحد في الواقع، وتعمل هذه الصناديق على أي منصة تعمـل عليهـا جافـا، وهـذا يعـني جميع المنصات تقريبًا، وتوجد صناديق أدوات أخرى يمكن الحصول عليها بشكل مستقل وتُستخدم في أي نظام تشغيل سواء كان لينكس أو ويندوز أو ماك، وتحتوي محولات adapters لتسمح للغات المختلفة باسـتخدامها، وبعض تلك الصناديق تجاري مدفوع، وبعضها مجاني أو حر، والأمثلة على هذه الصناديق تشـمل GTK وQt وTk وwxWidegets، ولها جميعًا مواقع وتدعم ويندوز وماك ولينكس:

- wxPython: النسخة الخاصة ببايثون من wxWidgets، وهو مكتوب بلغة C++، وتسمى رابطة بـايثون إلى wxWidgets باسم WxPython.
- PyQt, the Qt toolkit: يحتوي على روابط لأغلب لغات البرمجة، وتُعرف روابط بايثون إلى Qt باسـم PyQt.
- GTK+: مجموعة من العناصر والأدوات لإنشاء واجهات رسومية، ورابطة بايثون فيه اسمها PyGTk.

تُكتب أغلب برامج لينكس باسـتخدام Qt وGTk، وكلاهمـا مجـاني للاسـتخدامات غـير التجاريـة، ويـوفر Qt رخصةً تجاريةً لمن شاء، في حين أن رخصة GTk هي رخصة Gnu GPL التي لها شروطها الخاصة بها.

صندوق الأدوات الرسومي الخاص ببايثون والذي يأتي افتراضيًا مع اللغة هو Tkinter، المبني على Tk، وهو صندوق أدوات قديم متعدد نظم التشـغيل، وسندرسـه هنـا، حيث توجـد منـه إصـدارات للغـات Tcl وHaskell وRuby وPerl وبايثون، وتختلـف المبـادئ المسـتخدمة في Tk عن صـناديق الأدوات الأخـرى قليلًا، لـذا سـنذكر باختصـار نظـرةً على صـندوق أدوات رسـومي آخـر لبـايثون و C و C++، يكـون تقليـديًا في منظـوره العـام، لكن سنتعرف أولًا على بعض المفاهيم العامة.

19.1.2 عناصر الواجهات الرسومية الأساسية

ذكرنا سابقًا أن التطبيقات الرسومية ذات طبيعة حدَثيـة -مدفوعـة بالأحـداث event-driven- في الغـالب، ويمكن الرجوع إلى الفصل الثامن عشر: البرمجة المدفوعة بالأحداث، للاطلاع على هذا المفهوم، وسنفترض أنـك مستخدم معتاد على التعامل مع الواجهات الرسومية، وسـنركز على كيفيـة عمـل الـبرامج الرسـومية من منظـور المبرمج، ولن ندخل في تفاصيل كيفية كتابة واجهات رسومية معقدة وكبيرة لها نوافذ متعـددة أو واجهـات MDI أو غيرها، بل سـنكتفي بأساسـيات إنشـاء تطـبيق وحيـد النافـذة فيـه بعض العنـاوين والأزرار والحقـول النصـية وصناديق الرسائل.

نحتاج أولًا إلى التحقق من المصـطلحات الـتي سنسـتخدمها، فلبرمجـة الواجهـات مجموعتهـا الخاصـة من المصطلحات البرمجية، وأكثر المصطلحات استخدامًا فيها هي:

ا. النافذة window

جزء من الشاشـة يتحكم فيـه التطـبيق، وتكـون النوافـذ مربعـةً في العـادة، لكن قـد تسـمح بعض البيئـات الرسومية بأشكال أخرى، وقد تحتوي النوافذ على نوافذ أخرى داخلها، ويُعامل كل متحكم رسومي GUI control على أنه نافذة بذاته.

ب. المتحكم Control

كائن رسومي يُستخدم للتحكم في التطبيق، وتحتوي المتحكمات على خصائص properties، وتولّد أحـداثًا events، وتسـتجيب عـادةً لكائنـات على مسـتوى التطـبيق، حيث تُرفَـق الأحـداث بتوابـع الكـائن الموافـق corresponding object، فإذا وقـع الحـدث نفّـذ الكـائن أحـد توابعـه، وتـوفر الواجهـة الرسـومية آليـات لربط الأحداث بالتوابع.

ج. الودجِت Widget

متحكم مقيَّد عادةً بالمتحكمات المرئية، إذ يمكن ربط بعض المتحكمـات -مثـل المؤقتـات timers- بنافـذة ما، لكن دون أن تكون مرئيةً، أما الودجات فهي فئة مرئية فرعية من المتحكمات، ويمكن للمسـتخدم أو المـبرمج أن يعدل فيها.

سـنغطي في هـذا الفصـل الودجـات Frame وLabel وButton وEntry Text وbox Message، كمـا سنتعرض لاحقًا للودجـات Text box وRadio Button، أمـا الودجـات الـتي لن نـذكرها مطلقًـا فهي Canvas الخاصة بالرسم، وCheck button الخاصة بالاختيار المتعدد، وImage الخاص بعرض صور BMP وGIF وJPEG وPNG، وListbox للقوائم، و Menu/MenuButton لبناء القوائم المنسدلة menus، وScale/Scrollbar الـتي توضح الموضع.

د. الإطار Frame

نوع من الودجات يُستخدم لدمج ودجات أخرى معًا، ويُستخدم عادةً لتمثيل النافذة ككل، ويمكن دمج إطارات أخرى فيه.

ه. التخطيط Layout

توضع المتحكمات في إطار وفقًا لمجموعة محددة من القواعد أو الإرشادات، وتشكل تلك القواعد ما يعرف بالتخطيط، وقد يُحدَّد بعدة طرق، مثل استخدام إحداثيات محددة بالبكسل على الشاشة، أو باستخدام مواضع نسبية لمكونات أخرى (مثل اليسار، الأعلى)، أو باستخدام شبكة أو جدول. ومن السهل فهم نظام الإحداثيات، لكن تصعب إدارته عند تعديل حجم النوافذ مثلًا، ويُنصح المبتدئون باستخدام نوافذ لا يمكن تغيير حجمها إذا استخدموا تخطيطات مبنيةً على إحداثيات.

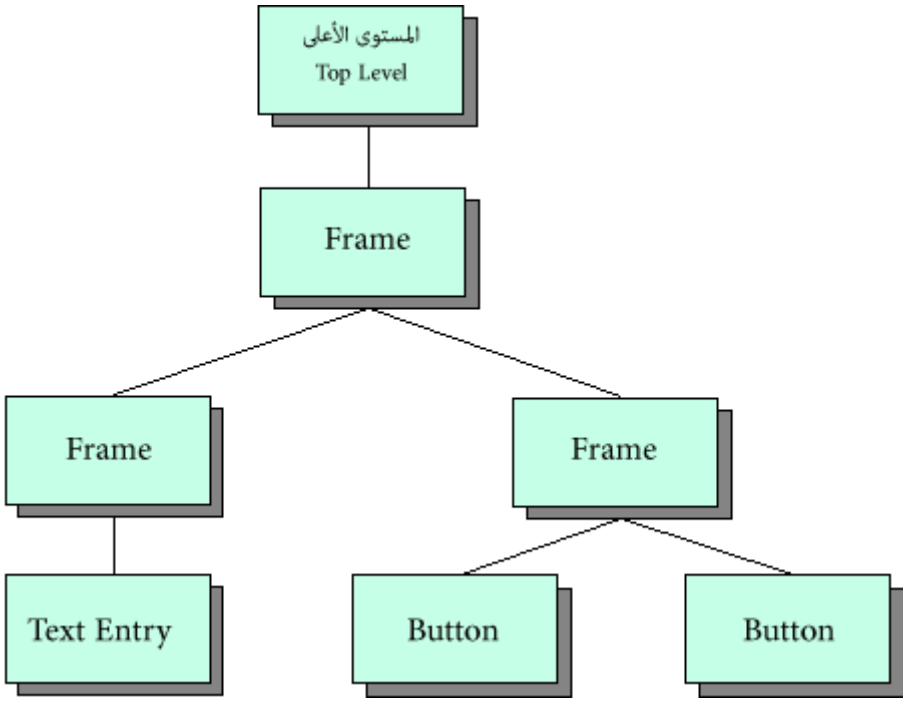
و. الأب/الابن Parent/Child

تميل التطبيقات الرسومية لأن تتكون من تسلسل هرمي من المتحكمات أو الودجات، حيث يحتوي إطار المستوى الأعلى الذي يشغل نافذة التطبيق على إطارات فرعية تحتوي على إطارات أو متحكمات أخرى، وينظر إلى تلك المتحكمات على أنها هيكل شجري يحتوي كل متحكم فيه على أب واحد وعدة أبناء، بل يُخزَّن هذا الهيكل عادةً بصراحة بواسطة ودجات، ليستطيع المبرمج -أو البيئة الرسومية نفسها غالبًا- تنفيذ بعض الإجراءات الشائعة لمتحكم ما وجميع أبنائه، فإغلاق الودجت العليا ينتج عنه إغلاق جميع الودجات الفرعية.

19.1.3 شجرة الاحتواء Containment tree

من المفاهيم التي يجب استيعابها في برمجة الواجهات الرسومية هرمية الاحتواء containment hierarchy، حيث تُحتوى الودجات داخل هيكل شجري تتحكم فيه ودجت المستوى الأعلى بالواجهة كلها، ويكون لذلك الهيكل ودجات فرعية يمكن أن تحتوي بدورها على ودجات أخرى فرعية خاصة بها، وتصل الأحداث إلى ودجت فرعية تمرر الحدث -إذا لم تستطع معالجته- إلى الودجت الرئيسية (الأب) لها، وهكذا إلى أن نصل إلى ودجت المستوى الأعلى، وإذا أُعطي أمر لرسم ودجت ما فسيرسَل الأمر إلى الودجات الفرعية، وعليه فإن أمر الرسم إلى ودجت المستوى الأعلى قد يعيد رسم التطبيق كله، في حين أن أمر الرسم المرسَل إلى زر ما لن يعيد إلا رسم الزر فقط.

ومفهوم صعود الأحداث إلى أعلى الشجرة ودفع الأوامر إلى الأسفل ضروري لفهم كيفية عمل الواجهات الرسومية بالنسبة للمبرمج، وهو السبب في أننا نحتاج إلى تحديد أب للودجت عند إنشائها، لتعرف الودجت مكانها في شجرة الاحتواء. وتُرسم شجرة الاحتواء لتطبيق بسيط سننشئه لاحقًا في هذا الموضوع كما يلي:



توضح الصورة أعلاه ودجت المستوى الأعلى التي تحتوي على إطار `Frame` واحد يمثل الحد border الخارجي للنافذة، والذي يحتوي بدوره على إطارين آخرين، في الأول منهما ودجت `Text Entry`، وفي الثاني زران `Buttons` يُستخدمان للتحكم في التطبيق، وسنشير إلى هذا المخطط لاحقًا حين نأتي لبناء الواجهة الرسومية.

19.2 نظرة على بعض الودجات الشائعة

سنستخدم محث بايثون التفاعلي في هذا القسم لإنشاء بعض النوافذ والودجات البسيطة، ورغم أن IDLE نفسه ما هو إلا تطبيق Tkinter لكن لا يُمكن الاعتماد عليه في تشغيل تطبيقات Tkinter داخله، وإنما نستطيع إنشاء الملفات باستخدامه مثل محرر، ثم نشغلها في بيئة التطوير IDE كالمعتاد، رغم احتمال حدوث أشياء غير متوقعة هنا، فإذا وقعت تصرفات غريبة فيجب أن نشغل التطبيق من سطر أوامر النظام قبل أن نجرب شيئًا آخر، فربما تكون المشكلة تعارضًا بين إطار Tkinter الداخلي لبيئة IDLE وبين نسخته لبرنامجنا، فإذا لم تُحل المشكلة فسنبحث عن العلل البرمجية bugs في شيفرة البرنامج. أما مستخدمو Pythonwin فيستطيعون تشغيل تطبيقات Tkinter مباشرةً دون مشاكل، لأن Pythonwin لا تستخدم Tkinter داخليًا، لكن حتى هنا قد تحدث سلوكيات غير متوقعة في تطبيقات Tkinter، ولهذا نستخدم محث بايثون الأساسي من نافذة نظام التشغيل الطرفية دومًا:

```
>>> from tkinter import *
```

هذا أول متطلبات أي برنامج Tkinter، وهو استيراد أسماء الودجات، يجب أن نستورد الوحدة، لكننا لا نريد كتابة `tkinter` قبل كل اسم مكون، لذا نستخدم الاسم البديل `tk`:

```
>>> import tkinter as tk
```

وهـذا يعـني أننـا نحتـاج إلى سـبق الأسـماء بـ `tk.` فقـط، لكن بمـا أننـا نجـرب هنـا فمن الأسـهل اسـتيراد كل شيء:

```
>>> top = Tk()
```

ينشئ هذا ودجت المسـتوى الأعلى في هرميـة الودجـات، ونلاحـظ حالـة الأحـرف في `Tk`، حيث كـان اسـم الوحدة بالأحرف الصغيرة، لكن اسم الودجت بأحرف كبيرة، وتُنشأ جميع الودجات الأخرى فروعًا لودجت `top`.

يعتمد ما يحدث هنا على المكان الذي نكتب البرنامج فيه، فإذا كنا نستخدم بايثون من محث النظام فيجب أن تظهر نافذة جديدة كاملة بشريط عنوان فار غ مع شعار Tk أيقونةً، وأزرار التحكم المعتادة من التكبير والتصغير وغيرها، أما إذا كنا نستخدم بيئة تطوير IDE فقد لا نرى شيئًا، وقد لا تظهر النافذة إلا عند إكمال واجهة المستخدم الرسومية والبدء بتشغيل حلقة الحدث الرئيسية.

سنضيف المكونات إلى هذه النافذة أثناء بنائنا للتطبيق، ولننظر الآن إلى ما لدينا:

```
>>> dir(top)
[....lots of stuff!...]
```

توضح دالة `dir()` الأسماء المعروفـة للوسـيط، ونسـتطيع اسـتخدامها على الوحـدات لكننـا ننظـر هنـا إلى المكونات الداخلية للكائن `top`، وهو نسخة من الصنف `Tk`، حيث نلاحظ وجود سمات كثيرة للكائن `top`، نخص بالـذكر منهـا `children` و`master` اللتـان تمثلان روابـط إلى شـجرة احتـواء الـودجت، كمـا نلاحـظ سـمة `_tclCommands_`، لأن Tkinter بُني على صندوق أدوات من Tcl اسمه Tk.

```
>>> F = Frame(top)
```

لننشئ ودجت إطار `Frame` هنا تحتوي المتحكمات/الودجات الفرعية التي نستخدمها، ويحدد `Frame` كـائن `top` على أنه معامله الأول -والوحيد في هذه الحالة-، وعليـه فـإن `F` سـتكون ودجت فرعيـةً للكـائن `top`، ,يمكن التحقق من هذا بسهولة كما يلي:

```
>>> top.children
{'!frame': <tkinter.Frame object .!frame>}
>>> F.master
<tkinter.Tk object .>
```

نـرى هنـا أن `top.children` مـا هـو إلا قـاموس يربـط اسـمًا غريبًـا قليلًا هـو `'!frame'` بمرجـع كـائن object reference، وهذه التسمية الغريبة خاصة بـ Tcl/Tk وانتقلت إلى Tkinter، كما أن `F.master` ما هـو

إلا مرجع إلى top، وننصحك هنا بالتدرب على السمتين master وchildren لودجاتنا، مما يسهل عليك فهم الصلة بين الودجات وشجرة الاحتواء التي ذكرناها من قبل.

```
>>> F.pack()
```

لاحظ الآن تقلص نافذة Tk -إذا كانت مرئيةً- إلى حجم ودجت الإطار المضاف، وهي صغيرة للغاية لأن الودجت فارغة، لكن ينبغي أن نكون قادرين على تغيير حجمها بالفأرة وأزرار التصغير والتكبير في شريط عنوانها.

يستدعي التابع pack() مدير تخطيط يُعرف باسم المحزِّم Packer، وهو سهل الاستخدام في التخطيطات البسيطة، لكنه يصبح صعبًا كلما زاد تعقيد التخطيط، وسنستخدمه الآن لسهولته، حيث يكدس هذا المحزِّم الودجات بعضها فوق بعض، ونلاحظ أن الودجات لن تكون مرئيةً في تطبيقنا حتى نحزمها أو نستخدم تابع مدير تخطيط آخر، وسنتحدث عن مدراء التخطيطات لاحقًا، بعد إنهاء هذا البرنامج.

```
>>> lHello = Label(F, text="Hello world")
```

ننشئ هنا كائنًا جديدًا هو lHello، وهو نسخة من الصنف Label مع ودجت أب هي F، وسمة text قيمتها "Hello World"، نلاحظ أنه من المعتاد استخدام تقنية المعامِل المسمى بتمرير الوسائط إلى كائنات Tkinter بسبب ميل منشئات كائنات Tkinter لامتلاك عدة معامِلات لكل منها قيمته الافتراضية، كما نلاحظ أن الكائن ليس مرئيًا بعد لأننا لم نحزّمه.

كما نلاحظ استخدام اصطلاح تسمية هنا، وهو حرف l الذي يشير إلى كلمة Label قبل الاسم Hello الذي يذكرنا بالغرض منه، ومسألة اصطلاحات التسمية هذه هي في الحقيقة أمور شخصية، لكنها مفيدة لهذا الغرض الذي ذكرناه.

```
>>> lHello.pack()
```

نستطيع الآن أن نراها، وينبغي أن تكون النافذة التي أنشأتها لديك شبيهة بهذه:

تُحدَّد خصائص Label -مثل الخط واللون- باستخدام معامِلات منشئ الكائن أيضًا، ونستطيع الوصول إلى الخصائص الموافقة corresponding باستخدام التابع configure الخاص بودجات Tkinter كما يلي:

```
>>> lHello.configure(text="Goodbye")
```

لقد تغيرت الرسالة هنا، لذا نرى أن تقنية configure ممتازة عند تغيير عدة خصائص مرةً واحدةً لتمريرها جميعًا على أنها وسطاء، لكن إذا أردنا تغيير خاصية واحدة فقط في كل مرة كما فعلنا أعلاه فيمكن معاملة الكائن على أنه قاموس، وهذا أقصر وأسهل في الفهم:

```
>>> lHello['text'] = "Hello again"
```

إن ودجات العناوين مملة، ولا تعرض إلا نصوصًا قابلة للقراءة فقط، وإن كانت بألوان وخطوط مختلفة، رغم إمكانية استخدامها لعرض رسوم بسيطة، لكننا لن نتطرق إلى هذا هنا، نقول هذا لننظر في نوع كائن آخر، لكن ثمة شيء نفعله وهو إعداد عنوان النافذة، وذلك باستخدام تابع من ودجت المستوى الأعلى top:

```
>>> F.master.title("Hello")
```

كان بإمكاننا استخدام top مباشرةً، لكن الوصول من خلال الخاصية الرئيسية للإطار مفيد، كما سنرى لاحقًا:

```
>>> bQuit = Button(F, text="Quit", command=F.quit)
```

ننشئ هنا ودجت جديدةً هي زر له عنوان Quit، ويرتبط بالأمر F.quit، لاحظ أننا نمرر اسم التابع ولا نستدعيه بإضافة أقواس بعده، وهذا يعني أننا يجب أن نمرر كائن دالة وفقًا لبايثون، سواء كان تابعًا مضمنًا في Tkinter -كما ف حالتنا- أو أي دالة أخرى نعرّفها.

يجب ألا تأخذ الدالة أو التابع أي وسطاء، ويُعرَّف التابع quit في صنف أساسي -وكذلك التابع pack- وترثه جميع ودجات Tkinter، لكنه يُستعدى في الغالب في مستوى النافذة العليا للتطبيق.

```
>>> bQuit.pack()
```

يجعل التابع pack الزر مرئيًا مرةً أخرى هنا، رغم أنك قد تحتاج إلى تغيير حجم النافذة لتراه وفقًا لإعدادات نظام التشغيل لديك، فلن يحدث شيء إذا ضغطت عليه، لأننا لا نملك حلقة أحداث تعمل الآن لتلتقط حدث ضغط الزر وتعالجه.

```
>>> top.mainloop()
```

وأخيرًا نبدأ حلقة حدث Tkinter، لاحظ اختفاء محث بايثون >>>، وهذا يخبرنا أن Tkinter هو المتحكم الآن، فإذا ضغطنا زر Quit فسيعود المحث ليثبت لنا أن خيار command يعمل، لا تتوقع أن تنغلق النافذة، فلا زال مفسر بايثون يعمل ولم نرد إلا الخروج من دالة mainloop، فإذا خرجنا من بايثون فستُدمَّر الودجات الموجودة، ويكون هذا -في البرامج الحقيقية- بعد انتهاء mainloop مباشرة.

ونلاحظ أنه إذا شغلنا هذا من IDLE أو Pythonwin فلم نكن لنرى شيئًا إلى الآن، ولحصلنا على نتيجة مختلفة قليلًا، وإذا حدث هذا معك فاكتب الأوامر التي كتبناها إلى الآن في سكربت بايثون ثم شغلها من سطر أوامر النظام، ومن المناسب الآن أن تجرب ذلك على أي حال بما أنه الكيفية التي ستشغل بها برامج Tkinter في الممارسة العملية، واستخدم الأوامر الأساسية التي شرحناها حتى الآن وكما وضحنا، واستخدم نمط الاستيراد المفضل:

```
import tkinter as tk

# إعداد النافذة ذاتها
top = tk.Tk()
F = tk.Frame(top)
F.pack()

# إضافة الودجات
lHello = tk.Label(F, text="Hello")
lHello.pack()
bQuit = tk.Button(F, text="Quit", command=F.quit)
bQuit.pack()

# تشغيل الحلقة التكرارية
top.mainloop()
```

يبدأ استدعاء التابع `top.mainloop` حلقة حدث Tkinter لتوليد الأحداث، والحدث الوحيد الذي نلتقطه في هذه الحالة سيكون حدث ضغط الزر المتصل بالتابع `F.quit`، وينهي الأخير التطبيق وستنغلق النافذة هذه المرة لأن بايثون قد خرجت هي الأخرى، جربها بنفسك الآن، ينبغي أن تبدو كما يلي:

لاحظ أننا نسينا السطر الذي يغير عنوان النافذة، جرب إضافة ذلك السطر بنفسك وتحقق من نجاح ذلك.

19.3 استكشاف التخطيط

سنبدأ من الآن بإعطاء الأمثلة في ملفات سكربتات بايثون بدلًا من أوامر في محث `>>>`، وسنوفر مقتطفات من الشيفرات في الغالب لتكتب أنت الاستدعاءات إلى `Tk()` و`mainloop()` بنفسك، فاستخدم البرنامج السابق قالبًا، وسننظر في هذا القسم في توضع الودجات في النافذة في Tkinter، وقد رأينا ودجات `Frame` و`Label` و`Button` من قبل، وهي كل ما نحتاج إليه في هذا القسم، وقد استخدمنا التابع `pack` الخاص بالودجت في المثال السابق لتحديد موقعها في الودجت الأب لها، والواقع أن ما نفعله هو استدعاء مدير تخطيط المحزِّم الخاص بـ Tk، ويطلق عليه أحيانًا اسم المدير الهندسي Geometry Manager، ووظيفته تحديد أفضل تخطيط للودجات بناءً على الإرشادات التي يوفرها المبرمج، إضافةً إلى القيود مثل حجم النافذة

التي يتحكم بها المستخدم، ويستخدم بعض مدراء التخطيطات نفس المواقع داخل النافذة محددةً بالبكسل، وهذا أمر شائع في بيئات ويندوز مثل Visual Basic.

ويحتوي Tkinter على مدير تخطيط واضع placer layout manager يستطيع تنفيذ ذلك من خلال تابع `place`، ولن ننظر فيه لوجود خيارات أخرى أفضل وأكثر ذكاءً للمدراء، لأنها توفر علينا التفكير -نحن المبرمجين- في ما يحدث عند تغيير حجم نافذة ما، وأبسط مدير تخطيط في Tkinter هو برنامج التحزيم الذي كنا نستخدمه، وهو يكدس الودجات بعضها فوق بعض، ويمكن تغيير هذا السلوك لتكديس الودجات يسارًا ويمينًا لكن هذا محدود للغاية، ونادرًا ما نريد ذلك من الودجات العادية، لكن إذا أردنا بناء تطبيقاتنا من إطارات `Frames` فيجب أن نكدس الإطارات فوق بعضها، ثم نستطيع وضع الودجات الأخرى في الإطارات باستخدام المحزِّم أو مدير تخطيط آخر داخل كل إطار حسب الحاجة، ويمكن أن يكون لكل إطار مدير التخطيط الخاص به، لكننا لا نستطيع الجمع بين المدراء في إطار واحد، ويمكن رؤية مثال على ذلك في الفصل الثاني والعشرين: دراسة حالة برمجية.

يوفر المحزِّم -وإن كان بسيطًا- عدة خيارات، إذ نستطيع ترتيب ودجاتنا رأسيًا أو أفقيًا، ونستطيع تعديل أحجامها وتعديل الفواصل بينها والإطار التي تحاذيه، لننظر في المثال التالي على التحزيم الأفقي:

```
lHello = tk.Label(F, text="Hello")
lHello.pack(side="left")
bQuit = tk.Button(F, text="Quit", command=F.quit)
bQuit.pack(side="left")
```

سيؤدي هذا إلى إجبار الودجات على الانتقال إلى اليسار، لذا ستظهر الودجت الأولى -وهي العنوان- في أقصى اليسار، متبوعةً بالودجت التالية -الزر-، فإذا عدّلنا الأسطر في المثال أعلاه فسيبدو كما يلي:

وإذا غيرنا `"left"` إلى `"right"` فسيظهر العنوان على أقصى اليمين، وسيظهر الزر على يساره، كما يلي:

يجب أن نلاحظ أن الشكل غير لطيف، لأن الودجات مضغوطة إلى جانب بعضها البعض، ويزودنا المحزِّم ببعض المعامِلات للتعامل مع ذلك، لعل أسهلها الحشو padding الذي يُحدَّد بحشو أفقي `padx` ورأسي `pady`، وتُكتب تلك القيم بالبكسل، لنضف حشوًا أفقيًا إلى مثالنا:

```
lHello.pack(side="left", padx=10)
bQuit.pack(side='left', padx=10)
```

يجب أن يبدو كما يلي:

إذا حاولنا تغيير عرض النافذة فسنرى أن الودجات تحافظ على مواضعها النسبية، لكنها ستظل في مركز النافذة، لأننا -رغم إضافتنا الحشو إلى اليسار- حزمنا الودجات في إطار Frame؛ وحزّمنا الإطار نفسه دون جانب side، لذا فإن موضعه يكون إلى الأعلى والمنتصف، وهو الافتراضي للمحزِّمات، فإذا أردنا أن تبقى الودجات في الجانب الصحيح من النافذة فيجب أن نحزم الإطار إلى الجانب الصحيح كذلك:

```
F.pack(side='left')
```

نلاحظ هنا أن الودجات تظل في المنتصف إذا غيرنا حجم النافذة الرأسي، وهذا هو السلوك الافتراضي للمحزِّمات أيضًا، سنترك لك تجربة تغيير padx وpady لترى تأثير القيم المختلفة عليها. وتسمح side وpadx/pady بمرونة كبيرة في تحديد مواضع الودجات باستخدام المحزِّم، وتوجد خيارات أخرى يضيف كل منها شكلًا خفيفًا من أشكال التحكم، يُرجَع فيها إلى توثيق Tkinter.

يوجد عدة مدراء تخطيطات آخرين في Tkinter مثل الشبكة grid والواضع placer، إضافةً إلى وحدة Tix التي تعزز Tkinter وتوفر مدير التخطيط Form، لكننا لن نشرح هذه الوحدة هنا لأنها أُهملت في المكتبة القياسية رسميًا.

نستخدم grid() إذا أردنا استخدام مدير الشبكة grid manager، بدلًا من pack() التي استخدمناها أعلاه، أما في الواضع placer فنستدعي place() بدلًا من pack()، ولكل منها مجموعة خيارات خاصة، وبما أننا سنشرح المحزِّم فقط هنا فيُرجَع إلى توثيق Tkinter لمزيد من التفاصيل عن هؤلاء المدراء، لكن النقاط الأساسية التي نريد الإشارة إليها هنا هي ما يلي:

- تنظم الشبكة المكونات في "شبكة" داخل النافذة، وهذا مفيد في الصناديق الحوارية التي تحوي صناديق إدخال نصيةً مرتبةً مثلًا، ويفضل العديد من مستخدمي Tkinter استخدام الشبكة على المحزِّم، لكن قد يحتاج المبتدئ وقتًا حتى يتعلمها، خاصةً عندما يريد أن يشغل مكون عدة خلايا من الشبكة.
- يستخدم الواضع إحداثيات ثابتةً بالبكسل أو إحداثيات نسبيةً داخل النافذة، وتسمح الأخيرة بتغيير حجم المكونات مع تغير حجم النافذة، كأن تظل المساحة التي تشغلها 75% من المساحة الرأسية للنافذة مثلًا، لكن قد يبدو مظهر الأزرار غريبًا إذا زاد عرضها كثيرًا، وتظهر فائدة هذا في التصاميم المعقدة للنوافذ، لكنها تحتاج إلى كثير من التخطيط المسبق، لذا يُنصح باستخدام الورق والقلم!

19.4 التحكم في المظهر باستخدام الإطارات والمحزم

تحتوي ودجت الإطار Frame على العديد من الخصائص المفيدة التي يمكن استخدامها، فمن الجميل أن يكون لدينا إطار منطقي غير مرئي حول المكونات، لكننا قد نرغب في رؤيته، خاصةً عند جمع عدة متحكمات مثل أزرار الانتقاء radio buttons أو صناديق الاختيار check boxes، ويحل الإطار هذه المشكلة بتوفير حد يُعرف بخاصية المساعدة relief property، على غرار العديد من ودجات Tkinter الأخرى، يمكن أن تأخذ `Relief` إحدى القيم التالية:

- `sunken`: غائر.
- `raised`: مرتفع.
- `groove`: محفور.
- `ridge`: مشطوف.
- `flat`: مسطح.

لنستخدم القيمة `sunken` على صندوق حوارنا البسيط، بتغيير سطر إنشاء الإطار `Frame` إلى ما يلي:

```
F = Frame(top, relief="sunken", border=1)
```

لاحظ أن عليك وضع حد border أيضًا، فإن لم تفعل فسيكون الإطار غاطسًا لكنه سيكون مخفيًا لأن حدوده غير مرئية، فلن ترى فرقًا إذا لم تضف الحد، ولا تضع حجم الحد بين علامتي اقتباس، وهذا أحد الأمور المربكة في برمجة Tk، أي معرفة متى نستخدم علامات الاقتباس حول خيار ما ومتى لا نستخدمها، لكن القاعدة العامة هي أنه يمكن ترك علامات الاقتباس إذا كانت القيمة عدديةً، أما إذا كانت مزيجًا بين الأرقام والحروف أو سلسلةً نصيةً فيجب وضع علامات الاقتباس، يمكن قول ذلك على حالة الأحرف التي يجب استخدامها، لكن للأسف لا توجد طريقة سهلة لمعرفة ذلك هنا، بل يجب أن نتعلمها بالخبرة، وتعطيك بايثون عادةً قائمةً من الخيارات الصالحة في رسائل الخطأ الخاصة بها.

ومما يجب ملاحظته أيضًا أن الإطار لا يملأ النافذة، ونستطيع إصلاح ذلك بخيار محزِّم آخر هو `fill`، فنفعل ما يلي عند تحزيم الإطار:

```
F.pack(fill="x")
```

ستُملأ النافذة عرضيًا كما هو واضح من الإحداثي x، فإذا أردنا ملأها كلها فنستخدم `fill='y'` أيضًا، ويوجد خيار ملء خاص هو `both` بسبب شيوع هذه العملية:

```
F.pack(fill="both")
```

ينبغي أن تكون النتيجة النهائية بعد تشغيل السكربت كما يلي:

19.5 إضافة ودجات أخرى

لننظر الآن في الودجت `Entry`، وهو السطر الواحد ذو الشكل المألوف لصندوق الإدخال النصي، والذي يتشارك الكثيرَ من التوابع مع ودجت النص متعدد الأسطر الذي استخدمناه في الفصل الثامن عشر: البرمجة المدفوعة بالأحداث، كما سنستخدمه في الفصل الثاني والعشرين: دراسة حالة برمجية، وسنستخدم `Entry` لالتقاط النص الذي يكتبه المستخدم، ولمحو ذلك النص عند الحاجة.

بالعودة إلى مثال Hello World، سنضيف ودجت إدخال نصي داخل إطار خاص به، ثم نضع زرًا في إطار ثانٍ يستطيع مسح ذلك النص الذي نكتبه داخل خانة الكتابة، ثم نضيف زرًا آخر للخروج من التطبيق، وهذا يوضح كيفية إنشاء واستخدام ودجت الإدخال، وكيفية تعريف دوال معالجة الأحداث الخاصة بنا وتوصيلها بالودجات.

```
import tkinter as tk

# أنشئ معالج الحدث لمسح النص
def evClear():
   eHello.delete(0,tk.END)

# أنشئ نافذة/إطار المستوى الأعلى
top = tk.Tk()
F = tk.Frame(top)
F.pack(fill="both")

# والآن الإطار الذي فيه الإدخال النصي
fEntry = tk.Frame(F, border=1)
eHello = tk.Entry(fEntry)
fEntry.pack(side="top")
eHello.pack(side="left")

# وأخيرًا الإطار الذي فيه الأزرار
# سنجعل هذا غاطسًا لنبرزه
```

```
fButtons = tk.Frame(F, relief="sunken", border=1)
bClear = tk.Button(fButtons, text="Clear Text", command=evClear)
bClear.pack(side="left", padx=5, pady=2)
bQuit = tk.Button(fButtons, text="Quit", command=F.quit)
bQuit.pack(side="left", padx=5, pady=2)
fButtons.pack(side="top", fill="x")

# والآن شغل حلقة الحدث
F.mainloop()
```

نلاحظ هنا أننا عرّفنا معالج الحدث مثل أي دالة أخرى، وبما أننا نريد إسنادها إلى حدث الأمر command event لزر فنعرف أنها يجب ألا تحتوي على معامِلات، رغم أن بعض معالجات الأحداث -مثل أحداث الفأرة- قد تأخذ معامِلات، لكن يجب التحقق من التوثيق لمعرفة المطلوب للحدث.

لاحظ أيضًا أننا نمرر أسماء معالجات الأحداث `evClear` و`F.quit` -دون أقواس- قيمًا للمعامل `command` للأزرار، ولاحظ استخدام اصطلاح التسمية evXXX لربط معالج الحدث بالودجت XXX الموافقة له، لذا سيكون `evClear` هو معالج الحدث لودجت `bClear`.

يستدعي معالج الحدث التابع `delete` الخاص بالودجت `Entry`، ورغم أن نظام الفهرسة المستخدم للوسطاء معقد قليلًا إلا أننا نستطيع في هذا المستوى أن نقول بأنه يمسح النص من الموضع `0` -أي من البداية- إلى الموضع `tk.END` -آخر موضع-، ولاحظ أن `tk.END` ثابت معرَّف في tkinter، ويوجد غيره مما يمكن استخدامه بدلًا من السلاسل الاختيارية `right` و`left` و`top` وغيرها، وذلك راجع لما يفضله كل منا، وبتشغيل البرنامج نحصل على النتيجة التالية:

فإذا كتبت شيئًا في صندوق الإدخال النصي فاضغط Clear Text لمسحه مرةً أخرى.

لاحظ أننا بنينا الواجهة الرسومية التي رسمها مخطط الاحتواء في بداية هذا الفصل، فللودجت العليا إطار `Frame` تحتها، وفيه إطاران تحته، واحد فيه ودجت إدخال والآخر فيه زران، وهذا ما نراه في المخطط بالضبط.

ولا توجد فائدة تذكر من وجود ودجت إدخال إلا إذا كنا نستطيع الوصول إلى النص الذي فيها، ونفعل ذلك باستخدام التابع `get` الخاص بالودجت، وسنوضح هذا بنسخ النص من الودجت إلى عنوان label قبل مسحه، لنستطيع رؤية النص الأخير الذي كان في الودجت، ولفعل ذلك نحتاج إلى إضافة ودجت عنوان تحت ودجت الإدخال، وتوسيع معالج الحدث evClear لينسخ النص، وسنلون نص العنوان بلون أزرق فاتح لإبرازه.

```
import tkinter as tk

# أنشئ معالج الحدث لمسح النص
def evClear():
  lHistory['text'] = eHello.get()
  eHello.delete(0,tk.END)

# أنشئ نافذة/إطار المستوى الأعلى
top = tk.Tk()
F = tk.Frame(top)
F.pack(fill="both")

# والآن الإطار الذي فيه الإدخال النصي
fEntry = tk.Frame(F, border=1)
eHello = tk.Entry(fEntry)
eHello.pack(side="left")
lHistory = tk.Label(fEntry, foreground="steelblue")
lHistory.pack(side="bottom", fill="x")
fEntry.pack(side="top")

# وأخيرًا الإطار الذي فيه الأزرار
# سنجعل هذا غائرًا لنبرزه
fButtons = tk.Frame(F, relief="sunken", border=1)
bClear = tk.Button(fButtons, text="Clear Text", command=evClear)
bClear.pack(side="left", padx=5, pady=2)
bQuit = tk.Button(fButtons, text="Quit", command=F.quit)
bQuit.pack(side="left", padx=5, pady=2)
fButtons.pack(side="top", fill="x")

# والآن شغل حلقة الحدث
F.mainloop()
```

نرى هنا أننا أضفنا السطر التالي عند إنشاء معالج الحدث الذي يمسح النص:

```
  lHistory['text'] = eHello.get()
```

كما أضفنا الأسطر التالية في الإطار الذي يحتوي على إدخال نصي:

```
lHistory = tk.Label(fEntry, foreground="steelblue")
lHistory.pack(side="bottom", fill="x")
```

من الممكن إسناد النص إلى متغير بايثون عادي لنستخدمه لاحقًا في برنامجنا، رغم أننا أسندناه مباشرةً إلى خاصية `Label` هنا.

19.6 أحداث الربط: من الودجات إلى الشيفرة

لقد استخدمنا الخاصية command للأزرار إلى الآن لربط دوال بايثون مع أحداث الواجهة الرسومية، لكننا قد نريد تحكمًا أكثر من هذا أحيانًا، لالتقاط تجميعة مفاتيح معينة مثلًا، ونفعل ذلك باستخدام دالة `bind` لربط حدث ما مع دالة بايثون صراحةً.

سنعرِّف الآن مفتاحًا ساخنًا مثل CTRL+C لحذف النص في المثال أعلاه، سنحتاج هنا أن نربط تجميعة المفاتيح CTRL+C بنفس معالج الحدث الخاص بزر Clear، لكننا سنواجه مشكلةً غير متوقعة هنا، إذ يجب ألا تأخذ الدالة المحددة أي وسطاء عند استخدامنا للخيار command، أما حين نستخدم دالة الربط bind لأداء نفس الوظيفة فيجب أن تأخذ الدالة المحددة وسيطًا واحدًا، لذا نحتاج إلى إنشاء دالة جديدة ذات معامِل وحيد تستدعي `evClear`، أضف الشيفرة التالية بعد تعريف `evClear`:

```
def evHotKey(event):
    evClear()
```

ثم أضف السطر التالي مباشرةً بعد تعريف ودجت الإدخال `eHello`:

```
eHello.bind("<Control-c>",evHotKey) # تعريف المفتاح حساس لحالة الأحرف
```

شغّل البرنامج الآن مرةً أخرى، ستستطيع مسح النص الآن بضغط الزر أو باستخدام تجميعة المفاتيح CTRL+C، ويمكن استخدام الربط لالتقاط نقرات الفأرة أو فقدان تركيز النافذة Focus -أي كونها نشطةً أو غير نشطة-، أو حتى كون النافذة مرئيةً أم لا، ويُرجَع في هذا إلى توثيق Tkinter لمزيد من المعلومات، وسيكون الجزء الأصعب هو معرفة صيغة وصف الحدث.

19.7 الرسالة القصيرة

من الممكن إبلاغ رسائل قصيرة للمستخدمين باستخدام MessageBox، وهذا سهل للغاية في Tk ويمكن تنفيذه باستخدام دوال وحدة `messagebox` كما يلي:

```
from tkinter import messagebox
messagebox.showinfo("Window Text", "A short message")
```

هناك أيضًا صناديق الخطأ والتحذير وصناديق نعم ولا Yes/No وموافق وإلغاء Ok/Cancel التي يمكن استخدامها من خلال دوال showXXX المختلفة، ويمكن تمييزها بأيقوناتها وأزرارها المختلفة، ويستخدم الصندوقان الأخيران askxxx بدلًا من showxxx، ويعيد قيمةً لتوضيح على أي زر ضغط المستخدم، كما يلي:

```
res = messagebox.askokcancel("Which?", "Ready to stop?")
print res
```

فيما يلي بعض الأمثلة لصناديق الرسائل في Tkinter:

وهذا شبيه بصناديق alert وMsgBox التي استخدمناها في برامج الويب من جافاسكربت وVBScript في دروسنا الأولى.

كما توجد صناديق حوارية قياسية يمكن استخدامها للحصول على أسماء الملفات أو المجلدات من المستخدم، تشبه صناديق "Open File" أو "Save File"، ولن نشرحها هنا لكن يمكن الاطلاع على أمثلة عنها في صفحات Tkinter المرجعية، تحت قسم Standard Dialogs.

19.8 تغليف التطبيقات مثل الكائنات

من الشائع في برمجة الواجهات الرسومية أن نغلف التطبيق كله مثل صنف واحد، وهذا يطرح سؤال كيف نلائم ودجات تطبيق Tkinter في هيكل هذا الصنف؟

لدينا خياران هنا، فإما أن نقرر جعل التطبيق نفسه صنفًا فرعيًا من إطار Tkinter، وإما أن نجعل أحد الحقول الأعضاء (الخاصيات) يخزن مرجعًا إلى نافذة المستوى الأعلى، ويُستخدم المنظور الثاني بكثرة في صناديق الأدوات الأخرى لذا سنتبعه، أما المنظور الأول فيمكن رؤيته في الفصل الثامن عشر: البرمجة المدفوعة بالأحداث، الذي يحوي أيضًا توضيحًا لاستخدام بسيط لودجت النص الخاصة بـ Tkinter، إضافةً إلى مثال آخر لاستخدام bind.

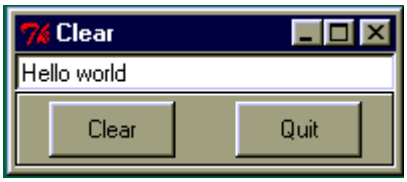
سنحول المثال أعلاه إلى هيكل كائني التوجه باستخدام حقل إدخال وزر Clear وزر Quit، لكننا سننشئ أولًا صنف تطبيق، ونجمّع الأجزاء المرئية للواجهة الرسومية داخل الباني Constructor، ثم نسند الإطار الناتج إلى self.mainWindow، سامحين بهذا للتوابع الأخرى للصنف بالوصول إلى إطار المستوى الأعلى، وتُسنَد الودجات الأخرى التي قد نحتاج إلى الوصول إليها -مثل حقل الإدخال- إلى متغيرات أعضاء member variables للتطبيق.

تصبح معالجات الأحداث توابعًا لصنف التطبيق باستخدام هذه التقنية، ويكون لها وصول إلى أي أعضاء بيانات data members لتطبيق -رغم عدم وجود أي منها في حالتنا هنا- من خلال مرجع self، مما يوفر تكاملًا سلسًا للواجهة الرسومية مع كائنات التطبيق الأساسية:

```
import tkinter as tk

# أنشئ معالج الحدث لمسح النص
class ClearApp:
    def __init__(self, parent):
        # create the top level window/frame
        self.mainWindow = tk.Frame(parent)
        self.eHello = tk.Entry(self.mainWindow)
        self.eHello.insert(0,"Hello world")
        self.eHello.pack(fill="x", padx=5, pady=5)
        self.eHello.bind("<Control-c>", self.evHotKey)

        # والآن أنشئ الإطار ذا الأزرار.
        fButtons = tk.Frame(self.mainWindow, height=2)
        self.bClear = tk.Button(fButtons, text="Clear",
                                width=10, height=1,command=self.evClear)
        self.bQuit  = tk.Button(fButtons, text="Quit",
                                width=10, height=1,
command=self.mainWindow.quit)
        self.bClear.pack(side="left", padx=15, pady=1)
        self.bQuit.pack(side="right", padx=15, pady=1)
        fButtons.pack(side="top", pady=2, fill="x")
        self.mainWindow.pack()
        self.mainWindow.master.title("Clear")

    def evClear(self):
        self.eHello.delete(0,tk.END)

    def evHotKey(self, event):
        self.evClear()

# والآن أنشئ التطبيق وشغل حلقة الحدث
top = tk.k()
```

```
app = ClearApp(top)
top.mainloop()
```

ستكون النتيجة كما يلي:



تبدو النتيجة مشابهةً للصورة السابقة، رغم أننا عدلنا بعض الإعدادات وخيارات التحزيم لتبدو أشبه بمثال wxPython أدناه.

لا شك أننا نستطيع إنشاء صنف مبني على إطار يحتوي مجموعةً قياسيةً من الأزرار، ونعيد استخدامه في بناء نوافذ حوارية مثلًا، فليس التطبيق الرئيسي وحده هو الذي يمكن تغليفه مثل كائن، بل يمكن إنشاء صناديق كاملة واستخدامها في مشاريع مختلفة، أو توسيع إمكانيات الودجات القياسية بإنشاء أصناف فرعية لها، لإنشاء زر يتغير لونه وفقًا لحالته مثلًا، وهو ما فعلناه في وحدة Tix التي ذكرناها أعلاه، وما هي إلا توسيع للصنف Tkinter.

يحتوي Tkinter بدءًا من الإصدار 3.1 على بعض المزايا الجديدة التي تُعرف باسم الودجات ذات السمات themed widgets، وتوجد في وحدة `tkinter.ttk`، وهي تحسن كثيرًا من مظهر Tkinter إلى حد يصعب التفريق بين نوافذه وبين ودجات النظام المضمّنة، لكننا لن نشرحها هنا، ويُرجَع فيها إلى موقع Tcl/Tk.

19.9 صندوق الأدوات البديل wxPython

توجد عدة صناديق أدوات أخرى للواجهات الرسومية، لعل أشهرها صندوق WxPython الذي يغلف ودجات صندوق أدوات C++، وهذا الصندوق -أي WxPython- أكثر شيوعًا من صندوق أدوات Tkinter عمومًا بين صناديق الواجهات المرئية، حيث يوفر وظائف قياسيةً افتراضيًا أكثر من Tk، مثل التلميحات tooltips، وشرائط الحالة status bars وغيرها، أما في Tk فيجب إنشاؤها يدويًا، وسنستخدم wxPython لإعادة إنشاء مثال Hello World أعلاه.

لكن المشكلة هنا هي أن wxPython ليس متاحًا بعد للإصدار الثالث من بايثون حتى الوقت الذي كتبنا فيه هذه الكلمات، مما يعني أننا سنعامل شيفرة الإصدار الثاني أدناه على أنها مجرد تدريب للقراءة لا غير، أو يمكن تثبيت الإصدار 2.7 من بايثون إذا أردنا التطبيق العملي، وتنزيل الحزمة من موقع wxPython، إذ لن ندخل كثيرًا في التفاصيل هنا، وعمومًا يعرِّف صندوق الأدوات إطار عمل يسمح لنا بإنشاء نوافذ وملئها بعناصر التحكم، وربط توابع بهذه المتحكمات، وبما أن هذا كائني التوجه فيجب استخدام التوابع وليس الدوال، ويبدو المثال كما يلي:

```
import wx

# --- عرّف إطارًا مخصصًا. سيكون هو النافذة الأساسية ---
class HelloFrame(wx.Frame):
    def __init__(self, parent, id, title, pos, size):
        super().__init__(parent, id, title, pos, size)
        # we need a panel to get the right background
        panel = wx.Panel(self)

        # أنشئ ودجتي النص والزر
        self.tHello = wx.TextCtrl(panel, -1, "Hello world", pos=(3,3),
size=(185,22))
        bClear = wx.Button(panel, -1, "Clear", pos=(15, 32))
        self.Bind(wx.EVT_BUTTON, self.OnClear, bClear)
        bQuit = wx.Button(panel, -1, "Quit", pos=(100, 32))
        self.Bind(wx.EVT_BUTTON, self.OnQuit, bQuit)

    # هذه معالجات الأحداث الخاصة بنا
    def OnClear(self, event):
        self.tHello.Clear()

    def OnQuit(self, event):
        self.Destroy()

# --- عرّف كائن التطبيق ---
# لاحظ أن كل برامج wxPython يجب أن تعرّف صنف تطبيق
# مشتق من wx.App
class HelloApp(wx.App):
    def OnInit(self):
        frame = HelloFrame(None, -1, "Hello", (200,50), (200,90) )
        frame.Show(True)
        self.SetTopWindow(frame)
        return True

# أنشئ نسخة وابدأ حلقة الحدث
HelloApp().MainLoop()
```

نلاحظ استخدام اصطلاح التسمية onXXXX للتوابع التي يستدعيها إطار العمل، واستخدام ثوابت EVT_XXX لربط الأحداث بالودجات، وتوجد مجموعة كبيرة منها.

يحتوي wxPython على ودجات كثيرة، أكثر من Tk، ويمكن بناء واجهات رسومية معقدة بها، لكنها للأسف تميل لاستخدام نظام موضعة مبني على الإحداثيات، وهذا متعب في العمل، لكن يمكننا استخدام نظام شبيه بمحزِّم Tk، مع أنه غير موثَّق جيدًا.

وقد يكون من المهم ملاحظة أن هذا المثال ومثال Tkinter الشبيه به أعلاه يحتويان على نفس عدد الأسطر البرمجية تقريبًا، إذ يحتوي مثال Tk على 23 سطرًا، ومثال wxPython على 21، فإذا رغبنا في واجهة رسومية سريعة لأداة نصية فسيكون Tk كافيًا، أما إذا أردنا بناء تطبيقات كاملة تعمل على عدة منصات تشغيل فينبغي استخدام wxPython.

ومن صناديق الأدوات الأخرى MFC وNET.، ولا ننسى Curses التي هي واجهة رسومية مبنية على نصوص، ويمكن تطبيق كثير من الدروس التي تعلمناها مع Tkinter على كل صناديق الأدوات هذه، لكن لكل منها مزاياه وعيوبه، فاختر واحدًا وتعرف عليه واستمتع ببرمجة الواجهات الرسومية.

أخيرًا نشير إلى أن العديد من صناديق الأدوات لها أدوات بناء رسومية، مثل Blackadder لـ Qt ، وGlade لـ GTK ، وكذلك يحتوي wxPython على بانٍ رسومي هو Boa Constructor رغم أنه لا زال في مرحلة الإصدار Alpha مما يعني أنه غير مستقر، كما يوجد بانٍ رسومي لصندوق Tk يسمى GUI Builder كان مخصصًا ابتداءً لبناء واجهات Tcl/Tk، لكنه يستطيع توليد شيفرات في عدة لغات بما فيها بايثون.

توجد عدة كتب أخرى لاستخدام Tcl/Tk وعدة كتب أخرى من بايثون لديها فصول عن Tk، وسنعود إليه في الفصل الثاني والعشرين: دراسة حالة برمجية، حين نشرح طريقة تغليف برنامج وضع باتش batch mode في واجهة رسومية لتحسين الاستخدام.

19.10 خاتمة

نرجو في نهاية هذا الفصل أن تكون تعلمت ما يلي:

- تُعرف عناصر تحكم الواجهات الرسومية بالودجات.
- تُجمَّع الودجات في هرمية احتوائية.
- توفر صناديق أدوات الواجهات الرسومية مجموعات مختلفةً من الودجات، رغم وجود مجموعة أساسية افتراضية فيها جميعًا.
- تسمح الإطارات بجمع الودجات المتشابهة، وتشكيل أساس لمكونات واجهة رسومية قابلة للاستخدام.
- ترتبط دوال معالجة الأحداث أو التوابع بالودجات من خلال ربط أسمائها بخاصية `command` الخاصة بالودجات.
- تستطيع البرمجة كائنية التوجه تبسيط برمجة الواجهات الرسومية كثيرًا بإنشاء كائنات تتوافق مع مجموعات الودجات، وتوابع تتوافق مع الأحداث.

هل تطمح لبيع منتجاتك الرقمية عبر الإنترنت؟

استثمر مهاراتك التقنية وأطلق منتجًا رقميًا
يحقق لك دخلًا عبر بيعه على متجر بيكاليكا

أطلق منتجك الآن

20. التعاودية Recursion

قد لا نحتاج إلى هذا الفصل في أغلب التطبيقات التي نكتبها، إذ إنه موضوع متقدم في البرمجة (خصوصًا في مراحل تعلم البرمجة الأولى)، وسنعرضه هنا لمجرد الدراسة واحتمال احتياجه في مشروع ما، ولا تقلق إذا وجدت أن عناصره صعبة الفهم.

سنشرح في هذا الفصل ما يلي:

- تعريف التعاودية، وكيفية عملها.
- كيف تساعد التعاودية في تبسيط بعض المشاكل الصعبة.

20.1 تعريف التعاودية

رغم قولنا سابقًا إن الحلقات التكرارية loops هي إحدى ركائز البرمجة، إلا أنه يمكننا كتابة برامج كاملة دون استخدام صريح لهذه البنية، بل إن بعض اللغات مثل Scheme لا تحوي بنية حلقة تكرارية صريحةً مثل `For` و`While` وغيرهما، وإنما تستخدم تقنيةً تسمى التعاودية، وقد تبين أن هذه التقنية قوية للغاية في حل بعض أنواع المشاكل، وهي تعني تطبيق دالة كجزء من تعريف نفس الدالة، ولننظر مثالًا على ذلك أحد الاختصارات التعاودية المشهورة في الكلمات، وهو أحد أساليب التلاعب بالاختصارات يحتوي الاختصار نفسه على كلمة تطابق حروف الاختصار، مثل مشروع GNU مثلًا -وهو أحد أهم المشاريع في البرمجيات مفتوحة المصدر- والذي تشير حروف كلمته إلى "نظام GNU ليس يونكس" أو GNU's Not UNIX، فتكون اختصارًا تعاوديًا لأن كلمة GNU التي في الاختصار هي نفسها كلمة GNU المختصرة كلها، أما معنى الكلمة الحرفي فهو حيوان الثور الإفريقي.

ويجب أن يوجد في الدالة شرط إنهاء، بحيث تتفرع الدالة إلى حل غير تعاودي عند نقطة ما، على عكس مثال GNU الذي ليس فيه هذا الشرط ويظل يتعاود إلى ما لا نهاية، وهو ما نطلق عليه الحلقة اللانهائية infinite loop.

لننظر هنا في مثال بسيط، تُعرَّف فيه دالة المضروب الرياضي factorial function المشار إليها بعلامة التعجب بعد العدد n!، على أنها ناتج ضرب جميع الأعداد من الواحد حتى العدد المطلوب -بما في ذلك العدد نفسه-، ومضروب الصفر هو الواحد، فإذا أردنا التعبير عن هذا المثال بطريقة أخرى فسنقول إن مضروب N يساوي N(N-1)، وعليه سيكون:

```
= 1
= 1 x 2 = 2
= 1 x 2 x 3 = 2! x 3 = 6
  N! = 1 x 2 x 3 x .... (N-2) x (N-1) x N = (N-1)! x N
```

ويمكن التعبير عن ذلك في بايثون كما يلي:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

يجب أن تنتهي الدالة بما أننا نقلل قيمة N في كل مرة ونتحقق هل تساوي 1 أم لا، لكن ثمة مشكلة بسيطة في هذا التعريف، إذ سيدخل في حلقة لا نهائية إذا استدعيناه برقم سالب، ولحل هذا نضيف اختبارًا للتحقق من أن n أقل من صفر، ويعيد None إذا كان كذلك لأن مضروب العدد السالب غير معرَّف undefined.

يظهر هذا مدى سهولة ارتكاب أخطاء في شروط الإنهاء، وهي أشهر حالة للعلل البرمجية bugs في الدوال التعاودية، إذ يجب أن نتأكد من اختبار جميع القيم حول حالة الانتهاء لضمان التنفيذ الصحيح.

لنرى الآن كيف يكون هذا عند تنفيذه، لاحظ أن تعليمة الإعادة تعيد:

```
n * (نتيجة استدعاء المضروب التالي)
```

فنحصل على ما يلي:

```
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
```

```
factorial(1) = 1
```

وتعود بايثون أدراجها لتستبدل القيم كما يلي:

```
factorial(2) = 2 * 1 = 2
factorial(3) = 3 * 2 = 6
factorial(4) = 4 * 6 = 24
```

ليس من الصعب كتابة دالة مضروب دون استخدام التعاودية، ويمكنك تجـريب هـذا، إذ يجب أن تمـر على جميع الأعداد حتى N؛ وتنفذ عمليات ضرب أثناء ذلك المرور التكراري، لكن قد يصعب كتابـة بعض الـدوال دون التعاودية، كما سنرى أدناه.

20.2 التعاودية على القوائم

إحدى الحالات التي تكون التعاودية مفيدةً فيها هي معالجة القوائم Lists، بشـرط أن نسـتطيع التحقـق من فراغ القائمة، وتوليد قائمة دون عنصرها الأول، ونفعل هذا في بايثون باستخدام تقنية تسمى التشـريح Slicing، لكن كل ما يجب معرفته في هذا الفصل هـو أن اسـتخدام فهـرس `[1:]` يعيـد جميـع العناصـر من العنصـر ذي الفهرس 1 حتى نهاية القائمة، لذا نكتب ما يلي لنصل إلى العنصر الأول من قائمة اسمها L:

```
first = L[0] # استخدم الفهرسة العادية
```

وللوصول إلى بقية القائمة:

```
# استخدم التشريح للوصول إلى العناصر 1،2،3 وما بعدها
butfirst = L[1:]
```

لنجرب ذلك في محث بايثون، لنتأكد أنه يعمل:

```
>>> L =[1,2,3,4,5]
>>> print( L[0] )
1
>>> print( L[1:] )
[2,3,4,5]
```

نعود الآن إلى استخدام التعاودية لطبع القوائم، ولنفرض حالةً نطبع فيها كل عنصر من قائمة سلاسل نصـية باستخدام الدالة `printList`:

```
def printList(L):
    if L:
```

```
        print( L[0] )
        printList(L[1:])
```

إذا تحققت L -أي كانت true ولم تكن فارغةً- فسنطبع العنصر الأول، ثم نعالج بقية القائمة كما يلي:

```
# شيفرة وهمية ليست بايثون
PrintList([1,2,3])
   prints [1,2,3][0] => 1
   runs printList([1,2,3][1:]) => printList([2,3])
   => we're now in printList([2,3])
         prints [2,3][0] => 2
         runs printList([2,3][1:]) => printList([3])
         => we are now in printList([3])
               prints [3][0] => 3
               runs printList([3][1:]) => printList([])
               => we are now in printList([])
                     "if L" is false for an empty list, so we return
None
         => we are back in printList([3])
               it reaches the end of the function and returns None
   => we are back in printList([2,3])
         it reaches the end of the function and returns None
=> we are back in printList([1,2,3])
   it reaches the end of the function and returns None
```

لاحظ أن الشرح أعلاه مأخوذ من شرح في نشرة تعليم بايثون البريدية بواسطة Zak Arnston بتاريخ يوليو 2003.

يسهل تنفيذ هذا الأمر لقائمة بسيطة باستخدام حلقة for، لكن ماذا لو كانت القائمة معقدةً وتحتوي قوائم أخرى فيها، فإذا استطعنا التحقق من كون عنصر ما قائمةً باستخدام دالة `type()` المضمنة وكان قائمة حقًا؛ فسنستدعي `printList()` تعاوديًا، أما إن لم يكن قائمةً فنطبعه:

```
def printList(L):
    # لا تفعل شيًا إن كانت فارغة
    if not L: return
    # إذا كانت قائمة فاستدع printList على العنصر الأول
    if type(L[0]) == list:
        printList(L[0])
```

```
    else: # لا توجد قوائم لذا نطبع هنا
        print( L[0] ) # L نعالج بقية عناصر
    printList( L[1:] )
```

سيصعب تنفيذ ذلك باستخدام الحلقة التكرارية العادية، ويظهر الفرق مع استخدام التعاودية في تسهيل ذلك التنفيذ، لكن ثمة مشكلة هنا، فالتعاودية على بنى البيانات الكبيرة يستهلك الذاكرة كثيرًا، لذا عند وجود ذاكرة صغيرة أو بنى بيانات كبيرة لمعالجتها، فيجب تفضيل الشيفرة المعتادة للأمان، وبسبب مشكلة الذاكرة تلك واحتمال أن يتعطل المفسر interpreter بسببها فقد وضعت بايثون حدًا لعدد مستويات التعاودية التي تسمح بها، فإذا تجاوزنا ذلك الحد فسيُنهى برنامجنا مع خطأ RecusrionError، والذي نلتقطه باستخدام try/except:

```
>>> def f(n): return f(n+1)
```

نلاحظ أن سبب هذه الحالة هو عدم وجود شرط إنهاء، لكن يجب أن تكون مجموعةً كبيرةً من بيانات الدخل كافيةً لإطلاقها حتى في الدوال المكتوبة بإتقان، وهنا يكون الحل الوحيد هو إعادة الكتابة مرةً أخرى باستخدام الحلقات التكرارية المعتادة، وهذا ممكن دومًا مهما بدا صعبًا.

20.3 جافاسكربت ولغة VBScript

تدعم كل من لغة جافاسكربت ولغة VBScript التعاودية، لكن بما أننا ذكرنا كل شيء تقريبًا فسنتركك مع نسخة تعاودية من دالة المضروب للغتين:

```
<script type="text/vbscript">
Function factorial(N)
   if N < 0 Then
      Factorial = -1   'a negative result is "impossible"
   if N = 0 Then
      Factorial = 1
   Else
      Factorial = N * Factorial(N-1)
   End If
End Function

Document.Write "7! = " & CStr(Factorial(7))
</script>

<script type="text/javascript">
```

```
function factorial(n){
   if (n < 0)
      return NaN  // NaN - Not a Number - يعني أنه غير صالح
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
};

document.write("6! = " + factorial(6));
</script>
```

لننظر الآن في البرمجة الدالية Functional Programming (أو البرمجة الوظيفية) في الفصل التالي.

20.4 خاتمة

نأمل بنهاية هذا الفصل أن تكون تعلمت ما يلي:

- تستدعي الدوال التعاودية نفسها من داخل تعريفها.
- يجب أن تحتوي الدوال التعاودية على شرط إنهاء غير تعاودي نصل إليه في النهاية، وإلا فسنقع في حلقة لا نهائية من التكرار.
- التعاودية مستهلكة للذاكرة عادةً، لكن ليس في كل الحالات.

21. البرمجية الوظيفية Functional Programming

سننظر في كيفية دعم بايثون لأسلوب آخر من أساليب البرمجة، ألا وهو البرمجية الوظيفية Functional Programming، واختصارًا FP، وهو موضوع متقدم بالنسبة للمبتدئين في البرمجة، كما ذكرنا في شأن التعاودية في الفصل السابق، وربما تصرف نظرك عنه الآن إلى أن تقطع شوطًا في البرمجة بنفسك.

يعتقد المؤيدون للبرمجة الوظيفية أنها الأسلوب الأمثل لتطوير البرمجيات.

سنغطي في هذا الفصل ما يلي:

- الفرق بين أسلوب البرمجة التقليدي والأسلوب الدالّي في البرمجة.
- الدول والتقنيات الخاصة بالبرمجية الوظيفية في بايثون.
- دوال لامدا Lambda Functions.
- تقييم الدارة المقصورة البولياني Short Circuit Boolean evaluation، والتعابير الشرطية.
- البرامج كتعابير.

21.1 ما هي البرمجية الوظيفية؟

ينبغي ألا نخلط بين البرمجية الوظيفية والأسلوب الإلزامي أو الإجرائي في البرمجة imperative style، وهو الذي كنا نستخدمه في أغلب الفصول حتى الآن، كما أنها تختلف عن البرمجة كائنية التوجه قليلًا بما أن المفاهيم التي سنراها هنا هي مفاهيم برمجية مألوفة لكننا نعبّر عنها تعبيرًا مختلفًا نوعًا ما، كما أن الفلسفة التي تقوم عليها البرمجية الوظيفية في حل المشاكل مختلفة عن باقي الأساليب أيضًا.

وتدور البرمجية الوظيفية حول التعابير، بل يمكن القول إن البرمجية الوظيفيـة هي البرمجـة تعبيريـة التوجـه expression oriented programming، لأن كل شيء فيها يؤول إلى تعبير في النهاية، وقد ذكرنا أن التعبير هو تجميعة من العمليات والمتغيرات التي ينتج عنها قيمة واحدة، فيكون `x == 5` تعبـيرًا بوليانيًـا boolean، و`(Y-7) + 5` تعبيرًا حسابيًا، و`"Hello world".uppercase()` تعبيرًا نصـيًا، وهـذا التعبـير الأخـير هـو اسـتدعاء دالـة function أيضًـا، أو اسـتدعاء تـابع method بـالأحرى، على كـائن السلسـلة النصـية `"Hello world"`، وسنرى أهمية الدوال في البرمجية الوظيفية، كما هو واضح من الاسم.

تُستخدم الدوال في البرمجية الوظيفية مثـل كائنـات، أي أنهـا تُمـرَّر من مكـان لآخـر داخـل البرنـامج بنفس طريقة تمرير المتغيرات، وقد رأينا أمثلةً على ذلك في بـرامج الواجهـة الرسـومية الـتي أنشـأناها من قبـل، حيث أسندنا اسم الدالة إلى سمة `command` الخاصة بمتحكم الزر، وعاملنا دالة معالج الحـدث على أنهـا كـائن وأسـندنا إلى الزر مرجعًا إلى الدالة، وهذا المفهوم الخاص بتمرير الدوال في البرامج أمـر أساسـي في البرمجيـة الوظيفيـة، كما تميل البرامج الوظيفية إلى أن تكون قائمية التوجه List Oriented.

تحاول البرمجية الوظيفية التركيز على ماهية المشاكل وليس كيفية حلها، أي أنها تصف المشكلة الـتي نريـد حلها، بدلًا من التركيز على آلية الحل نفسها، وتوجد عـدة لغـات برمجـة تميـل إلى التصـرف بهـذه الطريقـة، لعـل أوسعها انتشـارًا هي Haskell، ويحتـوي موقـع Haskell على أوراق عديـدة تصـف فلسـفة البرمجيـة الوظيفيـة، إضافةً إلى لغة Haskell نفسها، رغم أننا نرى أن مؤيدي هذا النمط من البرمجة يبالغون في ذلك الهدف.

ويهَيكل البرنامج الوظيفي بتعريف تعبير يلتقط الهدف من البرنامج، وكل شرط term في التعبير هو تعليمة لخاصية من خصائص المشكلة -وربما يوضع الشرط نفسه في تعبير آخر-، ونحصل على الحل من خلال تقييم كل شرط من هذه الشروط.

لكن هل هذا الأسلوب ناجح؟ الجواب: أحيانًا نعم وبكفاءة، لكننا للأسف نحتاج في كثير من المشاكل الأخـرى إلى أسلوب تفكير أكثر تجريدًا abstract، ويتأثر كثيرًا بالمفاهيم الرياضـية، وتصـعب قـراءة الشـيفرة الناتجـة من قبل المبرمج العادي، وتكون عادةً أقصر من الشيفرة الإلزامية المكافئة لها، وأكثر موثوقيةً منها، وقد دفعت هـذه المزايا الأخيرة -من الاختصار والموثوقية- الكثير من المبرمجين الذين يستخدمون الأسـلوب الكـائني أو الإلـزامي إلى النظر في البرمجية الوظيفية، إذ توجد العديد من الأدوات القوية التي يمكن اسـتخدامها، حـتى لـو لم ينتقـل المبرمج إلى اتباع هذا النمط كليًا.

21.1.1 موثوقية البرمجية الوظيفية

تأتي موثوقية البرامج الوظيفية من العلاقة الوطيدة بين البنى الخاصة بالبرمجية الوظيفية والمواصفات الاصطلاحية formal specification languages مثل: Z أو VDM، فإذا حُددت مشكلة في لغة رسمية فمن البديهي أن نترجم التحديد إلى لغة وظيفية التوجه مثل Haskell، لكن إذا كان التحديد خاطئًا فسيكون البرنامج الناتج مرآة لذلك الخطأ!

يُعرف هذا المبدأ في علوم الحاسوب باسم "Garbage In, Garbage Out" أو "قمامة داخلة، قمامة ناتجة"، ولا تزال هذه الصعوبة في التعبير عن متطلبات النظام بأسلوب موجز لا لبس فيه من أعظم التحديات في هندسة البرمجيات.

21.2 كيف تنفذ بايثون البرمجية الوظيفية

توفر بايثون دوالًا عديدة تمكننا من استخدام منظور البرمجية الوظيفية، وتمتلئ الدوال بالمزايا السهلة، أي يمكن كتابتها في بايثون بسهولة، أما ما يجب النظر إليه فهو الغرض المضمَّن في توفير تلك الدوال، وهو السماح لمبرمج بايثون بالعمل بأسلوب البرمجية الوظيفية إذا شاء.

سننظر الآن في بعض الدوال المتوفرة في بايثون ونرى كيف تعمل على بعض أمثلة هياكل البيانات التي نعرّفها على النحو التالي:

```
choices = ['eggs','chips','spam']
numbers = [1,2,3,4,5]

def spam(item):
    return "Spam & " + item
```

21.2.1 الدالة map(aFunction, aSequence)

تطبق الدالة الدالة `aFunction` الخاصة ببايثون على كل عضو من `aSequence`، ويكون التعبير كما يلي:

```
L = map(spam, choices)
print( list(L) )
```

ينتج عن هذا إعادة قائمة جديدة في L، مع السابقة `Spam &` في حالتنا قبل كل عنصر، ونلاحظ كيف مررنا الدالة `spam()` إلى دالة `map()` مثل قيمة، أي أننا لم نستخدم الأقواس لتنفيذ شيفرة الدالة، بل استخدمنا اسمها مرجعًا إلى الدالة، وهذه الخاصية في معاملة الدوال مثل قيم هي إحدى المزايا الرئيسية في البرمجية الوظيفية.

يمكن تحقيق نفس النتيجة بكتابة ما يلي:

```
L = []
for i in choices:
   L.append( spam(i) )
print( L )
```

لكن نلاحظ أن دالة map تسمح لنا بإلغاء الحاجة إلى كتلة شيفرة متشعبة، مما يقلل تعقيد البرنامج، وسنرى أن هذه سمة متكررة في البرمجية الوظيفية، حيث يقلل استخدام الدوال التعقيد النسبي للشيفرة بالتخلص من الكتل البرمجية.

21.2.2 الدالة filter(aFunction, aSequence)

تستخلص filter كل عنصر في التسلسل aSequence تعيد له الدالة aFunction القيمة True، وإذا عدنا إلى قائمة الأعداد الخاصة بنا فيمكن أن ننشئ قائمةً جديدةً من الأعداد الفردية فقط:

```
def isOdd(n): return (n%2 != 0) # mod استخدم العامل
L = filter(isOdd, numbers)
print( list(L) )
```

نلاحظ مرةً أخرى أننا نمرر اسم الدالة isodd إلى filter قيمة وسيط، بدلًا من استدعاء isodd() مثل دالة، وعلى أي حال نستطيع كتابة الأسلوب البديل التالي:

```
def isOdd(n): return (n%2 != 0)
L = []
for i in numbers:
   if isOdd(i):
      L.append(i)
print( L )
```

ونلاحظ هنا أيضًا أن الأسلوب التقليدي يحتاج إلى مستويين إضافيين من الإزاحات لتحقيق نفس النتيجة، وهذه الزيادة في مستويات الإزاحة دليل على زيادة التعقيد.

توجد عدة أدوات أخرى للبرمجة الوظيفية في وحدة اسمها functools يمكن استيرادها وتصفحها في محث بايثون، ويُرجع إلى dir() وhelp() عند الحاجة.

21.2.3 الدالة لامدا Lambda

إحدى الخصائص الواضحة في الأمثلة السابقة هو أن الدوال التي مُرِّرت إلى دوال البرمجية الوظيفية كانت قصيرة جدًا، وغالبًا ما كانت سطرًا واحدًا فقط، وتوفر بايثون دعمًا جديدًا للبرمجة الوظيفية لتوفير الجهد المبذول

لتعريف هذه الدوال الصغيرة، وهي دالة لامدا `lambda`، والتي يأتي اسمها من فر ع في الرياضيات هو حسابات لامدا Lambda Calculus، الذي يستخدم حرف لامدا الإغريقي λ لتمثيل مفهوم قريب من هذا.

ويُستخدم مصطلح لامدا في البرمجية الوظيفية للإشارة إلى دالة مجهولة تمثل كتلةً برمجيةً يمكن تنفيذها كما لو كانت دالةً لكن دون اسم، ويمكن تعريف دوال لامدا في أي مكان داخل البرنامج يمكن أن يحدث فيه تعبير بايثون، وهذا يعني أننا نستطيع استخدامها داخل دوال البرمجية الوظيفية الخاصة بنا.

وتبدو الدالة لامدا بالشكل التالي:

```
lambda <aParameterList> : <a Python expression using the parameters>
```

وعلى ذلك يمكن كتابة دالة `isodd` سالفة الذكر كما يلي:

```
isOdd = lambda j: j%2 != 0
```

ونتجنب هنا تعريف السطر بالكامل من خلال إنشاء الدالة لامدا داخل الاستدعاء على `filter` كما يلي:

```
L = filter(lambda j: j%2 != 0, numbers)
print( list(L) )
```

ويُنفَّذ الاستدعاء إلى map باستخدام ما يلي:

```
L = map(lambda s: "Spam & " + s, choices)
print( list(L) )
```

ونلاحظ هنا أننا كنا نحول نتائج `map()` و`filter()` إلى قوائم، لأنهما صنفان يعيدان نسخًا من شيء يدعى المتكرر itreable، وهو يتصرف مثل التسلسل أو التجميعة إذا استُخدم على حلقة تكرارية، ويمكن تحويله إلى قائمة، لكن كفاءته تظهر في استخدام الذاكرة، وصديقتنا القديمة `range()` قابلة للتكرار كذلك، وتسمح بايثون بإنشاء أنواع قابلة للتكرار خاصة بنا، لكننا لن نشرحها.

21.2.4 استيعاب القوائم

استيعاب القوائم list comprehension هي تقنية لبناء قوائم جديدة، مستوردة من لغة Haskell وأُدخلت إلى بايثون في الإصدار الثاني، وبنيتها غريبة قليلًا وتشبه الصياغة الرياضية، مثلًا:

```
[<expression> for <value> in <collection> if <condition>]
```

والتي تكافئ ما يلي:

```
L = []
for value in collection:
```

```
    if condition:
        L.append(expression)
```

مما يوفر علينا كتابة بعض الأسطر كما في بقية البنى في البرمجية الوظيفية، ومستويين من الإزاحة كـذلك، لننظر في بعض الأمثلة العملية، حيث سننشئ أولًا قائمةً من جميع الأعداد الزوجية الأصغر من 10:

```
>>> [n for n in range(10) if n % 2 == 0 ]
[0, 2, 4, 6, 8]
```

والذي يقول إننا نريد قائمةً من القيم n التي تُختار من المجال 0-9، وتكون زوجيةً (`n % 2 == 0`)، ويمكن استبدال دالةٍ بالشرط الأخير لا شك، شرط أن تعيد الدالة قيمةً تستطيع بايثون أن تفسرها مثـل قيمـة بوليانيـة، وعليه يمكن إعادة كتابة المثال السابق كما يلي:

```
>>>def isEven(n): return ((n%2) == 0)
>>> [ n for n in range(10) if isEven(n) ]
[0, 2, 4, 6, 8]
```

والآن لننشئ قائمةً من تربيعات أول 5 أعداد:

```
>>> [n*n for n in range(5)]
[0, 1, 4, 9, 16]
```

نلاحظ أن تعليمة if الأخيرة لم تكن ضروريةً لكل حالة، فالتعبير الابتدائي هنا هو n*n، حيث نسـتخدم جميـع قيم المجال، لنستخدم الآن تجميعةً موجودةً مسبقًا بدلًا من دالة المجال:

```
>>> values = [1, 13, 25, 7]
>>> [x for x in values if x < 10]
[1, 7]
```

يمكن استخدام ذلك لاستبدال دالة المرشح التالية:

```
>>> print( list(filter(lambda x: x < 10, values)) )
[1, 7]
```

لا يقتصـر اسـتيعاب القـوائم على متغـير واحـد أو اختبـار واحـد، لكن سـيزداد تعقيـد الشـيفرة كلمـا زادت المتغيرات والاختبارات، ويعود إليك الاختيار بين استيعاب القوائم أو الدوال التقليدية بحسب ما تـراه أسـهل، إذ تستطيع استخدام دوال البرمجية الوظيفية السابقة أو اسـتيعابات القـوائم الجديـدة عنـد إنشـاء تجميعـة جديـدة مبنية على واحدة موجودة مسبقًا، لكن الأسهل في إنشاء التجميعات الجديدة هو الاستيعاب.

ورغم أن هذه البُنى تبدو مغريةً إلا أن التعابير اللازمة للحصول على النتيجة التي نريدها قد تصبح معقدةً للغاية بحيث يسهل توسيعها إلى مكافئاتها التقليدية في بايثون، ولا عيب في هذا إذ إن سهولة القراءة أفضل من غموض الشيفرة، خاصةً إذا كان ذلك الغموض لمجرد التذاكي.

21.3 بنى أخرى

رغم أن هذه الدوال مفيدة في ذاتها إلا أنها لا تكفي للسماح بنمط برمجة وظيفية كامل داخل بايثون، إذ يجب تغيير هياكل التحكم أو على الأقل استبدالها بمنظور وظيفية، ويمكن تنفيذ هذا بتطبيق أثر جانبي لكيفية تقييم بايثون للتعابير البوليانية.

21.3.1 التقييم المقصور short-circuit evaluation

لعلك تذكر من الفصل العاشر: مقدمة في البرمجة الشرطية أن بايثون تستخدم التقييم المقصور short-circuit evaluation للتعابير البوليانية، ويمكن استغلال بعض خصائص هذه التعابير في توفير أسلوب وظيفية للتحكم في البرامج، والتقييم المقصور (أو القصير) باختصار هو بدء تقييم التعبير البولياني من التعبير الأيسر إلى الأيمن، ويتوقف التقييم عند عدم الحاجة إلى تقييم أكثر لتحديد النتيجة النهائية، لننظر في بعض الأمثلة لنرى كيف يعمل هذا التقييم:

```
>>> def TRUE():
...     print( 'TRUE' )
...     return True
...
>>> def FALSE():
...     print( 'FALSE' )
...     return False
...
```

نعرِّف أولًا دالتين تخبراننا متى تُنفَّذان وتعيدان قيمة أسمائهما، ونستخدم ذلك لنرى كيفية تقييم التعابير البوليانية، لاحظ أن الخرج بالأحرف الكبيرة ناتج عن الدوال، والخرج بالأحرف مختلطة الحالة ناتج عن التعبير:

```
>>> print( TRUE() and FALSE() )
TRUE
FALSE
False
>>> print( TRUE() and TRUE() )
TRUE
TRUE
```

```
True
>>> print( FALSE() and TRUE() )
FALSE
False
>>> print( TRUE() or FALSE() )
TRUE
True
>>> print( FALSE() or TRUE() )
FALSE
TRUE
True
>>> print( FALSE() or FALSE() )
FALSE
FALSE
False
```

نلاحظ أنه إذا تحقق الجزء الأول من تعبير AND -أي كان `True`- وفقط إذا تحقق؛ فسيقيَّم الجزء الثاني، أمـا إذا لم يتحقق الجزء الأول -كان `False`- فلن يُقيَّم الجزء الثاني بما أن التعبير ككل لا يمكن أن يتحقق.

وبالمثل ففي التعبير المبني على OR، إذا كان الجزء الأول `True` فلا توجد حاجة إلى تقييم الجزء الثـاني، لأن التعبير الكلي يجب أن يكون `True`، وذاك يتحقق بأحد الجزئين فقط.

هناك ميزة أخرى في تقييم بايثون للتعابير البوليانية يمكن استغلالها، وهي أنها لا تعيد -عند تقييم تعبير ما- `True` أو `False` فقط، بل تعيد القيمة الحقيقية للتعبير، لذا ستعيد بايثون السلسلة النصـية نفسـها إذا تحققنـا من سلسلة فارغة -والتي يجب أن تُعد `False`- كما يلي:

```
if "This string is not empty": print( "Not Empty" )
else: print( "No string there" )
```

يمكن استخدام هذه الخصائص لإعادة إنتاج سلوك شبيه بالتفريع branching، لنفترض مثلًا أن لدينا جـزءًا من شيفرة كما يلي:

```
if TRUE(): print( "It is True" )
else: print( "It is False" )
```

يمكن استبدال بنية وظيفية دالية بها:

```
V =  (TRUE() and "It is True") or ("It is False")
```

```
print( V )
```

جرب العمل على هذا المثال واستبدل استدعاءً إلى FALSE() بالاستدعاء إلى TRUE().

وهكـذا نكـون قـد وجـدنا طريقـةً للتخلص من تعليمـات if/else الشـرطية في برامجنـا باسـتخدام التقـييم المقصور للتعابير البوليانية، وقد ترى هذه الأساليب في البرامج القديمة، لكنها قد تأتي بنتائج عكسية، لهذا توجد بنية أُدخلت حديثًا إلى بايثون تسمى بالتعبير الشرطي تسمح لنا بكتابة شرط if/else مثل تعبير، كما يلي:

```
result = <True expression> if <test condition> else <False expression>
```

وسيبدو مثال حقيقي بها كما يلي:

```
>>> print( "This is True" if TRUE() else "This is not printed" )
TRUE
This is True
```

وإذا استخدمنا else:

```
>>> print( "This is True" if FALSE() else "We see it this time" )
FALSE
We see it this time
```

وقد ذكرنـا في الفصـل العشـرين: مفهـوم التعاودية، أنـه يمكن اسـتخدام التعاوديـة لاسـتبدال بنيـة الحلقـة التكرارية، فإذا جمعنا التعاودية مع التعابير الشرطية فيمكننا التخلص من بنى التحكم القديمة كلها من برنامجنـا، ونستبدل بها تعابير صرفةً، وهذه خطوة كبيرة نحو تفعيل حلول البرمجية الوظيفية.

ولنضـع هـذا في تـدريب عملي سـنكتب برنـامج المضـروب factorial بأسـلوب وظيفيـة كليًـا، باسـتخدام lambda بدلًا من def، والتعاودية بدلًا من الحلقات التكرارية، والتعابير الشرطية بدلًا من if/else المعتادة:

```
>>> factorial = lambda n: ( None if n < 0 else
if (n == 0) else
                            factorial(n-1) * n )
>>> print( factorial(5) )
120
```

وهذا كل ما في الأمر، وقد لا تكون هذه الشيفرة سهلة القراءة مثل شيفرة بايثون العادية لكنهـا تعمـل، وهي دالة بنمط البرمجية الوظيفية كليًا لأنها تعبير خالص، ونلاحظ أننا استخدمنا مجموعةً من الأقـواس حـول التعبـير كله، وهو ما يسمح لنا بتوزيعه على عدة أسطر لتحسين قراءته، ثم حاذينا قيم الإعادة المحتملة الثلاثـة رأسـيًا في أسطر منفصلة، وهذا اصطلاح شائع مستورد من لغات Lisp التي تميل إلى استخدام التعاودية بكثرة.

21.4 استنتاجات

لعل السؤال الذي يطرح نفسه الآن هو الجدوى من كل ذلك، فرغم أن البرمجية الوظيفية تروق لكثير من الأكاديميين في مجال علوم الحاسوب وعلماء الرياضيات كذلك؛ إلا أن أغلب المبرمجين العاملين يقتصدون في استخدام تقنيات البرمجية الوظيفية، ويدمجونها مع الأساليب الإلزامية التقليدية وفق ما يرونه مناسبًا.

وعند الحاجة إلى تطبيق عمليات على عناصر في قائمة مثل `map` أو `filter` تكون البرمجية الوظيفية هي المثلى للتعبير عن الحل، وبالمثل قد نجد أحيانًا أن التعاودية أفضل من الحلقات التكرارية، كما قد نجد استخدامًا للتقييم المقصور أو التعبير الشرطي بدلًا من تعابير `if/else` الشرطية، خاصةً داخل تعبير ما، والمهم أنه يجب على المتعلم ألا يتحمس كثيرًا لتقنية أو فلسفة برمجية بعينها، وإنما يستخدم الأداة المناسبة للمهمة التي بين يديه، ويكفيه حينئذ أن يعلم بوجود حلول بديلة.

لدينا أمر أخير حول الدالة `lambda`، إذ يمكن استخدامها خارج مجال البرمجية الوظيفية، وهي تعريف معالجات الأحداث في برمجة الواجهات الرسومية، حيث تكون معالجات الأحداث دوالًا قصيرةً في الغالب، أو تستدعي دوالًا أكبر منها بقيم وسائط مدمجة فيها، وفي كلا الحالتين يمكن استخدام دالة لامدا مثل معالج حدث يتجنب الحاجة إلى تعريف الكثير من الدوال المنفردة وشغل فضاء الاسم بأسماء لن تُستخدم إلا مرةً واحدةً فقط، ويجب تذكر أن تعليمة لامدا تعيد كائن دالة يُمرَّر إلى الودجت widget ويُستدعى في وقت وقوع الحدث، وقد عرَّفنا ودجت الزر في Tkinter من قبل، لذا ستبدو لامدا كما يلي:

```
b = Button(parent, text="Press Me",
           command = lambda : print("I got pressed!"))
b.pack()
```

نلاحظ أنه رغم عدم السماح لمعالج الحدث بامتلاك معامِل إلا أننا نسمح بتحديد سلسلة داخل متن لامدا، ونستطيع الآن إضافة زر ثانٍ بسلسلة مختلفة كليًا:

```
b2 = Button(parent, text="Or Me",
           command = lambda : print("I got pressed too!"))
b2.pack()
```

كما نستطيع توظيف `lambda` عند استخدام تقنية الربط bind technique التي ترسل كائن حدث مثل وسيط:

```
b3 = Button(parent, text="Press me as well")
b3.bind(<Button-1>, lambda ev : write("Pressed"))
```

21.5 البرمجية الوظيفية في جافاسكربت

لا نريد الخوض في تفاصيل البرمجية الوظيفية هنا، لكن من المهم أن ندرك أن جافاسكربت متأثرة كثيرًا بمفاهيم البرمجية الوظيفية، بل إن الاستخدام الحديث لها يشجع استخدام البرمجية الوظيفية أكثر من البرمجة الكائنية، ومفتاح البرمجية الوظيفية فيها هو أن إمكانية كتابة تعريفات الدوال بأسلوب مشابه لتعابير لامدا، فلا تتطلب إلا تغييرًا بسيطًا في الأسلوب والبنية اللغوية لتعريف الدالة، انظر هذا المثال من الفصل الحادي عشر: البرمجة باستخدام الوحدات:

```
<script type="text/javascript">
var i, values;

function times(m) {
    var results = new Array();
    for (i = 1; i <= 12; i++) {
        results[i] = i * m;
        }
    return results;
}

// استخدم الدالة
values = times(8);

for (i=1;i<=12;i++){
    document.write(values[i] + "<br />");
    }
</script>
```

لاحظ أننا عرّفنا الدالة بوضع الاسم `times` بعد الكلمة المفتاحية `function`، لكن جافاسكربت تسمح لنا بتسمية الدالة من خلال الإسناد، كما في أسلوب لامدا الخاص بايثون أعلاه:

```
times = function(m) {
    var results = new Array();
    for (i = 1; i <= 12; i++) {
        results[i] = i * m;
        }
    return results;
```

```
}
```

لذا يكون `times` الآن متغيرًا يحمل مرجعًا إلى كائن الدالة، ويمكن استدعاؤه كما فعلنا من قبل:

```
values = times(8);

for (i=1;i<=12;i++){
    document.write(values[i] + "<br />");
    }
```

بل يمكن استخدام `function()` لإنشاء دوال مجهولة كما في لامدا، باستثناء أنه ليس لجافاسكربت قيود على أنواع الدوال التي ننشئها، فيمكن أن تكون بأي طول وتعقيد نريده، مما يؤدي إلى نمط في برمجة جافاسكربت ستقابله غالبًا في صفحات الويب، ولذا سنسرد مثالًا مختصرًا له هنا، وهو يتضمن استخدام دوال جافاسكربت التي تأخذ دوالًا أخرى معامِلات لها، وقد رأينا هذا في برامج الواجهة الرسومية من قبل وسميناه رد النداء callback، لأن العديد من مكتبات جافاسكربت تستخدم رد النداء ذاك، ويكون عادةً من دالة رد نداء تُحدَّد على أنها المعامِل الأخير، وهنا مثال من رد نداء مُستخدَم في صفحة ويب بسيطة، وهي مثال كامل، لذا يمكن تحميله وتجربته في المتصفح:

```
<html>
<head>
<title>Callback test</title>
<script type="text/javascript">
window.setTimeout( function() {
       document.write("رأيتني الآن!");
     }, 3000);
</script>
</head>

<body>
<p>انتظر نهاية الوقت....<p>

</body>
</html>
```

نلاحظ أن الدالة التي نفذها الوقت المستقطع لي لها اسم، فقد أُنشئت وسيطًا أول في استدعاء `setTimeout` نفسه، ثم أضيف زمن الانتظار الذي هو 3000 مللي ثانية وسيطًا ثانيًا، فإذا حملته في المتصفح لديك فسترى وقت الانتظار يحل محل الرسالة الأولى بعد ثلاث ثوانٍ.

صار هذا الأسلوب من تعريف الدوال المضمَّنة شائعًا للغاية في مجتمع جافاسكربت، وما هو إلا برمجة وظيفية خالصة، وتحتوي jQuery وهي إحدى أشهر مكتبات الويب لجافاسكربت، على دوال كثيرة تأخذ دوالًا أخرى مثل معامِلات، لنحصل على أسلوب برمجي وظيفي للغاية.

أما VBScript فليس فيها دعم مباشر للبرمجة الوظيفية، لكن يمكن استخدامها بأسلوب وظيفي إذا كان لدى المبرمج صبر على ذلك، بهيكلة البرامج مثل التعابير وعدم السماح للآثار الجانبية بتعديل متغيرات البرنامج.

21.5.1 مصادر أخرى للاستزادة

توجد مصادر أخرى يمكن الاطلاع عليها للتعمق في البرمجية الوظيفية، منها:

- مقالة David Mertz على موقع IBM حول البرمجية الوظيفية في بايثون، وهو يذكر بتفصيل هياكل التحكم ويعطي أمثلةً مفصلةً على ذلك.
- تدعم بعض اللغات الأخرى البرمجية الوظيفية أفضل من بايثون، مثل Lisp و Scheme و Haskell وML وغيرها، ويحتوي موقع Haskell على معلومات كثيرة عن البرمجية الوظيفية.
- توجد مجموعة بريدية هي `comp.lang.functional` يمكن من خلالها الاطلاع على آخر المستجدات والإجابات حول البرمجية الوظيفية.
- توجد كتب عديدة مذكورة في الروابط السابقة، وأحد الكتب الكلاسيكية كتاب Structure & Interpretation of Computer Programs، وهو كتاب يتعلق بالبرمجية الوظيفية بالكامل لكنه يغطي المبادئ جيدًا، ويركز على لغة Scheme التي هي نسخة من Lisp يفضلها الكثير من الأكاديميين، وقد كان المرجع الذي استقينا منه هذا الشرح هو كتاب The Haskell School of Expression.

21.6 خاتمة

نرجو في نهاية هذا الفصل أن تكون تعلمت ما يلي:

- البرامج الوظيفية ما هي إلا تعابير خالصة.
- توفر بايثون `map` و `filter` و `reduce` و list comprehensions لدعم أسلوب البرمجية الوظيفية.
- تعابير لامدا هي كتل من الشيفرات المجهولة التي لا تحمل اسمًا، ويمكن إسنادها إلى متغيرات أو استخدامها مثل دوال.
- تقيِّم التعابير البوليانية عند الحاجة فقط لضمان النتيجة التي تمكن تلك التعابير من استخدامها مثل هياكل تحكم.

- عند جمع مزايا البرمجية الوظيفية لبايثون مع التعاودية فمن الممكن كتابة أي دالة بأسـلوب وظيفي في بايثون، رغم أننا لا ننصح بهذا.
- تدعم جافاسكربت البرمجية الوظيفية، وهو الموجود الآن في أغلب صفحات الويب الحديثة.

22. دراسة حالة برمجية: تصميم برنامج لحساب عدد الكلمات

سنتطرق في هذا الفصل إلى برنامج عدّ الكلمات الذي كتبناه من قبل، وسننشئ برنامجًا يحاكي برنامج wc في يونكس، من حيث حساب عدد الأسطر والكلمات والمحارف التي في ملف ما، ثم نتعمق أكثر في هذا لنخرج عدد الجمل وأجزاء الجمل clauses والفقرات أيضًا، وسنتابع تطوير هذا البرنامج مرحلةً تلو الأخرى، وسنزيد من إمكانياته تدريجيًا، ثم ننقله إلى وحدة module ليكون قابلًا لإعادة الاستخدام بأن نجعله كائني التوجه object oriented لتحقيق أقصى حد ممكن من الإمكانيات، ثم نغلفه أخيرًا في واجهة رسومية لسهولة الاستخدام، ورغم أننا سنستخدم بايثون في هذا البرنامج إلا أنه يمكن كتابة نسخ منه باستخدام جافاسكربت أو VBScript بإجراء بعض التعديلات.

ويمكن إضافة مزايا أخرى لهذا البرنامج، لكننا سنتركها للقارئ للتدريب، من تلك المزايا:

- حساب فهرس FOG للنصوص، والذي يوضح مدى تعقيد النص، ويُعرَّف بأنه:

```
0.4 * (نسبة الكلمات الأكثر من 5 أحرف) + (متوسط عدد الكلمات للجملة الواحدة)
```

- حساب عدد الكلمات الفريدة المستخدمة، ومرات تكرارها.
- إنشاء نسخة جديدة تحلل ملفات RTF.

22.1 حساب عدد الأسطر والكلمات والحروف

إذا نظرنا إلى عداد الكلمات السابق:

```
import string
def numwords(s):
    lst = string.split(s)
    return len(lst)

with open("menu.txt","r") as inp:
   total = 0
   # accumulate totals for each line
   for line in inp.readlines():
      total += numwords(line)

print( "File had %d words" % total )
```

فسنجد أننا بحاجة إلى إضافة عدد الأسطر والأحرف، وعدد الأسطر سهل لأننا نكرر على كل سطر، لذا سنحتاج إلى متغير يتزايد مع كل تكرار على الحلقة التكرارية، أما عدد الأحرف فسيكون أصعب قليلًا، لأننا نستطيع التكرار على قائمة الكلمات مضيفين أطوالها في متغير آخر.

كما ينبغي أن نجعل البرنامج عام الأغراض بقراءة اسم الملف من سطر الأوامر، أو نطلب من المستخدم أن يزودنا بالاسم إذا لم يكن متاحًا، كما يمكن قراءته من مجرى الدخل القياسي standard input، وهو ما يفعله برنامج wc.

سيبدو `wc.py` النهائي كما يلي:

```
import sys, string

# احصل على اسم الملف من سطر الأوامر أو المستخدم
if len(sys.argv) != 2:
   name = input("Enter the file name: ")
else:
   name = sys.argv[1]

# اضبط العدادات على الصفر. هذا ينشئ المتغيرات
# words - lines - chars = 0, 0, 0
```

```
with open(name,"r") as inp:
   for line in inp:
      lines += 1
      # Break into a list of words and count them
      lst = line.split()
      words += len(lst)
      chars += len(line) # Use original line which includes spaces
etc.

print( "%s has %d lines, %d words and %d characters" % (name, lines,
words, chars) )
```

يستطيع من يستخدم برنامج wc في يونكس تمرير اسم ملف بمحارف بديلة wildcards؛ للحصول على إحصائيات لجميع الملفات المطابقة إضافةً إلى المجموع الكلي، أما برنامجنا فيتعامل مع اسم ملف واحد فقط، فإذا أردت توسيع ذلك إلى محارف البدل فألق نظرةً على الوحدة glob، وابن قائمةً من الأسماء، ثم كرر على قائمة الملفات، وستحتاج إلى عدادات مؤقتة لكل ملف، ثم عدادات تراكمية للمجموع الكلي، أو استخدم قاموسًا.

22.2 عد الجمل بدلًا من الأسطر

إذا أردنا توسيع هذا البرنامج ليشمل عد الجمل والكلمات بدلًا من "مجموعات المحارف"، فنكرر أولًا على الملف لاستخراج الأسطر إلى قائمة، ثم نكرر على كل سطر لنستخرج الكلمات إلى قائمة أخرى، ثم نعالج بعد ذلك كل "كلمة" لحذف المحارف الدخيلة عليها.

لكن توجد طريقة أبسط، بأن نجمع السطور ونحلل محارف الترقيم لعد الجمل وأجزاء الجمل وغيرها، من خلال تعريف الجملة أو جزء الجملة بحسب عناصر الترقيم، لنجرب ذلك في شيفرة وهمية:

```
لكل سطر في الملف:
    زِد عدد الأسطر بمقدار واحد
    إذا كان السطر فارغًا:
        زِد عدد الفقرات
    عد نهايات أجزاء الفقرات
    عد نهايات الجمل.

أنشئ تقريرًا بالفقرات والأسطر والجمل وأجزاء الجمل والمجموعات والكلمات.
```

لنحول الآن الشيفرة الوهمية إلى شيفرة حقيقية، وسنستخدم التعابير النمطية في حلنا، لذا ينبغي مراجعة الفصل السادس عشر: التعابير النمطية في البرمجة، للاطلاع عليها ودراستها:

```
import re,sys

# استخدم التعابير النمطية للعثور على الأجزاء
sentenceStops = ".?!"
clauseStops = sentenceStops + ",;:\-" # escape '-' to avoid range
effect
sentenceRE = re.compile("[%s]" % sentenceStops)
clauseRE = re.compile("[%s]" % clauseStops)

# احصل على اسم الملف من سطر الأوامر أو من المستخدم
if len(sys.argv) != 2:
   name = input("Enter the file name: ")
else:
   name = sys.argv[1]

# اضبط العدادات الآن
lines, words, chars = 0, 0, 0
sentences,clauses = 0, 0
paras = 1  # assume always at least 1 para

# عالج الملف
with open(name,"r") as inp:
   for line in inp:
      lines += 1
      if line.strip() == "": # empty line
         paras += 1
      words += len(line.split())
      chars += len(line.strip())
      sentences += len(sentenceRE.findall(line))
      clauses += len(clauseRE.findall(line))

# اعرض النتائج
output = '''
The file %s contains:
        %d\t characters
        %d\t words
        %d\t lines in
```

```
        %d\t paragraphs with
        %d\t sentences and
        %d\t clauses.
''' % (name, chars, words, lines, paras, sentences, clauses)
print( output )
```

هناك عدة نقاط ينبغي الانتباه إليها في الشيفرة السابقة:

- نستخدم التعابير النمطية هنا لتحسين كفاءة عمليات البحث، رغم إمكانية استخدام عمليات بحث بسيطة عن السلاسل النصية، لكن كنا سنحتاج حينئذ إلى البحث عن كل محرف ترقيم على حدة، وتزيد التعابير النمطية هنا من كفاءة البرنامج، من خلال إيجاد جميع العناصر التي نريدها في بحث واحد، لكن من ناحية أخرى يسهل حدوث الأخطاء فيها، فقد نسينا أثناء محاولتنا الأولى لكتابة الشيفرة أن نهرب المحرف - ، فرآه التعبير النمطي مجالًا range، لذا عومل أي عدد في الملف على أنه فاصل لجزء من جملة، واستغرق الأمر كثيرًا حتى عرضنا المشكلة على مجتمع بايثون للعثور على الخطأ، فلما أضفنا محرف " حُلت المشكلة، كما صرَّفنا التعبيرات مسبقًا precompiled، مما حسّن من الأداء أكثر من استخدام نفس التعبير عدة مرات، كما نلاحظ استخدام التابع `findall` للحصول على جميع التطابقات في سطر باستدعاء واحد.
- تظهر كفاءة هذا البرنامج في أنه ينفذ ما نريده بالضبط، وإن كان أقل كفاءةً من منظور إمكانية إعادة الاستخدام، لعدم وجود دوال يمكن استدعاؤها من برامج أخرى.
- ليست اختبارات الجمل مثاليةً، لأن العناوين المختصرة -مثل "Mr."- ستُحسب جملةً، لأنها تحتوي على محرف النقطة، ونستطيع تحسين التعبير النمطي ليبحث عن النقطة متبوعةً بمسافة واحدة أو أكثر، ثم متبوعةً بحرف إنجليزي كبير، لكن ذلك لن يحل مشكلة "Mr." لأنها تُتبع عادةً بمسافة ثم كلمة بحرف كبير، وهذا يوضح مدى صعوبة معالجة اللغات الطبيعية بكفاءة، فإذا احتجنا حقًا إلى مثل هذا البحث في الممارسة العملية فهناك وحدات متاحة على الإنترنت مصممة خصيصًا لذلك.

سنتناول النقطة الثانية المتعلقة بإمكانية إعادة الاستخدام أثناء دراسة الحالة التي بين أيدينا، وننظر في المشاكل المتعلقة بتحليل النصوص بتفصيل أكثر، رغم أننا لن ننتج محلل نصوص مثاليًا في النهاية، حيث تحتاج هذه المهمة مهارات أكبر من التي نتوقعها من مبرمج مبتدئ.

22.3 تحويل البرنامج إلى وحدة

إذا أردنا تحويل البرنامج الذي كتبناه إلى وحدة فيجب أن نتبع بعض مبادئ التصميم الأساسية، فنضع الجزء الأكبر من الشيفرة في دوال ليستطيع مستخدمو الوحدة الوصول إليها، ثم ننقل شيفرة البدء التي تحصل على

اسم الملف إلى جزء منفصل من الشيفرة؛ لا يُنفَّذ عند استيراد الوحدة، وأخيرًا سنترك التعريفات العامـة global definitions متغيرات على مستوى الوحدة ليستطيع المستخدمون تغيير قيمها إذا أرادوا.

لننقل كتلـة المعالجـة الأساسـية الآن إلى دالـة نسـميها `analyze()`، وسـنمرر كـائن ملـف معـامِلًا إليهـا، وستعيد الدالة قائمةً من قيم العد في صف tuple، وستبدو الشيفرة كما يلي:

```
##############################
# Module: grammar
# Created: A.J. Gauld, 2010/12/02
# Modified: A.J. Gauld, 2018/01/12
#
# Function:
'''
Provides facilities to count words, lines, characters,
paragraphs, sentences and 'clauses' in text files.
It assumes that sentences end with [.!?] and paragraphs
have a blank line between them. A 'clause' is simply
a segment of sentence separated by punctuation. The
sentence and clause searches are regular expression
based and the user can change the regex used. Can also
be run as a program.'''
##############################

import re, sys

# اضبط المتغيرات العامة
paras = 1 # We will assume at least 1 paragraph!
lines, sentences, clauses, words, chars = 0,0,0,0,0
sentenceMarks = '.?!'
clauseMarks = '&();:,/\-' + sentenceMarks
sentenceRE = None  # set via a function call
clauseRE = None
format = '''
The file %s contains:
        %d\t characters
        %d\t words
        %d\t lines in
```

```
        %d\t paragraphs with
        %d\t sentences and
        %d\t clauses.
'''


############################
# عرّف الدوال التي ستنفذ العمل

# تسمح لنا setCounterREs بإعادة تصريف التعبير النمطي إذا
# غيرنا قائمة الأجزاء
def setCounterREs():
    "compiles global regexs from global punctuation sets"
    global sentenceRE, clauseRE
    sentenceRE = re.compile('[%s]' % sentenceMarks)
    clauseRE = re.compile('[%s]' % clauseMarks)

# تستدعي analyze() تصفير العدادات
def resetCounters():
    " reset global counter variables to initial values "
    chars, words, lines, sentences, clauses = 0,0,0,0,0
    paras = 1


# توجَّه reportStats لشيفرة التعريفات، إذ توفر تقريرًا نصيًا بسيطًا
def reportStats(theFile):
    " prints out results from global results "
    print( format % (theFile.name, chars, words, lines,
                     paras, sentences, clauses) )

# هي الدالة الأساسية التي تعالج الملف analyze()
def analyze(theFile):
    ''' analyze(aFile) -> None

    Analyzes the input file object putting results in global variables
    '''
```

```
    global chars,words,lines,paras,sentences,clauses
    # تحقق إن كان REs مصرَّفًا بالفعل
    if not (sentenceRE and clauseRE):
        setCounterREs()
    resetCounters()
    for line in theFile:
        lines += 1
        if line.strip() == "": # empty line
            paras += 1
        words += len(line.split())
        chars += len(line.strip())
        sentences += len(sentenceRE.findall(line))
        clauses += len(clauseRE.findall(line))


# اسمح بتشغيلها إذا استُدعيت من سطر الأوامر
# '__main__' على '__name__' يُضبط متغير
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print( "Usage: python grammar.py <filename>" )
        sys.exit()
    else:
        with open(sys.argv[1],"r") as aFile:
            analyze(aFile)
            reportStats(aFile)
```

إن أول ما نلاحظه هنا هو التعليقات التي في الأعلى، وهذا سلوك شائع لإعطاء فكرة عامة لمن يقـرأ الملـف عن محتوياته وكيفية استخدامه، كما أن معلومات الإصدار التي تشمل المؤلف والتاريخ مهمة عند موازنة النتائج مع شخص آخر قد يستخدم إصدارًا أحدث، ونلاحظ أن وصف الوحدة هو سلسلة نصية غير مخصصة لمتغـير مـا، وتذكر أن هذا ينشئ سلسلة توثيق documentation string في بايثون، يمكن الاطلاع عليها باستخدام الدالـة help، كما لدينا سلاسل توثيق على كل تعريف دالة مستقلة، ولن نضيف تعليقـات كثـيرةً في الأمثلـة الأخـرى، رغم أننا سنعرض بعض الأنماط الأخرى من التعليقات والتوثيقات، لأن النص الأساسي يصف ما يحـدث، فهـذا المثال يوضح ما يمكن أن نوفره.

أما الجزء الأخير فهو خاصية في بايثون تستدعي أي وحدة محمَّلة في سطر الأوامر "__main__"، نستطيع تجريب المتغير الخاص والمضمَّن __name__، حيث نعرف أن الوحدة ستُستورد وتشغَّل، لذا ننفذ شيفرة التشغيل driver code داخل الكتلة if.

تتضمن شيفرة التعريف تلك إرشادًا حول كيفية تشغيل الملف إذا لم يتوفر اسم ملف، أو إذا توفرت أسماء ملفات كثيرة، فتسأل المستخدم عن اسم الملف باستخدام input()، ونلاحظ أن الدالة analyze() تستخدم دوال الضبط لضمان ضبط العدادات والتعابير العادية ضبطًا صحيحًا قبل أن تبدأ، مما يفيد المستخدم عند استدعاء analyze عدة مرات، خاصةً بعد تغيير التعابير النمطية المستخدمة في عد الجمل وأجزائها.

كما نلاحظ استخدام global لضمان ضبط المتغيرات على مستوى الوحدة بواسطة الدوال، ومن دونها كنا سننشئ متغيرات محليةً ليس لها تأثير على متغيرات الوحدة.

22.3.1 استخدام الوحدة grammar

بعد أن أنشأنا وحدةً نستطيع استخدامها مثل برنامج في سطر أوامر النظام، كما فعلنا من قبل عن طريق ما يلي:

```
C:\> python grammar.py spam.txt
```

إلا أننا نستطيع استيراد الوحدة إلى برنامج آخر أو في محث بايثون، شرط أن نكون قد حفظنا الوحدة في موقع تستطيع بايثون أن تراه.

لنجرِ الآن بعض الاختبارات على ملف اختبارات اسمه spam.txt، والذي يعطي الناتج التالي:

```
This is a file called spam. It has
lines, 2 sentences and, hopefully,
clauses.
```

أي هذا ملف اسمه spam، فيه 3 أسطر وجملتين وخمسة أجزاء جمل. لنشغل بايثون الآن ونجرب:

```
>>> import grammar
>>> grammar.setCounterREs()
>>> txtFile = open("spam.txt")
>>> grammar.analyze(txtFile)
>>> grammar.reportStats(txtFile)
The file spam.txt contains:
characters
words
lines in
```

```
paragraphs with
sentences and
clauses.

>>> txtFile.close()
>>> txtFile = open('spam.txt')
>>> grammar.resetCounters()
>>>  # redefine sentences as ending in vowels!
>>> grammar.sentenceMarks = 'aeiou'
>>> grammar.setCounterREs()
>>> grammar.analyze(txtFile)
>>> print( grammar.sentences )
21
>>> txtFile.close()
```

نلاحظ أنه يفضل استخدام `open/cloSe` على الملف عند استخدام المحث التفاعلي بدلًا من استخدام `with`، لأن with ستؤخر التنفيذ حتى نكتب جميع عمليات الملف التي ستكون داخل الكتلة `with`، أما استخدام `open` صراحةً فسيمكننا من تنفيذ كل أمر تنفيذًا منفصلًا.

نستطيع الآن أن نرى أن إعادة تعريف أجزاء الجمل قد غيرت عدد الجمل تغييرًا جذريًا، ولا شك أن التعريف الجديد غريب لكنه يظهر أن الوحدة قابلة للاستخدام والتخصيص، ونلاحظ أننا كنا نستطيع طباعة عدد الجمل مباشرةً، دون الحاجة إلى استخدام الدالة `reportStats()`، وهذا يظهر قيمة مبدأ مهم في التصميم، وهو فصل البيانات عن العرض display، فلما فصلنا عرض البيانات عن حسابها صارت وحدتنا أكثر مرونةً للمستخدمين.

ولننهي مشروعنا نعيد صياغة وحدة القواعد grammar لتستخدم تقنيات كائنية التوجه، ثم نضيف واجهةً رسوميةً بسيطةً، وسنرى خلال ذلك كيف يعطينا المنظور الكائني وحدات أكثر مرونةً للمستخدم، وقابلةً للتوسيع أيضًا.

22.4 الأصناف والكائنات

من أكبر المشاكل التي يواجهها من يستخدم وحدتنا الاعتماد على المتغيرات العامة، مما يعني أنها تستطيع تحليل ملف واحد في كل مرة، وسيؤدي تحليل أكثر من ملف إلى تغيير القيم العامة، فإذا نقلنا هذه المتغيرات العامة إلى صنف class، فسنستطيع إنشاء عدة نسخ من الصنف -واحد لكل ملف-، وستحصل كل نسخة على مجموعتها الخاصة من المتغيرات، وإذا جعلنا التوابع دقيقةً بما يكفي فيمكن إنشاء بنية يسهل من خلالها -على

منشئ نوع جديد من كائن المستند- تعديل معايير البحث ليوافق احتياجات النوع الجديد، فإذا رفضنا جميع وسوم HTML من قائمة الكلمات مثلًا فيمكن معالجة ملفات HTML إضافةً إلى ملفات آسكي ASCII الخالصة.

أدت محاولتنا الأولى في هذا إلى إنشاء الصنف Document لتمثيل الملف الذي نعالجه:

```
#! /usr/local/bin/python
################################
# Module: document.py
# Author: A.J. Gauld
# Date:   2010/12/10
# Version: 3.1
################################
'''
Provides 2 classes for parsing text/files.
A Generic Document class for plain ASCII text,
and an HTMLDocument for HTML files.

 Primary services available include
    - analyze(),
    - reportStats().
'''

import sys,re

class Document:
    sentenceMarks = '?!.'
    clauseMarks = '&()/\-;:,' + sentenceMarks

    def __init__(self, filename):
        self.filename = filename
        self.setREs()
        self.resetCounters()

    def resetCounters(self):
        self.paras = 1
        self.lines = self.getLines()
        self.sentences, self.clauses, self.words, self.chars = 0,0,0,0
```

```
    def setREs(self):
        self.sentenceRE = re.compile('[%s]' % Document.sentenceMarks)
        self.clauseRE = re.compile('[%s]' % Document.clauseMarks)

    def getLines(self):
        with open(self.filename)as infile:
            lines = infile.readlines()
        return lines

    def analyze(self):
        self.resetCounters()
        for line in self.lines:
            self.sentences += len(self.sentenceRE.findall(line))
            self.clauses += len(self.clauseRE.findall(line))
            self.words += len(line.split())
            self.chars += len(line.strip())
            if line.strip() == "":
                self.paras += 1

    def formatResults(self):
        format = '''
        The file %s contains:
        %d\t characters
        %d\t words
        %d\t lines in
        %d\t paragraphs with
        %d\t sentences and
        %d\t clauses.
        '''
        return format % (self.filename, self.chars,
                         self.words, len(self.lines),
                         self.paras, self.sentences, self.clauses)

class TextDocument(Document):
    pass
```

```
class HTMLDocument(Document):
    pass

if __name__ == "__main__":
       if len(sys.argv) == 2:
           doc = Document(sys.argv[1])
           doc.analyze()
           print( doc.formatResults() )
       else:
           print( "Usage: python document3.py  <file>" )
           print( "Failed to analyze file" )
```

نلاحظ استخدام متغيرات الصنف في بداية تعريفه، لتخزين محددات الجمل وأجزائها، حيث تُشارَك متغيرات الصنف بين جميع نسخه، لذا فهي مكان ممتاز لتخزين المعلومات المشتركة، ويمكن الوصول إليها باستخدام اسم الصنف كما فعلنا هنا، أو باستخدام `self`، ونفضل استخدام اسم الصنف لأنه يبرز حقيقة أن هذه المتغيرات لصنف.

لا زلنا بحاجة إلى `resetCounters()` للمرونة في التعامل مع أنواع المستندات الأخرى، رغم أننا نخزن المتغيرات الآن داخل الصنف، فعلى الأرجح أننا سنستخدم مجموعة عدادات أخرى عند تحليل ملفات HTML -مثل عدد الوسوم-، ونستطيع التعامل مع أي نوع من المستندات تقريبًا إذا جمعنا `resetCounters()` مع `formatResults()`، ووفرنا تابع `analyze()` جديد، أما التوابع الأخرى فهي أكثر استقرارًا، لأن قراءة أسطر الملف أمر قياسي بغض النظر عن نوعه، وضبط التعبيرين النمطيين فرصة جيدة للتدرب، فإذا لم نكن بحاجة إلى ذلك فلا نفعله.

لدينا الآن وظائف مماثلة لنسخة وحدتنا الخاصة لكننا عبرنا عنها في صنف، ونريد أن نستغل الأسلوب الكائني من خلال فك أجزاء من صنفنا كي لا يحتوي المستوى الأساسي أو `Document` المجرد إلا على أجزاء عامة، وسننقل الأجزاء الخاصة بمعالجة النصوص إلى الصنف `TextDocument` أكثر تحديدًا، وتُعرف هذه العملية بإعادة التصميم (Refactoring) في أوساط البرمجة الاحترافية، وسنرى كيفية تنفيذ ذلك فيما يلي.

22.5 المستند النصي

المستندات النصية مألوفة، لكننا يجب أن نتريث لنوضح الغرض من موازنة المستند النصي بالمفهوم العام للمستندات، تتكون المستندات النصية من محارف مرتبة في سطور، تحتوي مجموعات من الأحرف مرتبةً في كلمات تفصل بينها مسافات وعلامات ترقيم أخرى، وإذا جمعنا تلك الأسطر في مجموعات فسيتكون لدينا فقرات يُفصل بينها بأسطر فارغة.

والمستند الافتراضي -يُطلق عليه vanilla document أو مستند الفانيليا أحيانًا في إشارة إلى نكهة المثلجات الافتراضية- يتكون من أسطر من المحارف التي لا نعرف عن صياغتها إلا القليل، لذا يجب أن يكون صنفنا `Document` الأساسي قادرًا على فتح الملف وقراءة محتوياته إلى قائمة من الأسطر، وربما يعيد عدد المحارف والأسطر مثلًا، كما يوفر توابع خطافيةً hook methods فارغةً للأصناف الفرعية الخاصة بالمستند لاستخدامها، لاحظ أن نص آسكي هو أحد أقدم وأبسط الطرق للتعبير عن النصوص، إلا أن الأبجديات التي أضيفت إليه وإضافة اليونيكود قد جعلته معقدًا.

وفقًا لما شرحناه في الفقرات السابقة يجب أن يبدو الصنف Document كما يلي:

```
##############################
# Module: document
# Created: A.J. Gauld, 2010/12/15
# Version 3
# Function:
''' Provides abstract Document class to count lines, characters
and provide hook methods for subclasses to use to process
more specific document types'''
##############################

import sys,re

class Document:
   def __init__(self,filename):
       self.filename = filename
       self.lines = self.getLines()
       self.chars = sum( [len(L) for L in self.lines] )
       self._initSeparators()

   def getLines(self):
       f = open(self.filename,'r')
       lines = f.readlines()
       f.close()
       return lines

   # قائمة بالتوابع الخطافية التي يجب تغييرها
   def formatResults(self):
```

```
    return "%s contains $d lines and %d characters" % (len(self.lines),
                                                       self.chars)
    def _initSeparators(self): pass
    def analyze(self): pass
```

نلاحظ أن التابع `_initSeparators` يحوي شرطةً سفليةً قبل اسمه، حيث يستخدم هذا الاصطلاح مبرمجو بايثون للإشارة أن هذا التابع لا يُستدعى إلا من داخل توابع الصنف، وليس مخصصًا ليصل إليه مستخدمو الكائن، ويسمى مثل هذا التابع أحيانًا في بعض اللغات الأخرى بالتابع المحمي protected أو الخاص private.

كما نلاحظ أننا استخدمنا الدالة `sum()` لحساب عدد المحارف، وهي تعيد مجموع قائمة من الأعداد، والقائمة في حالتنا هي قائمة أطوال الأسطر في الملف المنتج بواسطة `list comprehension`.

لم نوفر خيارًا قابلًا للتشغيل باستخدام `if __name__ == etc` لأن هذا الصنف مجرد abstract. يجب أن يبدو المستند النصي الآن كما يلي:

```
class TextDocument(Document):
    def __init__(self,filename):
        super().__init__(filename)
        self.paras = 1
        self.words, self.sentences, self.clauses = 0,0,0

    # غيّر الخطاطيف الآن
    def formatResults(self):
        format = '''
        The file %s contains:
              %d\t characters
              %d\t words
              %d\t lines in
              %d\t paragraphs with
              %d\t sentences and
              %d\t clauses.
        '''
        return format % (self.filename, self.chars,
                         self.words, len(self.lines),
                         self.paras, self.sentences, self.clauses)
```

```
    def _initSeparators(self):
        sentenceMarks = "[.!?]"
        clauseMarks = "[.!?,&:;-]"
        self.sentenceRE = re.compile(sentenceMarks)
        self.clauseRE = re.compile(clauseMarks)

    def analyze(self):
        for line in self.lines:
            self.sentences += len(self.sentenceRE.findall(line))
            self.clauses += len(self.clauseRE.findall(line))
            self.words += len(line.split())
            self.chars += len(line.strip())
            if line.strip() == "":
                self.paras += 1

if __name__ == "__main__":
    if len(sys.argv) == 2:
        doc = TextDocument(sys.argv[1])
        doc.analyze()
        print( doc.formatResults() )
    else:
        print( "Usage: python <document> " )
        print( "Failed to analyze file" )
```

يحقق دمج الأصناف هذا نفس ما يحققه الإصدار غير الكائني الأول، وإذا وازنّا بين طول هذه النسخة وطول الملف الأصلي فسندرك أن بناء كائنات قابلة لإعادة الاستخدام ليس سهلًا، لذا ينبغي أن نكتب إصدارات غير كائنية دومًا؛ ما لم نكن بحاجة إلى إعادة استخدام الكائنات، كأن نخطط لتوسيع التصميم مستقبلًا، كما سنفعل بعد قليل.

ومن المهم أن نراعي الموقع الفعلي للشيفرة، فقد كان بإمكاننا عرض إنشاء ملفين، واحد لكل صنف، وهو سلوك شائع في البرمجة الكائنية، ويحافظ على النظام العام، رغم أنه يكون على حساب كثير من الملفات الصغيرة، وكثير من تعليمات الاستيراد في الشيفرة عند استخدام تلك الملفات أو الأصناف.

ونفضل أن نعامل الأصناف المرتبطة ببعضها بشدة مثل مجموعة، ونضعها جميعًا في ملف واحد، بما يكفي على الأقل لإنشاء برنامج صغير عامل، لذا جمعنا الصنفين `Document` و`TextDocument` في وحدة واحدة، وميزة ذلك أن الصنف العامل يوفر قالبًا للمستخدمين ليقرؤوه مثالًا على توسيع الصنف المجرد، لكن

عيبه أن أي تعديل في TextDocument سيؤثر على صنف المستند، وبالتالي سيعطل أجزاءً من الشيفرة، فلا يوجد حل صالح هنا، وتوجد أمثلة على كلا النمطين في مكتبة بايثون، فاختر أحدهما والتزم به.

من مصادر المعلومات المفيدة في مثل هذا النوع من التعديل على الملفات النصية كتاب "معالجة النصوص في بايثون" أو Text Processing in Python لـ David Mertz، لكن لاحظ أن هذا الكتاب متقدم وموجه إلى المبرمجين المحترفين، لذا قد تجد صعوبةً في استيعاب مادته إذا كنت مبتدئًا في البرمجة، لكنك ستجد فيه دروسًا مفيدةً للغاية بالمثابرة والتعلم.

22.6 مستند HTML

الخطوة التالية في تطوير تطبيقنا هي توسيع إمكانياته لنستطيع تحليل مستندات HTML، وسنفعل ذلك بإنشاء صنف جديد، وبما أن مستند HTML ما هو إلا مستند نصي يحتوي على الكثير من وسوم HTML وقسم للترويسة في الأعلى؛ فكل ما نحتاج إليه هو حذف هذه العناصر الإضافية ومعاملته بعدها على أنه نص عادي، لذا سننشئ صنفًا جديدًا نسميه HTMLDocument مشتقًا من TextDocument، وسنغير التابع getLines() الذي ورثناه من Document بحيث يحذف الترويسة ووسوم HTML.

سيبدو الصنف HTMLDocument الآن كما يلي:

```
class HTMLDocument(TextDocument):
    def getLines(self):
        lines = super().getLines()
        lines = self._stripHeader(lines)
        lines = self._stripTags(lines)
        return lines

    def _stripHeader(self,lines):
        ''' remove all lines up until start of body '''
        bodyMark = '<body>'
        bodyRE = re.compile(bodyMark,re.IGNORECASE)
        while bodyRE.findall(lines[0]) == []:
            del lines[0]
        return lines

    def _stripTags(self,lines):
        ''' remove anything between < and >, not perfect but ok for
now'''
        tagMark = '<.+>'
```

```
        tagRE = re.compile(tagMark)
        lines2 = []
        for line in lines:
            line = tagRE.sub('',line).strip()
            if line: lines2.append(line)
        return lines2
```

استخدمنا التابع الموروث داخل `getLines`، وهذا سلوك شائع عند توسيع تـابع مـوروث،حيث ننفـذ ذلـك بمعالجة أولية، أو نستدعي الشيفرة الموروثة ثم نكمل باقي العمل في الصنف الجديد كما فعلنا هنا، وينفّـذ هـذا في التابع `__init__` الخاص بالصنف `TextDocument` أعلاه.

وصـلنا إلى التـابع المـوروث `getLines` من خلال `super()`، رغم أن التـابع معـرَّف فعليًـا في الصـنف `Document` المجـرد في الأعلى، ويـرث `TextDocument` جميـع مزايـا `Document`، وبالتـالي يحتـوي على `getLines` أيضًا، وهكذا نرى أن بايثون و`super` تجدان التابع المناسب دومًا.

أما التابعان الآخران فهما محميان نظريًا -كما هو واضح من الشرطة السفلية قبل اسميهما-، وهما هنا لإبقاء المنطق مستقلًا، ولتسهيل توسيع هذا الصنف في المستقبل إلى مستند XML مثلًا، فتدرب على بناء أحدهما.

من الصعب أن نحذف جميع وسوم HTML باسـتخدام التعـابير النمطيـة بسـبب إمكانيـة تشـعب الوسـوم، وبسبب احتمال حدوث خطأ في احتساب محـرفي < و> غـير المهـرَّبيْن على أنهمـا وسـوم في حين أنهمـا ليسـا كذلك، كما أن الوسوم قد تأتي على أكثر من سطر، وغير ذلك من احتمالات الخطأ الواردة، فالأسلم هنـا اسـتخدام محلل HTML -مثل الموجود في وحدة `html.parser` القياسية- لتحويـل ملفـات HTML إلى نص عـادي، لـذا أعد كتابة الصنف `HTMLDocument` لتستخدم وحدة المحلل هنا لتوليد الأسطر النصية.

نحتـاج الآن إلى تعـديل شـيفرة التعريـف في نهايـة الملـف لتكـون على الصـورة التاليـة من أجل اختبار HTMLDocument:

```
if __name__ == "__main__":
        if len(sys.argv) == 2:
          doc = HTMLDocument(sys.argv[1])
          doc.analyze()
          print( doc.formatResults() )
        else:
          print( "Usage: python <document> " )
          print( "Failed to analyze file" )
```

وإذا كنت على دراية ببعض أنواع الملفات الأخرى مثل ملفات PDF و LaTeX و RTF و Postscript وغيرها، فهل تستطيع إنشاء أصناف تحقق وفحص لها؟

22.7 إضافة واجهة رسومية

سنستخدم Tkinter الذي شرحناه باختصار في الفصل الثامن عشر: البرمجة المدفوعة بالأحداث، ثم توسعنا فيه بتفصيل أكثر في الفصل التاسع عشر: برمجة الواجهات الرسومية، أما هنا فستكون الواجهة الرسومية أكثر تعقيدًا وتستخدم مزيدًا من الودجات widgets التي يوفرها Tk، ومن العوامل التي ستعيننا على إنشاء النسخة الرسومية تجنبنا لوضع أي تعليمات طباعة في أصنافنا، ووضعنا لطباعة الخرج في شيفرة التعريف بدلًا من ذلك، وهذا سيفيدنا حين نستخدم الواجهة الرسومية، حيث سنستطيع استخدام سلسلة الخرج نفسها وعرضها في ودجت، بدلًا من طباعتها على مجرى الخرج القياسي stdout، ويُعد تغليف التطبيق في واجهة رسومية بسهولة من أهم الأسباب التي نتجنب استخدام تعليمات الطباعة في الدوال أو التوابع الخاصة بمعالجة البيانات لأجلها.

22.7.1 تصميم الواجهة الرسومية

إن الخطوة الأولى في بناء أي برنامج رسومي هي محاولة تصور شكله النهائي، فمثلًا سنحتاج إلى تحديد اسم ملف، لذا سنضيف متحكمًا للتعديل أو الإدخال النصي، كما سنحدد إذا كنا نريد تحليل ملف نصي أم ملف HTML، فنمثل هذا الاختيار من متعدد بمجموعة من متحكمات أزرار الانتقاء radio buttons، ويجب أن تُجمع هذه الأزرار معًا لإظهار ارتباطها ببعضها.

أما الشرط الثاني فهو أسلوب عرض النتائج، ورغم أنه يمكن اعتماد عدة متحكمات للعناوين؛ بحيث يكون لدينا عنوان لكل عداد، إلا أننا سنستخدم متحكمًا نصيًا بسيطًا نستطيع إدخال سلاسل نصية فيه، وهو بهذا أقرب إلى خرج سطر الأوامر، مع أن هذا يرجع بالنهاية لمصمم البرنامج، لكن هذا التصميم الذي سنعتمده لن يكون الأجمل مظهرًا.

وأخيرًا نحتاج إلى وسيلة لبدء التحليل والخروج من التطبيق، وبما أننا نستخدم متحكمًا نصيًا لعرض النتائج؛ فقد يكون من الأفضل أن يكون لدينا وسيلة لإعادة ضبط الشاشة، ويمكن تمثيل جميع خيارات الأوامر بمتحكمات أزرار.

إذا رسمنا هذه الأفكار أعلاه فقد نحصل على شكل قريب مما يلي:

```
+-------------------------+-----------+
|    FILENAME             | O TEXT    |
|                         | O HTML    |
+-------------------------+-----------+
|                                     |
|                                     |
|                                     |
```

```
|                                     |
|                                     |
+-------------------------------------+
|                                     |
|   ANALYZE      RESET       QUIT     |
|                                     |
+-------------------------------------+
```

يشبه هذا التصميم ثلاثة إطارات بالعرض الكامل فوق بعضها، وفي الإطار العلوي إطاران جنبًا إلى جنب، ويحتوي الإطار الأيمن العلوي على زري انتقاء مرتبين رأسيًا، أما الإطار السفلي فيحتوي على ثلاثة أزرار مرتبة أفقيًا، ونستطيع استخدام تخطيط المحزِّم pack لكل ما سبق، وبهذا نعلم الودجات التي نحتاج إليها، ومدير التخطيط المطلوب لكل إطار، ويتبقى لدينا مزية واحدة جديدة، وهي أزرار الانتقاء، وسننظر فيها لاحقًا.

لنحول ما سبق إلى شيفرة:

```
import tkinter as tk
import document

################### CLASS DEFINITIONS ######################
class GrammarApp(Frame):
  def __init__(self, parent=0):
    super().__init__(parent)
    self.type = 1    # create variable with default value
    self.master.title('Grammar counter')
    self.buildUI()
```

استوردنا وحدتي `tkinter` و`document`، وجعلنا كل أسماء Tkinter مرئيةً داخل الوحدة الحالية، أما بالنسبة للوحدة الثانية فسنحتاج إلى سبق الأسماء بـ `document.`.

كما عرَّفنا التطبيق على أنه صنف فرعي للصنف `Frame`، ويستدعي التابع `__init__` تابع الصنف الرئيسي `Frame.__init__` لضمان إعداد Tk بالطريقة الصحيحة داخليًا، ثم ننشئ سمةً تخزن قيمة نوع المستند، ونستدعي التابع `buildUI` الذي ينشئ جميع الودجات لنا، وسننظر الآن فيه:

```
    def buildUI(self):
      # إطار الخيارات أولًا
      # - اسم الملف ونوعه
      fOpts = tk.Frame(self)
      fFile = tk.Frame(fOpts)
```

```
        tk.Label(fFile, text="Filename: ").pack(side="left")
        self.eName = tk.Entry(fFile)
        self.eName.insert(tk.INSERT,"test.htm")
        self.eName.pack(side='left', padx=5)
        fFile.pack(side='left', padx=3)

        # والآن أزرار الانتقاء
        fType = Frame(fFile, borderwidth=1, relief=tk.SUNKEN)
        self.rText = Radiobutton(fType, text="TEXT",
                                 variable = self.type, value=1,
                                 command=self.doText)
        self.rText.pack(side=tk.TOP, anchor=tk.W)
        self.rHTML = Radiobutton(fType, text="HTML",
                                 variable=self.type, value=2,
                                 command=self.doHTML)
        self.rHTML.pack(side=tk.TOP, anchor=tk.W)

        # اجعل TEXT هو الخيار الافتراضي
        self.rText.select()
        fType.pack(side=tk.RIGHT, padx=3)
        fOpts.pack(side=tk.TOP, fill=tk.X)


        # يحتوي الصندوق النصي على الخرج، فأضف له حشوة
        # لإضافة حد إليه، واجعل الأب هو إطار التطبيق (أي self)
        self.txtBox = Text(self, width=60, height=10)
        self.txtBox.pack(side=tk.TOP, padx=3, pady=3)


        # ضع بعض أزرار التحكم
        fButts = Frame(self)
        self.bAnal = Button(fButts, text="Analyze",
                            command=self.doAnalyze)
        self.bAnal.pack(side=tk.LEFT, anchor=tk.W, padx=50, pady=2)
        self.bReset = Button(fButts, text="Reset",
                             command=self.doReset)
```

```
        self.bReset.pack(side=tk.LEFT, padx=10)
        self.bQuit = Button(fButts, text="Quit",
                            command=self.quit)
        self.bQuit.pack(side=tk.RIGHT, anchor=tk.E, padx=50, pady=2)

        fButts.pack(side=tk.BOTTOM, fill=tk.X)
        self.pack()
```

لن نشرح كل هذا إذ يجب أن يكون مفهومًا لمن قرأ الفصل التاسع عشر: برمجة الواجهات الرسومية، أما لمن أراد الاستزادة فعليه فيمكنه الرجوع إلى مراجع وتوثيقات أخرى خارجية، والقاعدة العامة هي أن ننشئ ودجات من أصنافها الموافقة لها، ونوفر الخيارات مثل معاملات ذوات أسماء، ثم نحزّم الودجت في إطارها الحاوي لها.

كذلك قد ترغب في تجربة إضافة زر "browse" الذي يفتح صندوق FileOpen الحواري، ويمكن تنفيذ ذلك باستخدام صناديق Tk الافتراضية، انظر الدليل أعلاه لمزيد من الشرح.

أما النقاط الأخرى التي يجب ملاحظتها فهي استخدام ودجات `Frame` الفرعية لتحوي أزرار الانتقاء `Radiobuttons` وأزرار الأوامر `Button`، كما تأخذ أزرار الانتقاء زوجًا من الخيارات هما `variable` و`value`، ويربط الأول أزرار الانتقاء معًا، من خلال تحديد نفس المتغير الخارجي `self.type`، ويعطي الخيار الثاني قيمةً فريدةً لكل زر منها.

لاحظ أيضًا الخيارات `command=xxx` الممررة إلى المتحكمات، فهي توابع سيستدعيها Tk عند الضغط على الزر. الشيفرة الممثلة لما سبق هي كما يلي:

```
    ################## EVENT HANDLING METHODS ####################

    # استعد الإعدادات الافتراضية
    def doReset(self):
        self.txtBox.delete(1.0, tk.END)
        self.rText.select()

    # اضبط قيم الانتقاء
    def doText(self):
        self.type = 1

    def doHTML(self):
        self.type = 2
```

تأتي أزرار الانتقاء مع القليل من سحر Tk الذي يسمح لها أن تُربط مع أنواع خاصة من متغيرات بايثون، بحيث إذا ضُغط على الزر فستُسند قيمته تلقائيًا إلى المتغير المرتبط به، ولم نستخدم هذه التقنية لأنها خاصة بصندوق Tk وحده، واستخدمنا الآلية المعتادة لمعالجة الأحداث بدلًا منها ليكون الشرح واضحًا، وليكون صريحًا في الدالة.

للاطلاع على المزيد حول الخيار المؤتمت فانظر `IntVar` و`StringVar` في توثيق Tkinter، حيث يمكن استخدام هذه الأنواع الخاصة من المتغيرات مع العديد من ودجات ضبط القيم في Tk، لكنها خارج نطاق شرحنا.

يتبقى لدينا آخر معالج أحداث، وهو الذي ينفذ التحليل:

```
    # أنشئ نوع المستند المناسب وحلله.
    # ثم اعرض النتائج في الاستمارة
    def doAnalyze(self):
      filename = self.eName.get()
      if filename == "":
         self.txtBox.insert(tk.END,"\nNo filename provided!\n")
         return
      if self.type == 1:
         doc = document.TextDocument(filename)
      else:
         doc = document.HTMLDocument(filename)
      self.txtBox.insert(tk.END, "\nAnalyzing...\n")
      doc.analyze()
      resultStr = doc.formatResults()
      self.txtBox.insert(tk.END, resultStr)
```

تتحقق هذه الشيفرة من وجود اسم ملف صالح قبل إنشاء كائن المستند، رغم أنها قد لا تتحقق من صلاحية الاسم.

تستخدم قيمة `self.type` المحددة بواسطة أزرار الانتقاء لتحديد نوع المستند الذي يجب إنشاؤه.

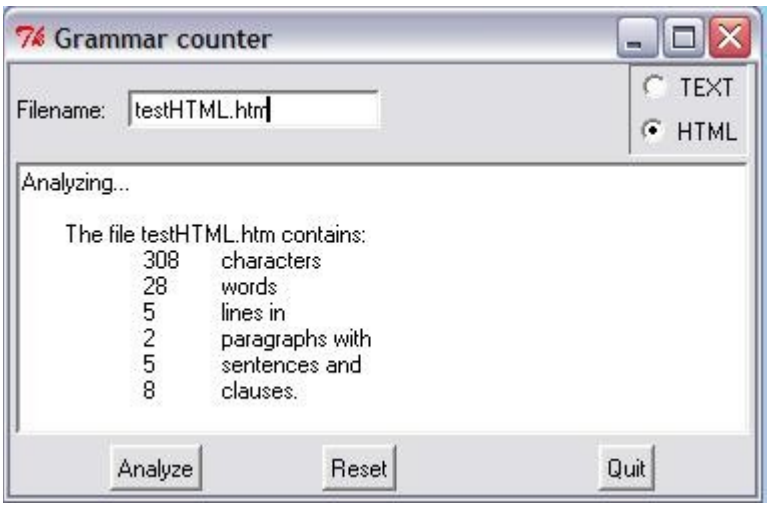
تلحَق النتائج بالحقل النصي -الوسيط `tk.END` في `insert`- مما يعني أننا نستطيع التحليل عدة مرات وموازنة النتائج، وهذه ميزة الصندوق النصي هنا عن أسلوب خرج العناوين المتعددة multiple label output.

وكل ما نحتاج إليه الآن هو إنشاء نسخة لصنف التطبيق `GrammarApp` وتشغيل حلقة الحدث:

```
if __name__ == "__main__":
    myApp = GrammarApp()
```

```
    myApp.mainloop()
```

لننظر الآن إلى النتيجة النهائية في نظام ويندوز، والتي تعرض نتائج تحليل ملف HTML:



من الممكن جعل معالجة ملف HTML أكثر تعقيدًا إذا أردنا، حيث نتحقق أكثر من الأخطاء -مثل التحقق من عدم وجود الملف-، وننشئ وحدات لأنواع المستندات الجديدة، ونستبدل عدة عناوين مجموعة في إطار واحد بالصندوق النصي، كما يمكن استخدام قائمة منسدلة لأنواع المستندات، خاصةً إذا أضفنا أنواعًا جديدة.

22.8 خاتمة

ينظر الفصل التالي في الجانب العملي من بايثون في مشاريع حقيقية، وستكون الأمثلة أطول ولن نكثر من التفاصيل في الشرح، إذ يجب أن تكون الآن قادرًا على متابعة الأمثلة بسهولة.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

الباب الرابع: تطبيقات

23. مشاريع تطبيقية باستخدام بايثون

سنركز في هذا الجـزء من الكتـاب على بعض التطبيقـات العمليـة الـتي يمكن اسـتخدام لغـة بـايثون فيهـا، وكذلك وحدات المكتبات التي سنستخدمها في كتابة تلك التطبيقات، كما سنتعلم بعض التقنيات الجديدة أثناء كتابة تلك التطبيقات، مثل قواعد البيانات وشبكات الحواسيب والشبكة العالمية -الويب-، وكذلك بعض المزايـا الأساسية لنظم تشغيل الحواسيب، لكننا لن نتعمـق في هـذه التقنيـات كثـيرًا لأننـا نهتم بتعليم البرمجـة فقـط، ويمكنك الاستعانة بمواقع أخرى أو مصادر يمكن الاستزادة منها.

وقد اخترنا موضوعات تعكس المجالات التي تتكـرر في قائمـة Python tutor البريديـة (القائمـة البريديـة لمعلمي بايثون)، والتي يفترض أن تكون الأفضل في تلبية احتياجـات المـبرمجين الجـدد، فـإذا لم يكن المجـال الذي تريده موجودًا فيها فقد تجد روابط إلى مصادر تتعلم فيها عن ذلك المجال.

تعتمد جميع الفصول الباقية من الكتاب على بايثون وحدها، ولا شك أن الإمكانيات التي سنذكرها قد تكون موجودةً في جافاسكربت وVBScript، إلا أن الاختلافات في هذا المستوى من العمق والتفاصيل أكـبر من أوجـه التشابه بكثير، لأن هذه تطبيقات عملية وليست مفاهيم نظريـةً.، فأسـهل طريقـة للوصـول إلى وينـدوز مثلًا من جافاسكربت أو VBScript سـتكون من خلال مضـيف سـكربت وينـدوز WSH الـذي ذكرنـاه من قبـل، لكن هـذا يختلف كليًا عن وحدة نظام التشغيل في بايثون، فلا معنى للموازنة بينها.

23.1 الفصول التالية من الكتاب

سنتعلم فيما بقي من الكتاب عدة تطبيقات عملية، وستكون بهذا الترتيب:

23.1.1 التعامل مع قواعد البيانات

سنصل لا شك -كوننا مبرمجين- إلى مرحلة نحتاج فيها إلى تخزين مجموعات معقدة من البيانات وجلبها مرةً أخرى عند الحاجة إليها عند تصميم الأنظمة والبرامج ، وتوفر بايثون عدة طرق لتخزين البيانات البسيطة بسهولة، لكن أقوى آلية لتخزين البيانات هي قواعد البيانات العلائقية relational databases.

ويشرح هذا الفصل المبادئ التي تقوم عليها قواعد البيانات تلك، ولغة الاستعلامات الهيكلية Structured Query LAnguage أو المشهورة باسم SQL، والتي تستخدم في التعامل مع قواعد البيانات، ثم نختم بمثال بسيط لاستخدام قاعدة بيانات داخل بايثون، حيث نوسع مثال دليل جهات الاتصال الذي درسناه في عدة فصول من قبل.

23.1.2 استخدام نظم تشغيل الحواسيب

نظام تشغيل الحاسوب هو روحه التي تشغّل الجسد الصلب من لوحة مفاتيح وشاشة ووحدة معالجة مركزية وغيرها، فهو يشكل القاعدة التي تجري فيها تعاملاتنا مع الحاسوب، وكثيرًا ما نحتاج إلى تنفيذ تلك التعاملات أثناء عمل البرنامج، فقد نرغب في نسخ الملفات أو نقلها، أو إنشاء مجلدات أو بدء برنامج آخر، أو طباعة مستند، ومن الجيد أن نظام التشغيل يوفر واجهةً قابلةً للبرمجة وواجهةً للمستخدم، وسننظر في هذا الفصل في المزايا المتاحة لنا فيما يخص نقل بنى الملفات traversing file structures والعمل مع البيئة.

23.1.3 الاتصالات بين العمليات

تتكون أغلب برامج المبتدئين من عملية حاسوبية واحدة تعمل بشكل مستقل، لكن مع ازدياد قوة الأنظمة التي نبنيها يفضل تقسيم البرنامج إلى أجزاء منفصلة يعمل كل منها في العملية الخاصة به، فيما يسمى بتصميم العميل-الخادم client-server، أو قد يكون سبب تقسيم البرنامج أحيانًا رغبتنا في الوصول إلى خرج برنامج آخر، وسننظر في هذا الفصل في المبادئ الأساسية لتقسيم العمليات والاتصالات بينها، ثم نشرح مثالًا لكلا النوعين اللذين سنذكرهما، بما في ذلك إنشاء نسخة عميل-خادم محلية من برنامج دليل جهات الاتصال.

23.1.4 برمجة الشبكات

توجد عدة طرق ليتصل حاسوب بآخر من داخل برنامج ما، وسننظر في هذا الفصل في آلية بايثون لذلك والتي تسمى بالمقبس Socket، وسنختم بإصدار شبكي من دليل جهات الاتصال، يسمح لنا بتشغيل عملية الخادم على حاسوب بعيد.

23.1.5 كتابة عملاء الويب Web Clients

بعد إتقان البرمجة الأساسية للشبكات نصل إلى أكثر أشكالها شيوعًا، وهي الويب أو الشبكة العنكبوتية العالمية، وتوفر بايثون وحدات تسهل من برمجة الويب، وسننظر في هذا الفصل في أتمتة بعض المهام البسيطة للويب، مثل جلب معلومات من موقع ما بانتظام دون اللجوء إلى متصفح.

23.1.6 كتابة تطبيقات الويب

سننتقل في هذا الفصل من منظور مستخدم الويب إلى منظور منشئ الموقع، وسنتعلم كيفية كتابة تطبيق ويب بسيط باستخدام أبسط بروتوكول لتطبيقات الويب، وهو CGI.

23.1.7 استخدام إطارات عمل الويب

تبين بالممارسة العملية أن بايثون مناسبة للغاية لبناء أُطر عمل لتطوير تطبيقات الويب، حيث يوجد اثنا عشر إطارًا لذلك، من الحزم الصناعية كبيرة الحجم القادرة على معالجة أعداد ضخمة من المعاملات؛ إلى الأطر البسيطة التي يسهل استخدامها لكنها تفتقر إلى إمكانيات إخوتها الكبار، ويركز هذا الفصل على إطار فلاسك Flask الذي يقع في الناحية الأصغر من هذه الإطارات، لكن لا زال بالإمكان إنتاج مواقع عملية به تصلح للشركات الصغيرة، كما سنبني واجهةً أماميةً لقاعدة البيانات الخاصة بدليل جهات الاتصال، بعد شرح المبادئ الأساسية لإطار فلاسك.

23.1.8 المعالجة المتزامنة

مع ازدياد تعقيد البرامج سنرغب غالبًا في تنفيذ عدة أشياء في نفس الوقت، إذ يكون النموذج المتسلسل البسيط للتنفيذ والذي تحدثنا عنه غير كافٍ، حيث توفر بايثون عدة تقنيات في هذا الصدد، وسندرس اثنتين منها في هذا الفصل، وسنتعلم كيفية استخدام وحدتي `multiprocessing` و`threading` في بايثون، لإنشاء عمليات متوازية بسيطة في برنامج.

24. التعامل مع قواعد البيانات

سننظر في هذا الفصل من هذا الكتاب في كيفية تخزين البيانات ومعالجتها من خلال حزمة قاعدة البيانات، وقد رأينا سابقًا كيفية استخدام الملفات في تخزين كميات صغيرة من البيانات في برنامج دليل جهات الاتصال الذي مررنا عليه عدة مرات من قبل، غير أن استخدام الملفات يزداد تعقيدًا مع زيادة تعقيد البيانات نفسها، وزيادة حجمها، وتعقيد العمليات التي تُجرى عليها، مثل الفرز والبحث والترشيح filtering وغير ذلك، وتوجد عدة حزم لقواعد البيانات للعناية بإدارة الملفات، وكشف البيانات في شكل أكثر تجريدًا ويسهل التعديل عليه، بعضها عبارة عن مكتبات للشيفرات code libraries تبسط عمليات الملفات التي رأيناها من قبل مثل وحدتي pickle و shelve اللتين تأتيان مع بايثون، لكننا في هذا الفصل سندرس حزمًا أكثر قوةً صممت للتعامل مع أحجام كبيرة من البيانات المعقدة، حيث سنشرح الحزمة SQLite، وهي حزمة مجانية مفتوحة المصدر وسهلة التثبيت والاستخدام، ومع هذا فهي قادرة على معالجة حاجات أغلب المبرمجين المبتدئين والمتوسطين أيضًا، ولا يحتاج المبرمج في الغالب إلى حزمة أقوى منها إلا إذا كان يتعامل مع مجموعات كبيرة للغاية من البيانات -ملايين السجلات مثلًا-، وحتى في تلك الحالة نستطيع نقل ما تعلمناه من SQLite إلى الحزمة الجديدة.

يمكن تحميل حزمة SQLite من موقعها، فاختر حزمة سطر الأوامر -أي الأدوات- المناسبة لمنصتك، وبعد التحميل اتبع إرشادات التثبيت الموجودة في الموقع لتثبيت الحزمة.

توجد عدة بيئات تطوير لـ SQLite، غير أننا لا نحتاجها في هذا الكتاب.

سندرس في هذا الفصل ما يلي:

- مفهوم قواعد البيانات وSQL.
- إنشاء الجداول وإدخال البيانات.
- استخراج البيانات والتعديل عليها.

- ربط مجموعات البيانات بعضها ببعض.
- الوصول إلى SQL من بايثون.

24.1 مفاهيم قاعدة البيانات العلائقية

يمكن وصف قواعد البيانات العلائقية relational databases بأنها مجموعة من الجداول، حيث يمكن لخلية في جدول فيها أن تشير إلى صف في جدول آخر، وتسمى الأعمدة في تلك الجداول بالحقول fields، كما تسمى الصفوف بالسجلات records.

سيبدو جدول بيانات الموظفين في إحدى الشركات كما يلي:

ManagerID	Grade	HireDate	Name	EmpID
1020311	Foreman	20030623	Hasan Saleh	1020304
1020304	Labourer	20040302	Amin Akbar	1020305
1020304	Labourer	19991125	Ayat Othman	1020307

نلاحظ بعض المصطلحات هنا:

1. لدينا حقل معرِّف `ID` فريد يعرف كل صف، ويُعرف باسم المفتاح الأساسي primary key، ويمكن أن يكون لدينا عدة مفاتيح أخرى، لكن سيكون لدينا حقل ID دومًا لتعريف سجل ما، وهذا مفيد إذا كان لموظفين اثنين نفس الاسم مثلًا.
2. يمكن ربط صف بآخر بأن يحمل حقلٌ ما المفتاحَ الأساسي لصف آخر، فمثلًا يُحدَّد مدير الموظف بواسطة معرف المدير `ManagerID` الذي هو مرجع إلى حقل `EmpID` آخر، ويتضح بالنظر إلى البيانات التي لدينا أن لكل من `Amin` و`Ayat` المدير نفسه وهو `Hasan`، و Hasan هذا له مدير آخر لكننا لا نرى بياناته في هذا الجزء من الجدول.

ويمكن إنشاء جدول آخر للرواتب `Salary` مثلًا، فلسنا مقيدين بربط البيانات داخل جدول واحد، ويُربط هذا بالدرجة `Grade` الخاصة بكل موظف، فنحصل عندها على جدول شبيه بما يلي:

Amount	Grade	SalaryID
60000	Foreman	000010
35000	Labourer	000011

نستطيع الآن أن نبحث عن درجة موظف ما مثل `Hasan`، وسنجد أن درجته هي كبير عمال `Foreman`، وإذا بحثنا في جدول `Salary` فسنجد أن راتب هذه الدرجة هو 60000$، وإمكانية ربط صفوف الجداول معًا في علاقات هي التي تعطي قواعد البيانات العلائقية اسمها.

توجد قواعد بيانات أخرى مثل قواعد بيانات الشبكات network databases، وقواعد البيانات الهرمية hierarchical databases، وقواعد بيانات الملفات المسطحة flat-file databases، ولكن القواعد العلائقية هي أكثرها شهرةً، رغم أن الاتجاه السائد الآن في معالجة الأحجام الهائلة من البيانات هو NoSQL -أي ليست SQL فقط "Not only SQL"-، وهي قواعد بيانات تبنى في الغالب على هياكل شبكية أو هرمية.

يمكن إجراء استعلامات أعقد من تلك التي أجريناها، وسنرى كيفية ذلك فيما يلي، لكن يجب أن ننشئ قاعدة بيانات أولًا ونضع فيها بعض البيانات.

24.2 لغة الاستعلامات الهيكلية SQL

لغة الاستعلامات الهيكلية أو Structured Query Language - SQL هي أداة قياسية تُستخدم لتعديل قواعد البيانات العلائقية، ويسمى التعبير فيها عادةً استعلامًا query حتى لو لم يجلب أي بيانات، وتتكون SQL من جزأين هما: لغة تعريف البيانات Data Definition Language واختصارًا DDL، وهي مجموعة من الأوامر التي تُستخدم لإنشاء وتعديل هيكل قاعدة البيانات نفسها، وتكون عادةً خاصةً لكل قاعدة بيانات، ويوفر كل مزود قواعد بيانات صيغةً مختلفةً قليلًا لمجموعة أوامر SQL الخاصة بتعريف البيانات، أما الجزء الآخر فهو لغة تعديل البيانات Data Manipulation Language، واختصارًا DML، وهي قياسية أكثر بين قواعد البيانات، وتُستخدم لتعديل محتويات البيانات، وهي التي سنستخدمها غالبًا في التعامل مع قواعد البيانات، لذا سنتعلم بعض أوامر DDL التي تكفينا لإنشاء قاعدة بياناتنا باستخدام التعليمة `CREATE`، وتدمير جداولها باستخدام التعليمة `DROP`، ثم ننتقل بعدها إلى ملء الجداول بالبيانات، ثم نجلب تلك البيانات بطرق مختلفة باستخدام أوامر DML مثل `INSERT` و`SELECT` و`UPDATE` و`DELETE` وغيرها.

ربما تجدر الإشارة إلى مزية أخرى في SQL، وهي أنها ليست حساسةً لحالة الأحرف، على عكس بايثون وجافاسكربت، لذا يمكننا استخدام `CREATE` أو `create` أو `Create` أو حتى `CrEaTe`، فلن يهتم مفسر SQL بهذا، ومع هذا يتبع مبرمجو SQL نسقًا بحيث تكون كلمات SQL المفتاحية بأحرف كبيرة، بينما تكون المتغيرات وأسماء الجداول والحقول بأحرف صغيرة، وسنتبع هذا النسق في الشرح، لكننا ذكرنا الملاحظة أعلاه لننبه إلى أن SQL لا تهتم لحالة الأحرف.

ومن الأمور التي تجعل SQL مختلفةً عن اللغات الأخرى أنها مصممة للتعبير عن الخرج المطلوب بدلًا من إخبار الحاسوب بكيفية تنفيذه، أي أننا نخبر المفسر بما نريده فقط، وليس بالكيفية التي نريد تنفيذه بها، ونترك آلية التنفيذ للمفسر، ويستطيع المبرمجون الخبراء في قواعد البيانات أو مدراء النظم adminstrators أن يغيروا سلوك المفسر في تنفيذ المهام من خلال تعريف خطة التنفيذ أو تعديلها، غير أن هذا مستوىً متقدم خارج نطاق شرحنا.

24.3 إنشاء الجداول

نستخدم الأمر CREATE لإنشاء جدول في SQL، وهو أمر سهل ويأخذ الصورة التالية:

```
CREATE TABLE tablename (fieldName, fieldName,....);
```

نلاحظ أن تعليمات SQL تنتهي بفاصلة منقوطة ; ، وهي لا تهتم بمستويات الإزاحة أو المسافات البيضاء، لكننا سنرى اتباع اصطلاح بعينه -مع أنه ليس لازمًا، ولا تهتم SQL له إطلاقًا- ولا نستخدمه إلا لاتساق سير العمل.

لنجرب الآن إنشاء جداول الموظفين والرواتب في SQLite، حيث سيكون أول ما علينا فعله هو بدء المفسر عن طريق استدعائه مع وسيط هو اسم الملف، فإذا كانت قاعدة البيانات موجودةً فسيفتح الملف، أما إذا لم تكن موجودةً فسينشئها، وعليه فمن أجل إنشاء قاعدة بيانات الموظفين نبدأ SQLite كما يلي:

```
E:\PROJECTS\SQL> sqlite3 employee.db
```

سينشئ هذا قاعدة بيانات فارغةً اسمها employee.db، ويتركنا عند محث sqlite> لنكتب أوامر SQL، ثم ننشئ بعض الجداول كما يلي:

```
sqlite> CREATE TABLE Employee
   ...> (EmpID,Name,HireDate,Grade,ManagerID);
sqlite> CREATE TABLE Salary
   ...> (SalaryID, Grade,Amount);
sqlite>.tables
EmployeeSalary
sqlite>
```

لاحظ أننا نقلنا قائمة الحقول إلى سطر منفصل لتسهيل رؤيتها، وتُرتَّب الحقول هنا بالاسم، وليس لها أي معلومات تعرِّفها -مثل نوع البيانات- وهذا في SQLite وحدها، إذ تطلب أغلب قواعد البيانات الأخرى تحديد النوع مع الاسم، ويمكن تحديد النوع في SQLite، كما سنرى بعد قليل.

ونلاحظ أيضًا أننا تحققنا من عمل تعليمات CREATE باستخدام الأمر .tables لسرد جميع الجداول في قاعدة البيانات، وتحتوي SQLite على العديد من هذه الأوامر المنقوطة، والتي نستخدمها لجلب معلومات عن قاعدة البيانات، وللحصول على قائمة بتلك الأوامر يُستخدم الأمر ..help

يمكننا تحديد قيود على القيم، بالإضافة إلى التصريح عن أنواع البيانات في كل عمود، فمثلًا NOT NULL تعني أن القيمة إلزامية ويجب ملؤها، ونجعل عادةً حقل المفتاح الأساسي غير خالٍ NOT NULL وفريدًا UNIQUE، كما نستطيع تحديد الحقل الذي سيكون المفتاح الأساسي PRIMARY KEY.

سنترك تعريف الجدول الأساسي كما هو، وننتقل إلى التعديل في البيانات نفسها.

24.4 إدخال البيانات

أول ما نفعله بعد إنشاء الجداول هو ملؤها بالبيانات، وذلك باستخدام تعليمة INSERT في SQL، والتي لها هيكل أساسي بسيط هو:

```
INSERT INTO tablename ( column1, column2... ) VALUES ( value1,
value2... );
```

كما توجد لها صيغة أخرى تستخدم استعلامًا لاختيار البيانات من مكان آخر في قاعدة البيانات، لكن هذا مستوىً متقدم ننصح بالقراءة عنه في دليل SQLite.

يمكننا أن ندخِل بعض الصفوف في جدول موظفينا كما يلي:

```
sqlite> INSERT INTO Employee (EmpID, Name, HireDate, Grade, ManagerID)
   ...> VALUES ('1020304','Hasan
Saleh','20030623','Foreman','1020311');
sqlite> INSERT INTO Employee (EmpID, Name, HireDate, Grade, ManagerID)
   ...> VALUES ('1020305','Amin
Akbar','20040302','Labourer','1020304');
sqlite> INSERT INTO Employee (EmpID, Name, HireDate, Grade, ManagerID)
   ...> VALUES ('1020307','Ayat
Othman','19991125','Labourer','1020304');
```

وكذلك في جدول الرواتب:

```
sqlite> INSERT INTO Salary (SalaryID, Grade,Amount)
   ...> VALUES('000010','Foreman','60000');
sqlite> INSERT INTO Salary (SalaryID, Grade,Amount)
   ...> VALUES('000011','Labourer','35000');
```

وبهذا نكون قد انتهينا من إنشاء جدولين وملئهما بالبيانات الموافقة للقيم الموصوفة في المقدمة أعلاه، وسننتقل الآن إلى إجراء بعض التجارب على البيانات.

24.5 استخراج البيانات

تُستخرج البيانات من قاعدة البيانات باستخدام الأمر SELECT في SQL، والذي هو لب SQLite، وهو أعقد الأوامر هيكلًا، لذا سنبدأ بأبسط صورة ثم نضيف مزايا جديدةً أثناء العمل.

ستبدو أبسط صورة ممكنة لتعليمة SELECT كما يلي:

```
SELECT column1, column2... FROM table1,table2...;
```

فلاختيار أسماء جميع العاملين نستخدم:

```
sqlite> SELECT Name FROM Employee;
```

حيث سنحصل على قائمة بجميع الأسماء في جدول الموظفين، وهي ثلاثة أسماء في حالتنا، لكن إذا كان لدينا قاعدة بيانات كبيرة فسنحصل على معلومات أكثر مما نريد، وسنحتاج إلى تحسين بحثنا بطريقة ما للتحكم في الخرج، وتسمح لنا SQL بفعل ذلك بإضافة الشرط WHERE إلى تعليمة SELECT، كما يلي:

```
SELECT col1,col2... FROM table1,table2... WHERE condition;
```

حيث الشرط condition هو تعبير بولياني معقد وعشوائي، ويمكن أن يتضمن تعليمات SELECT متشعبةً داخله، لنستخدم الشرط WHERE لتحسين بحث الأسماء، حيث نريد البحث عن أسماء الموظفين العمال فقط، أي أصحاب الدرجة labourer.

```
sqlite> SELECT Name
   ...> FROM Employee
   ...> WHERE Employee.Grade = 'Labourer';
```

سنحصل الآن على اسمين فقط، ونستطيع توسيع الشرط باستخدام معامِلات بولیانية مثل AND وOR وNOT وغيرها، لاحظ أن استخدام الشرط = في حالة السلسلة النصية مهم، فلم يكن البحث عن labourer لينجح لولاه، وسنرى كيفية حل هذه المشكلة لاحقًا.

كما نلاحظ أننا استخدمنا الصيغة النقطية dot notation في شرط WHERE لإبراز حقل Grade، ولم يكن ذلك ضروريًا في هذه الحالة لأننا نعمل مع جدول واحد، لكن عند وجود عدة جداول محددة فسنحتاج إلى توضيح الجدول الذي ينتمي إليه الحقل، فمثلًا لنغير استعلامنا ليبحث عن أسماء جميع الموظفين الذين يحصلون على راتب أكثر من 50000$، حيث سنحتاج إلى النظر في بيانات كلا الجدولين:

```
sqlite> SELECT Name, Amount FROM Employee, Salary
   ...> WHERE  Employee.Grade = Salary.Grade
   ...> AND    Salary.Amount > '50000';
```

نلاحظ استخدام المسافات البيضاء لترتيب شكل الاستعلام، وقد وضعنا شرط FROM هذه المرة في السطر الأول، وهذا أمر تنسيقي بحت يُستخدم لتحسين القراءة، فلغة SQLite لا تهتم بالفراغات.

سنحصل هنا على اسم واحد كما توقعنا، وهو اسم كبير العمال foreman، لكن انتبه إلى أننا سنحصل على الراتب لأننا أضفنا Amount إلى قائمة الأعمدة المحددة، ولدينا شرط WHERE مكوَّن من جزأين مدمجين معًا باستخدام المعامِل البولياني AND، حيث يربط الجزء الأول الجدولين معًا عن طريق ضمان تساوي الحقول المشتركة، وهو ما يُعرف بالربط join في SQL، وقد يصبح أمر الربط معقدًا للغاية وفقًا لكل حالة، ولهذا يفضل القائمون على اعتماد المزايا الجديدة في SQL صورةً أكثر صراحةً من الربط، وهي مشروحة بالتفصيل في موقع guru99.

علينا تحديد كلا الجدولين اللذين ستظهر النتيجة منهما، لأن الحقول التي نختارها تأتي من جدولين، ويكون ترتيب أسماء الحقول هو الترتيب الذي نحصل به على البيانات مرةً أخرى، لكن ترتيب الجداول نفسه لا يهم طالما أن الحقول تظهر في تلك الجداول.

لقد حدَّدنا اثنين من أسماء الحقول الفريدة، فإذا أردنا عرض الدرجة الوظيفية Grade التي تظهر في كلا الجدولين، لكنا استخدمنا الصيغة النقطية لتحديد الجدول الذي نريده، كما يلي:

```
sqlite> SELECT Employee.Grade, Name, Amount
   ...> FROM Employee, Salary
   etc/...
```

آخر ما نريد الحديث عنه من مزايا SELECT هي القدرة على تصنيف الخرج، رغم وجود عدة مزايا أخرى يمكن الرجوع إليها في توثيق SQL، فقواعد البيانات تحتفظ بالبيانات بالترتيب الذي يسهل به إيجادها أو بالترتيب الذي أُدخلت به، وفي كلا الحالتين لا يكون هو الترتيب الذي نريد عرض البيانات به، لذا نستخدم الشرط ORDER BY الخاص بتعليمة SELECT لحل هذه المشكلة:

```
SELECT columns FROM tables WHERE expression ORDER BY columns;
```

نلاحظ أن شرط ORDER BY الأخير قد يأخذ عدة أعمدة، وهذا يمكننا من الحصول على طلبات الفرز والتصنيف الأولية والثانوية، لنستخدم ذلك الآن للحصول على قائمة بأسماء الموظفين مصنفة وفق تاريخ التوظيف HireDate:

```
sqlite> SELECT Name FROM Employee
   ...> ORDER BY HireDate;
```

لم يبقَ إلا ذكر أننا لم نستخدم شرط WHERE هنا، فإذا استخدمناه فسيأتي قبل شرط order by، لذا ورغم أن SQL لا تمانع إذا أهملنا الشرط إلا أنها تدقق كثيرًا في ترتيب الشروط داخل التعليمة.

24.6 تعديل البيانات

توجد طريقتان لتعديل البيانات في قواعد البيانات، إما بتغيير محتويات سجل واحد أو مجموعة سجلات، أو بحذف السجلات أو الجدول كاملًا، والحالة الأشهر هي تغيير محتويات سجل موجود بالفعل من خلال الأمر UPDATE في SQL، وأبسط صورة هي:

```
UPDATE tablename SET column = value WHERE condition;
```

نستطيع تجربة ذلك في قاعدة البيانات التي لدينا بتغيير راتب كبير العمال foreman إلى 70000$:

```
sqlite> UPDATE Salary
   ...> SET Amount ='70000'
   ...> WHERE Grade = 'Foreman';
```

لاحظ أن جميع البيانات التي أدخلناها واخترناها كانت أنواعًا نصيةً string types، لأن SQLite تخزن بياناتها داخليًا في سلاسل نصية، لكنها تدعم عدة أنواع مختلفة من البيانات بما فيها الأعداد، لذا كان بالإمكان تحديد الراتب في صيغة رقمية لتسهيل العمليات الحسابية، وسنرى كيفية فعل ذلك فيما يلي.

لكن المشكلة هنا أن SQL ستعدل كل الصفوف التي تطابق الشرط، فإذا كنا نريد تعديل صف واحد فقط فعلينا أن نتأكد أن الشرط WHERE يحدد حتمًا صفًا واحدًا فقط، إذ كثيرًا ما يغير المبرمجون المبتدئون دون قصد حقلًا في كل صف في الجدول أو فرع منه، لذا يجب الحذر عند استخدام أوامر التعديل لصعوبة إصلاح هذا الخطأ، ومن الأفضل التحقق من شرط WHERE بوضعه في تعليمة SELECT أولًا لمعرفة القيمة المعادة.

أما الصورة الأخرى للتغيير الجذري الذي نستطيع تنفيذه على بياناتنا فهو حذف صف أو مجموعة صفوف، باستخدام الأمر DELETE FROM، كما يلي:

```
DELETE FROM Tablename WHERE condition
```

فإذا أردنا حذف Ayat Othman من جدول الموظفين فسنكتب ما يلي:

```
sqlite> DELETE FROM Employee WHERE Name = 'Ayat Othman';
```

فإذا طابق شرطنا أكثر من صف فستُحذف تلك الصفوف جميعها، لأن SQL تنفِّذ على جميع الصفوف التي تطابق الاستعلام، فهي ليست مثل استخدام البحث المتتابع sequential search لملف ما أو سلسلة نصية باستخدام تعبير نمطي.

أما لحذف الجدول كله بمحتوياته فنستخدم الأمر DROP، ويجب توخي الحذر الشديد عند استخدام مثل هذه الأوامر التدميرية مثل DELETE وDROP لما لها من آثار قد لا يمكن إصلاحها.

24.7 ربط البيانات بين الجداول

تحدثنا عن ربط البيانات بين الجداول من قبل في القسم الخاص بتعليمة SELECT، أما الآن فننتقل إلى جزء مهم في نظرية قواعد البيانات.

24.7.1 قيود البيانات Data Constraints

تمثل الروابط بين الجداول علاقات Relations بين وحدات البيانات التي تعطي قاعدة البيانات العلائقية -مثل SQLite- اسمها، وتحتفظ قاعدة البيانات بالبيانات الخام عن الكيانات، بل تحتفظ بمعلومات عن العلاقات بينها أيضًا.

تُخزَّن المعلومات عن العلاقات في صيغة قيود لقاعدة البيانات، والتي تتصرف مثل قواعد تحدد نوع البيانات الذي يمكن تخزينها، بالإضافة إلى مجموعة قيمها الصالحة، وتطبَّق تلك القيود عندما نعرِّف هيكل قواعد البيانات باستخدام تعليمة CREATE، عادةً نعبر عن قيود كل حقل على حدة، لذا نستطيع توسيع التعريف الأساسي في تعليمة CREATE حيث نعرِّف أعمدتنا من:

```
CREATE TABLE Tablename (Column, Column,...);
```

إلى:

```
CREATE TABLE Tablename (
ColumnName Type Constraint,
ColumnName Type Constraint,
...);
```

حيث أغلب القيود:

```
NOT NULL
PRIMARY KEY [AUTOINCREMENT]
UNIQUE
DEFAULT value
```

يسهل فهم عمل القيد NOT NULL إذ يشير إلى أن القيمة يجب أن تكون موجودةً أي ليست NULL، حيث تشير القيمة NULL إلى قيمة غير محددة أو معرَّفة، وبالتالي فالعبارة NOT NULL تعني أنه يجب إعطاء قيمة لذلك الحقل، وإلا فسنحصل على خطأ ولن تُدخَل البيانات.

أما المفتاح الرئيسي PRIMARY KEY فيخبر SQLite أن تستخدم هذا العمود مفتاحًا رئيسيًا لعمليات البحث، مما يُحسّن تنفيذ عمليات بحث أسرع.

كما تعني AUTOINCREMENT أن قيمة النوع هي INTEGER ستُسنَد تلقائيًا عند كل عملية إدخال INSERT، وتتزايد القيمة بمقدار واحد، مما يوفر على المبرمج كثيرًا من حيث المحافظة على أعداد مستقلة، ولا تُستخدم الكلمة المفتاحية AUTOINCREMENT حقيقةً، وإنما تكون مضمنةً في تجميعة نوع/قيد بدمج INTEGER PRIMARY KEY، وهذه خاصية غير واضحة في توثيق SQLite إلى الحد الذي يجعلها من أبرز الأسئلة في الأسئلة الشائعة حول SQLite.

وتدل UNIQUE على أن القيمة يجب أن تكون فريدةً في العمود، فإذا حاولنا إدخال قيمة موجودة مسبقًا في عمود له قيد UNIQUE فسنحصل على خطأ ولن يُدخَل الصف، ويُستخدم قيد UNIQUE عادةً لأعمدة المفاتيح الرئيسية غير العددية.

يصاحب القيد DEFAULT قيمة دومًا، وهذه القيمة هي التي تخبرنا بما ستدخِله SQLite في ذلك الحقل إذا لم يقم المستخدم بذلك صراحةً، وهذا يفيد في أن الأعمدة التي لها القيد DEFAULT لا تكون NULL إلا نادرًا، لأن علينا ضبط القيمة NULL بصراحة إذا أردنا إنشاءها، ونستطيع رؤية مثال سريع على استخدام DEFAULT هنا:

```
sqlite> CREATE TABLE test
   ...> (id Integer PRIMARY KEY,
   ...> Name NOT NULL,
   ...> Value Integer DEFAULT 42);
sqlite> INSERT INTO test (Name, Value) VALUES ('Alan',24);
sqlite> INSERT INTO test (Name) VALUES ('Heather');
sqlite> INSERT INTO test (Name,Value) VALUES ('Linda', NULL);
sqlite> SELECT * FROM test;
1|Alan|24
2|Heather|42
3|Linda|
sqlite>
```

نلاحظ هنا كيف تعينت القيمة الافتراضية لحقل value الموافقة للاسم المدخَل Heather، وأن قيمة value للمدخل Linda غير موجودة أو NULL، وهذا اختلاف جوهري بين NOT NULL وDEFAULT، فالأول لن يسمح بقيم NULL افتراضيًا أو صراحةً، والقيد DEFAULT يمنع NULL غير المحددة، لكنه يسمح في نفس الوقت بالإنشاء المتعمد لها.

لقد استخدمنا محرف النجمة * مكان قائمة الحقل في آخر تعليمة SELECT، وهذه طريقة بسيطة لجلب جميع الحقول في الجدول، وهي ممتازة لمثل هذه التجارب، لكن يجب ألا تُستخدم في البرامج العملية لأن أي تغيير في هيكل البيانات سيتسبب في تغيير النتائج أو تعطيل أي شيفرة تعتمد على عدد أو ترتيب الحقول التي طُلبت.

توجد قيود يمكن تطبيقها على الجدول نفسه، لكننا لن نناقشها في هذا الكتاب.

أما النوع الآخر من القيود الذي نستطيع تطبيقه كما ذكرنا من قبل فهو تحديد نوع العمود، وهو يشبه مفهوم الأنواع في لغة البرمجة، ومجموعة الأنواع الصالحة في SQLite هي ما يلي:

- TEXT
- INTEGER
- REAL
- NUMERIC
- BLOB
- NULL

يجب أن تكون هذه الأنواع مفهومةً وواضحةً، باستثناء `NUMERIC` و`BLOB`، فالأول يسمح بتخزين أعداد الفاصلة العائمة floating-point numbers والأعداد الصحيحة، أما `BLOB` فيُستخدم لتخزين البيانات الثنائية، مثل الصور أو المستندات غير النصية، فهو أفضل في تخزين مثل تلك العناصر في ملفات منفصلة مع وجود مرجع reference فقط يشير إليها في قاعدة البيانات.

أما `NULL` فهو ليس نوعًا حقيقيًا، وإنما يشير إلى أننا لا نحتاج إلى تحديد نوع مطلقًا، فأغلب قواعد البيانات تأتي بمجموعة واسعة من الأنواع بما فيها النوع `DATE`، لكن SQLite لديها أسلوب غير تقليدي في الأنواع التي تجعل مثل هذه التفاصيل غير مهمة.

> أنشئ معيار SQL بواسطة لجنة شُكلت من جميع الجهات المزودة لقواعد البيانات، لذا فإن قائمة الأنواع تشمل جميع الأنواع المختلفة التي يسمح بها هؤلاء المزودون، والتي كانت موجودةً قبل وضع الأنواع القياسية، وتدعم SQLite العديد من تلك الأنواع على مستوى الصياغة syntactical، لكنها في الممارسة العملية تُسمى بأسلوب بديل alias لأفضل نوع مكافئ محلي، لذا فإن النوع VARCHAR في قاعدة Oracle مثلًا هو اسم بديل للنوع TEXT الخاص بـ SQLite، والفكرة هنا هي القدرة على استيراد سكربتات SQL إلى SQLite بأقل قدر ممكن من التغيير.

تطبق أغلب قواعد البيانات الأنواع المحددة بصرامة، لكن SQLite تتبع نهجًا أكثر ديناميكيةً ومرونةً، حيث يكون النوع المحدَّد أشبه بالتلميح أو الإرشاد hint، ويمكن تخزين أي نوع من البيانات في الجدول، وعند تحميل بيانات من نوع مختلف إلى الحقل فإن SQLite ستستخدم النوع المصرَّح عنه لتحاول تحويل البيانات إليه، فإن لم تستطع فستخزنها في صورتها الأصلية، فإذا صُرِّح عن حقل على أنه عددي `INTEGER` ثم مُررت القيمة النصية '123'، فستحول SQLite السلسلة النصية '123' إلى العدد 123، لكن إذا كانت القيمة النصية `TEXT` هي 'Amindy' فلن ينجح تحويلها إلى عدد، وستخزنها SQLite في صورتها كما هي في الحقل، وقد يسبب هذا

سلوكًا غريبًا إذا لم يكن المبرمج على علم بهذا العيب، أما بقية قواعد البيانات فتعدّ التصريح عن الأنواع قيدًا صارمًا، وتفشل عند تمرير قيمة غير مسموح بها.

24.7.2 نمذجة العلاقات مع القيود

لنر الآن كيف تساعدنا هذه القيود في صنع نماذج للبيانات والعلاقات، من خلال العودة إلى قاعدة البيانات البسيطة ذات الجدولين التي بدأنا الفصل بها:

ManagerID	Grade	HireDate	Name	EmpID
1020311	Foreman	20030623	Hasan Saleh	1020304
1020304	Labourer	20040302	Amin Akbar	1020305
1020304	Labourer	19991125	Ayat Othman	1020307

Amount	Grade	SalaryID
60000	Foreman	000010
35000	Labourer	000011

ينبغي أن يكون نوع قيمة المعرِّف `ID` عددًا صحيحًا، أي `INTEGER`، ويحمل القيد `PRIMARY KEY`، أما العمود الآخر فيجب أن يكون `NOT NULL`، باستثناء `ManagerID` الذي يجب أن يكون عددًا صحيحًا.

ونرى هنا في جدول الرواتب `Salary` أن معرِّف الراتب `SalaryID` يجب أن يكون عددًا صحيحًا `INTEGER` مع قيد `PRIMARY KEY`، كما يجب أن يكون عمود مقدار الراتب `Amount` عددًا صحيحًا، وسنطبق القيمة الافتراضية `DEFAULT` التي مقدارها 10000، وأخيرًا يجب أن يكون العمود `Grade` مقيدًا بقيد التفرد `Unique` بما أننا لا نريد أكثر من راتب واحد لكل درجة وظيفية، رغم أن هذه الفكرة غير عملية، لأن الراتب يتغير بعوامل عدة، مثل مدة العمل والدرجة، لكننا سنتجاهل هذا التفصيل الآن لتبسيط الشرح، فلو كان هذا الجدول في حالة حقيقية لسميناه جدول الدرجات وليس الرواتب.

ستبدو SQL المعدلة كما يلي:

```
sqlite> CREATE TABLE Employee (
   ...> EmpID INTEGER pRIMARY kEY,
   ...> Name NOT NULL,
   ...> HireDate NOT NULL,
   ...> Grade NOT NULL,
   ...> ManagerID INTEGER
   ...> );

sqlite> CREATE TABLE Salary (
```

```
    ...> SalaryID INTEGER PRIMARY KEY,
    ...> Grade  UNIQUE,
    ...> Amount INTEGER DEFAULT 10000
    ...> );
```

يمكنك تجربة هذه القيود بإدخال البيانات التي تتسبب في تعطيلها لترى ما يحدث، وينبغي أن ترى رسالة خطأ.

الأمر الذي تجب الإشارة إليه هنا هو أن تعليمات `INSERT` التي استخدمناها من قبل لم تعد مناسبةً، فقد أدخلنا قيمنا الخاصة من قبل لحقول ID، أما الآن فهي تُملأ تلقائيًا، فينبغي أن نهملها من البيانات المدرجة، غير أن هذا يفتح الباب لصعوبة جديدة، فكيف نملأ حقل معرِّف المدير `managerID` إذا كنا لا نعرف المعرف التوظيفي `EmpID` له؟

والإجابة هي أننا نستخدم تعليمة `SELECT` متشعبة، وقد رأينا أن ننفذ هذا على مرحلتين باستخدام حقول `NULL` أولًا، ثم استخدام تعليمة `update` بعد إنشاء جميع الصفوف، ولتجنب تكرار الكتابة قد وضعنا جميع الأوامر في بضعة ملفات، سميناها `employee.sql` لأوامر إنشاء الجداول، و`employee.dat` لتعليمات الإدراج، وهذا يشبه إنشاء ملف سكربت بايثون ذي الامتداد `.py` لتوفير كتابة كل الأوامر في مِحَث `>>>`.

سيكون ملف `employee.sql` كما يلي:

```
DROP TABLE IF EXISTS Employee;
CREATE TABLE Employee (
EmpID INTEGER PRIMARY KEY,
Name NOT NULL,
HireDate NOT NULL,
Grade NOT NULL,
ManagerID INTEGER
);

DROP TABLE IF EXISTS Salary;
CREATE TABLE Salary (
SalaryID INTEGER PRIMARY KEY,
Grade UNIQUE,
Amount INTEGER DEFAULT 10000
);
```

نلاحظ هنا أننا أسقطنا الجداول -أي حذفناها- قبل إنشائها، فأمر DROP TABLE الذي ذكرناه من قبل يحذف الجدول وأي بيانات موجودة فيه، وهذا يضمن أن قاعدة البيانات ستكون خاليةً نظيفةً قبل أن ننشئ جدولنا الجديد، كما أضفنا شرط IF EXISTS الذي يمنعنا من محاولة حذف جدول حذف سابقًا.

أما ملف employee.dat فسيكون كما يلي:

```
INSERT INTO Employee (Name, HireDate, Grade, ManagerID)
  VALUES ('Hasan Saleh','20030623','Foreman', NULL);
INSERT INTO Employee (Name, HireDate, Grade, ManagerID)
  VALUES ('Amin Akbar','20040302','Labourer',NULL);
INSERT INTO Employee (Name, HireDate, Grade, ManagerID)
  VALUES ('Ayat Othman','19991125','Labourer',NULL);

UPDATE Employee
SET ManagerID = (SELECT EmpID
                 FROM Employee
                 WHERE Name = 'Hasan Saleh')
WHERE Name = 'Amin Akbar' OR
      Name = 'Ayat Othman';

INSERT INTO Salary (Grade, Amount)
       VALUES('Foreman','60000');
INSERT INTO Salary (Grade, Amount)
       VALUES('Labourer','35000');
```

نلاحظ هنا استخدام تعليمة SELECT المضمنة في أمر UPDATE، وكذلك استخدامنا لأمر UPDATE واحد لتعديل صفَّي الموظف باستخدام شرط OR البوليـاني، ويمكن إضافة موظفين أكثر مع نفس المدير بسهولة بتوسيع شرط OR، وهذا مثال للمشاكل التي قد نواجهها عند ملء قاعدة بيانات للمرة الأولى، إذ سنحتاج إلى تخطيط ترتيب التعليمات بعناية لضمان توفير البيانات لكل صف يجب أن يحتوي على قيمة مرجعية إلى جدول آخر، وذلك من أجل الإشارة إليها، وهذا أشبه بالبدء من أوراق شجرة ما إلى جذعها، إذ يجب إنشاء وإدراج البيانات التي لا تحوي مراجع في البداية، ثم البيانات التي تشير مرجعيًا إلى تلك البيانات الأولى، وهكذا. فإذا أضفنا البيانات بعد الإنشاء الأولي فسنحتاج إلى استخدام استعلامات للتحقق من وجود البيانات التي نحتاج إليها، وإضافتها إن لم تكن موجودةً، وهنا تبرز أهمية لغة مثل بايثون.

ثم نشغل هذه الملفات من محث sqlite كما يلي:

```
sqlite> .read employee.sql
```

```
sqlite> .read employee.dat
```

تأكد أولًا من حل أي مشاكل تتعلق بمسارات الملفات، إما بتشغيل Sqlite من نفس مجلد سـكربتات SQL كما فعلنا هنا، أو بتوفير المسار الكامل إلى السكربت.

والآن لنجرب استعلامًا للتحقق من عمل هذه الملفات كما يجب:

```
sqlite> SELECT Name FROM Employee
   ...> WHERE Grade IN
   ...> (SELECT Grade FROM Salary WHERE amount >50000)
   ...> ;
Hasan Saleh
```

يبدو أننا نجحنا هنا، إذ أن Hasan Saleh هو الموظف الوحيد الذي يتقاضى أكثر من 50000$، ونلاحظ أننا استخدمنا الشرط IN مع تعليمة SELECT مضمنة أخرى، وهذه صورة مختلفة عن استعلام مشابه أجريناه سـابقًا باستخدام وصلة بين الجداول cross-table join، ورغم أن كلا التقنيتين ستعملان إلا أن طريقة الوصلة أسرع.

24.8 العلاقات التعددية بين الجداول

أحد السيناريوهات التي لم نتحدث عنها هو ربـط جـدولين معًـا بعلاقـات تعدديـة، أي يُربـط صـف في أحـد الجدولين بعدة صفوف في الجدول الآخر، في نفس الوقت الذي يمكن ربط صف من الجدول الآخر بعدة صفوف من الجدول الأول، فمثلًا لنفرض أننا نكتب قاعدة بيانات لـدعم دار نشـر للكتب، حيث سـيكون لـدينا قائمـة من المؤلفين وقائمة من الكتب، وسيكتب كل مؤلف كتابًا أو أكثر، وفي نفس الوقت قد يكون للكتـاب الواحـد عـدة مؤلفين، فكيف نعبر عن هذه العلاقات في قاعدة بيانات؟

الجواب هنا هو تمثيل العلاقة بين الكتب والمؤلفين في جدول مستقل بذاته، ويُدعى هـذا الجـدول بجـدول التقاطع intersection table أو جدول الربط mapping table، وكل صف في ذلك الجدول يمثـل علاقـةً من النوع كتاب/مؤلف، فقد يكون لكل كتاب عدة علاقات كتاب/مؤلف، لكن لكل علاقـة كتابًـا واحـدًا ومؤلفًـا واحـدًا، وبذلك نكون قد حولنا علاقة متعدد-متعـدد إلى علاقـتي واحـد-متعـدد، وبمـا أننـا نعـرف كيـف نبـني مثـل هـذه العلاقات باستخدام المعرِّفات، فلنر ذلك عمليًا:

```
DROP TABLE IF EXISTS author;
CREATE TABLE author (
ID INTEGER PRIMARY KEY,
Name TEXT NOT NULL
);
```

```
DROP TABLE IF EXISTS book;
CREATE TABLE book (
ID INTEGER PRIMARY KEY,
Title TEXT NOT NULL
);

DROP TABLE IF EXISTS book_author;
CREATE TABLE book_author (
bookID INTEGER NOT NULL,
authorID INTEGER NOT NULL
);

INSERT INTO author (Name) VALUES ('Jane Austin');
INSERT INTO author (Name) VALUES ('Grady Booch');
INSERT INTO author (Name) VALUES ('Ivar Jacobson');
INSERT INTO author (Name) VALUES ('James Rumbaugh');

INSERT INTO book (Title) VALUES('Pride & Prejudice');
INSERT INTO book (Title) VALUES('Emma');
INSERT INTO book (Title) VALUES('Sense & Sensibility');
INSERT INTO book (Title) VALUES ('Object Oriented Design with
Applications');
INSERT INTO book (Title) VALUES ('The UML User Guide');

INSERT INTO book_author (BookID,AuthorID) values (
(SELECT ID FROM book WHERE title = 'Pride & Prejudice'),
(SELECT ID FROM author WHERE Name = 'Jane Austin')
);

INSERT INTO book_author (BookID,AuthorID) VALUES (
(SELECT ID FROM book WHERE title = 'Emma'),
(SELECT ID FROM author WHERE Name = 'Jane Austin')
);

INSERT INTO book_author (BookID,AuthorID) VALUES (
(SELECT ID FROM book WHERE title = 'Sense & Sensibility'),
(SELECT ID FROM author WHERE Name = 'Jane Austin')
```

```
);

INSERT INTO book_author (BookID,AuthorID) VALUES (
(SELECT ID FROM book WHERE title = 'Object Oriented Design with
Applications'),
(SELECT ID FROM author WHERE Name = 'Grady Booch')
);

INSERT INTO book_author (BookID,AuthorID) VALUES (
(SELECT ID FROM book WHERE title = 'The UML User Guide'),
(SELECT ID FROM author WHERE Name = 'Grady Booch')
);

INSERT INTO book_author (BookID,AuthorID) VALUES (
(SELECT ID FROM book WHERE title = 'The UML User Guide'),
(SELECT ID FROM author WHERE Name = 'Ivar Jacobson')
);

INSERT INTO book_author (BookID,AuthorID) VALUES (
(SELECT ID FROM book WHERE title = 'The UML User Guide'),
(SELECT ID FROM author WHERE Name = 'James Rumbaugh')
);
```

يمكن أن نجـرب الآن بعض الاسـتعلامات لـنرى كيـف سـتعمل، فمثلًا لنبحث عمـا نشـرته جين أوسـتن Jane Austin من كتب:

```
sqlite> SELECT title FROM book, book_author
   ...> WHERE book_author.bookID = book.ID
   ...> AND book_author.authorID = (SELECT ID FROM author
   ...>                             WHERE name = "Jane Austin");
```

لعل الأمر صار معقدًا قليلًا، لكن الفكرة ستتضح مع التكرار والتدريب، ولاحظ كيف نحتـاج إلى إدراج كـل من الجدولين المشار إليهما `book` و`book_author` في قائمة الجداول بعد `SELECT`، أما الجدول الثـالث `author` فليس موجودًا هناك لأنه مدرج مقابل تعليمة `SELECT` الخاصة به.

لنجرب الآن بالطريقة المعاكسة، أي لنر من ألف كتاب The UML User Guide:

```
sqlite> SELECT name FROM author, book_author
```

```
    ...> WHERE book_author.authorID = author.ID
    ...> AND book_author.bookID = (SELECT ID FROM book
    ...>                           WHERE title = "The UML User Guide");
```

بالنظر إلى تلك الشيفرة سنجد تطابق هيكل الاستعلامين، فلم نغير إلا أسماء الحقل والجدول.

سنعود الآن إلى مثال دليل جهات الاتصال الذي تركناه في فصل التعامل مع الملفات، ويُفضل الرجوع لهذا الفصل قبل متابعة القراءة لنرى كيف سنحوله من تخزين مبني على الملفات إلى قاعدة بيانات كاملة.

24.9 إعادة النظر في دليل جهات الاتصال

في دليل جهات الاتصال الذي كتبناه من قبل وبنيناه على الملفات، استخدمنا قاموسًا فيه اسم الشخص هو المفتاح، وعنوانه عنصر بيانات وحيد، ولا بأس بهذا إذا كنا نعرف الاسم الذي نريده، أو إذا أردنا تفاصيل العنوان كلها، لكن ماذا لو أردنا جميع الأسماء الموجودة في مدينة بعينها؟ أو كل من اسمه Hasan؟

يمكننا كتابة شيفرة بايثون لكل استعلام، لكن مع زيادة عدد الاستعلامات الخاصة سيزيد الجهد المطلوب لكتابة تلك الشيفرات، وهنا يأتي دور قواعد البيانات حيث يمكننا فيها إنشاء استعلامات ديناميكيًا باستخدام SQL، وسيبدو دليل جهات الاتصال قاعدة بيانات من جدول واحد، ويمكن تقسيم البيانات إلى عنوان وشخص، ثم ربطهما معًا، فقد يكون لدينا عدة أصدقاء يعيشون في نفس المنزل، لكننا سنلتزم بالتصميم الأصلي ونستخدم جدولًا بسيطًا، إلا أننا سنقسم البيانات إلى عدة حقول، حيث سنقسم الاسم إلى الاسم الأول والاسم الأخير، والعنوان إلى أجزائه المكونة له، بدلًا من أن يكون لدينا هيكل لاسم واحد وعنوان واحد، وعلى الرغم من كثرة الدراسات التي أجريت حول أفضل الطرق لتقسيم هذه البيانات إلا أننا لم نصل إلى إجابة محددة، لكنها جميعًا تتفق في أن حقل العنوان الواحد فكرة سيئة لأنها تفتقر للمرونة، وستكون حقول جدول قواعد البيانات والقيود التي نريد تطبيقها كما يلي:

Field Name	Type	Constraint
First Name	String	Primary Key
Last Name	String	Primary Key
House Number	String	NOT NULL
Street	String	NOT NULL
District	String	
Town	String	NOT NULL
Post Code	String	NOT NULL
Phone Number	String	NOT NULL

نلاحظ عدة أمور هنا:

1. لدينا مفتاحان رئيسيان `primary keys` وهذا غير مسموح به، وسنتعامل معه بعد قليل.
2. جميع البيانات من نوع `TEXT` رغم أن `House Number` قد يكون عددًا صحيحًا `INTEGER`، إلا أن أرقام المنازل تتضمن أحرفًا، لذا يجب استخدام `TEXT`.
3. الحقل الاختياري الوحيد هو الحقل `district`.
4. الرمز البريدي محدد الصيغة للغاية، لكنه يختلف وفقًا لكل دولة، وهذا يعني أن علينا أن نجعله من النوع `TEXT` ليناسب جميع الاحتمالات.
5. رغم أن رقم الهاتف `Phone Number` قد يبدو مناسبًا لوضع قيد `UNIQUE` إلا أن هذا لن يسمح بوجود شخصين يتشاركان نفس رقم الهاتف، وهي حالة محتملة.

بالعودة إلى النقطة الأولى -وجود مفتاحين رئيسيين- وهذا غير مسموح في SQL، لكن نستطيع جمع عمودين معًا في ما يسمى بالمفتاح المركب composite key، والذي يسمح بمعاملتهما مثل قيمة واحدة فيما يخص تعريف الصف، وعلى ذلك يمكن إضافة سطر في نهاية تعليمة `create table` ليجمع الاسم الأول `FirstName` و`LastName` في مفتاح رئيسي واحد، وسيبدو ذلك كما يلي:

```
CREATE TABLE address (
FirstName NOT NULL,
LastName NOT NULL,
...
PhoneNumber NOT NULL,
PRIMARY KEY (FirstName,LastName)
);
```

نلاحظ هنا السطر الأخير `PRIMARY KEY (FirstName,LastName)` الذي يحوي الأعمدة التي نريد استخدامها لتكون مفتاحًا مركبًا، وهو مثال على قيد قائم على الجدول table-based constraint، غير أن هذه الفكرة غير سديدة، فإذا كنا نعرف شخصين بنفس الاسم فلن نستطيع تخزينهما معًا، ولن نخزن إلا واحدًا فقط منهما، وسنتعامل مع هذا بتعريف حقل `integer primary key` لتعريف جهات اتصالاتنا تعريفًا فريدًا رغم أننا لن نستخدم ذلك في الاستعلامات إلا نادرًا.

نعرف كيفية التصريح عن قيد `INTEGER PRIMARY KEY` حيث فعلنا ذلك في مثال الموظف، ونستطيع تحويل ذلك مباشرةً إلى سكربت إنشاء بيانات SQLite كما يلي:

```
-- احذف الجداول إذا كانت موجودة من قبل وأعد إنشاءها.
-- استخدم القيود لتحسين كفاءة البيانات.
DROP TABLE IF EXISTS address;
```

```
CREATE TABLE address (
ContactID INTEGER PRIMARY KEY,
First NOT NULL,
Last NOT NULL,
House NOT NULL,
Street NOT NULL,
District,
Town NOT NULL,
PostCode NOT NULL,
Phone NOT NULL
);
```

السطران الأولان في الشيفرة السابقة ما هما إلا تعليقات، فأي شيء متبوع بشرطتين -- هنا يُعد تعليقًا في SQL، كما في حالة رمز # في بايثون.

نلاحظ أننا لم نعرّف النوع لأن TEXT هو النوع الافتراضي في SQLite، فإذا أردنا تحويل هذا المخطط أو تخطيط الجدول -أو نقله port بالاصطلاح الحاسوبي- إلى قاعدة بيانات أخرى فسيتوجب علينا إضافة بعض المعلومات.

أما الخطوة التالية فهي تحميل بعض البيانات إلى الجدول لنبدأ تنفيذ الاستعلامات، وسنترك ذلك تدريبًا للقارئ -باستخدام رمز الإدراج أعلاه قالبًا-، لكن سنستخدم مجموعة البيانات التالية في الأمثلة أدناه:

Phone	PostCode	Town	District	Street	House	Last	First
567890 01234	ABC123	MyTown	SomePlace	Any Street	42	Akbar	Mona
543129 01234	ABC234	MyTown	SomePlace	Any Street	17	Masoud	Jamil
456459 01234	ABC345	Metropolis	Hotspot	Crypt Drive	9	Mohammad	Yousef
567890 01234	ABC123	MyTown	SomePlace	Any Street	42	Akbar	Yasein
784310 01394	DEF174	AnyTown		Double Street	12A	Akbar	Yasein
784310 01394	DEF174	AnyTown		Double Street	12A	Akbar	Amal

لدينا الآن بعض البيانات ونريد إجراء التجارب عليها، لنرى كيفية استخدام الإمكانيات الموجودة في SQL لاستخراج البيانات بطرق لم نكن لنحلم بها في مثال القاموس المبني على الملفات في بايثون.

24.9.1 من يعيش في هذا الشارع؟

هذا الاستعلام بسيط ومباشر نوعًا ما، لأننا قسمنا بيانات العنوان إلى حقول منفصلة، فلو لم نفعل ذلك لاحتجنا إلى كتابة شيفرة تحليل لاستخراج بيانات الشارع، وهذا أكثر تعقيدًا لا شك، وسيبدو استعلام SQL الذي نريده كما يلي:

```
sqlite> SELECT First,Last FROM Address
   ...> WHERE Street = "Any Street";
```

24.9.2 من يحمل اسم Akbar؟

هذا أيضًا تعبير SELECT/WHERE بسيط في SQL:

```
sqlite> SELECT First,Last FROM Address
   ...> WHERE Last = "Akbar";
```

24.9.3 ما هو رقم هاتف Yasein؟

وهذا أيضًا استعلام بسيط إلا أننا سنحصل على عدة نتائج:

```
sqlite> SELECT First,Last, Phone FROM Address
   ...> WHERE First LIKE "Yas%";
```

نلاحظ أننا استخدمنا LIKE في شرط WHERE، وهذا يستخدم أسلوب الموازنة الخاص بمحرف البدل wild card، ويتجاهل حالة الأحرف، لاحظ أن رمز محرف البدل في SQL هو % بدلًا من محرف * الشائع، ونتيجةً لهذا نحصل على مطابقة أكثر مرونةً من التساوي الذي يتطلب تطابقًا تامًا، ونلاحظ أننا لو استخدمنا Y% فقط في محرف البدل لحصلنا على Yosef في النتائج أيضًا.

24.9.4 ما هي الأسماء المتكررة؟

هذا استعلام أكثر تعقيدًا، وسنحتاج إلى اختيار مداخل الجدول التي تكررت أكثر من مرة، وهنا يبرز دور المفتاح ContactID:

```
sqlite> SELECT DISTINCT A.First, A.Last
   ...> FROM Address AS A, Address AS B
   ...> WHERE A.First = B.First
   ...> AND A.Last = B.Last
   ...> AND NOT A.ContactID = B.ContactID;
```

نستخدم هنا بعض المزايا الجديدة، حيث نضيف A وB -وهما اسمان بديلان aliases- إلى الجداول في شرط FROM، كما نضيف أسماءً بديلةً للقيم الناتجة أيضًا لتقليل الكتابة، ونستخدم هذه الأسماء عند الإشارة إلى الحقول الناتجة باستخدام الصيغة النقطية المعتادة، يمكن استخدام الاسم البديل في أي استعلام، لكننا مجبرون على استخدامه هنا لأننا نستخدم نفس الجدول Address في المرتين -ومن ثم نضمه إلى نفسه-، لذا نحتاج إلى اسمين بديلين للتمييز بين النسختين في شرط where، كما نضيف الكلمة المفتاحية DISTINCT التي تحذف أي نتائج مكررة.

وخلاصة الفقرات السابقة أن الاستعلام يبحث عن الصفوف الـتي لهـا نفس الاسـم الأول والأخـير، لكن لهـا معرِّف جهة اتصال `ContactID` مختلف، ثم يحذف الصفوف المتشابهة قبل عرض النتائج.

لمحث SQLite التفاعلي نفس قوة محث بايثون من حيث القدرة على تطوير استعلامات معقدة مثل هـذا، فقـد نبـدأ باسـتعلام بسـيط ثم نزيـد التعقيـد لاحقًـا، فمثلًا آخـر جـزء أضـفناه إلى الاسـتعلام الأخـير كـان كلمـة `DISTINCT`، على الرغم من أنها الكلمة الثانية فيه.

24.10 الوصول إلى SQL من بايثون

توفر SQLite واجهة برمجة تطبيقـات API تتكـون من عـدد من الـدوال القياسـية الـتي تسـمح للمـبرمجين بتنفيذ جميع العمليات الممكنة في محث SQL، وقد كُتبت API الخاصة بـ SQLite بلغة C، لكن توجد مغلِّفـات لها للغات الأخرى، بما في ذلك بايثون.

24.10.1 الاتصالات Connections

أول ما نحتاج إليه للتعامل مـع قواعـد البيانـات هـو الاتصـالات، ويـأتي الاسـم من حقيقـة أن أغلب قواعـد البيانات ما هي إلا برامج مخدمات server programs تعمل على حاسوب مركزي في مكـان مـا في الشـبكة، ونريد أن نتصل بها، عادةً بواسطة التسجيل باسم مستخدم وكلمة مرور.

ورغم أن SQLite ما هي إلا ملف موجود في نظام الملفات لدينا، إلا أن API تطلب الاتصـال بـه -أي فتحـه- للحفاظ على اتساق العمليات.

24.10.2 المؤشرات Cursors

من المهم عند استخدام قاعدة بيانات من داخل برنامج ما أن نعرف كيفية الوصـول إلى الصـفوف المتعـددة التي يُحتمل أن تعيدها تعليمة `SELECT`، وذلك باستخدام ما يُعرف بمؤشرات SQL أي SQL cursors، والمؤشر هنا يشبه تسلسل بايثون في القدرة على الوصول فيه إلى صف واحد في كل مرة، وعليـه فـإن اسـتخراج بياناتنـا إلى مؤشر ثم استخدام حلقة تكرارية loop للوصول إليه يمكّننا من معالجة تجميعات كبيرة من البيانات.

ولا يخزن المؤشر كل البيانـات الناتجـة، وإنمـا يخـزن مرجعًـا إليهـا يُحفـظ في جـدول مـؤقت داخـل قاعـدة البيانات، وهذا أقل أهميةً بالنسبة لـ SQLite التي تكون في الغالب على نفس الحاسـوب الـذي عليـه البرنـامج، لكنه مهم عند التعامل مع قاعدة بيانات متصلة بشبكة العميل/الخادم.

يسمح المؤشر بجلب البيانات بكميات صغيرة، وهذا أسر ع وأوفر للذاكرة من نسـخ جميـع البيانـات الناتجـة إلى البرنامج مرةً واحدةً، وتبرز أهمية ذلك عند العمل على قواعد بيانات كبيرة جدًا أحجامها بالجيجابايت، لكن هذا يعني أننا عندما نعالج جزءًا من البيانات الناتجـة في وقت مـا، فيجب ألا ننفـذ أي اسـتعلامات أخـرى باسـتخدام نفس المؤشر، وإلا فسنفقد الوصول إلى مجموعة النتائج الأصلية، لذا يجب إنشاء مؤشر جديد للاستعلام الجديد.

أما في البرامج الصغيرة -مثل التي لدينا- فنستطيع نسخ جميع البيانات إلى هيكل بيانات بايثون لحل المشكلة، لكن يجب الانتباه إلى أننا قد نحتاج إلى عدة مؤشرات في البرامج الكبيرة.

24.11 الواجهة البرمجية لقواعد البيانات DB API

يمكن قراءة توثيق الإصدار الأخير من واجهة برمجة التطبيقات لقواعد البيانات DB API في بايثون على موقع بايثون في قسم Database Topic Guide، ويجب قراءته بعناية خاصةً عند برمجة قواعد بيانات ذات أهمية باستخدام بايثون.

24.11.1 تثبيت تعريفات SQLite

تأتي تعريفات SQLite في مكتبة بايثون القياسية افتراضيًا، فإذا أردنا استخدام قاعدة بيانات أخرى، مثل SQL Server الخاصة بمايكروسوفت أو MySQL أو Oracle، فسنحتاج إلى تنزيل الوحدات المناسبة لكل منها وتثبيتها، ويمكن الحصول على تعريفات أغلب قواعد البيانات المشهورة من `pip` أو في ملفات تنفيذية.

سيكون أمر استيراد SQLite كما يلي:

```
    import sqlite3
```

24.11.2 استخدام DBI الأساسي

لن نغطي جميع المزايا الموجودة في DBI، لكننا سنذكر ما يكفي ليمكننا من الاتصال بقاعدة بياناتنا، وتنفيذ بعض الاستعلامات، ومعالجة النتائج، وسنختم بإعادة كتابة برنامج دليل جهات الاتصال ليستخدم قاعدة بيانات جهات الاتصال بدلًا من ملف نصي.

```
>>> db = sqlite3.connect('address.db')

>>> cur = db.cursor()
>>> cur.execute('SELECT * FROM address')

>>> print( cur.fetchall() )
```

ستكون النتيجة ما يلي:

```
[(1, 'Mona', 'Akbar', '42', 'Any Street', 'SomePlace', 'MyTown',
'ABC123', '01234 567890'),
 (2, 'Jamil', 'Masoud', '17', 'Any Street', 'SomePlace', 'MyTown',
'ABC234', '01234 543129'),
```

```
 (3, 'Yousef', 'Mohammad', '9', 'Crypt Drive', 'Hotspot',
'Metropolis', 'ABC345', '01234 456459'),
 (4, 'Yasein', 'Akbar', '42', 'Any Street', 'SomePlace', 'MyTown',
'ABC123', '01234 567890'),
 (5, 'Yasein', 'Akbar', '12A', 'Double Street', '', 'AnyTown',
'DEF174', '01394 784310')]
```

تعيد `cursor.fetchball()` قائمةً من الصفوف tuples، وهذا مشابه لما بدأنا به في الفصل الخامس: البيانات وأنواعها، ونستطيع استخدام هذه القائمة في برنامجنا كما لو قرأناها من ملف مستخدمين قاعدة البيانات آليةً ثابتةً، غير أن قوة قواعد البيانات الحقيقية تكمن في قدرتها على تنفيذ استعلامات معقدة باستخدام `SELECT`.

24.12 دليل جهات الاتصال

لا زال برنامج دليل جهات الاتصال الخاص بنا يعتمد على سطر الأوامر وليست له واجهة رسومية، فإذا أردنا إضافة واجهة مستخدم فيجب أن نعيد هيكلة الشيفرة refactoring لفصل الوظائف عن العرض، على كل سنضيف واجهة ويب للبرنامج.

لن نشرح كل تفصيل في الشيفرة هنا، فيجب أن يستطيع القارئ في هذا المستوى أن يفهمها بنفسه، لكننا سنركز على بعض النقاط ونناقشها.

```
################################
# Addressbook.py
#
# Author: A J Gauld
#
'''
    Build a simple addressbook using
    the SQLite database and Python
    DB-API.
'''
################################

# set up the database and cursor
import sqlite3
dbpath = "D:/DOC/Homepage/Tutor2/sql/"
def initDB(path):
    try:
```

```
            db = sqlite3.connect(path)
            cursor = db.cursor()
        except sqlite3.OperationalError:
            print( "Failed to connect to database:", path )
            db,cursor = None,None
            raise
        return db,cursor

    # Driver functions
    def addEntry(book):
        first = input('First name: ')
        last =  input('Last name: ')
        house = input('House number: ')
        street = input('Street name: ')
        district = input('District name: ')
        town =  input('City name: ')
        code =  input('Postal Code: ')
        phone = input('Phone Number: ')
        query = '''INSERT INTO Address
                   (First,Last,House,Street,District,Town,PostCode,Phone)
                   VALUES (?,?,?,?,?,?,?,?)'''

        try:
           book.execute(query,(first, last, house, street, district, town,
    code, phone))
        except sqlite3.OperationalError:
           print( "Insert failed" )
           raise
        return None

    def removeEntry(book):
        name  = input("Enter a name: ")
        names = name.split()
        first = names[0]; last = names[-1]
        try:
           book.execute('''DELETE FROM Address
                        WHERE First LIKE ?
```

```
                    AND Last LIKE ?''',(first,last))
    except sqlite3.OperationalError:
       print( "Remove failed" )
       raise
    return None

def findEntry(book):
    validFields = ('first','last','house','street',
                   'district','town','postcode','phone')
    field = input("Enter a search field: ")
    value = input("Enter a search value: ")
    if field.lower() in validFields:
       query = '''SELECT
first,last,house,street,district,town,postcode,phone
                  FROM Address WHERE %s LIKE ?''' % field
    else: raise ValueError("invalid field name")
    try:
        book.execute(query, (value,) )
        result = book.fetchall()
    except sqlite3.OperationalError:
       print( "Sorry search failed" )
       raise
    else:
        if result:
           for line in result:
               print( line )
        else: print("No matching data")
    return None


def testDB(database):
    database.execute("SELECT * FROM Address")
    print( database.fetchall() )
    return None

def closeDB(database, cursor):
    try:
```

```
        cursor.close()
        database.commit()
        database.close()
    except sqlite3.OperationalError:
        print( "problem closing database..." )
        raise

# User Interface functions
def getChoice(menu):
    print( menu )
    choice = input("Select a choice(1-4): ")
    return choice

def main():
    theMenu = '''
Add Entry
Remove Entry
Find Entry
Test database connection

Quit and save
    '''

    try:
        theDB, theBook = initDB(dbpath + 'address.db')
        while True:
            choice = getChoice(theMenu)
            if choice == '9' or choice.upper() == 'Q':
                break
            if choice == '1' or choice.upper() == 'A':
                addEntry(theBook)
            elif choice == '2' or choice.upper() == 'R':
                removeEntry(theBook)
            elif choice == '3' or choice.upper() == 'F':
                try: findEntry(theBook)
                except: ValueError: print("No such field name"))
```

```
            elif choice == '4' or choice.upper() == 'T':
                testDB(theBook)
            else: print( "Invalid choice, try again" )

        except sqlite3.OperationalError:
            print( "Database error, exiting program." )
            # raise
        finally:
            closeDB(theDB,theBook)

if __name__ == '__main__': main()
```

نلاحظ عدة أمور هي:

1. استخدمنا الشرط `try/except` لالتقاط أي أخطاء في قاعدة البيانات، وبما أن الخطأ هو نوع مخصص معرَّف داخل وحدة `sqlite3`، فسنحتاج إلى سبقه باسم الوحدة.
2. استخدمنا الكلمة المفتاحية `raise` بعد طباعة رسالة الخطأ، فنتج عن ذلك رفع الاستثناء الأصلي إلى المستوى التالي، والذي هو `main` في حالتنا، حيث التقط وطُبعت رسالة أخرى.
3. استخدمنا كذلك محرف البدل `?` في سلاسل الاستعلامات لتحمل متغيرات البيانات، وهذا يشبه محدِّدات `%` المستخدمة في صياغة السلاسل النصية، لكن تُدخل القيم هنا جزءًا من تنفيذ الاستعلام بواسطة `book.execute`، حيث نمرر صف القيم المدرجة وسيطًا ثانيًا، وميزة هذا تكمن في التحقق الأمني من قيم الدخل مما يحسِّن من أمان الشيفرة، فإذا لم نتحقق من الدخل، أو استخدمنا الصياغة القياسية للسلاسل النصية؛ فقد يكتب أحد المستخدمين شيفرةً بدلًا من قيمة الدخل، لتدخل تلك الشيفرة إلى الاستعلام وتخرب قاعدة البيانات، وهذا يُعرف بهجمات الحقن injection attack في دوائر الأمن الرقمي، وهو أحد أكثر الاختراقات الأمنية شهرةً في الويب هذه الأيام.
4. نستخدم كلًا من اسم الحقل وقيمة البحث حقول إدراج في الدالة `findEntry`، مما يجعل دالة البحث أكثر تنوعًا، ولولاه لاحتجنا إلى دالة بحث لكل معيار بحث criteria وهذا أمر مرهق جدًا. لكن توجد مشكلة هنا سببها أن آلية معامِلات SQLite تعمل للقيم فقط، وليس لعناصر SQL، مثل أسماء الحقول أو الجداول، ولحل هذا نحتاج إلى استخدام صياغة السلاسل النصية في بايثون لإدراج اسم الحقل، ونحتاج إلى التحقق من أن اسم الحقل هو أحد الأسماء المعرّفة قبل إدراجه في الاستعلام، لنضمن أن الاستعلام آمن، فإذا لم يكن الحقل صالحًا فسنرفع استثناء بايثون قياسي من النوع `ValueError`، ثم نحتاج إلى التقاط ذلك في دالة `main()`، ونلاحظ أن الاستعلام يستخدم تعبير البحث `LIKE` الذي يسمح لنا باستغلال خيار محرف البدل `%` في SQL في سلسلة البحث الخاصة بنا.

5. تحتوي الدالة `closeDB` على استدعاء `commit`، وهذا يجبر قاعدة البيانات على كتابة جميع التغييرات في الجلسة الحالية إلى الملف، ويمكن النظر إليها على أنها تشبه التابع `file.flush`، فهي تنهي العملية transaction نوعًا ما.

6. تغلف الدالة `main` كل شيء داخل بنية `try/except/finally`، ويلتقط الشرط `except` الاستثناءات التي رفعتها دوال المستوى الأدنى كما ذكرنا أعلاه، لكن نلاحظ أنه يحوي تعليمة `raise` التي أُخرجت من التنفيذ بوضع علامة تعليق قبلها، لأن التعقب الخلفي الكامل للخطأ مفيد جدًا في تنقيح الأخطاء رغم كونه غير مفضل للمستخدم من حيث قابلية القراءة، لذلك يمكن إلغاء التعليق من `raise` التي في المستوى الأعلى أثناء التطوير لنحصل على تعقب خلفي كامل على الشاشة، وبعد حلّ جميع العلل البرمجية bugs نعيد التعليق إلى تعليمة `raise` مرةً أخرى لاستعادة العرض النهائي للبرنامج، وهذا الأسلوب ليس مقصورًا على قواعد البيانات وحدها، بل يمكن استخدامه في أي برنامج يحتمل رفع أخطاء كثيرة فيه ولا نريد إظهار ذلك للمستخدمين، لكننا نحن المكورون نريد أن نراها.

7. يُستخدم الشرط `finally` في دالة `main` لضمان إغلاق قاعدة البيانات بأناقة بغض النظر قابلنا أخطاءً أم لا، وهذا يقلل خطر تخريب البيانات.

24.12.1 ملاحظة عن الأمن الرقمي

ذكرنا أعلاه أن استخدام DB API لمحددات `?` بدلًا من استخدام `%` المعتادة كان بداعي الأمان، ويمكن استخدام صياغة السلاسل النصية المعتادة في الشيفرة، وسنجد أنها ستعمل، مما يغري باستخدامها، غير أنه يُفضل عدم فعل ذلك لما نعلم من كثرة الهجمات السيبرانية، فيجب هنا اتباع هذه الإرشادات لتصبح عادات للمبرمج للحفاظ على أمان الشيفرة.

ورغم أنها حل غير كامل وقد نخسر معها رؤية استعلامات SQL الحقيقية المرسَلة إلى قاعدة البيانات، والذي كنا سنستفيد منه في تنقيح الأخطاء، لكن هذه أفضل طريقة للتغلب على احتمال أن يُدخل المستخدم بيانات شاذةً.

24.13 كلمة أخيرة

استخدمنا SQLite في أمثلتنا لأنها متاحة مجانًا، وسهلة التثبيت والاستخدام، ومرنة في مدى الأخطاء التي تسمح بها، غير أن هذه البساطة تعني أن المزايا الأكثر تقدمًا والموجودة في الحزم الأقوى غير موجودة فيها، فإمكانيات معالجة النصوص ومجال القيود المتاحة محدود للغاية، ويجب قضاء وقت كافٍ في قراءة الوثائق المرجعية عند التعامل مع قواعد بيانات مثل Oracle أو قاعدة بيانات DB2 من IBM، لأن استخدام المزايا التي توفرها قاعدة البيانات يقلل من كم الشيفرات المخصصة التي ستُكتب، ويحسن الأداء أيضًا.

24.14 المزايا المتقدمة لقواعد البيانات

24.14.1 المفاتيح الخارجية

تحتوي أغلب قواعد البيانات -بما فيها SQLite- على مفاتيح خارجية foreign keys، تسمح لنا بتحديد الروابط أو العلاقات بين الجداول، بحيث تتكون هذه المفاتيح من مفاتيح رئيسية لجداول أخرى، وأحيانًا في قواعد بيانات أخرى.

ورغم أننا لم نتحدث كثيرًا عن الروابط بين الجداول إلا أنها من أكثر المزايا شيوعًا في قواعد البيانات العلائقية خاصةً عند زيادة حجم البرامج.

24.14.2 التكامل المرجعي

التكامل المرجعي Referential integrity هو القدرة على عدم السماح بقيم بيانات في عمود إلا إذا كانت موجودةً في مكان آخر، ففي قاعدة بيانات الموظفين مثلًا، كان بإمكاننا تقييد القيمة في حقل `Employee.Grade` لتسمح بالقيم المعرَّفة في جدول `Salary.Grade` فقط، وهذه أداة بالغة القوة في الحفاظ على اتساق البيانات عبر قاعدة البيانات، خاصةً عندما تُستخدم القيم مفاتيح لربط جدولين، كما هو الحال في أعمدة `grade`.

تدعم SQLite صورةً محدودةً من التكامل المرجعي في صورة قيد، لكنها تتطلب بعض المعالجة الخاصة، وهي أقل شموليةً من قواعد البيانات الأقوى.

24.14.3 الإجراءات المخزنة

الإجراءات المخزنة Stored Procedures هي دوال مكتوبة بلغة برمجة خاصة proprietary programming language يوفرها مزود قاعدة البيانات، وتُخزَّن في قاعدة البيانات، ومزيتها أنها إجراءات مصرَّفة compiled، وبالتالي أسرع كثيرًا من استخدام أوامر SQL المكافئة، كما أنها توفر من استهلاك الإنترنت وحجم البيانات المطلوب، من خلال طلب اسم الدالة والوسطاء فقط ليرسلها برنامج العميل، ولكونها مبنيةً داخل الخادم فهي تسمح لنا ببناء سلوكيات مشتركة -مثل قواعد العمل المعقدة- في قاعدة البيانات حيث يمكن تشاركها بواسطة جميع التطبيقات باستمرار، لكن عيبها أنها خاصة ومغلقة، فإذا أردنا تغيير مزود قاعدة البيانات لدينا فيجب إعادة كتابة جميع الإجراءات المخزنة، في حين أن SQL القياسية ستعمل دون تغيير على أي قاعدة بيانات، ولا تدعم SQLite أي إجراءات مخزنة.

24.14.4 العروض

العروض Views هي جداول افتراضية مكونة من جداول حقيقية أخرى، وقد تكون مجموعةً فرعيةً من البيانات في جدول آخر من أجل تبسيط التصفح، وفي الحالة الأشهر تكون بعض الأعمدة من جدول وبعضها من جدول آخر بحيث يكون الجدولان مرتبطين معًا بمفتاح ما.

ويمكن النظر إليها على أنها استعلام SQL يُنفّذ باستمرار وتُخزن النتيجة في العرض، وسيتغير العرض مع تغير البيانات التي فيه، وقد تسمح بعض قواعد البيانات بقراءة البيانات في العرض، غير أنها جميعًا تسمح بالتحديثات.

تُستخدم العروض في الغالب لتقسيم البيانات بحيث يستطيع المستخدم الواحد أن يرى مجموعة البيانات المتعلقة به فقط، وتدعم SQLite العروض، لكننا لم نشرحها هنا.

24.14.5 عمليات الحذف المتتالية

إذا أجرينا حذفًا متتاليًا cascaded delete بين عنصري بيانات فهذا يعني أنه عند حذف العنصر الرئيسي فستُحذف العناصر الثانوية أيضًا، وأحد أشهر الأمثلة على ذلك هو الطلبات orders، حيث يتكون الطلب عادةً من الطلب نفسه بالإضافة إلى العناصر المطلوبة، ويُخزن كل منها عادةً في جدول منفصل، فإذا حذفنا الطلب فإننا نرغب بالتأكيد في حذف عناصره، وتُهيأ عمليات الحذف المتتالية في تعليمات DDL لقواعد البيانات المستخدمة لإنشاء مخططات قواعد البيانات، وهي أحد أنواع القيود، ولا تدعم SQLite عمليات الحذف المتكررة.

24.14.6 المحفزات Triggers

تشبه المحفزات Triggers الأحداث إلى حد ما، فهي جزء من SQL يُنفَّذ عند وقوع حدث معين، كما في حالة إدراج صف جديد في جدول ما مثلًا، وهي مفيدة في التحقق من صلاحية البيانات تحققًا أعمق من مجرد النوع والقيم، كما تكون مفيدةً للغاية عند ربطها بالإجراءات المخزنة.

أما عيبها فهو إمكانية إساءة استخدامها، وشرهها الزائد للموارد، مما يؤدي لإبطاء قاعدة البيانات كثيرًا، لذا يجب استخدامها بحذر، وتدعم SQLite محفزات SQL الأساسية والخالصة.

24.14.7 أنواع البيانات المتقدمة

تسمح بعض قواعد البيانات بتخزين مجموعات متنوعة من أنواع البيانات، فقد يكون لدينا عناوين شبكية network addresses وكائنات ثنائية binary كبيرة يشار إليها اختصارًا باسم BLOBS لملفات الصور وغيرها، إضافةً إلى البيانات العددية وبيانات المحارف والتاريخ والوقت المعتادة، ومن أنواع البيانات المشهورة أيضًا نوع يسمى بالنوع العشري ذي الدقة الثابتة fixed precision decimal type، وهو يُستخدم في البيانات المالية

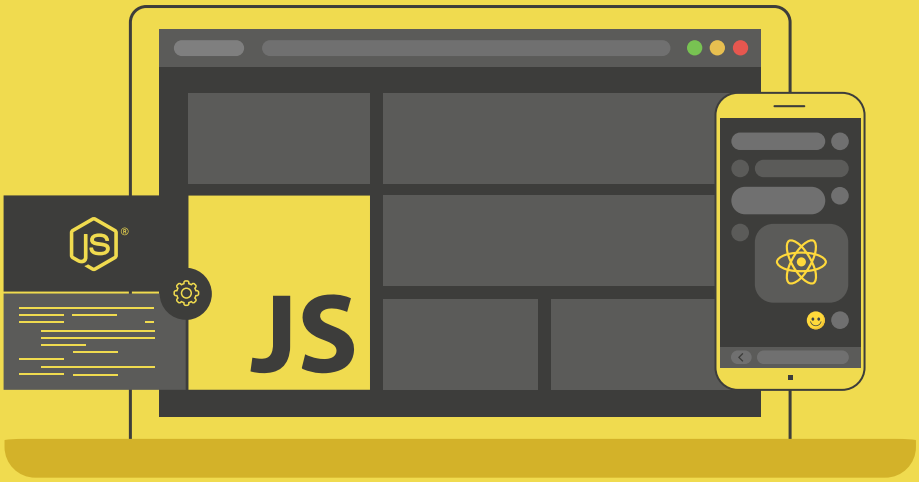
لتجنب أخطاء التقريب الموجودة في أعداد الفاصلة العائمة التقليدية، وتدعم SQLite بيانات BLOB لكنها لا تدعم بقية الأنواع المتقدمة، وللاطلاع على المزيد من الاستخدامات المعقدة لـ SQLite يمكن العودة إلى موقع sqlitetutorial الذي يحتوي على شرح ممتاز يسهل تعلم ما فيه بناءً على ما شرحنا هنا.

24.15 خاتمة

في نهاية هذا الفصل نرجو أن تكون تعلمت ما يلي:

- قواعد البيانات تنظم البيانات في جداول.
- تتكون السجلات من حقول، وتشمل صفوف الجداول.
- SQL هي لغة تُستخدم في إدارة البيانات.
- الأوامر الأساسية في لغة SQL هي: CREATE وINSERT وSELECT وUPDATE.
- توفر لغات البرمجة مغلِّفات SQL للوصول إلى البيانات من البرامج.
- تخزن المؤشرات نتائج استعلامات SQL في صورة مؤقتة، لكن يمكن الوصول إليها.
- تُستخدم محدِّدات API DB لإدراج قيم في الاستعلامات، ولا تُستخدم صيغة السلاسل النصية القياسية.

دورة تطوير التطبيقات
باستخدام لغة JavaScript



احترف تطوير التطبيقات بلغة جافا سكريبت
انطلاقًا من أبسط المفاهيم وحتى بناء تطبيقات حقيقية

التحق بالدورة الآن

25. التواصل مع نظام التشغيل عبر بايثون

سننظر في هذا الفصل في دور نظام التشغيل وكيفية الوصول إليه من بايثون، وسنشرح فيه:

- دور نظام التشغيل ووظيفته.
- وصول بايثون إلى نظام التشغيل.
- العمل مع الملفات والمجلدات.
- التعامل مع العمليات.
- معرفة البيانات الخاصة بالمستخدمين.

25.1 التعرف على نظام التشغيل

يدرك أغلب مستخدمي الحواسيب أن حواسيبهم تعمل بنظم تشغيل، سواء كانت ويندوز أو لينكس أو ماك أو غير ذلك، لكنهم قد يجهلون الدور الذي تقوم بـه تلـك النظم على وجـه الدقـة، خاصـةً أن أغلب نظم التشـغيل التجارية تأتي مع بـرامج كثـيرة ليسـت جـزءًا من نظـام التشـغيل، مثـل بـرامج عـرض الصـور، وتصـفح الـويب، ومحررات النصوص، لكن الجهة المصدرة للنظام تحزمها معـه لأن المسـتخدم لن يسـتطيع الاسـتفادة من نظـام التشغيل وحده.

25.1.1 مبدأ طبقات الكعكة

إذا أردنا معرفة حقيقة نظام التشغيل فيجب أن ننظر أولًا في الطريقـة الـتي بُـنيت بهـا الحواسـيب، والـتي يمكن النظر إليها على أنها كعكـة متعـددة الطبقـات multi-layer cake، حيث يكـون عتـاد الحاسـوب والقطـع

الإلكترونية المختلفة فيه الطبقة الدنيا، بما في ذلك وحدة المعالجة المركزية CPU، والقرص الصلب، والذاكرة، ونظام الإدخال والإخراج الذي يحوي مداخل USB، وفتحات بطاقات الذاكرة، ووصلات الشبكة وغيرها.

أما الطبقة التي فوقها فهي BIOS، أو نظام الإدخال والإخراج البسيط Basic Input Output System، وهي أول طبقة برمجية في تكوين الحاسوب، والمسؤولة عن إقلاع الحاسوب وتوفير أبسط واجهة ممكنة للعتاد، إذ تسمح بنقل رؤوس القرص الصلب من مسار لآخر، ومن قطاع لآخر داخل المسار الواحد، وبقراءة أو كتابة البايتات المنفردة في المخازن المؤقتة للبيانات data buffers المتصلة بكل منفذ، ومع هذا لا تعرف BIOS نفسها أي معلومات عن الملفات أو المجلدات أو أي من المفاهيم العليا الأخرى التي نتعامل معها نحن المستخدمون، وإنما تعرف كيفية التعامل مع الأجهزة الإلكترونية البسيطة التي يتكون منها الحاسوب، بل قد لا تتعامل معها مباشرةً، وإنما من خلال برمجيات يثبتها مزود العتاد نفسه في أماكن حساسة داخل BIOS، وهذا شائع في بطاقات الرسوم graphics cards على وجه الخصوص، إذ تثبت الشركات المصنعة لتلك البطاقات روابط إلى تعريفات الرسوم الخاصة بها في مكان معياري داخل شيفرة BIOS، لتستدعي BIOS واجهةً متفقًا عليها، مع توفير الشركة لبرنامجها الخاص بها، وتُخزَّن BIOS عادةً في نوع خاص من رقاقات الذاكرة ذات طبيعة شبه دائمة، أي أنها لا تحذَف عند قطع التيار الكهربي، لكن يمكن إعادة كتابتها عند الحاجة لتحديث BIOS.

أما الطبقة التالية لطبقة BIOS فهي طبقة نظام التشغيل، ويختلف هيكل هذه الطبقة كثيرًا وفقًا لكل نظام تشغيل، غير أنها تتكون في الغالب من نواة kernel، أو مجموعة أساسية من الخدمات مع تعريفات للعتاد device drivers، وقد توضع تعريفات العتاد داخل النواة، أو تكون وحدات تحمّلها النواة عند الحاجة، وهذا شبيه بتحميل البرامج لوحدات بايثون عند الحاجة، أما وظيفة هذه الطبقة فهي الترجمة من العتاد منخفض المستوى إلى البُنى المنطقية التي نستطيع فهمها واستخدامها، مثل الملفات والمجلدات، ومن المهم هنا ملاحظة أن نفس العتاد والـ BIOS يستطيعان تشغيل عدة نظم تشغيل مختلفة معًا، فمن السهل إعداد النظام والقرص الصلب بحيث يمكن تثبيت عدة نظم تشغيل على نفس الحاسوب، ويختار المستخدم بينها عند الإقلاع.

ننتقل إلى الطبقة قبل الأخيرة في هذه الكعكة، وهي الصدفة Shell أو بيئة المستخدم، وهي واجهة رسومية في أغلب أنظمة التشغيل الحديثة، رغم توفر واجهة سطرية لها بشكل أو بآخر.

أما الطبقة الأخيرة، فهي طبقة برمجيات المستخدم، التي تتكون من حزمة من البرامج التي يثبتها المستخدم أو تأتي مثبتةً مع النظام، وفيها متصفحات الويب، وبرامج البريد، ومعالجات النصوص وغيرها، كما تحتوي على أدوات برمجية مثل بايثون.

لندرس مثالًا نفهم من خلاله التفاعل بين هذه الطبقات، حيث سنرى ما الذي يحدث عند فتح ملف في بايثون وقراءته.

- نستخدم صدفة نظام التشغيل لبدء بايثون، وتحميل ملف سكربت أيضًا.
- تستدعي شيفرة بايثون الدالة `open()` الخاصة ببايثون.

- تستدعي بايثون داخل هذه الدالة دالة نظام تشغيل لفتح نفس الملف.
- يبحث نظام التشغيل عن بعض البيانات الداخلية، ويترجم اسم الملف إلى مجموعة من المسارات والقطاعات على القرص الصلب. -ويكون قد عرف أي قرص يستخدم بداهةً!-، ويحدد ذلك الملف على أنه ملف مفتوح، ويمنع تعديلات المستخدمين الآخرين عليه إن دعت الحاجة إلى ذلك.
- يستدعي برنامج بايثون `file.read()`.
- تستدعي بايثون دالة القراءة الخاصة بنظام التشغيل.
- يستدعي نظام التشغيل عدة دوال من BIOS لوضع رؤوس القراءة في القرص الصلب عند المواضع المناسبة.
- يخبر نظام التشغيل BIOS أن تقرأ أعداد البايتات المناسبة من ذلك الموضع.
- يكرر نظام التشغيل عملية الموضعة والقراءة تلك حتى الانتهاء من قراءة جميع البيانات المطلوبة للملف.
- تعيد دالة النظام البيانات إلى بايثون.
- تعالج بايثون البيانات، وتعرض النتائج للمستخدم من خلال دوال أخرى لنظام التشغيل والبيوس BIOS.

قد يبدو هذا معقدًا، ولن نقول إنه ليس كذلك، لكن هذا التعقيد نفسه هو سبب وجود نظم التشغيل، لتوفر على المستخدمين والمبرمجين تلك المهام، فما علينا نحن المبرمجون إلا استدعاء `open()` و`read()`.

25.1.2 التحكم في العمليات

يوفر نظام التشغيل إمكانية بدء البرامج وتشغيلها، إضافةً إلى ما ذكرناه سابقًا من التحكم في الوصول إلى العتاد، كما يوفر الآليات اللازمة لإدارة تلك البرامج التي تعمل بالتوازي -أي في نفس الوقت- على الحاسوب، لأن عدد المعالجات الموجود على اللوحة الأم أقل بكثير من البرامج التي ستعمل على نظام التشغيل، لذا تنفذ نظم التشغيل عمليةً تسمى التقسيم الزمني time slicing، وهي إعطاء كل برنامج من البرامج العاملة حاليًا على الحاسوب حصةً صغيرةً من المعالج؛ ولفترة قصيرة من الزمن، قبل أن تنقل تلك الحصة بسرعة إلى عملية أخرى أو برنامج آخر، فتعطي ذلك الإحساس بأن جميع البرامج تعمل في نفس الوقت، وتختلف كفاءة نظم التشغيل في تنفيذ هذه العملية، فلم تكن الإصدارات الأولى من ويندوز وماك MacOS قادرةً على إجرائها وإدارتها إلا بمساعدة البرامج نفسها، فإذا فشل أحد البرامج في توفير نقطة توقف مناسبة فإن الحاسوب كله يتوقف عن الاستجابة، فيما يعرف بالتعليق Hanging.

وتستخدم أغلب نظم التشغيل الحديثة نظامًا يسمى تعدد المهام الوقائي pre-emptive multi-tasking، حيث يقاطع نظام التشغيل البرامج بغض النظر عما تفعله تلك البرامج عند المقاطعة، ويعطي البرنامج التالي

الوصول إلى المعالج تلقائيًا، وتُستخدم عدة خوارزميات هنا لتحسين كفاءة تلك العملية وفقًا لنظام التشغيل نفسه، مثل التبديل الحلَقي round robin، أو الأحدث استخدامًا most recently used، أو الأقدم استخدامًا least recently used، ولا تهمنا هذه الخوارزميات نحن المبرمجون، ولا بأس إن افترضنا أن تلك البرامج المتعددة تعمل بالتوازي.

تأتي أغلب الحواسيب الحديثة بمعالجات متعددة قد تكون رقاقات متعددةً من السليكون، أو عدة نوى -عدة معالجات منفردة- على رقاقة واحدة، ووظيفة نظام التشغيل هنا أن يخصص العمليات الحالية إلى المعالجات والأنوية ليضمن الاستخدام الأمثل، ونعيد التذكير هنا أن ذلك لا يهمنا، ونترك هذه المهمة لنظام التشغيل نفسه.

25.1.3 صلاحيات المستخدمين والأمان

تسمح أغلب نظم التشغيل الحديثة بوجود عدة مستخدمين على نفس الحاسوب، لكل واحد منهم ملفاته الخاصة وإعدادات سطح مكتبه وغير ذلك، وتزيد بعض نظم التشغيل على ذلك بأن تسمح بولوج عدة مستخدمين إلى نفس النظام في نفس الوقت، فيما يعرف باسم الجلسات المتعددة multi-session، لكن تعدد المستخدمين هذا تأتي معه مشكلة الأمان، فمن المهم ألا يستطيع سعيد رؤية بيانات محمد، وكذلك العكس، إلا إذا أعطى محمد صلاحية الوصول إلى ملفاته إلى سعيد، ويكون نظام التشغيل مسؤولًا عن ضمان حماية ملفات كل مستخدم، عن طريق منع الوصول إليها إلا من الأشخاص المصرح لهم فقط.

25.2 استخدام نظام التشغيل في البرامج

قد نتساءل عن جدوى هذا الحديث عن أنظمة التشغيل لنا نحن المبرمجون، بما أن وظيفة ذلك النظام أن يجنبنا كل تلك التفاصيل التي ذكرناها، من التعامل مع العتاد وتنظيم عمليات تشغيل البرامج.

والإجابة على ذلك أننا قد نحتاج أحيانًا إلى التعامل مع ذلك العتاد، ولا تستطيع الدوال البرمجية المعتادة تنفيذ ما نريده، أو ربما نحتاج إلى تشغيل برنامج آخر من داخل برنامجنا مثلًا، بل ربما نحتاج -من داخل برنامجنا- إلى التحكم في الحاسوب بنفس الطريقة التي يتحكم بها المستخدم فيه، وحل ذلك كله أن يكون لنا صلاحية الوصول إلى طبقة نظام التشغيل نفسه.

وتوفر بايثون عددًا من الوحدات التي يمكن استخدامها للتفاعل مع نظام التشغيل، لعل أهمها وحدة os، التي توفر واجهةً مشتركةً لأي نظام تشغيل، من خلال تحميل وحدات منخفضة المستوى lower level، وستتصرف أنظمة التشغيل تصرفًا مختلفًا وفقًا للطريقة التي تنفذ بها تلك الدوال داخليًا، ولا ينتج عن هذا السلوك مشاكل غالبًا، لكن إذا واجهنا سلوكًا غريبًا من وحدة os فنرجع إلى التوثيق لنرى إن كان ثمة قيود على نظام التشغيل الخاص الذي نستخدمه.

أما وحدات النظام الأخرى التي سندرسها فهي الوحدة `shutil` التي توفر للمبرمجين تحكمًا في الملفات والمجلدات على مستوى المستخدم، والوحدتان `os.path` و`glob` اللتان توفران أدوات للتنقل في نظام ملفات الحاسوب، وهذا هو الجزء الذي سننظر فيه أولًا.

انتبه: ذكرنا أن نظم التشغيل مختلفة عن بعضها، لذا فإن الأمثلة التي سنذكرها أدناه لن تعمل على ويندوز في الغالب، وستحتاج إلى البحث عن أوامر CMD المكافئة لها وأماكنها، واستبدالها بأوامر يونكس التي سنكتبها هنا، أما مستخدمو لينكس أو MacOS X فلن تكون لديهم تلك المشكلة لأن كلا النظامين يونكس، مادامت البرامج المطلوبة موجودةً ومثبتةً، لأن بعض أنظمة يونكس لا تأتي بتلك البرامج افتراضيًا.

25.3 معالجة الملفات

شرحنا كيفية التعامل مع الملفات في فصل التعامل مع الملفات في البرمجة فما الذي نريده من نظام التشغيل طالما نستطيع التعامل معها؟

والجواب أن ما ذكرناه في الفصل الثاني عشر هو أننا نستطيع إنشاء الملفات وتعديلها، لكننا لم نذكر حذفها لأننا لم نكن نستطيع ذلك بتوابع الملفات العادية، أما هنا فنستطيع استخدام نظام التشغيل في حذف الملفات، كما أن `open()` ستكون كافيةً إذا علمنا مكان الملف، لكن كيف نجده إذا لم نعرف مكانه، بل ماذا لو أردنا معالجة مجموعات من الملفات؟ مثل ملفات الصور في مجلد ما، وكذلك التحكم الدقيق في القراءة من الملفات، حيث كانت التوابع القياسية التي شرحناها من قبل تقرأ سطرًا واحدًا فقط أو الملف كله، فماذا لو أردنا قراءة بضعة بايتات فقط؟ إن دوال نظام التشغيل هي الحل لتلك الحالات كلها.

25.3.1 إيجاد الملفات

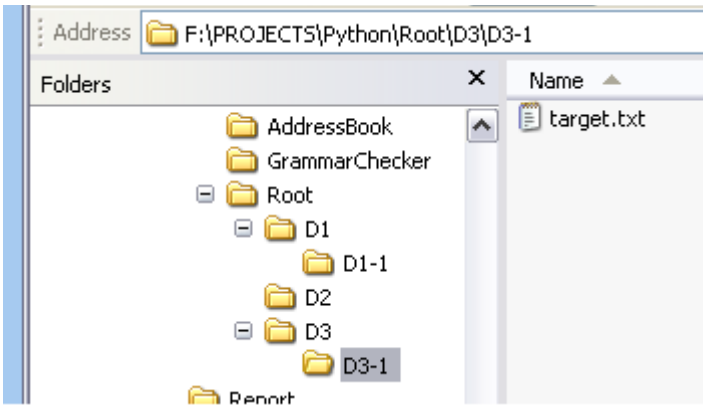
أول وحدة سندرسها هي الوحدة `glob` التي نستخدمها للعثور على الملفات، والتي تجلب قوائم بأسماء الملفات، وتعود تسميتها إلى نظام يونكس، حيث استُخدم المصطلح لوصف عملية تحديد مجموعات من الملفات باستخدام محارف بدل wildcards، أما الاسم نفسه فيعود تاريخه إلى أمر قديم في نظم التشغيل استُخدم لإجراء عملية التوسع، وهو اختصار global، والوحدة سهلة الاستخدام، فبعد استيرادها نجد دالةً وحيدةً هي `glob()`، ونمرر إليها نمطًا pattern لتطابقه، فتعيد قائمةً بأسماء الملفات:

```
import glob
files = glob.glob("*.txt")
print( files )
```

سنحصل على قائمة من الملفات النصية في المجلد الحالي، وهذا يطرح سؤالًا حول كيفية معرفة المجلد الحالي الذي نحن فيه، وهل نستطيع تغييره أم لا؟ ولا شك أننا نستطيع معرفة المجلد وتغييره باستخدام وحدة os:

```
import os
print( os.getcwd() )  #cwd=current working directory
os.chdir("/usr/local") #chdir=change working directory
print( os.getcwd() )
print( os.listdir('.') ) #عرض المجلدات الجديدة
```

تعلمنا البحث عن ملف في المجلد الحالي، وكيف ننتقل من ذلك المجلد إلى أي مجلد نريد، لكن لا زال البحث عن ملف ما عمليةً متعبةً، ولتسهيلها سنستخدم الدالة `os.walk()`، فللعثور على ملف في مكان ما من نقطة بداية نستخدم `os.walk`، كما سننشئ دالة `findfile` التي نستطيع استخدامها في برامجنا، لكن سننشئ أولًا بيئة اختبار تتكون من هرمية مجلدات تحت مجلد الجذر `Root`، وسنضع بعض الملفات في كل مجلد، وسيكون الملف الذي نبحث عنه في أحدها، وقد سميناه traget.txt، ويمكن رؤية هذه الهرمية في لقطة الشاشة التالية:



تأخذ دالة `os.walk` معاملاً هو نقطة البداية، وتعيد مولدًا generator هو أشبه بقائمة وهمية تبني نفسها عند الحاجة، ويتكون من صفوف tuples فيها ثلاثة عناصر، تسمى أحيانًا بالثلاثي triplet أو tuple-3، وهذه العناصر هي: الجذر، قائمة المجلدات في الجذر الحالي، قائمة الملفات الحالية في مجلد الجذر.

فإذا نظرنا إلى الهرمية التي أنشأناها، فسنتوقع أن يبدو صف tuple الأول كما يلي:

```
( 'Root', ['D1','D2','D3'], ['FA.txt','FB.txt'] )
```

نتحقق بسهولة من ذلك بكتابة حلقة `for` في المحث التفاعلي:

```
>>> for t in os.walk('Root'):
...     print( t )
...
('Root', ['D1', 'D2', 'D3'], ['FA.txt', 'FB.txt'])
('Root/D1', ['D1-1'], ['FC.txt'])
('Root/D1/D1-1', [], ['FF.txt'])
```

```
('Root/D2', [], ['FD.txt'])
('Root/D3', ['D3-1'], ['FE.txt'])
('Root/D3/D3-1', [], ['target.txt'])
>>>
```

يوضح هذا المسارَ الذي أخذته `os.walk`، كما يوضح كيف يمكن العثور على ملف، وإنشاء مساره الكامل بالبحث في عنصر `files` من الصفوف التي تعيدها `os.walk`، ودمج الاسم بمجرد العثور عليه مع قيمة `root` للصف الذي يحويه.

وإذا كتبنا دالتنا بحيث تستخدم التعابير النمطية وتعيد قائمةً، فنستطيع إنشاء دالة تكون أقوى من `glob.glob` البسيطة التي رأيناها من قبل، لكنها ستكون أبطأ أيضًا، لنجربها وننظر كيف تبدو:

```
# تحتوي وحدة findfile.py على دالة واحدة
# هي دالة find_file()، المبنية على استخدام دالة os.walk()

import os,re

def find_file(filepattern, base = '.'):
    regex = re.compile(filepattern)
    matches = []
    for root,dirs,files in os.walk(base):
        for f in files:
          if regex.match(f):
             matches.append(root + '/' + f)
    return matches
```

احفظ ذلك في ملف اسمه findfile.py، ثم لنختبر المحث التفاعلي كما يلي:

```
>>> import findfile
>>> findfile.find_file('t.*','Root')
['Root/D3/D3-1/target.txt']
>>> findfile.find_file('F.*','Root')
['Root/FA.txt', 'Root/FB.txt', 'Root/D1/FC.txt',
'Root/D1/D1-1/FF.txt', 'Root/D2
/FD.txt', 'Root/D3/FE.txt']
>>> findfile.find_file('.*\.txt','Root')
['Root/FA.txt', 'Root/FB.txt', 'Root/D1/FC.txt',
'Root/D1/D1-1/FF.txt', 'Root/D2
```

```
/FD.txt', 'Root/D3/FE.txt', 'Root/D3/D3-1/target.txt']
>>> findfile.find_file('D.*','Root')
[]
```

لاحظ أن الوسيط الأول في `find_file()` هو `'t.*'`، وتذكر أنه تعبير نمطي وليس محرف بدل لاسم ملف؛ مثل الذي استخدمناه مع `glob`، لذا فإنه يشير إلى `t` متبوعةً بعدد من المحارف يساوي صفرًا أو أكثر، وليس ملفًا فقط مثل t.txt.

يظهِر الخرج أن برنامجنا يعمل، ونلاحظ في المثال الأخير أنه يعمل فقط في حالة الملفات لأن أسماء المجلدات تكون في قوائم `dirs` التي لا نستطيع التحقق منها. جرب إضافة دالة جديدة إلى وحدة `findfile` سمِّها `find_dir()`، للبحث عن المجلدات التي تطابق تعبيرًا نمطيًا ما، ثم ادمجهما معًا لإنشاء دالة ثالثة هي `find_all()` تبحث في كل من الملفات والمجلدات.

25.3.2 نقل الملفات ونسخها وحذفها

تحدثنا في فصل التعامل مع الملفات عن كيفية نسخ ملف من خلال قراءته ثم كتابته إلى موقع جديد، إلا أنه يمكن استخدام نظام التشغيل لينفذ هذا العمل بدلًا منا بتعليمة واحدة، ونستخدم في بايثون وحدة `shutil` لمثل هذه المهام، وهي تحتوي على عدة دوال مفيدة، سندرس منها ما يلي:

ا. الدالة copy(src, dst)

تنسخ هذه الدالة الملف المصدر `src` إلى الملف أو المجلد الوجهة `dst`، فإذا كانت الوجهة مجلدًا فسيُنشأ ملف له نفس اسم الملف المصدر، أو يكتب فوقه overwritten في المجلد المحدد، وتُنسخ بتات الصلاحية permission bits كذلك، وتعطى أسماء المصدر `src` والوجهة `dst` في صورة سلاسل نصية.

ب. الدالة move(src, dst)

تنقل هذه الدالة الملف أو المجلد تعاوديًا recursively إلى موقع جديد، فإذا كانت الوجهة على نظام الملفات الحالي فستعاد تسمية الملف المصدر `src`، وإلا فستنسخ الدالة الملف `src` إلى الوجهة `dst`، ثم تحذف المصدر `src`.

كما سندرس أيضًا الدوال التالية من وحدة `os`:

ج. الدالة remove(path)

تحذف هذه الدالة مسار الملف، فإذا كان المسار `path` مجلدًا فيُرفع `OSError`، ولحذف المجلد انظر في `rmdir()`.

د. rename(src, dst)

تعيد هذه الدالة تسمية الملف أو المجلد المصدر `src` ليكون اسم الملف الوجهة `dst`، فإذا كانت الوجهة مجلدًا `dst` فسيُرفع خطأ `OSError`.

وأبسط طريقة لنرى بها هذه الدوال عمليًا هي تجربتها في المحث التفاعلي باستخدام بنية المجلد/الملف التي أنشأناها لمثال `os.walk` السابق:

```
>>> import os
>>> import shutil as sh
>>> import glob as g
>>> os.chdir('Root')
>>> os.listdir('.')
['D1', 'D2', 'D3', 'FA.txt', 'FB.txt']

>>> sh.copy('FA.txt', 'CA.txt')
>>> os.listdir('.')
['CA.txt', 'D1', 'D2', 'D3', 'FA.txt', 'FB.txt']

>>> sh.move('FB.txt','CB.txt')
>>> os.listdir('.')
['CA.txt', 'CB.txt', 'D1', 'D2', 'D3', 'FA.txt']

>>> os.remove('FA.txt')
>>> os.listdir('.')
['CA.txt', 'CB.txt', 'D1', 'D2', 'D3']

>>> for f in g.glob('*.txt'):
...     newname = f.replace('C','F')
...     os.rename(f,newname)
...
>>> os.listdir('.')
['D1', 'D2', 'D3', 'FA.txt', 'FB.txt']
>>>
>>>
```

في الأمثلة السابقة نقلنا الملفات ونسخناها وحذفنا الملف الأصلي، ثم أعدنا التسمية لاستعادة المجلد إلى حالته الأولى، ويستطيع المستخدم تنفيذ كل تلك عمليات في سطر الأوامر أو في مدير الملفات، لكننا نفذناها هنا باستخدام بايثون.

لاحظ استخدام حلقة `for` لتنفيذ التغييرات المتعددة، وقد كان باستطاعتنا إضافة جميع أشكال التحققات والقواعد داخل الحلقة، للسماح بإنشاء بعض الأدوات القوية للتعديل في الملفات، وبكتابة تلك الشيفرة في سكربت يمكننا تنفيذ تلك التغييرات بالقدر الذي نشاء بمجرد تشغيل السكربت.

25.3.3 اختبار خصائص الملفات

نحتاج عند التعامل مع الملفات إلى معرفة خصائصها، فمثلًا عندما نقرأ قائمة مجلد من `glob`، سنرغب في معرفة هل ما نريده ملف أم مجلد، وكذلك معرفة آخر مرة عدِّل عليه فيها، أو مراقبة ذلك لنرى إن كان يُعدَّل بانتظام، مما يدل على أن مستخدمًا غيرنا -أو حتى برنامجًا آخر- له وصول إلى الملف، أو ربما نريد مراقبة حجم الملف، لنرى إن كان يزيد مثلًا، ونستطيع فعل كل تلك الأمور من برامجنا باستغلال مزايا نظام التشغيل، ولكن يجب أن نرى أولًا كيف نتحقق من نوع الملف الذي نتعامل معه:

```
import os.path as p
import glob
for item in glob.glob('*')
   if p.isfile(item): print( item, ' is a file' )
   elif p.isdir(item): print( item, ' is a directory' )
   else: print( item, ' is of unknown type' )
```

لاحظ أن دوال الاختبار توجد في وحدة os.path، وأن هناك عدة اختبارات أخرى متاحة، ويمكن الرجوع إليها والقراءة عنها في توثيق وحدة `os.path`.

سننظر الآن في ميزة عمر الملف، إذ توجد عدة تواريخ مهمة في حياة كل الملف، أولها تاريخ إنشائه، ثم تاريخ آخر تعديل عليه، وأخيرًا تاريخ آخر وصول إليه، وقد لا توفر بعض أنظمة التشغيل جميع تلك التواريخ، لكن أغلبها توفر تواريخ الإنشاء والتعديل، ويمكن الوصول إليهما في بايثون من خلال وحدة `os.path` باستخدام الدالتين `ctime()` و`mtime()` على الترتيب، وسننظر في بعض الملفات الموجودة في بنية `Root` لدينا، وقد أنشئت جميعها بنفس الوقت تقريبًا، مع اختلاف طفيف لملفات المستوى الأعلى، لأننا أجرينا تعديلات عليها في مثالنا السابق باستخدام `rename()`.

```
>>> import time as t
>>> os.listdir('.')
>>> for r,d,files in os.walk('.'):
...   for f in files:
```

```
...     print( f,' created: %s:%s:%s' %
t.localtime(p.getctime(r+'/'+f))[3:6] )
...     print( f,' modified: %s:%s:%s' %
t.localtime(p.getmtime(r+'/'+f))[3:6])
...
FA.txt  created: 13:42:11
FA.txt  modified: 13:36:27
FB.txt  created: 13:42:11
FB.txt  modified: 17:32:5
FC.txt  created: 17:32:46
FC.txt  modified: 17:32:5
FF.txt  created: 17:34:3
FF.txt  modified: 17:32:5
FD.txt  created: 17:33:12
FD.txt  modified: 17:32:5
FE.txt  created: 17:33:53
FE.txt  modified: 17:32:5
target.txt  created: 17:34:28
target.txt  modified: 17:32:5
>>>
```

لاحظ النتيجة الغريبة لملف `FA` حيث يظهر أنه عُدّل قبل إنشائه! لأننا أنشأناه نسخةً من الملف الأصلي ثم حذفنا الأصل، وأعدنا تسمية النسخة بنفس اسم الملف الأصلي، فرأى نظام التشغيل أن إعادة التسمية لم تغير شيئًا من المحتوى فلم يغير تاريخ التعديل -وهو وقت عملية النسخ- لكنه رأى عملية إعادة التسمية على أنها توقيت إنشاء اسم الملف الحالي، فلابد أن هذا ملف `FA.txt` جديد بما أننا حذفنا الملف الأصلي!

يختلف أمر الملف `FB.txt` قليلًا، إذ لم يتغير المحتوى لأننا نقلنا الملف بدلًا من نسخه، لذا تاريخ آخر تعديل هو نفسه تاريخ الإنشاء الأصلي -انظر الملفات الأخرى-، لكن مرةً أخرى يرى نظام التشغيل عملية إعادة التسمية على أنها إنتاج لملف `FB.txt` جديد، فتظهر هنا قيمة الوقت الأخرى.

في نظام التشغيل دالة تستطيع إعادة أغلب المعلومات التي نحتاجها عن ملف في صف tuple واحد، وهي دالة `stat()`، ولها عدة صور، إلا أننا لن ندرس إلا النسخة الموجودة في وحدة `os`، حيث تعيد `os.stat()` صف tuple يحتوي على ما يلي:

- `st_mode`: بتات الحماية.
- `st_ino`: رقم مؤشر الفهرسة inode.
- `st_dev`: جهاز.

- `st_nlink`: عدد الروابط الصلبة hard links.
- `st_uid`: معرِّف المستخدم المالك.
- `st_gid`: معرِّف المجموعة للمالك.
- `st_size`: حجم الملف بالبايت.
- `st_atime`: وقت آخر وصول للملف.
- `st_mtime`: وقت آخر تعديل للمحتوى.
- `st_ctime`: وقت الإنشاء، لكن وفقًا للمنصة.

لاحظ أنه قد توجد بعض الحقول الإضافية أحيانًا وفقًا لما يدعمه نظام التشغيل نفسه، فتحقق من توثيق المنصة لديك. لننظر المثال التالي المطبق على ملف المستوى الأعلى `FA.txt`:

```
>>> fmtString = "protection: %s\nsize: %s\naccessed: %s\ncreated: %s"
>>> stats = os.stat('FA.txt')
>>> print( fmtString % stats[0],stats[6],stats[7],stats[9] )
protection: 33279
size: 0
access: 1132407387
created: 1132407731
```

نحتاج لفك ترميز جميع القيم لنفهمها، باستثناء الحجم الذي ما هو إلا عدد البايتات في الملف، وسنرى كيفية العمل مع كل منها لاحقًا، أما العلامات الزمنية timestamps فهي سهلة لأن الأرقام فيها تمثل عدد الثواني لكل دورة، وقد شرحنا ذلك من قبل، ونستطيع استخدام دوال وحدة `time`، مثلما فعلنا في `localtime()` أعلاه لتحويلها إلى بنية بيانات ذات فائدة لنا أو إلى سلسلة نصية.

وتحتاج صيغة الحماية protection لفك ترميزها أولًا، باستخدام بعض القيم الخاصة الموجودة في وحدة `stat`، إلا أننا نحتاج إلى استخدام بعض العوامل operators الأخرى، والتي تعرف بالعوامل الثنائية bitwise operators.

١. العوامل الثنائية والرايات

تحتوي وحدة `stat` على مجموعة من الثوابت مسبقة التعريف، مثل المتغيرات ذات القيم التي لا تتغير، وتسمح تلك الثوابت بفك ترميز بيانات الصلاحيات باستخدام العوامل الثنائية، وهذه العوامل هي نفسها العوامل البولیانية المنطقية التي استخدمناها من قبل مثل `and` و`or` و`not`، إضافةً إلى عامل جديد هو `xor`، لكن الفرق أن هذه تعمل على البتات الثنائية المنفردة من البيانات، بدلًا من القيمة الكلية.

يمكن العثور على القيم الموجودة في stat بالنظر إلى المتغيرات المعرَّفة مثل قيم ثنائية، وتوفر بايثون خيار صيغة ثنائية مضمنة فيها هي bin، والتي تبدو كما يلي:

```
>>> bin(22)
'0b10110'
```

يمثل الجزء 0b في بداية السلسلة النصية أسلوب بايثون في إخبارنا أن هذا عدد ثنائي، ونستطيع تحويل السلسلة النصية الثنائية إلى عدد عشري باستخدام دالة int، من خلال توفير وسيط ثانٍ قيمته 2، لأن قاعدة العدد الثنائي هي 2 في الاصطلاح الرياضي:

```
>>> int('0b10110',2)
22
>>> int('10110',2)
22
```

لاحظ أن 0b في بداية السلسلة النصية اختيارية، لأن الوسيط الثاني 2 يخبر بايثون بوجوب معاملة السلسلة النصية على أنها ثنائية، وهذا مفيد إذا كنا نقرأ السلسلة من المستخدم مثلًا بدلًا من استخدام خرج bin.

العوامل الثنائية Bitwise Operators

سننظر الآن في كيفية عمل العوامل الثنائية باستخدام دالة bin لعرض قيم الدخل والخرج، ولننظر أولًا في تأثير and الثنائية التي رمزها &:

```
>>> print( bin(5) )
'0b101'
>>> print( bin(1) )
'0b001'
>>> print( bin(2) )
'0b010'
>>> print( bin(5 & 1) )
'0b001'
>>> print( bin(5 & 2) )
'0b000'
```

سنراجع تلك النتائج لنرى ما يحدث، تذكر أن and المنطقية تتحقق -أي تكون true- إذا وفقط إذا تحققت كلا القيمتين، أي كانتا true، وبالمثل فإن & الثنائية تكون true -قيمتها 1- إذا تحقق البتّان الموافقان -كان لهما القيمة 1-، وإذا نظرنا إلى 1 & 5 فسنجد أن البت الذي في أقصى اليمين يتحقق في كل من 5 و1، أي

تكون قيمتها 1، وبناءً عليه فإن البت الذي في أقصى اليمين لـ `1 & 5` سيكون 1 أيضًا، أما في حالة `2 & 5` فلا توجد حالة تكون فيها لبتين اثنين القيمة 1 في كل من 5 و2، وعليه تكون النتيجة كلها أصفارًا بالنسبة لـ `2 & 5`.

يقود هذا السلوك إلى ميزة خاصة لعمليات & الثنائية، فمن خلال "جمع" قيمة ثنائية مع عدد يحتوي على رقم ثنائي وحيد مضبوط على القيمة 1، نستطيع معرفة إن كانت قيمة البت الموافقة في القيمة التي نختبرها هي 1 أيضًا أم لا، فإن كانت كذلك فسنحصل على نتيجة غير صفرية، وهي الموافقة للقيمة `True` في الاصطلاح البولياني، لنستعرض الآن مثالًا نفترض فيه أننا نريد التحقق من كون البت الثاني في عدد معيَّنًا أم لا، ونحن نعلم مما سبق أن القيمة التي فيها بت واحد في الموضع الثاني -إذا بدأنا العد من اليمين- هي 2:

```
BIT2 = 2
for n in range(10):   if n & BIT2: print( n,' = ',bin(n) )
```

ستكون النتيجة أن البت الثانية معينة -أي set- في كل من 2 و3 و6 و7.

نستطيع تنفيذ أمور مشابهة لذلك باستخدام `or` الثنائية، والتي نستخدم لها الرمز `|`، وكذلك `not` الثنائية التي لها الرمز `~`، رغم أن هذه الأخيرة قد يكون لها بعض النتائج الغريبة المتعلقة بكيفية تخزين الحواسيب للأعداد السالبة، جرب استخدام دالة `bin()` مع هذه العوامل لعرض دخل وخرج البتّات لترى سلوكها، وتذكر أن توازن بين القيم بتًا بتًا.

أما العامل الثنائي الأخير فهو or الحصرية أو `xor`، التي لها الرمز `^`، وتتحقق -أي تكون true- إن تحققت قيمة واحدة فقط من قيمتي الاختبار، لكنها لا تتحقق عند تحقق كلا القيمتين، وسنحصل على نتائج مثيرة للاهتمام وفقًا لهذه القاعدة، فمثلًا إذا طبقنا هذا العامل على أي عدد مع نفسه فسنحصل على صفر، أما إذا طبقناه على أي عدد مع مفتاح فسنحصل على نتيجة، وإذا طبقنا هذا العامل مرةً ثانيةً على النتيجة والمفتاح فسنحصل على القيمة الأصلية، وهذا السلوك مفيد جدًا في علم التشفير. لننظر في بعض الأمثلة قبل أن نعود إلى وحدة `stat` ونبحث في قيم الصلاحيات.

```
>>> print( bin(5 ^ 2) )
'0b111'
>>> print( bin(5^5) )
'0b000'
>>> print( bin((5^2)^2) )
'0b101'
```

لاحظ أن النتيجة في الحالة الأخيرة هي سلسلة ثنائية للعدد 5، أي أننا إذا طبقنا العامل `xor` مرتين فسنحصل على القيمة الأصلية.

الرايات Flags

يسمى المتغير المستخدَم لتخزين قيمة بوليانية رايةً flag، لأن الراية يمكن رفعها أو خفضها، وعندما يكون لدينا قيم رايات كثيرة تتعلق بوحدة واحدة؛ فمن الشائع استخدام عدد واحد لتخزين الفئة المجمَّعة للرايات، باستخدام بتات بيانات منفردة لتمثيل كل راية منفردة، ويمكن جلب قيم الرايات فيما بعد باستخدام العوامل الثنائية التي شرحناها أعلاه، مدمجةً مع قيمة فك ترميز تعرف باسم القناع mask، مما يسمح لنا باستخراج البتات التي نريدها تحديدًا، فعلى سبيل المثال، كانت القيمة `BIT2` أعلاه قناعًا لاستخراج البت الثاني، وتشكل وحدة `stat` مجموعةً من الأقنعة مسبقة التعريف لاختبار رايات الصلاحيات التي تعيدها الدالة `os.stat()`.

25.3.4 استخدام ثوابت stat مع العوامل الثنائية

سننظر الآن إلى بعض قيم `stat` مثل أعداد ثنائية، ونرى إن كنا نستطيع معرفة كيفية استخدامها:

```
>>> import stat
>>> dir(stat)
['ST_ATIME', 'ST_CTIME', 'ST_DEV', 'ST_GID', 'ST_INO', 'ST_MODE',
'ST_MTIME',
'ST_NLINK', 'ST_SIZE', 'ST_UID', 'S_ENFMT', 'S_IEXEC', 'S_IFBLK',
'S_IFCHR',
'S_IFDIR', 'S_IFIFO', 'S_IFLNK', 'S_IFMT', 'S_IFREG', 'S_IFSOCK',
'S_IMODE',
'S_IREAD', 'S_IRGRP', 'S_IROTH', 'S_IRUSR', 'S_IRWXG', 'S_IRWXO',
'S_IRWXU',
'S_ISBLK','S_ISCHR', 'S_ISDIR', 'S_ISFIFO', 'S_ISGID', 'S_ISLNK',
'S_ISREG',
'S_ISSOCK', 'S_ISUID', 'S_ISVTX', 'S_IWGRP', 'S_IWOTH', 'S_IWRITE',
'S_IWUSR',
'S_IXGRP', 'S_IXOTH', 'S_IXUSR', '__builtins__', '__doc__',
'__file__',
'__name__']
>>> print( bin(stat.S_IREAD) )
'0b100000000'
>>> print( bin(stat.S_IWRITE) )
'0b010000000'
>>> print( bin(stat.S_IEXEC) )
'0b001000000'
```

الملاحظة الأولى هنا هي تعريفنا للكثير من الثوابت، والثانية أن القيم الثلاثة التي طبعناها هي القيم التي تحدد إمكانية قراءة الملف أو كتابته أو تنفيذه على الترتيب، ولاحظ أن لكل قيمة مجموعة بتات واحدة، كما في

قيمة BIT2 في الأمثلة السابقة، لذا نستطيع استخدام عامل and الثنائي لإيجاد صلاحيات ملفنا، بعد استدعاء الدالة os.stat()، كما يلي:

```
import os, stat
permission = os.stat('FA.txt')[0]
if permission & stat.S_IREAD:
   print( 'The file is readable' )
if permission & stat.S_IWRITE:
   print( 'The file is writeable' )
if permission & stat.S_IEXEC:
   print( 'The file is executable' )
```

هذه هي الصلاحيات التي نريدها غالبًا، ويمكنك الرجوع إلى توثيق وحدة stat للمزيد من الصلاحيات، والتحقق من فهمها بتجربتها في محث بايثون.

كما توجد دالة access() المساعِدة والموجودة في وحدة os، والتي تسمح لنا بالتحقق من أغلب صلاحيات الوصول الشائعة، غير أن منظور القناع البِتي bitmask الموصوف أعلاه يغطي خيارات أكثر، لذا يمكن استخدامه في المواضع التي لا يصلح استخدام access() فيها.

25.3.5 تغيير صلاحيات الملفات

بعد أن عرفنا صلاحيات ملف ما، نستطيع استخدام وحدة os لتغييرها إلى ما يناسبنا، وتستخدم بايثون اصطلاحات يونكس لتغييرها، حيث يكون لكل ملف مجموعة من ثلاث رايات هي (قراءة read، كتابة write، تنفيذ execute)، لكل تصنيف من تصنيفات المستخدمين الثلاثة (المالك owner، المجموعة المالكة group، العالم world أي بقية المستخدمين)، لذا سيكون لدينا 9 رايات لكل ملف، وتمثَّل تلك الرايات بتسعة بتّات، تشكل البتات اليمنى في راية الصلاحيات التي تعيدها os.stat.

تمثَّل مجموعات الصلاحيات تلك في التوثيق بسلسلة من تسعة محارف، تتكون من ثلاث مجموعات من المحارف rwx، مع شرطةٍ بدل المحرف إذا لم تكن الصلاحية مضبوطةً أو معيَّنةً set، وبناءً على ذلك ستعني السلسلة النصية "rwxr-xr--" أن للمستخدم صلاحية القراءة والكتابة والتنفيذ rwx، وللمجموعة صلاحية القراءة والتنفيذ r-x، وللنوع الثالث الذي هو العالم صلاحية القراءة فقط r--، وإذا أردنا تغيير الصلاحيات فسنعين البتات وفقًا لما نريد التغيير إليه، لذلك سنستخدم الدالة chmod() في وحدة os، حيث تأخذ وسيطًا هو الرقم المكون من 9 بتات، فعلى سبيل المثال يمثل الرقم 0b111101100 الصلاحيات rwxr-xr--، ونستخدم ذلك لضبط الوصول لملف ما كما يلي:

```
>>> os.chmod('FA.txt',0b111101100)
```

وإذا كانت لديك خبرة سابقة بالأعداد الثُمانية octal فستعرف أن كل رقم ثُماني يمثل ثلاثة بتات ثنائية، وعلى ذلك نستطيع التعبير عن الصلاحيات بسهولة في ثلاثة أرقام ثُمانية، ويربط الجدول التالي القيم الثُمانية بنظائرها المكافئة لها من الأرقام الثنائية وصلاحيات `rwx`:

الثُماني	الثنائي	"rwx"
0	0b000	"---"
1	0b001	"--x"
2	0b010	"-w-"
3	0b011	"-wx"
4	0b100	"r--"
5	0b101	"r-x"
6	0b110	"rw-"
7	0b111	"rwx"

لن يجد مستخدمو يونكس مشكلةً في التعبير عن الصلاحيات بهذه الطريقة، ويمكن استخدام ذلك في بايثون أيضًا لجعل استدعاء `chmod` يبدو كما يلي:

```
>>> # يجب استخدام صفر في البداية ليُعامل على أنه ثُماني
>>> os.chmod('FA.txt',0754)
```

ينفذ المثالان أعلاه نفس الأمر، حيث يعيّنان صلاحيات المالكين لتكون قراءةً وكتابةً وتنفيذًا، ويعيّنان صلاحيات المجموعة لتكون قراءةً وتنفيذًا فقط، وصلاحيات العالم لتكون قراءةً فقط، وتجدر الإشارة إلى أن النسخة الثُمانية أسهل في الكتابة إذا أردت التعود على التحويلات بين الثُمانية والثنائية.

25.4 المسارات والملفات والمجلدات

من الشائع عند تطوير برنامج ما أن تكون ملفات البيانات في نفس المجلد الذي فيه ملفات البرنامج، ليسهل العثور على الملفات عند الحاجة، أما إذا كان استخدام البرنامج عامًا فلا يمكن افتراض أن الملفات ستكون في مواضع معروفة، لذا قد نحتاج إلى البحث عنها باستخدام `glob` أو `os.walk` اللتين شرحناهما أعلاه، فإذا وجدنا الملف الذي نريده وأردنا فتحه أو فحص خصائصه؛ فسنحتاج إلى تعيين المسار الكامل له، أما إذا كان لدينا اسم المسار الكامل فقد نفككه لاستخراج اسم الملف أو المجلد فقط، لوضعه في متغير مثلًا، وتوفر `os.path` إمكانية تنفيذ مثل تلك المهام.

يُنظر إلى أسماء الملفات في بايثون على أنها تتكون من عدة أجزاء، فهناك حرف اختياري يمثل القرص الذي يوجد عليه الملف، رغم أن هذا قد لا يكون في جميع أنظمة التشغيل، إذ ليس لأنظمة التشغيل غير ويندوز هذا المفهوم الذي يضع اسم القرص الصلب جزءًا من اسم الملف، ثم يأتي تسلسل من أسماء المجلدات تفصل

بينها بعض المحارف المحددة، ففي بايثون مثلًا نستخدم /، لكن قد يكون لبعض أنظمة التشغيل محارفها الخاصة، وأخيرًا لدينا اسم الملف أو الاسم القاعدي basename، والذي يحوي صورةً من صور امتدادات الملفات extension:

```
F:/PROJECTS/PYTHON/Root/FA.txt
```

يخبرنا المسار أعلاه أن الملف المسمى FA.txt موجود في مجلد الجذر Root، الذي يوجد بدوره في مجلد PYTHON تحت مجلد PROJECTS، في مجلد المستوى الأعلى للقرص F:، ويحمل الملف امتداد الملفات النصية .txt.

نستطيع الآن استخراج الاسم القاعدي base name، والامتداد extension، وتسلسل المجلدات من المسار الكامل، باستخدام الدوال الموجودة في وحدة os.path، كما يلي:

```
>>> pth = F:/PROJECTS/PYTHON/Root/FA.txt
>>> stem, aFile = os.path.split(pth)
>>> print( 'stem : ',stem, '\nfile : ',aFile )
stem :  F:/PROJECTS/PYTHON/Root
file :  FA.txt

>>> # this only works on OS with drive concept, like Windows
>>> print( os.path.splitdrive(pth) )
('F:', '/PROJECTS/PYTHON/Root/FA.txt')

>>> print( os.path.dirname(pth) )
F:/PROJECTS/PYTHON/Root

>>> print( os.path.basename(pth) )
FA.txt

>>> print( os.path.splitext(aFile) )
('FA', '.txt')
```

وندمجها معًا بالشكل:

```
>>> print( os.path.join(stem,aFile) )
F:/PROJECTS/PYTHON/Root\\FA.txt
```

يجب ملاحظة أن `os.path.join` تستخدم محرف الفصل الرسمي للنظام، فإذا أردنا بناء مسار يصلح للعمل على نظم تشغيل مختلفة فيجب استخدام `os.path.join` بدلًا من كتابة المسار في برنامجنا، كما يمكن تحديد أي عدد نرغب فيه من عناصر المسار في قائمة الوسطاء، فيمكن كتابة المثال أعلاه بالشكل:

```
>>> print( os.path.join("F:\\", "PROJECTS", "PYTHON", "Root",
"FA.txt"))
F:\\PROJECTS\\PYTHON\\Root\\FA.txt
```

25.4.1 واصفات الملفات وكائنات الملفات

تستخدم بعض الوحدات من عائلة `os` آليةً مختلفةً قليلًا للوصول إلى الملفات، وهو ما يُعرف باسم واصف الملفات file descriptor، وهو مرتبط بشدة بمفهوم الملف في نظم التشغيل بدلًا من كائن الملف file object الذي استخدمناه حتى الآن، ومزيته أننا نستطيع الوصول إلى حزمة من عمليات الملفات منخفضة المستوى low-level operations، والتي تمكننا من الحصول على تحكم أكبر في الملفات وبياناتها، ويمكن إنشاء واصف ملف من كائن ملف والعكس، لكن من الأفضل ألا نخلط بين وضعي العمليات داخل دالة واحدة أو برنامج واحد، فإما أن نستخدم واصفات الملفات أو كائنات الملفات.

تشمل دوال واصفات الملفات جميع عمليات الملفات المعتادة، مثل فتح open وقراءة read وكتابة write وإغلاق close، أما البرامج منخفضة المستوى فتكون صعبة الاستخدام، إضافةً إلى احتمال حدوث أخطاء عند استخدامها، لهذا يجب استخدام الوصول منخفض المستوى عند الضرورة فقط، أما في أغلب الحالات فيكفي استخدام كائنات الملف القياسية.

لكن ما هي الحالات التي قد نحتاج فيها إلى وصول منخفض المستوى إلى الملفات؟

تستخدم العمليات القياسية مبدأً يعرف باسم التخزين المؤقت للدخل/الخرج أو buffered IO، حيث تُحفظ البيانات في مناطق تخزين تسمى بالمخازن المؤقتة buffers أثناء عمليات القراءة والكتابة، وقد تتسبب تلك المخازن في مشاكل عند الوصول إلى عتاد غريب أو إجراء عمليات ذات توقيت حرج critical time، ففي مثل تلك الحالات يكون الحل هو العمليات منخفضة المستوى، لكن إذا لم تكن متأكدًا من سبب استخدامها فلا تفعل، لكن مع هذا العيب الكبير في العمليات منخفضة المستوى، فإننا نقول إن استخدامها ليس بتلك الصعوبة التي تظهر من سياق الحديث، وإنما قصدنا أن هناك بعض المشاكل التي يجب تجنبها، لنبدأ بإجراء مهمة بسيطة، مثل فتح ملف نصي، وكتابة بعض البيانات فيه، ثم إغلاقه:

```
fname = 'F:/PROJECTS/PYTHON/Root/FL.txt'
mode = os.O_CREAT | os.O_WRONLY # create and write
access = 0777 # read/write/execute for all
data = 'Test text to check that it worked'
fd = os.open(fname, mode, access) # NB. os version not the builtin!
```

```
length = os.write(fd, data)
if length != len(data):
   print( 'Amount of data written doesn't match the data!' )
os.close(fd)
```

بالمثل نستطيع قراءة البيانات من الملف:

```
mode = os.O_RDONLY # read only
fd = os.open(fname,mode) # no access needed this time
result = os.read(fd, length)
print( result )
os.close(fd)
```

نلاحظ عدة أمور هي:

- الطريقة التي نعين بها نوع وصول الملف أعقد هنا، ويجب أن نستخدم عامل or الثنائي لدمج جميع الرايات المطلوبة كما وفرتها وحدة os.
- نستطيع توفير مستوى وصول غير المستوى الافتراضي، وهذا أمر تمتاز به عن توابع كائن الملف القياسية.
- العدد الثُماني -لأنه يبدأ بصفر- هو نفسه العدد المذكور في قسم "تغيير صلاحيات الملفات" السابق.
- عند قراءة البيانات يجب أن نمرر طول البيانات المقروءة، باستخدام read القياسية، لكن هذا إجباري بالنسبة للعمليات منخفضة المستوى.

أما البيانات المقروءة والمكتوبة فعلًا فهي من النوع سلسلة بايتات bytestring، وهذا لن يكون مشكلةً عند التعامل مع سلاسل المحارف، لأننا نستطيع استخدام تابع decode الخاص باليونيكود لتحويل البايتات إلى محارف، لكن تظهر المشكلة عند التعامل مع أنواع بيانات أخرى، إذ يجب استخدام وحدة struct كما شرحنا في فصل التعامل مع الملفات.

25.5 معالجة العمليات

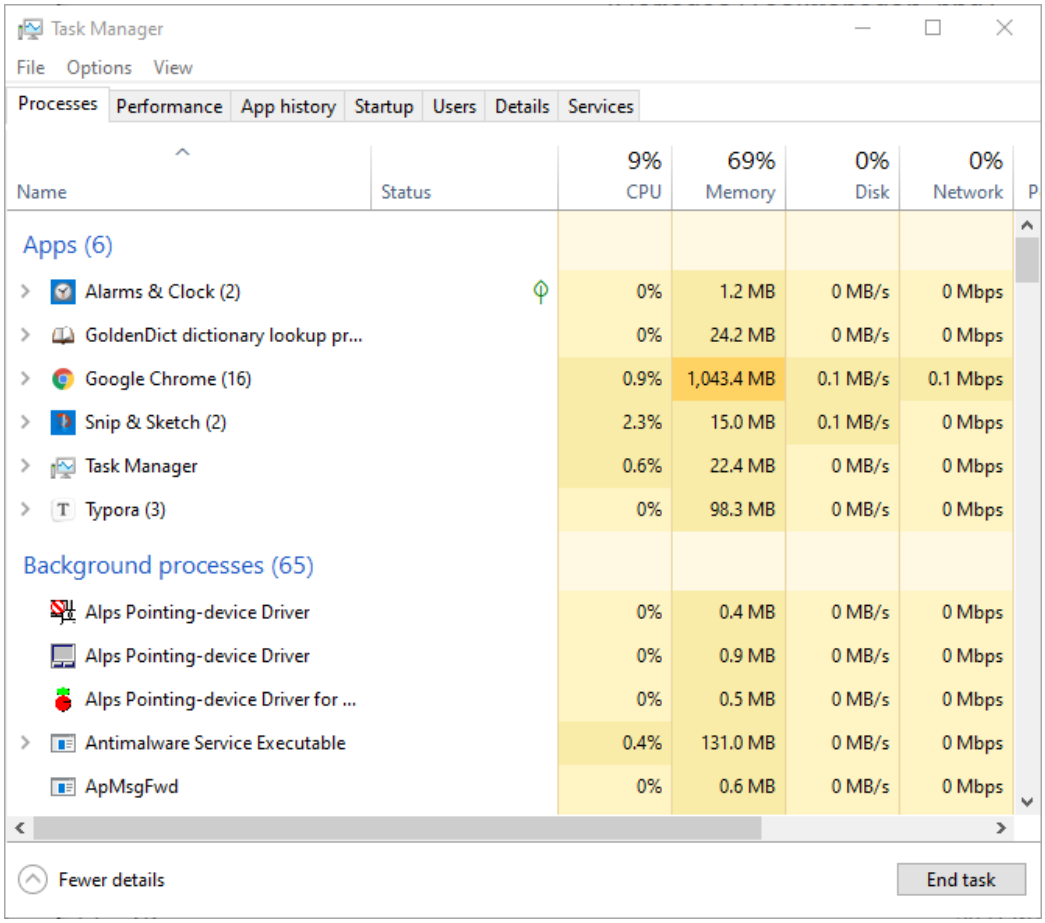
يُعد تنفيذ البرامج أو تشغيلها من أكثر المهام التي ننفذها عن استخدام نظام التشغيل، ونفعل ذلك غالبًا من خلال واجهة رسومية أو صدفة سطر أوامر، كما يمكن أيضًا أن نبدأ البرامج من داخل برامج أخرى، فقد لا نحتاج أحيانًا إلى أكثر من بدء برنامج والسماح له بالعمل حتى نهايته، أما في أحيان أخرى فقد نحتاج إلى توفير بيانات الدخل أو قراءة الخرج إلى برنامجنا، ويُعرف هذا اصطلاحًا باسم التواصل البيني للعمليات أو التواصل بين

العمليـات communication inter-proces، أو IPC اختصـارًا، وسنشـرح هـذا المفهـوم بالتفصـيل في الفصل التالي.

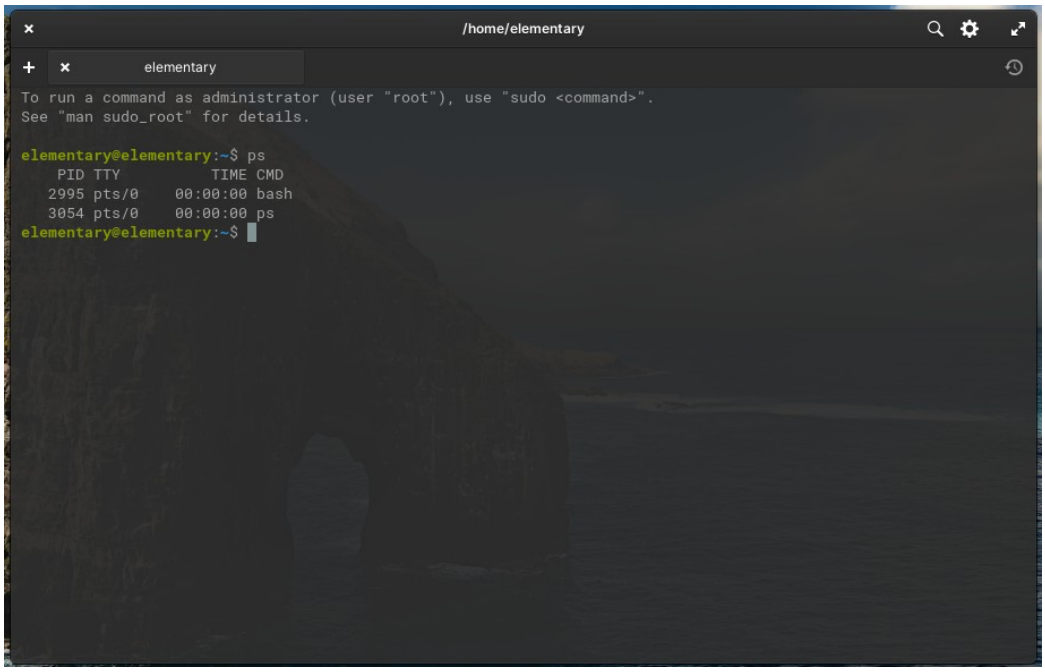
25.5.1 تعريف العملية

العملية Process في الاصطلاح الحاسوبي هي ما يطلق عليه المستخدمون "تشغيل البرنامج"، فإذا كان لدينا ملف تنفيذي على الحاسوب ونريد تنفيذه؛ فسيبدأ التشغيل داخل مساحة الذاكرة الخاصة به، وقد يكون من الممكن بدء أكثر من نسخة من نفس الملف التنفيذي، ويشغل كل منها مساحةً خاصةً به من الذاكرة، ويدير بياناته الخاصة به، ويطلَق اسم العملية على كل واحد من تلك البرامج التنفيذية مع بيئة التشغيل الخاصة به.

نستطيع أن نرى العمليات الحالية على الحاسوب باستخدام الأدوات التي يوفرها نظام التشغيل، فإذا كنت على ويندوز فاضغط على Ctrl+Alt+Del لتشغيل برنامج Task Manager، وانظر في التبويب المسمى Processes، وسترى قوائم طويلةً من العمليات الحالية التي قد تتعرف على بعضها من البرامج الخاصة بك، أما بقية العمليات فتكون خدمات بدأها نظام ويندوز نفسه، كما قد تلاحظ أن بعض التطبيقات تبدأ عدة عمليات، مثل قواعد البيانات العلائقية relational databases وخوادم الويب، وتوضح الصورة التالية مثالًا لبرنامج Task Manager:



أما على لينكس أو نظام MacOS فيُستخدم الأمر `ps` لعرض العمليات أو المهام الحالية، أو `top` الذي يعطي عرضًا حيًا لتلك العمليات، وسيبدو أمر `ps` كما يلي:



25.5.2 تشغيل برنامج خارجي: الدالة os.system()

نعلم الآن الفرق بين البرنامج والعملية، وسننظر في كيفية تنفيذ برنامج من بايثون، وتوجد الكثير من الطرق لذلك -لأسباب تاريخية-، لكننا سنكتفي باثنتين فقط، والأولى منهما بسيطة الاستخدام للغاية، لكنها محدودة الإمكانيات، أما الثانية فهي المنظور الذي يُنصح به هذه الأيام، وهي أكثر قوةً ومرونةً.

الطريقة الأولى السهلة والقديمة هي استخدام دالة `system()` من وحدة `os`، وهذا ينفذ سلسلة أمر string command، ويعيد شيفرة خطأ تعكس انتهاء الأمر على النحو الصحيح أو لا، ولا توجد طريقة للوصول إلى الخرج الفعلي للبرنامج المستدعى ولا لتزويد العملية الحالية بالدخل المناسب، لذا تناسب `system()` تنفيذ البرامج التي لا تحتاج إلى متابعة، مثل إخلاء شاشة الطرفية، حيث لا نحتاج أن نعرف هل نُفّذ الأمر بنجاح أم لا، ولا أن نتفاعل مع الأمر بمجرد بدئه، ويمكن رؤية مثال على ذلك في يونكس:

```
>>> import os
>>> errorcode = os.system("clear")
>>> print( errorcode )
0
```

ويختلف الأمر قليلًا في أنظمة التشغيل المبنية على ويندوز:

```
>>> errorcode = os.system("CLS")
>>> print( errorcode )
```

```
0
```

غير أن النتيجة في الحالتين يجب أن تكون إخلاء النافذة الطرفية، ويجب أن يكون رمز الخطأ errorcode هو الصفر إشارةً إلى نجاح التنفيذ، وقد لا يكون هذا مفيدًا؛ لكننا نستطيع استخدام system في سكربتاتنا حين نرغب في عرض الخرج الطبيعي للأمر، أو عندما لا يهمنا إلا نجاح الأمر أو فشله، فمثلًا نستطيع أن نطلب من المستخدم أن يعدل إعدادات ملف من خلال بدء جلسة محرر editor، ثم يغلق جلسة التحرير عند انتهائه، ويعود التحكم إلى برنامجنا الذي يستطيع عندئذ أن يقرأ الملف المعدَّل.

```
>>> filename = 'xxyyzz.config'
>>> errorcode = os.system('nano %s' % filename)
>>> if not errorcode:
...     with open(filename) as config:
...         # هنا ملف config للعملية
```

يُظهر المثال بعض التقنيات المفيدة، حيث يُظهر طريقةً لوصف استدعاءات النظام باستخدام تنسيق السلاسل النصية، كما يُظهر تفسير رمز الخطأ لتحديد نتيجة العملية.

إذا نجحت عملية التعديل فسيكون رمز الخطأ صفرًا، حتى لو لم يحدث أي تعديل على الملف، لكن إذا فشلت -كما في حالة عدم العثور على المحرر مثلًا- فسنحصل على رمز خطأ قيمته غير الصفر.

ومع أن system سهلة الاستخدام إلا أنها ليست مرنةً، ولا توجد طريقة مباشرة لتوصيل أي بيانات إلى برامجنا، ونستطيع محاكاة ذلك بالتقاط الخرج إلى ملف نصي مؤقت، ثم فتح ذلك الملف ومعالجته مثلما اعتدنا، لكن ثمة طريقة أفضل لتحقيق نفس النتيجة وهي استخدام وحدة subprocess.

25.5.3 إدارة العمليات باستخدام الوحدة subprocess

بعد العديد من المحاولات الفاشلة لتحسين التحكم في العمليات، حدث تطور كبير عند إدخال الوحدة subprocess إلى الإصدار 2.4 من بايثون، وقد جاءت تلك الوحدة خصيصًا لاستبدال جميع الآليات القديمة، ويمكن رؤية العديد من الأمثلة على كيفية استخدامها في توثيقها، أما هنا فسنغطي أغلب استخدامها العام، وتوجد دوال سهلة ومريحة إذا رغبت في محاكاة وظيفة os.system أو ما شابهها.

بُنت الوحدة subprocess على صنف يسمى Popen، لاحظ الحرف الأول بالحرف الكبير، ويمكن استخدامه لإنشاء نسخة من أمر ما، لكن توثيق هذا الصنف قد يسبب الرهبة للبعض لأن باني Popen يحتوي على الكثير من المعامِلات الرائعة، لكن الجيد في الأمر أن لجميعها تقريبًا قيمًا افتراضية، ويمكن تجاهلها في الحالات بالغة البساطة، وبناءً على ذلك لتشغيل أمر من أوامر نظام التشغيل من داخل سكربت ما، لا نحتاج إلا إلى فعل ما يلي:

```
import subprocess
ps = subprocess.Popen(['ps', '-ef'], shell=False)
```

الوسيط الأول قائمة من السلاسل النصية، التي تمثل الأمرَ وجميع وسطائه، ففي المثال أعلاه ننفذ أمر `ps -ef`.

يمنع الوسيط الثاني `shell=False` تمرير الأمر من خلال برامج الصدفة الخاصة بالمستخدمين -مثل باش Bash-، مما قد ينتج عنه مشاكل أمنية، بسبب الأسماء البديلة aliases المخصصة للأوامر مثلًا، فيجب أن نستخدم `shell=False` كلما أمكن ذلك، لكن من الضروري أحيانًا أن نفسر الأمر بواسطة الصدفة، كما في حالة تمرير محارف بدل wild-cards مثل "*.jpg" إلى تطبيق معالجة للصور، فالصدفة هي التي توسع محرف البدل إلى قائمة من أسماء الملفات فعلًا.

خزّنا نسخة Popen الناتجة في متغير سميناه باسم الأمر الذي ننفذه، وهذا ليس ضروريًا لكنه سلوك حسن إذا كنا نشغّل عدة عمليات مساعِدة في نفس الوقت.

توجد دالة أخرى اسمها `call` يمكن استخدامها بدلًا من `os.sytem` في المثال أعلاه:

```
subprocess.call(['ps', '-ef'], shell=False)
```

تكاد دالة `call` أن تطابق استخدام صنف `Popen` الذي شرحناه من قبل، غير أنه ليس لها تلك الخيارات المتاحة في `Popen`، ولا تنشئ نسخًا، لذا فهي أوفر في استهلاك موارد النظام، لكن لها نفس عيوب `os.system`.

لوحدة `subprocess` التي نستخدمها هنا مزية عن الدوال القديمة، وهي أنها ترفع استثناءً `OSError` إذا لم يوجد الأمر المطلوب، أما الدوال الأخرى فكانت ستتركنا دون أي إشارة على وجود أخطاء.

25.5.4 التواصل مع العمليات باستخدام Popen

من الممكن تنفيذ كل ما فعلناه بوحدة `subprocess` باستخدام `os.system`، لكن هذا على وشك التغير الآن حيث سنتعرف على كيفية تبادل البيانات مع عملية جارية بدأناها باستخدام `subprocess.Popen`، فإذا عدنا قليلًا إلى أمر `ps` الذي نفذناه من قبل فسنجد أننا شغلنا البرنامج، لكننا لم نستطع الوصول إلى خرجه، ونحتاج إلى إجراء تعديل بسيط لنتمكن من الوصول إلى ذلك الخرج:

```
import subprocess as sub
ps = sub.Popen(['ps', '-ef'], shell=False, stdout=sub.PIPE)
print( ps.stdout.read().decode('utf8') )
```

لقد أضفنا وسيطًا هنا يخبر Popen أن يرسل خرجه القياسي `stdout` إلى أنبوب عملية فرعية subprocess Pipe وقد شرحنا مجرى الخرج والدخل القياسيين `stdin` و`stdout` في فصل قراءة المدخلات

من المستخدم، ونستطيع الآن أن نصل إلى بيانات الخرج باستخدام الخاصية stdout لنسخة Popen التي نقرؤها مثل أي ملف عادي، وتُحوَّل سلسلة البايتات bytestring الناتجة إلى محارف يونيكود باستخدام decode ثم تُطبع، لكن من الممكن أن نسندها إلى متغير ونعالجها بأي طريقة نشاء.

انتبه، توجد بعض المشاكل في الوصول إلى مجرى الخرج القياسي بهذه الطريقة، تدور أغلبها حول إدارة عدة عمليات متزامنة معًا، لذا نشجع استخدام التابع Popen.communicating الذي يعيد قائمةً من مجاري البيانات data streams يكون مجرى الخرج القياسي stdout هو الأول فيها، ثم مجرى الدخل القياسي stdin، ثم مجرى الخطأ القياسي stderr.

فإن مثّل ذلك مشكلةً فيمكن إعادة كتابة المثال السابق كما يلي:

```
import subprocess as sub
ps = sub.Popen(['ps', '-ef'], shell=False, stdout=sub.PIPE)
print( ps.communicate()[0].decode('utf8') )
```

نستخدم Popen.communicate([0]) هنا للوصول إلى أنبوب الخرج القياسي بطريقة آمنة.

ويمكن إرسال بيانات إلى عملية ما بطريقة مشابهة، إذ نخبر الصنف Popen أن يستخدم أنبوبًا لمجرى الدخل القياسي، ثم يكتب البيانات إلى مجرى البيانات ذاك مثل الملف العادي، رغم أن البيانات هنا يجب أن تكون سلسلة بايتات وليست سلسلة محارف.

في المثال التالي، نفتح محرر أسطر يونكس ex، ونرسل إليه بعض البيانات لينشئ ملفًا نصيًا، ثم نغلق المحرر قبل التحقق من وجود ذلك الملف الجديد:

```
import subprocess as sub
import os
ex = sub.Popen(['ex', '/tmp/testex.txt'],stdin=sub.PIPE)
ex.stdin.write(b'i\nthis is some text\n.\n')
ex.stdin.write(b'wq\n')
ex.stdin.close()
print( os.listdir('/tmp') )
```

لاحظ أن المدخلات كلها سلاسل بايتات، وأننا نحتاج إلى إدراج محارف الإرجاع carriage return على أنها محددات أسطر جديدة في السلاسل، كما أن النقطة الموجودة في أول سطر من ex.stdin.write هي التي تخبر ex بانتهاء تسلسل الدخل، ويؤكد استدعاء os.listdir أن الملف الجديد موجود في مجلد tmp، أو يمكن أن نتحقق من ذلك بواسطة مدير الملفات.

وبهذا استطعنا الكتابة في عملية جارية، رغم أنها أقل شيوعًا من القراءة من العمليات.

25.6 مسألة الأمان

توفر نظم التشغيل هذه الأيام أغلب الأدوات التي تضمن أمان بيئة التشغيل التي يستخدمها المستخدم على الحاسوب، يمكن الوصول إليها -مثل باقي أدوات نظم التشغيل- من خلال واجهة برمجة التطبيقات API الخاصة بنظام التشغيل نفسه، لكن مع شرط تنظيم نظام التشغيل وصولنا إلى مزايا معينة؛ وفقًا للصلاحيات الممنوحة للمستخدم الذي يشغل البرنامج، فإذا أردنا الوصول إلى ملفات أحد المستخدمين فلا بد أن نملك صلاحية الوصول إلى ملفاته أولًا، وهذا لا يعني ترك مزايا الأمان المضمنة في نظام التشغيل.

سندرس بعض الدوال المتعلقة بالأمان، مثل تحديد معرِّف المستخدم، وتغيير ملكية الملفات، واستخدام متغيرات البيئة للحصول على بيانات حول بيئة المستخدم الحالي.

25.6.1 المستخدمون وملكية الملفات

يمكن الحصول على معرِّفات المستخدمين باستخدام دالة `os.getuid`، والتي تعيد معرّف المستخدم في صورة رقم، ولا نحتاج إلى تحويله إلى اسم المستخدم نفسه إلا نادرًا،لأننا نستطيع الحصول على ذلك الاسم باستخدام دالة `getpass.getuser()` التي تنظر في متغيرات البيئة التي قد تحمل تلك المعلومة.

```
>>> import getpass
>>> print( getpass.getuser() )
```

أما معرف المستخدم فيكون القيمة التي يحتاج البرنامج إليها لتغيير إعدادات الأمان، ونحصل عليه كما يلي:

```
>>> import os
>>> print( os.getuid() )
```

لعل أشهر استخدام لذلك هو تغيير ملكية ملف -أنشأناه سابقًا في جزء من برنامجنا- برمجيًا، فمثلًا سنستخدم أحد الملفات التي أنشأناها سابقًا في هذا الفصل:

```
import os

os.chdir('src/Python/Root')
os.system('ls -l *.txt')
id = os.getuid()
os.chown('FA.txt',id,-1)
os.system('ls -l *.txt')
```

نستخدم `os.chdir` لضبط مجلد العمل ليكون المكان الذي فيه الملفات، ثم نستخدم `system()` لعرض قائمة المجلدات بما فيها صلاحيات الملفات قبل وبعد استدعاء `chown()`، لنستطيع أن نرى التغيرات إن وجدت.

نستدعي `chown()` مع معرف المستخدم الذي حصلنا عليه من `getuid()`، ونستخدم `-1` للمعامل الثالث الخاص بالدالة `chown()`، للإشارة إلى أننا لا نريد تغيير ملكية المجموعة، لكن إذا أردنا ذلك فسنستخدم الدالة `os.getid()` التي تجلب معرف المجموعة.

لاحظ أنه لن يكون للسكربت تأثير إلا إذا شغلناه من مستخدم مختلف عن الحالي، ويجب أن يكون للمستخدم صلاحيات تلك التغييرات، لذا ننصح أن تسجل الدخول بالمستخدم المدير administrator أو الجذر root.

لا تخبرنا `chown()` أي معلومات عن الخرج، لذا إذا أردنا التحقق من النتيجة فيجب أن نستخدم شيئًا مثل `os.stat` للتحقق من قيمة معرف المستخدم قبل استدعائها وبعده، وبهذا نتحقق من حدوث التغييرات التي نتوقعها.

25.6.2 بيئة المستخدم

يرث البرنامج عند بداية تشغيله سياق الذاكرة من البرنامج الذي شغّله، حيث يكون البرنامج المشغِّل غالبًا صدفة سطر الأوامر الخاصة بالمستخدم، CMD في ويندوز أو Bash أو غيرها في يونكس، وتشمل بيئة المستخدم تلك معلومات كثيرةً عن النظام، مثل اسم المستخدم، ومجلد home، والمجلد الحالي، والمجلد المؤقت، ومسارات البحث، وهذا يعني أن إعداد متغيرات البيئة المختلفة يمكّن كل مستخدم من تخصيص كيفية عمل نظام التشغيل وبرامجه إلى حد ما، فمثلًا تراقب بايثون متغير البيئة `PYTHONPATH` عند البحث عن الوحدات، لذلك قد يكون لكل مستخدم على نفس الحاسوب مسارات بحث مستقلة للوحدات، بما أن كل واحد منها يضبط قيمةً مختلفةً للمتغير `PYTHONPATH`، ويستفيد المبرمجون من ذلك بتعريف بعض متغيرات البيئة لبرنامج ما بحيث يستطيع المستخدم تغيير قيم البرنامج الافتراضية، ويجب أن نتمكن من قراءة البيئة الحالية للعثور على تلك القيم، بأن نقرأ متغيرًا واحدًا باستخدام دالة `os.getenv()`، أو نقرأ جميع المتغيرات المضبوطة حاليًا بالنظر في متغير `os.environ` الذي يحتوي قاموسًا بأزواج الاسم/القيمة.

سنطبع قائمةً بجميع متغيرات البيئة، حيث تحتوي تلك القائمة على معلومات كثيرة:

```
>>> import os
>>> print( os.environ )
```

لا شك أننا نستطيع استخدام عمليات القاموس والسلاسل النصية المعتادة، لكن يُفضل في أغلب الحالات أن نحصل على قيم المتغيرات قيمةً قيمةً، كما يلي:

```
>>> os.getenv('PYTHONPATH')
```

أو:

```
>>> os.environ['PYTHONPATH']
```

يبين لنا هذا هل ضبطنا المتغير PYTHONPATH أم لا؟ وعلى أي قيمة أيضًا.

تتيح getenv() إمكانية ضبط نتيجة افتراضية إذا لم يكن المتغير معرَّفًا، نظرًا لخطورة عدم تعريفه، ويمكن استخدام تابع القاموس القياسي get() لتنفيذ نفس الشيء في قاموس environ، ويستخدَم هذا عادةً أثناء تهيئة البرامج، حين نضبط الإعدادات، مثل المجلد الذي ستوجد فيه ملفات البيانات، ففي المثال التالي سننظر أين يجب تخزين دليل جهات الاتصال الخاص بنا، باستخدام الإعداد الافتراضي للمجلد الحالي إن لم يكن ثمة متغير:

```
# ... خطوات تهيئة هنا.
folder = os.getenv('PY_ADDRESSES', os.getcwd())
# ... بقية البرنامج
```

تعيد getenv() وسيطها الثاني عند عدم وجود قيمة افتراضية للمتغير PY_ADDRESSES، وهو الموقع الافتراضي لنا.

ينشئ المستخدم متغيرات البيئة تلك ويضبطها يدويًا باستخدام نظام التشغيل، ففي ويندوز مثلًا ينفّذ ذلك بتسلسل الإعدادات التالي:

```
MyComputer->Properties->Advanced->Environment Variables
```

أما في لينكس وماك فيُنفذ في سطر الأوامر باستخدام الأمرين export وsetenv وفقًا للصدفة المستخدمة.

يمكن تغيير قيم متغير البيئة الحالي في بعض أنظمة التشغيل، لكن يجب استخدام ذلك بحذر، إذ قد يؤدي إلى الكتابة فوق قيم أخرى، كما قد تعكس بعض أنظمة التشغيل تلك التغييرات على بيئات المستخدمين، غير أن هذا لا ينطبق إلا في سياق عملية الكتابة غالبًا، فإذا كان نظام التشغيل يدعم تلك العملية فنستطيع كتابة قيمة المجلد الافتراضي الخاصة بنا إلى بيئة المستخدمين لضمان استخدام نسخ برنامجنا الأخرى لنفس الموقع، إلا أننا لا ننصح بذلك لهشاشة تلك الآلية، وإنما ننصح باستخدام ملف config لأنه أكثر موثوقية لمثل هذه الإعدادات.

```
# شيفرة أخرى كما سبق
putenv('PY_ADDRESSES', folder)
```

```
# ... بقية البرنامج
```

تُستخدم بعض متغيرات البيئة في يونكس من قبل برامج عديدة، منها على سبيل المثال:

- EDITOR: يحدد هذا المتغير برنامج التحرير الذي يفضله المستخدم، ويكون ed أو vi أو ViM أو Emacs، فتشغّل البرامج الأخرى هذا المحرر إذا احتاج المستخدم إلى تعديل ملف نصي في جزء من البرنامج، فمثلًا يمكننا استخدام `getenv ('EDITOR')` لإطلاق المحرر الخاص بالمستخدم في مثال `os.system` أعلاه، بدلًا من ضبط المحرر nano ليكون هو الافتراضي.
- PRINTER: يحدد هذا المتغير إعدادات المستخدم المفضلة لطباعة الملفات.
- PAGER: يحدد هذا المتغير برنامج عرض الملفات الذي يفضله المستخدم، ويُضبط غالبًا على more أو less أو view.

سنكتفي بهذا الشرح عن البيئات، وسنعود إليها في فصل لاحق، لكن نذكر أنه عند الحاجة إلى الحصول على بيانات تخص المستخدم فيجب التحقق من وجود متغير بيئة بالفعل أم لا، أو نتيح للمستخدم إمكانية ضبط ذلك من خلال متغير بيئة خاص ببرنامجنا.

25.7 مزيد من المعلومات حول نظم التشغيل

تحتوي وحدة `os` ومثيلاتها على إمكانيات أكثر مما يمكن أن نشرحه هنا، بل إن توثيق بايثون يحتاج إلى عدة صفحات HTML لشرح وحدة `os` وحدها، وصفحة لكل وحدة أخرى، ويمكن الرجوع إلى تلك التوثيقات لتصفح وظائف تلك الوحدات، وستجد فيها أسماءً غريبةً، تأتي أغلبها من يونكس وواجهة برمجة تطبيقاته.

توفر `os` وظائف مكافئةً على أي نظام تشغيل، لكن إذا أردت معرفة المزيد عن وظائف تلك الدوال فيجب الرجوع إلى توثيق يونكس نفسه، فإن لم يكن لديك أحد أنظمة يونكس فيمكن الرجوع إلى كتاب Unix Systems Programming for SVR4، وإذا أعجبك ما شرحناه عن نظم التشغيل فربما تود قراءة كتاب Fundamentals of Operating Systems لصاحبه ليستر A. M. Lister، فهو كتاب قصير ويسهل فهم شرحه المدعوم بمخططات توضيحية، أما إذا رغبت في إلقاء نظرة فاحصة على نظم التشغيل فانظر كتاب أندرو تانينباوم Andrew Tanenbaum الشهير Operating Systems: Design and Implementation، الذي شجع لينوس تورفالدز Linus Torvalds على كتابة نواة لينكس.

ننصح أيضًا بقراءة كتاب أنظمة التشغيل للمبرمجين فهو كتاب قيم ومهم لكل مبرمج ودارس لمجال أنظمة البرمجيات وهندسة البرمجيات.

25.8 خاتمة

نرجو في نهاية هذا الفصل أن تتذكر ما يلي:

- يوفر نظام التشغيل بيئةً يمكن تشغيل العمليات فيها.
- كما يوفر وصولًا إلى عتاد الحاسوب.
- يمكن الوصول إلى نظام التشغيل من خلال واجهة برمجة التطبيقات، التي تكتَب عادةً بلغة C.
- توفر وحدة `os` الخاصة ببايثون مغلِّفًا لواجهة برمجة التطبيقات الخاصة بنظام التشغيل.
- تسهل الوحدتان `os.path` و`glob` الوصول إلى الملفات.
- توفر كـل من `os.system()` و`subprocess.Popen` مسـتويات مختلفـةً من التحكم في العمليـات والتواصل بينها IPC.
- تسمح `getuid()` و`getenv()` والمزايا الشبيهة بهما بالتعرف على المستخدم وإعداداته المفضلة.

26. التواصل بين العمليات في البرمجة

سنستخدم في هذا الفصل الدالة `os.fork()` التي لا توجد في نظام ويندوز، لأنه بطيء جدًا -موازنةً بأنظمة يونكس- في إنشاء العمليات الجديدة، إلى درجة عدم استخدام `fork()` حتى لو استطعنا ذلك، لكن يمكن حل هذه المشكلة باستخدام الخيوط threads، التي تعمل على كل من يونكس وويندوز، وسنشرحها في فصل لاحق، فإذا كنت تستخدم ويندوز فيفضّل أن تقرأ القسم الخاص بالمفاهيم هنا لأنها ستُستخدم في الفصل التالي، لكن لا داعي لكتابة الأمثلة الموجودة هنا لأنها لن تعمل معك، أما إذا كنت تستخدم أحد أنظمة يونكس فلن تواجهك مشكلةً فيها.

سنشرح في هذا الفصل:

- الحالات التي تستدعي IPC وأسبابها.
- أساسيات الأنابيب pipes.
- نسخ العمليات باستخدام `os.fork()`.
- التواصل من خلال الأنابيب.
- إنهاء العمليات باستخدام `os.kill()`.

26.1 تعريف التواصل بين العمليات

التواصل بين العمليات inter-process communication واختصارًا IPC، هو آلية تمكّن عمليةً ما من التواصل وتبادل البيانات مع عملية أخرى، وقد شرحنا في الفصل السابق أن العملية برنامج تنفيذي، وأنها تستطيع التواصل مع غيرها من العمليات باستخدام المزايا الموجودة في وحدة `subprocess`، وعلى الرغم من أن هذه التقنيات مفيدة في التواصل مع البرامج الأخرى، إلا أنها لا توفر التحكم الدقيق المطلوب أحيانًا

للتطبيقات الكبيرة، فمن الشائع في تلك التطبيقات استخدام عدة عمليـات في نفس الـوقت، بحيث تنفـذ كـل منها مهمةً مستقلةً، وتطلب بقية العمليات خدمات منها، فمثلًا قد يكون لخـادم الـويب عمليـة تـراقب طلبـات الويب من المتصـفحات وتخـدمها في صـفحات HTML بسـيطة، لكنـه يسـتخدم عمليـةً أخـرى لتوفـير طلبـات البيانات الأعقد، وربما عملية أخرى لمعالجة طلبات ftp، وتُصمم كل عملية بحيث تنفذ مهمةً واحدةً فقط بكفاءة عالية، ويسمح هذا التصميم للمدير بمشاركة حمل المعالجة مع عدة عمليات، فإذا كـان لـدينا طلبـات ftp كثـيرة مثلًا، فيمكن بدء عملية ftp جديدة، وتوزيع طلبات ftp على العمليتين.

26.1.1 أهمية التواصل بين العمليات

مع أننا قد لا نكتب مثل تلك التطبيقات الكبيرة، إلا أنها تقنيـة مفيـدة، وسـنحتاج المفـاهيم الـتي فيهـا في فصول لاحقة، وقد نستفيد منها في برامج يتشارك فيها عدة مستخدمين أحد الموارد، كقاعدة بيانات مثلًا، حـتى لو كانت تلك البرامج صغيرةً، وكذلك عند قراءة البيانات من عدة منافذ اتصال، حيث سيكون من الأفضل وجـود عملية لكل اتصال شبكي، كي لا يؤثر انقطاع إحدى الشبكات على قراءة بقية الشبكات.

26.1.2 هل تشبه هذه التقنية تقنية العميل/الخادم؟

يكثر استخدام مصطلح العميل/الخادم في التطبيقات التجارية، وهو مصطلح خـاص بمعماريـة البرمجيـات، ويشير إلى نوع محدد من إعدادات التواصل بين العمليات، ترسـل فيـه إحـدى العمليـات -وهي العميـل client- طلبات إلى عملية أخرى -هي الخادم server-، لكن الخادم لا يطلب أي شيء أبدًا من العميـل، وقـد يكـون لـدينا عدة عمليات من نوع "العميل" تصل إلى خادم واحد، ويمكن لذلك الخادم نفسـه أن يكـون عميلًا لخـوادم أخـرى، فيما يعرف بحوسبة العميل/الخادم متعدد المسـتويات N-Tier Client/Server، الـتي تسـتخدم التواصـل بين العمليات، إلا أنه لا يُشترط أن تكون تقنية التواصل بين العمليات مبنيةً على تقنية العميل/الخادم، ومن الممكن أن يكون لدينا شبكة من العمليات ترسل كل منها رسائل إلى بقية العمليات دون أي تخطيـط ثـابت بين العملاء والخوادم، ويُسمى هذا بحوسبة النِد للنِد peer-to-peer computing، وقد يبدو هذا نموذجًا جذابًا إلا أنه صعب الإدارة إذا زاد عـدد العمليـات، لـذا يظـل النمـوذج التقليـدي للعميـل/الخـادم هـو الأكـثر اسـتقرارًا والأسـهل في تصميمه وتشغيله.

26.1.3 هل لهذه التقنية علاقة بالعتاد؟

تكلمنا في وصفنا السابق للعميل والخادم عن البرمجيات والعمليات فقط، لكن ربما تكون قد سمعت هـذه المصطلحات في العتاد أيضًا، خاصةً مصطلح الخادم server الذي يشير إلى الحواسـيب القويـة الـتي يتشـاركها عدة مستخدمين، وهذه إساءة إلى المعنى الحقيقي لحوسبة العميل/الخادم الـتي هي معماريـة برمجيـة خالصـة، ومستقلة عن العتاد.

تميل عمليات الخوادم في الواقع العملي إلى العمل على حواسيب مستقلة وقوية، ويُشار إلى الجمع بين العتاد وعمليات الخوادم باسم الخادم، ويأخذنا هذا الأسلوب من استخدام تقنية العميل/الخادم إلى عالم حوسبة الشبكات الذي ننظر فيه في الفصل التالي.

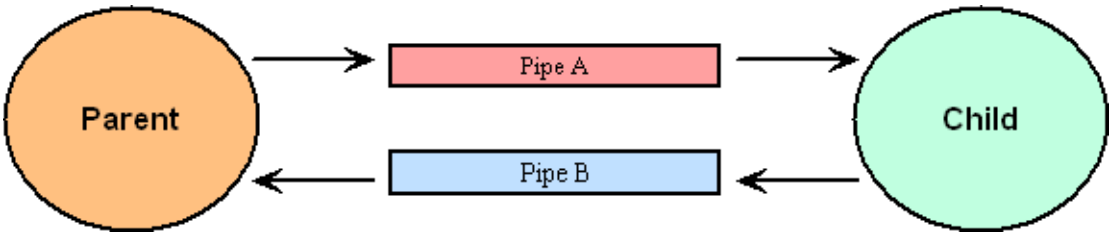
يمكن ملاحظة تشابه هذه المصطلحات مع تلك المستخدمة في البرمجة كائنية التوجه، لأن استخدام الرسائل بين العمليات يشبه إرسال الرسائل بين الكائنات في البرمجة الكائنية، وفعليًا توجد الكثير من الأمور المشتركة بين النموذج الكائني ونموذج التواصل بين العمليات، وقد طُورت بعض معماريات IPC لتستفيد من هذه الأوجه المشتركة، مثل معمارية وسيط طلب الكائن المشترك Common Object Request Broker أو COBRA اختصارًا، حيث نسجل الكائنات في تلك المعمارية مع وسيط طلب لكائن مركزي ORB، وتُوجَّه الرسائل المرسَلة إلى الكائن من أي مكان في الهيكل إلى العملية التي سجلت الكائن، ولن ننظر فيها هنا، ويمكن العودة إلى تطبيقات بايثون لوسطاء طلبات الكائنات المركزية ORB لمزيد من المعلومات.

سنبدأ الآن كتابة بعض الشيفرات، لكن يجب أن ننظر أولًا في بعض آليات التواصل بين العمليات، وسندرس اثنتين منها، الأولى هي تقنية الأنبوب pipe المستخدمة لتبادل البيانات بين عمليتين.

26.2 تعريف الأنبوب Pipe

يمكن النظر إلى الأنبوب Pipe على أنه يشبه الخرطوم، حيث نسكب البيانات في أحد طرفيه ليخرج من الآخر، وهو يشبه الملف من حيث التعامل معه على أنه تدفق متسلسل للبيانات، لكنه يختلف عن الملف في أن له طرفين، لذا نحصل عند إنشائه على نقطتي نهاية، لنقرأ من إحداهما، ونكتب إلى الأخرى. وعلى عكس الملف لا تُخزّن البيانات حقيقةً عند إغلاق الأنبوب، لذا سنفقد كل ما كتبناه في أحد طرفي الأنبوب ولم نقرأه من الجانب الآخر.

يمكن توضيح استخدام الأنابيب مثل قنوات بين العمليات كما يلي:



نرى هنا عمليتين سميناهما `Parent` و`Child`، لأسباب ستبينها بعد قليل، وتستطيع عملية `Parent` أن تكتب في `Pipe A`، وتقرأ من `Pipe B`، أما عملية `Child` فتستطيع القراءة من `Pipe A` والكتابة في `Pipe B`، ويُستخدم كل أنبوب لإرسال طلب أو إعادة بيانات وفقًا للعملية التي بدأت عملية التبادل، ويعطينا ذلك تحديًا مثيرًا للاهتمام في سبب تسمية الأنابيب.

سنكتب الآن بعض التعليمات البرمجية لنرى كيف يمكن بناء آلية IPC باستخدام بايثون، وسيكون المبدأ العام هو إنشاء عملية تكون أصلًا parent وتفتح أنبوبين، ثم نشتق نسخةً منها لتكون فرعًا فيه نفس الأنبوبين، ثم نستخدم الأنابيب للتواصل بين الأصل والفرع، وأول ما سنفعله هو معرفة كيفية استخدام الأنابيب لإرسال البيانات واستلامها، حيث سننشئ أنبوبًا باستخدام دالة `os.pipe()` تعيد اثنين من واصفات الملفات، واحد لكل طرف من أطراف الأنبوب، ثم نستخدم دوال `os.read/write` لإرسال البيانات في الأنبوب:

```
import os

# أنشئ الأنبوب
receive, transmit = os.pipe()

data = 'Here is a data string'
length = os.write(transmit, bytes(data, 'utf8'))
print('Length of data sent: ', length)

# مخزن مؤقت حجمه 1024
# لضمان استقبال جميع البيانات
print( 'The pipe contains:', os.read(receive, 1024).decode('utf8') )
```

لاحظ أننا نحتاج إلى تحويل البيانات من وإلى مصفوفات البايتات byte arrays، تحويلًا مشتركًا مع معظم عمليات مستوى النظام، ويحدث كل ذلك في عملية واحدة، لذا لا يكون ذا فائدة كبيرة، لكننا نستطيع فصل شيفرة القراءة والكتابة والبدء في تنفيذ شيء ما، بمجرد استنساخ عمليتنا، فكيف ننشئ عملية الاستنساخ تلك؟

26.3 عمليات الاستنساخ

إن الآلية المستخدمة في توليد الفروع spawning أو اشتقاقها forking هي استخدام استدعاء النظام `os.fork()` الذي يعيد قيمًا مختلفةً وفقًا لمكاننا في العملية الأصل أو الفرع، وتكون قيمة الإعادة في العملية الأصلية هي معرِّف العملية أو pid للعملية الفرع، اما إذا كنا في العملية الفرع فستكون قيمة إعادة `fork` صفرًا، وهذا يعني أنه سيكون لدينا في الشيفرة تعليمة `if` تتحقق من قيمة إعادة `fork()`، فإن كانت صفرًا فستنفَّذ دوال العملية الفرع، وإن لم تكن كذلك فستنفَّذ دوال العملية الأصل، ويفضَّل وضع الدوال في وحدات منفصلة واستدعاؤها حسب الحاجة، للسيطرة على مجريات التنفيذ، إلا أننا لن نفعل ذلك هنا لأن الشيفرة قصيرة وستكفينا قائمة واحدة، ونعيد التذكير هنا أن ويندوز لا يفعل هذا، أي لا يدعم دالة `fork()`، لذا لن تعمل الشيفرة في ويندوز، وسيضطر مستخدم ويندوز إلى الانتظار إلى الفصل التالي ليعرف كيفية كتابة برامج العميل/الخادم.

سننشئ الآن عمليةً فرعيةً تستطيع إجراء عملية بسيطة لتنسيق النصوص، وتعيد القيمة التي نمررهـا إليهـا مسبوقةً ومتبوعةً بالجملة Ni.

```
import os,signal

# أنشئ الأنابيب

ServerReceive,ClientSend = os.pipe()
ClientReceive, ServerSend = os.pipe()

pid = os.fork()
if pid == 0:  # في الفرع
   while True: # قدمها بلا نهاية
      data = os.read(ServerReceive,1024)
      if data:  # تحقق من استلام شيء ما!
         data = 'Ni!\n' + data + '\nNi!'
      else: data = "Error: received empty message"
      os.write(ServerSend,data)
else: # في الأصل
   data = ['The Knights who say Ni!',
           'Appear in the film "Monty Python and the Holy Grail" ']
   for line in data:
       os.write(ClientSend,line)
       print os.read(ClientReceive,1024)

   # والآن أنه العملية الفرع
   os.kill(pid,signal.SIGTERM)
```

لاحظ أننا نستخدم pid الذي استلمناه من fork لإنهاء العمليـة الفـرع، فـإذا فشـلنا في ذلـك فستصـبح العملية الفرع عمليةً خلفيةً أو خفيةً daemon، تعمل بصمت بلا نهاية بانتظار ظهور البيانات على أنبوب الـدخل الخاص بها، أما الإنهاء الفعلي فيكـون باسـتخدام دالـة os.kill()، الـتي تُسـتخدم لإرسـال أي رسـالة إلى أي عملية -بغض النظر عن اسمها-، وتسمى إشارة الإنهاء هنا SIGTERM كما هو محدد في وحدة signal، وتختلف القائمة الكاملة وفقًا لكل منصة، وتكون موثقةً في مكتبة libc الخاصة بلغـة C لمنصـتك، لكن يمكن الحصـول عليها بسهولة كما يلي في محث بايثون:

```
>>> dir(signal)
```

أو في سطر أوامر يونكس (لاحظ أن هذا حرف L وليس العدد 1):

```
$ kill -l
```

ونحصل في حالة يونكس على القيمة العددية التي يمكن استخدامها في أمر `kill()` مباشرةً، لكن هذه القيمة لن تعمل في منصات مختلفة.

أنشأنا الأنابيب ونقلنا البيانات عبرها، كما أنشأنا عمليةً فرعيةً وأرسلنا البيانات إليها، وعدلنا البيانات في العملية الفرعية وأرسلنا البيانات إلى العملية الأصل، وأخيرًا أنهينا العملية الفرع، وهذا كل ما نحتاج إليه في التواصل الأساسي بين العمليات، أما الآن فسنرى مثالًا حقيقيًا عمليًا، نعود فيه إلى برنامج دليل جهات الاتصال مرةً أخرى.

26.4 دليل جهات الاتصال بتقنية العميل/الخادم

بنينا نسخةً من دليل جهات الاتصال في فصل التعامل مع الملفات باستخدام قاموس، وسنعيد استخدام ذلك المثال لكننا سنبني هذه المرة نسخة العميل/الخادم.

لاحظ أن الشيفرة الأصلية كسرت إحدى قواعد الممارسة السليمة التي ذكرناها من قبل، وهي أننا أدرجنا شيفرة واجهة المستخدم في دوالنا المساعِدة، فإذا كنا سنستخدم تلك الشيفرة مرةً أخرى فسنحصل على رسائل مختلطة من العملية الفرعية والعملية الأصل، ونريد أن نعدل الشيفرة قليلًا كي نحولها إلى نموذج يمكن استخدامه، والمهم هنا إزالة أي تعليمة `print` من الدوال، وتمرير البيانات وسطاء، كما نريد إعادة النتيجة من كل دالة، بمجرد إنهاء ذلك نستطيع استيراد الشيفرة والوصول إلى الدوال المساعدة دون تنفيذ دالة `main()`، وعلى ذلك تكون الدوال التي سنوفرها هي:

- `readBook(filename)`
- `saveBook(book, filename)`
- `addEntry(book, name, data)`
- `removeEntry(book, name)`
- `findEntry(book, name)`

وستبدو الشيفرة المعدلة كما يلي:

```
def readBook(filename='addbook.dat'):
    import os
    book = {}
    if os.path.exists(filename):
       store = open(filename,'r')
```

```
        for line in store:
            name,entry = line.strip().split(':')
            book = entry
    else:
        store = open(filename,'w') # أنشئ ملفًا فارغًا جديدًا
    store.close()
    return book

def saveBook(book, filename = "addbook.dat"):
    store = open(filename,'w')
    for name,entry in book.items():
        line = "%s:%s" % (name,entry)
        store.write(line + '\n')
    store.close()

def addEntry(book, name, data):
    book = data
    return 'Added entry for ' + name

def removeEntry(book, name):
    del(book)
    return 'Deleted entry for ' + name

def findEntry(book, name):
    if name in book.keys():
        result = "%s : %s" % (name, book)
    else: result = "Sorry, no entry for: " + name
    return result
```

لاحظ أننا تجاهلنا دوال واجهة المستخدم لأننا لا نحتاج إليها هنا، لكن قد نرغب في إجراء التعديلات المناسبة للسماح لها بالعمل مثل برنامج منفصل، يكون وحدةً لنسخة العميل/الخادم، ولتكتب ذلك ليكون تدريبًا عمليًا لك.

بمجرد أن نصلح الشيفرة وتكون صالحةً للعمل في برنامج مستقل، أو بحفظ الشيفرة أعلاه في ملف address_srv.py، وهو ما فعلناه، نستطيع أن نتابع كتابة شيفرة العميل/الخادم، وستتكون من الهيكل القياسي لإنشاء الأنابيب واشتقاق العملية، حيث سنقرأ في العملية الفرعية الأنبوب الوارد، ونفسر البيانات على أنها أمر متبوع بالوسطاء التي توفرت، ثم نستدعي دالة قاعدة البيانات المناسبة، وسنعرض -سواء في العملية

الأصل أو الفرع- على المستخدم قائمةً، ونطلب أي بيانات إضافية وفقًا لما اختير قبل إرسال سلسلة البيانات المدمجة إلى العملية الفرع أو الخادم، ثم يقرأ العميل الاستجابة ويقدمها إلى المستخدم، وسيبدو البرنامج الرئيسي كما يلي:

```
import os, signal, address_srv

fromClient,toServer = os.pipe()
fromServer,toClient = os.pipe()

pid = os.fork()
if pid == 0:    # نحن الخادم
   addresses = address_srv.readBook()
   while True:
      s = os.read(fromClient,1024)
      cmd,data = s.split(':')
      if cmd == "add":
         details = data.split(',')
         name = details[0]
         entry = ','.join(details[1:])
         s = address_srv.addEntry(addresses, name, entry)
         address_srv.saveBook(addresses)
      elif cmd == "rem":
         s = address_srv.removeEntry(addresses, data)
         address_srv.saveBook(addresses)
      elif cmd == "fnd":
         s = address_srv.findEntry(addresses, data)
      else: s = "ERROR: Unrecognized command: " + cmd
      os.write(toClient,s)
else:     # We are the client
   menu = '''
Add Entry
Delete Entry
Find Entry

Quit
   '''
```

```
    while True:
       print( menu )
       try: choice = int(input('Choose an option[1-4] '))
       except: continue
       if choice == 1:
          name = input('Enter the name: ')
          num = input('Enter the House number: ')
          street= input('Enter the Street name: ')
          town = input('Enter the Town: ')
          phone = input('Enter the Phone number: ')
          data = "%s,%s,%s,%s,%s" % (name,num,street,town,phone)
          cmd = "add:%s" % data
       elif choice == 2:
          name = input('Enter the name: ')
          cmd = 'rem:%s' % name
       elif choice == 3:
          name = input('Enter the name: ')
          cmd = 'fnd:%s' % name
       elif choice == 4:
          break
       else:
          print( "Invalid choice, must be between 1 and 4." )
          continue

       os.write(toServer, cmd)
      print( "\nRESULT: ", os.read(fromServer,1024) )

    os.kill(pid, signal.SIGTERM)
```

لا شك أننا نستطيع ترتيب تلك الشيفرة لتكون أفضل باستخدام بعض دوال سلاسل `if/elif`، لكن المثال أصغر من أن يستدعي ذلك، ويجب ملاحظة بعض النقاط هي:

- استخدام `break` لإيقاف الحلقة التكرارية.
- استخدام `continue` للعودة إلى قمة الحلقة مرةً أخرى.

وقد شرحنا ذلك باختصار في فصل مقدمة في البرمجة الشرطية، ويُرجع إلى توثيق كل منها للمزيد من التفاصيل.

من الممكن توسيع هذا التدريب ليستخدم نسخة قاعدة البيانات التي شرحناها من قبل لتكـون هي الخـادم بدلًا من نسخة القاموس، وسنترك ذلك تدريبًا للقارئ.

تكمن العقبة في مثل هذا النمـوذج من برمجـة العميـل/الخـادم في أن الخـادم لا يسـتطيع التحـدث إلا مـع العميل الذي بدأه، مع أنه يسهل إنشاء آلية لإنتاج عمليـات إضـافية تـدرك أنهـا يجب أن تتصـرف مثـل عملاء لا خوادم، وكان من الأفضل بدء تشغيل الخادم أولًا ثم إلحاق عدة عملاء به، وهو ما سنفعله في نهاية الفصل التالي حيث نشرح الشبكات بمزيد من التفصيل.

26.5 خاتمة

في نهاية هذا الفصل نأمل أن تكون تعلمت ما يلي:

- تستطيع العمليات أن تتواصل فيما بينها بواسطة الأنابيب.
- تستطيع العمليات الأصل أن تستنسخ نفسها باستخدام `os.fork`.
- يجب إنهاء العمليات الفرعية وإلا ستعمل بلا نهاية، وتستهلك موارد الحاسوب، وننهيها باسـتخدام دالـة `os.kill`.

27. تواصل البرامج والعمليات البرمجية عبر الشبكة

نظرنا حتى الآن في هذا الكتاب في نظام التشغيل وقدرته على إدارة العمليات، وفي كيفية جعل السكربتات تنفذ برامج موجودةً من قبل، وكذلك استنساخ البرامج والتواصل بين تلك النسخ باستخدام الأنابيب pipes، وسندرس في هذا الفصل الاتصال بالعمليات التي تعمل على حاسوب آخر عبر شبكة ما، أو بعمليات قد تكون جاريةً على نفس الحاسوب، حيث ستكون الشبكة هنا شبكةً منطقيةً logical network، وهذا النوع هو المثال الأول الذي سننظر فيه. وعلى ذلك سنشرح في هذا الفصل ما يلي:

- مقدمةً بسيطةً عن الشبكات.
- أساسيات المقابس sockets.
- إنشاء عملية الخادم server process.
- إنشاء عملية العميل client process.
- التواصل من خلال المقابس.

27.1 مقدمة في الشبكات

يعمل خادم الويب web server على حاسوب في مكان ما في الشبكة، ونستطيع الوصول إليه من حاسوبنا إذا كان لدينا عنوان الويب Uniform Resource Locator واختصارًا URL، والذي هو نوع من عناوين الشبكات يحوي معلومات حول العنوان في الشبكة، واللغة التي يتحدث بها -أو البروتوكول protocol-، وكذلك مكانه على خادم الملفات التي نريد جلبها، وسنفترض أن للقارئ خلفيةً أساسيةً عن الإنترنت، ويعرف أن للحواسيب المتصلة بها عناوين.

لكن كيف تتصل تلك الحواسيب ببعضها؟

طبعًا لا يتسع المقام هنا لشرح مفصل حول الشبكات، لذا يُرجع في هذا إلى أكاديمية حسوب التي فيها شرح ماتع في أساسيات الشبكات يمكن النظر فيه، وكذلك في هذا المصدر بالإنجليزية، وخلاصة ما نريد قوله في الشبكات هو أنه عند حاجة حاسوبين على شبكة ما للاتصال ببعضهما فإنها يتصلان من خلال إرسال حزمة بيانات من حاسوب لآخر، وتشبه تلك الحزمة إلى حد كبير مظروفًا يُرسل في طرد عبر البريد مع ورقة بداخله، حيث تمثل تلك الورقة البيانات، ويمثل المظروف ترويسة الطرد packet header التي تحوي عناوين المرسل والمستقبل، ويحدد جهاز التوجيه router أو المحول switch موقع الحاسوب المستقبِل على الشبكة، ثم يوجه الحزمة إلى جهاز توجيه أو راوتر في تلك المنطقة، وتصل الحزمة في النهاية إلى نفس الجزء من الشبكة الذي يحوي الحاسوب الهدف، ويتعرف الحاسوب الهدف على عنوانه ويفتح الطرد أو الحزمة، ثم يرسل حزمة تأكيد مرةً أخرى إلى المرسل ليخبره أن الرسالة قد وصلت.

وللحزم قيمة عظمى لا تتعداها، فيما يتعلق بحجم البيانات التي ترسلها، خلافًا للخدمات البريدية التقليدية، ويمكن تشبيه ذلك بخدمة لإرسال خطاب يتكون من ورقة واحدة فقط، أما الرسائل الطويلة فنحتاج إلى إرسالها في عدة خطابات يحوي كل منها ورقةً واحدةً فقط، ويجب على الطرف المستقبل أن يجمع تلك الأجزاء بالترتيب، حيث يضاف رقم متسلسل إلى الورقة -مثل ترقيم الصفحات-، وذلك لتُرسَل رسالة خطأ من المستقبل إلى المرسل إذا لم تصل صفحة ما أو لم تظهر في الوقت المحدد لها، وعلى الرغم من انتفاء الحاجة إلى ذلك غالبًا، لأن الحاسوب يتكفل به وكذلك نظام التشغيل وبرمجيات الشبكات، لكن هذه الأمور تستحق أن نعلمها على أي حال، نظرًا لتعذر الاعتماد على موثوقية نقل البيانات أو استمراريته عند استخدام شبكة ما، إذ تقع الأخطاء العرضية بلا شك، ويجب أن نكون مستعدين لفقدان البيانات أو تلفها.

27.2 الاتصال بالشبكة

لنترك الكلام المجرد ولندرس التفاصيل العملية لكيفية كتابة تطبيق متصل بالشبكة، حيث نحتاج إلى إنشاء برنامج يعمل مثل خادم server ويكون على حاسوب ما، وبرنامج للعميل client على حاسوب أو عدة حواسيب أخرى متصلة بالشبكة التي يتصل بها الخادم، كما نحتاج إلى آلية تتيح التواصل بين البرنامجين وتعمل في كامل الشبكة، وقد رأينا أن لكل حاسوب عنوانًا، يتكون من عنوان IP الذي يحوي أربعة أرقام تفصل بينها نقاط، لا شك أننا رأيناها من قبل في عناوين الويب، ويضيف التطبيق المتصل بالشبكة عنصرًا آخر إلى ذلك العنوان، يُعرف باسم المنفذ port.

27.2.1 المنافذ والبروتوكولات

يُحدَّد المنفذ بنقطتين رأسيتين : متبوعتين برقم المنفذ، ويضاف ذلك إلى عنوان IP العادي، فنصل مثلًا إلى المنفذ 80 في العنوان `127.0.0.1` بالشكل `127.0.0.1:80`، وتُحفظ بعض أرقام المنافذ لأغراض خاصة تكون في الغالب لبروتوكولات تطبيقات الإنترنت المختلفة، والبروتوكول -أو الميثاق- هو مجموعة من القواعد وتعريفات الرسائل التي تحدد كيفية عمل الخدمة، فالمنفذ 80 هو المنفذ القياسي المستخدم في خوادم الويب

لطلبات http، أما المنفذ 25 فيُستخدم لبريد SMTP، وبناء عليه يتصرف الحاسوب مثل خادم لبعض الخدمات في نفس الوقت، بعرض تلك الخدمات من خلال منافذها المختلفة، ويمكن توضيح هذا بسهولة بإضافة رقم المنفذ 80 إلى عنوان خادم الويب مثل http://www.google.com:80، حيث ينبغي أن تفتح الصفحة بلا مشاكل لأن المتصفح يتصل بالمنفذ 80 افتراضيًا إن لم يتوفر منفذ آخر، لهذا يكثر استخدام المنفذ 8080 مثل منفذ اختبار للإصدارات الجديدة من مواقع الويب قبل إطلاقها.

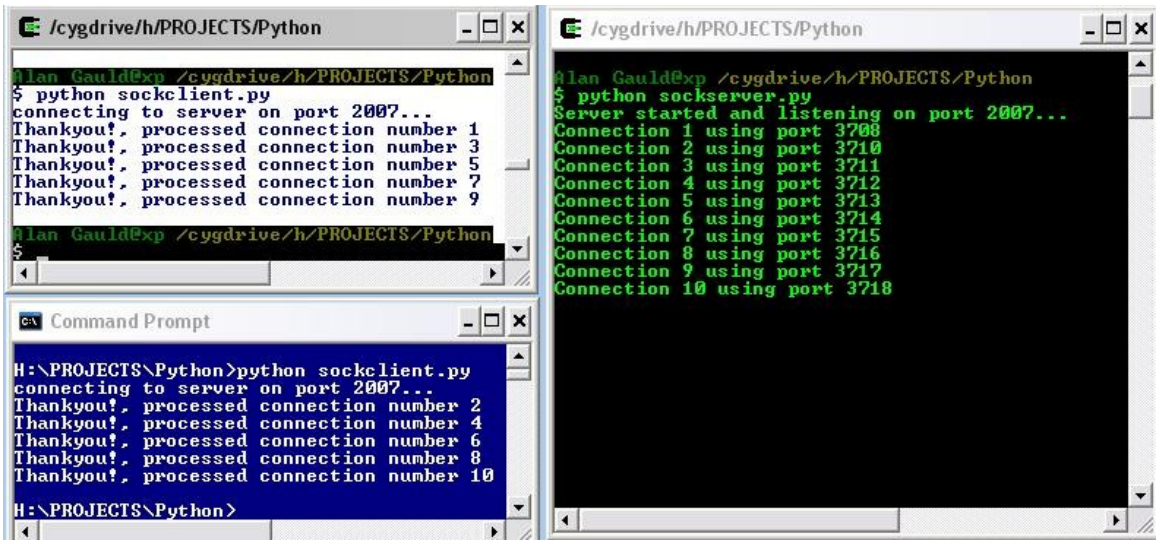
ورغم قلة عدد تلك المنافذ المحجوزة إلا أننا نستطيع استخدام أرقام المنافذ التي بين 1000 و60000 للتطبيقات المخصصة bespoke applications دون تعارض، لكن يفضل جعل رقم المنفذ قابلًا للإعداد من خلال متغير لبيئة النظام مثلًا، أو ملف إعدادات config file، أو بواسطة معامل سطر أوامر، نظرًا لوجود احتمال -ولو ضئيل- أن يختار برنامج آخر نفس المنفذ على الحاسوب، ولن نشرح ذلك هنا لكن يجب الانتباه إلى أنه وارد في التطبيقات الحقيقية، حين لا يكون لدينا تحكم كامل بحاسوب الخادم، لذا يجب اتخاذ مثل تلك الاحتياطات.

بعد أن عرفنا الآلية فإن السؤال التالي هو: كيف نصل الشيفرة بأحد تلك المنافذ؟

27.3 المقابس Sockets

المقابس هي أبسط آلية اتصال يمكن استخدامها بين الشبكات، وفيها يُقدَّم المقبس للشبكة على أنه منفذ في عنوان IP، وتُنشأ المقابس في بايثون وتُستخدم من خلال استيراد وحدة `socket`، ويجب أن نكتب خادمًا لينشئ المقبس، ويربطه مع منفذ، لنتمكن بعدها من استخدامه، ونراقب ذلك المقبس منتظرين الطلبات الواردة إليه، ثم نكتب عميلًا ليتصل بالمقبس على ذلك المنفذ، ثم يتصل العميل بالمنفذ ويقبل الخادم ذلك الاتصال، ثم ينشئ الخادم منفذًا مؤقتًا جديدًا يُستخدم لعملية التواصل -أي عمليات الإرسال والاستقبال send/recv الفعلية- مع الخادم أثناء عملية النقل، مما يحرر المنفذ لمزيد من طلبات الاتصال.

يمكن توضيح ما سبق في الصورة التالية:



تظهر هنا مشكلة اختبار مثل تلك التطبيقات، إذ لا نستطيع أن نعرف إن كان الخادم يعمل أم لا بدون العميل، كذلك فإن العميل لا يستطيع فعل شيء دون الخادم، لذا ينبغي أن يكون لدينا كل من العميل والخادم.

غير أنه يمكن إنشاء أي عدد من العملاء الآخرين بمجرد كتابة الخادم، شرط أن يتصلوا بمقبسه من خلال بروتوكول الرسائل المناسب، ونرى أمثلة ذلك في متصفحات الويب المختلفة التي تستطيع جميعها أن تتصل بأي خادم http، ويمكن بطريقة مماثلة كتابة العديد من الخوادم المختلفة بمجرد نشر البروتوكول، وينبغي هنا ألا توجد مشكلة في عمل أي عميل وأي خادم معًا، وهذا السلوك هو أحد أسباب انتشار تلك التطبيقات، فهو يشكل بيئةً مفتوحةً وقابلةً للتوسع، تكون فيها إحدى نهايات أي زوج من أزواج العميل/الخادم قابلةً للتحسين دون تعطيل النهاية الأخرى.

27.4 إنشاء الخادم

سننشئ خادمًا بسيطًا يستجيب للطلبات بأن يعيد رسالة ترحيب وعدد الطلبات التي عالجها من قبل.

```
import socket

# أنشئ مقبس InterNET و STREAMing
# المعروف باسم TCP/IP
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# استخدم localhost والمنفذ 2007
serversocket.bind(('localhost', 2007))

# استعد لاستقبال الطلبات
serversocket.listen(5)

connections = 0
while True:
  # عالج الاتصالات من العملاء
  (clientSocket, address) = serversocket.accept()
  connections += 1
  print( "Connection %d using port %d" % (connections, address[1]) )

  # افعل شيئًا باستخدام clientSocket
  while True:
     req = clientSocket.recv(100)
     if not req: break # client closed connection
```

```
        message = 'Thankyou!, processed connection number %d' %
connections
        clientSocket.send(message)
    clientSocket.close()
```

نلاحظ هنا عدة أمور:

- يشير هذا المزيج من `AF_INET` و`SOCK_STREAM` إلى أننا سنستخدم بروتوكول TCP/IP، وكل بروتوكولات عناوين IP الأخرى متاحة من خلال استخدام تجميعات ثوابت أخرى، لكن TCP/IP هو الأشهر منها وهو ما سنستخدمه.
- مررنا القيمة `5` إلى `listen()`، وهي تمثل العدد الأقصى للاتصالات التي يمكن أن تنتظر في طابور المنفذ، لأننا نُعالج الطلب قبل أن تتجمع طلبات كثيرة في قائمة الانتظار تلك، ونشتق عمليةً مستقلةً لتنفيذ المعالجة الحقيقية إذا أردنا تحسين كفاءتها -انظر الملاحظة الخامسة أدناه-، مما يسمح للخادم أن يسحب الرسائل من قائمة الانتظار في أسر ع وقت ممكن، ولا نحتاج زيادة عدد الاتصالات المسموح لها بالانتظار عن 5 اتصالات إلا في حالة الخوادم شديدة الزحام.
- ينشئ العملاء اتصالًا جديدًا لكل عملية تبادل بيانات، ولا تكون لدينا بيانات متاحة إذا انتهت عملية التبادل، وحينئذ ننهي حلقة `while` الداخلية ونعود إلى انتظار اتصال جديد.
- عالجنا طلب العميل في شيفرة الخادم، ولا بأس بهذا لأنها معالجة طفيفة، لكنها قد تستغرق وقتًا كبيرًا إذا كانت في أحد التطبيقات التجارية الكبيرة، عندها نشتق عمليةً أخرى في تلك الحالة لمعالجة تلك العملية خاصةً -ربما باستخدام وحدة `subprocess` التي ذكرناها في فصل نظم التشغيل-، ونترك الخادم يعود لسحب الطلبات من قائمة انتظاره.
- لا توجد طريقةً لإنهاء عملية الخادم، فهي تعمل بلا نهاية إلا إذا حدث خطأ، فإذا أردنا إنهاءها فسنستخدم أداةً من نظام التشغيل نفسه، من خلال برنامج TaskManager في ويندوز مثلًا، أو `kill` في يونكس.

ينبغي أن يكون الخادم جاهزًا للعمل الآن وينتظر طلبات العملاء، لكن ليس لدينا عميل ليرسل تلك الطلبات، لذا سننشئه الآن.

27.5 إنشاء العميل

إنشاء العميل client هي عملية سهلة بقدر سهولة إنشاء الخادم، إذ يرسل طلبات متكررةً إلى الخادم بفواصل زمنية قدرها ثانية واحدة، ويطبع استجابة الخادم:

```
import socket,time

```

```
# أنشئ المقبس
serverAddress = ('localhost',2007)

 # أرسل بعض الطلبات
for n in range(5):
   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
   sock.connect(serverAddress)
   try:
      sock.send('dummy request\n')
      data = sock.recv(100)
      if not data: break # لا توجد بيانات من الخادم
      print( data )
      time.sleep(1)
   finally:
      # نرتب الآن ما سبق
      sock.close()
```

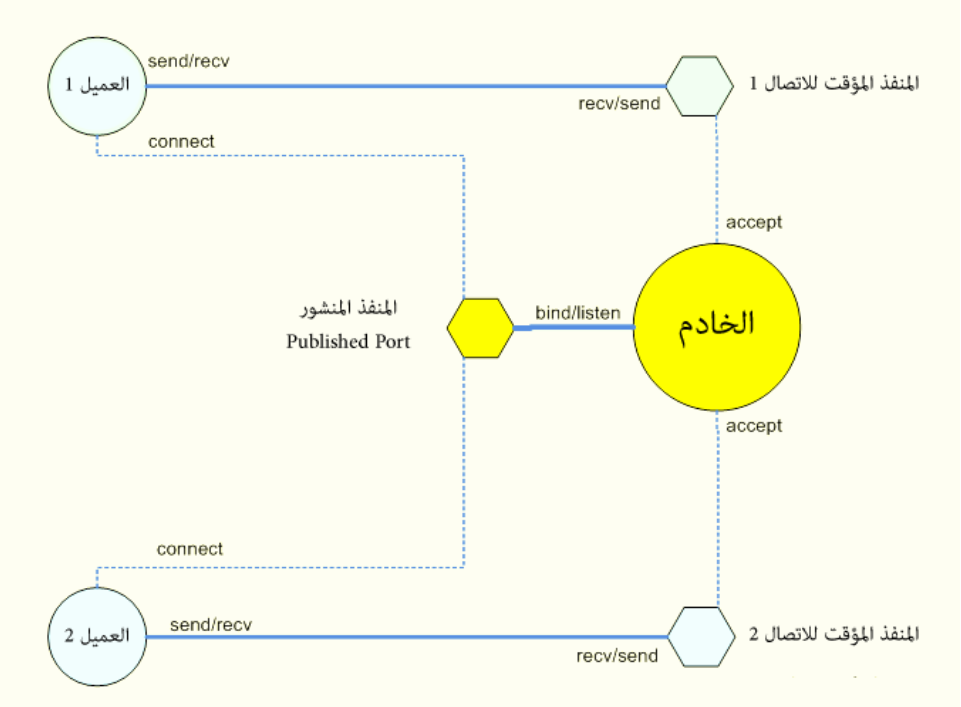
لدينا بعض الملاحظات هنا:

- نستخدم `connect` للوصول إلى المقبس، ثم نستخدم نفس واجهة `send/recv` التي يستخدمها الخادم، لكن يكون التسلسل معكوسًا لأن العميل هو الذي يبدأ عملية تبادل البيانات.
- كان من الممكن إرسال أو استقبال بيانات أكثر في عملية واحدة، لكننا اخترنا هذه الطريقة ببساطة لإبراز أرقام الاتصال المختلفة القادمة من الخادم، فالعميل هو الذي يقرر متى ينهي عملية التبادل وليس الخادم، إلا إذا حدث خطأ ما.
- لاحظ استخدام `try/finally` لضمان إغلاق المقبس حتى في حالة رفع استثناء exception، وهذا مفيد لتقليل المراجعة والتصحيح لاحقًا، لأن بعض أنظمة التشغيل تترك المقابس مفتوحةً لفترة طويلة، مما يعني أنها ستستهلك موارد النظام.

27.6 تشغيل البرامج

نريد أن نتأكد أن الخادم يعمل أولًا قبل أن نستطيع تشغيل البرامج، ثم نبدأ برنامج عميل واحد أو أكثر ليتصل به، ولا نستطيع تشغيل هذه البرامج عبر الشبكة لأننا جعلنا الخادم على الحاسوب المحلي فقط `localhost`، لذا نحتاج إلى بدء عدد من جلسات الطرفية على حاسوبنا.

توضح الصورة التالية الخادم وهو يعمل على يمين الصورة، واثنين من العملاء على يسارها:



لاحظ أن خرج كلا العميلين يُظهر تسلسل الرسائل التي استلمت من الخادم، وتُظهر رسائل الخادم الاتصالات وأرقام المنافذ المؤقتة المسندة إلى الخادم.

27.7 دليل جهات الاتصال الشبكي

بنينا في فصل التواصل بين العمليات نسخةً تعمل على خادم من دليل جهات الاتصال الذي نطوره، وسميناها address_srv.py، وسنستخدمها في هذا الفصل لبناء نسخة مبنية على المقابس، وسيكون الاختلاف الواضح بين نسخة IPC السابقة وبين هذه النسخة هو إمكانية وجود أكثر من عميل واحد يستطيع الوصول إلى دليل جهات الاتصال، وسيكون العملاء على حواسيب مختلفة قطعًا.

وكانت الدوال التي المتاحة في `address_srv` هي:

- `readBook(filename)`
- `saveBook(book,filename)`
- `addEntry(book, name, data)`
- `removeEntry(book, name)`
- `findEntry(book, name)`

27.7.1 برنامج الخادم

لا زلنا بحاجة إلى كتابة برنامج خادم يعالج الطلبات الـواردة من العملاء ويسـتدعي الدالـة المناسـبة، رغم أننـا حولنا البرنامج إلى دوال مخدمات server style functions في المرة السابقة، وتسمى مثل تلك الآليـة ببعث الرسائل dispatching messages، وستكون الشيفرة شبيهةً للغاية بالأمثلة البسيطة السابقة.

سيكون البرنامج الرئيسي كما يلي:

```
import socket, address_srv

addresses = address_srv.readBook()

# أعد المقبس
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 2007))
serversocket.listen(5)

print( 'Server started and listening on port 2007...' )

# عالج الاتصالات من العملاء
while True:
   (clientSocket, address) = serversocket.accept()

   # عالج أوامر دليل جهات الاتصال
   while True:
       s = clientSocket.recv(1024)
       try: cmd,data = s.split(':')
       except ValueError: break
       print( 'received request: ', cmd )
       if cmd == "add":
          details = data.split(',')
          name = details[0]
          entry = ','.join(details[1:])
          s = address_srv.addEntry(addresses, name, entry)
          address_srv.saveBook(addresses)
       elif cmd == "rem":
          s = address_srv.removeEntry(addresses, data)
```

```
            address_srv.saveBook(addresses)
        elif cmd == "fnd":
            s = address_srv.findEntry(addresses, data)
        else: s = "ERROR: Unrecognised command: " + cmd
        clientSocket.send(s)
    clientSocket.close()
```

نلاحظ هنا أن شيفرة المعالجة الرئيسية للمقبس هي نفسها الشيفرة السابقة، بينما وضعنا معالجة بيانات الطلب في بنية `try/except` لالتقاط البيانات غير المكتملة من العميل، أما غير ذلك فإن هذه النسخة تكاد تطابق نسخة IPC في الفصل السابق.

27.7.2 برنامج العميل

صار لدينا الآن خادم يعمل في الخلفية، ونريد كتابة برنامج عميل يتحدث إليه، وسيكون مشابهًا لنسخة IPC أيضًا، لكنه سيوجد في سكربت خاصة به، وسنتمكن من تشغيل أكثر من نسخة في نفس الوقت:

```
import socket

serverAddress = ('localhost', 2007)
menu = '''
Add Entry
Delete Entry
Find Entry

Quit
'''

# اتصل بالخادم
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(serverAddress)

while True:
    print( menu )
    try: choice = int(input('Choose an option[1-4] '))
    except: continue
    if choice == 1:
        name = input('Enter the name: ')
```

```
            num = input('Enter the House number: ')
            street= input('Enter the Street name: ')
            town = input('Enter the Town: ')
            phone = input('Enter the Phone number: ')
            data = "%s,%s %s, %s, %s" % (name,num,street,town,phone)
            cmd = "add:%s" % data
        elif choice == 2:
            name = input('Enter the name: ')
            cmd = 'rem:%s' % name
        elif choice == 3:
            name = input('Enter the name: ')
            cmd = 'fnd:%s' % name
        elif choice == 4:
            break
        else:
            print( "Invalid choice, must be between 1 and 4." )
            continue

        # تحدث إلى الخادم
        try:
            sock.send(cmd)
            data = sock.recv(250)
            if not data: break  # no data from server
            print( data )
        finally:
            sock.close()
```

نلاحظ هنا أن عملية معالجة القائمة هي نفسها في المثال السابق، وأن أوامر الاتصالات مجموعة في بضعة أسطر في الأسفل، وهذا أيضًا مشابه لمثال العميل الذي تقدم شرحه.

27.8 الانتقال إلى الشبكة

كانت مقابسنا إلى الآن على حاسوب محلي، ونريد أن ننقلها إلى شبكة حقيقية ويكون لدينا عمليات عميل/خادم حقيقية أيضًا، ويسهل تنفيذ ذلك بتغيير العناوين المستخدمة في استدعاء `bind()` في الخادم واستدعاء `connect()` في العميل، واستبدال عنوان IP -سواءً النسخة الرقمية أو الاسمية منه- للحاسوب الذي يعمل عليه الخادم بالمرجع المشير إلى `'localhost'`، ويمكن تشغيل عدة نسخ من العميل على كل حاسوب في نفس الوقت إذا كان لدينا عدة أجهزة على نفس الشبكة، وسيعالج الخادم الطلبات من تلك النسخ.

أما في الحياة العملية فقد نبذل مزيدًا من الجهد في التعامل مع تحليل أسماء DNS مثلًا، ونضيف اختبارات تتحقق من الأخطاء وآليات المهل الزمنية timeouts لكي لا يُعلِّق الخادم ويتأثر، لأن الشبكات الحقيقية أقل موثوقيةً وأكثر عرضةً للأخطاء، غير أننا لن نتحدث عن تلك الآليات والمهام، وإنما نذكرها فقط لوجوب حسابها عند حل مثل تلك المشكلات.

27.9 المزيد من المعلومات

كُتبت الكثير من الشروحات في برمجة المقابس، لعل أبرزها Socket How-To الذي كتبه جوردُن ماكمِلان Gordon McMillan، وهو يشرح كثيرًا من عيوب برمجة المقابس ويقدم الحلول المقترحة للتعامل معها، إضافةً إلى توثيق وحدة socket في بايثون، والذي لا غنى عن قراءته.

كما تحتوي العديد من الكتب على أقسام عن البرمجة باستخدام المقابس، ونخص بالذكر منها كتاب Python Network Programming الذي كتبه جون جورزن John Goerzen، ويدور حول برمجة الشبكات مغطيًا كثيرًا من جوانب برمجة المقابس.

والجميل في الأمر أنه يمكن تنفيذ أغلب مهام برمجة الشبكات بمستوىً أعلى إذا كنا نستخدم أحد بروتوكولات الإنترنت القياسية، مثل http و smtp و telnet وغيرها، ففي بايثون مثلًا وحدات تنفذ تلك البروتوكولات في طبقة المقابس لئلا نضطر نحن إلى ذلك، وسندرس في الفصول القادمة كيف نبسط برمجة الويب باستخدام http، من خلال وحدات المستوى الأعلى تلك، أما عند اختيار العمل في برمجة الشبكات ليكون تخصصًا مهنيًا فتُستخدم لغة rebol لأنها تولي أهميةً لخصوصية المهام، وفيها دعم للعديد من مهام الشبكة.

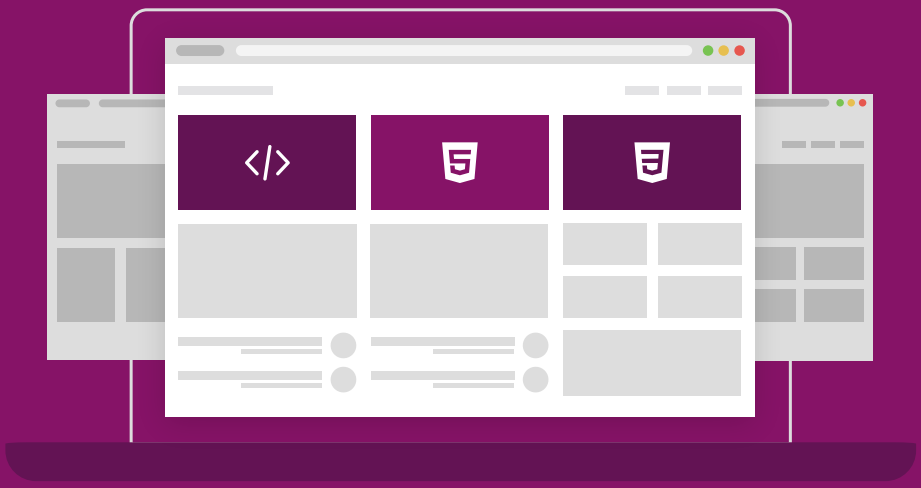
وإن أردت التعمق أكثر في الشبكات، فقد ترجمت أكاديمية حسوب كتابًا مهمًا ونشرت مقالاته تحت وسم "أساسيات الشبكات" يمكنك الرجوع إليه.

27.10 خاتمة

نرجو في نهاية هذا الفصل أن تكون تعلمت ما يلي:

- تمثَّل اتصالات شبكات الحواسيب بعناوين IP ومنافذ ports.
- تتصل المقابس بالمنافذ.
- تراقب مقابس الخوادم الاتصالات وتستمع لها عبر `listen`، وتقبلها إذا التقطتها بواسطة `accept`، وهذا يحدد منفذًا جديدًا ليستخدمه العميل الجديد.
- يجب أن تستقبل الخوادم رسائل العملاء على المنفذ الجديد من خلال `recv`، وترسل `send` الردود إلى العملاء.
- يتصل عملاء المقابس بمقبس الخادم من خلال `connect`، ويرسلون البيانات إليه من خلال `send`، ثم يستقبلون البيانات مرةً أخرى عبر `recv` في صورة ردود.
- يغلق العميل المقبس باستخدام `close` عند انتهاء عملية تبادل البيانات.

دورة تطوير واجهات المستخدم



ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية
فور انتهائك من الدورة

التحق بالدورة الآن

28. التعامل مع الويب

سنغطي في هذا الفصل:

- تاريخ موجز للويب.
- أساسيات بروتوكول HTTP.
- معماريات الويب وأطر العمل فيه.
- دور كل من HTML و CSS وجافاسكربت وبايثون.

> لا يحتاج طالب البرمجة إلى أغلب المفاهيم التي سنشرحها في هذا الفصل، ولا تؤثر الخبرة فيها على كتابة الشيفرات والبرامج والأمثلة التي سنكتبها إلا قليلا، وسنوفر روابط للتعمق في تلك التقنيات متى أمكن.

28.1 تاريخ موجز للويب

ابتكر تِم برنرز-لي Tim Berners-Lee الشبكة العالمية -الوب- لحل مشكلة واجهته هو وزملاؤه في CERN، وهي مشكلة مشاركة المستندات فيما بينهم، فقد كان لديهم مستندات كثيرة في مواقع مختلفة وبصيغ مختلفة، ويحتاج كل منها إلى مفاهيم أو مواد موجودة في مستندات أخرى بعيدة، وكان الكثير من تلك المستندات مشاريع جاريةً يعمل عليها عدة أشخاص معًا.

أدرك برنرز-لي أن هذه المشكلة قد تُحل باستخدام تقنية موجودة آنذاك اسمها النصوص الفائقة أو المتشعبة Hypertext، وهي تقنية قديمة يعود تاريخها إلى ما قبل الحواسيب الحديثة!، فعكف يدرس المشكلة وينظر في حلول النصوص الفائقة الممكنة، فوجدها غالية ومغلقة المصدر، أو لا تستطيع العمل إلا على حاسوب واحد، فكانت المساهمة التي قدمها لحل هذه المشكلة هي أخذ فكرة النصوص الفائقة وتطويرها لتعمل عبر شبكة، مما مكّن العاملين من التعاون على مشروع واحد في نفس الوقت، والوصول إلى نفس المستندات من

أماكن متفرقة، من خلال السماح للمستخدم من الوصول إلى أي حاسوب متصل بالشبكة، فصارت المكتبة الكبيرة مثل مستند عملاق تتصل فيه جميع الملفات ببعضها وتتكامل فيما بينها، مثل شبكة عنكبوتية كبيرة من المعلومات.

28.1.1 النصوص المتشعبة ولغة HTML والمحتوى الثابت

بنى برنرز-لي شكلًا من أشكال نظم النصوص المتشعبة، وقد كانت لديه في نفس الوقت خبرة بالإنترنت الموجود آنذاك والمفاهيم التي بني عليها، لذا كانت الخطوة المنطقية التالية هي دمج الفكرتين معًا، فابتكر عدة أجزاء تشكل في مجموعها الويب الذي نعرفه اليوم.

واحتاج مفهوم النص المتشعب إلى آلية لربط المستندات معًا، فابتكر لي صيغةً نصيةً أو لغةً توصيفيةً سماها لغة توصيف النصوص الفائقة Hypertext Markup Language أو HTML اختصارًا، مبنيةً على معيار موجود من قبل اسمه لغة التوصيف المعيارية العامة Standard Generalised Markup Language أو SGML اختصارًا، وتُكتب صفحات الويب الحالية في أبسط صورها بلغة HTML أو XHTML -وهي صورة أخرى تتوافق مع HTML-، يدويًا أو بواسطة برنامج حاسوبي، وتعرض المتصفحات صفحاتها من خلال تفسير وسوم HTML تلك وإخراج المحتوى المنسَّق بها على شاشة المستخدم، ويمكن الرجوع إلى توثيق موسوعة حسوب للغة HTML للمزيد عن هذه اللغة.

استطاع لي بعد كتابة هذه اللغة أن ينشئ مستندات فائقةً -أو تشعبيةً وهو مصطلح آخر لوصف مستندات HTML- وإن كانت بسيطةً نسبةً إلى الموجود الآن، أتاحت تنسيق الصفحات، وإدراج الصور والروابط إلى مستندات أخرى، لاحظ أننا لم نتعرض حتى الآن لمفهوم الشبكات، فكانت الخطوة التالية هي تمكين الوصول إلى تلك المستندات من حواسيب أخرى عبر الشبكة، وقد تطلب ذلك تعريف صيغة قياسية لعناوين تلك المستندات وذلك المحتوى، وهو ما صار يُعرف فيما بعد باسم المحدد الموحد للموارد Uniform Resource Locator أو URL اختصارًا، يشير أول جزء منه إلى البروتوكول المستخدم، والذي يكون عادةً `http` أو `https` في حالة الويب، أما الجزء التالي فيستخدم تسميةً قياسيةً في الإنترنت لتحديد الخادم -ومنفذًا اختياريًا هو عادةً المنفذ 80 كما شرحنا في الفصل السابق-، ثم يأتي الجزء الأخير وهو الموقع المنطقي logical للمحتوى على الخادم، ونستخدم كلمة المنطقي هنا لأن هذا الموقع قد لا يكون حقيقيًا أصلًا، وإنما متعلق بموقع ثابت ما يعرفه الخادم، وإن كان يبدو مثل مسار مطلق لمجلد في نظام يونكس، وقد يترجم الخادم الموقع إلى استدعاء تطبيق أو صورة أخرى من مصادر البيانات، لذا إذا نظرنا إلى الرابط الكامل للصفحة http://www.alan-g.me.uk/l2p2/tutweb.htm مثلًا، فسنجد أنه يتكون مما يلي:

البروتوكول	عنوان IP	المنفذ	الموقع
`http://`	`www.alan-g.me.uk`	`:80`	`/l2p2/tutweb.htm`

لكن المشكلة التي تأتي مع النصوص التشعبية الثابتة هي أن نمط تنسيق الصفحات هذا كان للقراءة فقط، فلا يمكن التعديل عليه أو التفاعل معه، وقد أراد لي طريقةً للتفاعل الديناميكي مع المحتوى.

28.1.2 الصفحات الديناميكية وواجهة البوابة المشتركة CGI

حُلّت مشكلة توفير المحتوى الديناميكي بإلحاق البيانات بنهاية الرابط فيما يعرف بالسلسلة المرمَّزة encoded string، أو السلسلة المهرَّبة escaped string، وهي تلك السلاسل الطويلة من الأحرف والأرقام ومحارف النسبة المئوية التي تكون في شريط عنوان المتصفح بعد الرابط، وتبدو للوهلة الأولى بلا معنى، وسندرسها بالتفصيل لأننا نحتاج إليها للاتصال بالمواقع الديناميكية.

وتعطينا إمكانية التفاعل مع المحتوى هذه عدة مزايا، لعل أهمها توفير وصول آمن للصفحات من خلال مطالبة المستخدم بإدخال اسم مستخدم وكلمة مرور، أو الطلب من المستخدمين تسجيل الاهتمام بالصفحات قبل منحهم صلاحية الوصول إليها، وتُستخدم نفس المزية في التقاط أسماء المستخدمين، كما تُستخدم لاستلام بيانات بطاقات الائتمان والاختيار من قوائم المنتجات، وقد مهدت هذه المزايا الطريق لمجال التجارة الإلكترونية والتسوق عبر الإنترنت.

أما التقنية المستخدمة في إنشاء المحتوى الديناميكي للويب فتسمى واجهة البوابة المشتركة CGI، وتمتاز بسهولة تنفيذها وحياديتها اللغوية بحيث لا تتعلق بلغة بعينها، فقد كُتبت تطبيقات الويب الأولى المبنية على CGI بلغات مثل C و C++ وملفات باتش لنظام DOS -أو batch files- وسكربتات صدفة يونكس، إلا أنها لم تلبث أن صارت تُكتب بلغة Perl وحدها بسبب القدرة الكبيرة لهذه اللغة على معالجة النصوص، ووجود وحدة CGI library لمعالجة طلبات الويب، وتحتوي بايثون أيضًا على وحدة `cgi` التي تُستخدم لبناء تطبيقات ويب ديناميكية، وإن كانت بسيطة، وسندرس ذلك في فصل خوادم الويب.

كل هذا العمل يضع إمكانيات التفاعل مع المستخدمين على عاتق الخادم، لذا صار جليًا أننا نحتاج إلى نوع من البرمجة داخل المتصفح، لأننا نريد سرعة الاستجابة في هذا التفاعل، وقد وفر متصفح Netscape ذلك في صورة جافاسكربت -و VBScript في متصفح إنترنت إكسبلورر-. وبهذه البرمجة النصية scripting languages -التي لا تحتاج إلى تصريف compiling قبل تشغيلها- صار المتصفح ينشئ نموذجًا برمجيًا للمستند داخل ذاكرته، فيما يسمى بنموذج كائن المستند Document Object Model أو DOM اختصارًا -وقد شاعت هذه التقنية إلى أن صارت معيارًا موحدًا على مستوى المتصفحات- ثم تستطيع اللغات النصية scripting languages أن تبحث في ذلك النموذج عن مكونات المستند، مثل الفقرات أو الجداول، وتعدلها، بتغيير لونها مثلًا أو موضعها في الصفحة، وقد كان ذلك مفيدًا في التحقق من صحة مدخَلات المستخدم مثلًا قبل إرسال البيانات إلى الخادم.

غير أننا نواجه مشكلةً في البرمجة القياسية لواجهة البوابة المشتركة CGI، وهي إنشاء خادم الويب لعملية جديدة لكل طلب، مما يبطئ التنفيذ ويستهلك الموارد، خاصةً على الحواسيب التي تعمل بنظام تشغيل ويندوز،

وكان من الطبيعي ظهـور آليـات أكـثر تطـورًا تسـتغل بروتوكـول CGI، مـع تمكينهـا لخـوادم الـويب من معالجـة الطلبات بكفاءة أكبر في نفس الوقت.

من النادر كتابة تطبيقات الويب باستخدام هذا البروتوكول حاليًا، وقـد فقـدت Perl موقعهـا في كونهـا لغـةً مفضلةً للويب في ظل وجود تقنيات مثل صفحات الخوادم النشطة Active Server Pages أو ASP اختصارًا، وصفحات جافا للخوادم Java Server Pages أو JSP، وكذلك PHP وغيرها، وسنلقي نظرةً على تقنية من تلـك التقنيات المبنية على بايثون في فصل إطار عمل تطبيقات الويب.

أما الآن فنسنلقي نظرةً على الهيكل العام لتطبيقات الويب، والتقنيات المستخدمة في كل مرحلـة، قبـل أن نكتب تطبيقات ويب على الخادم، وسنبدأ بإلقاء نظرة على البروتوكول الأساسي للشبكة، والـذي طـوره برنـرز-لي لتمكين عملاء النصوص الفائقة من التواصل مع خوادم المستندات.

28.2 بروتوكول النصوص التشعبية HTTP

يجب أن نفهم آلية عمل الويب قبل أن نكتب برامج له، وبالتالي لا بد أن نعـرف مـا هـو برتوكـول النصـوص الفائقة أو التشعبية، وهو بروتوكول ذو أساس نصي يتصـل من خلال مقـابس، بنفس الطريقـة الـتي رأيناهـا في فصل برمجة الشبكات من قبل، ويوفر عدة رسائل:

- GET: لقراءة المستند من الخادم.
- POST: لإرسال البيانات إلى الخادم.
- PUT: لإنشاء مستند جديد على الخادم، أو بديل له.
- DELETE: لحذف المستند من الخادم.

توجد رسائل أخرى غير التي ذكرناها، لكننا ذكرنا الرسائل الأساسية المسـتخدمة في بنـاء التطبيقـات، ولعـل أكثرها استخدامًا GET و POST، غير أننا نستطيع أن نرى أن لدينا أساسًا من نمط CRUD التقليـدي هنـا، والـذي يعني إنشاء Create وقراءة Read وتعديل Update وحذف Delete.

فإذا تتبعنا رابط http أعلاه فسنجد أننا نحتاج إلى تعلم الكثير لنكون خبراء في هذا الـبروتوكول، لكن لحسـن الحظ فإن مستوى المعرفة المطلوب لاستخدامه بسيط، لأن وحدات المكتبات تنفذ أغلب العمل المبـني عليـه، إذ يرسل عميل الويب طلب http GET، ثم يرد الخادم برسالة خطأ فيها رمز من بضعة رموز قياسـية لكـل منهـا معنىً محدد، أو بمستند HTML.

ورغم إمكانية تنفيذ كل ذلك يدويًا باستخدام المقابس و السلاسل المنسقة، إلا أن الأسـهل اسـتخدام حزمـة `urllib` الموجودة في مكتبة بايثون القياسية، وتحديدًا وحدة `urlliv.requests` التي تعالج إرسال الطلبات إلى الخادم.

28.3 استخدام الوحدة urllib.requests

لجلب صفحة ويب بسيطة نستورد `urllib.request`، ثم نفتح رابطًا لقراءة النتيجة، كما يلي:

```
>>> import urllib.request as url
>>> site = url.urlopen('http://www.alan-g.me.uk/l2p2/index.htm')
>>> page = site.read()
>>> print( page )
```

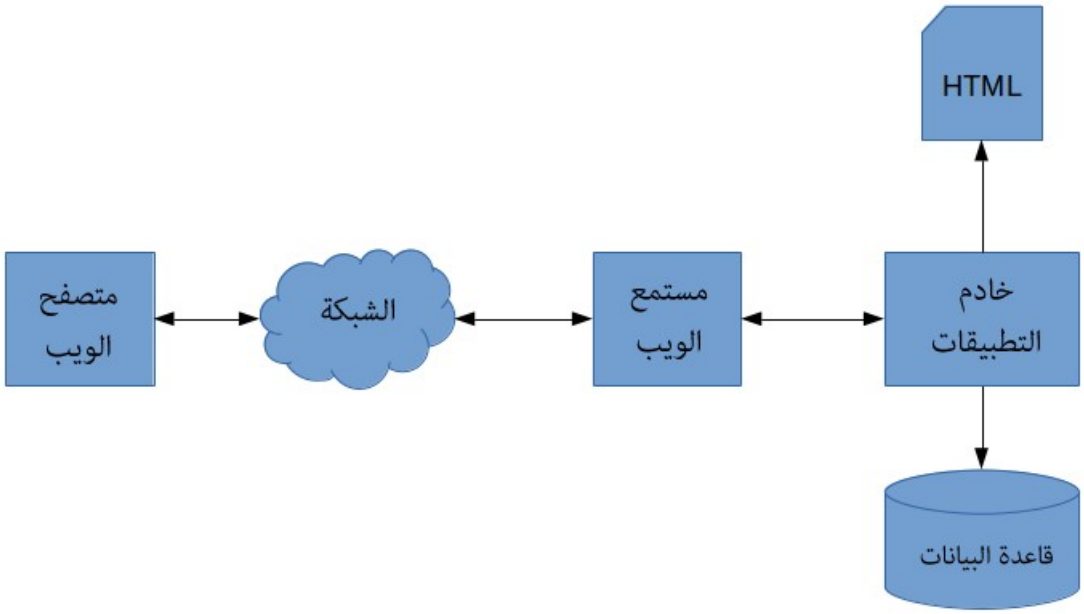
هذا هو كل ما نحتاج إليه، فلدينا الآن سلسلة نصية تحتوي على شيفرة HTML للصفحة الرئيسية للنسخة الإنجليزية من هذا الكتاب، ونستطيع موازنتها إن شئنا مع نتيجة استخدام `View>Source` في المتصفح، حيث سنرى أن لهما نفس المحتوى تقريبًا، باستثناء أن المتصفح ينسق HTML لجعلها أسهل في القراءة.

يبقى أن نسأل أنفسنا ماذا نفعل بالمحتوى الذي لدينا؟ والجواب أننا بلا شك نريد استخراج معلومات منه من خلال تحليله، ويتضمن هذا التحليل في أبسط صوره البحث عن سلسلة أو تعبير نمطي ما، إذ نستطيع أن نبحث في جميع وسوم الصور مثلًا في الصفحة من خلال البحث عن السلسلة `'<img'`، أو `'<IMG'`، أو التعبير النمطي `'<img'` مع تعيين الراية `re.IGNORECASE`.

غير أننا سنواجه عقبات أخرى في الممارسة العملية كما سنرى في الفصل التالي، لكن قبل أن نتعرف على تحليل ملفات HTML يجب أن ننظر إلى الصورة الكبيرة للكيفية التي تُبنى بها تطبيقات الويب، حيث توجد عدة طبقات تستخدم كل منها مجموعةً من التقنيات، وهذه أبرز الخصائص اللازمة لمن يكتب برامج للويب، وهي الحاجة إلى تعلم لغات كثيرة إذا أراد أن يبني تطبيقات تصلح للسوق، وتمثل بايثون هنا جزءًا واحدًا فقط من تلك اللغات.

28.4 هيكل تطبيق الويب

يبدأ تطبيق الويب التقليدي بمتصفح ويب يعمل على حاسوب عميل، يرسل طلبًا عبر الشبكة إلى خادم في مكان ما، ويشغّل الخادم جزءًا من البرنامج يسمى خادم الويب -أو مستمع الويب أحيانًا- ليلتقط الطلبات، ويرسلها -إذا كانت صالحةً- إلى خادم تطبيقات application server قد يكون -أو لا يكون- جزءًا من حزمة خادم الويب، بل قد يكون على حاسوب منفصل داخل مركز بيانات كبير، يفسر خادم التطبيقات الرسالة التي استلمها، ويعيد ملف HTML ساكنًا من نظام ملفاته، أو يعالج البيانات المستلَمة، ويولّد مستند HTML جديدًا حسب الطلب، بمساعدة البيانات المخزنة في قاعدة بيانات أحيانًا، وترسَل HTML في كلا الحالتين إلى المتصفح الذي يستقبلها، ويفسرها، ثم يعرضها على شاشة المستخدم في صورة صفحة ويب منسقة.



تختلف التقنيات المستخدمة في كل مرحلة عن الأخرى، وسننظر الآن في تلك الوحدات بالترتيب.

28.4.1 متصفح الويب Web Browser

المتصفح غالبًا هو أحد المتصفحات الشهيرة، مثل كروم أو سفاري أو فاير فوكس، وسيحتوي على البرمجية اللازمة لتفسير HTML وجافاسكربت وعرض صفحة الويب على الشاشة، كما سيرسل طلبات http إلى خادم الويب، ويستلم الردود عليها.

28.4.2 مستمع الويب Web Listener

حزمة قياسية -مثل خادم Apache المعروف-، رغم وجود عدة حزم أخرى، ولا تحتوي هذه الحزمة على أي مزايا يمكن للمستخدم برمجتها، باستثناء بعض الإعدادات.

28.4.3 خادم التطبيق Application Server

تكون هذه الحزمة غالبًا حزمة خادم قياسيةً، إما متكاملةً مع مستمع الويب، أو مستقلةً بذاتها، كما في حالة حاوية Tomcat Java Server Container، أو قد تكون إطار عمل برمجي، مثل جانغو Django للغة بايثون، أو إطار فلاسك Flask الذي سندرسه لاحقًا.

وخادم التطبيق -في كل من الحالات السابقة- هو المكان الذي سيكون فيه منطق التطبيق وشيفرته، ويمكن كتابته بأي لغة تقريبًا، لكننا سنكتبه ببايثون في شرحنا، وهو أيضًا المكان الذي ستخزَّن فيه موارد الويب الأخرى، مثل الصور وملفات الوسائط.

28.4.4 ملفات HTML

ملفـات HTML هي ملفـات نصـية قياسـية تُخـزَّن على الخـادم، وتُكتب بمـزيج من لغـات HTML وCSS وجافاسكربت، وربما VBScript إذا كانت في بيئة ويندوز، وسبب استخدام هذه اللغات الثلاثة مجموعـةً؛ هـو أن كلًا منها يؤدي دورًا مستقلًا في بناء الصفحة وجعلها قابلة للتعـديل اللاحـق، فتحـدد لغـة HTML بنيـة المسـتند، مثل العناوين والجداول والفقرات والصور وغيرها، وتوفر أوراق الأنماط المتتاليـة -أو التنسـيقات الموروثـة- CSS المظاهر الجمالية، مثل الألوان والمواضع على الصفحة والخطوط، وتتحكم في طريقة عرض عناصـر HTML على الصفحة، بينما تتحكم جافاسكربت في السلوك الديناميكي للعناصر، مثـل القـوائم المنسـدلة والعناصـر القابلـة للطي وغيرها، وقد استُخدمت في البداية للتحقق من البيانات، غير أن خيارات التحقـق الـتي أتت في HTML5 حلت محلها في هذه الوظيفة.

28.4.5 قاعدة البيانات

حزمة قاعدة البيانات في الغالب حزمة قياسية مثل التي شـرحناها في فصـل العمـل مـع قواعـد البيانـات، وتتكون من جداول SQL واستعلاماتها وبعض الإجـراءات المخزنـة في أي لغـة تـدعمها قاعـدة البيانـات، ويكـون الوصول إلى تلك البيانات غالبًا باستخدام واجهة برمجة التطبيقات API من شيفرة خادم التطبيق، كمـا رأينـا في فصل قواعد البيانات سالف الذكر.

ويتبين مما سبق أننـا نحتـاج إلى أن نتعلم HTML وCSS وجافاسـكربت وبـايثون وSQL أيضًـا إذا أردنـا بنـاء تطبيق ويب كامل، وكذلك كيفية إعداد الخوادم المختلفة وإدارتها، إلا أن مطوري الويب يتخصصون غالبًا إما في تطوير الواجهات الأمامية فقط، فيتعلمون اللغات الـتي تبـني مـا يـراه المسـتخدم النهـائي، مثـل HTML و CSS وجافاسكربت ونحوها، أو تطوير الواجهات الخلفيـة فقـط، فيتعلمـون بـايثون أو أي لغـة تطـوير أخـرى مثـل جافا وPHP وروبي Ruby وSQL.

وقـد وُضـعت آليـات مختلفـة لقولبـة مهـام التخصصـات لتوضـيح الفـروق بينهـا، حيث يسـتطيع مصـمم الواجهـات مثلًا أن يصـمم صـفحات الـويب، ويـترك علامـات markers في الشـيفرة يسـتطيع مطـور الواجهـة الأمامية، وكذلك مطور الواجهة الخلفية، استخدامها لوضع بيانـاتهم فيهـا، ويفيـد هـذا التخصـص في المهـام أن مصممي الواجهـات ليس عليهم تعلم التقنيـات البرمجيـة، وبالمثـل فليس على المطـورين أن يشـغلوا أنفسـهم بأمور التصميم المرئية.

سندرس في الفصلين التـاليين جانبًـا من برمجـة الـويب من جـانب العميـل client side وجـانب الخـادم server side، فسنرى كيف نعالج HTML كما لو كانت في متصفح، وهذا ضروري إذا أردنـا بنـاء قاعـدة بيانـات من المعلومـات الـتي نحصـل عليهـا من الـويب، فقـد نحصـل على البيانـات من مـوقعي ويب مختلفين، ونريـد جمعهمـا معًـا لتحليـل الظـواهر الشـائعة trends، وهـو مـا يُعـرف ببرمجـة جـانب العميل client side programming.

أما برمجة جانب الخادم server side programming فهي الـتي ننشـئ فيهـا موقـع ويب فيـه محتـوىً ديناميكي، وسندرس تنفيذ ذلك باستخدام CGI التقليديـة، إذ من الضـروري فهم التقنيـات الأساسـية وراء هـذه المهام عند العمل مع أطر العمل المتقدمة لاحقًا.

ثم سندرس أحد أطر العمل الحديثة، وهو إطار فلاسك Flask، وهو إطار عمل خفيـف ويسـهل تعلمـه، لكنـه قوي بما يكفي لبناء تطبيقات عملية للبيئات التجارية، ثم نختم بنقل تطبيق دليل جهـات الاتصـال الـذي نطـوره ليكون تطبيق ويب هذه المرة.

28.5 خاتمة

نأمل في نهاية هذا الفصل أن تكون تعلمت ما يلي:

- تُرسل الطلبات إلى خوادم الويب باستخدام طلبات http GET.
- يرد خادم الويب بمستند HTML.
- نحتاج إلى تحليل HTML لاستخراج البيانات التي نريدها.
- التعابير النمطية ليست أفضل أداة لتحليل HTML.
- توفر بايثون حزم `browser` و`urllib` و`html` لاستخدامها في برمجة جانب العميل.
- توفر بايثون كذلك عدة وحدات للتحليل، لعل أبرزها ElementTree.
- يُفضل التحكم في سلوك المتصفح من داخل صفحة الويب باستخدام جافاسكربت.

29. برمجة عملاء ويب باستخدام بايثون

سنغطي في هذا الفصل ما يلي:

- إرسال الطلبات إلى خادم الويب.
- إرسال البيانات إلى الخادم.
- معالجة HTML بواسطة html.parser.
- الخيارات الأخرى.

29.1 برمجة عملاء الويب

يمكن وصف برمجة عملاء الويب في تعريفين رئيسيين هما:

الأول، استخدام جافاسكربت لتعديل محتوى صفحة الويب داخل المتصفح، بتغيير ألوان العناصر وتحريك اللوحات حولها، وإظهار العناصر أو إخفائها، وجلب أجزاء من البيانات الأولية من الخادم، مثل حالة البحث الحي live search مثلًا. وهي ليست معقدةً، لكنها تتطلب معرفةً عميقةً بكل من HTML و CSS، ومن ثم فهي خارج نطاق شرحنا، فإذا أردت معرفة المزيد عن مثل هذا النوع من برمجة عملاء الويب فانظر سلسلة مدخل إلى html5 مثلًا في أكاديمية حسوب، وكتاب نحو فهم أعمق لتقنيات HTML5، كما توجد عدة مكتبات وأطر عمل لجافاسكربت يجب البحث فيها وتعلمها، لعل أشهرها jQuery، والتي يمكن القراءة عنها في توثيق jQuery في موسوعة حسوب، وغيرها من السلاسل والمواد العلمية في الأكاديمية والموسوعة.

الثاني، إنشاء برنامج يعمل على حاسوب ويتصل بخادم ويب متظاهرًا بأنه برنامج متصفح ويب، حيث يجلب هذا البرنامج بعض البيانات للتحليل، وهو ما يسمى بالبوت bot -اختصار robot-، أو يتبع بعض الروابط

من موقع لآخر بحثًا عن بيانات تتعلق بموضوع ما، وهو ما يسمى بعناكب الويب web spiders أو زاحفات الوب، وسندرس بعض هذه الأنواع من برامج عملاء الويب في هذا الفصل.

29.2 التعلم بالتطبيق

سنخالف في هذا الفصل النهج الذي كنا نتبعه في شرحنا، إذ سنستخدم مثالًا عمليًا لتوضيح كيفية كتابة برنامج عميل ويب، وسنبدأ ببساطة في البداية، ثم نضيف المزايا لإبراز بعض المشاكل المعقدة بالتدريج، لكنه سيظل برنامجًا بسيطًا للغاية عند تمامه، لكن القصد منه أن يعطيك فكرةً عما يمكن تنفيذه بمثل تلك البرامج.

مهمة هذا التدريب هي بناء برنامج يولد صفحة ويب جديدةً، تحتوي على تجميعة من جميع محتويات قائمة "سنغطي في هذا الفصل" (What will we cover?) الموجودة في أول كل فصل من هذه الفصول (الأجنبية طبعًا)، ونحتاج إلى تنفيذ بضعة أمور لبناء ذلك البرنامج:

1. قراءة محتوى HTML لملف الفهرس.
2. تحليل محتويات الصفحة، واستخراج جميع الروابط الموجودة في الجزء الأيسر إلى قائمة.
3. جلب محتوى HTML الخاص بكل رابط من روابط تلك القائمة.
4. استخراج قائمة النقاط الموجودة في قسم "What will we cover" الموجود في أول كل فصل أجنبي إلى قائمة جديدة.
5. توليد صفحة HTML باستخدام البيانات التي جمعناها.

29.2.1 جلب المحتوى

رأينا سابقًا كيفية جلب صفحة HTML بسيطة من خادم باستخدام الوحدة `urllib.request`:

```
import urllib.request as url
site = url.urlopen('http://www.alan-g.me.uk/l2p2/index.htm')
page = site.read()
```

لدينا الآن سلسلة نصية تحتوي على شيفرة HTML الخاصة بالصفحة الرئيسية، أو صفحة المستوى الأعلى للنسخة الأجنبية من الفصول، وسنلاحظ صعوبة قراءة الخرج، لهذا نستخدم خيار عرض المصدر view source الموجود في المتصفح لننظر في شيفرة HTML، لنرى أن جدول المحتويات موجود في زوج من وسوم `nav` -اختصار للتنقل navigation- وأن كل قائمة من قوائم الفصول الموجودة في كل قسم تُجمع في قائمة غير مرتبة unordered list، تحمل الوسم `ul`، ونميز كل عنصر فيها بوجود وسم `li` فيه، وتمثل جميع عناصر القائمة روابط تشعبيةً إلى ملفات الفصول، لهذا نجدها محاطةً بوسم الرابط التشعبي `<a>`.

مهمتنا التالية استخراج جميع عناصر `<a>` داخل لوحة `nav` من الصفحة.

29.2.2 استخراج محتوى الوسوم

ذكرنا في المقدمة أننا نستطيع استخدام عمليات البحث النصية البسيطة للعثور على الوسوم وغيرها في صفحات الويب، لكن يفضل استخدام محلل HTML مناسب، لذا سنستخدم الصنف `HTMLParser` الموجود في وحدة المكتبة القياسية `html.parser`، وهو محلل لغوي مجرَّد abstract، مدفوع بالأحداث أو حدَثي event-driven، ويجب أن نقسمه إلى أصناف فرعية subclasses لتوفير المزايا التي نريدها، مع ملاحظة أنه يستدعي تابعين، الأول هو `handle_starttag()` عند كل وسم HTML افتتاحي، والثاني هو `handle_endtag()` عند كل وسم إغلاق، ويجب أن نوفر النسخ الخاصة بنا من تلك التوابع لتنفذ الإجراءات المناسبة عند العثور على الوسوم التي نريدها، ونريد في حالتنا هذه أن نجد كل الروابط الموجودة في لوحة `nav`، لذا يجب تعيين راية flag نسميها `in_nav` في كل مرة نعثر فيها على وسم `nav`، فإذا وجدنا وسم `a` وكانت الراية `True` فسنحفظ خاصية `href` في قائمة، حيث تُمرَّر الخاصيات إلى التابع في صف tuple من ثنائيات المفتاح/القيمة، ونريد أخيرًا التقاط وسم الإغلاق `/nav`، وإعادة تعيين الراية إلى `False`، لضمان أننا لا نجمع أي روابط من خارج لوحة المحتويات، وعليه سنعِدّ المحلل اللغوي، ونتأكد من قدرته على تعرّف الوسوم الثلاثة المطلوبة باستخدام تعليمات الطباعة كما يلي:

```
import urllib.request as url
import html.parser

class LinkParser(html.parser.HTMLParser):
    def __init__(self):
        super().__init__()
        self.links = []  #list to store the links
        self.is_nav = False  # the flag

    def handle_starttag(self, name, attributes):
        if name == 'nav':
            print("We found a <nav> tag")
        elif name == 'a':
            print("We found an <a> tag")

    def handle_endtag(self, name):
        if name == 'nav':
            print("We found a </nav> tag")

site = url.urlopen('http://www.alan-g.me.uk/l2p2/index.htm')
```

```
page = site.read().decode('utf8') # convert bytes to str

parser = LinkParser()
parser.feed(page)
```

عند تشغيل الشيفرة السابقة ستجلب الصفحة، وتحللها، ثم تطبع الرسائل الموافقة لكل وسم تجده، وهي بهذا تثبت أن المحلل يعمل بكفاءة، وأننا بحاجة إلى راية `is_nav` لأن لدينا رابطًا خارج لوحة `nav`، نستطيع الآن أن نغير تعليمات الطباعة للشيفرة الحقيقية التي نريدها:

```
import urllib.request as url
import html.parser

class LinkParser(html.parser.HTMLParser):
    def __init__(self):
        super().__init__()
        self.links = []  # قائمة لتخزين الروابط
        self.is_nav = False  # الراية

    def handle_starttag(self, name, attributes):
        if name == 'nav':
            self.is_nav = True
        elif name == 'a' and self.is_nav:
            for key,val in attributes:
                if key == "href":
                    self.links.append(val)

    def handle_endtag(self, name):
        if name == 'nav':
            self.is_nav = False

site = url.urlopen('http://www.alan-g.me.uk/l2p2/index.htm')
page = site.read().decode('utf8') # حوِّل سلسلة البايت إلى سلسلة نصية

parser = LinkParser()
parser.feed(page)
print(parser.links)
```

نلاحظ هنا أن التغييرات التي أجريناها كانت في التابعين handle_starttag() وhandle_endtag()، إضافةً إلى السطر الأخير في الشيفرة print(parser.links)، ويجب أن تصبح النتيجة كما يلي:

```
['tutintro.htm', 'tutneeds.htm', 'tutwhat.htm', 'tutstart.htm',
'tutseq1.htm',
'tutdata.htm', 'tutseq2.htm', 'tutloops.htm', 'tutstyle.htm',
'tutinput.htm',
'tutbranch.htm', 'tutfunc.htm', 'tutfiles.htm', 'tuttext.htm',
'tuterrors.htm',
'tutname.htm', 'tutregex.htm', 'tutclass.htm', 'tutevent.htm',
'tutgui.htm',
'tutrecur.htm', 'tutfctnl.htm', 'tutcase.htm', 'tutpractice.htm',
'tutdbms.htm',
'tutos.htm', 'tutipc.htm', 'tutsocket.htm', 'tutweb.htm',
'tutwebc.htm',
'tutwebcgi.htm', 'tutflask.htm', 'tutrefs.htm']
```

29.2.3 استخراج النقاط من الفصول

بعد أن حصلنا على قائمة الفصول نريد إنشاء دالة تستطيع استخراج النقاط التي في أول كل صفحة، لذا سنستخدم خاصية View Source الموجودة في المتصفح مرةً أخرى، إذ نفحص شيفرة HTML الخاصة بإطار الفصل، فنرى أننا ننظر في مجموعة من عناصر القائمة الموجودة داخل وسم div، مع تعيين الخاصية class على القيمة "todo"، وهذا يشبه تقريبًا ما فعلناه عند بحثنا عن الروابط، وسننشئ الآن دالةً تأخذ سلسلة HTML وتعيد قائمةً من سلاسل li، وسنضيف اسم الملف في الشيفرة ونثبته ليكون tutstart.htm.

```
import urllib.request as url
import html.parser

class BulletParser(html.parser.HTMLParser):
    def __init__(self):
        super().__init__()
        self.in_todo = False
        self.is_bullet = False
        self.bullets = []

    def handle_starttag(self, name, attributes):
        if name == 'div':
```

```
            for key, val in attributes:
                if key == 'class' and val == 'todo':
                    self.in_todo = True
        elif name == 'li':
            if self.in_todo:
                self.is_bullet = True

    def handle_data(self, data):
        if self.is_bullet:
            self.bullets.append(data)
            self.is_bullet = False  # أعد تعيين الراية

    def handle_endtag(self, name):
        if name == 'div':
            self.in_todo = False

topic_url = "http://www.alan-g.me.uk/l2p2/tutstart.htm"

def get_bullets(aTopic):
    site = url.urlopen(aTopic)
    topic = site.read().decode('utf8')
    topic_parser = BulletParser()
    topic_parser.feed(topic)
    return topic_parser.bullets

print( get_bullets(topic_url) )
```

لاحظ أن لدينا تابع معالجة حدث إضافي سنعيد تعريفه، وهو `handle_data()`، لأننا نريد استخراج البيانات الموجودة في وسوم `<li>` بدلًا من الوسم نفسه أو خصائصه، وفيما عدا ذلك تشابه هذه الشيفرة المثال السابق، حيث نعين الراية `in_todo` لتوضيح متى نكون داخل صندوق المحتويات، وكذلك الراية `is_bullet` التي تشير إلى أننا وجدنا عنصر قائمة داخل الصندوق، ثم نعيد تعيين راية `is_bullet` بمجرد قراءتنا للبيانات، ونعيد تعيين `is_todo` عندما نترك الصندوق، أي عند `</div>`.

يمكن دمج البرنامجين معًا، بإضافة الصنف والدالة الجديدين إلى الملف السابق، ويتبقى أن نكتب حلقة `for` للتكرار على الروابط من المحلل الأول وإرسالها إلى دالة `get_bullets()`، ثم تُجمع النتائج في قاموس عام global dictionary مرتب وفق الفصول، كما سننظمها أكثر من خلال إنشاء دالة `get_topics()` التي

تشبه `get_bullets()`، يمكن أن نضيف شيئًا لمعالجة الأخطاء هنا، ونحوله إلى تنسيق وحدة في نفس الوقت، كما يلي:

```
import urllib
import urllib.request as url
import html.parser

###### Link handling code  ####

class LinkParser(html.parser.HTMLParser):
    def __init__(self):
        super().__init__()
        self.links = []  #list to store the links
        self.is_nav = False  # the flag

    def handle_starttag(self, name, attributes):
        if name == 'nav':
            self.is_nav = True
        elif name == 'a' and self.is_nav:
            for key,val in attributes:
                if key == "href":
                    self.links.append(val)

    def handle_endtag(self, name):
        if name == 'nav':
            self.is_nav = False

def get_topics(aSite):
    try:
        site = url.urlopen(aSite)
        page = site.read().decode('utf8') # convert bytestring to str
        link_parser = LinkParser()
        link_parser.feed(page)
        return link_parser.links
    except urllib.error.HTTPError:
        return []
```

```
##### Bullet handling code

class BulletParser(html.parser.HTMLParser):
    def __init__(self):
        super().__init__()
        self.in_todo = False
        self.is_bullet = False
        self.bullets = []

    def handle_starttag(self, name, attributes):
        if name == 'div':
           for key, val in attributes:
               if key == 'class' and val == 'todo':
                   self.in_todo = True
        elif name == 'li':
            if self.in_todo:
               self.is_bullet = True

    def handle_data(self, data):
        if self.is_bullet:
            self.bullets.append(data)
            self.is_bullet = False  # reset the flag

    def handle_endtag(self, name):
        if name == 'div':
            self.in_todo = False

def get_bullets(aTopic):
    try:
        site = url.urlopen(aTopic)
        topic = site.read().decode('utf8')
        topic_parser = BulletParser()
        topic_parser.feed(topic)
        return topic_parser.bullets
    except urllib.error.HTTPError:
```

```
        return []

#### driver code  ####
if __name__ == "__main__":
    summary = {}
    site_root = "http://www.alan-g.me.uk/l2p2/"

    the_topics = get_topics(site_root+'index.htm')

    for topic in the_topics:
        topic_url = site_root + topic
        summary[topic] = get_bullets(topic_url)

    print(summary['tutdata.htm'])
```

لا زال بإمكاننا إجراء المزيد من التنظيم والترتيب، من خلال أخذ اثنين من أصناف المحلل والدوال المرتبطة بهما إلى وحدات منفصلة، لكن نترك هذا تدريبًا لك، لذا أنشئ الوحدتين linkparser.py و bulletparser.py، واستوردهما إلى الشيفرة المتبقية التي تستطيع حفظها باسم topic_summary.py، ويجب أن يبدو الملف الأخير أشبه بما يلي:

```
import linkparser as LP
import bulletparser as BP

if __name__ == "__main__":
    summary = {}
    site_root = "http://www.alan-g.me.uk/l2p2/"

    the_topics = LP.get_topics(site_root+'index.htm')

    for topic in the_topics:
        topic_url = site_root + topic
        summary[topic] = BP.get_bullets(topic_url)

    print(summary['tutdata.htm'])
```

١. إنشاء صفحة الملخص

بقي لمشروعنا مهمة واحدة، وهي أخذ البيانات التي جمعناها وإنشاء صفحة ويب لعرضها فيها، وهي أكثر من مجرد إنشاء ملف بسيط ومعالجة نصوص، مثلما فعلنا في مثال القائمة الذي شرحناها في فصل التعامل مع الملفات، فالتعقيد الوحيد هنا هو الحاجة إلى استخدام وسوم HTML لتنسيق البيانات، والذي ننفذه بكتابة بعض ترويسات HTML الثابتة، ثم إنشاء حلقة تتكرر على بيانات الفصول لتطبع قائمةً من الفصول والنقاط bulletins، ثم نختم بوسم تذييل HTML ثابت أيضًا، ونغلق الملف، ثم نفتح الملف في المتصفح لرؤية الصفحة النهائية، كما يلي:

```
import linkparser as LP
import bulletparser as BP
import time

def create_summary(filename,data):
    with open(filename, 'w') as outf:
        # اكتب الترويسة
        outf.write('''<!Doctype htm>
<html><body>
<h1>Summary of tutor topics</h1>
<dl>''')

        # اكتب اسم كل فصل...
        for topic in data:
            outf.write('<dt>%s</dt>' % topic)
            # ...and its bullets
            for bullet in data[topic]:
                outf.write("<dd>%s</dd>" % bullet)

        # اكتب التذييل
        outf.write('''
</dl>
</body></html>''')

if __name__ == "__main__":
    summary = {}
    site_root = "http://www.alan-g.me.uk/l2p2/"
```

```
    summary_file = './topic_summary.htm'

    the_topics = LP.get_topics(site_root+'index.htm')

    for topic in the_topics:
        topic_url = site_root + topic
        summary[topic] = BP.get_bullets(topic_url)
        time.sleep(1) # DOS تعطيل رؤية الخادم له على أنه هجمة

    create_summary(summary_file, summary)
    print('OK')
```

يجب أن نحصل عند تشغيل الشيفرة على ملف اسمه topic_summary.htm نستطيع فتحه في المتصفح، ويجب أن يكون مثل هذه الصفحة.

29.2.4 إرسال البيانات في الطلب

نحتاج أحيانًا إلى تنفيذ أمور أكثر من مجرد جلب ملف ثابت من الخادم، فقد نرغب في تسجيل الدخول، ونحتاج إلى إرسال اسم المستخدم وكلمة المرور أو بيانات اعتماد أخرى، أو قد نستخدم وسيلة بحث لجلب بعض الروابط، وتوجد طريقتان لإرسال البيانات إلى الخادم وفقًا لنوع رسالة HTTP المتوقعة.

رأينا من قبل أن HTTP تحتوي على العديد من الرسائل التي يمكن إرسالها، وقلنا إن أكثر تلك الرسائل استخدامًا هي GET وPOST، حيث ترسل GET جميع بياناتها في سطر العنوان، ونستطيع رؤية ذلك في المتصفح في صورة المحارف الغريبة التي تحتوي على الكثير من علامات الاستفهام وإشارات التساوي، فهي قيم البيانات التي في طلب GET.

ا. إرسال سلسلة بحث إلى GitHub

تنقسم برمجة عملاء الويب web clients عمليًا إلى جزء علمي وفق قواعد معروفة مسبقًا ونتوقع نتائجها، وجزء يأتي بالتجربة والخطأ، ويمكن تعلم الكثير عن موقع ويب بزيارته من خلال متصفح ويب، وإلقاء نظرة متفحصة في كل من شريط العنوان ومصدر الصفحات، فإذا زرنا مستودع الشيفرات مفتوحة المصدر في GitHub مثلًا، وبحثنا عن كلمة "python" فسنرى أن عنوان الصفحة المعادة يكون أشبه بما يلي:

```
https://github.com/search?utf8=%E2%9C%93&q=python&type=
```

يشير الجزء `utf8=%E2%9C%93` إلى محرف يونيكود -هو '✓'- الذي يخبرنا أن البحث يستخدم UTF-8، ونستطيع تجاهل هذا التفصيل وكذلك `type=` الفارغة في النهاية، ونرسل ما يلي:

```
http://github.com/search?q=python
```

وسنرى أنها تعطينا نفس النتيجة.

نستطيع اكتشاف نوع هيكل HTML المُعاد من خلال عرض مصدر الصفحة، وخصائص العنصر element properties باستخدام أدوات الفحص الخاصة بالمتصفح inspection tools، ونستطيع إجراء البحث النصي لشاشة مصدر الصفحة في حالة GitHub ونحصل منه على أول نتيجة -والتي كانت geekcomputers/Python عند بحثنا-، والتي بدت كما يلي:

```
<div class="repo-list-item d-flex flex-justify-start py-4 public
source">
  <div class="col-8 pr-3">
    <h3>
      <a href="/geekcomputers/Python" class="v-align-
middle">geekcomputers/<em>Python</em></a>
    </h3>
...
```

نرى هنا كيفية استخراج الروابط، وصعوبته أكثر مما في مثال topic_summary.htm السابق.

لا شك أن المشاكل لا حصر لها، وسنجد للمواقع آليات تمنع كل شيء -عدا المتصفحات- من الوصول إلى بياناتها، أو تستخدم تقنيات جافاسكربت متطورةً لعرض الصفحة، ولن تفلح تقنيات تحليل HTML البسيطة معها، لكن بنظرةً متفحصةً في تلك المواقع سنرى أنها توفر واجهة برمجة تطبيقات API يمكن استخدامها بديلًا، وهي أفضل من أدوات تحليل HTML، وقد يطلب الموقع مالًا لقاء رخصة استخدامها إن كان تجاريًا لأنها الطريقة التي يمول بها الموقع نفسه.

وتوجد عدة مشاكل أخرى لا يتسع لها شرحنا، لكن نشير إلى أهمها والتي ينبغي البحث فيها، مثل معالجة محاولات تسجيل الدخول، واستخدام ملفات تعريف الارتباط cookies، ومعالجة اتصالات https المشفرة، فيمكن تنفيذ كل ذلك بقليل من الجهد، غير أنها خارج سياق شرحنا كما ذكرنا، ويمكن استخراج أغلب المعلومات من صفحة HTML باستخدام المحلل بدمج تباديل مختلفة من تلك التقنيات، إلا أننا قد نحصل على شيفرة HTML غير متقنة الكتابة أو التنسيق، وعندها سنحتاج إلى كتابة شيفرة خاصة لتصحيح الشيفرة، بل قد نضطر إلى كتابتها إلى ملف نصي واستخدام متحقق HTML خارجي -مثل HTMLtidy- لتصحيحها، إذا كان التشوه الحاصل فيها أكثر مما يمكن تعديله يدويًا، وذلك قبل أن نحللها، أو يمكن استخدام حزمة من طرف ثالث -مثل Beautiful Soup- التي تستطيع التعامل مع أغلب المشاكل الشائعة في HTML.

29.2.5 اكتشاف رموز الخطأ

يعيد الخادم أحيانًا أخطاءً بدلًا من صفحات الويب التي نتوقعها، ويجب أن نتمكن من التقاط تلك الأخطاء وقراءتها، والتي يعرضها المتصفح في صورة خطأ "Page Not Found" المشهور، أو صورة عبارات ألطف قليلًا، لكن إذا كنا نحن الذين نجلب البيانات بأنفسنا من الخادم فسنجد أن الخطأ يأتي في صورة رمز خطأ في ترويسة http، التي يحولها `urllib.request` إلى استثناء `urllib.error.HTTPError`، وبما أننا نعرف كيفية التقاط الاستثناءات باستخدام بنية `try/except` العادية، فنستطيع التقاط تلك الأخطاء بسهولة كما يلي:

```
import urllib.request as url
import urllib.error
try:
   site = url.urlopen("http://some.nonexistent.address/foo.html")
except urllib.error.HTTPError, e:
   print e.code
```

تأتي القيمة الموجودة في `urllib.error.HTTPError.code` من أول سطر من رد HTTP الخاص بخادم الويب، مباشرةً قبل بداية الترويسات، مثل "HTTP/1.1 200 OK" أو "HTTP/1.1 404 Not Found"، ويتكون من رقم الخطأ، ويمكن الاطلاع على أشهر رموز الخطأ التي يعيدها خادم الويب في هذه الصفحة من موقع w3، وما يهمنا هي الأخطاء التي تبدأ بالرقم 4 أو 5، وأغلب تلك الأخطاء ستكون:

- 401: غير مصرح له Unauthorized.
- 404: الصفحة غير موجودة Page not found.
- 407: يُطلب توثيق الوكيل Proxy Authentication Required.
- 500: خطأ داخلي في الخادم Internal Server Error.
- 503: الخدمة غير متاحة Service Unavailable.
- 504: انتهت المهلة الزمنية للبوابة Gateway Timeout.

قد يكون المطلوب في بعض تلك الأخطاء مثل -503 و504- مجرد إعادة المحاولة بعد بضعة دقائق، أما في الخطأ 407 فقد نحتاج إلى مزيد من العمل للوصول إلى صفحة الويب، ولمزيد من التفاصيل حول هذه الرموز انظر مقال رموز الإجابة في HTTP.

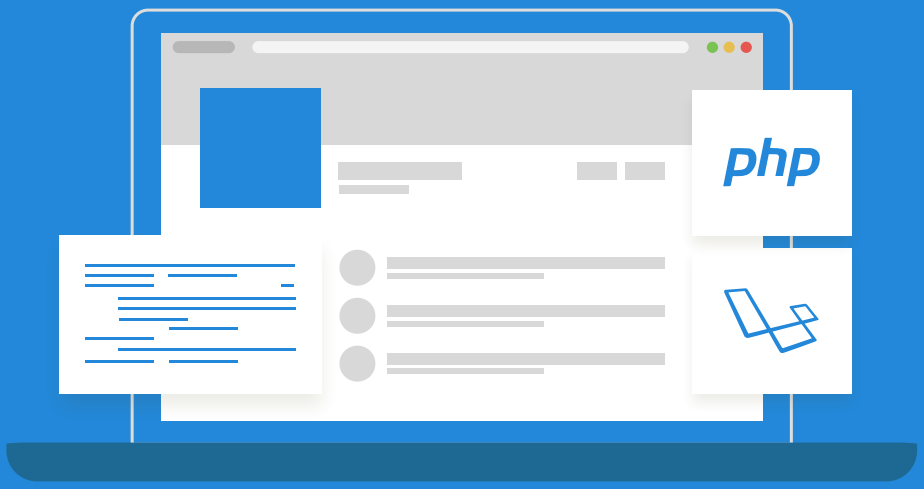
لننظر الآن في الجانب الآخر، فما الذي يحدث عند وصول طلباتنا إلى خادم الويب؟ وكيف ننشئ خادم ويب خاص بنا؟ هذا هو موضوع الفصل التالي.

29.3 خاتمة

في نهاية هذا الفصل نأمل أن تكون تعلمت ما يلي:

- تُنشأ طلبات إلى خوادم الويب باستخدام طلبات http GET.
- يرد خادم الويب بمستند HTML.
- نحتاج إلى تحليل HTML لاستخراج البيانات المطلوبة.
- توفر بايثون عدة وحدات للتحليل، وأبسطها وحدة `html.parser`.
- توفر وحدات من طرف ثالث، مثل BeautifulSoup، وسائل تحليل أسهل وأفضل.

دورة تطوير تطبيقات الويب
باستخدام لغة PHP



احترف تطوير النظم الخلفية وتطبيقات الويب
من الألف إلى الياء دون الحاجة لخبرة برمجية مسبقة

التحق بالدورة الآن

30. كتابة تطبيقات الويب

سنغطي في هذا الفصل:

- تاريخ موجز لخوادم الويب.
- أساسيات CGI: واجهة البوابة المشتركة.

30.1 مقدمة في برمجة خوادم الويب

في بدايات الإنترنت كنا نطلب صفحات HTML من خوادم الويب فتعطينا صفحات ثابتةً أو ساكنةً، ليس للمستخدمين أي يد في تعديل محتواها أو التفاعل معها، باستثناء بعض عمليات باتش batch processes التي استخدمتها بعض المواقع لإنشاء صفحات HTML من مصادر البيانات، لضمان تحديث معلومات تلك الصفحات، وفي هذه الحالة أيضًا لم يكن للمستخدمين دور في ما يظهر أمامهم على صفحة الويب.

ثم صرنا ندمج البرامج في خوادم الويب لتوليد الصفحات ديناميكيًا كلما طلبها المستخدم، وسميت الآلية المستخدمة في ذلك واجهة البوابة المشتركة Common Gateway Interface أو CGI اختصارًا، ولا يزال أثر تلك التقنية باقيًا في مواقع الويب التي نتصفحها حاليًا.

ومع استغلال المبرمجين إمكانيات واجهة CGI -مع وسم `<form>` في HTML- صارت المواقع التي يبنونها أكثر تطورًا وتعقيدًا، وصار من الممكن بناء مواقع التسوق الإلكتروني، والألعاب، والدعم الفني، وغيرها من الخدمات التي صارت اليوم بدهيات.

تُبنى واجهة CGI على مفهوم بسيط، هو أن بيانات الإدخال تُرسَل مثل جزء من رسالة http GET أو http POST، ويشير الرابط إلى مجلد خاص في خادم الويب، الذي يعرف أن عليه تنفيذ المورد resource بدلًا

من إعادته، وهذا المورد هو برنامج يرسل الخرج الخاص به إلى مجرى الخرج القياسي stdout في صفحة HTML، فيقرأ الخادم مجرى الخرج القياسي، ويوجهه إلى العميل الذي طلب الصفحة.

سندرس الآن جزءًا من تلك المرحلة المبكرة في تطور خوادم الويب، بأن ننشئ خادم ويب بسيط على حاسوبنا المحلي، ليعطينا صفحة HTML ثابتةً، ثم نوسّع الصفحة لتشمل استمارةً تلتقط اسم المستخدم وترسله إلى الخادم، ونختم بإنشاء برنامج CGI يقرأ اسم المستخدم ذاك، ويعيد رسالة ترحيب مخصصةً باسمه من الخادم.

30.2 إنشاء صفحة ترحيب باستخدام واجهة CGI

توفر بايثون وحدة خادم ويب بسيطةً نستطيع استخدامها للاختبار والتطوير، قبل نقل الشيفرة إلى منصة استضافة ويب حقيقية في شبكة ما أو على الإنترنت، وسنبدأ بالنظر في كيفية تشغيلها لإرسال صفحات ويب ساكنة static web pages.

30.2.1 صفحة ويب بسيطة

أول ما علينا فعله هو إنشاء صفحة الويب التي نريد إرسالها، ولن نشرح كيفية كتابة شيفرة HTML بالتفصيل هنا، وإنما سنشرح ما يكفي لتشغيل الأمثلة التي نكتبها، وستبدو صفحة "Hello World" بسيطةً كما يلي:

```
<!doctype html>
<html>
   <head>
     <title>Hello world web page</title>
   </head>
   <body>
       <h1>Hello World</h1>
   </body>
</html>
```

تتكون HTML من وسوم، وهي أسماء محددة مسبقًا في توصيف اللغة نفسها، تحيط بها علامتا < و>، وتأتي أغلب الوسوم في أزواج يكون وسم الإغلاق فيها مسبوقًا بشرطة مائلة /، كما قد تحتوي الوسوم على بيانات تُعرف بالخاصيات attributes، غير أن المثال أعلاه ليس فيه أي منها لأننا بدأنا به بسيطًا، لكننا سنراها لاحقًا.

تجدر الإشارة هنا إلى أن بعض العناصر إجبارية في HTML، فيجب أن تحتوي كل صفحة HTML عليهـا على الأقل لتكون صالحةً، رغم أن أغلب المتصفحات الحالية ستعرض الصفحة حتى لو لم تكن هذه العناصر موجودةً، وتلك العناصر هي:

1. التصريح `doctype`: يشير إلى أن هذا المستند هو مستند HTML، وقد كان هذا التصريح طويلًا ومفصلًا سابقًا، لكن -ومنذ صارت HTML5 قياسيةً ومعتمدةً- نكتفي بكتابة `html` في ذلك التصـريح. لاحـظ أن تنسيق الوسم في doctype مختلـف عن وسـوم HTML العاديـة، إذ توجـد علامـة تعجب مباشـرةً بعـد علامة <.
2. الوسم `html`: نستخدم زوجًا من الوسم `<html>` لتحديد بداية متن المسـتند نفسـه ثم وسـم إغلاق لـه، ويشكل هذا الوسم جذر المستند، وتمثَّل صفحات HTML في ذاكرة المتصفح في هيكل شجري، تكـون قمته الوسم `<html>`. وتعدِّل شيفرة جافاسكربت التي في المتصفح ذلك الهيكل، لذا تؤثر طريقة كتابة HTML كثيرًا في كيفية كتابة شيفرة جافاسكربت، وكذلك في قـراءة عملاء الـويب web clients -مثـل تلك التي كتبناها في الفصل السابق- لتلك الصفحة.
3. الوسـم `head`: يحـدد الوسـم `<head>` جـزءًا من الصـفحة يحـوي معلومـات عن الصـفحة نفسـها، لكن المتصفح لا يعرضها، فمثلًا تعرَّف فيه بيانات أوراق الأنماط المتتالية أو التنسيقات الموروثة CSS، وقـد يُستورد ملف CSS عند وجود عدة أنماط في الصفحة، كما تستورد ملفات جافاسكربت المسـتخدَمة في الصفحة. نلاحظ أننا أزحنا الوسوم في المثال أعلاه، لكن ذلك لا يؤثر على تنفيذ الشـيفرة، وهـو لتحسـين قراءة الشيفرة وإظهار الهيكل الشجري المتشـعب للمسـتند لا أكـثر، ولا فـرق بالنسـبة للمتصـفح سـواء أزيحت الشيفرة أم لا، وقسم `head` غير إلزامي عند كتابة مستندات HTML نظريًا، لكن وجوده يرجـع إلى حاجة أغلب المتصفحات الحديثة للمعلومات التي فيه.
4. الوسم `title`: يحتوي هذا الوسم على عنوان الصـفحة الـذي سيعرضـه المتصـفح، وهـو الجـزء الوحيـد المعروض من قسم الترويسـة، رغم تـأثير الأنمـاط وجافاسـكربت في المظهـر العـام لمحتـوى الصـفحة الرئيسية.
5. الوسم `body`: يغلف هذا الوسم المحتوى الأساسي للصفحة، والذي نراه في نافذة المتصفح.

هـذه هي العناصـر الأساسـية لصـفحة الـويب، وقـد اسـتخدمنا الوسـم `<h1>` الـذي يحتـوي رسـالة "Hello World"، وهو جزء من مجموعة عناصر `h` -التي تشـير إلى كلمـة ترويسـة header- وتُـرقَّم من 1 إلى 6 لتوضح ترتيب عناوين الفقرات، ويصغر حجم الخط المستخدم ومظهره كلمـا اقتربنـا من العنـوان السـادس، رغم إمكانية تعديل ذلك باستخدام CSS.

نحفظ شيفرة HTML في مستند باسم index.htm في مجلد باسم hello، ونتحقق من عملـه بتحميلـه في متصفح ويب، فإذا عرض المتصفح الصفحة كما نريدها فنكون قد نجحنا في هذه المهمة البسيطة، وننتقـل الآن إلى توفيره من خلال خادم الويب الخاص بنا.

30.2.2 تشغيل خادم الويب

يوجد خادم الويب الخاص بايثون في وحـدة `http.server`، ويمكن تشـغيله دون أي تعـديلات من خلال سطر أوامر نظام التشغيل، بتغيير المجلد العامل إلى مجلد hello الذي أنشأناه قبل قليل، كما يلي:

```
$ python -m http.server --cgi 8000
```

يحدد الخيار `-m` في الشيفرة أعلاه الوحدة التي يجب أن تشغلها بايثون، أمـا الرايـة `--cgi` فتفعّـل عمليـات واجهة CGI التي سنحتاج إليها لاحقًا، ويشير العدد 8000 إلى منفذ الشبكة الذي ستستخدمه.

30.2.3 تحميل صفحة الويب

نستطيع الآن الوصول إلى الخادم من خلال تحميل الرابط التالي في شريط عنوان المتصفح:

```
http://localhost:8000
```

إذا كان الخادم يعمل فستحمَّل تلك الصفحة وتُعرض كما فعلنا من قبل، إلا أنها الآن تُرسَل من قبل الخادم، كما يتضح من شريط العنوان في المتصفح، وبهذا نكون قد أرسلنا أول صفحة ويب لنا!

والسبب الذي يجعل هذه العملية ناجحةً دون تحديد اسم الملف هو أن خوادم الويب تبحث عن ملف باسـم `index.htm` تلقائيًا، فإذا كان موجودًا ولم نحدّد ملفًا مسبقًا فسيُعرض `index.htm`. كمـا يمكن إعـداد الخـادم ليبحث عن ملفات افتراضية أخرى، مثل `index.html` أو `index.php`.

30.3 استخدام واجهة CGI لعرض رسالة ترحيب بالمستخدم

نريد الآن أن نوسع صفحة الويب السابقة لتلتقـط اسـم المسـتخدم وترسـله إلى الخـادم، الـذي يـرد بإرسـال رسالة ترحيب مخصصة باسم المستخدم.

30.3.1 إنشاء استمارة HTML

الخطوة الأولى لذلك هي تعديل ملف HTML ليتضمن حقل إدخال لاسم المستخدم، كما يلي:

```
<!doctype html>
<html>
  <head>
    <title>Hello user page</title>
```

```
    </head>
    <body>
      <h1>Hello, welcome to our website!</h1>
      <form name="hello"
            method="get"
            action="http://localhost:8000/cgi-bin/sayhello.py">
        <p>Please tell us your name:</p>
        <input type="text" id="username" name="username"
               size="30" required autofocus/>
        <input type="submit" value="Submit" />
      </form>
    </body>
  </html>
```

نلاحظ هنا أن عناصر `<form>` تحتوي على خاصيات في وسومها، حيث تشكل الخاصية `name` جزءًا من البيانات المرسَلة إلى الخادم مع القيمة التي في صندوق الإدخال النصي، وتخبر الخاصية `method` الخاصة بوسم `form` المتصفح نوع رسالة http التي يجب إرسالها -وهي GET في هذه الحالة-، كما تخبره الخاصية `action` بالمكان الذي يجب أن يرسلها إليه،.

تخبر خاصيتا العنصر `input` الأخيرتان المتصفح ألا يرسل الاستمارة إلا إذا احتوى الحقل النصي على قيمة، وأن يضع المؤشر في ذلك الحقل ليكون مستعدًا لاستقبال الإدخال، كما تخبر القيمة `submit` الموجودة في نوع الإدخال `type` المتصفح أن يعرض هذا العنصر مثل زر تظهر عليه الكلمة الموجودة في قيمة الخاصية `value` -والتي هي `submit` أيضًا-، وسيرسل ذلك الزر الاستمارة عند الضغط عليه إلى الخاصية `action` الخاصة بالاستمارة.

نحفظ هذا في نفس المجلد كما فعلنا من قبل، لكن نستدعي hello.htm في هذه المرة، ونتحقق من صحة عملها بكتابة ما يلي في شريط العنوان:

```
http://localhost:8000/hello.htm
```

ينبغي أن نرى رسالةً فيها حقل إدخال نصي يطلب منا إدخال الاسم مع زر تحته، إلا أنه لن يحدث شيء عند ضغطنا على الزر لأننا لم نكتب أي شيفرة تعالج تلك النقرة على الخادم، لذا نحتاج إلى إنشاء برنامج CGI يُستدعى عند إرسال الاستمارة إلى الخادم بالضغط على الزر.

30.3.2 كتابة شيفرة CGI

يشبه برنامج CGI برامج بايثون التي كتبناها من قبل، باستثناء أمرين هما:

- برنامج CGI يستورد وحدة CGI.
- يوجد في مجلد باسم cgi-bin، في جذر خادم الويب، ويكون هذا البرنامج تنفيذيًا executable، لنحقق الأمر الثاني ننشئ مجلد `cgi-bin` داخل مجلد `html`، ثم ننشئ ملفًا داخل ذلك المجلد الجديد ونسميه sayhello.py يحتوي على الشيفرة التالية:

```
#!/usr/bin/env python3
import cgi

# استخرج حقول البيانات
data = cgi.FieldStorage()
username = data["username"].value

# أرسل ترويسة http الإجبارية مع اللاحقة \n\n:
print("ContentType: text/html\n\n")

# والآن، أرسل محتوى HTML
print("<!doctype html>\n<html><head>")
print("<title>Hello %s</title></head>" % username)
print('''
<body>
   <h1>Welcome %s</h1>
</body>
</html>''' % username)
```

تنفَّذ العمليات الخاصة بالبيانات داخل استدعاء `FieldStorage`، حيث تجمع وحدة `cgi` جميع البيانات من طلب http، وتضعها في قاموس لنستطيع الوصول إليها بسهولة باستخدام مفاتيح نصية هي خاصيات `name` من الاستمارة الخاصة بنا.

نلاحظ كيف تمكننا علامات الاقتباس الثلاثية الخاصة ببايثون من هيكلة الخرج في HTML بصورة يمكن قراءتها -انظر قسم head و body.

بعد أن نحفظ الصفحة، نتأكد من تغيير الصلاحيات ليكون الملف قابلًا للتنفيذ من جميع المستخدمين، فإذا أعدنا تحميل صفحة hello.htm، وملأنا الاستمارة فسنحصل على رسالة ترحيب من الخادم عند الضغط على الزر.

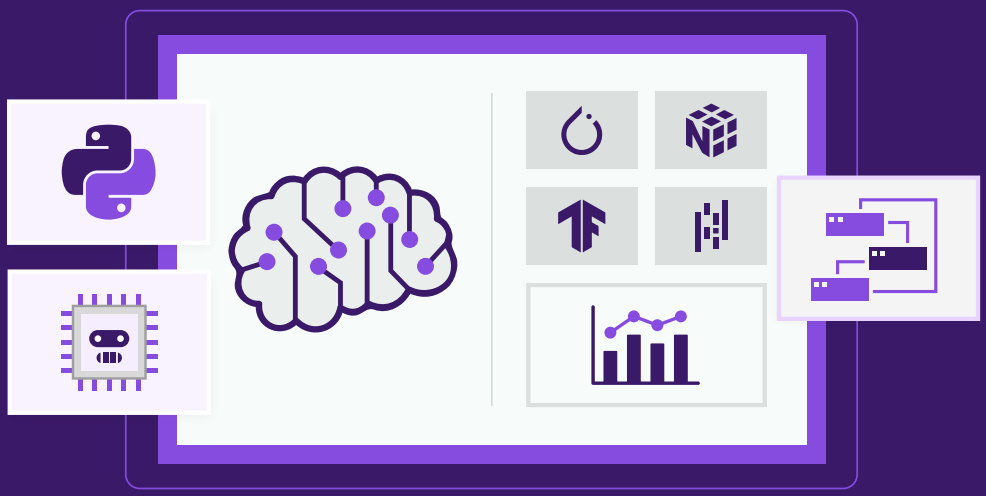
تصلح هذه التقنية لمثل هذه التطبيقات البسيطة، وتمتاز بشفافية الخطوات، ومن ثم سهولة تصحيحها، غير أنه مع زيادة حجم المواقع وتعقيد البيانات لا تكون واجهة CGI مناسبةً للعمل معها، وهنا يأتي دور أطر عمل الويب frameworks، والتي سننظر فيها في الفصل التالي.

30.4 خاتمة

نأمل في نهاية هذا الفصل أن تكون تعلمت ما يلي:

- تُنشأ الطلبات إلى خوادم الويب باستخدام طلبات http GET.
- يرد خادم الويب بمستند HTML.
- تُمرَّر البيانات في رابط الطلب أو متغيرات البيئة.
- تعالج وحدة CGI البروتوكول الأساسي.
- تُجلب بيانات الاستمارة من خلال قاموس `FieldStorage`.
- واجهة CGI مناسبة لاستمارات الويب البسيطة، مثل شاشات تسجيل الدخول، إلا أنها لا تناسب التطبيقات الكبيرة.

دورة الذكاء الاصطناعي



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

31. استخدام أطر العمل في برمجة تطبيقات الويب: فلاسك نموذجًا

سنغطي في هذا الفصل من ما يلي:

- أساسيات أطر عمل الويب.
- استخدام إطار Flask في مثال Hello World.
- بناء برنامج دليل جهات الاتصال في تطبيق ويب.

31.1 أطر عمل الويب Web Frameworks

تُعد البرامج التي تعتمد على الواجهة CGI التي تحدثنا عنها في الفصل السابق وسيلةً فعالةً لبناء مواقع ويب تفاعلية، إلا أن لها عدة مشاكل لا يمكن تجاهلها، لعل أخطرها بدء الخادم عمليةً جديدةً في كل طلب، مما يسبب بطءً واستهلاكًا للموارد، لذا لا يمكن استخدامها في التطبيقات ذات العمليات الكثيرة المتزامنة، والمشكلة الثانية وجود الشيفرة وتعليمات HTML والبيانات في ملف واحد، مما يصعّب عملية الصيانة ويعرّضها للأخطاء، فقد يتسبب تعديل بسيط في HTML بأخطاء تركيبية في شيفرة بايثون، وقد جُربت عدة أساليب مختلفة للتعامل مع هذه المشاكل، إلا أن الأسلوب الذي استمر والذي سننظر فيه هو إطار عمل الويب web framework، وأشهر هذه الأطر:

- صفحات خوادم مايكروسوفت النشطة Active Server Pages: واختصارًا ASP، يعمل هذا الإطار جنبًا إلى جنب مع خادم ويب مايكروسوفت IIS وبيئة ".NET" الخاصة بها، وهذا يعني إمكانية استخدام أي لغة مبنية على ".NET" -بما في ذلك نسخة بايثون الخاصة بها- في إطار ASP.

- صفحات خوادم جافا Java Server Pages: واختصارًا JSP، هي إطار عمل جافا الخاص بتطوير الويب، و هي تقنية لتوليد تطبيقات الخوادم المصغرة Servlets، التي هي عمليات خفيفة تشبه برامج CGI، لكنها أسهل في التعامل معها.
- إطار عمل Ruby on Rails: يشبه إطار العمل ريلز rails الخاص بلغة روبي إطار عمل Flask الخاص بايثون، والذي سندرسه قريبًا.
- إطار عمل جانغو Django: يُعد جانغو أكثر أطر عمل بايثون شهرةً، خاصةً في بناء المواقع الكبيرة، وله شروحات كثيرة في كتب وتوثيقات إلكترونية، وهو غني بوحدات الإضافات plug-in modules المتاحة لتوسيعه، إلا أنه صعب التعلم موازنةً بإطار Flask.
- إطار العمل Flask: وهو ما سندرسه في هذا الفصل، إذ يوفر جميع العناصر الأساسية الموجودة في أطر العمل، وهو سهل التعلم نسبيًا، ويمكن استخدامه ببساطة دون كثير من التعقيد الذي يشتت الانتباه عن الأفكار الرئيسية.

تتمثل الفكرة الأساسية في جميع أطر عمل الويب في أنها تقلل الحمل على الخادم، باستخدام تقنيات البرمجة المتزامنة، والتي سندرسها في الفصل التالي، كما تفصل المنطق logic عن شيفرة العرض presentation code، لتسهيل صيانة الموقع، حيث تسعى أحدث المعايير الصادرة عن w3c في كل من HTML5 و CSS3 إلى تحقيق هذا الفصل، إذ يجب أن تُستخدم HTML حصرًا في هيكلة بناء المستند، بينما تُستخدم التنسيقات الموروثة CSS للمظهر، من خطوط وألوان ومواضع وتحكم في رؤية العناصر أو إخفائها وغير ذلك، وبهذا تصبح برمجة الويب عملًا منظمًا يمكن صيانته، من خلال الجمع بين تلك الممارسات في كل من CSS وHTML مع أطر عمل الويب، وذلك من خلال تقنيتين رئيسيتين:

1. توجيه نقطة النهاية routing endpoint للروابط، والتي يشار إليها أحيانًا بالربط mapping، حيث يوجَّه الجزء الأخير من الرابط من ملف إلى دالة أو تابع في إطار العمل.
2. توفر قوالب المستندات طريقةً لإنشاء ملفات HTML ساكنة، فيها محددات للمواضع place-markers لإدراج البيانات فيها، ويمكن لدوال إطار عمل الويب أن تجهز البيانات ثم تمررها إلى محرك القالب، الذي يجلب القالب المناسب، ويدرج البيانات في محددات مواضعها لتوليد خرج HTML النهائي الذي يُعاد إلى متصفح المستخدم.

سنرى استخدام كلا التقنيتين في Flask، ولأطر عمل الويب غالبًا نفس المفاهيم، رغم اختلاف تفاصيل بنائها اللغوي syntax، واصطلاحات الاستدعاء المتبعة.

31.2 تثبيت Flask

لا يوجد إطار عمل Flask في مكتبة بايثون القياسية، لذا يجب تثبيته إما باستخدام أداة `pip` التي تـأتي مـع بايثون، أو استخدام حزمة تنفيذية خاصة، وهو ما يُنصح به مسـتخدمو لينكس، خاصـةً عنـدما يُثبت على النظـام أكثر من إصدار بايثون.

عند استخدام `pip` نفتح طرفية النظام (وقد نضطر إلى تشغيل أمر `pip` بصلاحيات مدير النظام، خاصةً عنـد تثبيت بايثون لجميع المستخدمين على الحاسوب) ونكتب فيها ما يلي:

```
$ pip install flask
```

يمكن التحقق منها بتشغيل بايثون، واستيراد Flask بالأمر `import flask`، فـإذا لم نحصـل على أخطـاء نكون قد نجحنا.

31.3 استخدام Flask في مثال Hello World

يوفر إطار عمل Flask وحدة خادم ويب بسيطةً، يمكن استخدامها للاختبار والتطوير قبـل نقـل الشـيفرة إلى منصة استضافة ويب في شبكة تابعة لشـركة أو على الإنـترنت، وسـندرس الآن كيفيـة تشـغيله وتوفـير صـفحة ويب بسيطة.

31.3.1 صفحة ويب بسيطة

لن نحتاج هنا إلى إنشاء ملف HTML كما فعلنا في حالة CGI في الفصل السابق، بل سنعيد شيفرة HTML من دالة Flask، كما يلي:

```
import flask

helloapp = flask.Flask(__name__)

@helloapp.route("/")
def index():
   return """
<html><head><title>Hello</title></head>
<body>
<h1>Hello from Flask!</h1>
</body></html>"""

if __name__ == "__main__":
```

```
    helloapp.run()
```

يجب أن نشير هنا إلى بضعة أمور، حيث يُسند أول سطر بعد `import flask` نسخةً من كائن تطبيق `Flask` إلى متغير `helloapp`، وهو اسم عشوائي رغم أنه نفس اسم تطبيقنا، ثم نمرر المتغير الخاص `__name__`، الذي رأيناه في فصول سابقة، إلى التطبيق الذي يستخدمه ليعرّف الموقع الرئيسي (الجذر) لموقع الويب.

أما الأمر التالي فهو الدالة التي تعالج طلب http GET، والتي تسمى `index` اصطلاحًا لتوافق تسمية ملف `index.htm`، غير أن المثير للاهتمام هنا هو المزخرِف `@helloapp.route("/")` الذي يسبقها، والذي يخبر إطار Flask أن أي طلب إلى جذر الموقع `/` يجب توجيهه إلى التابع التالي المسمى `index` في حالتنا هذه، ويمكن توجيه عدة نقاط نهاية end points إلى نفس التابع بتكديس المزخرفات decorators فوق بعضها البعض، وقد ذكرنا من قبل أن توجيه نقاط النهاية هو إحدى التقنيات المستخدمة في أطر عمل الويب، وهذه هي طريقة إطار عمل Flask لتوجيهها.

31.3.2 تشغيل خادم ويب Flask

نبدأ تشغيل شيفرة Flask التي صارت جاهزةً الآن بالطريقة المعتادة، باستدعاء ما يلي من المجلد الذي يحوي التطبيق:

```
$ python hello.py
```

ينبغي أن نرى رسالةً تخبرنا أن الخادم يعمل وينتظر الطلبات:

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

نستطيع الآن العودة إلى المتصفح لندخل العنوان الموجود في الرسالة:

```
http://127.0.0.1:5000/
```

ستظهر رسالة "Welcome from Flask"، وبهذا نكون قد بنينا أول تطبيق ويب بإطار عمل Flask.

31.3.3 مقدمة في القوالب

تتعامل شيفرة إطار Flask التي رأيناها حتى الآن مع مشاكل توسع التطبيقات، وزيادة حجمها، وبعض الفوضى التي تسببها برمجة CGI، إلا أنها لا تستطيع فصل الشيفرة عن HTML، فلا زلنا نكتب سلاسل HTML داخل شيفرة البرنامج، ولمعالجة تلك المشكلة نستخدم قوالب Flask.

والقـوالب في إطـار Flask مـا هي إلا ملفـات HTML سـاكنة فيهـا محـددات markers أو مواضـع خاصـة لاستقبال البيانات من التطبيق، وهي تشبه سلاسل التنسيق format strings التي في بايثون، فمثلًا نريد قالبًا يستطيع استقبال رسالة الترحيب في سلسلة، ثم يدرجها في قالب، سيكون هذا القالب كما يلي:

```
<!doctype  html>
<head><title>Hello from Flask</title></head>
<body>
   <h1>{{message}}</h1>
</body>
</html>
```

إذا نظرنا إلى `{{message}}` فسنجد المواضع التي ذكرناها، فتلك الأقواس المزدوجة هي مواضع استقبال البيانات، والمتغير `message` هو اسم المتغير الذي سيدخله محرك القالب إلى HTML.

نحفظ المثال السابق في ملـف باسـم hello.htm داخـل مجلـد باسـم `templates` في مجلـد المشـروع، واسم القالب مهم هنـا لأنـه المكـان الـذي يتوقـع إطـار Flask أن يجـد فيـه ملفـات القـوالب، يتبقى الآن بعض التعديلات على شيفرة بايثون الخاصة بنا لربطها بمحرك القوالب، وسنغير الشيفرة السابقة لتكون كما يلي:

```
from flask import Flask, render_template

helloapp = Flask(__name__)

@helloapp.route("/")
def index():
   return render_template("hello.htm", message="Hello again from
Flask")

if __name__ == "__main__":
    helloapp.run()
```

الاختلاف الأول في تعليمة `import`، حيث غيرنا النمط لنستطيع استيراد الأسماء التي نحتاج إليها مباشرةً.

نلاحظ الآن اختفاء شيفرة HTML من ملف بايثون الخاص بنا، ونمرر الرسالة مثل سلسـلة عاديـة إلى الدالـة `render_template` مع اسم القـالب، ونـترك البـاقي لإطـار Flask، لكننـا نريـد أن نضـمن مطابقـة الوسـائط المسماة في استدعاء هذه الدالة للأسماء الـتي في مواضـع القـالب، ثم نعـرف كيـف نقـرأ البيانـات الـواردة من طلبات http لنستطيع تكرار نسخة CGI التي نفذناها من تطبيق "hello user".

31.4 استخدام Flask في مثال Hello User

نريد الآن تعديل التطبيق ليعمل مع استمارة HTML التي استخدمناها من قبل، فنبدأ بتحويل HTML إلى قالب نرسله عند تحديد المستخدم لنقطة نهاية "hello"، ولا نحتاج إلى إضافة أية محددات خاصة إلى HTML لأننا لا ندرج أي بيانات، لكن يجب تغيير الخاصية `method` للاستمارة إلى `"POST"`، وكذلك الخاصية `action` لتعكس منفذ إطار Flask -الذي هو 5000- ونقطة النهاية المطلوبة للرابط، والتي هي `sayhello`، ثم نحفظ ذلك في مجلد `templates` باسم `helloform.htm` ليستطيع إطار Flask العثور عليه، وسيبدو بعد تلك التعديلات كما يلي:

```
<!doctype html>
<html>
  <head>
    <title>Hello user page</title>
  </head>
  <body>
    <h1>Hello, welcome to our website!</h1>8000
    <form name="hello"
          method="POST"
          action="http://localhost:5000/sayhello">
      <p>Please tell us your name:</p>
      <input type="text" id="username" name="username"
             size="30" required autofocus/>
      <input type="submit" value="Submit" />
    </form>
  </body>
```

31.4.1 كتابة شيفرة Flask

نحتاج الآن لإضافة تابعين جديدين إلى تطبيقنا، يعرض الأول منهما قالب `helloform.htm`، بينما يرد الثاني على طلب إرسال الاستمارة، ونلاحظ أن الدالة الثانية ستستخدم قالب hello.htm الأصلي، ولم نضف شيئًا سوى كتابة سلسلة رسالة جديدة:

```
from flask import Flask, render_template, request

helloapp = Flask(__name__)
```

```
@helloapp.route("/")
def index():
   return render_template("hello.htm", message="Hello again from
Flask")

@helloapp.route("/hello")
def hello():
   return render_template("helloform.htm")

@helloapp.route("/sayhello", methods=['POST'])
def sayhello():
   name=request.form['username']
   return render_template("hello.htm", message="Hello %s" % name)

if __name__ == "__main__":
    helloapp.run()
```

لاحظ أننا اضطررنا إلى إضافة `request` إلى قائمة العناصر المستورَدة لأننا نستخدمه للوصول إلى بيانات طلب http، كما عدلنا مزخرف معالج `sayhello` ليشير إلى أنه يستطيع معالجة طلبات النوع POST -وقد كان الافتراضي طلبات GET- حيث نستخرج بيانات `username` داخل الدالة من الاستمارة، وندخلها في السلسلة التي نعيدها إلى قالب hello.htm. أما معالج `hello` فهو أبسط من ذلك، إذ يرسل قالب `helloform.htm` إلى المستخدم.

إذا شغّلنا الشيفرة الآن فسيعمل الخادم، وإذا كتبنا العنوان التالي:

```
http://localhost:5000/hello
```

فسنرى نفس الاستمارة التي كانت لدينا في مثال CGI من قبل (لاحظ أن كلمة `localhost` هي اسم بديل لـ 127.0.0.1، ويمكن تذكرها بسهولة أكثر)، فإذا ملأنا الاستمارة وضغطنا زر `Submit`؛ فستظهر لنا رسالة ترحيبية.

وبهذا نكون ألغينا الحاجة إلى بدء عملية منفصلة، وبالتالي جعلنا التطبيق قابلًا للتوسيع وزيادة الحجم، كما أزلنا أي أثر لتعليمات HTML من شيفرة البرنامج لتسهيل صيانته وتعديله، وهذا غيض من فيض إمكانيات Flask، إذ يحوي الكثير من المزايا التي لم نرها بعد، فمثلًا يستطيع الاتصال تلقائيًا بقاعدة بيانات عند بدئه، وإغلاقها عندما ينتهي، كما يستطيع التحقق من أن المستخدمين سجلوا الدخول قبل تنفيذ أي تغييرات على البيانات، وهناك العديد من مصطلحات الترميز التي لم نستخدمها، والتي تعيننا في صيانة وتشغيل موقع ويب

كبير، وخاصةً في نظام القولبة الذي يحوي بدوره بعض المزايا الأخرى، وسننظر في بعضها إذ سننتقل الآن إلى إنشاء واجهة ويب أمامية لقاعدة بيانات دليل جهات الاتصال الخاص بنا.

31.5 دليل جهات الاتصال

كتبنا في فصل التعامل مع قواعد البيانات في البرمجة تطبيقًا لدليل جهات اتصال يعمل من سطر الأوامر، وسنعيد استخدام قاعدة البيانات تلك لبناء نسخة ويب مبنية على إطار Flask، وسنضيف بعض عناصر برمجة الويب الأخرى أثناء ذلك، وسيكون تطبيق ويب بسيط رغم تلك العمليات التي سننفذها، غير أنه سيوفر أساسًا كافيًا لبناء تطبيقات ويب وفهم التوثيقات والتدريبات الأخرى.

31.5.1 إعداد المشروع

لمشاريع إطار Flask عادةً بنية محددة، فهي تأخذ صورة هرمية مجلد، حيث يكون اسم المشروع في الأعلى ، متبوعًا بمجلد `static` للملفات الساكنة مثل الصور وملفات CSS، ومجلد `templates` للقوالب، وحزمة بايثون اسمها في الغالب هو اسم المشروع، والتي هي مجلد يحوي ملفًا اسمه `__init__.py` يتحكم في ما تصدره الحزمة، كما تحتوي عادةً على ملف `setup.py` يُستخدم لتثبيت الحزمة إذا كانت موزَّعةً من فهرس حزم بايثون Python Package Index أو PyPI اختصارًا، وتسهل هذه الهرمية توزيع التطبيق باستخدام أدوات بايثون القياسية مثل `pip`.

لكننا لن نوزع التطبيق في حالة دليل جهات الاتصال، لذا سنكتفي بترتيب بسيط مبني على المجلدات القياسية الثلاثة، إضافةً إلى مجلد رابع لملف قاعدة البيانات:

```
addressbook
    static
    templates
    db
```

سننسخ ملف قاعدة البيانات من الفصل السابق إلى مجلد db هنا.

31.5.2 إنشاء قالب HTML

سيكون لدينا صفحة ويب واحدة تتكون من استمارة بسيطة لإدخال البيانات، فيها ثلاثة أزرار هي `Create` و`Find` و`List all`، حيث سيستخدم زر `Create` البيانات التي في الاستمارة لإضافة مدخل جديد إلى قاعدة البيانات، ويعرض زر `Find` قائمةً بجميع العناوين المطابقة للبيانات التي في الاستمارة، بينما يعرض زر `List all` القائمة الكاملة للعناوين.

سنستخدم مزيةً جديدةً لمحرك القوالب، وهي القدرة على تكرار صفوف HTML وفقًا لمجموعة بيانات الدخل، وسيُدمج هذا مع عنصر HTML جديد هو وسم الجدول `<table>` وعائلته التي سنستخدمها لعرض قائمة العناوين.

وسيبدو القالب كما يلي:

```
<!doctype html>
<html>
  <head>
    <title>Flask Address Book</title>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1">

    <style>
      h1 {text-align: center;}
      label {
         width: 20em;
         text-align: left;
         margin-top: 2px;
         margin-right: 2em;
         float: left;}
      input {
         margin-left: 1em;
         float: right;
         width: 12em;}
      input.button {width: 6em; text-align: center; float: left;}
      br {clear: all;}
      div.buttons {float: left; width:100%; padding: 1em;}
    </style>

  </head>

  <body>
    <header>
    <h1>Address Book Website</h1>
    </header>
```

```
    <div class="content">
      <form name="hello"
            method="POST"
            action="http://localhost:5000/display">
      <fieldset><legend>Address</legend>
            <label>First name:
              <input type="text" id="First" name="First"
                required autofocus/>
            </label>
            <label>Second Name:
               <input type="text" id="Last" name="Last"
                      required />
            </label>
            <br />
            <label>House Number:
                   <input type="text" id="House" name="House" />
            </label>
            <label>Street:
                   <input type="text" id="Street" name="Street"
                          required />
            </label>
            <br />
            <label>District:
                   <input type="text" id="District" name="District" />
            </label>
            <label>Town:
               <input type="text" id="Town" name="Town"
                      required />
            </label>
            <br />
            <label>Postcode:
               <input type="text" id="Postcode" name="Postcode"
                      required />
            </label>
            <label>Phone:
               <input type="text" id="Phone" name="Phone" />
```

```
            </label>
      </fieldset>
      <div class="buttons">
           <input class="button" type="submit" value="List All"  />
           <input class="button" type="submit" name="Filter"
value="Filter" />
           <input class="button" type="submit" name="Add" value="Add"
/>
           <input class="button" type="reset" value="Clear" />
      </div>
      </form>
      <br />
      <div class="data">
         <table id="addresses">
         <tr>
            <th>First</th>
            <th>Second</th>
            <th>House#</th>
            <th>Street</th>
            <th>District</th>
            <th>Town</th>
            <th>Postcode</th>
            <th>Phone</th>
         </tr>
         {% for row in book %}
         <tr>
            <td>{{row.First}}</td>
            <td>{{row.Last}}</td>
            <td>{{row.House}}</td>
            <td>{{row.Street}}</td>
            <td>{{row.District}}</td>
            <td>{{row.Town}}</td>
            <td>{{row.PostCode}}</td>
            <td>{{row.Phone}}</td>
         </tr>
         {% endfor %}
         </table>
```

```
        </div>
    </div>

    </body>
</html>
```

يوجد عدد كبير من العناصر الجديدة هنا، بما في ذلك قسم `<style>` الذي يتحكم في تخطيط الاستمارة، ويمكن الرجوع إلى أي شرح للغة CSS -مثل توثيق CSS في موسوعة حسوب- لمعرفة ما تفعله الشيفرة هنا إذ لا يتسع المقام لشرحها بسبب بعدها عن الغرض من الفصل.

كما توجد بعض مزايا HTML الجديدة:

- يوضع ترميز المحارف في وسم `meta` داخل وسم `head`، وتجب إضافة هذا الوسم في صفحات الويب الحديثة خاصةً إذا كنا نستخدم محارف غير قياسية، وإلا فسيكون لدينا في الصفحة أجزاء غير قابلة للقراءة.
- الوسم `meta` هو منفذ للعرض viewport، وهو مصمم لتحسين عرض الصفحة على الأجهزة المحمولة، ويُفضل إضافة هذا الوسم إلى الصفحة.
- استخدمنا عددًا من وسوم `div`، وهي مجرد أدوات تنظيمية لا تظهر على الشاشة إلا إذا تعمدنا إظهارها باستخدام CSS، وهي تشبه ودجات `Frame` التي استخدمناها في برامج Tkinter الرسومية، وتفيدنا في التحكم في نطاق الأوامر لكل من CSS وجافاسكربت.
- تحتوي الاستمارة `form` على ثلاثة عناصر جديدة، يساهم كل منها في تحسين مظهرها، هي العنصر `fieldset` الذي يجمع عدة حقول داخل إطار مرئي يُدرج فيه العنصر `legend`، والعنصر `label` الذي يغلف حقول `input` ليجعلها معًا على الشاشة.
- نختم الاستمارة بثلاثة أزرار من النوع `submit` لها الخاصية `name` التي تُرسَل إلى الخادم، ونستطيع استخدام تلك البيانات لتحديد الدالة التي يجب تنفيذها، أما الزر الأخير فيكون من النوع `reset`، وهو يمسح حقول الاستمارة ولا يرسل شيئًا إلى الخادم.
- قسم `<table>` الذي يبدأ بصف من الترويسات `<th>` و`<tr>`.
- يوفر الجزء التالي ميزةً جديدةً لمحرك القوالب، وهي القدرة على تنفيذ حلقات تكرارية واستبدال عدة أجزاء من البيانات، وهي الحقول التي في كل صف في حالتنا، وتُحدَّد بنى التحكم هذه بمحارف `{%...%}`، ويدعم المحرك عدة بنىً مختلفة إضافةً إلى الحلقة الموضحة هنا.

31.5.3 كتابة شيفرة إطار Flask

لا نحتاج إلى إضافة الكثير من الشيفرات الجديدة في هذا المشروع، فعناصر إطار Flask مجرد توسيع طفيف من المثال السابق لتحديد أي زر من أزرار `submit` الثلاثة قد ضُغط، وبمجرد أن ننفذ ذلك نستدعي دوال بايثون العادية التي تعدل قاعدة البيانات باستخدام وحدة `sqlite3` كما شرحنا في فصل قواعد البيانات، وتعيد تلك الدوال قائمةً من عناصر القواميس، بمقدار عنصر واحد لكل صف في قاعدة البيانات، وتمرَّر تلك القائمة إلى محرك عرض القوالب template rendering engine الذي ينفذ مهمة إدخال HTML إلى الصفحة التي أنشئت، وستبدو الشيفرة كما يلي:

```
from flask import Flask, render_template, request, g
import sqlite3, os

addBook = Flask(__name__)
addBook.config.update(dict(
   DATABASE=os.path.join(addBook.root_path,'static','address.db'),
   ))

@addBook.route("/")
def index():
   data = findAddresses()
   return render_template("address.htm", book=data)

@addBook.route("/display", methods=['POST'])
def handleForm():
   if 'Filter' in request.form:
      query= buildQueryString(request.form)
      return render_template('address.htm', book=findAddresses(query))
   elif 'Add' in request.form:
      addAddress(request.form)  # add a new entry
   return render_template("address.htm", book=findAddresses())

# Note: flask.g is the "global context" object
# that lives for the life of the application.
def get_db():
    if not hasattr(g, 'sqlite_db'):
       try:
```

```
            file = addBook.config['DATABASE']
            db = sqlite3.connect(file)
            db.row_factory = sqlite3.Row  # return dicts instead of
tuples
        except: print("failed to initialise sqlite")
        g.sqlite_db = db
    return g.sqlite_db

@addBook.teardown_appcontext
def close_db(error):
    if hasattr(g, 'sqlite_db'):
         db = g.sqlite_db
         db.commit()
         db.close

def buildQueryString(aForm):
    base = "WHERE"
    test = " %s LIKE '%s' "
    fltr = ''

    for name in ('First','Last',
                 'House','Street',
                 'District','Town',
                 'PostCode','Phone'):
        field = aForm.get(name, '')
        if field:
            if not fltr:  fltr = base + test % (name, field)
            else: fltr = fltr + ' AND ' + test % (name, field)
    return fltr

def findAddresses(filter=None):
    base = """
SELECT First,Last,House,Street,District,Town,PostCode,Phone
FROM address %s
ORDER BY First;"""

    db = get_db()
```

```
    if not filter: filter = ""  # empty -> get all
    query = base % filter
    cursor = db.execute(query)
    data = cursor.fetchall()
    return data

def addAddress(aForm):
    db = get_db()
    cursor = db.cursor()

    first   = aForm.get('First','')
    last    = aForm.get('Last','')
    house   = aForm.get('House','')
    street  = aForm.get('Street','')
    district= aForm.get('District','')
    town    = aForm.get('Town','')
    code    = aForm.get('PostCode','')
    phone   = aForm.get('Phone','')

    query = '''INSERT INTO Address
               (First,Last,House,Street,District,Town,PostCode,Phone)
               Values ("%s","%s","%s","%s","%s","%s","%s","%s;");'''
%\
               (first, last, house, street, district, town, code,
phone)
    cursor.execute(query)

if __name__ == "__main__":
   addBook.run()
```

نلاحظ بعض النقاط المهمة هنا:

- نستورد الاسم `g` من إطار Flask، وهو كائن خاص يوفره Flask ليحمل قيم مستويات التطبيق، ونستفيد من المبدأ الذي يقوم عليه، رغم غرابة اسمه، وهو يشبه المعامِل `self` في البرمجة الكائنية لكنه ينطبق هنا على تعاملات الويب بدلًا من الكائن.
- نعِدّ موقع قاعدة البيانات في البنية الخاصة `addbook.config`، وهناك عدة عناصر أخرى يجب إعدادها هنا في الموقع الكامل، بما في ذلك اسم المستخدم وكلمة المرور للمدير admin، وتخزَّن هذه

الأمـــــور عـــــادةً في ملـــــف config، ويُســـــتخدم أمـــــر خـــــاص لتحميلهـــــا. كما يجب الانتباه إلى خيار DEBUG الذي يؤدي لطبع معلومات كثيرة إلى الطرفية عند ضبطه على القيمة True، لأن هذه المعلومات مفيدة للغاية في حال حدوث المشاكل أثناء التطوير.

- تتحقق الدالة get_db من كون الخاصية sqlite_db مضبوطةً أو لا قبـل أن تضـبطها، وهـذا يضـمن وجود اتصال واحد فقط لقاعدة البيانات.
- كما تضبط هذه الدالة row_factory من sqlite3.Row، وفائدته أنه يجعل Sqlite تعيـد قائمـةً من كائنات Row بدلًا من صفوف tuple تحتوي على قيم، وتُعامَل كائنات Row تلك على أنها قواميس، وهو ما يبحث عنه Flask تحديدًا في محرك القوالب الخاص به، لذا يوفر سطر الشيفرة ذاك علينـا كثـيرًا من تنسيق البيانات.
- ينبغي أن توجد معالجات أخطـاء try/except حـول الكثـير من التعليمـات البرمجيـة، خاصـةً أقسـام قواعد البيانات، لكننا لم نشأ الإطالة أكثر من اللازم، وسنرى مثالًا في دالة get_db يوضح أننـا نسـتطيع استخدام تعليمات print لعرض خرج التنقيح debug output في الطرفية.
- يُستخدم هنا مزخرِف جديد هو @addBook.teardown_appcontext لرفع راية إلى إطار Flask بأنـه يجب تشغيل الدالة close_db عند إغلاق التطبيق.
- تساعد الدالة buildQueryString في بناء شرط WHERE الخـاص باسـتعلامنا ديناميكيًـا، وقـد تبـدو معقدةً إلا أنها مجرد تعديل بسيط على سلسلة نصية، ويسمح لنا هـذا الأسـلوب باسـتخدام دالـة واحـدة لإيجاد جميع العناوين عند عدم استخدام مرشحات filters، أو العناوين المطلوبة فقط.
- تكـون عمليـات البحث أكـثر مرونـةً باسـتخدام عامـل LIKE الخـاص بـ SQL، بـدلًا من اختبـار التكـافؤ equality test، بما في ذلك القدرة على استخدام علامة % محرف بدل في SQL.
- نستخدم تابع القاموس get لجلب القيم من الاستمارة، وهذا يضمن حصولنا على قيمة افتراضـية عنـد عدم وجود المفتاح لسبب ما، وهي سلسلة فارغة في حالتنا.

31.5.4 تشغيل دليل جهات الاتصال

سنغير مجلد التطـبيق ونشـغل الملـف address.py، ثم نسـتخدم المتصـفح لزيـارة localhost:5000، وبهذا تطابق عملية التشغيل هذه عملية التشغيل السابقة، ويجب أن نرى نفس الأسماء والعناوين التي أنشأناها في فصل قواعد البيانات في المتصفح.

كما ينبغي أن نستطيع الآن إدخال القيم في الاستمارة، وترشيح النتائج لمطابقة كلمات البحث -نسـتخدم % محرف بدل wildcard- وكذلك نستطيع إضافة إدخالات جديدة، وبما أنه لا يوجد تحقق من الحقول أو الاسـتمارة فيجب أن نراعي إدخال قيم منطقية مناسبة، أو نتلافى أي أخطاء يدويًا باستخدام محث sqlite3.

ينبغي أن يقدم هذا الفصل فكرةً عن متطلبات إنشاء تطبيق ويب، وهناك المزيد لتعلمه بما في ذلك تقنيات برمجة المتصفحات، وأمور الأمان في جانب الخادم، وملفات تعريف الارتباط cookies، وإطلاق الموقع الحي في صورته النهائية على الإنترنت، وكل هذا خارج عن سياق شرحنا، لذا يُرجع فيه إلى تطوير التطبيقات باستخدام لغة Python من أكاديمية حسوب، أو دورة تطبيقات الويب باستخدام PHP، وغيرها من مقالات البرمجة والكتب البرمجية في الأكاديمية كما توفر شركات استضافة الويب توثيقات مفصلةً عن تحقيق الاستفادة القصوى من المواقع التي ننشئها.

31.6 خاتمة

نأمل في نهاية هذا الفصل أن تكون تعلمت ما يلي:

- تحسّن أطر عمل الويب إمكانية توسيع المواقع وصيانتها.
- توجه أطر عمل الويب نقاط النهاية إلى الدوال.
- توجد عدة إطارات عمل للويب في السوق الآن.
- إطار Flask يمثل حلًا وسطًا بين البساطة والإمكانيات.
- يستخدم Flask مزخرفات decorators لتوجيه نقاط النهاية.
- كما يستخدم نظام قوالب لتمرير البيانات إلى صفحات الويب.

دورة تطوير التطبيقات
باستخدام لغة بايثون

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

32. البرمجة المتزامنة وفائدتها في برمجة التطبيقات

سنغطي في هذا الفصل من ما يلي:

- آلية ووقت استخدام التزامن concurrency والخيوط threads.
- مثال على تقسيم الحمل بين العمليات.
- الوصول إلى البيانات المشتركة باستخدام الخيوط.
- بعض الاحتمالات والمحاذير الأخرى.

رأينا إضافة هذا الفصل إلى شرحنا بعد تفكير طويل نظرًا لأهمية موضوعه، وسبب ترددنا في ذلك أن التزامن عمومًا -والخيوط على وجه الخصوص- موضوع يصعب فهمه، رغم مسارعة الكثيرين إليه ظنًا منهم أنه يحل مشاكلهم، لكنهم يزيدونها سوءًا لضعف فهمهم لمفاهيم التزامن والخيوط، واللتان هما أداتان قويتان عند استعمالهما في المواضع المناسبة عن فهم وإدراك، لكن نرى أن يكونا الملاذ الأخير، نلجأ إليهما بعد فشل جميع الحلول الأخرى، وقد عملنا -يقصد المؤلف نفسه- على عدة مشاريع تستخدم التزامن بصورة أو بأخرى في الأعوام الأربعين الماضية، لكننا لم نستخدمها في مشاريع شخصية من قبل، وربما يعطي هذا تصورًا عن توقيت استخدام هذه التقنية، ونوعية المشاريع التي تُستخدم فيها.

32.1 مفهوم البرمجة التزامنية وتوقيت استخدامها

تشير البرمجة التزامنية Concurrent programming ببساطة إلى برنامج يحوي عناصر تعمل على التوازي في نفس الوقت، وهو ما نطبقه على حواسيبنا كل يوم، لكننا نترك أمر إدارة تلك البرامج لنظام التشغيل، ليجدولها لتعمل بالترتيب، ويوزع حمل التشغيل على المعالجات والنوى kernels الموجودة فيها، فمفهوم

تقسيم المهمة إلى عدة أجزاء والعمل على كل جزء على حدة ليس جديدًا، وهو يشبه إنشاء فرق مختلفة للتعامل مع مشكلة كبيرة تُقسم إلى عدة أجزاء، ليعمل كل فريق على جزء منها.

وقد اتبعنا تلك الأساليب في علم الحواسيب على مدار العقود الماضية، خاصةً في الخوادم، فمثلًا يستقبل خادم الويب طلبات ويعالجها، لكن إذا كان الموقع كبيرًا وطلباته كثيرة ومتلاحقة، وكان الخادم يُعالج كل طلب قبل بدء الطلب الذي يليه؛ فستطفح قائمة انتظار المدخلات للخادم، لذلك يقرأ الخادم الطلب من قائمة الانتظار، فإذا كان الطلب ملف html بسيطًا فيعيده، أما إن كان أعقد من هذا فيشتق الخادم عمليةً فرعيةً للتعامل معه، ويعود هو لقراءة الطلب التالي.

وقد رأينا من قبل كيفية إنشاء عمليات من شيفرة بايثون الخاصة بنا في فصل التواصل مع نظام التشغيل باستخدام وحدة `subprocess`، ورأينا منظورًا مختلفًا في فصل التواصل بين العمليات، حيث استخدمنا `fork` لإنشاء عملية فرعية تكون نسخةً من العملية الأصلية، وتعمل كلا التقنيتين بسلاسة إذا كنا نرغب في عمليات قليلة منفصلة تمامًا عن بعضها، لكنهما لا تناسبان الحالة التي نحتاج فيها إلى عمليات كثيرة، كما في حالة معالجة خادم الويب لآلاف الطلبات كل دقيقة، وإحدى المشاكل التي قد تحدث عند استخدامهما لحالة الطلبات الكثيرة هو ما ينتج عن بطء إنشاء العملية الجديدة -بمقياس سرعة الحواسيب-، كما تكلف كل عملية الكثير من الأحمال الزائدة من استهلاك الذاكرة وغيرها، وهنا يبرز دور الخيوط therads، وهي تشبه العمليات الدقيقة التي تعمل داخل العملية الأصل، فتتشارك جميعها نفس مساحة الذاكرة ونفس الشيفرة، وتأتي تسمية الخيوط من فكرة أن الشيفرة الخاصة بنا تعمل من الأعلى إلى الأسفل -مع قفزات وحلقات تكرارية في طريقها-، فهي مثل بكرة من القطن أمسكنا طرف خيطها وتركناها تسقط، فكل خيط جديد يشبَّه بكرة قطن جديدة نسقطها بالتوازي مع الخيط الأول، وكل خيط له مسار تنفيذ خاص به في نفس الشيفرة.

لقد أشرنا أن التزامنية تتعلق بشكل ما بالأداء، خاصةً في حالة الأحمال الكبيرة من البيانات، والحق أن هذا هو السبب الوحيد لاستخدامها، فإذا واجهنا مشكلة عند زيادة البيانات ونكون قد بذلنا كل ما نستطيع من تعديلات على الأداء الأساسي -لعملية تبادل أو وحدة بيانات واحدة-، فحينئذ قد نحتاج إلى إعادة التفكير في شأن تقسيم الأحمال.

32.2 اختيار أسلوب التزامن

بعد أن أثبتنا فائدة التزامن في تحسين الأداء في حالة الأحمال الكبير، كيف نقرر التقنية التي سنستخدمها لمعالجة مهمة ما؟ أيهما أفضل لمهمتنا العمليات المتعددة أم الخيوط المتعددة؟ والجواب أننا ننظر في عدة عوامل، حيث:

- نستخدم الخيوط إذا كان في المهمة انتظار لنشاط الإدخال والإخراج، مثل رسائل الشبكة أو نتائج استعلامات قواعد البيانات.

- نستخدم الخيوط إذا تطلبت المهمة مشاركة البيانات بين عدة "خيوط"، لكن يجب تذكر قفل lock البيانات المشارَكة أثناء التحديثات.
- نستخدم العمليات الفرعية إذا كان في المهمة معالجة خالصة للبيانات data crunching، وقد صممت الوحدة `multiprocessing` في بايثون خصيصًا لهذه الحالة.
- إذا كانت المهمة الفرعية تعمل لوقت طويل، أو لا تُستدعى إلا لوقت قصير، فننشئ عملية خادم طويلة التشغيل long running server process، كما فعلنا في فصل التواصل بين العمليات، لنرسل البيانات إليها عند الحاجة.

> تواجه بايثون مشكلةً كبيرةً في منظورها للخيوط تنطوي على آليةً داخليةً معقدةً، تسمى قفل المفسر العام Global Interpreter Lock أو GIL اختصارًا، يتمثل تأثير هذا القفل في منع خيوط العمليات الحسابية من العمل بالكفاءة التي نريدها، إلى الحد الذي دفع البعض إلى القول بعدم استخدام الخيوط في بايثون بالكلية، غير أن في مبالغةً كبيرةً، إذ إن الخيوط طريقة ممتازة للتعامل مع كل ما يجب قفله لعمليات الإدخال والإخراج، وهذا يعني "كل شيء" تقريبًا، والمهام الحسابية الخالصة فقط هي التي تواجه مشاكل، وليست المشاكل فيها بذلك السوء كما سنرى في المثال أدناه، وسنستخدم وحدة `multiprocessing` عندما نواجه تلك المشاكل، وهي مماثلة للخيوط في استخدامها، وستنفذ نفس المهمة لكن مع استخدام العمليات بدلًا من الخيوط، وهو الخيار الذي يستهلك الموارد كما ذكرنا في بداية الفصل.

32.3 البدء بالبرمجة المتزامنة

بما أننا قررنا استخدام التزامن فلننظر في الشيفرة المطلوبة لها، لدينا وحدتان في المكتبة القياسية تتعلقان بالخيوط، هما `Thread` و`threading`، والوحدة الأخيرة هي وحدة عالية المستوى بُنيت فوق وحدة `Thread`، ولا نحتاج إلا إلى النظر فيها، أما في حالة العمليات المتعددة فنستخدم وحدة `multiprocessing` التي تعمل تقريبًا بنفس طريقة `threading`.

32.3.1 تحديد المشكلة

لننظر في مشروع بسيط نريد فيه حساب مضروب أول N عدد من مجموعة ما من الأعداد، فنكتب دالةً لحساب المضروب ثم نضعها في حلقة تكرارية ونخزن النتائج، وستبدو كما يلي:

```
import time, sys

factorials = []

def fact(n):
    if n < 2: return 1
```

```
    result = 1.0
    for x in range(2,n+1):
        result *= x
    return result

def do_it(f,lo,hi):
    global factorials
    for n in range(lo,hi):
        factorials.append(f(n))

if __name__ == "__main__":
    hi = int(sys.argv[1]) + 1
    start = time.time()

    do_it(fact, 1, hi)
    print('Time for %s: %s' % (hi, time.time() - start))
```

نلاحظ أننا أنشأنا دالة `do_it` التي تغلف الحلقة التكرارية الخارجية واستدعاءات المضروب، وهي ليست ضروريةً لكننا سنرى نفعها لاحقًا عندما ننظر في التزامن، وقد جعلنا عدد مرات التكرار وسيطًا في سطر الأوامر، وأضفنا مؤقتًا لإظهار الزمن الذي تستغرقه.

احفظ ذلك في ملف باسم no_threads.py وشغله لأول 100 عدد كما يلي:

```
$ python3 no_threads.py 100
```

إذا شغلنا هذا الأمر فسنجد أن الوقت المستغرق أقل من ثانية، نظرًا لسرعة الحواسيب هذه الأيام، لكن جرب ذلك مع أول ألف عدد مثلًا وانظر الوقت المستغرق، وقد جربناها لكل من أول 100 عدد، وأول 1000، وأول 10000، وخرجنا بالنتائج التالية:

```
$ python3 no_threads.py 100
Time for 100: 0.0006973743438720703
$ python3 no_threads.py 1000
Time for 1000: 0.06539225578308105
$ python3 no_threads.py 10000
Time for 10000: 6.800917863845825
```

نلاحظ أن السرعة جيدة إلى أن نصل إلى الألف عدد، لكنها تسوء فجأةً مع زيادة العدد عن ذلك، ونريد تحسين هذا الأداء لتلك الأعداد الكبيرة، وهنا يأتي دور التزامن.

32.3.2 إضافة التزامن إلى العملية

باتباع الإرشادات التي شرحناها أعلاه، سنختار `multiprocessing` بدلًا من `threading` لهذه المهمة، ونريد أن نعرف مقدار العدد المُدخَل أولًا، فإذا كان أكثر من 1000، فسنقسم العملية إلى عدة فروع، وسيكون ذلك أبطأ في الأعداد الكبيرة، لذلك بدلًا من تقسيم البيانات بالتساوي بين العمليات سنقسمها عند 75% -عشوائيًا-.

ستكون الشيفرة كما يلي:

```
import time, sys
import multiprocessing

factorials = []

def fact(n):
    if n < 2: return 1

    result = 1.0
    for x in range(2,n+1):
        result *= x
    return result

def do_it(f,lo,hi):
    global factorials
    for n in range(lo,hi):
        factorials.append(f(n))

if __name__ == "__main__":
    hi = int(sys.argv[1]) + 1

    start = time.time()
    if hi > 1000:
        hi_1 = int(hi * 0.75)
        p1 = multiprocessing.Process(target=do_it, args=(fact,1,hi_1))
```

```
            p2 = multiprocessing.Process(target=do_it,
    args=(fact,hi_1,hi+1))
            p1.start()
            p2.start()
            p1.join()
            p2.join()
        else:
            do_it(fact, 1, hi)

        print('Time for %s: %s' % (hi, time.time() - start))
```

نلاحظ أننا أنشأنا كائنيْ `Process`، ومررنا الدالة التي نريد تشغيلها -وهي `do_it`- إضافةً إلى الوسائط اللازمة في صف tuple، ثم استدعينا `start` لتشغيل العمليات، أخيرًا استخدمنا التابع `join` لجعل البرنامج الرئيسي ينتظر انتهاء العمليات.

```
$ python3 processes.py 100
Time for 100: 0.0006771087646484375
$ python3 processes.py 1000
Time for 1000: 0.06577491760253906
$ python3 processes.py 10000
Time for 10000: 3.690302610397339
```

نلاحظ أن الوقت المستغرق للعملية الكبيرة قد انخفض إلى النصف تقريبًا، لكن ما نفذناه لم يكن بسيطًا أو مباشرًا، فلم نقسم البيانات بالتساوي، فإذا جربت تقسيمها عند 50% فستجد أن الوقت المستغرق زاد مرةً أخرى.

يبدو أن التزامن يعطينا تحسنًا كبيرًا في الأداء، لكن في الواقع لدينا خلل كبير في برنامجنا، فإذا طبعنا `factorials` -وهي قائمة النتائج- فسنجدها فارغةً!

والسبب في هذا أن كل عملية فرعية هي نسخة من العملية الأصل، وبالتالي لها نسختها الخاصة من قائمة `factorials`، والتي نفقد بياناتها عند انتهاء العملية، لذلك نحتاج إلى طريقة لتمرير البيانات إلى العملية الأصل، مما يعني الكتابة إلى ذاكرة مشتركة -انظر وحدة `mmap`- أو إلى قاعدة بيانات أو إلى ملف.

نستخدم في تلك الحالات كلها عمليات الإدخال والإخراج، لذا فلم لا نستخدم الخيوط بما أنها تتشارك الذاكرة! فنكون قد أنشأنا نسختنا الخاصة من معضلة Catch-22)!

لكن لحسن الحظ نادرًا ما يكون لدينا عمليـات نحتـاج إلى تقسـيمها بـالتوازي دون أن يكـون فيهـا عمليـات إدخال وإخراج، لذا فإن الخيوط هنا حل يمكن تنفيذه، وسننظر في مثال "مفتعل" إلى حد ما لمجرد موازنة هيكل الشيفرة مع مثال تعدد العمليات السابق.

32.4 استخدام الوحدة threading

سننشئ في هذا المثال خيطًا مساعِدًا لتحديث المتغير `theTime` في كل ثانية، وسيتكرر البرنامج الرئيسي بلا نهاية ليطبع المتغـير في كـل مـرة يتغـير فيهـا، ولضـمان تشـغيل الخيـوط بسلاسـة سنضـيف تـأخيرًا زمنيًـا باستخدام `time.sleep`، التي يعاملها المفسر على أنها عملية إدخال/إخراج.

```
import time, threading

theTime = 0

def getTime():
    global theTime
    while True:
        theTime = time.time()
        time.sleep(1)

def main():
    global theTime
    current = theTime

    thrd = threading.Thread(target=getTime)
    thrd.start()

    while True:
        if current != theTime:
            current = theTime
            print(time.asctime(time.localtime(current)))
        time.sleep(0.01)

if __name == "__main__":
   main()
```

لقد أنشأنا في المثال أعلاه دالةً تغلف مهمة الخيوط، لضمان وجود عملية `time.sleep` فيها -أو أي دالة I/O أخرى-، ثم أنشأنا خيطًا نسند فيه دالتنا على أنها الهدف، كما فعلنا في العمليات أعلاه، ونبدأ تشغيل الخيط في الخلفية ثم ننتقل إلى الحلقة الأساسية التي تتحقق من المتغير في كل جزء من مئة من الثانية.

عند الحاجة إلى إنهاء البرنامج نكتب Ctrl+C أو نستخدم مدير المهام في نظام التشغيل لإنهاء هذه العملية.

إذا شغلنا الشيفرة السابقة فستكون النتيجة شبيهةً بما يلي:

```
$ python3 clock.py
Sat Dec 30 11:30:33 2017
Sat Dec 30 11:30:34 2017
Sat Dec 30 11:30:35 2017
Sat Dec 30 11:30:36 2017
...
```

كان من الممكن تحقيق ذلك بسهولة دون استخدام الخيوط، لكنها تعطينا فكرةً عما تبدو عليه أبسط شيفرة تحوي خيوطًا، فإذا وازنّاها مع شيفرة العمليات المتعددة فسنجد أنهما متطابقتان تقريبًا في بنيتيهما، والفرق بينهما أن شيفرة الخيوط تستطيع تحديث متغير `getTime` العام الذي تستطيع الدالة الرئيسية الموجودة في العملية الأصل أن تراه، نستطيع الآن أن نعود ونستبدل العمليات بالخيوط في المثال السابق ونوازن بين النتائج، وسنجد أن نسخة الخيوط لم تقدم مزيةً زمنيةً على نسخة العمليات.

سنكتفي بهذا القدر من شرح التزامن لأنه موضوع معقد فيه الكثير من الفخاخ الدقيقة، وننصح بعدم استخدامه إلا عند الضرورة، وحتى في تلك الضرورة فالشرح الموجود هنا يمثل نقطة بداية فقط، يمكن الانتقال بعدها إلى شروح أكثر تفصيلًا وعمقًا.

للمزيد من المعلومات، ننصحك بالإطلاع على دورة تطوير التطبيقات باستخدام لغة Python التي تشرح الكثير من المفاهيم الأساسية لبناء التطبيقات في بايثون.

32.5 خاتمة

نأمل في نهاية هذا الفصل أن تكون تعلمت ما يلي:

- يُستخدم التزامن لتحسين الأداء.
- يُنفَّذ التزامن بتشغيل عمليات أو خيوط إضافية.
- تعمل العمليات بشكل مستقل عن بعضها، وتتواصل مع العملية الأصل باستخدام الأنابيب pipes.
- تعمل الخيوط افتراضيًا داخل العملية الأصل، وتتشارك الموارد -مثل الذاكرة وغيرها- فيما بينها.

- يجب إقفال الموارد المشتركة قبل تغيير الحالة لتجنب حدوث تعارضات.
- توجد طرق أخرى للتزامن، كما في وحدة `asyncio` في بايثون التي تستخدم منظورًا مختلفًا.

خاتمة

الحمد لله تعالى أن وفقنا إلى ترجمة مادة هذا الكتاب النافع، ليكون معينًا لمن يرغب في دراسة البرمجة والعمل بها، وهو جِد مفيد إذ ركز الكاتب على المنهج العملي أكثر من التبحر في النظريات الكامنة وراء المفاهيم البرمجية، والتي تهم طلاب علوم الحاسوب أكثر ممن يرغب في العمل بالبرمجة بعد بضعة أشهر.

وإن الغالب على من ينتهج مثل هذا الأسلوب أن يكون من العاملين في السوق مباشرة، بعيدًا عن أروقة الجامعات النظرية، وهي حالة صاحب هذا الكتاب، إذ أن له نحوًا من أربعة عقود في العمل البرمجي لحل مشاكل حقيقية في السوق، إضافة إلى خلفيته في هندسة الإلكترونيات التي تعطيه درجة أخرى من الفهم العميق لعمل البرمجيات مع العتاد نفسه، وينعكس هذا فيما يشرحه في الكتاب ويركز عليه.

وهذا النهج أقرب ما يكون لما سيلاقيه المبرمج في السوق، حيث يأتيه مديره في الشركة أو العميل بمشكلة يريد حلها، فيكتب لها برنامجًا أو تطبيقًا يحلها، ثم يطرأ تعديل جديد أو يخرج إطار عمل جديد بمزايا جديدة، فحينئذ يعيد كتابة الحل أو تعديله وفقًا للتغيرات المستجدة، وذلك تحديدًا ما اتبعه الكاتب في شرحه، إذ اختار برنامجًا أو أكثر ليكتبه في أحد الفصول، ثم يعود إليه كلما شرح مفهومًا جديدًا ليدخله في البرنامج، أو يعيد كتابة البرنامج وفقًا للمفهوم الجديد الأفضل.

وبعد، فإن الطريق بعد هذا الكتاب تتفرع وفق المنصة التي سيكتب المبرمج تطبيقاته لها، ونوعية المشاكل التي يرغب بحلها، فالبرامج التي ستعمل على منصة أندرويد تختلف لغاتها عن تلك الموجهة للهواتف العاملة بنظام iOS، وكلها تختلف عن البرامج الموجهة للحواسيب المكتبية التي تختلف كذلك باختلاف نظم التشغيل التي عليها، وهكذا، وإن كانت توجد أدوات ومنصات يمكن استخدامها لكتابة البرنامج مرة واحدة ثم تشغيله على تلك المنصات جميعها أو بعضها.

وسنذكر فيما يلي مصادر يستزيد منها المتعلم قراءةً وبحثًا ليرى المجال المناسب له، ثم نذكر بعض المواد التعليمية لأشهر تلك التخصصات لينطلق المتعلم فيها، وكيف يطور مهاراته البرمجية في مشاكل حقيقية

موجودة تنتظر حلها، ويضع يده مع الفرق العاملة على تلك المشاكل والبرامج فيستفيد منهم ويتعرف على بيئات العمل البرمجية كيف تكون.

32.6 موضوعات للدراسة

فيما يلي بعض العناوين التي ننصح القارئ بالبحث والقراءة فيها، حيث تمثل كل منها أحد أشهر التوجهات والتخصصات المطلوبة للبرمجة، وكل منها مثال فقط له نظائر وقرائن، وعلى القارئ هنا أن يلتقط أثناء قراءته ما يجذب فضوله، فالفضول سمة للراغب في العمل في البرمجة بأي حال!

- الواجهة الرسومية GUI مع Tk و Tix.
- برمجة الويب، CGI أو إطار عمل مثل فلاسك Flask وجانغو Django.
- صناديق أدوات مثل NLTK أو PyGame.
- أطر عمل مثل Twisted و Dabo.
- قواعد البيانات، بما في ذلك قواعد البيانات غير العلائقية NoSQL مثل Mongo.

32.7 مشاريع تعليمية

يحتوي الكتاب على أفكار عديدة ليجرب القارئ فيها ويتعلم منها ومعها، وفيما يلي أفكار إضافية تزيد كل منها صعوبة عن سابقتها بترتيب تصاعدي، لكن يمكن إنجاز أغلبها بالأدوات والمفاهيم المشروحة في الكتاب، ونزيد على هذا بأنها كلها يمكن تطويرها وتحسينها أكثر بالنظر في توثيق بايثون، كذلك ستحتاج بعض تلك الأفكار إلى أن يبحث القارئ بنفسه عن أسلوب تنفيذها.

- توسيع المدقق اللغوي ليضيف مزايا إضافية.
- بناء قاعدة بيانات للأقراص المدمجة، مع وسيلة بحيث لإيجاد الأقراص وتسجيل آخر مرة تم تشغيل القرص فيها، أو عدد المرات التي تم تشغيله.
- إنشاء أداة لتوليد صفحات HTML تعرض قائمة ملفات في مجلد في هيئة روابط، كي يمكن فتحها بمجرد النقر عليها.

كذلك، من الوسائل المفيدة في تعلم خبايا بايثون ووحداتها غير التقليدية، تحدي بايثون، وهي لعبة تتكون من 33 تحدي -وقت ترجمة هذا الكتاب-، يجب حلها باستخدام خصائص بايثون المختلفة، وتعطيك إجابة كل تحدي رابطًا إلى التحدي التالي، وتزداد صعوبتها تدريجيًا في كل مستوى.

أخيرًا، يمكن النظر في كل من SourceForge و GitHub، وهما المصدرين الرئيسيين للمشاريع البرمجية مفتوحة المصدر، فابحث عن المشاريع التي تستخدم بايثون -أو VBScript أو جافاسكربت-، واختر واحدًا منها

للتعديل عليه، حتى وإن كان توثيق شيء ما فيه -كتوثيق وحدة module مثلًا-، فتلك بداية لا بأس بها لتعلم كيفية عمل الشيفرات البرمجية، ثم حاول إصلاح إحدى العلل البرمجية التي أُبلغ عنها، ثم في المرحلة التالية حاول إضافة خاصية جديدة للبرنامج، وذلك كله ممتاز للغاية لتعريض نفسك إلى مشاريع كبيرة حقيقية، والعمل في فريق كذلك.

32.8 دورات متخصصة

نذكر فيما يلي بعض الدورات عالية الجودة، باللغة العربية، التي توجه القارئ لتعلم أحد التخصصات والعمل فيه، بأمثلة تطبيقية عملية، منها المدفوع ومنها المجاني، لكن يُنظر للمدفوع منها على أنه استثمار قريب الأجل، إذ أن العائد على الاستثمار منه قد يكون في أول مشروعين، أو في راتب أول شهر وظيفي.

وقد آثرت البدء ببعض الدورات التي تقدمها أكاديمية حسوب لعدة أسباب، أنها من نفس الشركة التي تشرف على منصتي مستقل وخمسات للعمل الحر، وهما أكبر منصتي عمل حر في الويب العربي، وأن تلك الدورات تُحدث باستمرار، مما يعني أن ما تدفعه فيها لأول مرة يغطي ثمن المساقات التعليمية التي تضاف إلى نفس الدورة في المستقبل ويكون لك وصول إليها، وكذلك ضمان استعادة ما دفعته فيها في خلال ستة أشهر، ومتابعة مدربين خبراء للدارسين للإجابة على أسئلتهم وحل مشاكلهم.

32.8.1 تطوير تطبيقات الويب باستخدام لغة روبي

تعلِّم هذه الدورة أسس لغة روبي Ruby وإطار العمل Ruby on Rails، وكذلك معمارية MVC، وأسلوب البرمجة كائنية التوجه، مستخدمة أمثلة عملية لبناء تطبيقات ويب ولعبة بسيطة.

تحتوي الأكاديمية دورة أخرى لبناء تطبيقات الويب ولكن باستخدام لغة PHP، تسير على نفس النهج العملي من شرح الأدوات التي يحتاجها الدارس، وتطبيقات عملية شبيهة بما يحتاجه العملاء في السوق.

32.8.2 تطوير التطبيقات باستخدام لغة JavaScript

هذه الدورة تمامٌ على ما ذُكر في الكتاب لمن كان يطبق الأمثلة بلغة جافاسكربت، فسيتعلم الدارس كيفية بناء تطبيقات حقيقية باستخدامها وباستخدام بيئة Node.js ومكتبة React.js وغيرها، وفيها أمثلة عملية يبني فيها الدارس تطبيقات محادثة فورية وتطبيقات لسطح المكتب وتطبيقات للجوال أيضًا.

32.8.3 تطوير تطبيقات الجوال باستخدام تقنيات الويب

تشرح هذه الدورة كيفية بناء تطبيقات للجوال باستخدام تقنيات الويب، من أجل بناء التطبيق وتطويره مرة واحدة ثم تشغيله على نظامي تشغيل أندرويد و iOS معًا، باستخدام منصة كوردوفا Cordova، لمن يرغب في دخول سوق تطبيقات الهواتف المحمولة لكن لا يريد قصر نفسه على إحدى المنصتين الشهيرتين.

32.8.4 مصادر أخرى

لا شك أن الإنترنت مليء بمصادر تعلم البرمجة على اختلاف لغات الدراسة ولغات البرمجة المستخدمة فيها، وكذلك المفاهيم الرياضية والهندسية التي تُبنى عليها تلك التقنيات واللغات، غير أننا نريد تسليط الضوء على المحتوى العربي منها لتقليل فرص التقاء القارئ بالحاجز اللغوي المانع للتعلم، فنحيل القارئ مرة أخرى إلى أكاديمية حسوب حيث يجد فيها مئات الشروح والكتب المترجمة في صورة سلاسل من المقالات والدروس التفصيلية للغات البرمجة والأدوات والتطبيقات التي تُبنى بها، أو يعرض مقال المدخل الشامل لتعلم علوم الحاسوب ومقال الدليل الشامل لتعلم البرمجة باستخدام المصادر العربية مواد عربية شتى للغات البرمجة والخوارزميات وغيرها من المفاهيم الرياضية التي يحتاج إليها الدارس، وكذلك الأدوات والمكتبات التي يحتاج إليها في تلك المواد التعليمية.

أحدث إصدارات أكاديمية حسوب