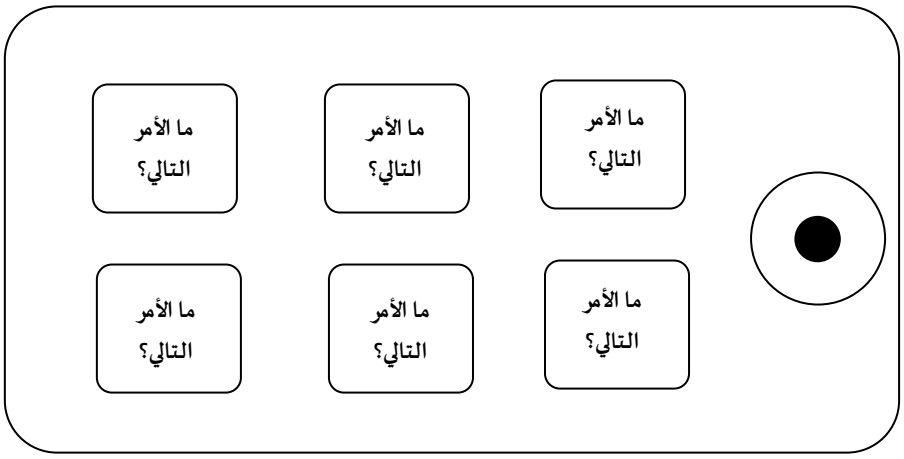
الفصل الأول

لماذا نتعلّم البرمجة؟

1 لماذا نتعلّم البرمجة؟

البرمجة نشاط ممتع، وعمليّة إبداعيّة مذهلة. تختلف الأسباب التي تدفع الناس لتعلُّم البرمجة، فمنهم من يتعلّمها لكسب الرزق، أو لتحليل البيانات المعقّدة، أو للتطوّع لحلِّ مشكلات الآخرين، أو حتّى للتسلية.

أنّ الجميع بحاجةٍ إلى تعلّم البرمجة حتّى إن لم ندرك الغاية منها في البداية، فنحن نعيش اليوم في عالم يعجُّ بالأجهزة الحاسوبية، مثل الحواسيب المحمولة والهواتف الذكية، وهي رهن إشارتنا، وكأنّ هذا العِتاد الصلب قد صُمِّمَ خصّيصًا ليقول لنا: "رغباتُك أوامر".



الشكل 1: المساعد الشخصي الرقمي

يزوِّد المبرمجون هذا العتاد بنظام تشغيل ومجموعة من التطبيقات، فنحصل بذلك على مساعدٍ شخصيٍّ رقميٍّ قادرٍ على مساعدتنا في تأدية مختلف المهام.

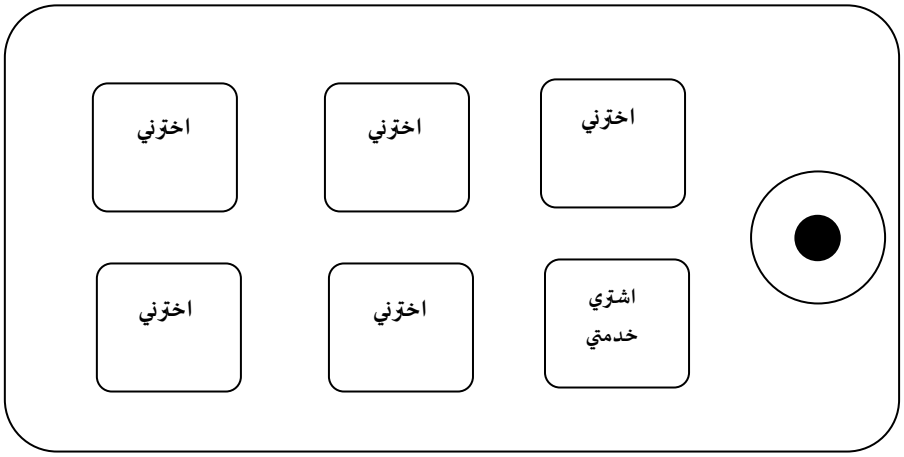
حواسيبُنا اليوم سريعة، ولها ذاكرة بحجم كبير، ويمكن أن تساعدنا في أداء مهامّنا الكثيرة والمتكرّرة إذا ما حدّثناها بلغتها، فهي قادرة على إنجاز المهام التي يعتبرها البشر مملّةٍ للغاية. على سبيل المثال، اقرأ أوّل مقطعين من هذا الفصل، واستخرج الكلمة الأكثر تكرارًا، مع ذِكر عدد مرّات تكرارها. صحيحٌ أنّك ستتمكَّن من قراءة وفهم هذه الكلمات بسرعة كبيرة، إلّا أنّ عدَّها مزعج لدماغك الذي لم يُخلق لحلِّ مثل هذه المسائل، خلافًا للحواسيب التي تُعَدُّ القراءة والفهم عمليّةً صعبة عليها، إلّا أنّها تستطيع بسهولة أن تعدّ مرّات تكرار كلمة ما، وتُظهر الكلمة الأكثر تكرارًا:

```
python words.py
Enter file: words.txt
to 16
```

أخبرَنا مساعدنا المخلص بكلّ سهولة أنّ الكلمة "to" تكرّرت ستّ عشرة مرّة في أوّل ثلاث فقرات في الملف word.txt. أتمنّى أن تجد في هذا المثال حافزًا لك لتعلّم لغة الحاسوب، فهي مناسبة لتأدية المهام الصعبة والمملّة بالنسبة للبشر، ممّا يوفّر لك الوقت والجهد الذي تحتاجه للتفكير والإبداع.

1.1 الإبداع والحافز

هذا الكتاب ليس موجَّهًا للمبرمجين المحترفين، مع العلم أن البرمجة الاحترافية تعود بالنفع على صاحبها، سواء ماليًّا أو شخصيًّا، فبناء برامج ذكيّة ومفيدة هو نشاط إبداعيّ بامتياز. يحتوي الحاسوب عادةً، أو لنقُل المساعد الشخصيّ الرقميّ (Personal Digital Assistant) PDA، على برامج متنوّعة صمّمها مبرمجون مختلفون، وتتنافس فيما بينها للحصول على انتباهنا واهتمامنا لتلبية احتياجاتنا وتوفير تجربة رائعة، وعادةً ما يتربَّح هؤلاء المبرمجون مباشرةً عند استخدامك لبرامجهم. وإن عرّفنا البرامج بأنّها نتاج إبداع عدّة مبرمجين، فيمكننا أن نتخيّل مساعدنا الشخصيّ كالآتي:

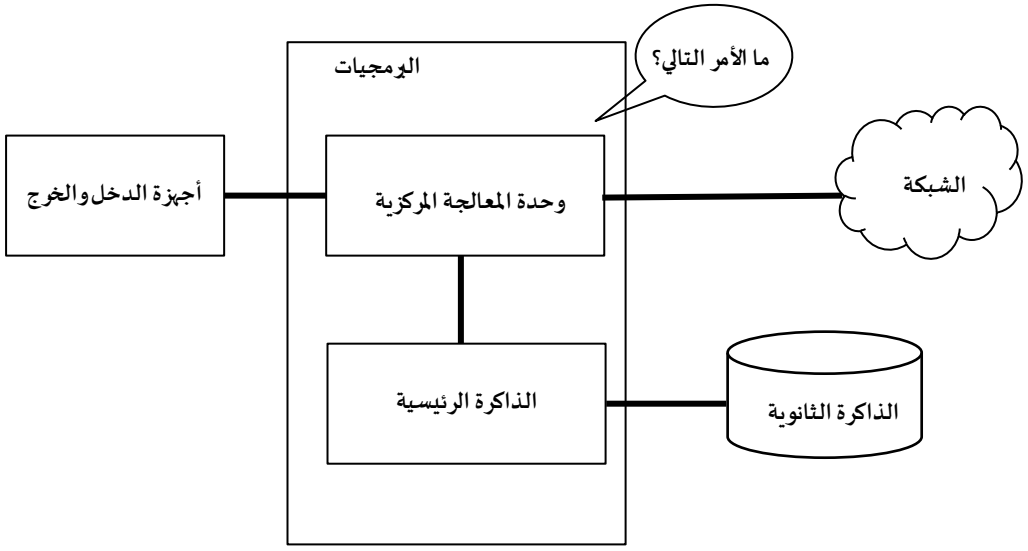


الشكل 2: كيف يخاطب المبرمجون المستخدم عبر تطبيقاتهم

فليكن دافعنا حاليًّا أن نتعامل مع البيانات والمعلومات التي نواجهها في حياتنا بفاعليّة أكبر، ولنترُك فكرة كسب المال أو إرضاء المستخدمين جانبًا، ففي البداية ستكون المبرمِج والمستخدِم في آنٍ واحد، وبمرور الوقت، ستكتسب مهارات أكثر، وستصبح البرمجةُ عمليّة ممتعة، حينها يمكنك تطوير البرامج للآخرين ومساعدتهم.

2.1 بنية الحاسوب

نحتاج أوّلًا إلى معرفة مكوّنات الحاسوب نفسه قبل البدء بتعلّم اللغة التي تسمح لنا بإعطاء الأوامر والتعليمات له، فإذا فكّكت هاتفًا أو حاسوبًا، ستجد فيه العناصر الآتية:



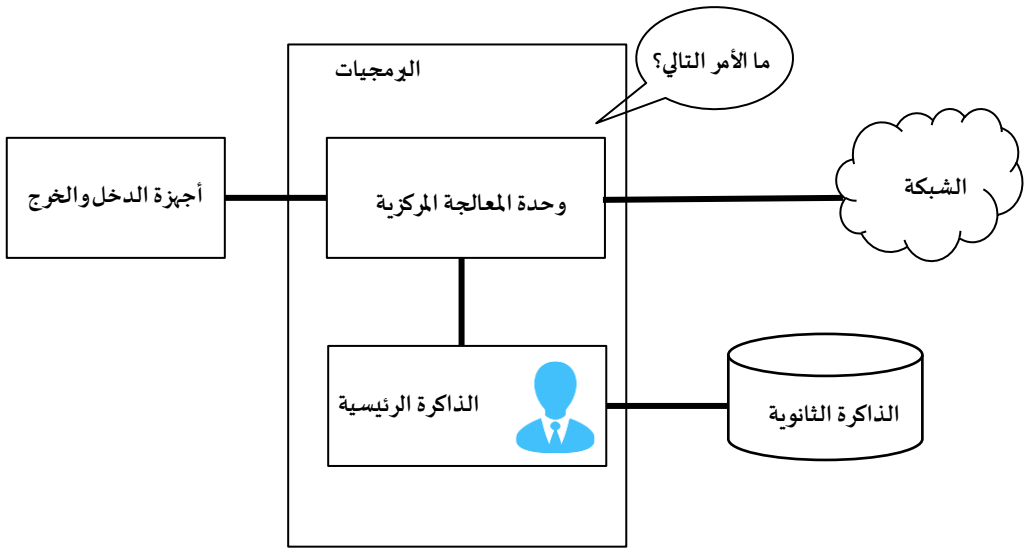
الشكل 3: بنية الحاسوب

فلنتناول كُلَّ عنصر باختصار:

- **وحدة المعالجة المركزيّة (أو المعالج) CPU:** هذا الجزء مسؤول خصّيصًا عن سؤال "ماذا أُنفِّذ؟"، وإذا كانت سرعة معالج الجهاز 3 جيجا هرتز، فسيتساءل ثلاثة مليارات مرّة في الثانية عن المهمّة التالية، لذلك يجب تعلُّم كيفيّة التواصل مع وحدة المعالجة المركزية بسرعة كبيرة.
- **الذاكرة الرئيسيّة:** تُستخدم لتخزين المعلومات التي يحتاج أن يصل إليها المعالج بسرعة، لذا فسرعتها تقارب سرعة المعالج، إلّا أنّ المعلومات تزول منها عند إطفاء تشغيل الحاسب.
- **الذاكرة الثانويّة:** تُستخدم أيضًا لتخزين المعلومات، إلّا أنّها أبطأ بكثير من الذاكرة الرئيسيّة، وتبرز فائدتها في قدرتها على تخزين المعلومات حتّى عند عدم تشغيل الحاسب. ومنها: أقراص التخزين أو الذاكرة الومضيّة (flash memory) الموجودة في وحدات تخزين متنقّلة USB ومشغّلات الموسيقى المحمولة.
- **أجهزة الإدخال والإخراج:** كالشاشة، ولوحة المفاتيح، والفأرة، والميكروفون، ومكبّر الصوت، ولوحة اللمس، وغيرها من الأجهزة التي تساعدنا في التفاعل مع الحاسب.

- تملك معظم الحواسيب حاليًّا اتّصالًا شبكيًّا لتبادل البيانات عبر الشبكة. يمكن اعتبار هذه الشبكة مكانًا بطيئًا جدًّا في تخزين وتبادل البيانات الفائضة، وبذلك قد تعمل الشبكة كذاكرة ثانويّة، ولكن بصورة أبطأ وأقلّ أمانًا من الذاكرة الثانويّة.

تلك كانت نبذة عن مختلف عناصر الحاسب التي ستساعدنا عند كتابتنا للبرامج في الفصول التالية، إلّا أنّنا لم نشغل بالنا بتفاصيل آليّة عمل هذه العناصر، وتركنا ذلك لمصمّمي الحواسيب، فوظيفتك كمبرمج تتمثّل في استخدام أجزاء الحاسوب المختلفة، والتنسيق بينها لحلّ المشكلة المطروحة، وتحليل البيانات الناتجة عن هذا الحلّ. غالبًا ما ستتعامل مع المعالج بإعطائه أوامر معيّنة لتأدية المهام التي تريدها، كاستخدام الذاكرة الرئيسيّة، أو الثانويّة، أو الشبكة، أو أجهزة الإدخال والإخراج.



الشكل 4 : المبرمج داخل الحاسوب!

أي أنّك أنت من سيأمر المعالج، ولكنّك قد تتضايق قليلًا إن قلّصنا حجمك إلى 5 مليمترات، ووضعناك داخل الحاسوب لإعطاء أوامر للمعالج بسرعة ثلاث مليارات مرّة في الثانية، لذا ستضطرّ إلى كتابة أوامرك مسبقًا، وتخزينها في ذاكرة الحاسوب؛ ليستدعيها المعالج في الوقت المناسب. تدعى هذه التعليمات المخزَّنة البرنامج، في حين تُسمَّى كتابة تلك التعليمات والحصول على تنفيذ صحيح لها البرمجة.

3.1 فهم البرمجة

سنحاول الأخذ بيدك نحو إتقان فنّ البرمجة في بقيّة فصول الكتاب، وستصير مبرمجًا في النهاية. قد لا تصبح مبرمجًا محترفًا، إلّا أنّك ستملك الفكر والمهارات اللازمة التي تؤهّلك لمعاينة المشكلة وتحليل بياناتها ومعلوماتها ثمّ إيجاد حلٍّ برمجيٍّ لها. وفي سبيل هذا ستحتاج إلى مهارتين أساسيّتين، وهما:

- أوّلًا، تعلُّم لغة البرمجة (بايثون) بمصطلحاتها وقواعدها، وهذا أشبه بتعلُّمك للغةٍ بشريّة، حيث تتعلّم تهجئة كلماتها أوّلًا حتّى تصيغ منها جملًا صحيحة.
- ثانيًا، اكتب قصة باستخدام البرمجة، فعندما تكتب قصّةً معيّنة تستخدم جملًا وعبارات لإيصال فكرتك إلى القارئ، فالقصّة مزيج من الفنّ والمهارة، وتتطوّر هذه المهارة بالتمرّن والحصول على آراء. ينطبق هذا على البرمجة، فالقصة هنا هي البرنامج، والفكرة هي المشكلة المطلوب حلُّها.

وجديرٌ بالذكر أنّه من السهل تعلّم لغةٍ برمجيّة أخرى، مثل C++ وجافا سكربت، بعد تعلّم لغةٍ واحدة مثل بايثون، فمهارة حلّ المسائل والمشكلات هي نفسها مهما اختلفت لغات البرمجة في تعليماتها وطرق استخدامها.

في حين يُعتبر تعلّم لغة بايثون نفسها عمليّة سهلة وسريعة، فستحتاج وقتًا أطول حتّى تستطيع كتابة برنامج قادر على حلّ مشكلة مستجدَّة. ستتعلّم معنا البرمجة بذات الطريقة التي تعلّمتَ بها الكتابة؛ فبدايةً سنقرأ ونشرح بعض البرامج، وبعدها ننتقل إلى كتابة برامجَ بسيطة، ثمّ إلى كتابة برامج أكثر تعقيدًا في النهاية. بعد فترة من التعلّم، ستصبح البرمجة عمليّة ممتعة وإبداعيّة، وستبدأ بتطوير طريقة تفكير خاصّة لتفكيك المعضلات التي تواجهك ومن ثمّ كتابةِ برامجَ لحلّها.

سنبدأ أوّلًا بتعليمات وبنية بايثون. تحلَّ بالصبر، وركِّز على الأمثلة في البداية كما لو أنّك تلميذ يتعلّم الكتابة والقراءة للمرّة الأولى.

4.1 مفردات بايثون وجملُها

على خلاف اللغات البشريّة، تتكوّن لغة بايثون من عددٍ قليلٍ جدًّا من المفردات، وتُسمّى هذه المفردات "الكلمات المحجوزة" لأنّ لها معنى واحدًا فقط بالنسبة لبايثون. أمّا لاحقًا عند كتابة برامجك، فستستطيع إنشاء مفرداتك الخاصّة، والتي تدعى "المتغيّرات"، ولك حريّة اختيار الأسماء لهذه المتغيّرات بشرط ألّا تكون من الكلمات المحجوزة.

لنشبِّه الأمر بالتعامل مع الكلاب المدرّبة، حيث نخاطبها بكلمات مثل: "اجلس" أو "ابقَ مكانك" أو "أحضر شيئًا ما"، ولكن إذا استخدمت كلمات غير محجوزة (أي أنّ الكلب غير مدرَّب عليها)، فسيرمقك بنظرة متعجِّبة حتّى تستخدم كلمة محجوزة. فإن قلتَ مثلًا: "أتمنى لو أنّ عددًا أكبر من الناس يمشون ليحافظوا على صحّتهم العامة"، فكل ما سيفهمه الكلب هو "المشي"، لأنّها كلمة محجوزة في لغة الكلاب، ويعتقد البعض أنّه ما من كلمات محجوزة بين البشر والقطط.

إليك بعضًا من الكلمات المحجوزة في لغة بايثون:

and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	
class	finally	is	return	
continue	for	lambda	try	
def	from	nonlocal	while	

وعلى عكس الكلاب، فبايثون مدرَّبة مسبقًا على هذه الكلمات، وعند كلِّ استخدام لكلمة "try" ستحاول بايثون تنفيذ التعليمات المطلوبة دون أن يتوقّف البرنامج أو يفشل. لا تشغل بالك في دلالات هذه الكلمات وسبل استخدامها، إذ سنتعلّم ذلك في وقتٍ لاحق. أمّا الآن، فلنبدأ بأمر بسيط، هذا الأمر يشبه أن تقول للكلب: "تحدَّث"، حيث بإمكاننا إخبار بايثون بما عليها أن تقوله بوضعه ضمن علامة اقتباس، مثل:

```
print ('Hello World!')
```

وبهذا نكون قد كوّنّا أول جملة صحيحة قواعديًّا في بايثون، والتي بدأت بالتابع print متبوعًا بنصّ من اختيارنا ضمن علامتي الاقتباس، مع مراعاة أنّ كلّ جمل التابع مكتوبة ضمن علامتي اقتباس، سواء المفردة '' أو المزدوجة ""، ويُفضِّل معظم الناس الإشارة المفردة، إلّا في حال كان المطلوب ظهورها نفسها في النص (كفاصلة عليا في اللغة الإنكليزيّة apostrophe)، حينئذٍ تُستخدم الإشارة المزدوجة.

5.1 مخاطبة بايثون

سنحتاج الآن إلى تعلّم كيفيّة مخاطبة بايثون بعد أن تعلّمنا كلمة وجملة بسيطة منها، ولكن قبل ذلك سنحتاج إلى تنصيب برنامج بايثون على الحاسوب، وتعلُّم طريقة تشغيله. تتضمّن هذه الخطوة تفاصيل عديدة، لذلك نقترح عليك زيارة موقع www.py4e.com، حيث وضَّحنا الإجراءات اللازمة للتنصيب والتشغيل على أنظمة ماكنتوش وويندوز مُرفقَةً بلقطات شاشة. أثناء ذلك ستصل إلى

مرحلة تستخدم فيها نافذة الأوامر command window أو terminal لتكتب كلمة "python"، ويبدأ مُفسِّر بايثون (interpreter) بالعمل، ويعرض لك ما يأتي:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

هذه الرموز >>> هي طريقة مُفسِّر بايثون ليسألك "ماذا تريدني أن أفعل الآن؟".

لنفترض أنّه لا علم لك حتّى بأبسط مفردات وتعليمات لغة بايثون، فلتحاول تجريب السطر الذي يستخدمه روّاد الفضاء للتواصل مع سكّان كوكبٍ مجهول عند هبوطهم على سطحه:

```
>>> I come in peace, please take me to your leader
File "<stdin>", line 1
  I come in peace, please take me to your leader
       ^
SyntaxError:  invalid syntax
>>>
```

لا يبدو هذا الوضع مبشِّرًا، وإن لم تتصرّف بسرعة، فقد يطعنك سكّان الكوكب برماحهم، ويثبّتونك على سيخٍ ليتناولوكَ على العشاء، ولكن لحسن الحظّ، أنت تملك نسخة من هذا الكتاب، وبإمكانك أن تفتح هذه الصفحة لتحاول مجدّدًا:

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

يبدو هذا أفضل بكثير، لذا ستحاول التحدّث معهم أكثر:

```
>>> print('You must be the legendary leader that comes from the sky')
You must be the legendary leader that comes from the sky
>>> print('We have been waiting for you for a long time')
We have been waiting for you for a long time
>>> print('Our legend says you will be very tasty with mustard')
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say



File "<stdin>", line 1
```

```
  print  'We will have a feast tonight unless you say
                                                    ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

كانت هذه المحادثة تسير على ما يرام حتّى اقترفت خطًا صغيرًا في بايثون، ممّا جعل سكّان الكوكب يرفعون رماحهم مرّةً أخرى. ضع في الحسبان أنّ لغة بايثون ليست ذكيّة كفاية للأسف، فعلى الرّغم من أنّها معقّدة تعقيدًا كبيرًا، إلّا أنّها غير مرنة عند ارتكاب الأخطاء القواعديّة (syntax errors)، فحديثك مع بايثون كحديثك مع نفسك، إنّما باستخدام قواعد لغويّة صارمة.

استخدامُك لبرنامجٍ كَتَبه سواك يشبه نوعًا ما أن تتحدّث مع مبرمجي هذا البرنامج، حيث تلعب بايثون دور وسيط بينكم، أي أنّ بايثون هي طريقة المبرمجين للتعبير عن مجرى المحادثة، وبعد بضعة فصول من هذا الكتاب ستصبح أحد أولئك المبرمجين، وستتواصل مع مستخدمي برامجك عن طريق بايثون.

أمّا الآن، فلعلّه من غير اللائق أن نترك سكّان كوكب بايثون دون أن نقول لهم "وداعًا":

```
>>> good-bye
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined
>>> if you don't mind, I need to leave
File "<stdin>", line 1
  if you don't mind, I need to leave
          ^
SyntaxError: invalid syntax
>>> quit()
```

كما تلاحظ، فالخطأ في أوّل محاولة يختلف عن ثاني خطأ: كلمة if من الكلمات المحجوزة، ممّا جعل المسكينة بايثون تعتقد أنّنا نحاول قول شيءٍ، ولكنّنا لم نُوفَّق في قولِه. وأخيرًا نجحنا في قول "وداعًا" لبايثون بكتابة quit() بعد شارة التلقين >>>، وبما أنّك ما كنت لتُخمِّن هذه الكلمة من تلقاء نفسك، تُعتبر الاستعانة بدليلٍ للغة (مثل هذا الكتاب) أمرًا مفيدًا.

6.1 المفسِّر والمترجِم

تُعدُّ بايثون لغة عالية المستوى، أي أنّها صُمِّمَت لتكون واضحة نسبيًّا للبشر، وللحواسيب في الوقت نفسه لتقرأها وتعالجها، ومن اللغات عالية المستوى أيضًا: C++, PHP, Basic, Ruby, Perl, JavaScript وغيرها الكثير.

لا تفهم وحدة المعالجة المركزية (CPU) أيًّا من هذه اللغات، بل تفهم فقط لغة واحدة ندعوها لغة الآلة، وهي لغة بسيطة جدًّا كالآتي:

```
00101000111010010010101010000001111
11100110000011101010010101101101
...
```

كلمة "بسيطة" هنا قد تكون خادعة، فهي بسيطة في بنيتها، إذ إنّها تتألّف من واحد وصفر، لكن ستجد أنّها صعبة ومعقّدة جدًّا مقارنةً مع بايثون حين تحاول أن تكتب برنامجًا بواسطتها، لذا يبرمج قلّة قليلة من المبرمجين بلغة الآلة ولأغراض محدَّدة. أنشأنا العديد من المترجِمات حتّى نستطيع البرمجة بلغات عالية المستوى كبايثون وجافا سكربت، وتحوّل هذه المترجمات تلك البرامج إلى لغة الآلة، حيث تنفِّذها وحدة المعالجة (CPU).

وباعتبار أنّ لغة الآلة مرتبطة بعِتاد الحاسوب الصلب، فما من طريقة لنقلها بين مختلف أنواع العتاد، بينما من الممكن نقل البرامج المكتوبة بلغة أكثر تعقيدًا باستخدام مُفسِّرٍ مختلف لآلة أخرى، أو إعادة جمع الشيفرة لإنشاء نسخةٍ من البرنامج بلغة الآلة من أجل آلة أخرى.

وتتمّ هذه العمليّة عبر المفسِّرات (interpreters) والمترجمات (compilers).

يقرأ المفسِّر البرنامج المصدريّ كما كتبه المبرمج، ويحلِّله، ويفسِّر تعليماته مباشرة للآلة. تَستخدم بايثون هذه التقنيّة، فعند تشغيلنا لبايثون بشكل تفاعليّ، نستطيع كتابة سطر برمجيٍّ لتعالجه بايثون مباشرةً، ثمّ تنتظر كتابة سطرٍ آخر.

قد نحتاج لتذكر قيم معيّنة لاستخدامها لاحقًا في البرنامج، فنختار أسماءً لتلك القيم لنتمكّن من حفظها واسترجاعها حين نحتاجها، وندعو هذه الأسماء "المتغيّرات".

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
```

```
>>>
```

في هذا المثال، طلبنا من بايثون تخزين القيمة 6، وحفظناها في متغيّر اسمه x لنتمكّن من استرجاعها لاحقًا، وتأكّدنا من أنّ بايثون قد تذكّرت ذلك عندما استخدمنا تابع الطباعة print. بعد ذلك، طلبنا من بايثون استعادة تلك القيمة لضربها بالعدد 7 لنحفظ الناتج في المتغيّر y، ثمّ طلبنا من بايثون أن تطبع قيمة y.

تعامِل بايثون هذه الأوامر كسلسلة مترابطة من التعليمات، حيث تسترجع قيمًا مخزَّنة من تعليمات سابقة مع أنّنا كتبنا كلّ سطر بمفرده، وبذلك نكون قد كتبنا برنامجنا الأوّل البسيط، وهو مكوّن من أربع عبارات مرتّبة ترتيبًا منطقيًّا.

يميل المفسِّر إلى نمط المحادثة التفاعليّة كما في المثال السابق، في حين يحتاج المترجم أن يستلم البرنامج كاملًا في ملفّ، حيث يحوّله إلى لغة الآلة، ثمّ يحفظ البرنامج الناتج في ملف لينفَّذ لاحقًا.

وهذه البرامج المكتوبة بلغة الآلة، والقابلة للتنفيذ، غالبًا ما تحمل اللاحقة ".exe" أو ".dll" على نظام ويندوز، والتي ترمز إلى "executable" و"dynamic link library"، بينما لا توجد لواحق مشابهة في أنظمة لينوكس وماكنتوش، وإن حاولت فتح ملف تنفيذ في محرّر النصوص، فسيظهر بشكلٍ غير مقروء كالآتي:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82^D^H
4^@^@^@\x90^]^@^@^@^@^@^@4^@
^@^G^@(^@$^@!^@^F^@^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\x
e0^@^@^@^E^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D
^H^S^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

ليس من السهل القراءة والكتابة بلغة الآلة، فمن حسن حظّنا أنّنا نملك المفسِّرات والمترجمات، ممّا يسمح لنا بالبرمجة بلغات عالية المستوى مثل بايثون وسي C.

لعلّك تتساءل الآن: ماذا عن مفسّر بايثون؟ وبأيّ لغة كُتب؟ وما الذي يحدث تمامًا عندما نكتب "Python"؟

مفسّر بايثون مكتوبٌ بلغة عالية المستوى تُدعى سي "C"، ويمكنك رؤية الشيفرة البرمجيّة المصدريّة له بالبحث عنه في موقع www.python.org. تُعدُّ لغة بايثون برنامجًا يترجَم بدورِه إلى لغة الآلة، لذلك

فإنّ تنصيبك لبايثون على حاسوبك يعني أنّك قد نقلت نسخة من برنامج بايثون المترجَم إلى لغة الآلة إلى نظامك، وفي ويندوز يكون ملف التنفيذ موجودًا غالبًا تحت الاسم التالي:

C:\Python35\python.exe

قد لا تحتاج هذه المعلومات لتبرمِج بلغة بايثون، لكن من المفيد الإجابة عن هذه الأسئلة المُلحّة منذ البداية.

7.1 كتابة برنامج

تُعتبر كتابة الأوامر في مفسّر بايثون طريقةً رائعة لاختبار بعض مميّزات بايثون، ولكن لا يوصى بها عند محاولة حلّ مشكلات معقّدة، لذلك سنستخدم محرِّر نصوص عند البرمجة لنكتب تعليمات بايثون في ملفّ يدعى نصًّا برمجيًّا (script)، ومن المتعارَف عليه أن تملك النصوص البرمجيّة في بايثون اللاحقة ".py".

لتنفيذ النصّ البرمجيّ، يجب أن تخبر مفسّر بايثون باسم الملف، حيث نكتب في نافذة الأوامر python hello.py ما يأتي:

```
$ cat hello.py
print ('Hello world! ')
$ python hello.py
Hello world!
```

تُعبِّر علامة الدولار $ عن إشارة نظام التشغيل، ويخبرنا الأمر cat hello.py أنّ الملف hello.py يحوي برنامجًا ذا سطر واحد يطبع نصًّا، ثمّ نستدعي مفسّر بايثون لنطلب منه قراءة الشيفرة المصدريّة من الملف hello.py بدلًا من أن نكتبه يدويًّا، وكما تلاحظ، فإنّ بايثون تعلَم أنّ عليها التوقّف عن التنفيذ عند الوصول إلى نهاية الملف الذي تقرأ منه برنامجك، لذلك لسنا مضطرّين لاستخدام quit() في نهاية البرنامج في الملف.

8.1 ما هو البرنامج؟

يُعرَّف البرنامج في بايثون باختصار على أنّه: سلسلة من تعليمات بلغة بايثون، وُضِعت لتنفيذ أمرٍ ما، وحتّى النصّ البرمجيّ hello.py يُعتبر برنامجًا ذا سطرٍ واحد، على الرغم من أنّه غير مفيد عمليًّا، فالبرنامج هو طريقة الحلّ التي نطرحُها للتغلّب على المشكلة التي تواجهنا.

لنتناول الآن مثالًا من أرض الواقع، ولنفترض أنّك تريد القيام ببحث اجتماعيٍّ يتناول منشورات فيسبوك، وتريد معرفة الكلمة الأكثر تكرارًا في مجموعة معيّنة من المنشورات. يمكنك طبعًا طباعة كلّ تلك المنشورات وفحصها يدويًّا لإيجاد الكلمة الأكثر شيوعًا، لكنّ هذه العمليّة ستستغرق وقتًا طويلًا، وقد لا تصل إلى الناتج الصحيح في النهاية، لكنّك باستخدام بايثون ستنجز هذه العمليّة بدقّة وسرعة، ممّا يسمح لك بقضاء وقتٍ ممتع في عطلة نهاية الأسبوع.

انظر مثلًا إلى النصّ أدناه، والذي يدور حول مهرّج وسيّارة، واستخرج منه الكلمة الأكثر تكرارًا وعدد مرّات تكرارها:

```
The clown ran after the car and the car ran into the tent and the tent fell down on the clown and the car
```

ثمّ تخيّل أنّك تستخرج الكلمة الأكثر تكرارًا من نصّ مؤلَّف من ملايين الأسطر. لعلّك اقتنعت الآن أنّه من الأسرع تعلّم لغة بايثون ثمّ كتابة برنامجٍ يُحصي لك عدد مرّات تكرار الكلمات، بدلًا من تنفيذ هذا يدويًّا. أمّا إذا أردت حلَّ هذه المعضلة الآن، فأنت محظوظ لأنّ هذا الكتاب يقدّم لك برنامجًا قد كُتِبَ واختُبِر، ونُقدّمه لك على طبق من ذهب لترى بعينك بعضًا من عظمة بايثون:

```
name = input ('Enter file: ')
handle = open (name, 'r')
counts = dict()

for line in handle:
        words = line.split()
        for word in words:
                counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
        if bigcount is None or count > bigcount:
                bigword = word
                bigcount = count

print(bigword, bigcount)
```

```
# Code: http://www.py4e.com/code3/words.py
```

تستطيع استخدام هذا البرنامج حتّى لو لم تكن تعلم لغة بايثون، ولكنّك ستحتاج أن تصبر حتّى الفصل العاشر من هذا الكتاب لفهم طريقة عمله، أمّا الآن فأنت مستخدِم للبرنامج وحسب، وبإمكانك أن تستخدمه وترى مدى ذكائه ومقدار الوقت الذي وفّره. كلّ ما عليك فعله هو أن تكتب هذا البرنامج في ملف، ثمّ أن تسمّيه words.py مثلًا، أو أن تنزِّل البرنامج الأصليّ من الموقع: http://www.py4e.com/code3/ ثم تشغِّله.

وهو مثال جيّد ليؤكّد على دور بايثون كوسيط بينك كمستخدِم، وبين المبرمِج. صار بوسعنا تبادل عدّة تعليمات مفيدة (أي برامج) باستخدام لغة شائعة يستطيع أيُّ شخصٍ استخدامها بمجرّد تنصيب بايثون على حاسوبه، فنحن لا نكلّم بايثون مباشرة، بل نتواصل فيما بيننا بواسطتها.

9.1 المكوّنات الأساسيّة للبرامج

في الفصول القادمة سنتعلّم أكثر حول مفردات بايثون، وبنيتها، وكيف ندمج بينها لبناء برامج مفيدة، لكن قبل ذلك، لنتعرّف إلى بعض المفاهيم الأساسيّة المستخدَمة لكتابة البرامج، وهي ليست خاصّة ببايثون، بل هي جزء من كلّ لغة برمجة، سواء كانت عالية المستوى أم لغة آلة، وهي:

الدخل: الحصول على البيانات من العالم الخارجيّ، كقراءة بيانات معيّنة من ملفّ ما، أو حسّاس كالمايكروفون، أو نظام تحديد المواقع GPS. سيكون دخلُ برامجنا الأولى عبر لوحة مفاتيح يتحكّم بها المستخدم.

الخرج: ويتمثّل في عرض نتائج البرنامج على شاشة ما، أو تخزينها في ملفّ، أو إرسالها إلى جهاز خرج كمكبّر الصوت لعرض موسيقى معيّنة أو قراءة نصّ.

التنفيذ التسلسليّ: تنفيذ الأوامر تِباعًا.

التنفيذ الشرطيّ: تفقُّد تحقُّق شروط معيّنة وتنفيذ أو تخطّي سلسلة من التعليمات بناءً على تلك الشروط.

التنفيذ التكراريّ: تنفيذ عدد معيّن من التعليمات بشكل متكرِّر، ويتضمّن هذا عادةً بعض التغييرات.

إعادة الاستخدام: كتابة عدد معيّن من التعليمات، ثمّ تسميتها باسم محدَّد لاستدعائها عند اللزوم خلال البرنامج.

لعلّك تجِد هذه المفاهيم ساذجة، كأنّنا نُعرّف المشي على أنّه عمليّة وضع قدم أمام الأخرى، لكن إليك السرّ: البرمجة فنّ ندمج فيه بين تلك العناصر الأساسيّة بطريقة تجعلها مفيدة وفعّالة.

10.1 ما الأخطاء التي يمكن أن نواجهها؟

ينبغي تحرِّي الدقّة عند التواصل مع بايثون كما رأينا سابقًا، فأصغر خطأ سيدفع بايثون للتوقّف عن تنفيذ البرنامج، لذلك يظنّ بعض المبرمجين المبتدئين أنّ هذا دليلٌ على أنّ بايثون تكرههم وتبغضهم، في حين أنّها تفضّل المبرمجين الآخرين عليهم، لذا فهي ترفض برامجهم المثاليّة، وتعتبرها غير صحيحة، متقصِّدةً إحراجهم.

```
>>> primt 'Hello world! '
File "<stdin>", line 1
   primt 'Hello world! '
                       ^
SyntaxError: invalid syntax
>>> primt (‘Hello world’)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined
>>> I hate you Python!
File "<stdin>", line 1
   I hate you Python!
         ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
File "<stdin>", line 1
  if you come out of there, I would teach you a lesson
               ^
SyntaxError: invalid syntax
>>>
```

لن يفيد الجدال مع بايثون، فهي مجرّد أداة بلا مشاعر، ومع ذلك هي مستعدَّة لخدمتك متى احتجتها، وإن بدت التحذيرات التي تظهِرها قاسية، فهي في الواقع تطلب مساعدتك، وكلّ ما في الأمر أنّها تفحّصت ما كتبتَه لها، إلّا أنّها لم تتمكّن من فهمه، إذ إنّها أشبه بحيوان أليف مدلَّل يحبّكَ بشدّة،

إلّا أنّه لا يفهم إلّا بضع كلمات مفتاحيّة، ويرمقك بنظرة بريئة <<< منتظرًا إيّاك أن تكتب شيئًا يفهمه، وعندما تقول بايثون: SyntaxError: Invalid syntax، فهي ببساطة تهزّ ذيلها وتقول: "يبدو أنّك أردت أن تقول شيئًا، إلّا أنّي لا أفهمه، ولكن أرجو أن تواصل التحدّث إليّ <<< ".

ستواجهك ثلاثة أخطاء رئيسيّة عند البرمجة باستخدام بايثون:

الأخطاء القواعديّة (Syntax errors): وهي أوّل الأخطاء التي سترتكبها، وأسهلها إصلاحًا، وتعني أنّك أخطأت في "القواعد اللغويّة" لبايثون، وستحاول بايثون الإشارة إلى السطر والحرف الذي لاحظت وجود الخطأ فيه. قد تخدعك بايثون بأن تُشير إلى وجود خطأ في موضع معيّن، في حين أنّ الخطأ الفعليّ يقع قبل ذلك، لذا ابدأ من حيث أشارت بايثون صعودًا حتّى تجد ذلك الخطأ.

الأخطاء المنطقيّة (Logic errors): وتحدث عندما لا توجد أخطاء قواعديّة، إنّما مشكلة في ترتيب بعض التعليمات أو الربط بينها، كأنْ يخبرك أحدهم أنّه فتح زجاجة الماء ليشرب منها، ثمّ وضعها في حقيبته وتابع سيره، وبعد ذلك أغلق الزجاجة.

أخطاء دلاليّة (Semantic errors): وتحدث عندما تكون قواعد برنامجك صحيحة ومرتّبة ترتيبًا منطقيًّا دون أن يؤدّي المطلوب منه، فلو أردت أن تعطي أحدهم التوجيهات للذهاب إلى أحد المطاعم فقلت له: "اتّجه يسارًا عندما تصل إلى التقاطع عند محطّة البنزين، ثمّ سِرْ ميلًا واحدًا وستجد مطعمًا ذا لونٍ أحمر إلى يسارك"، ثمّ بعد وهلة يتّصل بك هذا الشخص ليخبرك بأنّه وصل إلى مزرعة تحوي إسطبلًا وليس مطعمًا، فستسأله: "هل اتّجهت يمينًا أم يسارًا عند محطة البنزين؟"، ليجيبك: "لقد اتّبعت توجيهاتك بحذافيرها، بل إنّي كتبتها على ورقة حتّى لا أنساها"، ثمّ تفكّر للحظة وتحكّ رأسك ثمّ تقول: "أنا آسف يا صديقي، فقد كانت توجيهاتي صحيحة من حيث القواعد، إلّا أنّ خطًا في دلالاتها قد فاتني".

فما تقوم به بايثون في كلّ تلك الحالات هو تنفيذ ما تطلبه منها قدر استطاعتها.

11.1 التنقيح

عندما تعلن بايثون عن وجود خطأ، أو حتّى عندما تمنحك نتيجة مختلفة عمّا أردت، تبدأ عمليّة تنقيح البرنامج. والتنقيح هو عمليّة اكتشاف أسباب الأخطاء في برنامجك.

إليك أربعة أساليب لتستخدمها خاصّةً مع الأخطاء صعبة الملاحظة:

القراءة: افحص برنامجك واقرأه وتأكّد من أنّه مكتوب كما أردتَ تمامًا.

التجريب: جرِّب بعض التغييرات، ثمّ شغِّل البرنامج مرّة أخرى، وستبرز المشكلة إذا تأكّدت من أنّ كلّ شيء في مكانه الصحيح، لكن قد يستغرق الأمر وقتًا أحيانًا.

التريُّث: تمهَّل وفكّر واسأل نفسك حول نوع الخطأ الذي يواجهك: قواعديّ، أم خطأ أثناء التشغيل، أم دلاليّ؟ وما هي المعلومات التي ستحصل عليها من رسائل الأخطاء أو خرج البرنامج؟ وما التغيير الأخير الذي أجريته على البرنامج قبل ظهور المشكلة؟

التراجع: عند نقطة معيّنة، سيكون أفضل ما تستطيع فعله هو التراجع وإلغاء التغييرات التي أجريتها حتّى تحصل على برنامجٍ يعمل وبإمكانك فهمه، ومن ثمّ تستطيع أن تعيد بناءه ثانيةً.

يقع المبرمجون المبتدئون في خطأ الاعتماد على إحدى هذه الطرق دون غيرها، إلّا أنّ إيجاد خطأ صعب الملاحظة يتطلّب القراءة والتنفيذ والتريُّث، وأحيانًا التراجع، فإن لم تنجح أحدُها، جرِّب الأخرى.

فعلى سبيل المثال، قد تنجح طريقة القراءة إن كان الخطأ قواعديًّا، ولكنّها لن تفيد في حالة الأخطاء الدلاليّة، فالخطأ هنا موجود داخل رأسك، ولن تكتشفه إن كنت لا تفهم ما يفعله برنامجك حتّى ولو قرأت البرنامج مائة مرّة.

قد يساعد في حلِّ المشكلة إجراءُ تجاربٍ على البرنامج، إلّا أنّ ذلك غير ممكن دون قراءة وفهم برنامجك، وإلّا ستقع فيما نسمّيه في هذا الكتاب بنمط "البرمجة العشوائيّة"، وهو عمليّة تنفيذ تغييرات عشوائيّة على البرنامج حتّى ينفذ المطلوب، وهو نمط يستلزم وقتًا طويلًا بالطبع.

ستحتاج وقتًا للتفكير في كلّ الأحوال، فالتنقيح كالتجارب العمليّة، حيث تبدأ بوضع فرضيّة واحدة على الأقلّ حول ماهيّة المشكلة، وفي حال وجود احتمالين أو أكثر، تحاول وضع اختبار يستبعد أحد تلك الاحتمالات.

استرخِ، ثمّ حاول مرّة أخرى. تحدّث مع الآخرين، أو حتّى مع نفسك، وحاول شرح المشكلة لعلّك تجد الحلّ بمجرّد عرض المشكلة.

إن كان برنامجك يعجّ بالأخطاء، أو ضخمًا ومعقّدًا، فغالبًا لن تجدي معك أفضل تقنيّات التنقيح، وعندها يكون الحلّ الأفضل هو التراجع وتبسيط البرنامج لتحصل على برنامجٍ فعّال تستطيع فهمه.

لكن عادةً ما يستنكر المبرمجون المبتدئون عمليّة التراجع، حيث يعزّ عليهم حذف سطر من برنامجهم حتّى وإن كان خاطئًا. إن شعرتَ بذلك مستقبلًا، فبإمكانك نسخ برنامجك إلى ملفٍّ آخر قبل تجزئته، ثمّ تستطيع إعادة لصق كلّ جزء على حدة.

12.1 رحلة التعلّم

لا تقلق إن شعرت أنّ المفاهيم غير مترابطة جيّدًا أثناء قراءتك الأولى لهذا الكتاب، وتذكّر نفسكَ عندما بدأت تتعلّم التحدّث، حينما كنتَ تصدر أصواتًا طفوليّة ظريفة في البداية، ثمّ استغرقتَ حوالي ستّة أشهر لتعلّم تكوين جمل بسيطة من المفردات القليلة التي تعرفها، وبعد خمس أو ستّ سنوات انتقلتَ من الجمل إلى فقراتٍ كاملة، واحتجتَ بضع سنوات أخرى لتكتب قصّة قصيرة كاملة ومثيرة للاهتمام بمفردك.

نطمح لتعليمك بايثون خلال وقت أقصر بكثير، لذلك سنكثّف المحتوى في الفصول التالية. وكما في حال تعلُّمك لغةً جديدة، ستحتاج بعض الوقت لاستيعابها وفهمها قبل أن تعتاد عليها، ومن الطبيعيّ أن تشعر ببعض الارتباك أثناء طرحِنا لمواضيع تتعرّف عليها للمرّة الأولى، حيث نحاول شرح تفاصيل الصورة العامّة تدريجيًّا، إلّا أنّك غير مضطرّ لدراسة الكتاب بشكل منظّم، وتستطيع التقدّم بالقراءة، ومن ثمّ العودة إلى الفصول السابقة، فمجرّد تعرّضك لتلك المواضيع المتقدّمة يزيد من استيعابك للبرمجة بشكل كبير، حتّى وإن لم تتعمّق في تفاصيلها، وعند مراجعتك لما سبق وإعادة حلّ مسائله ستتيقّن من قدرتك على التعلُّم.

ستمرّ ببعض لحظات التجلّي التي يشعر بها فنّان يطرق بمطرقته وإزميله، ثمّ يتوقّف لينظر إلى جمال صنعه وعجيب نحته، وإن واجهت معضلة صعبة، فلا فائدة من التحديق بها طوال الليل، بل خذ استراحة أو غفوة، أو تناول وجبة خفيفة، واشرح لأحدهم مشكلتك (أو حتّى لحيوانك الأليف)، ثمّ بإمكانك العودة للدراسة مجدّدًا بذهن متيّقظ. نضمن لك أنّك حين تعود للفصول الأولى في هذا الكتاب بعد أن تتمكّن من البرمجة، ستجد أنّ الأمر كان بسيطًا وسهلًا، وكلّ ما كان يلزمك هو بعض الوقت لتستوعبه.

13.1 فهرس المصطلحات

- **الخطأ (Bug):** خطأ في البرنامج.
- **وحدة المعالجة المركزيّة (central processing unit):** قلب الحاسوب الذي يشغّل البرنامج الذي كتبناه، كما يُدعى "CPU" أو "المعالج".
- **الترجمة (compile):** ترجمة برنامج مكتوب بلغة عالية المستوى إلى لغة منخفضة المستوى دفعة واحدة تجهيزًا لتنفيذه لاحقًا.
- **لغة عالية المستوى (high-level language):** لغة برمجيّة، مثل بايثون، مصمَّمة لتكون سهلة القراءة والكتابة للبشر.

- **النمط التفاعليّ (interactive mode):** طريقة لاستخدام مفسِّر بايثون عبر كتابة الأوامر والتعليمات بعد إشارة التلقين.
- **التفسير(interpret):** تنفيذ برنامج مكتوب بلغة عالية المستوى بترجمة أسطره الواحد تلو الآخر.
- **لغة منخفضة المستوى (low-level language):** لغة برمجة مصمَّمة لتكون سهلة التنفيذ على الحاسوب، وتدعى أيضًا بلغة الآلة أو لغة التجميع.
- **شيفرة الآلة (machine code):** أقلّ مستوى من لغات البرمجة، وهي اللغة التي تنفِّذها وحدة المعالجة المركزيّة بشكل مباشر.
- **الذاكرة الرئيسيّة (main memory):** تخزِّن البرامج والمعلومات، وتفقد البيانات المخزَّنة فيها عند انقطاع الطاقة عنها.
- **التحليل (parse):** فحص برنامجٍ ما وتحليل بنيته القواعديّة.
- **قابليّة النقل (portability):** ميزة للبرنامج تسمح له بالعمل على عدّة أنواع من الحواسيب.
- **تابع print:** تعليمة تجعل مفسِّر بايثون يعرض قيمةً ما على الشاشة.
- **حلّ المشاكل (problem solving):** عمليّة تحليل المشكلةٍ وإيجاد حلّ لها والتعبير عنه.
- **البرنامج (program):** مجموعة من التعليمات تنفِّذ عمليّة حاسوبيّة.
- **موجِّه الأوامر (prompt):** عندما يعرض برنامج ما رسالة معيّنة منتظرًا المستخدم ليدخل قيمة إلى البرنامج.
- **الذاكرة الثانويّة (secondary memory):** تخزِّن البرامج والمعلومات وتحتفظ بها حتّى إن قُطعت عنها الكهرباء، ولكنّها أبطأ من الذاكرة الرئيسيّة، مثل: أقراص التخزين، والذواكر المتنقّلة USB.
- **دلالات (semantics):** معنى وهدف البرنامج.
- **خطأ دلاليّ (semantic error):** خطأ في البرنامج يجعله ينفّذ شيئًا مختلفًا عمّا أراده المبرمج.
- **البرنامج المصدريّ (source code):** برنامج مكتوب بلغة عالية المستوى.

14.1 تمارين

- **التمرين الأول:** ما وظيفة الذاكرة الثانويّة في الحاسوب؟
  - تنفيذ كلّ العمليّات الحاسوبيّة والمنطقيّة في برنامج ما.
  - استدعاء صفحات الويب عبر الإنترنت.
  - تخزين المعلومات لمدّة طويلة حتّى بعد انقطاع الكهرباء.
  - استلام الدخل من المستخدم.
- **التمرين الثاني:** ما تعريف البرنامج؟
- **التمرين الثالث:** ما الفرق بين المفسِّر والمترجم؟
- **التمرين الرابع:** أيٌّ ممّا يأتي يتضمّن شيفرة الآلة؟
  - مفسّر بايثون.
  - لوحة المفاتيح.
  - الملف المصدريّ لبايثون.
  - ملفّ نصّيّ.
- **التمرين الخامس:** ما الخطأ في البرنامج التالي:

```
>>> primt 'Hello world! '
File "<stdin>", line 1
  primt 'Hello world! '
                ^
SyntaxError: invalid syntax
>>>
```

- **التمرين السادس:** بعد تنفيذ السطر البرمجيّ التالي، أين يخزَّن المتغير "x" في الحاسب؟

```
X = 123
```

  - وحدة المعالجة المركزيّة.
  - الذاكرة الرئيسيّة.
  - الذاكرة الثانويّة.
  - أجهزة الدخل.
  - أجهزة الخرج.

- **التمرين السابع:** ما هي نتيجة البرنامج الآتي؟

```
x = 43
x = x + 1
print(x)
```

  - 43
  - 44
  - x+1
  - خطأ لأن x=x+1 غير صحيحة رياضيًا

- **التمرين الثامن:** اشرح كلًّا ممّا يأتي ذاكرًا مثالًا عن القدرة البشرية المكافئة:
  - وحدة المعالجة المركزيّة.
  - الذاكرة الرئيسيّة.
  - الذاكرة الثانويّة.
  - أجهزة الدخل.
  - أجهزة الخرج.

على سبيل المثال: ما هو المقابل البشريّ لوحدة المعالجة المركزيّة؟

- **التمرين التاسع:** كيف تصحّح الخطأ القواعديّ؟

الفصل الثاني

المتغيّرات والتعابير والتعليمات

2 المتغيّرات والتعابير والتعليمات

1.2 القيم و أنواع البيانات

تُعدّ القيمة (value) واحدة من المفاهيم الأساسيّة التي يتعامل معها البرنامج، فالحروف والأرقام هي بعض الأمثلة عن القيم، مثل 1 و2 و"Hello world". تنتمي هذه القيم إلى نوعَين مختلفين من أنواع البيانات، فالقيمة 2 هي عدد صحيح integer، أمّا "Hello world"، فهي سلسلة نصّيّة string، وسُمّيت بذلك لأنّها تحوي سلسلةً من المحارف. يمكن تمييز السلسلة النصّيّة من علامة الاقتباس المزدوجة " ".

تتعامل تعليمة الطباعة print مع كل من السلاسل النصّيّة والأعداد الصحيحة.

ليبدأ المفسِّر بالعمل، علينا كتابة الأمر python كما يلي:

```
python
>>> print(4)
4
```

إذا لم تكن متأكِّدًا من نوع القيمة، فالمفسِّر سيخبرك بذلك:

```
>>> type ('Hello, World! ')
<class 'str' >
>>> type(17)
<class 'int'>
```

للتوضيح، فالقيمة "Hello World" تنتمي إلى نوع السلسة النصّيّة، ويُعبَّر عنها برمز str. وبالمثل، تنتمي القيمة 17 إلى الأعداد الصحيحة int، أمّا الأرقام التي تحوي فاصلة عشريّة، فهي تنتمي إلى نوع الأعداد ذات الفاصلة العشريّة float، وتأتي التسمية من طريقة تمثيل هذه الأعداد، والتي تدعى floating point.

```
>>> type(3.2)
<class 'float'>
```

أمّا القيم مثل "17" و"3.2"، فهي تبدو كأرقام، ولكن بسبب وجود علامة الاقتباس تُعتبر سلاسل نصّيّة.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

قد يلجأ البعض عند كتابة عدد صحيح كبير إلى وضع فواصل بين خانة العشرات والمئات والألوف... إلخ، مثل 1,000,000، لكنّ لغة بايثون تعتبر هذا تمثيلًا خاطئًا للعدد الصحيح، ولكن في نفس الوقت ستتعامل معه في تعليمة الطباعة كما يلي:

```
>>> print(1,000,000)
1 0 0
```

نتيجةٌ غير متوقَّعة، فلغة بايثون تفسِّر 1,000,000 على أنّها مجموعة أعداد صحيحة مستقلّة تفصل بينها فاصلة، فيظهر على الخرج الأعداد المبيّنة وبينها فراغات. تُمثّل هذه الحالة ما يُعرف بالخطأ الدلاليّ (semantic error)، حيث تنفَّذ الشيفرة البرمجيّة دون رسالة خطأ، لكنّها لا تعطي الخرج أو النتيجة الصحيحة المتوقَّعة.

2.2 المتغيّرات

تُعدّ القدرة على التلاعب بالمتغيّرات أحد أقوى ميّزات لغة البرمجة، والمتغيّر هو اسم يشير إلى قيم.

تنشئ تعليمة الإسناد (assignment statement) متغيّرات جديدة، وتعطيها قيم:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

نلاحظ في المثال ثلاث عمليّات إسناد: الأولى إسناد سلسلة نصّيّة إلى متغيّر جديد يُسمّى message، أمّا الثانية، فإسناد العدد الصحيح 17 للمتغيّر n، أمّا الثالثة، فإسناد القيمة التقريبيّة لـ π للمتغيّر pi.

لإظهار قيمة المتغيّر بإمكانك استخدام تعليمة الطباعة print:

```
>>> print(n)
17
```

```
>>> print(pi)
3.141592653589793
```

نوعُ المتغيّر هو نوع القيمة التي يمثّلها:

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

3.2 أسماء المتغيّرات والكلمات المفتاحيّة

يختار المبرمجون عادة أسماء المتغيّرات (variables) بحيث تكون ذات معنى وتعكس الهدف من استخدامها.

يمكن لأسماء المتغيّرات أن تكون ذات أطوال مختلفة، وقد تتضمّن كلًّا من الأحرف والأرقام، لكن لا يمكن أن يبدأ اسم المتغيّر برقم، كما يُسمح باستخدام الحروف الكبيرة، ولكن من الأفضل أن يبدأ اسم المتغيّر بحروف صغيرة (سنرى السبب لاحقًا).

يُسمح بوجود رمز الشرطة السفليّة _ في اسم المتغيّر، وتُستخدم غالبًا في أسماء المتغيّرات التي تحوي العديد من الكلمات، مثل: my_name، أو air_speed_of_unladen_swallow.

وقد تبدأ أسماء المتغيّرات بالشرطة السفليّة _، لكن بشكل عامّ نتجنّب ذلك إن لَم نكُن نكتب شيفرة لمكتبة برمجيّة قد يستخدمها الآخرون.

إذا اخترت اسمًا غير جائز لمتغيّر، فستتلقّى رسالة خطأ قواعديّ (syntax error).

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

اسمُ المتغيّر 76trobones غير جائز لأنّه يبدأ برقم، واسم المتغيّر more@ غير جائز لأنّه يحتوي على

رمز @ غير الجائز، لكن ما المشكلة في اسم المتغيّر class؟

كلمة class هي إحدى الكلمات المفتاحيّة (keywords) في لغة بايثون، فالمفسِّر يستخدم الكلمات المفتاحيّة للتعرّف على بنية البرنامج، وبالتالي لا يمكن استخدامها كأسماء متغيّرات. تخزّن لغة بايثون 35 كلمة مفتاحيّة:

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	async
def	for	lambda	return	await

يُفضَّل أن تحتفظ بهذه القائمة أعلاه، وإذا أعطى المفسّر تنبيهًا حول أحد أسماء المتغيّرات ولم تعرف السبب، فانظر إن كانت إحداها في تلك القائمة.

4.2 التعليمات

التعليمة هي جزءٌ من الشيفرة البرمجيّة يستطيع مفسّر بايثون تنفيذها.

رأينا سابقًا نوعين من التعليمات: تعليمة print بوصفِها تعليمة تعبير (expression statement)، وتعليمة الإسناد (assignment).

عندما تُكتب تعليمة في الوضع التفاعليّ (interactive mode)، يُنفِّذها المفسّر ويعرض النتيجة كما لو أنّ هناك تعليمة واحدة فقط.

بينما يتضمّن النصّ البرمجيّ عادةً سلسلة من التعليمات، فتظهر النتائج واحدة تلو الأخرى أثناء تنفيذ التعليمات. ففي النصّ البرمجيّ التالي مثلًا:

```
print(1)
x = 2
print(x)
```

سيظهر الخرج بالترتيب:

```
1

2
```

ولا تُظهِر تعليمة الإسناد أيّ خرج.

5.2 العوامل والمعاملات

العوامل (operators) رموزٌ خاصّة بالعمليات الحسابيّة، مثل الجمع والضرب.

تُسمّى القيم التي تُطبَّق العوامل عليها بالمعاملات (operands).

العوامل `+ - * / **` تُمثّل الجمع والطرح والضرب والقسمة والرفع إلى قوّة، كما في الأمثلة التالية:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

وقد حصل تغييرٌ في عامل القسمة بين نسخة Python2.x ونسخة Python3.x، ففي Python3.x، تحوي نتيجة عمليّة القسمة التالية فاصلةً عشريّة:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

أمّا العامل ذاته في Python2.0 فيقسم العددين الصحيحين ويُقرّب النتيجة لعدد صحيح فقط:

```
>>> minute = 59
>>> minute/60
0
```

استخدِم عامل القسمة ذي التقريب للأدنى (//) للحصول على نفس الإجابة في Python3.0.

```
>>> minute = 59
>>> minute//60
0
```

تعمل توابع قسمة العدد الصحيح في Python3.0 كما لو أنّك تستخدم آلة حاسبة لحساب ناتج القسمة.

6.2 التعابير

يُعَدّ التعبير مزيجًا من القيم والمتغيّرات والعوامل، وتُعَدُّ القيمة بمفردها تعبيرًا، وينطبق الأمر ذاته على المتغيّر. وفي المثال التالي، تُعدّ جميع التعابير جائزة (على فرض أنّ المتغير x قد أُسنِد إلى قيمة):

```
17
x
x + 17
```

إذا كتبت تعبيرًا في الوضع التفاعليّ (interactive mode)، فسيفسِّره المفسّر ويعرض النتيجة:

```
>>> 1 + 1
2
```

إلّا أنّ التعبير لوحده لا يقوم بشيء في النصّ البرمجيّ، وهذا أحد الأمور الشائعة التي تحيّر المبتدئين.

التمرين 1: اكتب التعليمات التالية في مفسِّر بايثون لترى ما تقوم به:

```
5
x = 5
x + 1
```

7.2 تراتبيّة العمليّات

عندما يظهر أكثر من عامل في التعبير، تعتمد تراتبيّة الحلّ على قواعد الأسبقيّة، ففي العمليّات الرياضيّة، تتبع بايثون الاصطلاحات الرياضيّة المعروفة.

يُشكّل الاختصار **PEMDAS** طريقة مفيدة لتذكّر القواعد التالية:

- الأقواس (Parentheses) لها الأسبقيّة، ويمكن استخدامها للحلّ بالترتيب الذي تريده. بما أنّ التعبيرات بين قوسين تُحَلّ أوّلًا، فالعمليّة الرياضيّة (1-3)*2 تعطي 4، والعملية الرياضيّة (2-5)**(1+1) تعطي 8 .

ويمكن استخدام الأقواس لتسهيل قراءة التعبير، كما في المثال (minute*100)/60، حتّى لو لم تغيّر النتيجة.

- يمثّل عامل الرفع إلى قوّة (Exponentiation) الأسبقيّة التالية بعد الأقواس، فالعمليّة الرياضيّة 1+1**2 ناتجها 3، وليس 4، و3**1*3 ناتجها 3 وليس 27.

- أمّا الضرب (Multiplication) والقسمة (Division)، فلهما نفس الأسبقيّة، والتي تسبق عمليّتا الجمع (Addition) والطرح (Subtraction) اللتان لهما نفس الأولويّة، لذلك 1-3*2 ناتجها 5، وليس 4، و6+4/2 ناتجها 8، وليس 5.

تُقيَّم العوامل التي تملك نفس الأولويّة من اليسار إلى اليمين، لذلك ناتج التعبير 1-3-5 يساوي 1، وليس 3، لأنّ 3-5 تحدث أوّلًا، ثمّ 1 مطروح من 2. لتجنّب الشكّ في الأولويّة، ضع أقواسًا في تعبيراتك دائمًا للتأكّد من أنّ إجراء الحسابات يحدث بالترتيب الذي تريدهُ.

8.2 عامل باقي القسمة

يعمل مع الأعداد الصحيحة، ويعطي باقي قسمة المعامل الأوّل على الثاني.

يُمثَّل في لغة بايثون عامل باقي القسمة بإشارة %، والقواعد هي نفسها بالنسبة لأيّ عامل آخر.

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

أي 7 مقسومة على 3 يساوي 2 مع الباقي 1.

على عكس ما يبدو، يملك عامل باقي القسمة فائدة جمّة، إذ يمكنك مثلًا التحقّق من قابلية قسمة أحد الأرقام على رقم آخر، فإذا كانت X%Y (أي باقي قسمة X على Y) تساوي الصفر، يكون الرقم X قابلًا للقسمة على Y.

كما يمكن استخراج رقم أو عدّة أرقام موجودة في أقصى يمين عدد، فمثلًا العمليّة X%10 تعطي الرقم الموجود أقصى اليمين من X في الأساس 10 (على سبيل المثال 115%10 تعطي 5) وكذلك تعطي من أجل 100% قيمة آخر رقمين (15 في حالة العدد 115).

9.2 العمليّات على السلاسل النصّيّة

يعمل العامل + مع السلاسل النصّيّة، لكنّه لا يعني الجمع بمعناه الرياضيّ، بل يجري تجميع، والذي يعني ضمّ السلاسل معًا، مثل:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

كما يعمل العامل * مع السلاسل النصّيّة عبر تكرار محتوى السلسلة عددًا صحيحًا من المرّات، مثل:

```
>>> first = 'Test '
>>> second = 3
>>> print(first * second)
Test Test Test
```

10.2 إدخال البيانات من المستخدم

قد نحتاج أحيانًا إلى أخذ قيمة متغيّرٍ ما من المستخدم عبر لوحة المفاتيح، وتوفّر لغة بايثون تابعًا جاهزًا يدعى input، والذي يقبل قيمًا من لوحة المفاتيح. (كان هذا التابع يدعى raw_input في Python 2.0)

يتوقّف البرنامج وينتظر المستخدم لكتابة شيء ما عندما يُستدعى هذا التابع، وعندما يضغط المستخدم مفتاح الإدخال (enter) أو العودة (return)، يستأنف البرنامج، ويعيد التابع input ما كتبه المستخدم بشكل سلسلة نصّيّة.

```
>>> inp = input()
Some silly stuff
>>> print(inp)
Some silly stuff
```

من الأفضل طباعةُ عبارةٍ تخبر المستخدم بما عليه إدخاله قبل الحصول على مدخلات منه.

يمكنك تمرير سلسلة نصّيّة إلى التابع input كي تُعرض للمستخدم قبل التوقّف المؤقّت بانتظار الدخل.

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

تضيف السلسلة \n في نهاية موجّه الأوامر سطرًا جديدًا، وهي رمزٌ خاصّ يتسبّب في فصل الأسطر. لهذا السبب يظهر دخل المستخدم أسفل العبارة، وليس على نفس السطر.

إذا كنت تتوقّع أن يكتب المستخدم عدد صحيح، فيمكنك تحويل القيمة المعادة لعدد صحيح int باستخدام التابع int():

```
>>> prompt = 'What... is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What... is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

لكن إذا كتب المستخدم شيئًا آخر غير سلسلة نصّيّة من الأرقام، فستظهر رسالة خطأ:

```
>>> speed = input(prompt)
What... is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

سنرى كيفيّة التعامل مع الأخطاء من هذا النوع لاحقًا.

11.2 التعليقات

تزداد صعوبة قراءة البرامج مع ازدياد حجمها وتعقيدها، وغالبًا ما يكون من الصعب قراءة جزء من شيفرة برمجيّة ومعرفة ما يفعله البرنامج ولماذا، لذلك يفضَّل إضافة ملاحظات إلى برامجك لتشرح

فيها باللغة الطبيعيّة ما يفعله هذا البرنامج. تسمّى هذه الملاحظات التعليقات (comments)، وفي لغة بايثون يبدأ التعليق بالرمز #:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

في هذه الحالة، سيظهر تعليق على سطر بمفرده، كما يمكنك وضع التعليقات في نهاية السطر:

```
percentage = (minute * 100) / 60 # percentage of an hour
```

يُتَجاهَل كلّ شيء من الرمز # إلى نهاية السطر، ولا يكون له أيّ تأثير على البرنامج.

تبرز فائدة التعليقات عندما توثّق ميّزات غير واضحة للشيفرة البرمجيّة، وبما أنّه من المنطقيّ افتراض أنّ القارئ قادر على معرفة ما تفعله الشيفرة، يُعدّ التعليق أكثر فائدة عندما يشرح السبب.

ما من داعٍ لهذا التعليق غير مفيد:

```
v = 5 # assign 5 to v
```

أمّا هذا التعليق، فيحوي معلومة مفيدة غير موجودة في الشيفرة:

```
v = 5 # velocity in meters/second.
```

تقلّل أسماء المتغيّرات الجيّدة من الحاجة إلى التعليقات، لكنّ الأسماء الطويلة قد تعطي تعابير معقّدة تصعب قراءتها، لذلك تجِب المحافظة على التوازن بينهما.

12.2 اختيار أسماء متغيّرات سهلة التذكّر

باتّباعك لقواعد تسمية المتغيّرات البسيطة، وتجنّب الكلمات المحجوزة، ستجد أمامك العديد من الخيارات لتسمية المتغيّرات الخاصّة بك.

في البداية قد يكون الخيار مربكًا حين تقرأ برنامجًا، وحين تكتب برنامجًا بنفسك. فعلى سبيل المثال، البرامج الثلاثة التالية متطابقة من حيث الفعل، ولكنّها مختلفة جدًا عندما تقرؤها وتحاول فهمها:

```
a = 35.0
b =  12.50
c = a * b
print(c)
```

```
hours  = 35.0
rate   = 12.50
pay = hours * rate
print(pay)


x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

يَعتبِر مفسّرُ لغة البايثون البرامج الثلاثة متطابقة تمامًا، لكنّ الإنسان يرى ويفهم هذه البرامج بطريقة مختلفة للغاية، إذ سيفهم الإنسان الهدف من البرنامج الثاني على الفور لأنّ المبرمج يملك أسماء متغيّرات مختارة تعكس القيَم التي ستُخَزَّن.

تُدعى أسماء المتغيّرات المختارة بحكمة "أسماء المتغيّرات سهلة التذكّر" (mnemonic variable names). الكلمة mnemonic اختصار لـ memory aid، أي "مساعد للذاكرة"، ونستخدم هذا النوع من المتغيّرات للمساعدة في تذكّر سبب إنشائنا للمتغيّر في الأصل. وعلى الرغم من أنّ كلّ ذلك يبدو جيّدًا ومفيدًا، إلا أنّه قد يشكّل عائقًا أمام المبرمجين المبتدئين في القدرة على تحليل وفهم نوع الشيفرة، وذلك لأنّ المبرمجين المبتدئين لن يكونوا قد حفظوا الكلمات المحجوزة بعد (توجد 33 منها فقط). وقد تبدو المتغيّرات ذات الأسماء الوصفيّة وكأنّها جزءٌ من اللغة في بعض الأحيان، لا أسماءً مختارة بعناية وحسب.

انظر إلى النموذج التالي الذي يمثّل شيفرة برمجيّة بلغة البايثون، ويتعامل مع بيانات ضمن حلقة تكراريّة. سوف نناقش موضوع الحلقات قريبًا، لكن فلنحاول الآن اكتشاف معنى هذه الحلقة:

```
for word in words:
        print(word)
```

ماذا يحصل هنا؟ وأيٌّ من تلك الرموز (for وword وin وغيرها... إلخ) يُمثّل كلمات محجوزة؟ وأيٌّ منها يُمثِّل أسماء متغيّرات؟ وهل تفهم بايثون مفهوم الكلمات أساسًا؟

يواجه المبرمجون المبتدئون صعوبة في تمييز أجزاء الشيفرة التي اختارها المبرمج.

تشابه الشيفرة البرمجيّة التالية الشيفرة التي ذكرناها أعلاه:

```
for slice in pizza:
        print(slice)
```

من الأسهل للمبرمج المبتدئ النظر إلى هذه الشيفرة البرمجيّة ومعرفة أيّة أجزاء منها تُمثّل كلمات محجوزة محدّدة من قبل لغة بايثون، وأيُّ الأجزاء هي أسماء متغيّرات اختارها المبرمج.

من الواضح جدًّا أنّ بايثون غير قادرٍ على فهم الكلمتين pizza و slices، أو حقيقة أنّ البيتزا تتكوّن من مجموعة واحدة أو أكثر من الشرائح (slices)، لكن إذا كان برنامجنا متعلِّقًا بقراءة البيانات والبحث عن الكلمات في البيانات، فمن الصعب تذكّر أسماء متغيّرات مثل Pizza وSlices، واختيار أسماء متغيّراتك على هذا النحو سيسبِّب تشتُّتًا عن معنى البرنامج.

بعد فترة قصيرة سوف تتعلّم أكثر عن الكلمات المحجوزة الشائعة، وستتذكّرها تلقائيًّا.

أجزاء الشيفرة البرمجيّة التي تحدّدها لغة بايثون هي (for ، in ، print ، :)، أمّا المتغيّرات التي اختارها المبرمج فهي word وwords.

تتعامل العديد من برامج تحرير النصوص مع قواعد لغة بايثون، لذا تلوِّن تلك الكلمات بألوان مختلفة لإعطاء دليل يميّز المتغيّرات عن الكلمات المحجوزة.

ستبدأ بعد فترة بقراءة شيفرات برمجيّة مكتوبة بلغة بايثون، وستُميِّز بسرعة بين الكلمات المحجوزة والمتغيِّرات.

13.2 التنقيح

في هذه المرحلة ستواجه الخطأ القواعديّ غالبًا بسبب تسمية متغيّر غير مسموح، مثل class وyield، والتي تمثّل كلمات مفتاحيّة، أو odd~job وU$، والتي تحوي رموزًا غير جائزة.

إذا وَضعت فراغًا في اسم متغيّر، فستعتقد لغة بايثون أنّهما معاملان دون عامل:

```
>>> bad name = 5
SyntaxError: invalid syntax


>>> month = 09
  File "<stdin>", line 1
    month = 09
                ^
```

```
SyntaxError: invalid token
```

عندما تواجهك الأخطاء القواعديّة، فرسالة الخطأ لا تساعد كثيرًا. أكثر الرسائل شيوعًا هي "أخطاء قواعديّة لقواعد غير صالحة" (SyntaxError: invalid syntax)، و"أخطاء قواعديّة لرموز غير صالحة" (SyntaxError: invalid token).

الخطأ الذي يرجّح أن ترتكبه أثناء التشغيل (runtime error) هو عند محاولتك استخدام متغيّرٍ قبل إسناده إلى قيمة.

ويحدث إذا كتبتَ اسم المتغيّر بشكل خاطئ:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

مع مراعاة أنّ أسماء المتغيّرات حسّاسة لحالة الأحرف، فعلى سبيل المثال LaTeX غير latex.

السبب الأكثر احتمالًا لوقوعك في خطأ دلاليّ (semantic error) في هذه المرحلة هو ترتيب العمليّات. فمثلًا لإيجاد π1/2، قد تكتب:

```
>>> 1.0 / 2.0 * pi
```

لكنّ القسمة ستحدث أوّلًا، لذلك تحصل على π/2، وهي مختلفة عمّا كنت تقصده.

لا توجد طريقة لمعرفة ما كنت تقصد كتابته في لغة بايثون، لذلك لن تحصل على رسالة خطأ في هذه الحال، بل ستحصل على إجابة خاطئة.

14.2 فهرس المصطلحات

- **الإسناد (assignment):** التعليمة التي تسند قيمة لمتغيّر.
- **التجميع (concatenate):** ضمّ معاملين معًا.
- **التعليق (comment):** معلوماتٌ توضيحيّة في البرنامج موجهة لأيّ مبرمج أو قارئ للشيفرة البرمجيّة بحيث لا تؤثّر على تنفيذ الشيفرة.
- **تقييم (evaluate):** لتبسيط التعبير عبر إجراء العمليات بالترتيب للحصول على قيمة واحدة.

- **التعبير (expression):** مجموعة من المتغيرات والعوامل والقيم التي تمثل قيمة نتيجة واحدة.
- **الفاصلة العشريّة (floating point):** نوع بيانات يمثّل الأرقام ذات الفاصلة العشريّة.
- **العدد الصحيح (integer):** نوع بيانات يمثّل الأعداد الصحيحة.
- **الكلمة المفتاحيّة (keyword):** كلمة محجوزة مستخدَمة من المترجم للتعامل مع البرنامج (لا يمكنك استخدام كلمات مفتاحيّة مثل if وdef وwhile كأسماء متغيّرات).
- **سهل التذكّر (mnemonic):** مساعدة الذاكرة، وغالبًا نستخدم أسماء متغيّرات سهلة التذكّر لتساعدنا في تذكّر ما خُزِّن في المتغيّرات.
- **عامل باقي القسمة (modulus operator):** عامل يشار إليه بالإشارة %، يعمل مع الأعداد الصحيحة، وينتج الباقي عندما يكون الرقم مقسّم على آخر.
- **المعامل (operand):** أحد القيم التي يعمل عليها العامل.
- **العامل (operator):** رمز خاصّ، ويمثّل عمليّة حسابيّة بسيطة، كالجمع والضرب أو تجميع سلاسل نصّيّة.
- **قواعد الأولويّة (rules of precedence):** مجموعة من القواعد التي تحكم ترتيب التعابير التي تشمل العديد من العوامل والمعاملات.
- **التعليمة (statement):** جزء من الشيفرة البرمجيّة التي تمثّل أمرًا أو إجراءً. التعليمات التي رأيناها حتّى الآن هي تعليمة الإسناد وتعليمة طباعة.
- **السلسة النصّيّة (string):** نوع بيانات يمثّل سلسلة من المحارف.
- **نوع البيانات (type):** تصنيف للقيم التي رأيناها سابقًا، وهي int وfloat وstring.
- **القيمة (value):** إحدى الوحدات الأساسيّة للبيانات، مثل رقم أو سلسلة نصّيّة ، التي تتغيّر في البرنامج.
- **المتغيّر (variable):** اسم يشير إلى قيمة.

15.2 تمارين

- **التمرين الثاني:** اكتب برنامجًا يستخدم دخلًا input لتوجيه أمر للمستخدم لكتابة اسمه والترحيب به كما يلي:

```
Enter your name: Chuck
Hello Chuck
```

- **التمرين الثالث:** اكتب برنامجًا يسمح للمستخدم بإدخال ساعات ومعدّل الأجر لحساب الراتب الإجماليّ كما يلي:

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

لا داعي للقلق في حال تجاوزت قيمة الراتب pay رقمين بعد الفاصلة العشريّة.

بإمكانك باستخدام تابع التقريب المضمَّن في لغة بايثون لتقريب الراتب الناتج إلى منزلتين عشريّتين.

- **التمرين الرابع:** على فرض أنّنا ننفّذ تعليمات الإسناد التالية:

```
width = 17
height = 12.0
```

اكتب قيمة التعبير ونوع بيانات والناتج لكلّ من التعابير التالية:

  - width//2
  - width/2.0
  - height/3
  - 1 + 2 * 5

استخدم مفسّر بايثون للتحقّق من إجابتك.

- **التمرين الخامس:** اكتب برنامجًا يطلب من المستخدم إدخال قيمة درجة الحرارة لتحويلها من درجة مئوية (سيليسيوس) إلى فهرنهايت، واطبع نتيجة التحويل.

الفصل الثالث

التّنفيذ الشّرطيّ

3 التّنفيذ الشّرطيّ

1.3 التّعابير المنطقيّة

يُعرف التّعبير المنطقيّ بأنّه تعبيرٌ ذو قيمةٍ واحدة فقط، إمّا صائبة True، أو خاطئة False. يوضِّح المثال التّالي وظيفة العامل ==، والّذي يُقارِن بين معاملَين، ويقرّر إذا ما كانت هذه العمليّة True أم False.

```
>>> 5 == 5
True
>>> 5 == 6
False
```

تجدر الإشارة إلى أنّ True وFalse قيمتان خاصّتان تنتميان لصنف القيمة المنطقيّة class bool، أي أنّهما ليسَتا سلسلتين نصيتين (strings)، ويمكنك ملاحظة ذلك من خلال المثال التّالي:

```
>>>type (True)
<class 'bool'>
>>>type (False)
<class 'bool'>
```

يُعدّ العامل == أحد عوامل المقارنة الّتي يمكن تلخصيها كما يلي:

x لا يساوي y	x!=y
x أكبر تمامًا من y	x>y
x أصغر تمامًا من y	x<y
x أكبر أو يساوي y	x >= y
x أصغر أو يساوي y	x<= y
x مثل y	x is y
x ليس مثلy	x is not y

على الرّغم من أنّ هذه العمليّات قد تكون مألوفة لك، إلّا أنّ الرموز المستخدمة في لغة بايثون تختلف عن الرّموز الرّياضيّة لنفس العمليّات. على سبيل المثال، يُعتبر استخدام علامة مساواة واحدة = بدلًا من علامة مساواة مزدوجة == من الأخطاء الشّائعة الّتي قد يقع فيها المبرمج، وذلك لأنّ علامة المساواة الواحدة = تُعتبر عامل إسناد، بينما تُعتبر علامة المساواة المزدوجة == عامل مقارنة، كما أنّه لا وجود لرمز كهذا => أو هذا =< في لغة بايثون.

2.3 العوامل المنطقيّة

توجد ثلاثة عوامل منطقيّة في لغة بايثون، وهي: and و or و not، وتشابه معاني هذه العوامل في لغة بايثون معانيها في اللّغة الإنجليزيّة، فعلى سبيل المثال، تُعتبر هذه التّعابير محقَّقة فقط إذا كانت قيمة x أكبر تمامًا من 0 وأصغر تمامًا من 10.

```
x > 0 and x < 10
```

أمّا التّعبير:

```
n%2 == 0  or  n%3 == 0
```

فيُعتبر محقَّقًا أي True في حال تحقُّق أيٍّ من الشّرطين، سواء كان العدد يقبل القسمة على 2 أو على 3.

أخيرًا، يُستخدم عامل النّفي not لنفي التّعابير المنطقيّة، فمثلًا يُعتبر not (x > y) محقَّقًا True إذا كان x > y غير محقَّق False، أي إذا كان x أقلّ من أو يساوي y.

بالمعنى الدّقيق للكلمة، يجب أن تكون معاملات العوامل المنطقيّة عبارة عن تعبيرات منطقيّة، لكنّ لغة بايثون ليست صارمة للغاية؛ إذ تُفسّر أيّ رقم غير صفريّ على أنّه True.

```
>>> 17 and True
True
```

قد تكون هذه المرونة مفيدة، إلّا أنّ بعض التّفاصيل الدّقيقة قد تكون مربكة، ومن الأفضل تجنّبها ريثما تتأكّد من أنّك تعرف ما تفعله.

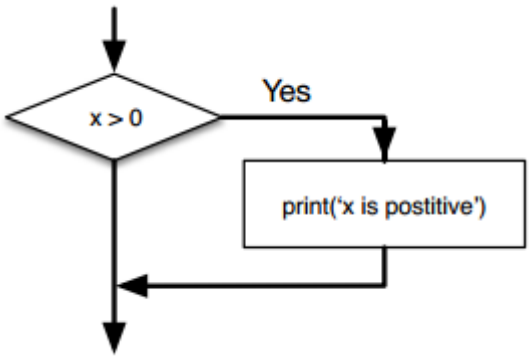
3.3 التّنفيذ المشروط

من أجل كتابة برامج مفيدة، نحتاج دومًا إلى التّحقُّق من الشّروط وتغيير سلوك البرنامج وفقًا لذلك، وتُستخدم العبارات الشّرطيّة لهذا الغرض.

يوضِّح المثال التّالي أبسط صيغة لعبارة if الشّرطيّة:

```
if x > 0 :
    print('x is positive')
```

يُسمّى التّعبير المنطقيّ بعد عبارة if بالشّرط. تنتهي تعليمة if برمز النّقطتين :، وتُضاف مسافة بادِئة قبل الأسطر البرمجيّة الّتي ستنفّذ في حال تحقُّق الشّرط في تعليمة if (بمقدار 4 فراغات أو باستخدام مفتاح tap في لوحة المفاتيح) للدّلالة على أنّها تنتمي إلى بنية if الشّرطيّة.



الشكل 5: آلية عمل if الشرطية

إذا كان الشّرط المنطقيّ محقَّقًا، فستُنفَّذ التّعليمات ذات المسافة البادئة (indented statement)، أمّا إذا كان الشّرط المنطقيّ غير محقَّق، فسيتم تجاهل تلك التّعليمات.

تملك عبارة if الشّرطيّة نفس البنية لتعاريف التّوابع (functions) أو حلقات for؛ إذ تتكوّن عبارة if الشّرطيّة من سطر أساسيّ ينتهي برمز النّقطتين : متبوعًا بمجموعة تعليمات ذات مسافة بادئة. تسمّى مثل هذه العبارات بالعبارات المركّبة لأنّها تتكوّن من أكثر من سطر.

يجب أن تَلي if تعليمةٌ واحدة ذات مسافة بادئة على الأقلّ، وما من حدٍّ أعلى لعدد التعليمات. من المفيد في بعض الأحيان ألّا تضع تعليمات ذات مسافة بادئة بعد عبارة if (عادةً ما تكون بمثابة بديل عن شيفرة برمجيّة لم تكتبها بعد). في هذه الحالة، يمكنك استخدام تعليمة pass الّتي لا تفعل شيئًا، كما في المثال التّالي:

```
if x < 0 :
    pass # need to handle negative values!
```

إذا كتبت عبارة if في مُفَسِّر لغة بايثون، فسيتغيّر رمز بداية الأسطر البرمجيّة من ثلاث علامات على شكل حرف V مقلوب <<<، أو ما يُعرف باسم شارة تلقين الأوامر، إلى ثلاث نقاط ... للإشارة إلى أنّك ضمن مجموعة التّعليمات الخاصّة بعبارة if، كما هو موضّح أدناه:

```
>>> x = 3
>>> if x < 10:
...         print('Small')
...
Small
>>>
```

عند استخدام مفسّر لغة بايثون، يجب أن تترك سطرًا فارغًا في نهاية كتلة التّعليمات، وإلّا ستُرجع لغة بايثون خطأ قواعديًّا بدلًا من تنفيذ تلك الأسطر البرمجيّة، كما هو موضّح في المثال التّالي:

```
>>> x = 3
>>> if x < 10:
...         print('Small')
...     print('Done')
  File "<stdin>", line 3
    print('Done')
        ^
SyntaxError: invalid syntax
```

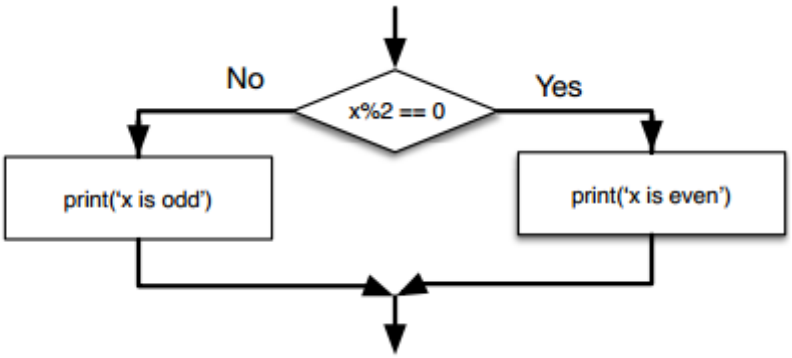
تجدر الإشارة إلى أنّ كتابة سطر فارغ في نهاية كتلة التّعليمات ليس ضروريًّا عند كتابة وتنفيذ نصّ برمجيّ (script)، ولكنّه قد يحسِّن قابليّة قراءة شيفرتك.

4.3 التّنفيذ البديل

الشّكل الثّاني من تعليمة if هو التّنفيذ البديل، حيث يوجد احتمالان، ويحدِّد الشّرطُ أيّهما يُنفّذ. تبدو بنية الجملة كما في المثال التّالي:

```
if  x%2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

كما هو معلوم، إذا كان باقي قسمة العدد x على 2 يساوي صفرًا، فإنّ x عددٌ زوجيٌّ، ويعرض البرنامج رسالة بهذا المعنى. أمّا إذا كان الشّرط غير محقَّق، فستُنفَّذ المجموعة الثّانية من التّعليمات، وهي عرض رسالة تقول إنّ x عددٌ فرديٌّ.



الشكل 6 : آلية عمل بنية if - else

نظرًا لأنّ الشّرط يجب أن يكون إمّا محقَّقًا أو غير محقَّق، فستُنَفّذ إحدى البدائل فقط، وتُسمّى البدائل بالفروع؛ لأنّها فروع في المسار التّنفيذيّ للبرنامج.

5.3 الشّروط المتسلسلة

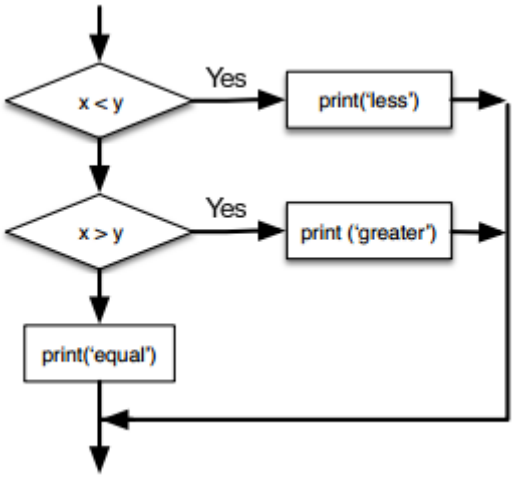
قد يكون هناك أكثر من احتمالين في بعض الأحيان، وعندها سنحتاج إلى أكثر من فرعين. في هذه الحالة، إحدى الطرق المستخدمة هي التّعبير الشّرطي المتسلسل، كما في المثال التّالي:

```
if x < y:
    print ('x is less than y')
elif x > y:
    print ('x is greater than y')
else:
   print ('x and y are equal')
```

elif هو اختصار لعبارة "else if". مرّة أخرى، سيُنفَّذ فرع واحد بالتّحديد.

ما من حدٍّ لعدد عبارات elif الشّرطيّة، وإذا كان هناك بند يحتوي على عبارة else، فيجب أن يكون في النّهاية، ولكن ليس من الضّروريّ أن يوجد.

```
if choice == 'a':
    print ('Bad guess')
elif choice == 'b':
    print ('Good guess')
elif choice == 'c':
    print ('Close, but not correct')
```



الشكل 7: بنية if – elif

يُوضِّح المثال أعلاه أنّ كلّ شرط يُفحص بالتّرتيب. إذا كان الشّرط الأوّل غير محقَّق، عندئذٍ يُفحص الشّرط التّالي، وهكذا دواليك. إذا تحقَّق شرطٌ ما، فسيُنفَّذ الفرع المقابل له، وتنتهي العبارة. حتّى لو تحقَّق أكثر من شرط واحد، سيُنفَّذ أوّل فرع تحقَّق شَرطه فقط.

6.3 الشّروط المتداخلة

من الممكن أيضًا أن يتداخل أحد الشّرطين مع الآخر. فعلى سبيل المثال، كان بإمكاننا كتابة مثال الفروع الثّلاثة السّابق بهذا الشّكل:

```
if x == y:
    print ('x and y are equal')
else:
    if x < y:
        print ('x is less than y')
    else:
        print ('x is greater than y')
```

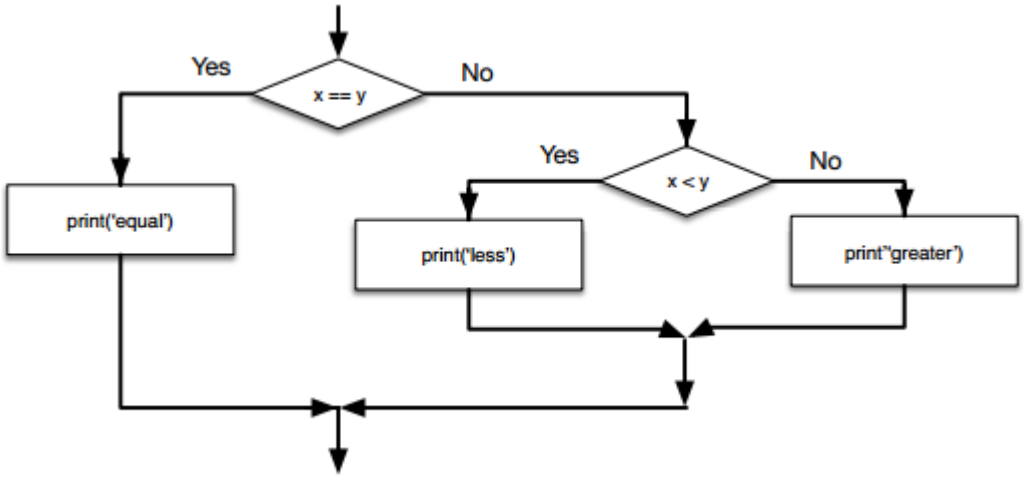
نلاحظ من المثال أعلاه أنّ الشّرط الخارجيّ يحتوي على فرعين. يحتوي الفرع الأوّل على تعليمة بسيطة، بينما يحتوي الفرع الثّاني على عبارة if شرطيّة أخرى تملك فرعين خاصّين بها. يحتوي هذان الفرعان على تعليمات بسيطة أيضًا، على الرّغم من أنّه كان من الممكن أن تكون عبارات شرطيّة مستقلّة.

على الرّغم من أنّ إزاحة التّعليمات تجعل هيكل الشّروط المتداخلة واضحًا، إلّا أنّه من الصّعب قراءة الشّروط المتداخلة بسهولة. عمومًا، من الجيّد تجنُّب استخدام الشّروط المتداخلة قدر الإمكان.

توفِّر العوامل المنطقيّة طريقة لتبسيط العبارات الشّرطيّة المتداخلة. فعلى سبيل المثال، يمكننا إعادة كتابة الشّيفرة البرمجيّة التّالية باستخدام شرط واحد فقط.

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

بما أن تعليمة print تنفَّذ فقط إذا تحقَّق الشّرطان السّابقان لها، فيمكننا الحصول على نفس النّتيجة باستخدام العامل المنطقيّ and، كما في المثال التّالي:



الشكل 8 : البنية الشرطية المتداخلة

```
if 0 < x and x < 10:
    print ('x is a positive single-digit number.')
```

7.3 التّعامل مع الاستثناء باستخدام بنية try وexcept

رأينا في وقت سابق مقطعًا من شيفرة برمجيّة، حيث استخدمنا تابعي input وint لقراءة وتمرير رقم صحيح أدخله المستخدم، ورأينا أيضًا كيف يمكن أن يكون القيام بذلك خادعًا، كما في المثال التّالي:

```
>>> prompt = "What is the air velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What is the air velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

عندما ننفِّذ هذه التّعليمات في مُفسِّر لغة بايثون، نحصل على موجِّه أوامر جديد من المُفسِّر، ونصاب بالحيرة، وننتقل إلى التّعليمة التّالية، ولكن إذا قمت بوضع هذه الشيفرة في مُحرّر نصوص خاصّ ببايثون وحدث هذا الخطأ، فإنّ النّصّ البرمجيّ سيتوقّف فورًا، وسيعرض رسالة تقرير

بالأخطاء، ولن يُنفِّذ التّعليمات التّالية. فيما يلي مثال لبرنامج يُحوّل درجة حرارة من وحدة الفهرنهايت إلى درجة حرارة مئويّة:

```
inp = input ('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Code: http://www.py4e.com/code3/fahren.py
```

إذا نفّذنا هذا البرنامج، وأدخلنا مُدخلًا غير مسموح به، فلن يُنفَّذ ذلك البرنامج ببساطة، وستظهر لنا رسالة الخطأ:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

توجد بنية تنفيذ شرطيّة مُضمّنة في لغة بايثون، تسمّى بنية try/except، ومهمّتها التّعامل مع الأخطاء المتوقّعة وغير المتوقّعة. تكمن فكرة try/except في أنّها تسمح للمبرمج بإضافة بعض التّعليمات (except block) لتُنفَّذ في حالة حدوث مشاكل في التّنفيذ التّسلسليّ للبرنامج. وفي حالة عدم وجود خطأ في تنفيذ البرنامج، فإنّ تلك الكتلة من التّعليمات لا تُنفَّذ، أو بمعنى آخر يتمّ تجاهلها.

يمكنك أن تشبّه خاصيّة try/except في بايثون بسياسة الضّمان للتّنفيذ التّسلسليّ للتّعليمات. بالاستفادة من هذه الخاصيّة، يمكننا إعادة كتابة برنامج تحويل درجة الحرارة من الفهرنهايت إلى

الدّرجة المئويّة بالشّكل التّالي:

```
inp = input ('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Please enter a number')

# Code: http://www.py4e.com/code3/fahren2.py
```

يبدأ البايثون في تنفيذ التّعليمات الخاصّة بكتلة try، فإن سار كلّ شيء كما هو مخطَّط له، عندئذ ستتجاهل بايثون مجموعة التّعليمة المندرجة في كتلة except. أمّا لو حدث خطأ ما، فسوف تنفَّذ التّعليمات الموجودة في كتلة except، أي أنّ البايثون سيقفز من كتلة try إلى كتلة except.

كما هو موضَّح في المثال التّالي: في الجزء الأوّل، يدخل المستخدم رقم 72، وهو رقم مقبول، لذا ستنفَّذ التّعليمة الخاصّة بتحويل درجة الحرارة. أمّا في الجزء الثّاني، يُدخل المستخدم سلسلة من الحروف fred بدلًا من إدخال عدد، وهذا غير مقبول، لذا تنفَّذ التّعليمة الموجودة في كتلة تعليمات except، وهي print ('please enter a number').

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222
python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

تُعرف عمليّة التّعامل مع الاستثناء (exception) باستخدام تعليمة try بالتقاط الاستثناء (catching an exception). في المثال السّابق، تظهِر تعليمات كتلة except رسالة خطأ.

بشكل عام، تمنحك خاصيّة التقاط الاستثناء فرصة لإصلاح المشكلة، أو المحاولة مرّة أخرى، أو على الأقلّ إنهاء البرنامج بأمان.

8.3 تجاوز التّحقُّق من التّعابير المنطقيّة

عندما يُعالج مُفسِّر لغة بايثون تعبيرًا منطقيًّا، مثل `x >= 2 and (x/y) > 2`، فإنّه يفحص التّعبير المنطقيّ من اليسار إلى اليمين. وبما أنّ العامل المنطقيّ هو `and`، فإذا كانت `x` أقلّ من 2، فإنّ التّعبير `x >= 2` يكون غير محقَّق `False`، وبالتّالي فإنّ التّعبير بأكمله يكون `False` بغض النّظر عمّا إذا قُيِّمَت `(x / y)> 2` كـ `True` أو `False`.

عندما يكتشف مُفسِّر لغة بايثون أنّه ما من داعٍ لتقييم بقيّة التّعبير المنطقيّ، فإنّه يتوقّف عن تقييمه، ولا يُجري الحسابات الخاصّة ببقيّة التّعبير المنطقيّ. تُعرف العمليّة الّتي تجعل مُفسِّر لغة بايثون يتوقّف عن تقييم التّعبير المنطقيّ لأن القيمة الإجماليّة معروفة بالفعل باسم تجاوز التحقّق من التّعابير المنطقيّة (short-circuiting the evaluation).

في حين أنّ هذه العمليّة قد تبدو وكأنّها خاصيّة جيّدة، فإنّ سلوك تجاوز التّحقُّق من التّعابير المنطقيّة يؤدّي إلى أسلوب ذكيّ في البرمجة، يسمّى نمط الحماية من الأخطاء (guardian pattern). لتوضيح ذلك، لاحظ تسلسل الشّيفرة التّالية في مُفسِّر لغة بايثون:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>\
```

في المثال أعلاه، فشلت العمليّة الحسابيّة الثّالثة، والسّبب أنّه أثناء تقييم مُفسِّر لغة بايثون للعمليّة

الحسابيّة (x/y) > 2، وُجد أنّ y = 0، ممّا تسبَّب بحدوث خطأ أثناء التّشغيل (runtime error). لكنّ المثالين الأوّل والثّاني نُفِّذا بنجاح، فالجزء الأوّل من هذه التّعبيرات x> = 2 قُيِّم كـ False في المثال الثّاني، لذا فإنّ (x/y) لم يُنفَّذ على الإطلاق بسبب قاعدة اختصار التّقييم ولم يكن هناك خطأ. يمكننا بناء التّعبير المنطقيّ لوضع نمط الحماية من الأخطاء بشكل استراتيجيّ قبل التّقييم مباشرة والّذي قد يتسبّب في حدوث خطأ بالشّكل التّالي:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

في المثال أعلاه، نلاحظ في التّعبير المنطقيّ الأوّل أنّ الشّرط المنطقيّ x >= 2، ولكنّ قيمة x تساوي 1، لذا فعبارة الشّرط المنطقيّ غير محقَّقة False، فلن يقيّم مُفسر لغة بايثون الجزء الثّاني من الشّرط المنطقيّ (x/y)>2 بسبب عدم تحقّق الجزء الأوّل من الشّرط المنطقيّ x >= 2، أي سيتوقّف مُفسِّر لغة بايثون عند عبارة and. أمّا في التّعبير المنطقيّ الثّاني، فالجزء الأوّل x >= 2 محقّق True، ولكنّ الجزء الثّاني من الشّرط المنطقيّ y != 0 غير محقّق False، لذلك فلن يقيّم مُفسِّر لغة بايثون الجزء الثّالث من الشّرط المنطقيّ (x/y)>2. وفي التّعبير المنطقيّ الثّالث، نلاحظ أنّ الشّرط y != 0 أتى بعد حساب (x/y)، لذلك يفشل تنفيذ هذه التّعليمات، وتظهر لنا رسالة خطأ بسبب القسمة على الصّفر.

خلاصة الأمر، في التّعبير الثّاني يمكننا أن نقول إنّ y != 0 تعمل كتعليمة حماية للتّأكّد من أنّنا ننفّذ (x/y) فقط، إذا كانت قيمة y غير صفريّة.

9.3 التّنقيح

يُعرَض تقرير بالخطأ عند حدوث خطأ ما. يحتوي هذا التّقرير على الكثير من المعلومات، ولكنّ هذه المعلومات قد تكون كثيرة لدرجة لا يقدّر المبرمج على استيعابها. عادةً ما تكون الأجزاء الأكثر فائدة في هذ المعلومات هي:

- ماهيّة الخطأ الّذي حدث.
- في أيّ جزء من الشّيفرة حدث ذلك الخطأ.

عادة ما يكون من السّهل العثور على الأخطاء القواعديّة، إلّا أنّ بعض التّفاصيل قد تكون مضلِّلة؛ إذ يمكن أن تكون أخطاء المسافات الفارغة (Whitespace) خادعة لأنّ الفراغات والمسافة tap غير مرئيّة ونحن معتادون على تجاهلها، كما في المثال التّالي:

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

في هذا المثال، تكمن المشكلة في أنّ السّطر الثّاني يحتوي على مسافة بادئة بمسافة واحدة. لكنّ رسالة الخطأ تشير إلى y، وهذ أمر مضلّل. بشكل عام، تشير رسائل الخطأ إلى مكان اكتشاف المشكلة، ولكنّ الخطأ الفعليّ قد يكون حدث في مكان سابق في التّعليمة، وأحيانًا في السّطر السّابق للتّعليمة الّتي يشير الخطأ إليها. بشكل عام، تخبرك رسائل الخطأ حول مكان اكتشاف المشكلة، ولكن غالبًا لا يكون هذا هو المكان الّذي حدثت فيه المشكلة بالضّبط.

10.3 فهرس المصطلحات

- **جسم التّعليمة (body):** تسلسل التّعليمات ضمن تعليمة مركّبة.
- **التّعبير المنطقيّ (Boolean expression):** هو تعبير قيمته الصّواب أو الخطأ (True or False).

- **الفرع (branch):** هو إحدى التّعليمات المتسلسلة البديلة في العبارات الشّرطيّة.
- **العبارات الشّرطيّة المتسلسلة (chained conditional):** هي العبارات الشّرطيّة الّتي تحتوي على مجموعة من الفروع التّسلسليّة.
- **عامل المقارنة (comparison operator):** هو أحد العوامل المنطقيّة الّتي تقارن بين قيم معاملاتها، مثل: == أو < أو >.
- **العبارة الشّرطيّة (conditional statement):** هي العبارة الّتي تتحكّم في تسلسل تنفيذ التّعليمات اعتمادًا على شرط ما.
- **الشّرط (condition):** هو التّعبير المنطقيّ الموجود في العبارات الشّرطيّة، والّذي يحدّد أيّ من الفروع سيُنفَّذ.
- **العبارات المركّبة (compound statement):** هي العبارات الّتي تتكوّن من جزأين: سطر أساسيّ، وكتلة تعليمات تابعة له. تكتب علامة النقتطين : في نهاية السّطر الأساسيّ، بينما تُزاح تعليمات الكتلة بمسافة بادئة لتشير إلى ارتباطها بالسّطر الأساسيّ.
- **نمط الحماية من الأخطاء (guardian pattern):** هو النّمط الّذي ينتج عند كتابة تعبير منطقيّ يحتوي على مقارنات إضافيّة للاستفادة من خاصيّة تجاوز التّحقّق من التّعابير المنطقيّة.
- **عامل منطقيّ (logical operator):** أحد العوامل الّتي تجمع بين التّعابير المنطقيّة، مثل: AND أو OR أو NOT .
- **العبارات الشّرطيّة المتداخلة (nested conditional):** عبارة شرطيّة تظهر في أحد فروع جملة شرطيّة أخرى.
- **عرض تقرير بالخطأ (Traceback):** قائمة بالتّوابع الّتي تُنفَّذ وتُطبع عند حدوث استثناء.
- **تجاوز التّحقُّق من التّعابير المنطقيّة (short circuit):** يقصد بها العمليّة الّتي يتوقّف فيها مفسِّر لغة بايثون عن تقييم تعبير منطقيّ ما لأنّ القيمة النّهائيّة للتّعبير المنطقيّ معروفة سلفًا دون الحاجة لتقييم بقيّة أجزاء التّعبير المنطقيّ.

11.3 تمارين

- **التمرين الأول:** اكتب برنامج لحساب الرّاتب لمنح الموظّف 1.5 ضعف سعر السّاعّة بالنّسبة لساعات العمل الّتي تزيد عن 40 ساعّة.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

- **التمرين الثاني:** أعِد كتابة برنامج الدّفع الخاصّ بك باستخدام `try /except` بحيث يتعامل البرنامج مع المدخلات غير الرّقميّة بشكل آمن عن طريق طباعة رسالة خطأ والخروج من البرنامج. المثال التّالي يوضّح عمليتي تنفيذ للبرنامج:

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input
Enter Hours: forty
Error, please enter numeric input
```

- **التمرين الثالث:** اكتب برنامجًا للمطالبة بالحصول على درجة تتراوح بين 0.0 و1.0. في حال كانت النّتيجة خارج النّطاق، اطبع رسالة خطأ، وإذا كانت الدّرجة بين 0.0 و1.0، اطبع تقديرًا يقابل قيمة الدّرجة باستخدام الجدول التّالي:

>= 0.9 A

>= 0.8 B

>= 0.7 C

>= 0.6 D

< 0.6 F

```
Enter score: 0.95
A
```

```
Enter score: perfect
Bad score
Enter score: 10.0
Bad score
Enter score: 0.75
C
Enter score: 0.5
F
```

نفّذ البرنامج بشكل متكرّر كما هو موضّح أعلاه لاختبار القيم المختلفة للإدخال.

الفصل الرابع

التّوابع

4 التوابع

1.4 استدعاء التوابع

في السّياق البرمجيّ، يُعرَّف التابع (function) على أنَّه سلسلة مُعرَّفة من التعليمات (العبارات البرمجيّة) التي تُنفّذ عمليّة حسابيّة. أي أنّ تعريف تابع ما يتطلَّب تحديد اسم التابع وتسلسل التعليمات، بحيث تستطيع "استدعاء" التابع من خلال اسمه لاحقًا.

لقد رأينا من قبل مثالًا عن استدعاء تابع:

```
>>> type (32)
<class ' int '>
```

اسمُ التابع هُنا هو type (بمعنى نوع)، أمّا التعبير داخل الأقواس، فيُسمّى وسيط التابع (argument)، وقد يكون الوسيط قيمةً ثابتة (value) أو متغيّرًا (variable) نُمرّرها إلى التابع بصفتِها دخلًا. نتيجة التابع type هي تحديد نوع الوسيط.

من الشّائع قَول إن التّابع "يأخُذ" الوسيط وَ "يُعيد" النتيجة، وتُدعى النتيجة هُنا القيمة المُعادة (return value).

2.4 التوابع الجاهزة

تُقدّم لغة بايثون العديد من التوابع الجاهزة التي يمكننا استخدامها دون الحاجة إلى تعريفها، حيث وضع مُبتكرو لغة بايثون مجموعة من التوابع لحلِّ مسائل شائعة، وضمَّنوا هذه التوابع في لغة بايثون ليتيحوا لنا استخدامها.

يُقدّم لنا التابعان max وَmin القيم الأكبر والأصغر على الترتيب ضمنَ قائمة (list).

```
>>> max ('Hello world')
' w '
>>> min ('Hello world')
' '
>>>
```

يُخبرنا التابع max بالمحرف الأكبر ضمن السلسلة النصّيّة ، وهو الحرف w. ويبيّن التابع min المحرف

الأصغر، وهو الفراغ (space).

يُعدُّ التابع len من التوابع الجاهزة شائعة الاستخدام، ويبيّن عدد العناصر الموجودة ضمن وسيطه، فإذا كان وسيط التابع len سلسلةً نصّيّة، يُعيد التابع عدد العناصر في السلسلة.

```
>>> len ('Hello world')
11
>>>
```

لا تقتصر هذه التوابع على السلاسل النصّيّة، بل يمكنها التعامل مع أيّة مجموعة من القيم، وهذا ما سنراه في الفصول القادمة.

يجب أن تُعامل أسماء التوابع الجاهزة بصفتِها كلمات محجوزة (أي أنَّه علينا تجنُّب استخدام كلمة max مثلًا بصفتها اسمًا لمُتغيّر).

3.4 توابع تحويل النوع

تُقدّم بايثون أيضًا توابعَ جاهزة تحوّل القيَم من نوع لآخر. يأخذ التابع int أيّة قيمة ويحوّلها إلى عدد صحيح (integer) إن أمكن، أو يظهر رسالة خطأ إذا لم يكن التحويل ممكنًا.

```
>>> int ( '32' )
32
>>> int ( 'Hello' )
ValueError: invalid literal for int() with base 10: 'Hello'
```

كما بإمكان التابع int تحويل الأعداد ذات الفاصلة العشريّة (float) إلى أعداد صحيحة، لكنّه لا يُقرّبها، بل يكتفي بإلغاء القسم العشريّ.

```
>>> int (3.99999)
3
>>> int (-2.3)
-2
```

يحوّل التابع float الأعداد الصحيحة والسلاسل النصّيّة إلى أعداد ذات فواصل عشريّة.

```
>>> float (32)
32.0
>>> float ( ' 3.14159 ' )
3.14159
```

أخيرًا، يحوّل التابع str وسيطَه إلى سلسلة نصّيّة.

```
>>> str (32)
' 32 '
>>> str (3.14159)
' 3.14159 '
```

4.4 التوابع الرياضيّة

تمتلك لغة بايثون وحدة رياضيّة تحوي معظم التوابع الرياضيّة المعروفة، وعلينا استدعاء هذه الوحدة حتّى نتمكّن من استخدامها:

```
>>> import math
```

تنشئ هذه التعليمة كائن وحدة (module object) يدعى math. ستحصل على بعض المعلومات عنه حين تضعه ضمن تعليمة الطباعة.

```
>>> print(math)
< module 'math' (built-in) >
```

يحتوي كائن الوحدة على التوابع والمتغيّرات المعرَّفة في الوحدة. للوصول إلى أحد هذه التوابع، عليك أن تحدّد اسم الوحدة واسم التابع مفصولَين بنقطة (dot)، والتي تُعرف أيضًا باسم (period)، وتدعى هذه الصيغة تأشيرة النقطة (dot notation).

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin (radians)
```

يحسب المثال الأول اللوغاريتم ذا الأساس 10 لنسبة الإشارة إلى الضجيج (ratio-noise-to-signal). كما تحوي الوحدة الرياضيّة تابعًا يُدعى log، والذي يحسب اللوغاريتم ذا الأساس النيبري e.

يوجِد المثال الثاني الجيب (sin) للعدد المسند في المتغيّر radians. يُعطي اسم المتغيّر تلميحًا إلى أنّ الجيب والدوال المثلثيّة الأخرى، مثل (cos, tan,...)، تأخذ قيمها بالراديان. للتحويل من الدرجات إلى الراديان نقسِّم على 360، ثمّ نضرب بـ 2π.

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

تُستخدم العبارة math.pi للحصول على قيمة المتغيّر pi من الوحدة math، والذي تمثّل قيمته العدد π بدقّة 15 خانة.

إن كنت خبيرًا في علم المثلّثات، يمكنك التحقّق من صحّة النتيجة السابقة بقسمة الجذر التربيعيّ للرقم 2 على 2 كما يلي:

```
>>> math.sqrt (2) / 2.0
0.7071067811865476
```

5.4 الأعداد العشوائيّة

تولّد غالبية البرامج الحاسوبيّة نفس قيم الخرج في كلّ مرّة تتلقى فيها قيم الدخل نفسها، لذلك تُدعى حتميّة (Deterministic). وعادةً ما تكون الحتميّة أمرًا جيّدًا، حيث أنّنا نتوقّع أن تثمر العمليّةُ الحسابيّة النتيجةَ ذاتها، إلّا أنّنا قد نحتاج أن يكون الحاسوب غير قابل للتنبّؤ في بعض التطبيقات. تعتبر الألعاب خير مثال، ولكن ثمّة تطبيقات أخرى سواها.

في الواقع، من غير السهل بناء برنامجٍ غير حتميٍّ بالمطلق، لكن ثمَّة بعض الأساليب التي تجعله يبدو كذلك على الأقل. أحد هذه الأساليب تكمن في استخدام الخوارزميّات (algorithms) التي تُولّد أعدادًا شبه عشوائيّة (pseudorandom). الأعداد شبه العشوائيّة ليست عشوائيّة بالمطلق، وذلك لأنّ عمليّة حاسوبيّة حتميّة تولّدها، إنّما يستحيل تمييز تلك الأعداد عن الأعداد العشوائيّة بمجرّد النظر إليها.

تُقدّم الوحدة العشوائيّة توابع تولّد أعداد شبه عشوائيّة (والتي سندعوها "عشوائيّة" للسهولة بدءًا من الآن).

يُعيد التابع random عددًا عشريًّا عشوائيًّا بين 0.0 وَ1.0 (متضمّنًا 0.0 دون 1.0).

في كلّ مرّة تستدعي التابع random ستحصل على عدد من سلسلة طويلة من الأعداد. لترى مثالًا على ذلك، شغّل الحلقة التالية:

```
import random
for i in range (10) :
    x = random.random()
    print (x)
```

ينتج عن البرنامج القائمةُ التالية المؤلّفة من 10 أعداد بين 0.0 وَ1.0 وغير المتضمّنة لـ 1.0:

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

التمرين الأول: شغّل البرنامج على حاسوبك وشاهد الأرقام التي ستحصل عليها. شغّل البرنامج أكثر من مرّة لترى الأعداد الناتجة.

يُعتبر التابع random واحدًا من عدّة توابعَ تتعامل مع الأعداد العشوائيّة.

يأخذ التابع randint معاملين الأول يمثل الحد الأدنى (low) والثاني الحد الأعلى (high) ويُعيد عددًا صحيحًا بينهما (مُتضمّنًا القيمتين).

```
>>> random.randint (5 , 10)
5
>>> random.randint (5 , 10)
9
```

لاختيار عنصر من مجموعة عشوائيّة، بإمكانك استخدام التعليمة choice:

```
>>> t = [1 , 2 , 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

بالإضافة إلى ذلك، تُؤمّن الوحدة random توابع لتوليد قيم عشوائيّة من التوزيعات المستمرّة، ومن ضمنها التوزيعات الغوصيّة (Gaussians)، والأسّيّة (Exponential)، وَغامّا (gamma)، وغيرها.

6.4 إضافة توابع جديدة

ما زلنا نستخدم التوابع الجاهزة في بايثون حتّى الآن، ولكن بإمكاننا أيضًا إضافة توابع جديدة. يُحدِّد تعريف التّابع (function definition) اسم التابع الجديد وسلسلة التعليمات التي تُنفَّذ عندما يُستدعى التابع.

حالما نعرّف تابعًا، يصبح بالإمكان إعادة استخدامه مرارًا وتكرارًا في البرنامج.

إليك المثال التالي:

```
def print_lyrics () :
    print (" I'm a lumberjack, and I'm okay. ")
    print (' I sleep all night and I work all day ')
```

def هي الكلمة المفتاحيّة الدالّة على تعريف التابع. اسم التابع هو print_lyrics. القواعد التي تنطبق على أسماء المتغيّرات تنطبق بدورها على أسماء التوابع، ويُسمح باستخدام الأحرف والأرقام وبعض علامات الترقيم، ولكن لا يجوز أن يكون المحرف الأول من اسم التابع رقمًا، كما أنّه ليس بإمكانك استخدام كلمة مفتاحيّة لتسمية التابع، بالإضافة إلى ذلك، ينبغي تجنُّب أن يكون للتابع وللمتغيّر الاسم ذاته.

تُشير الأقواس الفارغة بعد اسم التابع ( ) إلى أنّه لا يأخذ أي وسيط. سننشئ لاحقًا توابع تقبل الوسائط بصفتِها مُدخلات.

يُسمّى السطر الأوّل من تعريف التابع الترويسة (Header)، وتُسمّى البقيّة جسم التابع (body).

يجب أن ينتهي العنوان بنقطتي القول : ، أمّا جسم التابع، فيجب أن يكون مُزاحًا. اتُّفِقَ على أن تكون المسافة البادئة 4 فراغات دومًا، ويمكن لجسم التابع أن يحتوي أيّ عدد من التعليمات.

إذا كتبتَ تعريف التابع في الوضع التفاعليّ (interactive mode)، فإنّ المفسّر سيطبع ثلاث نقاط ... لإعلامكَ بأنّ التعريف غير كامل.

```
>>> def print_lyrics() :
...     print (" I'm a lumberjack, and I'm okay. ")
...     print (' I sleep all night and I work all day. ')
...
```

لإنهاء التابع، سيتعيّن عليك إدخال سطر فارغ (وهذا ليس ضروريًّا في حال كتابة النص البرمجيّ ضمن ملف).

إن عمليّة تعريف تابع تعطي بدورها مُتغيّرًا بنفس اسم التابع.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

قيمة التابع print_lyrics هي كائن لتابع (function object) من النوع "function"، وطريقة استدعاء تابع جديد مشابهة لاستدعاء التوابع الجاهزة:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay
I sleep all night and I work all day
```

حالما تعرّف تابع، يُمكنك استخدامه ضمن تابع آخر. مثلًا، لتكرار كلمات الأغنية السابقة، بالإمكان إضافة تابع يُدعى repeat_lyrics.

```
def repeat_lyrics() :
    print_lyrics()
    print_lyrics()
```

ثمّ استدعِ التابع repeat_lyrics.

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay
I sleep all night and I work all day.
```

7.4 التعاريف واستخداماتها

سيبدو البرنامج بأكمله على الشكل التالي بعد تجميع أجزاء النصّ البرمجيّ من القسم السابق:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
repeat_lyrics()
# Code: http://www.py4e.com/code3/lyrics.py
```

يحتوي هذا البرنامج على تعاريف لتابعين: print_lyrics وَrepeat_lyrics. تُنَفَّذ تعريفات التابع مثل غيرها من التعليمات، وهي تنشئ كائنات التابع. لا تُنفَّذ العبارات داخل التابع حتى يُستدعى التابع، كما أنّ تعريف التابع لا يُولّد خرجًا.

بالطبع عليك أن تنشئ تابعًا قبل أن تنفّذ محتواه. بمعنى آخر، يجب كتابة تعريف التابع قبل استدعائه لأوّل مرّة.

التمرين الثاني: انقل السطر الأخير من البرنامج السابق إلى الأعلى بحيث تصبح تعليمة استدعاء

التابع موجودة قبل التعريفات. شغّل البرنامج ولاحظ رسالة الخطأ الناتجة.

التمرين الثالث: انقل تعليمة استدعاء التابع إلى الأسفل ثانيةً، وانقل تعريف print_lyrics ليصبح بعد تعريف repeat_lyrics. ما الذي يحدث عند تشغيل البرنامج؟

8.4 تسلسل التنفيذ

من أجل ضمان أنّ التابع عُرِّفَ قبلَ استخدامه لأوّل مرّة، عليك أن تعرف الترتيب الذي تُنَفّذ وِفقه التعليمات، والذي يُعرف بتسلسل التنفيذ (Flow of execution).

يبدأ التنفيذ دومًا من التعليمة الأولى في البرنامج، حيث تُنفّذ التعليمات واحدة تلو الأخرى بالترتيب من الأعلى للأسفل. لا تُغيّرُ تعريفات التوابع من تسلسل التنفيذ في البرنامج، لكن تذكّر أنّ التعليمات لا تُنفّذ حتّى يُستدعى التابع.

تُعتبر عمليّة استدعاء التابع بمثابة انعطاف في تسلسل التنفيذ، فبدلًا من الذهاب إلى الجملة التالية، يقفز التسلسل إلى جسم التابع مُنفِّذًا جميع التعليمات هناك، ثمّ يعود بعد ذلك ليستأنف من حيث توقّف.

يبدو هذا سهلًا إلى أن تتذكّر أنّ بإمكان التابع نفسه استدعاء تابع آخر، حيث قد يضطر البرنامج - أثناء وصوله لمنتصف أحد التوابع- إلى تنفيذ تعليمات في تابع آخر، ولكن خلال تنفيذ التابع الجديد قد يُنفّذ البرنامج تابعًا آخر.

لحسن الحظّ، فإنّ لغة بايثون جيّدة في حفظ مسارها، إذ في كلّ مرّة يكتمل تنفيذ أحد التوابع، يستأنف البرنامج من حيث توقّف في التابع الذي استدعاه، وعندما يصل إلى نهاية البرنامج تنتهي العمليّة.

ما المغزى من هذه القصة الشيّقة؟ عندما تقرأ برنامجًا، قد لا تكون القراءة من الأعلى للأسفل فعّالةً دائمًا، فأحيانًا يكون تتبُّعُ تسلسل التنفيذ منطقيًّا أكثر.

9.4 المُعاملات والوسائط

تتطلّب بعض التوابع الجاهزة التي صادفناها وسائط، فمثلًا عندما تستدعي التابع math.sin، فإنّك تمرّر له رقم باعتباره وسيطًا. تأخذ بعض التوابع أكثر من وسيط: التابع math.pow يأخذ وسيطين، هما "الأساس والأس".

تُسند هذه الوسائط لمُتغيّرات داخل التابع تُدعى مُعاملات (Parameters).

إليك مثالًا عن التوابع المعرّفة من قبل المستخدم (User defined functions)، والتي تأخذ وسيطًا:

```
>>> def print_twice(bruce):
        print(bruce)
        print(bruce)
```

يسند هذا التابع الوسيط إلى مُعامل اسمه bruce. عندما يُستدعى التابع، فإنّه يطبع قيمة المعامل (مهما كانت) مرّتين.

يعمل هذا التابع مع أيّة قيمة يُمكن كتابتها.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

تنطبق على التوابعِ المعرّفة من قبل المستخدم قواعدُ إنشاء التوابع (ذاتها التي تنطبق على التوابع الجاهزة، لِذا بإمكاننا استخدام أي تعبير كوسيط للتابع print_twice.

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

يُحسَب الوسيط قبل استدعاء التابع، لذلك فإنّ التعابير 'spam' *4 وَmath.cos(math.pi) تُحسب مرّة واحدة فقط.

بإمكانك أيضًا استخدام مُتغيّرٍ كوَسيط:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

لا علاقة لاسم المتغيّر الذي مرّرناه كوسيط Michael باسم المُعامِل bruce، فبِغضِّ النظر عن الاسم الذي أُطلِقَ على القيمة في عملية الاستدعاء، هُنا في التابع print_twice ندعوها bruce.

10.4 التوابع المُنتجة والتوابع الخالية

بعض التوابع التي نستخدمها (كالتوابع الرياضيّة) تُعطي نتائجًا. ولعدم وجود اسم أفضل، ابتكرتُ لها اسم التوابع المُنتجة (fruitful functions).

التوابع الأخرى، مثل print_twice، تُنجز مهمّة، لكنّها لا تُرجِع قيمة. نُطلق على هذه التوابع اسم التوابع الخالية (Void functions).

عندما تستدعي تابعًا مُنتجًا، فهدفك في أغلب الأحيان هو الاستفادة من النتيجة.

مثلًا، قد تُسند النتيجة إلى مُتغيّر أو تستخدمها كجزء من التعبير:

```
X = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

عندما تستدعي تابعًا في الوضع التفاعليّ، فإنّ لغة بايثون تعرض النتيجة.

```
>>> math.sqrt(5)
2.23606797749979
```

لكن في وضع كتابة النصّ البرمجيّ ضمن ملف، إذا استدعيت تابعًا مُنتجًا ولم تُخزّن النتيجة في مُتغيّر، فإنّ القيمة المُرجعة ستختفي.

```
math.sqrt(5)
```

يَحسب هذا النصّ البرمجيّ الجذر التربيعيّ للعدد 5، ولكن بما أنّه لم يُخزّن النتيجة في متغيّر أو يعرضها، فهو غير مفيد.

قد تعرض التوابع الخالية شيئًا ما على الشاشة، أو قد تملك تأثيرًا آخر، لكنّها لا تملُك قيمة مُرجعة.

إذا حاولت أن تسند النتيجة إلى مُتغيّر، ستحصل على قيمة مميّزة تدعى None.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

القيمة None ليست ذاتها السلسلة النصّيّة "None"، بل تُعدُّ قيمة مميّزة ذات نوعٍ خاصّ.

```
>>> print (type (None))
<class 'NoneType'>
```

نستخدم التعليمة return في التابع الخاصّ بنا لإرجاع نتيجة التابع. مثلًا، بإمكاننا إنشاء تابع بسيط للغاية اسمه addtwo، والذي يجمع رقمين ويُرجع نتيجة الجمع.

```
def addtwo (a, b):
    added = a + b
    return added
x = addtwo(3, 5)
print(x)
# Code: http://www.py4e.com/code3/addtwo.py
```

عندما يُنفّذ هذا النصّ البرمجيّ، تطبع تعليمة print العدد 8 بسبب التابع addtwo الذي استُدعي ومُرِّر الوسيطان 3 و5 له.

داخل التابع، المعاملات a وَb هما 3 وَ5 على الترتيب.

يَحسب التابع ناتج جمع العددين، ويضعه في متغيّر محلّي للتابع اسمه added، ثمّ يستخدم تعليمة return لإرسال النتيجة المحسوبة إلى التابع المُستدعى كنتيجة للتابع، والتي تكون مُسندة بدورها إلى المتغيّر x، وتُكتب على الشاشة.

11.4 لماذا نستخدم التوابع

قد لا يكون واضحًا. لمَ يُعَد تقسيمِ البرنامج إلى توابع عمليّةً تستحقُّ العناء.

إليك عدّة أسباب:

- تمنحك عمليّة إنشاء تابع جديد الفرصةَ لتسمية مجموعة من التعليمات، ممّا يجعل برنامجك أسهل للقراءة والفهم والتصحيح.
- يمكن للتوابع أن تجعل برنامجك أصغر من خلال التخلّص من التعليمات المكرَّرة، بحيث إن أردتَ لاحقًا إجراء تغيير، فسينحصر ذلك في مكان واحد فقط.
- تتيح لك عمليّة تجزئة البرنامج الطويل تنقيحَ أجزاء البرنامج كلٌّ على حدة، ومن ثمّ تجميعها معًا في برنامج واحد.
- غالبًا ما تكون التوابع المصمَّمة بشكل جيّد مفيدة في العديد من البرامج. حالما تكتب وتنقّح أحدها، بإمكانك إعادة استخدامه.

في بقيّة الكتاب، سنستخدم غالبًا تعريف التابع لشرح مفهوم ما.

تتضمّن أساسيَّات مهارة إنشاء التوابع واستخدامها أن تملك تابعًا يُجسّد فكرة، مثل "أوجد القيمة الصّغرى في مجموعة من القيَم".

سنعرض عليك لاحقًا مجموعة تعليمات توجد أصغر قيمة ضمن مجموعة قيم، وسنقدّمها لك كتابع يدعى "min"، والذي يأخذ سلسلة القيم كوسائط له ويُعيد القيمة الأصغر بينها.

12.4 التّنقيح

إذا كنت تستخدم محرّر نصوص لكتابة نصوصك البرمجيّة، فستواجه غالبًا مشاكل متعلّقة بالفراغات (spaces) والإزاحات (tabs).

الطريقة المُثلى لتجنّب هذه المشاكل هي استخدام فراغات حصرًا (دون الإزاحات). تقوم غالبيّة محرّرات النصوص التي تتعامل مع لغة بايثون بهذا الأمر بشكل افتراضيّ، لكن البعض لا يفعل.

عادةً ما تكون الإزاحات والفراغات غير مرئيّة، ممّا يجعل تنقيحها أصعب، لذا حاول إيجاد محرّر نصوص يُنظّم لك المسافات البادئة.

إضافة إلى ذلك، لا تنسَ حفظ برنامجك قبل تشغيله. تقوم بعض بيئات التطوير بذلك بشكل تلقائيّ، ولكنّ بعضها الآخر لا يفعل، لذا قد يكون البرنامج الذي تشاهده في محرّر النصوص مختلفًا عن البرنامج الذي تشغّله.

ستأخذ عمليّة التنقيح وقتًا طويلًا إذا استمرّيت بتشغيل نفس البرنامج الخاطئ مرارًا وتكرارًا. احرص

على أن يكون النصّ البرمجيّ التي تنظر إليه هو ذات النص الذي تشغّله. وفي حال لم تكن مُتأكّدًا، اكتب شيئًا ما مثل print("hello") في بداية البرنامج وشغّله من جديد.

إذا لم تظهر لك الكلمة hello، فإنّك لا تشغّل البرنامج الصحيح.

13.4 فهرس المصطلحات

- **الخوارزميّة (Algorithm):** الخطوات العامّة لحلّ أنواع من المشكلات.
- **الوسيط (Argument):** قيمة تُقدّم للتابع عند استدعائه، تُسند هذه القيمة إلى المعامل المناسب في التابع.
- **جسم التابع (body):** سلسلة من التعليمات داخل تعريف التابع.
- **التركيب (composition):** استخدام تعبير كجزء من تعبير أوسع، أو تعليمة كجزء من تعليمة أوسع.
- **الحتمية (deterministic):** يُشير إلى البرنامج الذي ينفّذ الشيء ذاته عند إعطائه نفس المُدخلات في كلّ مرّة يجري تشغيله.
- **تأشيرة النقطة (dot notation):** صياغة تستخدم عند استدعاء تابع في وحدة عبر كتابة اسم الوحدة متبوعًا بنُقطة واسم التابع.
- **تسلسل التنفيذ (flow of execution):** الترتيب الذي تنفَّذ وفقه التعليمات خلال تشغيل البرنامج.
- **التابع المُنتج (fruitful function):** التابع الذي يُرجع قيمة.
- **التابع (function):** سلسلة مُعرّفة من التعليمات التي تنجز عمليّة مفيدة. قد تأخذ التوابع وسائط وقد لا تأخذ، كما قد تقدّم نتيجة وقد لا تفعل.
- **استدعاء التابع (function call):** تعليمة تؤدّي إلى بدء تنفيذ التابع، وتتألّف من اسم التابع متبوعًا بسلسلة وسائط.
- **تعريف التابع (function definition):** تعليمة تنشئ تابعًا جديدًا عبر تخصيص اسم له، ومُعاملات وتعليمات تؤدّي إلى تنفيذهِ.

- **كائن التابع (function object):** قيمة تنشأ بعد تعريف التابع. إن اسم التابع هو متغيّر يدلّ على كائن التابع.
- **الترويسة (header):** السطر الأوّل من تعريف التابع.
- **تعليمة import (Import statement):** تعليمة تقرأ ملفّ الوحدة، وتنشئ كائن منه.
- **كائن النموذج (module object):** القيمة التي تُنشئها تعليمة import، والتي تتيح الوصول إلى البيانات والشيفرات المعرّفة في الوحدة.
- **المُعامِل (parameter):** اسم يُستخدم داخل التابع للدلالة على قيمة مُمرّرة بصفتها وسيطًا.
- **شبه العشوائيّ (pseudorandom):** تُشير إلى سلسلة من الأعداد التي تبدو وكأنّها عشوائيّة، ولكنّها تولَّد من قبل برنامج حتميّ.
- **القيمة المُرجعة (return value):** نتيجة التابع. إذا استُخدم استدعاء التابع كتعبير، فإن القيمة المرجعة هي قيمة هذا التعبير.
- **التابع الخالي (void function):** هو التابع الذي لا يُرجع أيّة قيمة.

14.4 تمارين

- **التمرين الرابع:** ما هو الهدف من الكلمة المفتاحيّة "def" في لغة بايثون؟

  a. هي كلمة عامّيّة تعني "الشيفرة التالية رائعة".
  b. تُشير إلى بداية التابع.
  c. تُشير إلى أنّ المسافة البادئة التالية من الشيفرة مُخزَّنة للاستخدام لاحقًا.
  d. كلّ من b وc صحيح.
  e. لا شيء ممّا سبق.

- **التمرين الخامس:** ماذا سيعرض برنامج بايثون التالي على الخرج؟

```
def fred():
    print("Zap")
def jane():
```

```
    print("ABC")

jane()
fred()
jane()
```

a. Zap ABC jane fred jane

b. Zap ABC Zap

c. ABC Zap jane

d. ABC Zap ABC

e. Zap Zap Zap

- **التمرين السادس:** أعد كتابة برنامج حساب الراتب الذي يعطي قيمة 1.5 ضعف الأجر للوقت الإضافيّ، وأنشئ تابعًا يسمّى computepay بحيث يأخذ مُعاملين (hours وrate).

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.
```

- **التمرين السابع:** أعد كتابة برنامج الدرجات من الفصل السابق مستخدمًا تابعًا يدعى computegrade، والذي يأخذ النتيجة كمعامل له ويعيد الدرجة كسلسلة نصّيّة.

Score	Grade
>= 0.9	A
>= 0.8	B
>= 0.7	C
>= 0.6	D
< 0.6	F

```
Enter score: 0.95
A
```

```
Enter score: perfect
Bad score
Enter score: 10.0
Bad score
Enter score: 0.75
C
Enter score: 0.5
F
```

كرّرْ تشغيل البرنامج واختبره مع القيم المختلفة للدخل.

الفصل الخامس

التكرار

5 التكرار

1.5 تحديث قيم المتغيرات

تستخدم تعليمة الإسناد لتحديث قيمة متغير اعتمادًا على قيمته القديمة:

```
x = x + 1
```

ويعني هذا السطر: قم بإحضار قيمة المتغير x الحالية وأضف إليها واحدًا ثم اجعل الناتج قيمةً جديدةً للمتغير x.

سنحصل على رسالة خطأ في حال حاولنا أن نحدِّث قيمة متغير غير موجود سابقًا لأن لغة بايثون تنفِّذ الطرف الأيمن قبل أن تحدِّث قيمة المتغير x:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

لذا عليك تعريف المتغير أولًا قبل أن تحدِّث قيمته:

```
>>> x = 0
>>> x = x + 1
```

تسمى عملية تحديث قيمة متغير ما بإضافة 1 إلى قيمته القديمة بالزيادة (increment) أما في حال طرح 1 من القيمة القديمة فتسمى إنقاصًا (decrement).

2.5 حلقة while

تُستخدم الحواسيب عادةً لأتمتة المهام المتكررة، ففي حين تبرع الحواسيب في تكرار المهام المتطابقة أو المتشابهة، يعجز البشر عن ذلك دون ارتكاب العديد من الأخطاء، لذا فإن لغة بايثون تزودنا بالعديد من المميزات التي تسهل تنفيذ مثل هذه العمليات.

تعد حلقة while شكلًا من أشكال التكرار في لغة بايثون، وفيما يلي برنامجٌ بسيط يقوم بالعد التنازلي ابتداءً بالرقم 5 ثم ينتهي بعبارة: "Blastoff!".

```
n = 5
while n > 0:
    print(n)
```

```
        n = n – 1
print('Blastoff!')
```

يمكن فهم هذه التعليمات بسهولة فهي تعني: "اطبع قيمة n ثم اطرح منها واحدًا إذا كانت قيمة n أكبر من الصفر، وعندما تصبح قيمة n صفرًا اخرج من تعليمة while واطبع كلمة "Blastoff!"، حيث تنفذ تعليمة while كالآتي:

1. قيّم فيما إذا كان الشرط قد حقق أم لا
2. إذا لم يُحقق الشرط، اخرج من الحلقة ثم نفذ التعليمة التالية.
3. إذا حُقق نفذ محتوى (جسم) الحلقة ثم عد إلى الخطوة الأولى.

لعلك عرفت الآن سبب تسميتها بالحلقة (Loop) وذلك لأن الخطوة الثالثة تعود إلى الخطوة الأولى مرة أخرى.

نطلق على كل مرة يتم فيها تنفيذ جسم الحلقة بالتكرار (iteration)، فالحلقة المذكورة في المثال السابق لها خمس تكرارات أي أن جسم الحلقة ينفَّذ خمس مرات متتالية.

يجب أن تغير التعليمات الواردة في جسم الحلقة قيمة متغير معين نسميه متغير التكرار (iteration variable) حتى نصل إلى مرحلة لا يتحقق فيها شرط الحلقة ومن ثم يتوقف تنفيذها. وفي حال غياب متغير التكرار، سيتكرر تنفيذ الحلقة باستمرار وينتج عن ذلك ما يسمى بالحلقة اللانهائية (infinite loop).

3.5 الحلقات اللانهائية

يجد المبرمجون التعليمات المكتوبة على عبوات الشامبو فكاهيةً، حيث أن خطوات الاستخدام هي (ضع قليلًا من المستحضر، اشطف بالماء، وكرر ذلك) لكنها تمثِّل حلقة لانهائية لغياب متغير التكرار الذي يحدد عدد مرات تنفيذ هذه الحلقة.

نعلم في مثال العد التنازلي السابق أن الحلقة منتهية حيث أعطينا n قيمة محددة، ونلاحظ انخفاض قيمة n عند كل مرة ينفَّذ فيها جسم الحلقة حتى تصل أخيرًا إلى الصفر.

قد يغيب متغير التكرار في بعض الحالات الأخرى لتصبح الحلقة لانهائية، فقد لا نعلم مثلًا متى ينتهي تنفيذ الحلقة إلا في منتصفها، وعندها نتعمد كتابة حلقة لانهائية ثم نستخدم تعليمة الإيقاف break

للخروج من الحلقة عند تحقق شرط معين.

الحلقة التالية لانهائية وذلك لأن الشرط المستخدم فيها هو الثابت المنطقي True (وهو محقق دائمًا).

```
n = 10
while True:
        print (n, end=' ')
        n = n – 1
print('Done!')
```

إذا نفذت هذا البرنامج فإما أن تتعلم كيف تنهي مهمة بايثون الخارجة عن السيطرة أو ستضطر إلى استخدام زر الطاقة لإيقاف تشغيل الحاسوب، فالبرنامج سيستمر في العمل بشكل لانهائي أو حتى تنفذ بطارية جهازك.

قد تبدو هذه الحلقة اللانهائية بلا فائدة، لكن نستطيع توظيف هذا النمط من الحلقات إن أضفنا أمرًا إلى جسم الحلقة للخروج منها باستخدام تعليمة الإيقاف break عند تحقق شرط معين، فعلى فرض أننا نريد كتابة برنامج يأخذ من المستخدم دخلًا حتى يدخل كلمة done، يمكننا كتابة الحلقة التالية:

```
while True:
    line = input ('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
# Code: http://www.py4e.com/code3/copytildone1.py
```

لاحظ أن الشرط المستخدم في هذه الحلقة هو الثابت True وهو محقق دائمًا، وعليه فإن الحلقة ستتكرر حتى تنفذ تعليمة الإيقاف break. كلَّما نفذت الحلقة سيظهر للمستخدم إشارة > طالبًا منه إدخال ما يريد، وعند إدخال كلمة done تقوم تعليمة الإيقاف break بالخروج من الحلقة، وإلا فإن البرنامج سيطبع ما يدخله المستخدم ثم يعود إلى بداية الحلقة.

إليك تشغيلًا تجريبيًّا للبرنامج السابق:

```
> hello there
hello there
> finished
finished
> done
Done!
```

من الشائع استخدام هذه الطريقة في كتابة حلقة while، حيث أنه من الممكن التحقق من شرط الحلقة في جسمها وليس فقط في ترويستها، كما يمكن أن نعبر عن شرط الإيقاف بالشكل (توقف عند تحقق ذلك الشرط) بدلًا من التعبير (تابع التنفيذ حتى يحدث ذلك الشرط).

4.5 إنهاء التكرار باستخدام تعليمة Continue

أثناء تنفيذ أحد تكرارات الحلقة، قد نحتاج إلى إيقاف تنفيذ التكرار الحالي والعودة لبدء تكرار جديد، نستخدم تعليمة المتابعة continue التي توقف تنفيذ التكرار الحالي دون الخروج من الحلقة.

فيما يلي مثال عن حلقة تقوم بطباعة النص المدخل إلى أن يدخل المستخدم كلمة done، ولكنها تتجاهل السطور المدخلة التي تبدأ برمز # ولا تقوم بطباعتها (ما يشبه التعليقات المستخدمة في لغة بايثون):

```
while True:
    line = input ('> ')
    if line [0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print ('Done!')
# Code: http://www.py4e.com/code3/copytildone2.py
```

يظهر عند تشغيل البرنامج مع إضافة تعليمة المتابعة continue:

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

نلاحظ طباعة كل السطور التي تم إدخالها ما عدا السطر الذي ابتدأ بالرمز #، لأن تنفيذ تعليمة المتابعة continue يوقف تنفيذ التكرار الحالي ويعود لتنفيذ حلقة while للبدء بتكرار جديد وبالتالي تجاهل تعليمة الطباعة.

5.5 الحلقات المحددة باستخدام For

نحتاج أحيانًا أن نتعامل مع مجموعة من الأشياء كقائمة من الكلمات أو الأرقام أو حتى مع أسطر ملف نصي، حينئذ يستحسن استخدام الحلقة المحددة for. تعد حلقة while حلقة غير محددة لأنها ببساطة تتكرر حتى يصبح شرطها غير صحيح، في حين تعد حلقة for حلقةً محددة لأنها تتكرر بعدد الأشياء الموجودة في المجموعة.

إن قواعد كتابة حلقة for مماثلة لكتابة حلقة while حيث يوجد ترويسة لحلقة for ويتلوها جسم الحلقة. مثال:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
        print('Happy New Year:', friend)
print('Done!')
```

قد لا يتضح معنى هذه الحلقة للقارئ مباشرة كما هو الحال في حلقة while، ولكن إذا اعتبرنا المتغير friends قائمة تتكون من ثلاث عناصر من النوع سلسلة نصية فيمكن ان نصيغ معنى الحلقة كما

يلي: نفّذ التعليمات الواردة في جسم الحلقة مرة لكل عنصر friend موجود في القائمة المسماة friends.

نرى أن كلًا من for وin كلمات محجوزة للغة بايثون، وكل من friend وfriends متغيرات.

```
for friend in friends:
print ('Happy New Year:', friend)
```

نسمي المتغير friend متغير التكرار في الحلقة، حيث أنه يتغير لكل تكرار ويعد مسؤولًا عن اكتمال تنفيذ الحلقة، كما يتعاقب على العناصر النصية الثلاثة الموجودة في القائمة friends، وفيما يلي الخرج الناتج عن تنفيذ هذه الحلقة:

```
Happy New Year: Joseph

Happy New Year: Glenn

Happy New Year: Sally

Done!
```

6.5 أنماط كتابة الحلقات

نستخدم عادة كل من حلقتي for وwhile لتنفيذ عملية ما على مجموعة عناصر لقائمة (list) أو محتويات ملف، قد تكون هذه العملية البحث عن شيء ما كأكبر أو أصغر قيمة بين البيانات التي نتعامل معها.

تتم عادة هيكلة الحلقات كما يلي:

1- إعطاء قيمة ابتدائية لمتغير أو عدة متغيرات قبل بداية الحلقة.
2- تنفيذ عملية حسابية على كل عنصر في جسم الحلقة، وقد يترافق ذلك مع تغيير في قيم المتغيرات.
3- إظهار القيم الناتجة للمتغيرات بعد إتمام تنفيذ الحلقة.

سنورد تاليًا مثالًا نستخدم فيه قائمة من الأرقام لنوضح المفاهيم الواردة سابقًا وطريقة بناء عدة أنماط للحلقات.

1.6.5 حلقات العد والجمع

إذا أردنا ان نحصي عدد الأرقام الموجودة في قائمة ما، فيمكن أن نكتب الحلقة التالية:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
        count = count + 1
print ('Count: ', count)
```

بدايةً أعطينا المتغير count القيمة الابتدائية صفر، ثم كتبنا حلقة for لتنفيذها على قائمة الأرقام. إن متغير التكرار في هذا المثال هو المتغير itervar، ونلاحظ أننا لا نستخدم هذا المتغير مباشرة في جسم الحلقة، إلا أنه يتحكم في تنفيذ الحلقة ويتسبب بتنفيذ جسمها مرة لكل عنصر في القائمة.

أضفنا واحد إلى قيمة المتغير count في جسم الحلقة لكل عنصر من عناصر القائمة، وأثناء تنفيذ الحلقة فإن قيمة المتغير count تساوي عدد القيم التي مررنا بها حتى الآن.

عند اكتمال تنفيذ الحلقة تساوي قيمة المتغير count العدد الإجمالي للعناصر، الذي يظهر عند اكتمال التنفيذ.

لنرى الآن حلقةً مشابهةً للحلقة السابقة ولكنها تقوم بحساب مجموع قائمة من الأرقام:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print ('Total: ', total)
```

استخدمنا في هذه الحلقة فعليًّا متغير التكرار itervar، فبدلًا عن إضافة واحد إلى المتغير كما في الحلقة السابقة، فقد أضفنا القيمة الفعلية للعنصر (3 و41 و12..... إلخ) إلى المجموع الحالي عند كل تكرار للحلقة، وبالنظر إلى المتغير total فإنه يمثل قيمة المجموع الجاري للقيم التي مررنا بها حتى الآن، ولذلك فإننا نعطي هذا المتغير القيمة صفر قبل بداية الحلقة. يمثل ذلك المتغير عند اكتمال الحلقة مجموع القيم في القائمة.

أثناء تنفيذ الحلقة فإن المتغير total يجمّع أو يراكم قيم العناصر، لذا فهو يسمى المراكم

.(accumulator)

لا تعتبر أي من الحلقتين السابقتين حلقاتٍ مفيدة عمليًّا لوجود توابع جاهزة لهذا الغرض (التابع len() لحساب عدد العناصر في قائمة والتابع sum() لحساب مجموع العناصر في قائمة).

2.6.5 حلقات إيجاد القيم الكبرى والصغرى

للحصول على القيمة الكبرى في قائمة أو سلسلة يمكن أن نكتب الحلقة التالية:

```
largest = None
print ('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
        if largest is None or itervar > largest :
                largest = itervar
        print ('Loop:', itervar, largest)
print ('Largest:', largest)
```

وعند تنفيذ هذا البرنامج ينتج لدينا الخرج التالي:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

يمثل المتغير largest أكبر قيمة مررنا بها حتى الآن حيث قبل بداية الحلقة يحمل هذا المتغير القيمة None، يعتبر الثابت None قيمة مميزة يمكن أن نعطيها لمتغير ما لنقول عنه إنه فارغ (أي لا يحتوي أي قيمة).

قبل بداية تنفيذ الحلقة تكون القيمة None هي أكبر قيمة لأننا لم نمر بأي قيمة بعد، وفي أثناء

التنفيذ إذا كانت القيمة المخزنة في المتغير largest هي None نعتبر قيمة أول عنصر هي القيمة الأكبر، حيث نلاحظ عند تنفيذ البرنامج السابق أننا في أول تكرار للحلقة وعندما كان المتغير itervar يحمل القيمة None أصبحت قيمة largest تساوي 3.

بعد ذلك لم يعد المتغير largest يحمل القيمة None، ولذلك فإن القسم الثاني من التعبير المنطقي المركب يتحقق فقط إذا مررنا بقيمة أكبر من القيمة الحالية للمتغير largest. وعندئذ فإنها تصبح هي القيمة الأكبر والتي تخزن فيه، ويمكن أن نراقب تزايد القيمة الكبرى من 3 إلى 41 ثم إلى 74 في خرج المثال الموضح أعلاه.

عند انتهاء الحلقة نكون قد أجرينا مسحًا على كافة القيمة الموجودة في القائمة وعندها يحمل المتغير largest أكبر قيمة موجودة في القائمة.

لاستخراج القيمة الصغرى في قائمة ما نكتب حلقة مشابهة للحلقة السابقة مع تغيير بسيط:

```
smallest = None

print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
        if smallest is None or itervar < smallest:
                smallest = itervar
        print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

نقول إن المتغير smallest يحمل القيمة الصغرى الحالية قبل وأثناء وبعد تنفيذ الحلقة، وعند انتهاء الحلقة يحمل هذا المتغير أصغر قيمة موجودة في القائمة. وكما هو الحال مع حلقات الجمع والعد فإن وجود التوابع الجاهزة (()min و()max) يغني عن كتابة حلقات كهذه.

إليك نسخة مبسطة عن التابع الجاهز ()min في لغة بايثون:

```
def min(values):
    smallest = None
    for value in values:
```

```
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

لاحظ أننا قمنا بحذف أوامر الطباعة حتى نحصل على تابع مشابه للتابع الجاهز في لغة بايثون.

7.5 التنقيح

ستجد عندما تبدأ بكتابة برامج أعقد أنك تمضي وقتًا طويلًا في التنقيح، حيث أن كتابة المزيد من الشيفرات يزيد من احتمالية ارتكاب الأخطاء والأماكن التي يمكن أن تتوارى فيها.

لذا تعتبر عملية التنقيح بالتجزئة إحدى طرق تقليل الوقت المستهلك في التنقيح. فعلى سبيل المثال، إذا احتوى البرنامج على مئة سطر واختبرتها سطرًا سطرًا فستحتاج إلى مئة خطوة. لذا قسِّم المشكلة إلى نصفين عوضًا عن ذلك، وابحث في منتصف البرنامج -أو قرب المنتصف- عن قيمة وسطية يمكن التأكد منها، ثم أضف تعليمة الطباعة (أو إضافة أي تغيير يعطي أثرًا واضحًا) وقم بتشغيل البرنامج. إذا فشلت عملية التحقق التي أضفناها تكون المشكلة في النصف الأول للبرنامج، وفي حال كانت نتيجة هذه العملية واضحة فالمشكلة إذا في النصف الثاني. في كل مرة نقوم بتكرار هذه الطريقة فإننا نختصر عدد السطور التي نحتاج إلى التحقق منها إلى النصف، وبعد ست خطوات (وهو عدد اقل بكثير من مئة خطوة) فإننا سنحصر المشكلة في سطر أو سطرين فقط (نظريًّا على الأقل).

عند التطبيق العملي لهذه الطريقة، قد لا يكون من الواضح دائمًا المكان الذي يعتبر نصف البرنامج وقد لا يكون هذا الموضع قابلًا للتعيين، كما أنه من غير المعقول أن نعد الأسطر ونجد نقطة المنتصف تمامًا، فنقوم عوضًا عن ذلك بالبحث عن الأماكن التي يمكن أن تحتوي على أخطاء أو التي يسهل التحقق من نتيجتها ثم نختار نقطة تمثل المنتصف بالنسبة لهذه الأماكن.

8.5 فهرس المصطلحات

- **المراكم (accumulator)**: متغير يستخدم في الحلقة ليجمع النتائج مع بعضها.
- **العداد (counter)**: متغير يستخدم في حلقة ليقوم بعد المرات التي يحدث فيها شيء ما. يعطى هذا المتغير قيمة ابتدائية تساوي الصفر ونزيد قيمته بمقدار 1 كل مرة نعد شيئًا ما.
- **التنقيص (decrement)**: إنقاص قيمة المتغير.

- **إعطاء قيمة ابتدائية (initialize):** إسناد قيمة ابتدائية لمتغير ما سيتم تحديثه لاحقًا.
- **الزيادة (increment):** تحديث لقيمة متغير ما يسبب زيادتها (عادة بمقدار 1).
- **حلقة لا نهائية (infinite loop):** وهي حلقة لا يتحقق فيها شرط الإنهاء بتاتًا أو هي الحلقة التي يغيب عنها هذا الشرط أساسًا.

9.5 تمارين

- **التمرين الأول:** اكتب برنامجًا يقرأ الأرقام المدخلة بشكل متكرر حتى يدخل المستخدم كلمة done، وعندها يطبع البرنامج كل من المجموع والمتوسط والعدد الكلي لهذه الأرقام. إذا أدخل المستخدم أي محارف عدا الأرقام اكتشف هذا الخطأ باستخدام تعليمتي try وexpect وأظهر رسالة خطأ ثم انتقل إلى الإدخال التالي.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.333333333333333
```

- **التمرين الثاني:** اكتب برنامجًا يطلب قائمة من الأرقام كما في المثال السابق وعند النهاية يظهر القيمة الكبرى والصغرى للأرقام بدلًا عن المتوسط.

الفصل السادس

السلاسل النصية

6 السلاسل النصية

1.6 السلسلة النصية هي سلسلة من المحارف

تعدُّ السلسة النصية سلسلةً من المحارف التي يمكن الوصول إلى كلٍّ منها وصولًا منفصلًا باستخدام عامل القوس [ ].

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

تعيد التعلمية الثانيةُ المحرفَ الموجود في الموقع ذي الفهرس 1 من المتغير fruit ليُسند إلى المتغير letter.

يُدعى التعبير ما بين الأقواس [ ] بالفهرس الذي يشير إلى المحرف المرغوب وَفقًا لتسلسله (من هنا جاءت التسمية "السلسلة النصية")، لكنك قد لا تحصل على ما تتوقعه دائمًا:

```
>>> print (letter)
a
```

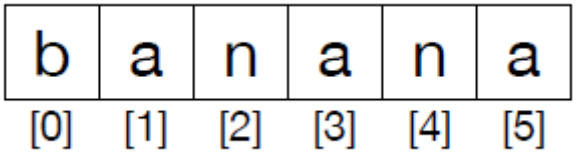
قد يكون من البديهي أن المحرفَ الأوَّل من كلمة banana هو b وليس a، بَيْدَ أنَّ قيمة الفهرس في لغة بايثون تُعبِّر عن الترتيب بدءًا من أوَّل السلسلة، وترتيب المحرف الأوَّل فيها هو الصِّفر.

```
>>> letter = fruit [0]
>>> print(letter)
b
```

نجد ممَّا سبق أن الحرف b هو الحرف الأوَّل (ذو الفهرس 0) من كلمة banana والحرف a هو الحرف الثاني (ذو الفهرس 1) وn هو الحرف الثالث (ذو الفهرس 2).

بالإمكان استخدام أي تعبير بما في ذلك من المتغيّرات والمعاملات على أنها فهرس، لكن قيمها يجب أن تكون عددًا صحيحًا وإلا فإنك ستحصل على خطأ: "خطأ في نوع البيانات: فهارس السلاسل النصية يجب أن تكون أعداد صحيحة"

```
>>> letter = fruit [1.5]
TypeError: string indices must be integers
```



الشكل 9 : فهارس السلسلة النصية

2.6 الحصول على طول السلسلة النصية باستخدام التابع len

يُعيد التابعُ len عددَ المحارف في السلسة النصية.

```
>>> fruit = 'banana'
>>> len (fruit)
6
```

قد تظن أنه يمكن كتابة التعليمات التالية للحصول على آخر محرف في السلسلة النصية:

```
>>> length = len (fruit)
>>> last = fruit [length]
IndexError: string index out of range
```

لكن سيعترضك خطأ "خطأ فهرسة: إنَّ فهرس السلسلة النصية خارج المجال".

يعود السبب في ظهور خطأ الفهرسة إلى عدم وجود محرف في كلمة banana الذي له الفهرسُ 6، وما دام أن العدّ يبدأ من الصفر؛ فالأحرفُ الستة مُفهرسة من 0 حتى 5، أي يجب طرح 1 من طول السلسلة النصية للحصول على المحرف الأخير:

```
>>> last = fruit[length-1]
>>> print(last)
a
```

وبطريقة أخرى، يمكنك استخدام الفهارس العكسية التي تُعدُّ عكسيًّا من نهاية السلسلة النصية؛ إذ إنَّ التعبيرَ fruit[-1] يُعيد الحرفَ الأخير و fruit[-2] يعيد الحرف ما قبلَ الأخير وهكذا دواليك.

3.6 التعامل مع محارف السلسة النصية باستخدام الحلقات

تطلب بعضُ البرامج معالجةَ محارف السلسلة النصية كل منها على حدى بدءًا من أوَّل محرف، حيث يُحدد المحرف ثم تُنفَّذ عملياتُ ما عليه والمتابعةُ بهذا النحْو حتى المحرف الأخير.

يُدعى هذا النَّمَط من عمليات المعالجة بالمرور على عناصر السلسلة (traversal) وتعدُّ حلقة while إحدى طرقِ تنفيذِه:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

تمر هذه الحلقة على عناصرَ السلسلة النصية وتُظهِر كلَّ محرف على سطر ظهورًا مستقِلًا. ولأن شرط هذه الحلقة هو index < len(fruit)، لذا سيختل الشرط عندما يتساوى طولُ السلسلة النصية والفهرس، فلا تنفذ تعليمات في جسم الحلقة.

آخر محرف وُصِل إليه هو الذي يملك الفهرس len(fruit)-1 الذي يدل على آخر محرف في السلسلة النصية.

التمرين الأول: استخدم حلقةَ while بحيث تبدأ من نهاية السلسلة النصية لتنتهي عند المحرف الأوَّل لها واطبع كل حرف على سطرٍ مستقِل.

حلقة for هي طريقة أخرى للمرور على عناصر السلسلة

```
for char in fruit:
    print(char)
```

يُسنَد المحرفُ الموجود في السلسلة النصية إلى المتحول char في كل دور من أدوار الحلْقة التي تستمر حتى آخر محرف في السلسلة.

4.6 تجزئة السلاسل النصية

نُسمِّي الجزءَ من السلسلة النصية بالشريحة (slice)، يشابه اختيارُ شريحة اختيارَ محرف في

السلسلة النصية

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

يعيد العامل [n,m] جزءًا من السلسلة النصية، من المحرف ذي الفهرس n إلى المحرف الذي يسبق المحرفَ ذا الفهرس m، أي يتضمن المحرف ذو الفهرس الأوَّل n ولا يتضمن ذو الفهرس الأخير m.

في حال حُذِفَ الفهرسُ n (قبلَ عامل النقطتين) فإن التجزئة ستبدأ من بداية السلسلة النصية، وفي حال حُذف الفهرسُ الثاني m (بعدَ عامل النقطتين) فإن التجزئة تستمرُّ حتى نهاية السلسلة النصية.

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

إذا كان الفهرس الأوَّل أكبرَ أو يساوي الثاني؛ فإن النتيجة هي سلسلة نصية فارغة تُمثَّل بعلامة اقتباس.

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

لا تحوي السلسلة النصية الفارغة أيَّ محارف وطولها صفر، وعلى الرغم من هذا، فهي سلسلةٌ نصية.

التمرين الثاني: بالعودة إلى السلسلة النصية fruit المُعطاة سابقًا، ما نتيجة التعليمة التالية fruit[:]؟

5.6 السلاسل النصية غير قابلة للتعديل

قد يبدو من المغري استخدامُ عامل الإسناد لتغيير محرفٍ في السلسلة النصية كما يلي:

```
>>> greeting = 'Hello, world!'
>>> greeting [0] = 'J'
TypeError: 'str' object does not support item assignment
```

فتظهر لك رسالة خطأ "خطأ تصنيف: الكائن str"،"لا يدعم إسناد العنصر".

"الكائن" في هذه الحالة هو السلسلةُ النصية و"العنصر" هو المحرف الذي حاولت أن تسندَه، يمكنك الأن اعتبار مفهوم الكائن مشابهًا تمامًا لمفهوم القيمة (ستتعرَّف هذا المفهومَ لاحقًا تعرُّفًا أفضل)، أمَّا العنصر فهو أحد القيم في سلسلة.

يظهر خطأ النوع لأن السلاسل النصية غيرُ قابلة للتعديل ممّا يعني أنك لا تستطيع تغيير سلسلة نصية، ما يمكنك القيام به هو إنشاءُ سلسلة نصية جديدة تمثِّل التغيُّر على السلسلة الأصلية.

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

أُضيفَ في هذا المثال حرفٌ جديد إلى جزء من السلسلة النصية greeting من دَون التعديل على السلسلة الأصلية.

6.6 استخدام الحلقات والعدّ

يحسب البرنامج التالي عددَ مرات ظهور المحرف "a" في السلسلة النصية

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

يُمثِّل هذا البرنامج نموذجًا لبرامج تقوم بعمليات حسابية كالعدّ، فالمتغير count يبدأ من القيمة 0 ثم يزداد في كل مرّة يظهر فيها المحرف a وعند انتهاء الحلقة يتضمن المتغير count النتيجةَ (العددَ الكليَّ لمرات ظهور المحرف a).

التمرين الثالث: أعد كتابة البرنامَج السابق في تابع سمِّه count بحيث يقبل السلسلة النصية والحرف المراد معرفةُ مرّات تَكراره باعتبارِهم وسائطَ.

7.6 العامل in

يعدُّ in عاملًا منطقيًّا يأخذ سلسلتين نصيتين ويعيد الثابت المنطقي True إذا كانت الأولى سلسلةً فرعية مِنَ الثانية.

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

8.6 مقارنة السلاسل النصية

تعمل عواملُ المقارنة على السلاسل النصية للتحقُّق من تساوي سلسلتين.

```
if word == 'banana' :
print ('All right, bananas.')
```

تُستخدم عواملُ مقارنة أخرى لترتيب مجموعة سلاسل نصية وَفقَ الترتيب الأبجدي.

```
if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

لا تتعامل لغةُ بايثون والأحرفَ الكبيرة والصغيرةَ كما يتعامل الناس معها، ففي لغة البايثون تأتي الأحرف الكبيرة قبلَ الأحرف الصغيرة دومًا، لذلك تأتي كلمةُ Pinapple قبلَ banana.

إحدى الطرائق الشائعة لتفادي هذه المشكلة هي توحيدُ نَمَط السلاسل النصية (حروف صغيرة فقط مثلًا) قبلَ القيام بعملية المقارنة، أبقِ هذه الملاحظة في بالك عند المقارنة بين كلمات ذات حروف صغيرة وكبيرة.

9.6 توابع السلاسل النصية

تعدُّ السلاسل النصية إحدى أمثلة الكائنات في لغة بايثون، يتضمن الكائن البياناتِ (السلسلة بحدّ ذاتها) وتوابع الصنف (methods) وهي توابع مبنية ضمنَ الكائن ومتاحة لأي نسخة من هذا الكائن.

يُظهر التابع dir في لغة بايثون التوابعَ المتاحة لكائن ما، في حين يُظهر التابعُ type نوعَ الكائن.

```
>>> stuff = 'Hello world'
>>> type (stuff)
<class 'str'>
>>> dir (stuff)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith' , 'expandtabs', 'find', 'format', 'format_map',
'index' , 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier' , 'islower', 'isnumeric', 'isprintable',
'isspace' , 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip' , 'maketrans', 'partition', 'replace', 'rfind',
'rindex' , 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split' , 'splitlines', 'startswith', 'strip', 'swapcase',
'title' , 'translate', 'upper', 'zfill']
>>> help (str.capitalize)
Help on method_descriptor:
capitalize(...)
```

```
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character

    have upper case and the rest lower case.

>>>
```

بالإمكان استعراضُ التوابع باستخدام التابع dir، كما يمكن استخدام help للحصول على شرح مُيسَّر عن تابع ما، أمَّا الحصولُ على ملفات تفصيلية خاصّة بتوابع السلاسل النصية، فمن الأفضل زيارة الرابط: https://docs.python.org/library/stdtypes.html#string-methods.

يشابه استدعاء تابع الصنف (method) استدعاءَ التابع العادي (function) حيث كلاهما يأخذان وسائطَ ويعيدان قيمة لكن قواعد الكتابة مُختلفة. نستدعي توابعَ الصنف عن طريق إلحاق اسمها باسمِ المتغير باستخدام عامل النقطة.

يأخذ التابع upper،-مثلًا- سلسلةً نصية ليعيد نسخة منها مكتوبةً بأحرف كبيرة؛ فبدلًا من كتابة التابع بالشكل upper(word) نكتب word.upper().

```
>>> word = 'banana'
>>> new_word = word.upper ()
>>> print(new_word)
BANANA
```

يحدِّد عاملُ النقطة اسمَ التابع upper واسمَ السلسلة النصية التي سيتعامل معها التابع word.

يشير وجود أقواس فارغة إلى أنَّ التابع لا يأخذُ وسائطَ.

يُدعى استخدامُ التابع بالاستدعاء وفي هذه الحالة يمكن القول: إنّنا نستخدم التابعَ upper على السلسلة النصية word.

فمثلًا، لدينا تابع لسلسلة نصية يدعى find يبحث عن موقع سلسلة نصية محدَّد في سلسلة نصية أخرى.

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
```

```
1
```

استخدمنا-في المثالِ السابق- التابعَ find على السلسلة النصية word وأدخلنا المحرفَ الذي نبحث عنه على أنَّه وسيط. يمكن لهذا التابع العثور على سلاسلَ فرعية أو محرفٍ ما.

```
>>> word.find('na')
2
```

قد يأخذ وسيطًا آخر يعبر عن الفهرس الذي يجب أن يبدأ عنده البحث.

```
>>> word.find('na', 3)
4
```

نستخدم التابعَ strip للتخلص من المسافات البيضاء (مسافات فارغة، إزاحة باستخدام المفتاح tap، محارف السطور الجديدة) من بداية السلسلة النصية أو نهايتها.

```
>>> line = '    Here we go    '
>>> line.strip()
'Here we go'
```

تعيد بعض التوابع، مثلُ startswith قيمًا منطقية.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True
>>> line.startswith('h')
False
```

ستلحظُ أن التابع startswith يحتاج إلى معامل لمطابقته، لذا من الأحسن تحويلُ سلسة نصية إلى أحرف صغيرة باستخدام التابع lower قبلَ القيام بأي عمليات مطابقة.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False
>>> line.lower()
```

```
'have a nice day'
>>> line.lower().startswith('h')
True
```

في المثال الأخير يُستدعى التابعُ lower، فنستخدم startswith للتحقق من أن السلسة النصية الناتجة تبدأ بمحرف"h"، وبالإمكان القيام بعددٍ من الاستدعاءات للتوابع في تعبيرٍ برمجيٍّ واحد مع أخذِ الترتيب بالحسبان.

التمرين الرابع: يوجد تابع صنف يُدعى count مشابهٌ تمامًا لما قمنا به في التمارين السابقة، وتتوفَّر معلومات عن هذا التابع في الرابط التالي:

https://docs.python.org/library/stdtypes.html#string-methods

اكتب شيفرة برمجية لعدِّ مرَّات ظهور الحرف "a" في كلمة "banana" باستدعاء هذا التابع.

10.6 تحليل السلاسل النصية

قد نحتاج أحيانًا إلى البحث عن سلسلة فرعية ضمنَ السلسلة النصية، على سبيل المثال: في حال لدينا مجموعة من الأسطر كالتالية:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

ونريد استخراجَ النصف الثاني من عنوان البريد الالكتروني المعطى، أي (uct.ac.za) من كل سطر: يمكننا القيام بذلك من خلال استخدام التابع find وتجزئة السلسلة النصية.

نبحث بدايةً عن موقع علامة @ في السلسلة النصية ثُم نحدد موقع أوَّل مسافة فارغة بعدَ علامة @ ومن ثَم نجزئ السلسلة النصية لاقتطاع الجزء المطلوب من السلسلة.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
```

```
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```

يسمح هذا الإصدارُ من التابع find بتحديد الموقع في السلسلة النصية الذي نريد منه بدءَ البحث. اقتطعنا بعملية التجزئة السابقة المحارفَ بدءًا من المحرف الذي يلي إشارة @ حتى المحرفِ الذي يسبق المسافة الفارغة.

لمزيد من المعلومات عنِ التابع find، بالإمكان زيارة الرابط التالي:

https://docs.python.org/library/stdtypes.html#string-methods

11.6 عامل التنسيق

يتيح عاملُ التنسيق % بناءَ سلاسلَ نصية واستبدال أجزاء منها، ببيانات مخزنة في المتغيرات. يُمثِل الرمزُ % عند استعماله مع الأعداد الصحيحة عاملَ باقي القسمة لكن عندما يكون المعامل الأوَّل سلسلةً نصية يكون عامل تنسيق.

يضمّ المعامل الأوَّل -وهو سلسلة نصية- رموزًا مختصّة لتحدد كيفية تنسيق المعامل الثاني حيثُ إنَّ نتيجة هذه العملية هي سلسلةٌ نصية؛ فمثلًا يشير رمز التنسيق %d إلى أن المعامل الثاني يجب أن يُنَسَّق باعتباره عددًا صحيحًا (d اختصارًا لـ decimal).

```
>>> camels = 42
>>> '%d' % camels
'42'
```

ينتج ممَّا سبق السلسلة النصية '42' ويجب ألَّا يُخلَط بينها وبين العدد الصحيح 42.

يمكن أن تظهر رموز التنسيق في أيِّ مكان من السلسلة حيثُ يمكّنك ذلك من إضافة جملة معينة.

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels. '
```

في حال وجود أكثرَ من رمز تنسيق في السلسلة النصية؛ فالوسيطُ الثاني يجب أن يكون من نوع البيانات صف (Tuple). كلُّ رمز تنسيق مرتبطٌ بعنصر من الصف حَسَبَ الطلب.

يستخدم المثالُ التالي %d لتنسيق عدد صحيح و%g لتنسيق رقم ذي فاصلة عشْرية و%s لتنسيق سلسلة نصية.

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')

'In 3 years I have spotted 0.1 camels. ′
```

يجب أن يساوي عددُ العناصر في الصف عددَ رموز التنسيق في السلسلة النصية، كما يجب أن يرتبطَ نوع العناصر بتسلسل التنسيق.

```
>>> '%d %d %d' % (1, 2)

TypeError: not enough arguments for format string
```

لا توجد وسائطُ كافيةٌ لتنسيق السلسلة.

```
>>> '%d' % 'dollars'

TypeError: %d format: a number is required, not str
```

رمز التنسيق %d يتطلب رقْمًا وليس سلسلة نصية.

لا يوجد في المثال الأوَّل عددُ عناصر كافٍ ونوع العنصر في الثاني خطأٌ.

عامل التنسيق قويٌّ ومفيد لكنة صعب الاستخدام، بالإمكان قراءة المزيد عنه من خلال الرابط التالي:

https://docs.python.org/library/stdtypes.html#printf-style-string-formatting

12.6 التنقيح

المهارة التي ينبغي لك صقلُها في داخلك-من حيثُ إنكَ مُبرمج- هي أن تسأل نفسك دومًا: "ما الخطأ الذي يمكن أن يحصل هنا أو بالأحرى ما هي الأشياء التي يمكن للمستخدم فعلُها لتفشل برامجُنا التي تبدو مثالية؟".

على سبيل المثال، لنعد إلى البرنامَج الذي استخدمناه لشرح حلقة while في فصل التَّكرار:

```
while True:
    line = input('> ')
```

```
    if line[0] == '#':
        continue
    if line == 'done':
        break
     print(line)
print('Done!')
# Code: http://www.py4e.com/code3/copytildone2.py
```

انتبه ما الذي يحصل عندما يُدخل المستخدم سطرًا فارغًا.

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

تظهر رسالة خطأ "خطأ في الفهرسة: فهرس السلسلة النصية خارج المجال"

سيعمل البرنامج عملًا سليمًا حتى يُدخل سطرٌ فارغ، في هذه الحالة لا يوجد محرف في الموقع [0] لذلك نحصل على تقرير بالخطأ، ثمة حلان لهذه المشكلة.

أحد هذه الحلول هي بسهولة استخدامُ التابع startswith الذي يعيد الثابت المنطقي False في حال كانت السلسلة النصية فارغة.

```
if line.startswith('#'):
```

الطريقة الأخرى أكثرُ أمانًا وهي باستخدام عبارة if الشرطية مع استخدام تعليمات تفادي الأخطاء

باستخدام شرطين بحيثُ لا يُتحقَّق من الشرط الثاني إلا في حال تحقَّق الأوَّل وهو وجودُ محرف واحد على الأقل.

```
if len(line) > 0 and line[0] == '#':
```

13.6 فهرس المصطلحات

- **العداد (counter):** هو متغيِّر لعد شيء ما، عادةً ما يبدأ من الصفر وتزداد قيمته.
- **سلسلة نصية فارغة (empty string):** هي سلسلة نصية دون أي محارف وطولها 0، تُمثَّل باستخدام علامتي الاقتباس.
- **عامل التنسيق (format operator):** هو العامل % يطلُب رموزَ التنسيق وَصف لتوليد سلسلة نصية تتضمن عناصر الصفّ مُنسّقة على وَفق رموز التنسيق.
- **رموز التنسيق (format sequence):** سلسلة من المحارف، مثلَ: %d تحدد كيف تُنَسَّق قيمةٌ معينة.
- **سلسلة التنسيق (format string):** سلسلة نصية تُستخدم مع عامل التنسيق بحيثُ تتضمن رموزَ تنسيق.
- **العلَم** (flag): متغير منطقيٌّ يشير فيما إذا كان الشرط محقَّق أو غير محقَّق.
- **استدعاء (invocation):** عبارة نستدعي من خلالها تابع الصنف.
- **غير قابل للتعديل (Immutable):** ميزةٌ للسلاسل بحيث لا يمكن تعديلُ عناصرها.
- **الفهرس (index):** عددٌ صحيح يُستخدم لتحديد عنصر في سلسلة مثل محرف ضمن سلسلة نصية.
- **عنصر (item):** أحدُ القيم في سلسلة ما.
- **تابع الصنف (method):** تابع مرتبِطٌ بكائن ويُستدعى باستخدام عامل النقطة.
- **الكائن (object):** شيءٌ يمكن للمتحول أن يشير إليه، حتى الآن يمكنك عدُّ "الكائن" و"القيمة" الشيءَ نفسَه.
- **البحث (search):** شكلٌ من أشكال المرور على عناصر سلسلة بحيث يتوقف عندما يجد ما

يبحثُ عنه.

- **سلسلة (sequence):** مجموعةٌ مرتبةٌ من القيم بحيث كلُّ قيمة معرفة باستخدام فهرس.
- **شريحة (slice):** جزءٌ من السلسلة النصية المحدَّد بعدد من الفهارس.
- **المرور على عناصر السلسة (traverse):** المرور على عناصر سلسلة وإجراء عمليات مماثلة في كلِّ مرة.

14.6 تمارين

- **التمرين الخامس:** الشيفرة البرمجية التالية مكتوبة بلغة بايثون تخزن سلسلة نصية:

```
str = 'X-DSPAM-Confidence:0.8475'
```

استخدم تعليمة find وتجزئة السلاسل النصية لاقتطاع الجزء من السلسلة الواقع بعدَ النقطتين ثُم استخدم التابع float لتحويل السلسلة المُقتطعة إلى عددٍ ذي فاصلة عشرية.

- **التمرين السادس:** اقرأ توصيف توابع السلسلة النصية عبر زيارة الرابط:

https://docs.python.org/library/stdtypes.html#string-methods

من المفيد التعامل مع أحدها للتأكد من فهمِها وفهم كيف عملها. مثل strip وreplace فهما مفيدان جدًّا.

قد يعترضك أثناء قراءة التوصيف جملًا قد تكون غير مفهومة، مثلًا:

```
in find (sub[, start[, end]])
```

تشير الأقواس إلى وسائطَ اختيارية، أي أن الوسيط sub مطلوبٌ لكن start اختيارية، وفي حال ضُمنت start تكون end اختيارية.

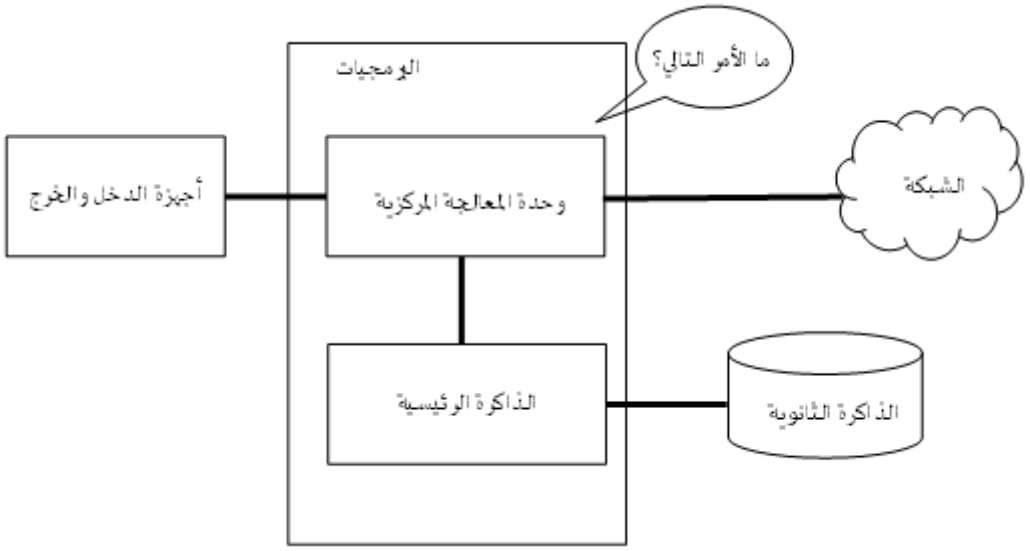
الفصل السابع

المِلفات

7 المِلفات

1.7 الإصرار على التعلم

تعلّمنا حتى الآن كيفية كتابة البرامج وتنفيذ ما نريده عبر وحدة المعالجة المركزية باستخدام التنفيذ المشروط والتوابع والتَّكرار. كما تعلمنا كيفية إنشاء بنى البيانات (data structures) واستخدامِها في الذاكرة الرئيسة. حيث يُخزن البرنامَج ويُنفذ في وحدة المعالجة المركزية والذاكرة ويُعتبران المكان الذي يحدث فيه "التفكير". ولكن إذا كنتَ تتذكر نقاشنا عن بنية الحاسب، فبمجرد فصل التغذية الكهربائية عن الحاسوب، يُحذف أي شيء مخزن في وحدة المعالجة المركزية أو في الذاكرة الرئيسية.



الشكل 10 :الذاكرة الثانوية

سنتعرَّف في هذا الفصل إلى وسيط تخزين جديد يُسمى الذاكرة الثانوية (أو المِلفات). تمتاز الذاكرة الثانوية بعدم فقدانها محتوياتها عند فقدان الطاقة. أو في حالة وحدة تخزين متنقلة (USB Flash)، يمكن إزالة البيانات-التي نكتبها من برامجنا- من النظام ونقلها إلى نظام آخرَ.

سنركز تركيزًا أساسيًّا على قراءة المِلفات النصية وكتابتِها، مثلَ تلك التي ننشئها في محرر النصوص. سنرى لاحقًا كيفية العمل مع مِلفات قواعد البيانات وهي مِلفات ثنائية (Binary)، مصمَّمة خصيصًا للقراءة والكتابة من خلال برمجيات مَعنية بقواعد البيانات.

2.7 فتح المِلفات

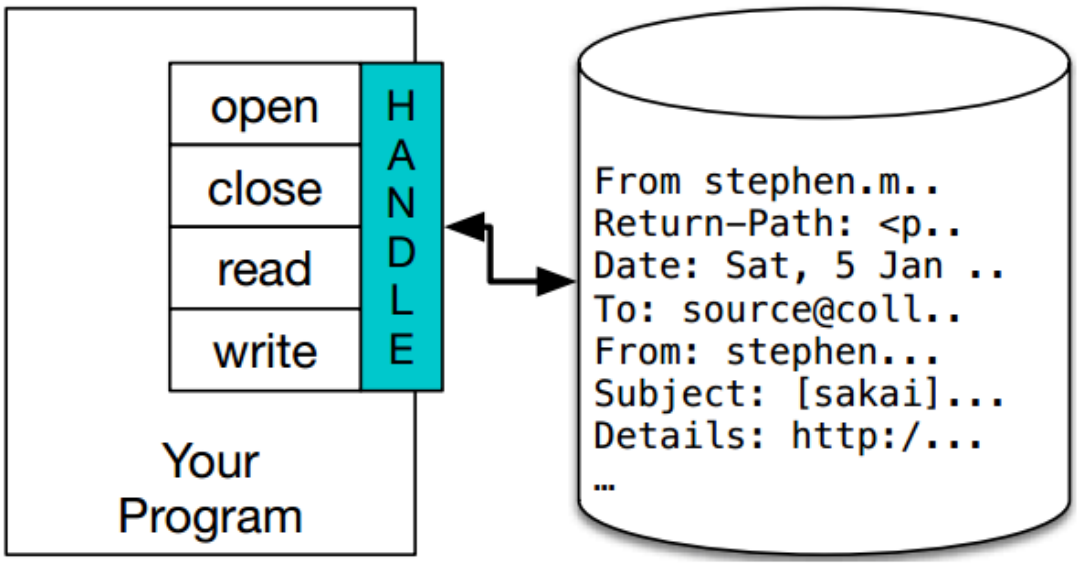
عندما نريد قراءة مِلف أو الكتابة عليه -على سبيل المثال: مِلف محفوظ على محرك الأقراص

الصلبة-، يجب أوّلًا فتح المِلف. يؤدي فتح المِلف إلى الاتصال بنظامِ تشغيلك، الذي يعرف مكان تخزين البيانات الخاصة بكل مِلف. عندما تفتح مِلفًا، فإنك تطلب من نظام التشغيل العثور على المِلف بالاسم والتأكد من وجوده. يوضح المثال أدناه طريقة فتح المِلف النصي mbox.txt، الذي يجب تخزينه في المجلد نفسه الذي استخدمته عند بدء تشغيل بايثون. يمكنك تنزيل هذا المِلف من خلال الضغط على الرابط التالي:

www.py4e.com/code3/mbox.txt

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

إذا فُتح المِلف بنجاحٍ، فسيعيد لنا نظام التشغيل معرّفًا للمِلف (file handle). لا يمثل المعرّف البيانات الفعلية الموجودة في المِلف، ولكنه بدلًا من ذلك يكون واجهةً يمكننا استخدامها لقراءة البيانات. تُمنَح معرّفًا للمِلف إذا كان المِلف المطلوب موجودًا ولديك الأذونات المطلوبة لقراءة المِلف.



الشكل 11: معرف الملف

إذا لم يكن المِلف موجودًا، فستفشل عملية الفتح وسيُعرض في الشاشة تقرير بالخطأ ولن تحصل على معرّف للوصول إلى محتويات المِلف كما يوضح المثال التالي:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

سوف نستخدم لاحقًا خاصية التعامل مع الاستثناءات try/except للتعامل بأمان أكثرَ مع الحالات التي نحاول فيه فتح مِلف غيرِ موجود.

3.7 مِلفات النصوص والأسطر

يمكن عدّ المِلف النصي على أنه سلسلة من الأسطر، مثلُ السلسلة النصية في لغة بايثون التي يمكن اعتبارها سلسلة من المحارف. مثلًا، هذه عينة من مِلف نصي يسجل نشاط البريد من أفراد مختلفين في فريق تطوير مشروعٍ مفتوحِ المصدر.

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

يمكنك الحصول على مِلف نشاطات البريد الإلكتروني كاملًا من خلال الضغط على الرابط التالي:

www.py4e.com/code3/mbox.txt

كما يمكنك الحصولُ على النسخة المختصرة من هذا المِلف من خلال الضغط على الرابط التالي:

www.py4e.com/code3/mbox-short.txt

هذه المِلفات ذات صيغة قياسية؛ إذ يحتوي المِلف على عدد من رسائل بريدية. والأسطر التي تبدأ بكلمة "From" تفصل بين الرسائل المختلفة. الأسطر التي تبدأ بكلمة "From" تُعد جزءًا من الرسائل. لمعرفة المزيد من المعلومات عن صيغة رسائل البريد، بالإمكان زيارة الرابط التالي:

https://en.wikipedia.org/wiki/Mbox

لتقسيم المِلف إلى عدد من الأسطر، يُستخدم محرفٌ خاص يُمثِل "نهاية السطر" ويطلق عليه اسم

محرف السطر الجديد (newline).

تستخدم في لغة باثيون شرطةٌ مائلة عكسية (backslash) مع حرف n في السلاسل النصية لإنشاء سطر جديد \n . على الرغم من أن محرف إنشاء سطر جديد يبدو محرفين، فهو في الواقع محرفٌ واحد. عندما نكتب المتغير stuff في مفسر لغة بايثون، فإنه يظهر \n في السلسلة النصية الناتجة، ولكن عندما نستخدم print لإظهار السلسلة، نرى السلسلة مقسَّمة إلى سطرين بواسطة محرف السطر الجديد. المثال التالي يوضح كيفية إنشاء سطر جديد في لغة بايثون:

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

يجدر بك أيضًا ملاحظةُ أن طول السلسلة X\nY هو ثلاثة؛ لأنَّ حرف السطر الجديد \n يُعتبر محرفًا واحدًا.

لذلك عندما ننظر إلى السطور في مِلف ما، علينا أن نتخيل أن ثمة محرفًا خاصًّا غيرَ مرئي يُسمى محرف السطر الجديد في نهاية كل سطر يُمثِّل نهاية السطر. أي أن محرف السطر الجديد يفصل المحارف في المِلف إلى سطور.

4.7 قراءة المِلفات

على الرغم من أن معرّف المِلف لا يحتوي على بيانات المِلف، فإنَّ من السهل جدًّا إنشاءَ حلقة for لقراءة كل سطر وعدِّهِ، من سطور المِلف:

```
fhand = open('mbox-short.txt')
count = 0
```

```
for line in fhand:
    count = count + 1
print('Line Count:', count)


# Code: http://www.py4e.com/code3/open.py
```

يمكننا استخدام معرّف المِلف سلسلةً للتَّكرار في حلقة for حيث تحسب حلْقة for بيُسرٍ عددَ الأسطر في المِلف وتطبعها. بمعنى آخر، يمكن تلخيص عمل حلقة for كالتالي: لكل سطر في المِلف الممثل بمعرّف المِلف، أضف واحدًا إلى المتغير count.

يعود السبب في أن تابع فتح المِلفات open لا يقرأ المِلف بالكامل إلى أن المِلف قد يكون كبيرًا جدًّا وقد يصل حجمُه إلى أكثر من واحد غيغابايت. تستغرق تعليمة open نفسَ القدر من الوقت بغض النظر عن حجم المِلف وتعمل حلقة for على قراءة البيانات من المِلف.

حين يُقرأُ المِلف باستخدام حلقة for بهذه الطريقة؛ تُقسِّم لغة بايثون البياناتِ الموجودةَ في المِلف إلى أسطر منفصلة باستخدام محرف إنشاء السطر الجديد \n. تقرأ لغة بايثون كل سطر عن طريق محرف إنشاء السطر الجديد وتضيف هذا المحرفَ على أنه محرفٌ أخير في المتغير line لكل تَكرار للحلقة for.

نظرًا إلى أن حلقة for تقرأ البيانات سطرًا واحدًا في كل مرة، فمن ثَم يمكنها قراءة الأسطر وحسابُها بكفاءة في المِلفات الكبيرة جدًّا دون نفاد الذاكرة الرئيسة لتخزين البيانات. يمكن للبرنامَج أعلاه حسابُ الأسطر في أي مِلف بأي حجم باستخدام ذاكرة ذات حجم صغير جدًّا؛ إذ يُقرأ كل سطر ويعد ثم نتخلص منه.

إذا كنت تعلم أن المِلف صغير نسبيًّا موازنة بحجم ذاكرتك الرئيسة، فيمكنك قراءة المِلف بأكمله في سلسلة واحدة باستخدام تابع القراءة read.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

في المثال أعلاه، قُرئت محتويات المِلف mbox-short.txt بالكامل -94626 محرفًا- مباشرةً في المتغير inp. كما يوضح المثال استخدام تعليمة تجزئة السلاسل النصية لطباعة أوَّل 20 حرفًا من بيانات السلسلة المخزنة في المتغير inp.

عند قراءة المِلف بهذه الطريقة، فإن جميع المحارف، بما في ذلك جميع الأسطر ومحارف إنشاء السطر الجديد، تُعتبر سلسلة واحدة كبيرة ضمن المتغير inp. تجدر الإشارة إلى أنه من الجيد تخزين ناتج القراءة ضمن متغير وذلك لأن كل استدعاء للقراءة يستنفد المورد وهذا ما يوضّحه المثال التالي:

```
>>> fhand = open('mbox-short.txt')
>>> print(len(fhand.read()))
94626
>>> print(len(fhand.read()))
0
```

كما عليك أن تضع في الحسبان أنه يجب استخدام هذه الصيغة لتابع فتح المِلفات open فقط إذا كانت بيانات المِلف مناسبة بشكل مريح للذاكرة الرئيسة لحاسوبك. أمَّا إذا كان المِلف كبيرًا جدًّا بحيث لا يتسع للذاكرة الرئيسة، فيجب عليك كتابة البرنامَج لقراءة محتويات المِلف في أجزاء باستخدام حلْقة for أو while.

5.7 البحث خلال مِلف

عندما تبحث عن بيانات في مِلف، من الشائع جدًّا تجاهل معظم السطور والتركيز في معالجة الأسطر التي تفي بشرط مُعين فقط. يمكننا دمج نَمط قراءة مِلف مع التوابع المستخدمة مع السلاسل النصية لإنشاء آليات بحث سهلة.

على سبيل المثال، إذا أردنا قراءة مِلف وطباعة الأسطر التي بدأت بالبادئة "From:" فقط، فيمكننا استخدام التابع startswith لتحديد تلك الأسطر التي تحتوي على البادئة المطلوبة فقط كما في المثال التالي:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    if line.startswith('From: '):
        print(line)
```

```
# Code: http://www.py4e.com/code3/search1.py
```

عندما يُنفذ هذا البرنامَج، نحصل على المخرجات التالية:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

تبدو هذه المخرجات رائعة لأن الأسطر الوحيدة التي نراها هي تلك التي تبدأ بـ from:، ولكن لماذا نرى الأسطر الفارغة الإضافية؟ يُعزى ذلك إلى استخدام محرف إنشاء السطر الجديد الذي لا يُطبَع بل يظهر تأثيره فحَسْب. ينتهي كل سطر بمحرف إنشاء السطر الجديد، لذا فإن تعليمة الطباعة التي تطبع السلسلة في المتغير line الذي يتضمن محرف سطر جديد ثم تضيف تعليمة الطباعة سطرًا جديدًا آخر، مما يؤدي إلى وجود تباعد أو مسافة مزدوجة بين الأسطر.

يمكننا استخدام طريقة تجزئة الأسطر لطباعة كل المحارف ما عدا المحرفَ الأخير، ولكن الطريقة الأنسب هي استخدام التابع rstrip الذي يحذف المسافات البيضاء (white spaces) الواقعة بين الأسطر كما في المثال التالي:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From: '):
        print(line)
# Code: http://www.py4e.com/code3/search2.py
```

وعند تنفيذ هذا البرنامَج، نحصل على المخرجات التالية:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

```
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
…
```

نظرًا إلى أن برامج معالجة مِلفاتك، تصبح أكثر تعقيدًا، فقد ترغب في تنظيم حلقات البحث باستخدام التعليمة continue. الفكرة الأساسية لحلقة البحث هي أنك تبحث عن الأسطر "المهمّة" وتتخطى الأسطر "غير المهمّة". ثم عندما نجد سطرًا مثيرًا للاهتمام؛ نفعل شيئًا ما به. يمكننا هيكلة الحلْقة لتتبع نمَط تخطي السطور غيرِ المهمة على النحو التالي:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:'):
        continue
    # Process our 'interesting' line
    print(line)
# Code: http://www.py4e.com/code3/search3.py
```

عند تنفيذ هذا البرنامَج؛ ستحصل على نفس المخرجات السابقة. بمعنى آخر، الأسطر غيرُ المهمة هي تلك الأسطر التي لا تبدأ بـ From: ، وهي التي نتخطاها باستخدام التعليمة continue . فيما يعالج السطور "المُهمة" (أي تلك التي تبدأ بـ From:). يمكننا استخدام التابع find لمحاكاة أداة البحث في محرِّر نصوص التي تعثر على السطور حيث تكون سلسلة البحث في أي جزأ من السطر. نظرًا إلى أن تعليمة find تبحث عن تواجد سلسلة داخل سلسلة أخرى وتقوم إمَّا بإرجاع موضع السلسلة وإمَّا طباعة -1. إذا لم تُعثر على السلسلة، فيمكننا كتابة الحلقة التالية لإظهار الأسطر التي تحتوي على السلسلة "@uct.ac.za" (أي أنهم ينتمون إلى جامعة كيب تاون في جنوب إفريقيا) كما في المثال التالي:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find ('@uct.ac.za') == -1: continue
    print(line)
# Code: http://www.py4e.com/code3/search4.py
```

عند تنفيذ هذا البرنامَج، نحصل على المخرجات التالية وهي عناوين البريد الإلكتروني لمُنتمين إلى جامعة كيب تاون في جنوب أفريقيا:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan 4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za

…
```

نستخدم هنا أيضًا الشكلَ المختصر لجملة if الشرطية، إذ نضع تعليمة continue على نفس السطر برُفقة تعليمة if الشرطية. يعمل هذا الشكل من جملة if الشرطية كما لو كانت تعليمة continue في السطر التالي ومسبوقة بمسافة بادئة.

6.7 السماح للمستخدم باختيار المِلف

لا نريد أبدًا أن نُضطرَّ إلى تعديل شيفرة لغة بايثون في كل مرة نريد فيها معالجة مِلف مختلف. سيكون من الأفضل أن تطلب من المستخدم إدخالَ اسم المِلف في كل مرة يتم فيها تشغيل البرنامَج حتى يتمكن من استخدام برنامَجنا على مِلفات مختلفة دون تغيير الشيفرة. يمكن تنفيذ هذا الأمر بسهولة من خلال قراءة اسم المِلف من المستخدم باستخدام التابع input على النحو التالي:

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)
# Code: http://www.py4e.com/code3/search6.py
```

نلحظُ من البرنامَج الأعلى أن اسم المِلف يدخله المستخدم. ونضعه في متغير يُسمى fname ونفتح هذا المِلف. الآن يمكننا تشغيل البرنامَج تشغيلًا متكررًا على مِلفات مختلفة كما هو موضح أدناه:

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

قبل إلقاء نظرة خاطفة إلى القسم التالي، ألق نظرة إلى البرنامَج الأعلى واسأل نفسك،«ما الخطأ المحتملُ هنا؟» أو «ما الذي يمكن أن يفعله مستخدمنا الودود ليجعل برنامَجنا الصغير اللطيف لا يُنفذ بل تظهر رسالة خطأ بدلًا من ذلك، مما يجعلنا نبدو مبرمجين غير محترفين في أعين مستخدمينا؟».

7.7 استخدام try وexcept وopen

أخبرتك للتو بألا تختلس النظر إلى هذا القسم قبل الإجابة على الاسئلة السابقة، هذه هي فرصتك الأخيرة.

ماذا لو كتب مستخدمنا شيئًا ما غير اسم المِلف؟ تمعّن في حالات التنفيذ التالية، ما الذي تلحظه؟

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):

  File "search6.py", line 2, in <module>
```

```
    fhand = open(fname)

FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

حسنًا، لا تضحك. سيفعل المستخدمون في النهاية كل ما يمكنهم فعله لتفشيل برنامَجك، إمَّا عن قصد وإمَّا بنيةِ سيئة. في واقع الأمر، إنَّ أي فريق تطوير برمجيات يجب أن يتضمن شخصًا أو فريقًا مسؤولًا عما يُسمى بضمان الجودة، وتتمثل مهمة هذا الفريق في القيام بأكثر الأشياء جنونًا في محاولة لكسر البرنامَج الذي أنشأه المُبرمج أو فريق البرمجة. يعتبر فريق ضمان الجودة مسؤولًا عن اكتشاف العيوب في البرامج قبل تسلُّم البرنامَج للمستخدمين الذين قد يشترون البرنامَج أو يدفعون رواتب المبرمجين. لذا، فريق ضمان الجودة هو أفضلُ صديق للمبرمج.

والآن بعد أن رأينا الخلل في البرنامَج، يمكننا إصلاحه بأناقة باستخدام بنية try / except. نحتاج إلى افتراض أن عملية استدعاء التابع open قد تُخفق، لذا سنقوم بإضافة شيفرة استعادة (recovery code) عند فشل عملية استدعاء التابع open على النحو التالي:

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search7.py
```

يعمل التابع exit على إنهاء البرنامَج؛ إذ نستدعي هذا التابعَ ولا يعود بأي قيمة. الآن عندما يكتب المستخدم (أو فريق ضمان الجودة) أسماء للمِلفات غير أسمائها الحقيقية، فإننا "نستدرك الوضع" بأمان كما هو موضح أدناه:

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt


python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

تعد حماية استدعاء التابع open مثالًا جيّدًا على الاستخدام الصحيح لبنية try و except في البرامج المكتوبة بلغة بايثون. نستخدم مصطلح "بايثونيّ" "Pythonic" عندما نفعل شيئًا بأسلوب محترف في لغة بايثون. يمكننا القول: إن المثال السابق هو "طريقة بايثونية" لفتح مِلف.

بمجرد أن تصبح أكثر مهارةً في لغة بايثون، يمكنك المشاركة في اقتراح حل بديل مع مبرمجي بايثون الآخرين لتحديد أي من الحلين المتكافئين لمشكلة ما هو "أكثر بايثونية". الهدف من أن تكون "أكثر بايثونية" يجسد فكرة أن البرمجة جزءٌ من الهندسة وجزء من الفن. لسنا مهتمين دائمًا فقط بإنجاح شيء ما، بل نريد أيضًا أن يكون حلنا أنيقًا وأن يحظى بتقدير أقراننا على أناقته.

8.7 كتابة المِلفات

لكتابة مِلف، عليك فتحه باستخدام الوضع "w" كمعامل ثانٍ للتابع open كما في المثال التالي:

```
>>> fout = open ('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

إذا كان المِلف موجودًا بالفعل، فإن فتحه في وضع الكتابة يؤدي إلى مسح البيانات القديمة ويبدأ من جديد، لذا كن حذرًا! أما إذا كان المِلف غير موجود، فسيُنشَأ مِلفٌ جديد.

يعمل تابع الكتابة write الخاص بكائن معرف المِلف على وضع البيانات في المِلف وإرجاع عدد الأحرف المكتوبة كما نلحظ في المثال الأدنى؛ إذ أُرجعتِ القيمةُ 24 التي تمثِّل عددَ الحروف الموجودة في السلسلة النصية المجودة بين علامتي التنصيص " ". إنّ الوضع الافتراضي هو مِلف نصي في حالتي كتابة السلاسل النصية وقراءتها.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

```
24
```

مرة أخرى، يحفظ كائن المِلف مكانه، لذلك إذا استدعيت تابع الكتابة write مرة أخرى، فإن البيانات الجديدة ستُضاف إلى النهاية.

يجب أن نتأكد من إدارة نهايات الأسطر في أثناء الكتابة على المِلف عن طريق إدراج حرف إنشاء السطر الجديد \n إدراجًا صريحًا آنَ نُريد إنهاء السطر. من هنا ينبغي أن نعرف أن تعليمة print تُضيف تلقائيًّا سطرًا جديدًا، لكن استعمال التابع write لا يضيف السطر الجديد تلقائيًّا.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

عند الانتهاء من عملية الكتابة، يجب عليك إغلاق المِلف كما في الشيفرة أدناه للتأكد من كتابة آخر جزء من البيانات فعليًا على القرص حتى لا يضيع هذا الجزء إذا انقطع التيار الكهربائي.

```
>>> fout.close()
```

يمكننا إغلاق المِلفات التي نفتحها للقراءة أيضًا، ولكن يمكن أن نكون مهملين بعض الشيء إذا كنا نفتح بعض المِلفات فقط لأن مُفسر بايثون يغلق جميع المِلفات المفتوحة عند انتهاء البرنامَج. أمّا عندما نكتب على المِلفات، فيجب علينا إغلاق المِلفات إغلاقًا قاطعًا وذلك تفاديًا من أي شيء قد يحدث.

9.7 التنقيح

عند قراءة المِلفات أو الكتابة عليها، قد تواجه مشكلات في الفراغات أو ما يسمى المسافات البيضاء. قد يكون من الصعب تصحيح هذه الأخطاء لأن المسافات والإزاحات والأسطر الجديدة عادة ما تكون غيرَ مرئية كما في المثال التالي:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2      3
 4
```

يمكن أن يساعد التابع الجاهز repr في حل هذه المشكلات؛ إذ يأخذ أي كائن كوسيط ويعيد سلسلة نصية تمثِّل الكائن. فيما يخص السلاسل النصية، إن ذلك التابع يُمثل رموز المسافات البيضاء

بسلسلة من الشرطات العكسية:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

يمكن أن يكون التنقيح مفيدًا، ولكن ثمة مشكلة أخرى قد تواجهها، وهي أن الأنظمة المختلفة تستخدم محارفًا مختلفة للإشارة إلى نهاية السطر؛ إذ تستخدم بعض الأنظمة الرمز \n لإنشاء سطر جديد، في حين تستخدم أنظمة أخرى الرمز \r، وتستخدم بعض الأنظمة كلا الرمزين. عند نقل المِلفات بين أنظمة مختلفة، فقد تتسبب هذه التناقضات في حدوث مشكلات.

فيما يخصُّ معظم الأنظمة، ثمة تطبيقات للتحويل من تنسيق إلى آخر. يمكنك العثور عليها (وقراءة المزيد عن هذه المشكلة) على الرابط: https://www.wikipedia.org/wiki/Newline.

أو، بالطبع، يمكنك إنشاء تطبيق بنفسك.

10.7 فهرس المصطلحات

- **استدراك الاستثناء (catch):** طريقة تُستخدم لمنع استثناء من إنهاء البرنامَج باستخدام عبارات tryوexcept.
- **محرفُ إنشاء السطر الجديد (newline):** محرف مَعنيٌّ يستخدم في المِلفات والسلاسل النصية للإشارة إلى نهاية السطر.
- **البايثونية (Pythonic):** هي إسلوب برمجي خاص بلغة بايثون، على سبيل المثال "استخدام try وexcept هي طريقة بايثونية في حالة كانت المِلفات التي اسُتدعيت في البرنامَج مفقودة".
- **ضمان الجودة (Quality Assurance):** هو شخصٌ أو فريق يتركز عمله على ضمان الجودة الشاملة لمنتج البرنامَج وغالبًا ما يشارك فريق ضمان الجودة في اختبار المنتج وتحديد المشكلات في البرنامَج قبل طرحه للبيع.
- **مِلف نصي (Text File):** عبارة عن سلسلة من الأحرف المخزنة بوحدة تخزين دائم مثلَ القرص الصلب.

11.7 تمارين

- التمرين الأول: اكتب برنامَجًا لقراءة مِلف وطباعة محتوياته (سطرًا بسطر) كلها بأحرف كبيرة بحيث يبدو تنفيذ البرنامَج على النحو التالي:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN 5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
        BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
        SAT, 05 JAN 2008 09:14:16 -0500
```

بإمكانك تنزيل المِلف من خلال الرابط التالي:

www.py4e.com/code3/mbox-short.txt

- **التمرين الثاني:** اكتب برنامَجًا لمطالبة المستخدم باسم مِلف، ثم اقرأ محتويات المِلف وابحث عن السطور التي تحتوي على الصيغة التالية: `X-DSPAM-Confidence: 0.8475`

وحينما تصادف سطرًا يبدأ بـ `"X-DSPAM-Confidence:"`، افصل السطر لاستخراج الرقْم ذي الفاصلة العشْرية من السطر.

واحسب عدد هذه السطور ثم احسب إجمالي القيم ذات الفواصل العشرية في هذه السطور (Average spam confidence) وعندما تصل إلى نهاية المِلف، اطبع متوسطَهم .

لاحظ أن تنفيذ البرنامَج سيبدو على الشكل التالي:

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

اختبر برنامَجك على مِلف `mbox.txt` ومِلف `mbox-short.txt`.

- **التمرين الثالث**: عندما يشعر المبرمجون أحيانًا بالملل أو يرغبون في الحصول على القليل من المرح، فإنهم يضيفون بعض المفاجئات إلى برنامَجهم. قم بتعديل البرنامَج الذي يطالب المستخدم باسم المِلف بحيث يطبع رسالة مضحكة عندما يكتب المستخدم اسم المِلف بالشكل التالي"na na boo boo". يجب أن يعمل البرنامَج عملًا طبيعيًّا مع جميع المِلفات الأخرى الموجودة وغير الموجودة. فيما يلي نموذج لتنفيذ البرنامَج:

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt


python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt


python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

تذكر بأن هذا مجردُ تمرين، فنحن لا نشجعك على ترك مفاجآت في برامجك!

الفصل الثامن

القوائم

8 القوائم

1.8 القائمة هي سلسلة

إنّ القائمة (List) هي سلسلة من القيم، كما هي السلاسل النصّيّة (string)، حيث تكون القيم في السلسلة النصّيّة عبارة عن محارِف، بينما يمكن أن تكون القيم في القائمة أيّ نوع بيانات. تُسمّى القيم في القائمة بالعناصر (elements أو items).

يوجد العديد من الطرق لإنشاء قائمة جديدة. الطريقة الأسهل هي حصر العناصر ضمن قوسين مُربّعين ("[" و"]"):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

يبيّن المثال الأوّل قائمة مؤلّفة من 4 أعداد صحيحة. بينما الثاني عبارة عن قائمة مؤلّفة من ثلاث سلاسل نصّيّة.

ليس بالضرورة أن تكون عناصر القائمة من النوع ذاته، حيث تحتوي القائمة التالية على سلسلة نصّيّة وعدد ذي فاصلة عشريّة وعدد صحيح. كما أنّ بإمكانها احتواء قائمة أخرى.

```
['spam', 2.0, 5, [10, 20]]
```

إنّ وجود قائمة ضمن قائمة أخرى يعني أنّها مُتداخلة (nested) أما القائمة التي لا تحتوي عناصر فتسمى بالقائمة الفارغة (empty list).

بإمكانك إسناد قيم القائمة إلى مُتغيّرات:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

2.8 القوائم قابلة للتعديل

إنّ القاعدة أو الطريقة المتّبعة في الوصول إلى عناصر قائمة هي ذاتها المُستخدمة في الوصول إلى

المحارف في السلسلة النصّيّة، أيّ عامل القوس (bracket operator). حيث يحدّد التعبير داخل الأقواس الفهرس المطلوب. تذكّر أنّ الفهارس تبدأ من الصفر.

```
>>> print(cheese[0])
cheddar
```

خِلافًا للسلاسل النصّيّة، فإنّ القوائم قابلة للتعديل، حيث بإمكانك تغيير ترتيب عناصر قائمة ما، أو إعادة تعيين عنصر فيها.

عندما يظهر عامل القوس في الجانب الأيسر من عمليّة الإسناد، فهو يحدّد عُنصرًا في القائمة ستُسنَد قيمة إليه.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

في المثال السابق، العُنصر الأوّل من القائمة المتضمّنة الأعداد، والذي كان ذا قيمة 123، أصبح الآن 5.

بإمكانك أن تَعدّ القائمة عبارة عن علاقة بين الفهارس والعناصر، وتسمّى هذه العلاقة بالربط (mapping)، حيث إنّ كلّ فهرس مُرتبط بأحد العناصر.

تعمل فهارس القوائم بنفس طريقة عمل فهارس السلاسل النصّيّة:

- أيّ تعبير يمثّل عدد صحيح يُمكن أن يُستخدم كفهرس.
- إذا حاولت أن تقرأ أو تكتب عُنصرًا غير موجود، ستحصل على خطأ فهرسة (IndexError).
- إذا كان لفهرس ما قيمة سالبة، فإنّه يبدأ العدّ بشكل عكسيّ ابتداءً من نهاية القائمة.

يُمكن للعامل in التعامل مع القوائم أيضًا.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
```

```
False
```

3.8 المرورعلى عناصرقائمة

يُعدّ استخدام حلقة for الطريقة الأكثر شُيوعًا للمرور على عناصر قائمة، وبشكل مماثل للتعامل مع السلاسل النّصيّة:

```
for cheese in cheeses:
    print(cheese)
```

هذا مُفيد إذا أردتَ فقط قراءة عناصر من القائمة، لكن إن أردت كتابة أو تحديث العناصر، فإنّكَ بحاجة إلى الفهارِس. من الشائع استخدام كِلا التابعين range وَlen لهذا الغرض:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

تُحدِّث هذه الحلقة قيمة كلّ عنصر في القائمة لدى مُرورها على العناصر تباعًا.

يُعيد التابع len عدد العناصر في القائمة. بينما يُعيد التابع range قائمة من الفهارِس من 0 حتّى n-1، حيث n هي عدد العناصر في القائمة.

في كُل مرّة ندخل في الحلقة، تُستخدَم قيمة i من قبل تعليمة الإسناد في جسم الحلقة، حيث تستخدِمَهُ لقراءة القيمة القديمة للعنصر، وإسناد القيمة الجديدة إليه، ويأخذ i قيمة فهرس العنصر التالي.

لا تُنفّذ حلقة for التعليمات في جسم الحلقة في حالة القائمة الفارغة:

```
for x in empty:
    print ('This never happens.')
```

بالرغم من أنّ بإمكان القائمة احتواء قائمة أخرى، إلّا أنَّ تلك القائمة تعدّ كعنصر مُنفرِد. لذا، فإنَّ طول القائمة التالية هو 4.

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

4.8 العمليّات على القوائم

يجمّع عامل الجمع + القوائم (يضعها جنباً إلى جنب بشكل متسلسل).

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

بشكل مُشابه، يُكرّر عامل النجمة -الضرب- * القائمة بمقدار عدد معلوم من المرّات.

```
>>> [0] * 4
[0, 0, 0, 0]

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

يُكرّر المثال الأوّل السلسلة 4 مرّات. بينما في المثال الثاني، تتكرّر السلسلة 3 مرّات.

5.8 تجزئة القوائم

يعمل عامل التجزئة أيضًا مع القوائم:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

إذا تجاهلت الفهرس الأوّل، فإنّ التجزئة تبدأ من بداية القائمة. أمّا إذا تجاهلت الفهرس الثاني، تستمرّ التجزئة حتّى النهاية. بينما إذا تجاهلت الاثنين معًا [:]، فإنّ التجزئة هي عبارة عن نسخة مُطابِقة للقائمة بأكملها.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

نظرًا إلى أنّ القوائم قابلة للتعديل، فمن المفيد غالبًا نسخ القائمة قبل إجراء عمليّات عليها. يُمكن لعامِل التجزئة على يسار عمليّة الإسناد أن يُحدّث قيم عدّة عناصر في نفس الوقت:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

6.8 توابع خاصّة بالقوائم

تتضمّن لغة بايثون عدّة توابع للعمل على القوائم. مثلًا، يضيف التابع append عُنصرًا جديدًا إلى نهاية قائمة.

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

في المثال التالي، يأخذ التابع extend قائمة كوسيط، ثمّ يضيف جميع عناصرها لقائمة أخرى دفعة واحدة:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

لم يطرأ على t2 في هذا المثال أيّ تعديل.

يرتّب التابع sort عناصر القائمة تصاعديًّا:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
```

```
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

إنَّ مُعظم توابع القوائم من النوع void، حيث إنّها تعدّل على القائمة، وتعيد None، لذا إذا حدث وكتبتَ t = t.sort()، ستحصل على نتيجة مخيّبة للآمال.

7.8 حذف العناصر

هناك عدّة طُرق لحذف العناصر من القائمة. إذا كُنت تعلم فهرس العنصر المُراد حذفه، بإمكانك عندئذٍ استعمال التابع pop.

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

تُجري التعليمة pop تعديلًا على القائمة، وتُعيد العنصر الذي أُزيل.

في حال لم تُحدّد فهرسًا معيّنًا، عندها تحذِف pop العنصر الأخير في القائمة، وتخزنه كقيمة مُرجعة.

بإمكانك استخدام العامل del إذا لم تكن بحاجة للاحتفاظ بالقيمة المحذوفة:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

إذا كنت على علم بالعنصر الذي ترغب بإزالتهِ، لكنّك لا تعلم الفهرس الخاصّ به، عندها بإمكانك استعمال remove.

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
```

```
['a', 'c']
```

القيمة التي تعيدها remove هي None.

لإزالة أكثر من عنصر، بإمكانك استخدام del مع فهرس التجزئة:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

كالعادة، التجزئة تشمل كلّ العناصر المحدّدة من الفهرس الأوّل حتّى الفهرس الذي يسبق الفهرس الثاني، أي لا تتضمّن العنصر ذا الفهرس الثاني.

8.8 القوائم والتوابع

يوجد عدد من التوابع الجاهزة التي بالإمكان استخدامها على القوائم، والتي تمنحك سلاسة البحث في القائمة دون الحاجة إلى استخدام الحلقات:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

يعمل تابع الجمع sum() فقط عندما تكون عناصر القائمة أعدادًا. أما التوابع الأخرى، مثل max() وَlen() وغيرها، تعمل مع قوائم ذات عناصر من نوع سلاسل نصّيّة وأنواع البيانات الأخرى القابلة

للمقارنة.

بإمكاننا إعادة كتابة البرنامج السابق الذي يحسب المتوسّط الحسابيّ للأعداد عبر استخدام القوائم.

في البداية، تمعّن في البرنامج الذي يحسب المتوسّط دون استخدام القوائم:

```
total = 0
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done':   break
    value = float(inp)
    total = total + value
    count = count + 1
average = total / count
print('Average:', average)

# Code: http://www.py4e.com/code3/avenum.py
```

في هذا البرنامج، لدينا كلٌّ من المتغيّرين count وَtotal، حيث يُخزّن المتغيّر count تعداد الأعداد، بينما يحفَظ total القيمة التراكمية للأعداد التي يدخلها المستخدم.

بإمكاننا ببساطة تخزين كلّ عدد عند إدخاله من قبل المستخدم، واستخدام توابع جاهزة لحساب تعداد الأعداد والمجموع في النهاية.

```
numlist = list()
while (True):
    inp = input('Enter a number: ')
    if inp == 'done':  break
    value = float(inp)
    numlist.append(value)
```

```
average = sum(numlist) / len(numlist)
print('Average:', average)

# Code: http://www.py4e.com/code3/avelist.py
```

أنشأنا قائمة فارغة قبل أن تبدأ الحلقة، وبعد ذلك في كلّ مرّة يكون لدينا عدد جديد نضيفه إلى القائمة. في نهاية البرنامج، نحسب مجموع الأعداد في القائمة ببساطة، ثمّ نقسّمه على عدد الأعداد لنحصل على المتوسّط الحسابيّ (المعدّل).

9.8 القوائم والسلاسل النصّيّة

إنّ السلسلة النصّيّة هي سلسلة من المحارف، بينما القائمة هي سلسلة من القيم، ولكنّ القائمة المؤلّفة من مجموعة محارِف لا تُعتبر سلسلة نصّيّة.

للتحويل من سلسلة نصّيّة إلى قائمة من المحارِف، بإمكانك استخدام التابع list:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

ولأنّ list هو اسم لتابع جاهز، عليك تجنُّب استخدامه كاسم لِمُتغيّر. وقد تجنّبت أيضًا استخدام الحرف "l" وذلك لشبهه بالعدد 1، لذلك استخدمت "t".

يقسّم التابع list السلسلة النصّيّة إلى أحرف منفصلة. أما إذا أردتَ تقسيم السلسلة النصّيّة إلى كلمات، بإمكانك استخدام التابع split.

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

بمجرّد أن تستعمل split لتقسيم السلسلة النصّيّة إلى كلمات، يكون بإمكانك استعمال عامل الفهرس (القوس القائم الزاوية) لاختيار كلمة محدّدة من القائمة.

يمكنك استدعاء split مع وسيط اختياريّ يُسمّى مُحدِّد (delimiter)، والذي يُحدّد المحرف الذي ستُقسَّم السلسلة وفقه. المثال التالي يستخدم الشَرْطِة (hyphen) كمُحدّد:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

يعمل join عكس عمل split، فهو يأخذ قائمة من السلاسل النصّيّة، ويجمّع العناصر. إنّ join تابع خاصّ بالسلسلة النصّيّة. لذلك، لاستخدامه، عليك استدعاءه باستخدام محدّد، وتُمرّر القائمة كمعامل.

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

وفي هذه الحالة يكون المُحدِّد هو مِحرف المسافة الفارغة ' '، وبالتالي فإنّ تعليمة join تضع فراغًا بين الكلمات. لكي ترتّب السلاسل النصّيّة دون فراغات، يمكنك استخدام السلسلة الخالية "" كمُحدِّد.

10.8 التعامل مع الأسطر في الملفات

عادةً، عند قراءتنا لملفّ، فنحن نرغب بالتعديل على الأسطر أكثر ما نرغب بمجرّد عرض السطر بأكمله.

في أكثر الأحيان، نرغب بإيجاد "الأسطر المهمّة"، ومن ثمَّ تحليل السّطر نفسهِ لإيجاد أجزاء مهمّة منه. على سبيل المثال، ماذا لو أردنا طباعة اختصار اسم يوم من أيّام الأسبوع الواردة في هذه الأسطر التي تبدأ بكلمة "From"؟

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

يعدّ تابع split فعّالًا جدًّا عندما نواجه هذا النوع من المسائل.

بإمكاننا كتابة برنامج صغير يبحث عن الأسطر التي تبدأ بـ From، ويفصل هذه الأسطر، ومن ثمّ طباعة الكلمة الثالثة في السطر.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
     print(words[2])

# Code: http://www.py4e.com/code3/search5.py
```

يُنتج البرنامج الخرج التالي:

```
Sat
Fri
Fri
Fri
...
```

لاحقًا، سنتعلّم بشكل مفصّل تقنيّاتٍ مُتقدمة لانتقاء الأسطر التي نُريد العمل عليها، وكيف نفرّق هذه الأسطر لإيجاد المعلومة التي نبحث عنها بدقّة.

11.8 الكائنات والقيَم

إذا نفَّذنا تعليمات الإسناد التالية:

```
a = 'banana'
b = 'banana'
```

نعلم أنّ كل من a وَb يُشيران إلى سلسلة نصّيّة، لكنّنا لا نعلم ما إذا كانا يُشيران إلى نفس السلسلة النصّيّة. هناك حالتان ممكنتان:

الشكل 12: القيم والكائنات

في الحالة الأولى: a وb يُشيران إلى كائنين مُختلفين لهما نفس القيمة.

في الحالة الثانية: a وb يُشيران إلى الكائن نفسه.

لِنختبر ما إذا كان هُناك مُتغيّران يُشيران إلى نفس الكائن. بالإمكان استخدام المعامل is.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

في هذا المثال، تُنشئ بايثون كائن واحد لسلسلة نصية، حيث يُشير إليه كل من a و b.

لكن عندما تُنشئ قائمتين فإنّك تحصل على كائنين.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

في هذه الحالة، نقول إنّ القائمتين متكافئتان (equivalent)، لأنّ لديهما نفس العناصر، لكنّهما غير متطابقتين (identical)، لأنّهما ليسا الكائن ذاته.

إذا كان لدينا كائنان متطابقان، فهذا يعني أنّهما متكافئان (متساويان)، ولكن كونهما متكافئين لا يعني بالضرورة أنّهما متطابقان.

إلى حدّ الآن، نحن نستخدم "الكائن" (object) و"القيمة" (value) بالتبادل، ولكن لتوخّي الدقّة نقول إنّ للكائن قيمة خاصّة به.

إذا نفّذت a = [1,2,3]، تُشير a إلى كائن لقائمة، والتي قيَمها هي سلسلة محدّدة من العناصر.

إذا امتلكت قائمة أخرى نفس العناصر، نقول إنّ لها نفس القيمة.

12.8 التسمية البديلة

إذا كان a يُشير إلى كائن، وأجريت بعمليّة إسناد b = a، عندها كلا المُتغيّرين سيُشيران إلى نفس الكائن:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

يُسمّى ارتباط المُتغيّر بالكائن بـ المرجع (reference)، في هذا المثال، يوجد مرجعين لنفس الكائن.

يكون للكائن الذي لديه أكثر من مرجع واحد أكثر من اسم واحد، لذلك نقول يملك هذا الكائن اسمًا بديلًا (aliased). إذا كان الكائن ذو الاسم البديل قابلًا للتعديل، فإنّ التغيّرات التي تحدث مع البديل تؤثّر على الآخر:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

على الرغم من أنّ هذا السلوك يُمكن أن يكون مُفيدًا، إلّا أنّه مسبّب للخطأ.

بشكل عامّ، إنّ تجنُّب التسمية البديلة يُعَدُّ أكثر أمانًا عند عملنا مع كائنات قابلة للتعديل. من أجل الكائنات غير القابلة للتعديل، مثل السلاسل النصّيّة، لا تُعدّ التسمية البديلة مشكلة. كما في المثال:

```
a = 'banana'
b = 'banana'
```

تقريبًا لا يوجد فرق فيما إذا كانت a أو b تُشير إلى نفس السلسلة النصّيّة، أم لا.

13.8 وسائط القائمة

عندما تُمرِّر قائمة إلى تابع، يحصل التابع على مرجع للقائمة. إذا عدّل التابع على مُعامل القائمة، تُلاحظ التغيير عند الاستدعاء. على سبيل المثال، يحذف التابع delete_head العنصر الأوّل من القائمة:

```
def delete_head(t):
    del t[0]
```

إليك كيفيّة استخدامه:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

المعامل t وَالمُتغيّر letters هما أسماء بديلة لنفس الكائن.

من المهمّ أن نُميّز بين العمليّات التي تُعدّل القوائم، والعمليّات التي تُنشئ قوائم جديدة. مثلًا، تعدّل append على القائمة، بينما يخلق عامل الجمع + قائمة جديدة.

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None
>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

يكون هذا الاختلاف مهمًّا عندما تكتب توابع من المُفترض بها أن تُعدّل القوائم.

مثلًا، هذا التابع لا يحذف أوّل عنصر في القائمة:

```
def bad_delete_head(t):
    t = t[1:]                    # WRONG!
```

يُنشئ عامل التجزئة قائمة جديدة، بحيث تُشير t إلى القائمة. ولكن ليس لأيٍّ من هذا تأثير على القائمة التي مُرِّرَت كوسيط.

البديل هو إنشاء تابع ينشئ قيمة جديدة ويُعيدها. على سبيل المثال: يُعيد التابع tail جميع العناصر عدا العنصر الأوّل من القائمة:

```
def tail(t):
    return t[1:]
```

يترك هذا التابع القائمة الأصليّة بلا أيّ تعديل. إليك كيفيّة استخدامه:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

التمرين الأول: أنشئ تابعًا باسم chop، بحيث يأخذ قائمة ويُعدّل عليها عن طريق إزالة العنصرين الأوّل والأخير، ويُعيد none، ثمَّ أنشئ تابعًا باسم middle يأخذ قائمة ويُعيد قائمة جديدة تحوي جميع العناصر عدا الأوّل والأخير.

14.8 التنقيح

إنّ سوء استخدام القوائم (والكائنات الأخرى القابلة للتعديل) قد يفضي إلى ساعات طويلة من عمليّة التنقيح. إليك بعض الحيَل والأساليب لتجنّب ذلك:

1 لا تنسَ أنّ مُعظم توابع القوائم تُعدّل الوسيط وتُعيد None (لا شيء). يُعدّ هذا عكسَ عمل توابع السلاسل النصّيّة التي تُعيد سلسلة نصّيّة جديدة، وتتغاضى عن السلسلة الأصليّة.

إذا كنتَ مُعتادًا على كتابة شيفرة السلسلة النصّيّة على الشكل التالي:

```
word = word.strip(
```

قد تكتب شيفرة القائمة على الشكل التالي:

```
t = t.sort()              # WRONG!
```

ولأنّ التابع sort يُعيد None، فإنّ العمليّة التالية التي تُجريها مع t مآلها الفشل.

قبل استخدام توابع وعوامل القوائم، عليك قراءة ملفّات التوثيق بعناية، واختبارها في الوضع التفاعليّ.

تصفح ملفات توثيق التوابع والعوامل التي تُشاركها القوائم مع سلاسل أخرى (كالسلاسل النصّيّة) في:

docs.python.org/library/stdtypes.html#common-sequence-operations

والتوابع والقوائم التي تُطبّق فقط على السلاسل القابلة للتغيير هنا:

docs.python.org/library/stdtypes.html#mutable-sequence-types

2 اختَر مُصطلحًا، والتزم به:

إنّ جزء من المشكلة مع القوائم يكمُن في تعدّد الأساليب للقيام بالأشياء. مثلًا، لإزالة عنصر من القائمة، يُمكنك استخدام كُلّ من pop، وَremove، وَdel، وحتّى التجزئة.

لإضافة عُنصر، يمكنك استخدام طريقة append، أو عامل الجمع +. لكن، لا تنسَ أنّ ما يلي يُعدّ صحيحًا:

```
t.append(x)
t = t + [x]
```

أمّا ما يلي، فهو خاطئ:

```
t.append([x])          # WRONG!
t = t.append(x)        # WRONG!
t + [x]                # WRONG!
t = t + x              # WRONG!
```

نفّذ هذه الأمثلة في الوضع التفاعليّ لتضمن أنك تفهم آليّة عملهم.

انتبه إلى أنّ السطر الأخير فقط يُسبّب خطأ تشغيل (runtime error)، بينما الثلاث أسطر الباقية مسموحة، لكنّها تنفّذ أمورًا خاطئة.

3 انسخ لتجنّب التسمية البديلة:

إذا كُنت تُريد استخدام تابع كـ sort الذي يُعدّل على الوسيط، ولكنّك تريد الاحتفاظ بالقائمة الأصليّة على أيّة حال، يمكنك عمل نسخة.

```
orig = t[:]
t.sort()
```

يُمكنك في هذا المثال أيضًا استخدام التابع الجاهز sorted، والذي يُعيد قائمة جديدة مُرتّبة، ويترك القائمة الأصليّة على حالها. لكن، في تلك الحالة، احرص على تجنُّب استخدام sorted كاسم لمُتغيّر.

4 استخدام القوائم والتابع split مع الملفّات:

عندما نقرأ ونحلّل الملفّات، هناك احتماليّة أنّ إحدى المدخلات قد توقف برنامجنا. لذا، فإنّ إعادة النّظر في نمط الحماية يُعدّ فكرةً جيّدةً عند كتابة البرامج التي تبحث في الملفّ "كالبحث عن إبرة في كومة قَش".

دعونا نُعِد النّظر في برنامجنا الذي يبحث عن يوم من أيّام الأسبوع من السطور الموجودة في ملفّ.

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

بمُجرّد تقسيمنا السطر إلى كلمات، بإمكاننا الاستغناء عن استخدام تعليمة startswith، والنّظر ببساطة إلى الكلمة الأولى من السطر لتحديد ما إذا كُنّا مُهتمّين في السطر ككلّ. كما بإمكاننا استخدام continue لتخطّي السطور التي لا تحتوي الكلمة From ككلمة أولى على النحو التالي:

```
fhand = open('mbox-short.txt')
for line in fhand:
        words = line.split()
        if words[0] != 'From' : continue
        print(words[2])
```

يبدو هذا أكثر سهولة. بالإضافة إلى أنّنا لا نحتاج حتّى إلى تنفيذ التابع rstrip لإزالة محرف السطر الجديد في نهاية الملف. لكن هل هذا أفضل؟

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
        if words[0] != 'From' : continue
IndexError: list index out of range
```

قد يبدو هذا عملًا صائبًا، بحيث أنّنا نحصل على اليوم من السطر الأوّل (السبت sat). لكن، بعد ذلك يفشل البرنامج مع وجود خطأ. ما الخطأ الذي حصل؟

ما هي البيانات التّالفة التي تسبّبت بفشل برنامج بايثون الأنيق والذكي الخاصّ بنا؟

قد تُحدّق مطوّلًا في البرنامج، وتتملّكُكَ الحيرة، أو يمكن أن تسأل شخص ما لمساعدتك. ولكنَّ النّهج الأذكى والأسرع هو إضافة تعليمة print. إنّ المكان الأفضل لإضافة تعليمة print هو بالتحديد قبل السّطر الذي أخفق به البرنامج، ثمّ طباعة البيانات التي تبدو مُسبِّبة للفشل.

قد يؤدّي هذا النّهج إلى عرض الكثير من الأسطر في الخرج، لكن على الأقلّ ستكون قد عثرت فورًا على طرف الخيط لحلّ المُشكلة، لذلك أضفنا تعليمة طباعة المتغيّر words قبل السّطر الخامس مُباشرةً. حتّى أنّنا أضفنا البادئة "Debug:" إلى السطر البرمجيّ، بحيث نتمكّن من المُحافظة على خرج البرنامج مُنفصلًا عن خرج عمليّة التنقيح.

```
for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From' : continue
    print(words[2])
```

عندما نُشغّل البرنامج، تعرض الكثير من السطور على الشاشة. لكن، في النهاية، نرى خرج التنقيح الخاصّ بنا، وخطأ التتبّع، لذا نُدرك ما حدث بالضبط قبل خطأ التتبُّع.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
```

```
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
        if words[0] != 'From' : continue
IndexError: list index out of range
```

كلّ سطر من عمليّة التنقيح يطبع قائمة من الكلمات، والتي نحصل عليها عند تفرقة السطر إلى كلمات.

عندما يفشل البرنامج، تكون قائمة الكلمات فارغة []. إذا فتحنا الملفّ في مُحرّر النصوص وتفحّصناه، سيظهر على الشكل التالي:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan 5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

يظهر الخطأ عندما يصطدم برنامجنا بسطر فارغ.

بالطبع، لا يحتوي السطر الفارغ على كلمات. لماذا لم نُفكّر بذلك عند كتابة البرنامج؟

عندما تبحث الشّيفرة عن الكلمة الأولى word[0] للتحقّق منها سعيًا في معرفة ما إذا كانت تُطابق From، نحصل على خطأ فهرس خارج النطاق (index out of range) .

بالطبع، هذا هو المكان المثاليّ لإضافة البعض من شيفرات الحماية – guardian code لتجنُّب عمليّة التحقّق من الكلمة الأولى في حال لم تكن الكلمة الأولى موجودة.

يوجد العديد من الأساليب لحماية هذه الشيفرة. سنلجأ إلى التحقّق من عدد الكلمات التي لدينا قبل البحث عن الكلمة الأولى:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
```

```
    words = line.split()
    # print('Debug:', words)
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print(words[2])
```

في البداية، تجاهلنا تعليمة التنقيح بدلًا من إزالتها، والسبب أنّه في حال فشل التعديل الذي قمنا به، فنحن بحاجة إلى التنقيح مُجدّدًا.

ثمّ أضفنا تعليمة حماية تتحقّق من حالة عدم وجود كلمات. وفي هذه الحالة، نستعمل التعليمة continue لتخطّي السطر التالي في الملفّ.

يُمكننا اعتبار أنّ تعليمتي continue تُساعداننا في تنقيح بعض الأسطر البرمجيّة "المُهمّة" بالنسبة لنا، والتي نُريد أن نُخضعها لمزيد من المعالجة.

إنّ السطر الذي لا يحوي كلمات يُعدّ "غير مهمّ"، لذلك نتخطّاه إلى السطر الذي يليه. أيضًا السطر الذي لا يحوي كلمة From ككلمة أولى غير مهمّ، ويمكن تخطّيهِ.

إنّ البرنامج -كما عُدِّلَ- يعمل بنجاح، لذا من الممكن أن يكون صحيح.

إنّ تعليمة الحماية تحرص على أنّ words[0] لن تفشل أبدًا. لكن، ربّما هذا غير كافٍ، لأنّنا عندما نُبرمج نتساءل دومًا، "ما الخطأ الذي قد يحدث"؟

التمرين الثاني: اكتشف أيّ سطر من البرنامج أعلاه لا يزال غير محميٍّ بشكل كامل.

تحقّق من إمكانيّة إنشاء ملفّ نصّيّ يؤدّي إلى فشل البرنامج، ثمَّ عدّل البرنامج بحيث تؤمّن حماية للسطر البرمجيّ. بعد ذلك، اختبرهُ للتأكّد من تعامله السليم مع ملفّك النصّيّ الجديد.

التمرين الثالث: أعِد كتابة شيفرة الحماية في المثال أعلاه دون استعمال تعليمتي if فبدلًا من ذلك، استخدم التعبير المنطقيّ المُركّب باستخدام العامل المنطقيّ or مع عبارة if واحدة.

15.8 فهرس المصطلحات

- **التسمية البديلة (Aliasing):** الحالة التي يكون فيها مُتغيّران أو أكثر يُشيران إلى نفس الكائن.

- **المُحدِّدْ (Delimiter):** مِحرف أو سلسلة نصّيّة، تُستعمل للإشارة إلى المكان الذي يجب أن تُفصل به السلسلة النصّيّة.
- **العُنصر (Element):** واحد من القيم في القائمة (أو سلسلة أخرى)، يمكن أن نسمّيه أيضًا item.
- **مُكافِئ (Equivalent):** لهُ القيمة ذاتها.
- **الفهرس (index):** قيمة صحيحة تُشير إلى عنصر في القائمة.
- **التطابُق (Identical):** مُطابق لنفس الكائن (بما يعني التكافؤ).
- **القائمة (List):** سلسلة من القيم.
- **المرور على عناصر قائمة (List traversal):** الوصول التسلسليّ إلى كلّ عنصر في القائمة.
- **القائمة المتداخلة (Nested list):** القائمة التي تكون عُنصرًا ضمن قائمة أُخرى.
- **الكائن (Object):** شيء أو مُتغيّر يُمكن الإشارة إليه، بحيث يكون للكائن نوع وقيمة.
- **المرجع (Reference):** يمثل الارتباط بين المُتغير وقيمته.

16.8 تمارين

- **التمرين الرابع:** حمّل نسخة من الملفّ من الرابط التالي:

www.py4e.com/code3/romeo.txt

اكتب برنامجًا لفتح الملف romeo.txt وقراءته سطرًا بسطر.

من أجل كُلّ سطر، فَرِّق السطر إلى كلمات مُستعملًا التابع split. ومن أجل كُلّ كلمة، تحقّق ما إذا كانت موجودة مُسبقًا في القائمة. في حال لم تكن موجودة، أضفها إلى القائمة.

عند اكتمال البرنامج، رتّب واطبع الكلمات الناتجة ترتيبًا أبجديًا.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
```

```
'with', 'yonder']
```

- **التمرين الخامس:** اكتب برنامجًا لقراءة بيانات الملفmail box . عندما تجد سطرًا يبدأ بـ From، قسِّم السطر إلى كلمات مستخدمًا التابع split. نحنُ نهتمُّ بمُرسل الرسالة، والتي هي الكلمة الثانية من السطر.

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

ستحلِّل السطر الذي يبدأ بـ From، ثمّ تطبع الكلمة الثانية من كُلّ سطر يبدأ بـ From. بالإضافة إلى ذلك، ستحصي عدد السطور التي تبدأ بـ From، وتطبع العدد في النهاية. فيما يلي خرج لعيّنة محذوف منها بضعة أسطر:

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

- **التمرين السادس:** أعِد كتابة البرنامج الذي يطلب من المستخدم قائمة مكوّنة من أعداد، ثُمَّ يطبع العدد الأعظميّ والأصغريّ من الأعداد عندما يُدخل المستخدم "done" في نهاية البرنامج.

اكتب البرنامج لتخزين الأعداد التي أدخلها المستخدم إلى القائمة. واستخدم التابعين

max() وَmin() لحساب القيمة الأعظميّة والأصغريّة للأعداد بعد اكتمال الحلقة.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

الفصل التاسع

القواميس

9 القواميس

يشابه القاموس (dictionary) القوائم (list) إلا أنه أكثر شمولية، ففي القوائم تكون فهارس المواقع أعداد صحيحة int على عكس القواميس حيث قد تكون من أي نوع، حيث يمكنك تخيل القاموس وكأنه يربط بين مجموعة فهارس والتي تدعى بالمفاتيح (keys) ومجموعة من القيم (values)، حيث يدعى ارتباط المفتاح مع القيمة بزوج مفتاح-قيمة (key-value pair) أو أحيانًا يدعى بالعنصر (item).

لتوضيح ما سبق، سننشئ قاموس يربط كلمات بالإنكليزية مع ترجماتها في اللغة الإسبانية بالتالي هنا كلا المفاتيح والقيم هما من نوع البيانات سلاسل نصية.

ينشئ التابع dict قاموسًا فارغًا بدون أيّ عناصر. لذا، يجب عليك تجنب تسمية متغيراتك بهذا الاسم، لأنه اسم تابع في لغة بايثون:

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

تمثل أقواس المجموعة {} قاموسًا فارغًا ولإضافة عناصر لهذا القاموس يجب استخدام الأقواس المربعة []:

```
>>> eng2sp['one'] = 'uno'
```

حيث يضيف هذا السطر البرمجي عنصر يرتبط فيه المفتاح 'one' بالقيمة "uno" فإذا أظهرنا محتوى القاموس ينتج لدينا زوج مفتاح-قيمة يفصل بينهما علامة النقطتين:

```
>>> print(eng2sp)
{'one' : 'uno'}
```

تماثل صيغة الدخل صيغة الخرج في المثال السابق، لكن في حال إنشاء قاموس بثلاثة عناصر قد تتفاجأ عند طباعة eng2sp حيث يظهر ما يلي:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

حيث ترتيب العناصر لا يكون نفسه في كل مرة ولا يمكن التنبؤ به في القواميس، حتى إن جربت كتابة نفس المثال على حاسوبك فقد تحصل على نتيجة مختلفة، إلا أن هذا لا يشكل مشكلة لأن فهارس العناصر في القواميس ليست عبارة عن أرقام صحيحة بل تُستخدم المفاتيح لإظهار القيمة الموافقة لها:

```
>>> print(eng2sp['two'])

'dos'
```

فالمفتاح 'two' مرتبط بالقيمة 'dos' دائمًا. فيكون ترتيب العناصر غير مهم، وستحصل على رسالة خطأ إن كان القاموس لا يحوي المفتاح المطلوب:

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

ويمكن استخدام التابع len مع القواميس ليظهر عدد العناصر في القاموس:

```
>>> len(eng2sp)
3
```

كما يمكن استخدام العامل in معها حيث يؤكد وجود مفتاح معين في القاموس من عدمه (لا يتعامل مع القيم):

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

أما للبحث عن قيمة ما ضمن القاموس، يمكن استخدام تابع يدعى values يعيد القيم كقائمة ثم نستخدم العامل in:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

مع الأخذ بعين الاعتبار أن العامل in يستخدم خوارزميات مختلفة لكلٍّ من القوائم والقواميس، ففي

القوائم، يعتمد على خوارزمية بحث خطية، مما يزيد الوقت اللازم للبحث في قيم القائمة كلما زاد طولها. بينما تستخدم بايثون للقواميس خوارزمية تدعى "hash table" والتي تتميز بأن العامل in سيستغرق نفس الوقت في البحث ضمن قاموسٍ ما بغض النظرِ عن عدد عناصره. ولا يسعني الحديث هنا عن ميزات هذه الخوارزمية الرائعة ولكن بإمكانك أن تقرأ القليل عنها من هنا:

www.wikipedia.org/wiki/Hash_table

التمرين الأول: حمّل نسخة من الملف الموجود على الرابط الآتي:

www.py4e.com/code3/words.txt

اكتب برنامجًا يقرأ الكلمات الموجود في هذا الملف ثم يخزنها كمفاتيح في قاموسٍ ما بغض النظر عما ستكون عليه القيم، ثم استخدم العامل in للتحقق من وجود كلمة معينة.

1.9 استخدام القواميس في العد

افترض وجود نصٍ أمامك تريد أن تستخرج منه عدد مرات تكرار كل حرف، توجد عدةُ طرق لحل هذا:

1. بإمكانك إنشاء 26 متغير حيث يقابل كل متغير أحد الأحرف الأبجدية الإنكليزية، ثم تمر على كل محرف على حدة لتزيد قيمة العداد الموافقة لكل متغير مستخدمًا سلسلة من العبارات الشرطية.
2. تستطيع إنشاء قائمة ذات 26 عنصر ثم تحويل كل محرف إلى رقم بواسطة التابع ord لاستخدامه كفهرس في القائمة وزيادة العداد الموافق له.
3. يمكن إنشاء قاموس حيث تشكل الأحرف المفاتيح فيه، وتشكل عددها القيم الموافقة للمفاتيح. فعند اكتشاف المحرف لأول مرة، سيُدخل عنصر جديد للقاموس بينما في المرات القادمة ستُزاد قيمة هذا العنصر فقط.

تحل كل طريقة من هذه الطرق المشكلة بأسلوب مختلف، ونجد هنا مفهوم التنفيذ ويعني أسلوب حل مسألة ما، وتكون بعض هذه الاساليب أفضل من غيرها، فعلى سبيل المثال تكمن فائدة تطبيق استخدام القاموس في عدم حاجتنا لمعرفة الأحرف التي ستظهر مسبقًا بل نخصص لها مكانًا معينًا بعد ظهورها، وسيبدو البرنامج كما يأتي:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

أي أننا علميًا حسبنا الهيستوغرام (histogram) وهو مصطلح إحصائي يدل على مجموعة من العدادات (أو التكرارات) للعناصر. وكما نرى فإن الحلقة for تمر على محارف النص، وفي كل مرة لا نجد المحرف ضمن القاموس ننشئ عنصر جديد فيه مفتاحه c وقيمته الابتدائية 1 (بما أن المحرف ظهر لمرة واحدة)، لكن إن كان المفتاح c موجودًا ضمنه مسبقًا فنكتفي بزيادة قيمة d[c] ليكون خرج البرنامج كالآتي:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

أي، يبين الهيستوغرام أن الأحرف a وb قد ظهرت مرة واحدة في حين تكرر o مرتين وهكذا دواليك، كما تملك القواميس تابع يدعى get يتطلب هذا التابع معاملين هما المفتاح وقيمة معينة يعيدها في حال عدم وجود المفتاح ضمن القاموس وإلا يعيد القيمة الموافقة للمفتاح والموجودة ضمن القاموس، وتمثل هذه التعليمة كما يأتي:

```
counts = { 'chuck': 1 , 'annie': 42, 'jan': 100}
>>> print ( counts.get ( ‘jan’, 0))
100
>>> print ( counts.get (‘tim’, 0))
0
```

وهذا يمكننا من كتابة برنامجنا بأسلوب مختصر أكثر فالتابع get يحل مسألة عدم وجود المفتاح ضمن القاموس تلقائيًا مما يؤدي إلى اختصار أربعة أسطر برمجية إلى واحد والاستعاضة عن تعليمة if:

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print(d)
```

إن اتباع هذا الأسلوب شائع في بايثون وسنستعمله عدة مرات في بقية الكتاب، لذلك قد تحتاج بعض الوقت لتقارن بين الأسلوبين حيث استخدمنا get للاستعاضة عن تعليمة if والعامل in في الحلقة فكلاهما يؤديان نفس الوظيفة إلا أن أحدهما أكثر إيجازًا.

2.9 القواميس والملفات

إن استخدام القواميس لعد عدد مرات تكرار الكلمات في الملفات النصية يعد استخدامًا شائعًا، لذلك سنبدأ بتوضيح هذا بمثال صغير مأخوذ من الملف النصي "Romeo and Juliet" حيث في الأمثلة الأولى سنستخدم نسخة مبسطة ومختصرة من النص بدون علامات ترقيم كما يأتي:

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

سنكتب الآن برنامج بلغة بايثون يقرأ أسطر النص ويجزئها إلى قائمة من الكلمات ثم يمر على كل منها باستخدام حلقة ليعد مرات تكرارها حافظًا النتيجة في قاموس. كما ستلاحظ أننا استخدمنا حلقتي for حيث تقرأ الأولى أسطر النص، بينما تمر الثانية على كلمات كل سطر. ويدعى هذا النمط بالحلقات المتداخلة (nested loops) وسبب التسمية يرجع لوجود حلقة خارجية وأخرى داخلية ضمنها. وتنفذ الحلقة الداخلية جميع تكراراتها من أجل كل تكرار للحلقة الخارجية، فيبدو وكأن الحلقة الداخلية تعمل بشكل سريع بينما تكون الخارجية أبطأ منها، وأيضًا يضمن لنا هذا النمط المرور على كل كلمة في كل سطر من النص المدخل:

```
fname = input('Enter the file name: ' )
try:
   fhand = open(fname)
```

```
except:
    print('File cannot be opened: ', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1
print(counts)
# Code: http://www.py4e.com/code3/count1.py
```

لقد استخدمنا في تعليمة else التعليمة البديلة المختصرة للزيادة العددية حيث counts[word]+=1 تكافئ counts[word]=counts[word]+1 ويمكن استخدام أيٍّ منهما لتغيير القيمة العددية بأي قيمة مطلوبة حيث توجد بدائل شبيهة مثل -= و *= و /= وعند تنفيذ البرنامج سنحصل على سطر من العدادات بصيغة غير مرتبة كما يأتي (يمكن الحصول على الملف romeo.txt من www.py4e.com/code3/romeo.txt):

```
python count1.py
Enter the file name: romeo.txt

{'and': 3, 'envious': 1, 'already': 1, 'fair': 1, 'is': 3, 'through': 1, 'pale': 1, 'yonder': 1, 'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1, 'window': 1, 'sick': 1, 'east': 1, 'breaks': 1, 'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1, 'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

ولكنه من غير المريح البحث عن أكثر الكلمات تكرارًا في القاموس، لذلك سنضيف بعض التعليمات البرمجية للحصول على الخرج المطلوب.

3.9 الحلقات والقواميس

إن حلقة for تمر على مفاتيح القاموس وفي مثالنا الآتي فإنها ستطبع المفتاح مع القيمة الموافقة له:

```
counts = { 'chuck': 1 , 'annie': 42, 'jan': 100}
for key in counts:
    print(key, counts[key])
```

فيكون الخرج:

```
jan 100
chuck 1
annie 42
```

وكما ذكرنا سابقًا فالمفاتيح غير مرتبة بترتيبٍ معين، ويمكننا استخدام هذا النمط لتنفيذ ما تعلمناه سابقًا، فعلى سبيل المثال سنكتب البرنامج الآتي للحصول على عناصر القاموس ذات قيمة أكبر من عشرة:

```
counts = { 'chuck': 1 , 'annie': 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```

ستمر الحلقة على مفاتيح القاموس لذلك نستخدم عامل الفهرس لاستدعاء القيمة الموافقة للمفتاح ويكون الخرج:

```
jan 100
annie 42
```

أي أننا حصلنا فقط على العناصر ذات القيم الأكبر من عشرة، أما لعرض المفاتيح بترتيب أبجدي فيجب علينا أولًا إنشاء قائمة من مفاتيح القاموس باستخدام التابع keys ثم ترتيبها، ثم طباعة الأزواج مفتاح-قيمة بالترتيب الأبجدي:

```
counts = { 'chuck': 1 , 'annie': 42, 'jan': 100}
lst = list(counts.keys())
```

```
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

ويكون الخرج:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

نرى أولًا قائمة المفاتيح غير المرتبة التي حصلنا عليها باستخدام التابع keys ثم نرى الأزواج مفتاح-قيمة المرتبة.

4.9 التعامل مع النصوص

لقد جعلنا النص في المثال السابق بأبسط شكل بإزالة كل علامات الترقيم منه، إلا أن النص الأصلي يحتوي على العديد من علامات الترقيم كما نلاحظ:

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
 Who is already sick and pale with grief,
```

وبما أن التابع split يعامل الكلمات كرموز تفصل بينها فراغات فستعامل "soft!" و"soft" ككلمتين مختلفتين وسيتم إنشاء مكان منفرد لكلٍّ منهما ضمن القاموس. وأيضًا سنعامل "who" و"Who" ككلمتين مختلفتين باعتبار أن النص يحوي حروف كبيرة وصغيرة، ولكن نستطيع حل المشكلتين باستخدام التوابع النصية lower وpunctuation وtranslate حيث أن الأخيرة هي الأفضل وتوصيفها كما يأتي:

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

والتي تعني: استبدل المحارف في fromstr بالمحارف ذات الموقع نفسه في tostr واحذف جميع المحارف

في deletestr، ويمكن أن يكون fromstr وtostr سلاسل نصية فارغة مع إهمال deletestr.

لن نعرّف tostr ولكن سنستخدم معامل deletestr لحذف علامات الترقيم كما سنطلب من بايثون أن تخبرنا بمجموعة المحارف التي تعتبرها كعلامات ترقيم وذلك كما يأتي:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

ويجب الإشارة إلى أن المعاملات المستخدمة مع translate كانت مختلفة في python 2.0، ثم نطبق التعديلات الآتية على برنامجنا:

```
import string

fname = input('Enter the file name: ' )
try:
   fhand = open(fname)
except:
   print( ' File cannot be opened: ' , fname)
   exit()
counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(line.maketrans(' ', ' ', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1
print(counts)

# Code: http://www.py4e.com/code3/count2.py
```

إن جزء من تعلم فن بايثون أو التفكير بطريقة بايثون هو إدراك أن بايثون تحتوي على قدرات (توابع جاهزة) لحل العديد من مشاكل تحليل البيانات، وسترى مع مرور الزمن أمثلة كافية وستقرأ ما يكفي من التوصيفات التي تجعلك تجيد البحث وتستفيد من البرامج المكتوبة من قبل مبرمجين آخرين مما يسهم في تسهيل عملك.

يكون الخرج المختصر للبرنامج السابق كما يأتي:

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2, 'kinsmen': 2, 'what': 11, 'thinkst': 1,
'love': 24, 'cloak': 1, 'a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40, 'maiden': 1,
'whiteupturned': 1, 'juliet': 32, 'gentleman': 1, 'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

ولكن البحث عما نريد ضمن هذا الخرج ما يزال غير عملي وبإمكاننا استخدام بايثون للحصول ما نريد بالضبط إلا أننا سنحتاج للحديث عن الصفوف (tuples) أولًا وسنعود بعدها إلى هذا المثال.

5.9 التنقيح

مع زيادة حجم البيانات يصبح من الصعب التنقيح عبر طباعة الخرج والتحقق من البيانات يدويًا لذا هاكَ بعض المقترحات لحل هذا:

1. **تقليص حجم الدخل:** إن أمكن، فعلى سبيل المثال إذا كان البرنامج يقرأ ملف نصي فابدأ بأول عشرة أسطر أو بأصغر مثال يمكنك إيجاده حيث تستطيع التعديل على الملفات مباشرةً أو تعديل البرنامج ليقرأ أول عدد ما من السطور وهذا محبذ أكثر، وفي حال وجود خطأ فيمكنك تقليص عدد الأسطر إلى عدد أقل حتى يظهر الخطأ ثم قم بزيادته تدريجيًا مع تصحيح الأخطاء.
2. **تفقد موجز البيانات و أنواعها:** اطبع موجز عن البيانات بدلًا من طباعتها وتفقدها بأكملها. مثل عدد عناصر قاموس ما أو مجموع قيم قائمة من الأرقام. وأيضًا السبب الأكثر شيوعًا للأخطاء أثناء التشغيل (runtime errors) هو وجود قيمة معينة من نوع خاطئ، ويكفي عادةً طباعة نوع هذه القيمة لتنقيح هذا النوع من الأخطاء.
3. **اكتب حالات اختبار**: أحيانًا يمكنك أن تكتب برنامج لتفقد الأخطاء تلقائيًا، كحالة حساب متوسط قيم قائمة ما، حيث يمكن التحقق ما إذا كان الناتج أصغر من أعظم قيمة فيها أو

أكبر من أصغر قيمة ويدعى هذا بالاختبار المنطقي فهو يكشف الأخطاء غير المنطقية على الإطلاق، كما يوجد اختبار آخر يسمى باختبار الاتساق أي يقارن بين ناتجي عمليتين حسابيتين للتأكد من توافقهما.

4. **اطبع الخرج:** إن طباعة خرج عملية التنقيح يسهل كشف الأخطاء.

وللتذكير فإن الوقت الذي تقضيه في كتابة وبناء أساس البرنامج بشكل صحيح يقلل الوقت الذي تقضيه في التنقيح.

6.9 فهرس المصطلحات

- **القاموس (dictionary):** يربط بين مجموعة من المفاتيح مع القيم المقابلة لها.
- **خوارزمية (hashtable):** خوارزمية مستخدمة في القواميس ضمن بايثون.
- **تابع هاش (hash function):** تابع تستخدمه الخوارزمية hashtable لتحديد موقع مفتاح ما.
- **الهيستوغرام (histogram):** لتمثيل التكرارات أو التعدادات.
- **عملية التنفيذ (implementation):** طريقة تنفيذ عملية حسابية ما.
- **عنصر (item):** اسم آخر لزوج المفتاح-قيمة.
- **مفتاح (key):** كائن يظهر في القاموس كأول جزء من زوج المفتاح-قيمة.
- **زوج مفتاح-قيمة (key-value pair):** تمثيل العلاقة بين المفتاح والقيمة الموافقة في قاموس ما.
- **البحث في القاموس (lookup):** عملية تنفذ في القواميس لإيجاد القيمة الموافقة لمفتاح ما.
- **الحلقات المتداخلة (nested loops):** وهذا عند وجود حلقة أو أكثر ضمن حلقة أخرى حيث تنهي الحلقة الداخلية تنفيذ جميع دوراتها من أجل كل دورة للحلقة الخارجية.
- **قيمة (value):** غرض في القاموس يمثل الجزء الثاني من الزوج مفتاح-قيمة وهي تختلف هنا عن الاستعمالات السابقة لكلمة value.

7.9 تمارين:

- **التمرين الثاني:** اكتب برنامجًا يصنف رسائل البريد الالكتروني بحسب يوم إرسالها. ولتنفيذ هذا، ابحث عن الأسطر التي تبدأ بكلمة From ثم ابحث عن الكلمة الثالثة وأنشئ عداد للتكرار لأيام الإرسال ثم اطبع محتوى قاموسك (الترتيب غير مهم).

مثال:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

مثال عن الخرج:

```
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

- **التمرين الثالث:** اكتب برنامجًا لقراءة سجل بريد إلكتروني معين وأنشئ هيستوغرام باستخدام القواميس لتبيان عدد الرسائل الواصلة له من كل إيميل ثم اطبع عناصر القاموس.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1, 'rjlowe@iupui.edu': 2,
'gsilver@umich.edu': 3, 'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2, 'ray@media.berkeley.edu': 1}
```

- **التمرين الرابع:** أضف بعض التعليمات لبرنامجك السابق لإيجاد الشخص الذي أرسل أكبر عدد من الرسائل الإلكترونية، أي ابحث في القاموس عن القيمة العظمى باستخدام الحلقات بعد قراءة البيانات وإنشاء القاموس (راجع الفصل الخامس: حلقات القيم العظمى والصغرى)، ثم اطبع عنوان البريد المطلوب مع عدد الرسائل التي أرسلت منه.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5
Enter a file name: mbox.txt
```

```
zqian@umich.edu 195
```

- **التمرين الخامس:** اكتب برنامجًا يسجل اسم النطاق (domain) فقط بدلًا من العنوان الكامل للبريد الإلكتروني. أي، من أين أرسلت الرسالة، لا من أرسلها، مع عدد مرات تكرارها ثم اطبع عناصر القاموس.

```
python schoolcount.py

Enter a file name: mbox-short.txt

{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7, 'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8
```

الفصل العاشر

الصفوف

10 الصفوف

1.10 الصفوف غير قابلة للتعديل

الصفوف عبارة عن سلاسل من القيم، وكما هو الحال مع القوائم، يمكن أن نخزن في الصفوف قيم من كافة الأنواع، كما أن هذه القيم تفهرس باستخدام الأعداد الصحيحة، ويكمن الاختلاف المهم الذي تمتاز به الصفوف في كونها غير قابلة للتعديل. كما أنّ الصفوف قابلة للمقارنة، ويمكن تطبيق خوارزمية الهاش (Hash) عليها، مما يسمح بترتيب قائمة من الصفوف أو استخدام الصفوف كمفاتيح في قواميس بايثون.

عند كتابة صف نجد أنه عبارة عن قائمة تتكون من سلسلة قيم يتخللها فواصل:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

وعلى الرغم من عدم الحاجة لوضع الصف بين قوسين يشيع ذلك لتسهيل عملية تمييز الصفوف في البرنامج:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

وعند إنشاء صف مكون من عنصر واحد يجب أن تضيف فاصلة إلى نهاية هذا الصف:

```
>>> t1 = ('a' , )
>>> type(t1)
<type 'tuple'>
```

وفي حال عدم إضافة هذه الفاصلة سيعامل الصف على أنه سلسلة نصية:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

ويمكن إنشاء صف باستخدام التابع الجاهز tuple، وبدون استخدام أي وسائط سنحصل على صف فارغ:

```
>>> t = tuple()
>>> print(t)
```

```
()
```

وفي حال كون الوسيط المستخدم مع التابع سلسلة من نوع ما (سلسلة نصية أو قائمة أو صف) فإن النتيجة ستكون صفًا من عناصر هذه السلسلة:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

ينبغي تجنب استخدام كلمة tuple كاسم للمتغيرات، وذلك باعتباره محجوزًا كتابع جاهز لإنشاء الصفوف.

تعمل غالبية عوامل القوائم على الصفوف. على سبيل المثال، فإن عامل القوس المربع يستدعي العناصر في الصف كما هو الحال مع القوائم:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

و يحدد عامل التجزئة نطاقًا من العناصر أيضًا:

```
>>> print(t[1:3])
('b', 'c')
```

ولكن إذا حاولت أن تعدل قيمة أحد العناصر في الصف فستحصل على رسالة خطأ:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

عوضًا عن ذلك يمكن أن تغير عنصرًا بآخر:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

2.10 مقارنة الصفوف

يعمل عامل المقارنة مع الصفوف وغيرها من السلاسل، حيث تبدأ المقارنة بالعنصر الأول من كل سلسلة، وفي حال تكافأ الطرفان فتنتقل المقارنة إلى العنصر التالي، وهكذا حتى العثور على عنصرين مختلفين. العناصر التي تتلو نقطة الاختلاف لا تؤخذ بعين الاعتبار (حتى لو كانت كبيرة جدًا):

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

يعمل التابع sort بنفس الطريقة السابقة، فيبدأ بالعنصر الأول وفي حال التكافؤ يُرتب وفق العنصر الثاني وهكذا.

تُستخدم هذه الميزة ضمن نمط العمليات المسمى (Decorate, Sort, Undecorate أو اختصارًا DSU) الذي يتألف من الخطوات الثلاثة السابقة الذكر:

1- مَيّز (Decorate) قيم سلسلة ما بإنشاء قائمة من الصفوف تحوي عنصر ترتيب واحد -علامة التمييز- أو أكثر يليه عنصر من السلسلة.
2- رتب (Sort) الصفوف باستخدام التابع sort.
3- إزالة التمييز (Undecorate) وذلك باستخراج عناصر السلسلة التي رتُّبت سابقًا.

فلنفترض مثلًا حاجتنا لترتيب قائمة من الكلمات من الكلمة الأطول إلى الأقصر، فنكتب البرنامج التالي:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))

t.sort(reverse=True)
```

```
res = list()
for length, word in t:
    res.append(word)
print(res)

# Code: http://www.py4e.com/code3/soft.py
```

تنشئ الحلقة الأولى قائمة من الصفوف، بحيث يتكون كل صف من كلمة مسبوقةٍ بعدد أحرفها.

يقارن تابع الترتيب sort العنصر الأول -طول الكلمة- ولا يأخذ العنصر الثاني بعين الاعتبار إلا لحسم التكافؤات. نستخدم الوسيط (reverse=True) لجعل التابع يرتب القيم تنازليًا.

تمر الحلقة الثانية على قائمة الصفوف وتنشئ قائمة من الكلمات تنازليًا حسب طولها والكلمتين المكونتين من أربعة أحرف تم ترتيبهما تنازليًا حسب الحرف الأول، ولذلك تظهر الكلمة "what" قبل الكلمة "soft" في القائمة المرتبة، ويكون خرج البرنامج كالتالي:

```
['yonder', 'window', 'breaks', 'light', 'what', 'soft', 'but', 'in']
```

3.10 إسناد الصفوف

تعتبر إمكانية استخدام تعليمة إسناد متغيرها عبارة عن صف من الميزات الفريدة في بايثون، حيث أنها تسمح بإسناد قيم لأكثر من متغير معًا، عندما تكون القيم عبارة عن سلسلة.

نرى في المثال التالي قائمة من عنصرين (سلسلة) نقوم بإسنادهما إلى المتغيرين x وy بتعليمة واحدة.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
```

تعامل لغة بايثون تعليمة الإسناد السابقة تقريبًا كما يلي:

```
>>> m = [ 'have', 'fun' ]
```

```
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
```

تجدر الإشارة إلى أن لغة بايثون لا تترجم التعليمات حرفيًا، ففي حال استخدمنا قاموس بدلًا عن القائمة في المثال السابق فلن يتم تنفيذ البرنامج -كما قد نتوقع-.

عادةً عندما نكتب تعليمة إسناد طرفها اليساري عبارة عن صف، فإننا نهمل الأقواس، ولكن في حال لم نفعل فإن ذلك مقبول وصحيح:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
```

ومن التطبيقات العملية لهذه الميزة هو القدرة على تبديل قيم متغيرين فيما بينهما:

```
>>> a, b = b, a
```

هنا كل من طرفي التعليمة عبارة عن صف، القسم الأيسر صف متغيرات، بينما القسم الأيمن صف من التعابير. كل قيمة من الطرف الأيمن تُسند إلى المتغير المقابل لها من الطرف الأيسر. يتم حساب كل التعابير في الطرف الأيمن قبل أي اسناد.

يجب أن يكون عدد المتغيرات في الطرف الأيسر مساويًا لعدد القيم في الطرف الأيمن، وفي حال عدم تساوي الطرفين نحصل على رسالة خطأ:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

ويمكننا تعميم ذلك حيث أن الطرف الأيمن يمكن أن يكون أي نوع من السلاسل (نص أو قائمة أو صف). يمكننا مثلًا أن نقسم عنوان البريد الإلكتروني إلى قسمين ونسند كل قسم إلى متغير:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

تعطينا تعليمة التقسيم split خرجًا على شكل قائمة بعنصرين، الأول يسند إلى المتغير uname والثاني إلى المتغير domain:

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

4.10 القواميس والصفوف

يُستخدم التابع items مع القواميس، ويعيد قائمة من الصفوف، كل صف عبارة عن زوج من مفتاح-وقيمة:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

وبطبيعة الحال فإن قيم القاموس غير مرتبة، ولكن بما أننا حصلنا على قائمة من الصفوف وبما أن الصفوف يمكن مقارنتها فيمكن أن نرتب هذه القائمة. إن تحويل القاموس إلى قائمة من الصفوف هي طريقة تسمح لنا بالحصول على محتويات القاموس مرتبة بطريقة ما:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
```

```
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

وينتج لدينا قائمة مرتبة ترتيبًا أبجديًا حسب قيمة المفتاح.

5.10 الإسناد المتعدد مع القواميس

عند استخدام كل من التابع items وعملية الإسناد للصفوف وحلقة for، يمكن أن نحصل على شيفرة نموذجية تستخدم للمرور على القيم والمفاتيح لقاموس باستخدام حلقة واحدة:

```
for key, val in list(d.items()):
    print(val, key)
```

تحوي هذه الحلقة متغيري تكرار، لأن التابع items تعيد قائمة من الصفوف، والمتغيرين key وval يشكلان تعليمة إسناد لصف يتكرر تنفيذ الحلقة عليه لكل زوج (مفتاح، قيمة) في القاموس، فينتقل المتغيران في كل تكرار للحلقة إلى القيمتين التاليتين في القاموس، ويكون خرج الحلقة كالتالي:

```
10 a
22 c
1 b
```

في حال دمجنا الأسلوبين السابقين يمكن أن نحصل على محتويات القاموس مرتبة وفق القيمة للأزواج (مفتاح، قيمة).

للقيام بهذه العملية يجب علينا أولًا إنشاء قائمة مكونة من صفوف، يتكون كل صف منها من زوج (قيمة، مفتاح) -بدلًا من (مفتاح، قيمة)-، حيث يعطينا التابع items قائمة من الصفوف كل منها مكون من (مفتاح، قيمة) موافقة له، ولكننا نريد في هذا المثال أن نرتب القاموس حسب القيم وليس حسب المفاتيح.

بمجرد حصولنا على قائمة مكونة من صفوف يحوي كل منها (قيمة، مفتاح)، يسهل علينا ترتيب هذه القائمة، ومن ثم ننشئ منها قاموسًا بالترتيب المطلوب.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
```

```
>>> for key, val in d.items() :
... l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

نستطيع من خلال إنشاء قائمة الصفوف هذه بتروٍّ، بحيث تكون القيمة في بداية كل صف من صفوفها، أن نرتب هذه القائمة، ومن ثم يمكن أن ننشئ منها القاموس المطلوب.

6.10 الكلمات الأكثر تكرارًا

بالعودة إلى التمرين السابق، الذي طبقناه على نص المشهد الثاني من الفصل الثاني من مسرحية روميو وجولييت، يمكن أن نكتب برنامجًا يستخدم الطريقة التالية، للحصول الكلمات العشر الأكثر تكرارًا في النص:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
        line = line.translate(str.maketrans(' ', ' ', string.punctuation))
        line = line.lower()
        words = line.split()
        for word in words:
            if word not in counts:
                counts[word] = 1
            else:
                counts[word] += 1
# Sort the dictionary by value
```

```
lst = list()
for key, val in list(counts.items()):
        lst.append((val, key))
lst.sort(reverse=True)
for key, val in lst[:10]:
        print(key, val)
# Code: http://www.py4e.com/code3/count3.py
```

بقي القسم الأول من البرنامج على حاله (القسم الذي يقرأ الملف النصي وينشئ القاموس الذي يحصي عدد الكلمات الموجودة فيه)، ولكن عوضًا عن طباعة أعداد الكلمات وإنهاء البرنامج ببساطة، فسننشئ قائمة من الصفوف للأزواج (قيمة، مفتاح) ثم سنرتب هذه القيم ترتيبًا تنازليًا.

بما أن القيم مذكورة أولًا في الصفوف، سنستخدمها عند المقارنة، وعند وجود أكثر من صف بنفس القيمة سيؤخذ العنصر الثاني (المفتاح) بعين الاعتبار، ولذلك فإن الصفوف التي تبدأ بقيم متماثلة سترتب أبجديًا حسب المفتاح.

في نهاية البرنامج سنكتب حلقة for تقوم بعملية إسناد متعدد مع تكرار، لتقوم بطباعة الكلمات العشر الأكثر تكرارًا عن طريق اجتزاء القائمة الأساسية باستخدام الأمر lst [:10]، وستظهر الكلمات الأكثر تكررًا حسب التحليل الذي أجريناه:

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

يوضح المثال السابق جليًا لِم تُعد بايثون خيارًا مناسبًا كلغة برمجة مستخدمة لاكتشاف المعلومات

حيث استطعنا اجراء تحليل لبيانات معقدة عبر كتابة برنامج بسيط من 19 سطرًا.

7.10 استخدام الصفوف كمفاتيح ضمن القواميس

تتميز الصفوف بإمكانية تطبيق خوارزمية الهاش (Hash) عليها على عكس القوائم، ولذلك فإنها الحل المثالي عند الرغبة في إنشاء قاموس بمفاتيح مركبة.

سنتعامل مع المفاتيح المركبة في حال الرغبة في إنشاء دليل هاتف يستخدم الاسم والكنية كمفتاح ورقم الهاتف كقيمة، وإذا أردنا أن نكتب تعليمة إسناد لإنشاء قاموس، فيمكننا أن نستخدم المتغيرين (first, last) كمفتاح والمتغير number كقيمة، كما في التالي:

```
directory[last,first] = number
```

يمثل التعبير المكتوب بين الأقواس المربعة صفًا، كما يمكن أن نستخدم تعليمة إسناد لصف ضمن حلقة for لتبديل الاسم بموضع الكنية والعكس في هذا القاموس:

```
for last, first in directory:
    print (first, last, directory[last,first])
```

تمر هذه الحلقة على المفاتيح (والتي هي عبارة عن صفوف) في القاموس، حيث أنها تسند عناصر كل صف إلى المتغيرين last وfirst ثم تطبع الاسم ورقم الهاتف المقابل له.

8.10 السلاسل: النصوص والقوائم والصفوف

ركزنا في هذا الفصل على استخدام قوائم من الصفوف، ولكن كل الأمثلة التي طرحناها تقريبًا قابلة للتطبيق على قوائم من القوائم، وصفوف من الصفوف، وصفوف من القوائم، وتجنبًا لتكرار وتعداد كل التركيبات الممكنة، فسنشير إليها بسلاسل من السلاسل تبسيطًا.

يمكن استخدام السلاسل المختلفة (السلاسل النصية والقوائم والصفوف) بشكل متبادل في غالب الأحيان، ولذلك يطرح السؤال التالي: كيف نختار أحد أنواع هذه السلاسل بدلًا من البقية ولماذا؟

نلاحظ بدايةً أن السلاسل النصية أكثر السلاسل محدودية لأن عناصرها يجب أن تكون محارفًا فقط، كما أنها غير قابلة للتعديل فإذا أردت أن تعدل المحارف الموجودة في نص ما (بدلًا عن انشاء نص جديد)، فيفضل أن تستخدم قائمة من المحارف عوضًا عن سلسلة نصية.

تستخدم القوائم بشكل أكثر شيوعا من الصفوف، ويعود ذلك بشكل أساسي إلى كونها قابلة

للتعديل، ولكن هناك بعض الحالات التي قد يكون استخدام الصفوف أفضل فيها:

1- من الأبسط في بعض الحالات كما هو الحال مع تعليمة return، أن ننشئ صفًا بدلًا عن قائمة، ولكن هذا غير مطلق، فقد تُفضل القائمة في بعض الحالات.
2- إذا رغبت أن تستخدم تسلسلًا كمفتاح في قاموس، فستحتاج نوعًا غير قابلٍ للتعديل كالسلاسل النصية أو الصفوف.
3- في حال كنت تستخدم سلسلة ما كوسيط لتابع ما، فإن استخدام الصفوف يقلل احتمالية السلوك غير المتوقع بسبب مشكلة التسمية البديلة (aliasing). لن تستطيع استخدام التوابع sort وreverse مع الصفوف كونها غير قابلة للتعديل، وهذه التوابع تستخدم لتعديل القوائم الموجودة مسبقًا، لكن حال الرغبة بالحصول على نتائج هذه التوابع، فإن لغة بايثون توفر توابعًا جاهزةً بديلة كالتابعين sorted وreversed الذين يأخذان أي سلسلة كمُدخل، ويعيدان سلسلة جديدة بنفس العناصر ولكن بترتيب آخر.

9.10 التنقيح

يُطلق اسم بنى البيانات (data structures) على كل من القوائم والقواميس والصفوف بشكل عام، وقد تناولنا بعض البنى المركبة كقوائم من الصفوف، والقواميس التي تحوي صفوفًا كمفاتيح وقوائمًا كقيم. تعد هذه البنى مفيدة في الاستخدام، ولكنها عرضة لما يسمى بأخطاء الشكل(shape errors) ، وهي الأخطاء التي تحدث عندما تكون بنية البيانات المستخدمة ذات نوع أو حجم أو كلاهما غير مناسب؛ أو حتى من الممكن أن تحدث هذه الأخطاء عند كتابة شيفرة ما ونسيانك لنوع البيانات التي استخدمتها. وكمثال عن ذلك، ففي حال كان لدينا برنامج يتوقع أن تكون البيانات المدخلة له عبارة عن قائمة مكونة من رقم وحيد، وقمنا بتزويده برقم (غير محتوى ضمن قائمة) فسنحصل على هذا النوع من الأخطاء.

10.10 فهرس المصطلحات

- **قابل للمقارنة (comparable):** نوع بيانات يمكن أن يحوي على مجموعة قيم نستطيع أن نفحص فيما إذا كانت أكبر أو أصغر أو تساوي قيم أخرى ضمن نفس النوع، يمكن أن توضع الأنواع القابلة للمقارنة في قائمة ثم يتم ترتيبها بشكل ما.

- **بنى البيانات (data structure):** وهي مجموعة من القيم التي يتم ترتيبها عادة في قوائم أو قواميس أو صفوف أو غيرها.
- **النمط ميّز، رتب، أزِل التمييز (DSU) اختصارًا للتعبير (Decorate, Sort, Undecorate):** وهو نمط يستخدم لتشكيل قوائم من صفوف، ليتم ترتيبها واستخراج جزء من النتيجة التي نحصل عليها.
- **التجميع (gather):** وهي العملية التي يتم فيها تجميع وسيط مكون من صف ذو طول متغير.
- **خاضع لخوارزمية الهاش (hashable):** أي نوع بيانات يمكن تنفيذ تابع الهاش (hash) عليه، الأنواع غير القابلة للتعديل مثل (float, integers, string) تقبل هذا التابع، أما البنى القابلة للتعديل لا تقبله.
- **التفريق (scatter):** وهي العملية التي يتم فيها التعامل مع سلسلة من البيانات على أنها قائمة من الوسائط.
- **شكل بنية البيانات (shape):** ملخص يصف نوع وحجم وتركيب بنية معطيات ما.
- **ذو العنصر الوحيد (singleton):** قائمة (أو سلسلة من نوع آخر) تحوي عنصرًا واحدًا فقط.
- **الصف (tuple):** سلسلة من العناصر غير القابلة للتعديل.
- **إسناد الصفوف (tuple assignment):** وهي تعليمة إسناد لسلسلة من القيم موجودة في طرفها الأيمن، وصف من المتغيرات في طرفها الأيسر، يتم حساب الطرف الأيمن أولًا، ثم يتم إسناد العناصر الموجودة فيه إلى المتغيرات الموجودة في الطرف الأيسر.

11.10 تمارين

- **التمرين الأول:** راجع تمرينا السابق، واكتب برنامجًا يقوم بقراءة وتجزئة السطور التي تبدأ بكلمة From، ويستخرج العنوان من كل سطر، ثم يقوم بحساب عدد الرسائل الواردة من كل شخص باستخدام القاموس.

بعد قراءة كل البيانات اعرض اسم الشخص ذو عدد الرسائل الأكبر، عن طريق إنشاء قائمة

تتضمن صفوفًا (لعدد الرسائل، وعنوان البريد) من القاموس الذي تم إنشاؤه سابقًا، ثم رتب القيم من الأكبر إلى الأصغر، واعرض عنوان البريد الذي ورد منه أكبر عدد من الرسائل.

مثال:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

- **التمرين الثاني:** يعد البرنامج الذي سنكتبه في هذا التمرين تكرار الساعات التي وصلت فيها رسائل البريد الالكتروني، حيث سنستخرج ساعة ورود الرسالة من السطر المبدوء بالكلمة From، عن طريق العثور على النص الذي يرمّز ساعة الوصول، ومن ثم تجزئته باستخدام عامل النقطتين : ، بمجرد القيام بِعد جميع ساعات وصول الرسائل، اعرض تكرار كل ساعة في سطر كالتالي:

```
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
```

```
17 2
18 1
19 1
```

- **التمرين الثالث:** اكتب برنامجًا يقرأ ملفًا نصيًا ليعرض عدد تكرار الأحرف فيه بترتيب تنازلي. يجب أن يحول البرنامج كل الأحرف إلى حالة الحرف الصغير، وأن يعد الأحرف من a إلى z فقط (ألا يعد أي شيء أخر عدا الأحرف كالمسافات وعلامات الترقيم)، بعد ذلك استخدم ملفات نصية من عدة لغات كمُدخل للبرنامج، وقارن تباين الأحرف الأكثر تكرارًا بين هذه اللغات وقارن النتائج التي توصلت إليها مع الجدول الموجود في الرابط التالي:

https://wikipedia.org/wiki/Letter_frequencies

الفصل الحادي عشر

التعابير النمطية

11 التعابير النمطية

حتى الآن كنا نقرأ الملفات ونبحث عن الأنماط ونقوم باستخراج بيانات مهمة بالنسبة لنا من الأسطر، حيث كنا نستخدم توابع السلاسل النصية مثل split وfind، ونستخدم القوائم وتجزئة السلاسل النصية لاستخراج أجزاء من الأسطر.

مهمة البحث والاستخراج هذه شائعة جدًا، لذلك تتوفر في لغة بايثون مكتبة برمجية فعالة جدًا تسمى مكتبة التعابير النمطية (*regular expressions*)والتي تتعامل مع العديد من هذه المهام برتابة مطلقة، وإن السبب في عدم طرح التعابير النمطية سابقًا في الكتاب هو أنه بالرغم من أن هذه التعابير فعّالة لكن بنفس الوقت معقدة قليلًا والقواعد الخاصة بها تتطلب ممارسةً للاعتياد عليها.

يمكن أن نقول عن التعابير النمطية أنها لغة برمجة خاصة بعمليات البحث والتحليل في السلاسل النصية.

لقد نُشرت كتب كاملة عن التعابير النمطية لذلك سنغطي في هذا الفصل أساسيات التعابير النمطية فق، وللمزيد من التفاصيل حول التعابير النمطية راجع الرابطين التاليين:

https://en.wikipedia.org/wiki/Regular_expression

https://docs.python.org/library/re.html

يجب أن تُستدعى مكتبة التعبير النمطي re ضمن برنامجك قبل استخدامها، وأبسط استخدام لها هو التابع search() والبرنامج التالي يوضح أحد استخداماته:

```
# Search for lines that contain 'From'
import re
hand = open ('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line)
        print(line)

# Code: http://www.py4e.com/code3/re01.py
```

نفتح الملف، ثم نمر على كل السطور باستخدام حلقة for ثم نستخدم التعبير النمطي search() فقط لطباعة الأسطر التي تحوي السلسلة النصية "From".

هذا البرنامج لا يظهر الفعالية الحقيقية للتعابير النمطية حيث كان بإمكاننا الحصول على نفس النتائج بالسهولة ذاتها باستخدام التابع line.find()، وتظهر الفعالية الحقيقية للتعابير النمطية عندما يمكننا إضافة الرموز الخاصة بالتعابير النمطية للسلسلة النصية والتي تسمح لنا بالتحكم بدقة أكبر بالأسطر التي تطابق سلسلة نصية ما.

تسمح إضافة هذه الرموز الخاصة لتعبيرنا النمطي بالقيام بعمليات مطابقة واستخراج متقدمة باستخدام عدد قليل من السطور البرمجية، فعلى سبيل المثال الرمز ^ يستخدم في التعبير النمطي لمطابقة بداية السطر بحيث بإمكاننا تغيير البرنامج السابق لمطابقة الأسطر بحيث "From" في بداية السطر فقط كما يلي:

```
# Search for lines that start with 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line):
        print(line)

# Code: http://www.py4e.com/code3/re02.py
```

هكذا نكون حصلنا فقط على الأسطر التي تبدأ ب "From:" وهذا مثال بسيط جدًا كان بالإمكان تنفيذه باستخدام التابع startswith() .

لكنه يهدف لتوضيح حقيقة بأن التعابير النمطية تستخدم رموز خاصة لمنحنا المزيد من التحكم بعمليات المطابقة.

1.11 مطابقة المحارف في التعابير النمطية

هناك عدد من الرموز الخاصة التي تسمح لنا ببناء تعابير نمطية أكثر فعالية ومن أكثرها استخدامًا وشيوعًا هي النقطة . والتي تمثل أي محرف.

في المثال التالي التعبير النمطي F..m: سيطابق أي من السلاسل النصية From أو Fxxm أو F12m أو

F!@m حيث النقط في التعبير النمطي تطابق أي محرف.

```
# Search for lines that start with 'F', followed by
# 2 characters, followed by 'm:'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line):
        print(line)


# Code: http://www.py4e.com/code3/re03.py
```

تزداد أهمية هذه الميزة عندما يمكننا الإشارة لإمكانية تكرار المحرف عددًا من المرات باستخدام رمز النجمة * أو رمز الزائد + في تعبيرك النمطي حيث تدعى الرموز * و + بـ (wildcard) تعني هذه المحارف الخاصة أنه بدلًا من مطابقة محرف واحد في السلسلة النصية فإنها تطابق في حال الرمز * صفر محرف أو أكثر من المحارف، أما في حال الرمز + تطابق محرف واحد أو أكثر.

يمكننا تضييق نطاق الأسطر التي نطابقها باستخدام الرموز السابقة في المثال التالي:

```
# Search for lines that start with From and have an at sign
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line):
        print(line)
# Code: http://www.py4e.com/code3/re04.py
```

السلسلة النصية ^From:.+@ ستطابق الأسطر التي تبدأ بـ From: متبوعة بمحرف أو أكثر +. ثم بإشارة @

لذلك هذا سوف يطابق السطر التالي:

```
From: stephen.marquard@uct.ac.za
```

يمكننا بهذه الحالة أن نقول أن الرمز+. يطابق جميع المحارف بين النقطتين : وإشارة @

```
From:.+@
```

يمكن الاعتبار أن الرمزين * و + رموز متعدية (أي أنّها تطابق أكبر قدر ممكن من المحارف)، فعلى سبيل المثال إن السلسلة النصية أدناه التي تتضمن عدة محارف @ لكن سيكمل الرمز+. المطابقة حتى محرف @ الأخير.

```
From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu
```

لكن من الممكن توظيف رمز * أو + بحيث لا يكون متعديًا في المطابقة عبر إضافة رمز خاص في التعبير النمطي، لذا راجع ملفات توثيق هذا المكتبة للحصول على معلومات عن إيقاف تشغيل السلوك الطماع لتلك الرموز.

2.11 استخراج البيانات باستخدام التعابير النمطية

إذا أردنا استخراج البيانات من سلسلة نصية في لغة بايثون فبإمكاننا استخدام التابع findall() لاستخراج السلاسل النصية الجزئية (substrings) التي تطابق التعبير النمطي.

على سبيل المثال لاستخراج أي سلسلة نصية قد تبدو كبريد الالكتروني من كل من الأسطر التالية:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
        for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

لن نرغب بكتابة شيفرة تتضمن تعليمات تجزئة وتقطيع مختلفة للتعامل مع كل سطر على حدة، بل نستخدم تابع findall() لإيجاد الأسطر التي تحوي عناوين البريد الإلكتروني واستخراج واحد أو أكثر من العناوين في كل الأسطر.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)
# Code: http://www.py4e.com/code3/re05.py
```

يبحث التابع findall() في الوسيط الثاني للتابع – من نوع سلسلة نصية - ويعيد قائمة بكل سلسلة نصية تبدو كالبريد الالكتروني أما الرمز \s فيستخدم للتعبير عن عدم وجود مسافات فارغة.

فيكون خرج البرنامج:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

وتفسير هذا التعبير النمطي يكون بأننا نبحث عن سلسلة نصية جزئية تحوي على الأقل محرف واحد لا يمثل مسافة فارغة متبوع بإشارة @ متبوعة على الأقل برمز واحد لا يمثل مسافة فارغة حيث الرمز \S+ يعني مطابقة أي عدد من المحارف باستثناء المسافة الفارغة.

سيطابق التعبير النمطي مرتين مع (csev@umich.edu , cwen@iupui.edu)

ولكن لن يطابق 2PM@ بسبب وجود مسافة فارغة قبل إشارة @

بإمكاننا استخدام التعبير النمطي السابق في البرنامج لقراءة كل الأسطر في الملف وطباعة أي شيء يشبه البريد الإلكتروني كما يلي:

```
# Search for lines that have an at sign between characters
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0:
        print(x)


# Code: http://www.py4e.com/code3/re06.py
```

إننا نقرأ كل سطر ثم نستخرج كل سلسلة نصية جزئية والتي تطابق تعبيرنا النمطي.

بما أن التابع findall() يعيد قائمة فيمكن ببساطة أن نتحقق إذا كان عدد العناصر في القائمة المعادة أكبر من صفر.

إذا شغلنا البرنامج على الملف mbox.txt سنحصل على الخرج التالي:

```
['wagnermr@iupui.edu']
```

```
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

تحوي بعض عناوين البريد الإلكتروني رموز مثل < أو ; في بدايتها أو نهايتها، ولنوضح أننا فقط مهتمين في الجزء الذي يبدأ أو ينتهي بحرف أو رقم، لذلك نستخدم ميزات أخرى من ميزات التعبير النمطي، فالأقواس المربعة تُستخدم لتوضيح مجموعة من الرموز المتعددة المقبولة والتي نرغب باعتبارها متطابقة فسابقًا تعلمنا أن \S تطابق أي محرف بخلاف المسافات الفارغة. الآن سنكون أكثر دقّة في المحارف التي سنطابقها.

هنا هو تعبيرنا النمطي الجديد:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

يصبح الموضوع أكثر تعقيدًا وقد تدرك لماذا التعابير النمطية هي لغة خاصة بذاتها.

إن تفسير هذا التعبير النمطي هو أننا نبحث عن سلسلة نصية جزئية تبدأ بحرف صغير أو حرف كبير أو رقم [a-zA-Z0-9]، ثم متبوع بصفر أو أي عدد من المحارف بخلاف المسافة الفارغة \S* ثم بإشارة @، ثم بصفر أو عدد أكبر من المحارف خلاف المسافة الفارغة \S* متبوع بحرف كبير أو صغير.

لاحظ أننا أبدلنا من + إلى * لنشير لصفر أو أكثر من المحارف خلاف المسافة الفارغة حيث

[a-zA-Z0-9] هي واحدة من المحارف خلاف المسافة الفارغة.

تذكر أن رمزي * أو + تطبق للرمز مباشرة الموجود إلى يسارهما

عندما نستخدم هذا التعبير في برنامجنا ستكون بياناتنا أكثر رتابة:

```
# Search for lines that have an at sign between characters
# The characters must be a letter or number

import re
```

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S+@\S+[a-zA-Z]', line)
    if len(x) > 0:
        print(x)
# Code: http://www.py4e.com/code3/re07.py
```

```
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

لاحظ في الأسطر source@collab.sakaiproject.org إن تعبيرنا النمطي استبعد حرفين من نهاية السلسة النصية ;> ، وذلك لأنه عندما نضيف [a-zA-Z] لنهاية التعبير النمطي فإننا نطلب أنه مهما كانت السلسلة النصية فإن عليها أن تنتهي بحرف لذلك عندما توجد > في نهاية "sakaiproject.org>;" فستتوقف المطابقة عند أخر حرف.

لاحظ أيضًا أن خرج البرنامج هو قائمة كل عنصر فيها هو من النوع سلسلة نصية.

3.11 تنفيذ عمليتي البحث والاستخراج معاً

إذا أردنا الحصول على الأرقام في الأسطر التي تبدأ بسلسلة نصية "X-" مثل:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

لا نريد فقط أي أعداد عشرية من أي سطر إنما نريد استخراج الأرقام من الأسطر ذات الصيغة أعلاه

لنستخدم التعبير النمطي التالي لاختيار الأسطر:

```
^X-.*: [0-9.]+
```

أي أننا نريد الأسطر التي تبدأ بـ X- متبوعة بصفر أو أكثر من المحارف *. ومتبوعة بنقطتين : ثم فراغ وبعد الفراغ نبحث عن محرف أو مجموعة من المحارف والتي ممكن أن تكون أرقام بين صفر حتى تسعة أو ذات فاصلة عشرية [0-9.]+

لاحظ داخل الأقواس المربعة إن النقطة تطابق فاصلة عشرية (ليست رمز النقطة الخاص بالتعابير النمطية).

إن التعبير التالي دقيق جدًا وسيطابق تمامًا الأسطر المطلوبة:

```
# Search for lines that start with 'X' followed by any non
# whitespace characters and ':'
# followed by a space and any number.
# The number can include a decimal.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line):
        print(line)


# Code: http://www.py4e.com/code3/re10.py
```

عندما نقوم بتشغيل البرنامج نرى البيانات تظهر بشكل واضح الأسطر التي نبحث عنها فقط.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

لحل مشكلة استخراج الأرقام بإمكاننا استخدام التابع split أو نستطيع استخدام ميزة أخرى من ميزات التعابير النمطية بحيث نقوم بعمليتي البحث وتحليل السطور في نفس الوقت.

إن إشارة القوسين () هي أحد رموز التعابير النمطية الخاصة لكنها لا تستخدم في عمليات المطابقة

بل مع التابع findall() حيث تشير إلى أنه بالرغم من سعيك لمطابقة كل التعبير لكنك فقط مهتم باستخراج جزء محدد من السلسلة النصية الجزئية المطابقة.

لذا قم بإجراء التغيير التالي لبرنامجنا:

```
# Search for lines that start with 'X' followed by any
# non whitespace characters and ':' followed by a space
# and any number. The number can include a decimal.
# Then print the number if it is greater than zero.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)
# Code: http://www.py4e.com/code3/re11.py
```

بدلًا من استدعاء search() بإمكاننا إضافة قوسين حول جزء من التعبير النمطي الذي يمثل عدد عشري، ليوضح أننا فقط نريد من التابع findall() أن يعطينا العدد ذي الفاصلة العشرية من السلسة النصية المطابقة.

يظهر خرج البرنامج كما يلي:

```
['0.8475']

['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
```

بالرغم أنّ الأرقام خزنت في قائمة، علينا إجراء تحويل من النوع سلسلة نصية إلى نوع عدد ذي الفاصلة العشرية، لكن يظهر المثال السابق فعالية التعابير النمطية لكل من عمليتي البحث والاستخراج.

كمثالٍ آخر عن هذه التقنية إذا ألقينا نظرة على الملف، نلاحظ أنه يتضمن عدد من الأسطر بالصيغة التالية:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

إذا أردنا استخراج كل أرقام المراجعة (rev) (العدد الصحيح في نهاية هذه الأسطر) باستخدام نفس الطريقة أعلاه فبإمكاننا كتابة البرنامج التالي:

```
# Search for lines that start with 'Details: rev='
# followed by numbers and '.'
# Then print the number if it is greater than zero
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9.]+)', line)
    if len(x) > 0:
        print(x)
# Code: http://www.py4e.com/code3/re12.py
```

لتفسير تعبيرنا النمطي فنحن نبحث عن أسطر والتي تبدأ بكلمة Details: متبوعة بأي عدد من المحارف *. متبوعة بـ rev= وثم بواحد أو أكثر من الأرقام. ولأننا لا نريد إيجاد الأسطر التي تطابق كل التعبير بل فقط نريد استخراج العدد الصحيح في نهاية السطر لذلك نحيط +[0-9] بقوسين.

عندما نقوم بتشغيل البرنامج فنحصل على الخرج التالي:

```
['39772']
['39771']
['39770']
['39769']
...
```

تذكر أن في التعبير +[0-9] رمز + متعدٍ أي سيحاول الحصول على أكبر سلسلة نصية ممكنة قبل استخراج الأرقام، يفسر هذا السلوك لماذا نحصل على خمس خانات لكل رقم.

انتبه إلى أن مكتبة التعابير النمطية تتوسع في كلا الاتجاهين حتى تقابل محرف غير الرقم، أي نحو

بداية و نهاية السطر.

الآن يمكننا استخدام التعابير النمطية لحل تمارين سابقة حيث كان محط اهتمامنا هو الوقت واليوم لكل رسالة بريدية:

حيث نظرنا سابقا إلى الأسطر بالصيغة:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

وأردنا استخراج الوقت لكل سطر، فسابقًا فعلنا ذلك باستدعاء التابع split مرتين:

أولًا عبر تجزئة السطر إلى كلمات ثم سُحبت الكلمة الخامسة ثم جزأناه مجددًا عبر عامل النقطتين : لسحب المحرفين المهتمين بهما، ولقد نجح ذلك ولكن هذه الطريقة ليس عملية حيث تفترض أن الأسطر ذات صيغة قياسية.

إذا أردتَ إضافة عملية تحقق من الأخطاء (أو كتلة تعليمات لبنية try/except)، لضمان عدم إخفاق برنامجك عندما تدخل له أسطر غير منسقة تنسيقًا صحيحًا فسيزداد حجم الشيفرة 10-15 سطرًا برمجيًا مما يجعل البرنامج صعب الفهم.

يمكن تبسيط ذلك باستخدام التعبير النمطي:

```
^From .* [0-9][0-9]:
```

تفسير هذا التعبير النمطي أننا نبحث عن الأسطر التي تبدأ بالكلمة From ثم فراغ متبوعًا بأي عدد من المحارف *. متبوعًا بفراغ متبوعًا برقمين [0-9] [0-9] متبوعًا بنقطتين :

هذا التعبير مناسب للأسطر التي نبحث عنها.

لاستخراج الساعة فقط باستخدام findall() نضيف قوسين حول الرقمين كما يلي:

```
^From .* ([0-9][0-9]):
```

فيكون البرنامج كالتالي:

```
# Search for lines that start with From and a character
# followed by a two digit number between 00 and 99 followed by ':'
# Then print the number if it is greater than zero
import re
hand = open('mbox-short.txt')
```

```
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0:    print(x)


# Code: http://www.py4e.com/code3/re13.py
```

وستعطي الخرج التالي عندما يتم تشغيل البرنامج:

```
['09']
['18']
['16']
['15']
...
```

4.11 محرف الهروب

عند استخدامنا لرموز التعابير النمطية لمطابقة بداية أو نهاية السطر أو الرموز الخاصة كـ * و + فإننا نحتاج طريقة لتفريقها عن المحارف العادية والتي قد نريد مطابقتها كإشارة $ أو ^

نضع رمز \ (الشرطة المائلة للخلف) لتبيان أننا نبحث عن مطابقة هذا المحرف وليس رمزًا من رموز التعابير النمطية. فعلى سبيل المثال بإمكاننا إيجاد قيم مالية في نص باستخدام التعبير النمطي التالي:

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

عندما نضع الشرطة المائلة للخلف قبل علامة الدولار $ فإنها تبحث حقًا عن إشارة $ في السلسلة النصية بدلًا من مطابقتها في "نهاية السطر"، ويطابق بقية التعبير النمطي رقم أو مجموعة أرقام.

لاحظ أنه داخل الأقواس المربعة، المحارف ليست خاصة بالتعبير النمطي، لذلك عندما نقول [0-9.] فهو يعني حقًا أرقام أو ذات فاصلة عشرية، أما خارج تلك الأقواس تعد النقطة أحد رموز التعبير النمطي وتطابق أي محرف. أي أن النقطة داخل الأقواس المربعة هي نقطة عادية تشير للأرقام ذات الفاصلة العشرية.

5.11 ملخص

هذا الفصل هو لمحة عن التعابير النمطية فقد تعلمنا قليلًا عن لغة التعابير النمطية. فهي طريقة للبحث في السلاسل النصية تعتمد على رموز خاصة تمكنك من التعبير عما تريد البحث عنه بلغة التعابير النمطية وهذا ما يعرف بالمطابقة وتمكنك من استخراج أجزاء محددة من تلك السلاسل النصية المطابقة.

إليك بعض الرموز الخاصة:

- ^ تطابق بداية السطر
- $ تطابق نهاية السطر
- . تطابق أي محرف تسمى (wildcard)
- \s تطابق المسافات الفارغة -انتبه حرف s هنا حرف صغير-.
- \S تطابق أي محرف خلاف المسافات الفارغة (أي عكس \s) تطبّق على المحرف أو المحارف التي تسبقها بشكل مباشر وتشير لمطابقة صفر أو أكثر من المرات
- *؟ تطبّق على المحرف أو المحارف التي تسبقها بشكل مباشر وتشير لمطابقة صفر أو أكثر من المرات (في الوضع غير المتعدي (أي غير الطماع))
- + تطبّق على المحرف أو المحارف التي تسبقها بشكل مباشر وتشير لمطابقة مرة أو أكثر من المرات
- +؟ تطبّق على المحرف أو المحارف التي تسبقها بشكل مباشر وتشير لمطابقة مرة أو أكثر من المرات (في الوضع غير المتعدي)
- ؟ تطبّق على المحرف أو المحارف التي تسبقها بشكل مباشر وتشير لمطابقة صفر أو مرة واحدة
- ؟؟ تطبّق على المحرف أو المحارف التي تسبقها بشكل مباشر وتشير لمطابقة صفر أو مرة واحدة (في الوضع غير المتعدي)
- [aeiou] تطابق حرف وحيد طالما أن المحرف في مجموعة معينة في هذا المثال ستطابق

"a" أو "e" أو "I" أو "o" أو "u" لكن لن تطابق أي محارف أخرى.

- [a-z0-9] بإمكانك تحديد نطاق المحارف باستخدام إشارة الناقص - ، وهذا المثال هو محرف وحيد يجب أن يكون حرف صغير أو رقم.
- [^A-Za-z] عندما أول رمز في مجموعة الرموز هو ^ فهو يعكس الحالة، وهذا المثال يطابق محرف وحيد مطابق لأي شيء إلّا حرف كبير أو صغير.
- () حين تُضاف الأقواس للتعابير النمطية لا تكون بغرض المطابقة، بل لاستخراج مجموعة فرعية معينة من السلسة النصية التي تمت مطابقتها.
- \b تطابق سلسلة نصية فارغة، لكن فقط في بداية أو نهاية الكلمة.
- \B تطابق سلسلة نصية فارغة، لكن ليست في بداية أو نهاية الكلمة.
- \d تطابق أي رقم من أرقام النظام العشري، وهذا مماثل للتعبير [0-9]
- \D تطابق أي محرف ليس رقم، وهذا مماثل [^0-9]

6.11 معلومات إضافية لمستخدمي نظامي Unix و Linux

أضيف البحث عن الملفات باستخدام التعابير النمطية في نظام تشغيل Unix عام 1960 وهو متاح في أغلب لغات البرمجة بشكل أو بآخر.

في الواقع يوجد برنامج أوامر(command-line) ضمن Unix ويسمى grep (محلل التعابير النمطية العام) والذي يعمل تقريبًا مثل الأمثلة التي استخدمنا فيها تابع search() في هذا الفصل، لذا إذا كان لديك نظام Macintosh أو Linux فبإمكانك تجربة الأوامر التالية في نافذة برنامج الأوامر:

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

يطلب الأمر السابق من برنامج grep أن يظهر لك الأسطر التي تبدأ بالسلسلة النصية From في الملف mbox-short.txt.

إذا جربت برنامج grep قليلًا وقرأت ملفات التوثيق الخاصة به سترى بعض الاختلافات الدقيقة بين التعابير النمطية في لغة بايثون والتعابير النمطي في grep، فمثلًا grep لا يدعم رمز \S فستحتاج إلى مجموعة رموز أكثر تعقيدًا [^] والذي يعني ببساطة مطابقة أي محرف عدا الفراغ.

7.11 التنقيح

تحوي لغة بايثون ملفات توثيق سهلة ومفيدة جدًا في حال احتجت منشط سريع لتحفيز ذاكرتك لاسترجاع اسم تابع ما، حيث يمكن عرض هذه الملفات في مفسر لغة بايثون في الوضع التفاعلي.

يمكنك جلب نظام المساعدة التفاعلي باستخدام help()

```
>>> help()
help> modules
```

إذا كنت تعلم أي وحدة (module) تريد استخدامها يمكنك استخدام أمر dir() لإيجاد التوابع في الوحدة كما يلي:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer' , 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse' , 'sub', 'subn', 'sys', 'template']
```

بإمكانك أيضا الحصول على توثيق مختصر عن أحد التوابع باستخدام الأمر help

```
>>> help (re.search)
Help on function search in module re:


search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

إن ملفات التوثيق ليست شاملة لكنها مفيدة في حال احتجت لمعلومة بسرعة أو عندما لا يكون لديك وصول إلى متصفح ويب أو محرك بحث.

8.11 فهرس المصطلحات

- **الشيفرة الهشة (brittle code):** هي الشيفرة التي تعمل عندما تكون بيانات الدخل في صيغة معينة قياسية لكنها ضعيفة جدًا إذا كان هناك بعض التغيرات عن الصيغة القياسية، ونسميها هشة لأنه من السهل كسرها.
- **المطابقة الطماعة (greedy matching):** الفكرة أن رموز + و * في التعبير النمطي تمتد لمطابقة أكبر عدد ممكن من محارف السلسلة النصية.
- **محلل التعابير النمطية العام (grep):** أمر متاح في أنظمة Unix يبحث عبر الملفات النصية لإيجاد أسطر تطابق التعابير النمطية، وهو اختصار لجملة (Generalized regular expression parser)
- **التعبير النمطي (regular expression):** لغة للبحث في السلاسل النصية، حيث يحوي التعبير النمطي رموزًا خاصة تظهر أن البحث فقط سيطابق بداية ونهاية أسطر بالإضافة العديد من الميزات المماثلة.
- **الرموز البديلة (Wildcard):** رمز خاص يطابق أي محرف، أحدها هو النقطة.

9.11 تمارين

- **التمرين الأول:** اكتب برنامج بسيط لمحاكاة عملية أمر grep في نظام Unix، واطلب من المستخدم إدخال تعبير نمطي واحسب عدد الأسطر التي تطابق التعبير النمطي.

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-
```

```
$ python grep.py
Enter a regular expression: java$
mbox.txt had 4175 lines that matched java$
```

- **التمرين الثاني:** اكتب برنامج ليبحث عن أسطر تحوي صيغة مشابهة لما يلي:

```
New Revision: 39772
```

استخرج العدد من كل سطر باستخدام تعبير نمطي والتابع findall() واحسب متوسط الأعداد واطبع المتوسط كعدد صحيح.

```
Enter file:mbox.txt
38549
Enter file:mbox-short.txt
39756
```

الفصل الثاني عشر

البرامج المرتبطة بالشبكات

12 البرامج المرتبطة بالشبكات

ركزنا في العديد من الأمثلة الواردة في هذا الكتاب على قراءة الملفّات والبحث عن بيانات ضمنها، إلا أن هناك العديد من مصادر المعلومات المختلفة كشبكة الإنترنت.

في هذا الفصل، سنعمل عمل مُتصفّح الإنترنت الذي يسترجع صفحات الويب باستخدام بروتوكول نقل النص التشعّبي (Hypertext Transfer Protocol)، بعد ذلك سنقرأ ونُحلّل بيانات تلك الصفحات.

1.12 برتوكول نقل النص التشعّبي HTTP

إنّ بروتوكول الشبكة الذي يحكم عمل شبكة الويب بسيط للغاية. كما تسهل المكتبة البرمجية الجاهزة في بايثون socket عملية إنشاء اتصالات عبر الشبكة واسترجاع البيانات عبر مآخذ الشبكة (Sockets) في برنامج بايثون.

تُشبه مآخذ الشبكة الملف إلى حد ما، لكن يكمن الاختلاف في أنّها تؤمّن إمكانيّة اتصال ثنائي الاتجاه بين برنامجين. حيث تستطيع القراءة والكتابة عبر مآخذ الشبكة ذاتها.

فإذا قُمتَ بكتابة شيء ما إلى مآخذ الشبكة، فإنّه يُرسَل إلى التطبيق في الجانب الآخر. بينما إذا قُمت بالقراءة منه فإنّ البيانات الواردة إليك مُرسلة من قِبل تطبيق آخر.

يجب عليك الانتظار عند محاولة قراءة مآخذ الشبكة في حال لم يرسل البرنامج في الطرف الآخر أيّ بيانات. إذا انتظرت البرامج في طرفي مآخذ الشبكة وصول بيانات بدون إرسال أي شيء، فلا شك أنها ستنتظر طويلًا. لذلك من المهم أن تتبع البرامج التي تتواصل عبر الإنترنت بروتوكولًا محددًا.

البروتوكول، هو مجموعة من القواعد تُحدّد أيُّ طرف سيبدأ في الاتصال أولًا وماذا سيُنفّذان، ثمّ ما هي الردود لتلك الرسالة، ومن سيُرسل تاليًا، وهكذا.

بمعنى أن التطبيقين على طرفي مآخذ الشبكة يتبعان خطوات متوافقة بدون أيّ تعارض.

تتوفر العديد من المستندات التي تشرح برتوكولات الشبكة. تجد بروتوكول نقل النص التشعبي HTTP مُوضّحًا في المستند التالي: https://www.w3.org/Protocols/rfc2616/rfc2616.txt

هذا المُستند طويل ومعقد من 176 صفحة مليء بالكثير من التفاصيل.

إذا وجدتَ أنّه مهم فلا تتردد بقراءته بالكامل، لكن إذا أردت العثور على القواعد حول طلبات GET فعليك الاطّلاع على الصفحة رقم 36 من المستند الموافق للرقم RFC2616.

لطلب مُستند من مخدم ويب سنُجري اتصالًا مع مخدّم الموقع www.pr4e.org على المنفذ (port) رقم 80 ثم نرسل أمرًا كالتالي:

```
GET http://data.pr4e.org/romeo.txt HTTP/1.0
```

بحيث يكون المعامل الثاني هو صفحة الويب التي طلبناها، ثم نقوم أيضًا بإرسال سطر فارغ. سيستجيب خادم الويب بإرسال بعض المعلومات الرئيسية عن المستند وسطر فارغ متبوعًا بمحتوى المستند.

2.12 مُتصفّح الويب الأبسط في العالم

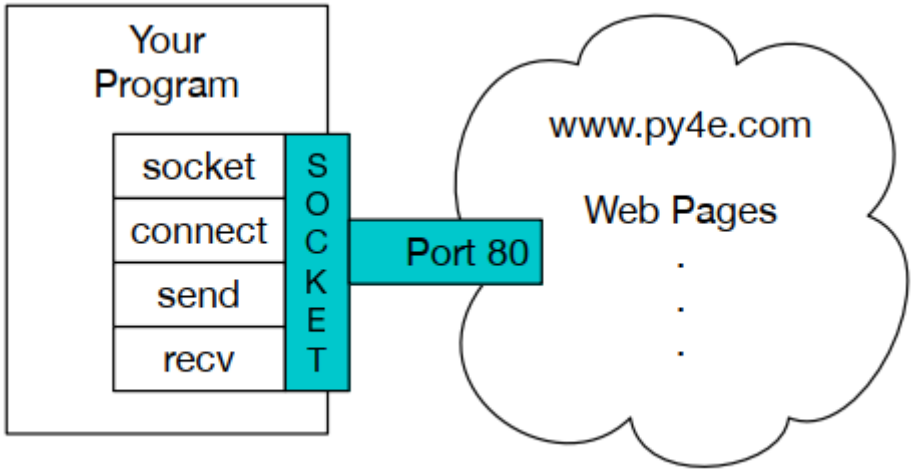
ربّما الطريقة الأسهل لإيضاح آلية عمل بروتوكول HTTP هي بكتابة برنامج بايثون بسيط يقوم بالاتصال خادم الويب وفق قواعد بروتوكول HTTP لطلب المستند ثم عرض الرد الذي يُرسله المخدم.

```
import socket


mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
mysock.send(cmd)
while True:
        data = mysock.recv(512)
        if len(data) < 1:
            break
        print(data.decode(),end=' ')
mysock.close()


# Code: http://www.py4e.com/code3/socket1.py
```

يقوم البرنامج في البداية بالاتصال مع المنفذ 80 على الخادم www.py4e.com بما أن برنامجنا يؤدّي دور مُتصفّح الإنترنت فإن بروتوكول نقل النص التشعّبي يفرض علينا أن نُرسل أمر GET متبوعًا بسطر فارغ.

الرّموز \r\n تُشير إلى EOL (End Of Line) أَيْ "نهاية السطر". لذا فإن الرموز \r\n\r\n تُشير إلى عدم وجود شيء بين تتابعي نهاية سطرين. وهذا يُكافئ السطر الفارغ.

بمجرّد إرسال السطر الفارغ نقوم بإنشاء حلقة تستقبل البيانات على شكل أجزاء بحجم 512 محرف للجزء الواحد من مآخذ الشبكة، ونستمر بطباعة البيانات حتى لا يبقى أيّ بيانات للقراءة، أي حتى يُعيد التابع recv() سلسلة نصية فارغة.



الشكل 13 : نموذج اتصال عبر مآخذ الشبكة

يُنتِج البرنامج الخرج التالي:

```
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:52:55 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Sat, 13 May 2017 11:22:22 GMT
ETag: "a7-54f6609245537"
Accept-Ranges: bytes
Content-Length: 167
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
```

```
Connection: close
Content-Type: text/plain


But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

يظهر في بداية الخرج الترويسة (header) التي أرسلها الخادم لوصف المُستند.

على سبيل المثال، تشير عبارة Content_Type إلى أن المستند هو مستند نصّي عادي (text/plain).

يُضيف الخادم بعد أن يُرسل لنا الترويسة سطر فارغ للإشارة إلى نهايتها، ثم بعد ذلك يُرسل البيانات الفعلية وهي الملف النصي romeo.txt.

يُوضّح هذا المثال كيفية إجراء اتصال شبكي منخفض المستوى بواسطة مآخذ الشبكة. حيث يمكن أن تستخدم مآخذ الشبكة للاتصال بخادم الويب أو خادم البريد أو أي خوادم أخرى. فكل ما هو مطلوب هو العثور على المستند الذي يشرح مبدأ عمل البروتوكول ومن ثم كتابة الشيفرة البرمجية لإرسال واستقبال البيانات وفقًا له.

على أيّ حال، بما أن البروتوكول الشائع استخدامه هو بروتوكول الويب HTTP فإن لغة بايثون تحتوي مكتبة صُمّمت خصيصًا لتدعمه وخصيصًا عمليات استرجاع المستندات والبيانات عبر الويب.

أحد مُتطلّبات استخدام بروتوكول HTTP هو إرسال واستقبال البيانات على أنّها سلسلة من البايتات (Bytes Objects) بدلًا من اعتبارها سلاسل نّصية، ففي المثال السابق، يحوّل التابعان encode() وَ decode() السلاسل النصية إلى سلسلة من البايتات وبالعكس.

يستخدم المثال التالي الرمز b'' لتخزين المتغير كسلسلة بايتات. إن كلّ من b'' وَ encode() مُتكافِئان.

```
>>> b'Hello world'
b'Hello world'
>>> 'Hello world'.encode()
b'Hello world'
```

3.12 استعادة صورة عن طريق بروتوكول HTTP

في المثال أعلاه، استعدنا ملف نصّي، وعرضنا ببساطة البيانات إلى الشاشة عند تنفيذ البرنامج. يمكننا استخدام برنامج مشابه لاستعادة صورة عن طريق HTTP، فبدلًا من عرض البيانات على الشاشة عند تنفيذ البرنامج، نقوم بتجميع البيانات في سلسلة وبعدها نحذف الترويسة ثم نحفظ بيانات الصورة في ملف كما هو موضّح:

```
import socket
import time

HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))
mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')
count = 0
picture = b" "

while True:
    data = mysock.recv(5120)
    if len(data) < 1: break
    #time.sleep(0.25)
    count = count + len(data)
    print(len(data), count)
    picture = picture + data

mysock.close()
```

```
# Look for the end of the header (2 CRLF)

pos = picture.find(b"\r\n\r\n")
print('Header length', pos)
print(picture[:pos].decode())

# Skip past the header and save the picture data

picture = picture[pos+4:]
fhand = open("stuff.jpg", "wb")
fhand.write(picture)
fhand.close()

# Code: http://www.py4e.com/code3/urljpeg.py
```

عند تشغيل البرنامج فإنّه يُولّد الخرج التالي:

```
$ python urljpeg.py
5120 5120
5120 10240
4240 14480
5120 19600
...
5120 214000
3200 217200
5120 222320
5120 227440
3167 230607
```

```
Header length 393

HTTP/1.1 200 OK

Date: Wed, 11 Apr 2018 18:54:09 GMT

Server: Apache/2.4.7 (Ubuntu)

Last-Modified: Mon, 15 May 2017 12:27:40 GMT

ETag: "38342-54f8f2e5b6277"

Accept-Ranges: bytes

Content-Length: 230210

Vary: Accept-Encoding

Cache-Control: max-age=0, no-cache, no-store, must-revalidate

Pragma: no-cache

Expires: Wed, 11 Jan 1984 05:00:00 GMT

Connection: close

Content-Type: image/jpeg
```

تستطيع أن تُلاحظ أنّه من أجل الرابط المستخدم يشير Content_Type في الترويسة إلى أن محتوى المستند هو عبارة عن صورة (image/jpeg). بمجرّد أن يتم تنفيذ البرنامج، بإمكانك استعراض بيانات الصورة عبر فتح الملف stuff.jpg في برنامج عرض الصور.

ستلاحظ عند تنفيذ البرنامج أننا لا نستقبل 5120 مِحرَف في كل مرة نستدعي التابع recv()، فحين نستدعي recv() نحصل على المحارف التي تم نقلها إلينا عبر الشبكة عن طريق خادم الويب. ففي هذا المثال، حصلنا على دفعات من البيانات تتراوح بين 3200 حتى 5120 محرف.

قد تكون نتائجك مختلفة اعتمادًا على سرعة الشبكة. ضع بالحسبان أيضًا أنّه عند الاستدعاء الأخير لـ recv() نحصل على 3167 بايت وهي أخر جزء من البيانات. وفي الاستدعاء التالي لـ recv() نحصل على سلسلة بطول صفري (Zero-length string) والتي تشير إلى أن المخدم استدعى التابع close() عند طرف مآخذ الشبكة، وأنّه لا يوجد المزيد من البيانات القادمة.

بإمكاننا إبطاء عملية الاستدعاء المتتالية لـ recv() عن طريق إلغاء تعليق استدعاء التابع time.sleep(). حيث ننتظر بهذه الطريقة ربع ثانية بعد كل استدعاء (نضيف تأخيرًا بمقدار ربع ثانية)

بحيث يمكن للخادم مجاراتنا وإرسال المزيد من البيانات لنا قبل أن نستدعي recv() مرة أخرى.

مع هذا التأخير يتم تنفيذ البرنامج على النحو التالي:

```
$ python urljpeg.py
5120 5120
5120 10240
5120 15360
...
5120 225280
5120 230400
207 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 21:42:08 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT

Connection: close
Content-Type: image/jpeg
```

الآن، وبغض النظر عن الاستدعاء الأول والأخير لـ recv() نحصل على 5120 مِحرَف في كل مرة نطلب

بيانات جديدة.

هُناك ذاكرة مؤقتة (Buffer) بين الخادم الذي يُنشئ طلبات التابع send() والتطبيق الخاص بنا الذي يُنشئ طلبات recv().

في مرحلة ما، وعند تشغيل البرنامج مع التأخير، قد يتسبب الخادم في ملء الذاكرة المؤقتة في مآخذ الشبكة ويجبره على التوقف حتى يبدأ برنامجنا بإفراغها.

إن عملية إيقاف كل من تطبيق الإرسال أو الاستقبال يُدعى التحكم في التدفّق (flow control).

4.12 استعادة صفحات الويب باستخدام مكتبة urllib

أرسلنا واستقبلنا سابقًا البيانات عن طريق HTTP مُستخدمين مكتبة socket، لكن هنالك طريقة أسهل لتنفيذ هذه المهمة مستخدمين مكتبة urllib.

تستطيع باستخدامك لمكتبة urllib التعامل مع صفحة الويب كما لو أنها ملف، فتُشير ببساطة إلى صفحة الويب التي تُريد استعادتها لتقوم المكتبة بالتعامل مع تفاصيل بروتوكول HTTP وتفاصيل الترويسة.

إنّ الشيفرة المُكافئة لقراءة ملف romeo.txt من الويب باستخدام مكتبة urllib هي كالتالي:

```
import urllib.request

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())


# Code: http://www.py4e.com/code3/urllib1.py
```

بمُجرّد أن تُفتح صفحة الويب باستخدام تعليمة urllib.urlopen، يُصبح بإمكاننا التعامل معها مثل الملف وقراءتها باستخدام حلقة for.

عند تشغيل البرنامج، نرى محتويات الملف فقط في الخرج، بالرغم أن الترويسة أرسلت بالفعل لكن

شيفرة urllib تتجاهلها وتعيد فقط محتوى الملف.

```
But soft what light through yonder window breaks

It is the east and Juliet is the sun

Arise fair sun and kill the envious moon

Who is already sick and pale with grief
```

على سبيل المثال، يمكننا كتابة برنامج لاسترداد البيانات الخاصة بـ romeo.txt وحساب تكرار كل كلمة في الملف على النحو التالي:

```
import urllib.request, urllib.parse, urllib.error


fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()

for line in fhand:

    words = line.decode().split()

    for word in words:

        counts[word] = counts.get(word, 0) + 1

print(counts)


# Code: http://www.py4e.com/code3/urlwords.py
```

من جديد، بمجرّد أن نفتح صفحة الويب بإمكاننا قراءتها كملف محلي (متوفر على جهازك).

5.12 قراءة الملفات المُشفّرة ثنائيًا باستخدام urllib

قد ترغب أحيانًا في استعادة ملف غير نصي مرمز ثنائيًا (binary) مثل صورة أو فيديو.

عمومًا، تُعتبر البيانات الموجودة في هذه الملفات غير مُفيدة عند عرضها على الخرج. لكنك ببساطة تستطيع إنشاء نسخة من عنوان URL إلى ملف محلي على قرصك الصلب باستخدام urllib.

الإجراء المُتّبع هنا هو فتح عنوان URL واستعمال التعليمة read لتخزين جميع محتويات الملف في مُتغيّر من نوع سلسلة نصية وليكن اسمه img ثم اكتب تلك المعلومات في ملف محلّي كما هو موضح

في الشيفرة التالية:

```
import urllib.request, urllib.parse, urllib.error


img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
fhand.close()


# Code: http://www.py4e.com/code3/curl1.py
```

يقرأ هذا البرنامج جميع البيانات دفعة واحدة ويُخزّنها في المتغير img في الذاكرة الرئيسية لحاسوبك. ثمّ يفتح الملف cover.jpg ويكتب البيانات على قرصك الصلب.

يفتح الوسيط wb في التابع open() ملفًا ثنائيًا للكتابة فقط مع العلم أن هذا البرنامج يعمل في حال كان حجم الملف أقل من حجم ذاكرة حاسوبك.

أما في حال كان الفيديو أو الملف الصوتي ذو حجم كبير، فإن البرنامج قد يتوقف، أو على أقل تقدير سوف يعمل ببطء شديد بينما تنفد ذاكرة حاسوبك.

سعيًا لتجنُّب نفاد الذاكرة، فإننا نسترجع البيانات ككتل ثمّ نكتب كل كتلة بيانات على القرص قبل استعادة الكتلة التالية. بهذه الطريقة يستطيع البرنامج قراءة أي ملف مهما كان حجمه دون استهلاك الذاكرة الموجودة في حاسوبك.

```
import urllib.request, urllib.parse, urllib.error


img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
fhand = open('cover3.jpg', 'wb')
size = 0
while True:
        info = img.read(100000)
```

```
        if len(info) < 1: break
        size = size + len(info)
        fhand.write(info)

print(size, 'characters copied.')
fhand.close()

# Code: http://www.py4e.com/code3/curl2.py
```

في هذا المثال، نقرأ فقط 100,000 مِحرَف معًا ثم نكتب هذه المحارف في ملف cover.jpg قبل استعادة الـ 100,000 محرف التالية من الويب. حيث يظهر خرج البرنامج على النحو التالي:

```
python curl2.py
230210 characters copied.
```

6.12 تحليل واستخراج البيانات من صفحات HTML

تُعتبر عملية استخراج البيانات من صفحات الويب أحد الاستخدامات الشائعة لمكتبة urllib في لغة بايثون.

يتجلّى مفهوم استكشاف أو تعقب الويب (Web scraping) عندما نكتب برنامجًا يتصرّف كمُتصفّح إنترنت ويقوم باسترجاع الصفحات، ثم يفحص البيانات الموجودة في تلك الصفحات بحثًا عن أنماط ما.

كمثال على ذلك، تعاين محركات البحث مثل غوغل Google مصدر صفحة ويب ما لتستخرج روابط الصفحات الأخرى ثم تستعيد هذه الصفحات ومن ثم تعود لتستخرج الروابط وهكذا..

بفضل هذه التقنية، يستطيع غوغل الوصول إلى كل الصفحات في الويب تقريبًا.

يستخدم غوغل أيضًا معدل تكرار رابط صفحة ما في باقي الصفحات على أنها معيار لمدى "أهمية" الصفحة ولتحديد ترتيبها في قائمة نتائج البحث.

7.12 تحليل صفحات HTML باستخدام التعابير النمطية

يعتبر استخدام التعابير النمطية أحد الأساليب البسيطة لتحليل صفحات HTML وخاصة لأجل عمليات البحث المتكررة واستخراج سلاسل نصية فرعية التي تتطابق مع نمط معيّن. فيما يلي صفحة ويب بسيطة:

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

يُمكننا إنشاء تعبير نمطي لاستخراج الرابط من النص أعلاه على النحو التالي:

```
href="http[s]?://.+?"
```

يبحث هذا التعبير النمطي عن السلاسل النصية التي تبدأ بـ href="http:// أو href="https:// متبوعة بمحرف أو أكثر .+?. ثمّ بعلامة اقتباس أخرى. كما تُشير علامة الاستفهام في التعبير [s]? إلى البحث عن السلسلة http متبوعة بـ صفر أو واحد s (أي وجود s واحدة أو عدمها).

علامة الاستفهام في .+? تُشير إلى أن التطابق سيكون من النمط غير المتعدي بدلًا من النمط المتعدي (Pushy) حيث يسعى النمط غير المتعدي لإيجاد أصغر سلسلة نصّية مطابقة ممكنة، بينما يسعى النمط المتعدي إلى العثور على أكبر سلسلة نصّية مطابقة ممكنة.

سنُضيف الأقواس إلى التعبير النمطي للإشارة إلى الجزء الذي نريد استخراجه من السلسلة المطابقة. ليصبح البرنامج كالتالي:

```
# Search for link values within URL input

import urllib.request, urllib.parse, urllib.error
import re
import ssl
```

```
# Ignore SSL certificate errors


ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
links = re.findall(b' href="(http[s]?://.*?)" ', html)
for link in links:
    print(link.decode())


# Code: http://www.py4e.com/code3/urlregex.py
```

تسمح مكتبة ssl لهذا البرنامج بالوصول إلى مواقع الويب التي تستخدم بروتوكول HTTPS. يُرجع التابع read الشيفرة المصدريّة لـ HTML كسلسلة من البايت بدلًا من إرجاعها ككائن HTTPResponse.

يعيد التابع findall قائمة من السلاسل النصية المطابقة لتعبيرنا النمطي، حيث يعيد فقط الرابط بين علامتي الاقتباس المزدوجة.

عند تشغيل البرنامج وإدخال رابط ما نحصل على الخرج التالي:

```
Enter - https://docs.python.org
https://docs.python.org/3/index.html
https://www.python.org/
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
```

https://www.python.org/dev/peps/

https://wiki.python.org/moin/BeginnersGuide

https://wiki.python.org/moin/PythonBooks

https://www.python.org/doc/av/

https://www.python.org/

https://www.python.org/psf/donations/

http://sphinx.pocoo.org/

تعمل التعابير النمطية بشكل رائع للغاية عندما تكون الشيفرة المصدرية لصفحة ال HTML مكتوبة بشكل منسق وقابل للتنبُّؤ. لكن نظرًا لوجود الكثير من صفحات HTML غير المنسقة جيدًا فإن هذا الحل (أيْ استخدام التعابير النمطية) قد يتسبب بفقدان بعض الروابط المتاحة أو الحصول على بيانات غير مفيدة. يُمكن حل هذه المشكلة باستخدام مكتبة خاصة للتعامل مع صفحات HTML.

8.12 تحليل صفحات HTML باستخدام مكتبة BeautifulSoup

على الرغم من أن صفحات HTML تبدو مُشابهة لـ **XML** (سيتم شرح ماهيّة XML في الفصل القادم) وبعض الصفحات مبنيّة على أساس XML، إلا أن معظم صفحات HTML تكون غير منسقة جيدًا، الأمر الذي يؤدي إلى رفض برمجية "XML parser" صفحة HTML بأكملها بسبب تنسيقها غير الصحيح.

يوجد العديد من المكتبات في لغة بايثون لمساعدتك في تحليل صفحات HTML واستخراج البيانات منها. كُل مكتبة من هذه المكتبات تمتلك نقاط قوة ونقاط ضعف وتستطيع اختيار المكتبة بناءً على احتياجاتك.

كمثال على ذلك، سنُحلّل ببساطة بعض مدخلات HTML وسنستخرج الروابط باستخدام مكتبة BeautifulSoup.

تتساهل مكتبة BeautifulSoup مع صفحات HTML التي تحوي عيوبًا كثيرة وتسمح لك باستخراج البيانات التي تحتاجها بسهولة. بإمكانك تحميل وتنصيب شيفرة البرنامج من الرابط:

https://pypi.python.org/pypi/beautifulsoup4

تتيح أداة فهرسة حزم بايثون (Python Package Index) اختصارًا "pip" معلومات تنصيب مكتبة

BeautifulSoup في الرابط التالي:

https://packaging.python.org/tutorials/installing-packages/

سنستخدم مكتبة urllib لقراءة الصفحة ثم نستخدم BeautifulSoup لاستخراج الخاصية href من الوسم <a>.

```
# To run this, download the BeautifulSoup zip file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file
import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl
# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: http://www.py4e.com/code3/urllinks.py
```

يطلب البرنامج عنوان صفحة ويب، ثم يفتح صفحة الويب ويقرأ البيانات، بعدها يُمرر هذه البيانات إلى مُحلل مكتبة BeautifulSoup، ثمّ يسترجِع كل وسوم <a> ليطبع قيمة الخاصية href لكل وسم.

عندما نشغّل البرنامج فإنه يُنتج الخرج التالي:

```
Enter - https://docs.python.org
genindex.html
py-modindex.html
https://www.python.org/
#
whatsnew/3.6.html
whatsnew/index.html
tutorial/index.html
library/index.html
reference/index.html
using/index.html
howto/index.html
installing/index.html
distributing/index.html
extending/index.html
c-api/index.html
faq/index.html
py-modindex.html
genindex.html
glossary.html
search.html
contents.html
bugs.html
about.html
license.html
```

```
copyright.html
download.html
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
genindex.html
py-modindex.html
https://www.python.org/
#
copyright.html
https://www.python.org/psf/donations/
bugs.html
http://sphinx.pocoo.org/
```

هذه القائمة أطول بكثير مما أردنا لأن بعض وسوم <a> في HTML هي مسارات نسبية (relative path) (على سبيل المثال: tutorial/index.html) أو مراجع داخلية (مثلًا: '#') التي لا تتضمن "http://" أو "https://" والذي كان أحد المتطلبات في تعبيرنا النمطي.

يمكنك أيضًا استخدام BeautifulSoup لاستخراج أجزاء أخرى من أي وسم:

```
# To run this, download the BeautifulSoup zip file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file
```

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import ssl
# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
url = input('Enter - ')
html = urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, "html.parser")
# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    # Look at the parts of a tag
    print('TAG:', tag)
    print('URL:', tag.get('href', None)
    print('Contents:', tag.contents[0])
    print('Attrs:', tag.attrs)
# Code: http://www.py4e.com/code3/urllink2.py
```

فيكون الخرج:

```
python urllink2.py

Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
```

```
Content: ['\nSecond Page']
Attrs: [('href', 'http://www.dr-chuck.com/page2.htm')]
```

إنّ المحلّل "html.parser" هو محلل HTML المتضمن في مكتبة Python 3 المعيارية.

تستطيع الحصول على معلومات عن محللات HTML أخرى عبر الرابط:

http://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser

تُظهر هذه الأمثلة مدى قوة مكتبة BeautifulSoup عندما يتعلق الأمر بتحليل صفحات HTML.

9.12 ميزات خاصة لمُستخدمي أنظمة لينُكس أو يونيكس

إذا كان لديك حاسوب يعمل بنظام تشغيل لينُكس (Linux) أو يونيكس (Unix) أو ماكنتوش (Macintosh) فعلى الأرجح أنك تمتلك أوامر جاهزة في نظام التشغيل. حيث تسترجِع هذه الأوامر النصوص والملفات المرمزة ثنائيًا باستعمال بروتوكول نقل النص التشعبي HTTP أو بروتوكول نقل الملفات File Transfer Protocol (FTP). وأحد هذه الأوامر هو curl:

```
$ curl -O http://www.py4e.com/cover.jpg
```

إنّ الأمر curl هو اختصار للتعبير copy URL.

إن المثالين الذين ذُكرا سابقًا لاسترجاع الملفات المرمزة ثنائيًا باستخدام urllib أُطلِق عليهما curl1.py وَ curl2.py على الموقع www.py4e.com/code3 حيث يُنفّذان وظائف مشابهة للأمر curl.

هُناك أيضًا البرنامج curl3.py الذي يُنجز هذه المهمة بفعالية أكبر، في حال كنت تريد استخدام هذا النمط في البرنامج الذي تكتبه.

الأمر الثاني الذي يؤدي الوظيفة بشكل مشابه هو wget:

```
$ wget http://www.py4e.com/cover.jpg
```

كلا الأمرين يسهلان عملية استرجاع صفحات الويب والملفات غير المخزنة محليًا.

10.12 فهرس المصطلحات

- مكتبة **BeautifulSoup**: مكتبة في لغة بايثون نستخدمها لتحليل صفحات HTML واستخراج البيانات منها والتي عادةً ما يتجاهلها المتصفّح. بإمكانك تحميل شيفرة مكتبة

BeautifulSoup من الموقع www.crummy.com

- **المنفَذ (Port)**: رقم يُشير بشكل عام إلى التطبيق المتصل به عندما تقوم بإجراء اتصال عبر مآخذ الشبكة مع الخادم. كمثال على ذلك: يُستخدم عادةً المنفذ 80 في عملية إرسال واستقبال البيانات عبر الويب، بينما للبريد الإلكتروني يستخدم المنفذ 25.
- **استكشاف أو تعقب الويب (Scrape)**: عندما يتظاهر البرنامج بأنه متصفح ويب ويسترجع صفحة ويب، ثمّ يُعاين محتواها. تتّبِع البرامج عادةً الروابط الموجودة في صفحة واحدة للعثور على الصفحة التالية لذلك بإمكانهم المرور على شبكة من الصفحات أو على شبكة اجتماعية.
- **مآخذ الشبكة (Socket)**: اتصال شبكي بين تطبيقين، حيث يُتاح للتطبيقات تبادل البيانات في كلا الاتجاهين (إرسال واستقبال).
- **المُتعقب (Spider)**: عندما يقوم محرك البحث باستعادة صفحة ثم كل الصفحات المرتبطة بهذه الصفحة وهكذا حتى يصل تقريبًا إلى كل الصفحات في الإنترنت، حيث يتم استخدام هذا في بناء فهرس البحث.

11.12 تمارين

- **التمرين الأوّل:** عدّل البرنامج socket1.py بحيث يطلب عنوان URL من المستخدم ليتمكّن البرنامج من الوصول إلى أَيْ صفحة ويب. يُمكنك استخدام التابع split('/') من أجل تجزئة عنوان URL إلى مكوناته بحيث تتمكّن من استخراج اسم المُضيف من أجل استدعاء التابع connect. أضِف ميزة تجنب الأخطاء باستخدام تعليمتي try وَexcept للتعامل مع الحالة التي يُدخل بها المستخدم روابط URL خاطئة أو غير موجودة.
- **التمرين الثاني:** عدّل البرنامج السابق بحيث يحسب عدد المحارِف التي استقبلها، ثمّ يتوقف عن إظهار أَيْ نص بعد عرض 3000 مِحْرف. يجب على البرنامج استعادة المُستند بالكامل وحساب العدد الإجمالي للمحارف وعرْضِهِ في نهاية المستند.
- **التمرين الثالث:** استخدمْ مكتبة urllib لتكرار التمرين السابق من أجل: (1) استعادة المُستند من عنوان URL، (2) عرض قُرابة الـ 3000 محرف، (3) حساب العدد الإجمالي

للمحارف في المستند. لا تقلق بشأن الترويسة في هذا التمرين، ما عليك سوى إظهار أوّل 3000 محرف من محتويات المستند.

- **التمرين الرابع:** قُم بتعديل برنامج urllinks.py لاستخراج وحساب وسوم الفقرات <p> من مُستند HTML الذي تم استعادته، ثمّ اعرِضْ عدد الفقرات كخرج لبرنامجك.

لا تعرِض نَص الفقرة بل قم بإحصائهم فقط. اختَبِرْ البرنامج على عدة صفحات ويب صغيرة بالإضافة إلى بعض صفحات الويب الكبيرة.

- **التمرين الخامس:** (مُتقدّم)

غيّر برنامجك socket1.py بحيث يعرض البيانات فقط بعد استقبال الترويسة وسطر فارغ.

تذكّر أن التابع recv يستقبل البيانات كمحارِف (محرف السطر الجديد أحدها) وليس كأسطر.

الفصل الثالث عشر

استخدام خدمات الويب

13 استخدام خدمات الويب

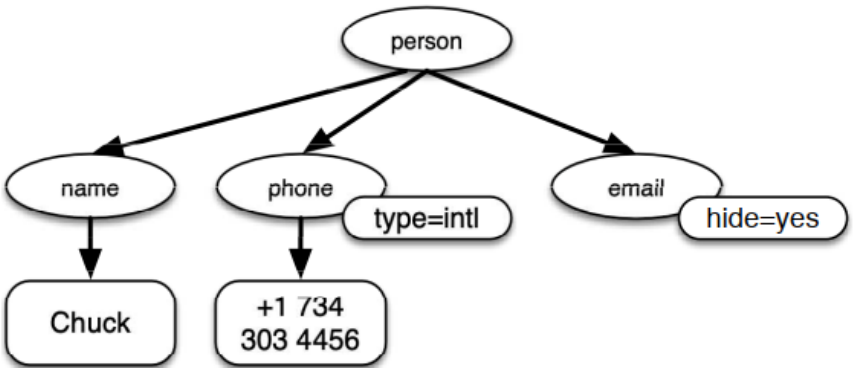
لم يستغرق الأمر طويلًا لتطوير منهجية لإنشاء ملفات صممت لتستخدمها برامج أخرى (مثال: فتح صفحة غير مبنية باستخدام HTML بواسطة المتصفح) بعد أن أصبحت عملية استدعاء الملفات وتحليلها سهلة التنفيذ عبر برامج تستخدم برتوكول HTTP، حيث يوجد صيغتين نستخدمهما عند تبادل البيانات عبر الويب، أولها لغة التوصيف الموسعة XML، والتي استخدمت لزمن طويل وتعتبر الأنسب لتبادل البيانات على شكل ملفات، بينما تستخدم البرامج ترميز جافا سكربت الغرضي JSON (للمزيد تصفح الموقع www.json.org) لتبادل القواميس والقوائم فيما بينها أو أي معلومات داخلية، وسنشرح كلتا الصيغتين.

1.13 لغة التوصيف الموسعة XML

تشبه XML الـ HTML ولكنها أكثر تنظيمًا، ونرى هذا في المثال الآتي:

```
<person>
  <name> Chuck </name>
  <phone type="intl">
     +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>
```

كما نلاحظ، يمثل كل زوج من وسوم الافتتاح مثل <person> والإغلاق مثل </person> عنصر أو عقدة (Node) بنفس اسم الوسم مثل person، ويمكن أن يكون لكل عنصر نص معين أو خصائص "سمات" مثل hide وعناصر متداخلة أخرى، وإذا كان العنصر فارغ بلا محتوى فيمكن أن يُغلق ذاتيًا مثل <email />، أي من المفيد أن تنظر إلى ملف XML على أنه ذو بنية شجرية، حيث يوجد عنصر رئيس في مثالنا السابق person ووسوم أخرى مثل phone وكأنها فروع من العناصر الرئيسية (الأبوية).



الشكل 14 :تمثيل شجري للغةXML

2.13 تحليل نصوص XML

فيما يلي تطبيق عن تحليل نص XML واستخراج بعض عناصر البيانات منه:

```
import xml.etree.ElementTree as ET

data = ' ' '
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person> ' ' '
tree = ET.fromstring(data)
print('Name':, tree.find('name').text)
print('Attr':, tree.find('email').get('hide'))
# Code: http://www.py4e.com/code3/xml1.py
```

تسمح إشارات التنصيص الأحادية والمزدوجة الثلاثية (' ' ' و""") بإنشاء سلاسل نصية تمتد على عدة أسطر، كما أن استدعاء fromstring يحول السلاسل النصية في XML إلى شجرة من العناصر، فعندما تكون XML في نمط شجرة يكون لدينا عدة توابع يمكننا استدعاؤها لاستخراج أجزاء من البيانات من السلاسل النصية، أما التابع find فيبحث في شجرة XML عن الوسم المطابق لما حدد ضمنه ويستدعيه.

```
Name: Chuck

Attr: yes
```

يسمح لنا محلل نصوص XML مثل ElementTree باستخراج البيانات من XML بدون القلق حول القواعد الكتابية لها، والتي تتوضح لنا في المثال السابق البسيط الذي عرضناه.

3.13 استخدام الحلقات للمرور على العقد

عادةً ما تحوي XML عدة عقد حيث نحتاج لكتابة حلقة لمعالجة كل تلك العقد، كما في البرنامج الآتي حيث نمر على كل عقد المستخدم:

```
import xml.etree.ElementTree as ET

input = ' ' '
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>' ' '
stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))
for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get('x'))

# Code: http://www.py4e.com/code3/xml2.py
```

يُرجع التابع findall قائمة مكونة من تفرعات تمثل بنية الوسم user في شجرة XML، نكتب بعدها حلقة for تمر على كل عقدة من عقد الوسم user وتطبع العناصر النصية name وid إضافةً إلى الخاصية x من عقدة الوسم user:

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

ومن المهم تضمين جميع العناصر الرئيسية (الأبوية) في تعليمة findall (مثل users/user) باستثناء عند التعامل مع عناصر المستوى الرئيسي الأول وإلا لن تجد بايثون أيّ من العقد المطلوبة:

```
import xml.etree.ElementTree as ET

input = ''' 
<stuff>
  <users>
    <user x="2">
       <id>001</id>
       <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
     </user>
  </users>
</stuff> '''

stuff = ET.fromstring(input)

lst = stuff.findall('users/user')
```

```
print('User count:', len(lst))

lst2 = stuff.findall('user')
print('User count:', len(lst2))
```

تخزن القائمة lst كل عناصر الوسم user المضمنة في الوسم users، بينما تبحث lst2 عن عناصر الوسم user المضمنة في وسم المستوى الأول stuff لكن لا تجد أيًا منها.

```
User count: 2
User count: 0
```

JSON 4.13

استُوحت هذه الصيغة من الصيغة الغرضية والمصفوفية في لغة جافا سكربت، إلا أن قواعد كتابة بايثون فيما يتعلق بالقواميس والقوائم أثرت على قواعد JSON باعتبار أنها وُجدت قبل جافا سكربت، لذلك تعتبر هذه الصيغة خليط من قوائم وقواميس بايثون، وفيما يأتي مثال عن ترميز JSON مكافئ لبرنامج XML المذكور سابقًا:

```
{
  "name" : "Chuck",
   "phone" : {
     "type" : "intl",
     "number" : "+1 734 303 4456"
           },
   "email" : {
     "hide" : "yes"
     }
}
```

قد تلاحظ بعض الفروق ففي XML نستطيع إضافة السمة int1 إلى الوسم phone بينما لدينا أزواج مفتاح-قيمة في JSON كما يختفي الوسم person هنا فقد استُبدل بالأقواس الخارجية.

عمومًا، فإن بنية JSON أبسط من بنية XML حيث تملك إمكانيات أقل، ولكن تملك الأفضلية من حيث الارتباط مباشرة مع تركيبة القواميس والقوائم، كما أنها صيغة بسيطة لجعل برنامجين يعملان معًا يتبادلان البيانات باعتبار أن جميع لغات البرمجة تقريبًا تملك مكافئًا لقواميس وقوائم

بايثون، إضافةً إلى أنها سرعان ما أصبحت خيارًا لصيغة معظم عمليات تبادل البيانات بين التطبيقات بسبب بساطتها مقارنةً مع XML.

5.13 تحليل نصوص JSON

ننشئ ملفات JSON بترتيب القواميس والقوائم داخل بعضهم البعض كما نحتاج، وفي هذا المثال نمثل قائمة مستخدمين بحيث يكون كل مستخدم مجموعة من أزواج مفتاح-قيمة (أي قاموس) أي لدينا قائمة من القواميس، كما سنستخدم مكتبة جاهزة لتحليل نص JSON وقراءة البيانات، وبإمكانك إجراء المقارنة مع المثال السابق في XML، حيث JSON تحوي تفاصيل أقل أي يجب أن نعلم مسبقًا أننا سنحصل على قائمة تمثل المستخدمين حيث كل مستخدم هو مجموعة من أزواج مفتاح-قيمة فـ JSON أكثر إيجازًا (وهي نقطة إيجابية) ولكنها صعبة التوصيف الذاتي (وهذه سلبية):

```
import json
data =' ' '
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
 {  "id" : "009",
    "x" : "7",
    "name" : "Brent"
  }
] ' ' '
info = json.loads(data)
print('User count:', len(info))
for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])

# Code: http://www.py4e.com/code3/json2.py
```

إذا قارنت شيفرة استخراج البيانات بين XML وJSON فستلاحظ أننا نحصل من التابع json.loads() على قائمة نمر على عناصرها بحلقة for ويمثّل كل عنصر في تلك القائمة قاموسًا، ونستطيع استخدام عامل الفهرس لاستخراج البيانات المختلفة لكل مستخدم بمجرد تحليل نص JSON، كما لسنا مضطرين لاستخدام مكتبة JSON لإجراء عملية التحليل باعتبار أن بنية البيانات هي بنية معروفة لبايثون، ويكون خرج هذا البرنامج مطابق لخرج البرنامج السابق في XML وهو:

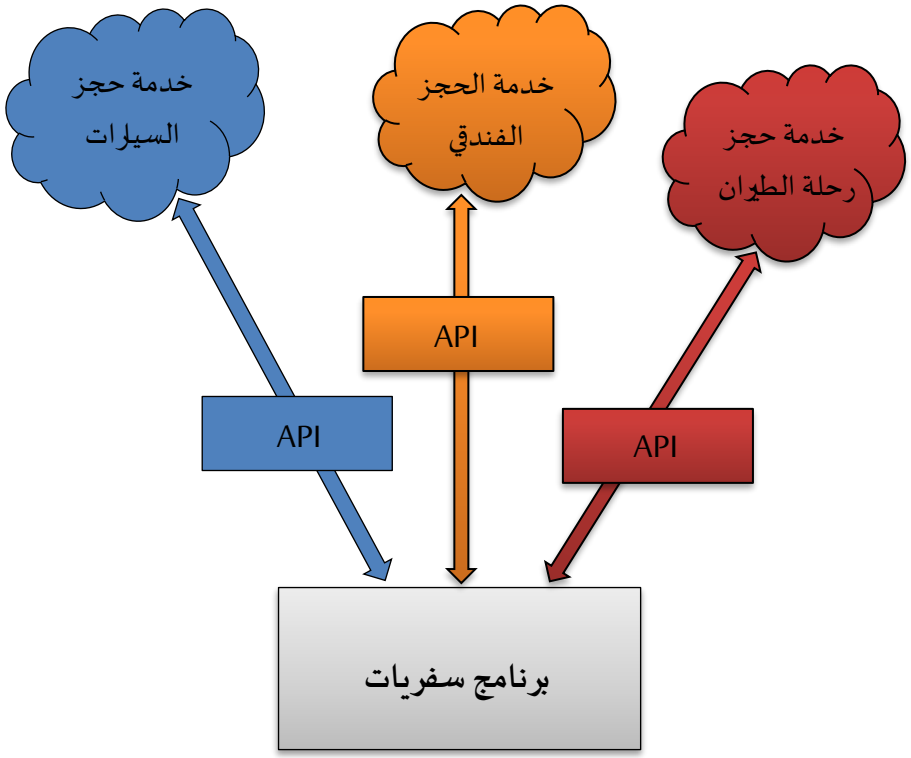
```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

عمومًا، يوجد توجه مهني نحو JSON بدلًا من XML فيما يتعلق بخدمات الويب، لأنها أبسط وتعبر عن بنى البيانات الأساسية الموجودة في لغات البرمجة بشكل أكبر إضافةً إلى كون عملية التحليل واستخراج البيانات أبسط ومباشرة بشكل أكبر، إلا أن XML قابلة للتوصيف الذاتي بشكل أفضل مما يجعل استخدامها أفضلية في بعض التطبيقات، فعلى سبيل المثال، معظم معالجات النصوص تخزن الملفات داخليًا باستخدام XML بدلًا من JSON.

6.13 واجهات برمجة التطبيقات

نتمتع اليوم بالقدرة على تبادل البيانات بين التطبيقات باستخدام بروتوكول HTTP مع طريقة لتمثيل البيانات المعقدة المتبادلة باستخدام لغة التوصيف الموسعة XML أو ترميز جافا سكربت الغرضي JSON، وتكمن الخطوة التالية في تحديد وتوثيق "الاتفاقيات" بين تلك التطبيقات عبر هذه التقنيات، الاسم العام لهذه الاتفاقيات بين التطبيقات هو واجهات برمجة التطبيقات API فعند استخدامها يجعل أحد البرامج مجموعة من الخدمات متاحة لتستخدمها تطبيقات أخرى كما ينشر تلك الواجهات (أي القواعد) التي يجب اتباعها للوصول إلى الخدمات التي يقدمها أحد البرامج، حيث أثناء تصميم البرامج التي تتطلب وظيفتها الوصول إلى خدمات برامج أخرى. ندعو هذا النهج باسم البنية خدمية التوجه SOA أي استخدام برنامجنا النهائي لخدمات تطبيقات أخرى. بينما يعرف نهج البنية لا خدمية التوجه non-SOA بأنه تطبيق قائم بذاته يحتوي على جميع التعليمات البرمجية اللازمة ليقدم خدماته.

عادةً نلاحظ العديد من أمثلة SOA أثناء استخدام الويب، فبإمكاننا الدخول إلى موقع ما وحجز تذكرة طيران أو إجراء حجز فندقي أو حجز سيارة من نفس الموقع، إلا أن بيانات الفنادق غير مخزنة على حواسيب خطوط الطيران، بل تتواصل هذه الحواسيب مع الخدمات على حواسيب الفندق لاستدعاء بياناته وعرضها للمستخدم، أي أن موقع خطوط الطيران يستخدم خدمة ويب أخرى موجودة في أنظمة الفندق عندما يوافق المستخدم على إجراء حجز في ذلك الفندق من خلال هذا الموقع، وبالتالي تدخل عدة حواسيب في هذه العملية حتى عند دفعك للأجور المستحقة.



الشكل 15 : البنية خدمية التوجه

ومن فوائد البنى خدمية التوجه (SOA):

1. نحتفظ بنسخة واحدة من البيانات فقط (وهذا مهم خاصةً فيما يشابه حجوزات الفنادق حيث لا يتم الالتزام لمدة طويلة).
2. بإمكان مالكي البيانات وضع قواعد لاستخدامها.

ومع هذه الفوائد يجب أن يصمم نظام SOA بحذر ليتميز بالأداء الجيد ويلبي حاجات المستخدم، ويظهر هنا مصطلح خدمات الويب حيث يجعل تطبيق ما مجموعة من الخدمات في واجهته متاحة عبر الويب.

7.13 الأمان واستخدام واجهات برمجة التطبيقات

من الشائع احتياجك إلى مفتاح معين لاستخدام واجهة برمجة التطبيقات العائدة لشركة ما، والمقصد رغبتهم في معرفة من يستخدم خدماتهم، وكمية استخدامه، لربما لديهم خدمات مدفوعة أو مجانية أو سياسة تحديد عدد الطلبات المتاحة للفرد خلال مدة زمنية معينة، وأحيانا بمجرد حصولك على المفتاح تضمنه كجزء من بيانات رسالة POST أو كمعامل في الرابط (URL) عند استدعاء الواجهة البرمجية، وفي بعض الأحيان تطلب الشركة ضمان أكبر فيما يتعلق بمصدر الطلبات لهذا يطلبون منك إرسال رسائل موقعة ومشفرة باستخدام المفاتيح المشاركة، أما التقنية الشائعة لتوقيع الطلبات عبر الإنترنت فتدعى OAuth وبإمكانك التعرف على هذا البروتوكول عبر الرابط www.oauth.net، ولحسن الحظ توجد بعض مكتبات OAuth المجانية والمناسبة لتجنب كتابة تطبيق OAuth من الصفر من خلال قراءة المواصفات فقط. كما تختلف تلك المكتبات بدرجة تعقيدها وسعتها، وأيضًا تستطيع الحصول على معلومات أكثر عن مكتبات OAuth من خلال زيارة الموقع المذكور أعلاه.

8.13 فهرس المصطلحات

- **واجهة برمجة التطبيقات (API):** اتفاقية بين التطبيقات تحدد أنماط التفاعل بين مكونات تطبيقين.
- **مكتبة ElementTree:** مكتبة برمجية مضمنة في لغة بايثون تستخدم لتحليل نصوص XML.
- **JSON:** صيغة تسمح بترميز بيانات مهيكلة اعتمادًا على القواعد الكتابية للكائنات في جافا سكربت.
- **البنى خدمية التوجه (SOA):** مصطلح يستخدم عند بناء تطبيق من مكونات متصلة ببعضها عبر شبكة ما.
- **لغة التوصيف الموسعة (XML):** صيغة تسمح بترميز بيانات مهيكلة.

9.13 التطبيق الأول: خدمة الترميز الجغرافي من غوغل:

لغوغل خدمة ذات فائدة كبرى، إذ تسمح لنا باستخدام قاعدة بياناتهم الضخمة الخاصة بالمعلومات الجغرافية، حيث نستطيع إجراء بحث جغرافي نصي مثل "Ann Arbor, MIT" ضمن

واجهة برمجة التطبيقات للترميز الجغرافي من غوغل لتعيد لنا أفضل تخمين للمكان التي يمكن أن يجد فيه العنوان المطلوب على الخريطة الجغرافية ويخبرنا بالمعالم المحيطة به. هذه الخدمة مجانية ولكنها محدودة، أي أن استخدامك للواجهة في التطبيقات التجارية محدود، ولكن إن كانت لديك بيانات حيث يدخل المستخدم موقع ما مجانًا فبإمكانك استخدام هذه الواجهة للتعامل مع البيانات بشكل جيد.

يجب أن تكون معتدلًا عند استخدام الواجهات المجانية كواجهة غوغل للترميز الجغرافي حيث يمكن لغوغل إلغاءها أو تقليل الخدمات المتاحة إن أساء عدد كبير من الناس استخدامها، كما يمكنك قراءة توصيف تلك الخدمة على الإنترنت وهو بسيط للغاية وتستطيع اختباره على أحد المتصفحات بكتابة الرابط الآتي:

http://maps.googleapis.com/maps/api/geocode/json?address=Ann+Arbor%2C+MI

لكن تأكد من إزالة الفراغات منه قبل لصقه إلى المتصفح، وسنضع مثالًا عن تطبيق يطلب من المستخدم إدخال مكان ما لنبحث عنه ثم يستدعي واجهة الترميز الجغرافي لغوغل ليستخرج المعلومات من ترميز JSON المعاد:

```
import urllib.request, urllib.parse, urllib.error
import json
import ssl

api_key = False
# If you have a Google Places API key, enter it here
# api_key = AIzaSy___IDByT70
#https://developers.google.com/maps/documentation/geocoding/intro

if api_key is False:
    api_key = 42
    serviceurl = 'http://py4e-data.dr-chuck.net/json? '
else :
    serviceurl = ' https://maps.googleapis.com/maps/api/geocode/json? '

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
```

```
ctx.verify_mode = ssl.CERT_NONE

while True:
    address = input('Enter location: ' )
    if len(address) < 1: break

    parms = dict()
    parms['address'] = address
    if api_key is not False: parms['key'] = api_key
    url = serviceurl + urllib.parse.urlencode(parms)

    print('Retrieving', url)
    uh = urllib.request.urlopen(url, context=ctx)
    data = uh.read().decode()
    print('Retrieved', len(data), 'characters')

    try:
        js = json.loads(data)
    except:
        js = None
    if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)
        continue
    print(json.dumps(js, indent=4))

    lat = js['results'][0]['geometry']['location']['lat']
    lng = js['results'][0]['geometry']['location']['lng']
    print('lat', lat, 'lng', lng)
    location = js['results'][0]['formatted_address']
    print(location)
# Code: http://www.py4e.com/code3/geojson.py
```

يستقبل البرنامج نص البحث وينشئ رابط (URL) مستخدمًا إياه كمعامل مشفَّر ثم يستخدم urllib لاستدعاء النص من واجهة غوغل للترميز الجغرافي، كما تعتمد البيانات التي نحصل عليها على المعاملات التي نرسلها والبيانات الجغرافية المخزنة في خوادم غوغل على عكس صفحات الويب الثابتة، وبمجرد حصولنا على بيانات JSON نحللها باستخدام مكتبة JSON لنجري بعدها عدة اختبارات للتأكد من جودة البيانات المستقبَلة، ثم نستخرج المعلومات التي نبحث عنها.

ويكون خرج البرنامج كالآتي (حُذفت بعض بيانات JSON المستقبلة):

```
$ python3 geojson.py
Enter location: Ann Arbor, MI
Retrieving http://py4e-data.dr-chuck.net/json?address=Ann+Arbor%2C+MI&key=42
Retrieved 1736 characters

{
  "results": [
    {
      "address_components": [
        {
          "long_name": "Ann Arbor",
          "short_name": "Ann Arbor",
          "types": [
            "locality",
            "political"
          ]
        },
        {
          "long_name": "Washtenaw County",
          "short_name": "Washtenaw County",
          "types": [
            "administrative_area_level_2",
            "political"
          ]
        },
        {
```

```
                "long_name": "Michigan",
                "short_name": "MI",
                "types": [
                  "administrative_area_level_1",
                  "political"
                ]
            },
            {
                "long_name": "United States",
                "short_name": "US",
                "types": [
                  "country",
                  "political"
                ]
            }
],

"formatted_address": "Ann Arbor, MI, USA",
"geometry": {
        "bounds": {
            "northeast": {
                "lat": 42.3239728,
                "lng": -83.6758069
              },
            "southwest": {
                "lat": 42.222668,
                "lng": -83.799572
              }
},
"location": {
    "lat": 42.2808256,
    "lng": -83.7430378
},
```

```
"location_type": "APPROXIMATE",
"viewport": {
      "northeast": {
      "lat": 42.3239728,
      "lng": -83.6758069
},
     "southwest": {
        "lat": 42.222668,
       "lng": -83.799572
       }
   }
},
"place_id": "ChIJMx9D1A2wPIgR4rXIhkb5Cds",
"types": [
      "locality",
      "political"
       ]
   }
],
"status": "OK"
}
```


lat 42.2808256 lng -83.7430378

Ann Arbor, MI, USA

Enter location:

ويمكنك تنزيل البرنامج www.py4e.com/code3/geoxml.py لاكتشاف اختلاف البرنامج بحالة استخدام XML لواجهة ترميز غوغل الجغرافي.

التمرين الأول: عدل أحد البرنامجين geojson.py أو geoxml.py لطباعة رمز الدولة الثنائي من البيانات المستقبلة وأضف تعليمات للتحقق من الأخطاء كي لا يفشل برنامجك إن لم يكن رمز

الدولة موجود، وبمجرد عمله ابحث عن المحيط الأطلسي "Atlantic Ocean" وتأكد أنه يستطيع التعامل مع مواقع غير موجودة ضمن حدود أي دولة.

10.13 التطبيق الثاني: تويتر

انتقلت تويتر من الواجهات مفتوحة المصدر والعامة إلى الواجهات التي تتطلب استخدام تواقيع OAuth لكل طلب وذلك مع ازدياد أهمية واجهاتها، ولأجل المثال التالي نزل الملفات twurl.py وhidden.py وoauth.py وtwitter1.py من www.py4e.com/code وضعهم في مجلد واحد معًا على حاسوبك، ولاستخدام هذه البرامج تحتاج حساب على تويتر وتفويض برنامجك كتطبيق وإعداد المفتاح وكلمة سر والرمز (token) وكلمة سر الرمز ومن ثم عدل الملف hidden.py وضع هذه السلاسل النصية ضمن متحولات مناسبة في البرنامج:

```
# Keep this file separate
# https://apps.twitter.com/
# Create new App and get the four strings

def oauth():
return {"consumer_key": "h7Lu...Ng",
            "consumer_secret" : "dNKenAC3New...mmn7Q",
            "token_key" : "10185562-eibxCp9n2...P4GEQQOSGI",
            "token_secret" : "H0ycCFemmC4wyf1...qoIpBo"}

# Code: http://www.py4e.com/code3/hidden.py
```

نصل لخدمات تويتر عبر الرابط الآتي:

https://api.twitter.com/1.1/statuses/user_timeline.json

ولكن بمجرد إضافة جميع معلومات الأمان فسيبدو الرابط كالآتي:

```
https://api.twitter.com/1.1/statuses/user_timeline.json?count=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
```

```
&oauth_signature_method=HMAC-SHA1
```

ويمكنك قراءة توصيف OAuth في حال أردت المزيد من المعلومات حول معاني المعاملات المختلفة المضافة لإيفاء متطلبات OAuth للأمان، سنخبئ كل التعقيدات في الملفات oauth.py وtwurl.py من أجل البرامج التي تعمل مع تويتر. بدايةً نضيف كلمة السر في hidden.py ثم نرسل الرابط المطلوب إلى التابع twurl.augment لتضيف المكتبة جميع المعاملات اللازمة إلى الرابط لأجلنا.

يحدد هذا البرنامج منشورات مستخدم تويتر محدد ويعيدها إلينا في صيغة JSON كسلسلة نصية لنظهر أول 250 محرف منها على الشاشة:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/statuses/user_timeline.json'
# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,{'screen_name': acct, 'count': '2'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
   data = connection.read().decode()
   print(data[:250])
```

```
    headers = dict(connection.getheaders())
    # print headers
    print('Remaining', headers['x-rate-limit-remaining'])

# Code: http://www.py4e.com/code3/twitter1.py
```

وعند تشغيل البرنامج نحصل على الخرج الآتي:

```
Enter Twitter Account: drchuck
Retrieving https://api.twitter.com/1.1/ …
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Remaining 178

Enter Twitter Account: fixpert
Retrieving https://api.twitter.com/1.1/ …
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
my wedding ring fits again! :)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Remaining 177

Enter Twitter Account:
```

تعيد تويتر أيضًا بيانات وصفية حول الطلب في ترويسة استجابة HTTP إضافةً إلى بيانات المنشورات، ويخبرنا أحد البيانات الوصفية وهو x-rate-limit-remaining بعدد الطلبات التي نستطيع إرسالها قبل إيقاف الخدمة مؤقتًا، كما يمكنك ملاحظة أن عدد مرات الاستدعاء تقل بواحد بعد كل طلب.

في المثال التالي، نستدعي قائمة أصدقاء مستخدم تويتر ونحلل نصوص JSON المستقبلة لاستخراج بعض المعلومات حول أولئك الأصدقاء، وأيضًا نتخلص من ملف JSON بعد تحليله ثم نطبع مؤشر معبر عنه من أربع محارف يسمح لنا بمسح البيانات إذا أردنا استخراج حقول معلومات إضافية:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()
    js = json.loads(data)
    print(json.dumps(js, indent=2))
    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])
    for u in js['users']:
```

```
        print(u['screen_name'])
        if 'status' not in u:
            print(' * No status found')
            continue
        s = u['status']['text']
        print(' ', s[:50])

# Code: http://www.py4e.com/code3/twitter2.py
```

وبما أن JSON تتحول إلى قوائم وقواميس بايثون متداخلة فنستطيع استخدام مزيج من عامل الفهرس وحلقات for لنمر عبر بنى البيانات المستقبلة باستخدام عدد قليل من تعليمات بايثون، وسيبدو خرج البرنامج كما يأتي (اختصرت بعض عناصر البيانات لتسع الصفحة):

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14

{
  "next_cursor": 1444171224491980205,
  "users": [
    {
      "id": 662433,
      "followers_count": 28725,
      "status": {
        "text": "@jazzychad I just bought one .__.",
        "created_at": "Fri Sep 20 08:36:34 +0000 2013",
        "retweeted": false,
    },
      "location": "San Francisco, California",
      "screen_name": "leahculver",
      "name": "Leah Culver",
    },
    {
      "id": 40426722,
```

```
        "followers_count": 2635,
        "status": {
            "text": "RT @WSJ: Big employers like Google ...",
             "created_at": "Sat Sep 28 19:36:37 +0000 2013",
   },
      "location": "Victoria Canada",
      "screen_name": "_valeriei",
      "name": "Valerie Irvine",
  }
],

"next_cursor_str": "1444171224491980205"
}
```


```
leahculver
   @jazzychad I just bought one .__.
_valeriei
    RT @WSJ: Big employers like Google, AT&T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
  @DeviceLabDC love it! Now where so I get that "etc

Enter Twitter Account:
```

نرى أن حلقة for تقرأ بيانات أحدث خمس أصدقاء لحساب تويتر @drchuck في القسم الأخير من الخرج وتطبع آخر تغريدة (منشور) لكل صديق، إلا أنه توجد بيانات أكثر متاحة ضمن ملف JSON المستقبل، كما ستلاحظ -بالنظر إلى خرج البرنامج أن خدمة "جِد الأصدقاء" لحساب تويتر له معدل

محدد ومختلف القيمة عن عدد طلبات الحصول على المنشورات المسموح لنا إجراؤها خلال مدة زمنية معينة.

إن هذه المفاتيح المؤمّنة الخاصة بالواجهات تسمح لتويتر بتكوين معرفة عميقة لمن يستخدم واجهاتهم وبياناتهم وعلى أي مستوى، بينما يسمح لنا مفهوم تحديد معدل الاستخدام بإجراء استدعاءات بسيطة وشخصية للبيانات ولكن لا يسمح بتصميم منتج يسحب البيانات من واجهاتهم مليون مرة باليوم الواحد.

الفصل الرابع عشر

البرمجة كائنية التوجه

14 البرمجة الكائنية التوجه

1.14 إدارة البرامج الكبيرة

مررنا في الفصول الأولى من الكتاب على أربع أنماط برمجية لبناء البرامج المختلفة وهي:

- الشيفرة التسلسلية
- الشيفرة الشرطية (بنية if)
- الشيفرة التكرارية (الحلقات)
- التخزين وإعادة الاستخدام (التوابع)

ثم تعرفنا في الفصول اللاحقة إلى المتغيرات إلى جانب بعض بنى البيانات مثل القوائم والصفوف والقواميس.

لقد كتبتَ حتى الآن العديد من البرامج، منها الممتاز ومنها غير المتقن وبالرغم من أن هذه البرامج بسيطة ولكن لابد أن تكون قد أدركت الآن أن البرمجة فن.

من المهم للغاية كتابة شيفرة سهلة الفهم خاصة حين يتكون البرنامج من ملايين الأسطر، فحينها لن يستطيع عقلك استيعابه. لذا اقتضت الحاجة أن يتم تقسيم البرنامج إلى قطع صغيرة حتى يتسنى لنا التركيز على حل مشكلة وإصلاح خطأ أو إضافة ميزة جديدة.

وهنا يأتي دور البرمجة كائنية التوجه، فهي طريقة لترتيب الشيفرات تمكنك من التركيز على 50 سطر من الشيفرة وفهمها وتجاهل الأسطر 999950 الأخرى.

2.14 مقدمة

كباقي النواحي البرمجية من الضروري تعلم مفاهيم البرمجة كائنية التوجه قبل استخدامها، فعليك التركيز في هذا الفصل على تعلم بعض مصطلحاتها ومفاهيمها وتنفيذ بعض الأمثلة البسيطة لوضع حجر الأساس لما هو آتٍ.

هدفنا الأساسي هو أن تفهم مبدئيًا كيف تُبنى الكائنات وطريقة عملها والأهم من ذلك كيف نستفيد من الكائنات الجاهزة التي تزودنا بها لغة بايثون ومكتباتها.

3.14 استخدام الكائنات

دعني أخبرك بشيء، لقد كنا نستخدم الكائنات بكثرة في هذا الكتاب حيث توفر لغة بايثون العديد من الكائنات الجاهزة، إليك بعض الشيفرات البسيطة، لاحظ الأسطر الأولى منها فستجدها مألوفة لديك:

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: http://www.py4e.com/code3/party1.py
```

لندع الآن ما تنفذه هذه الأسطر ولنلقي نظرة على ماذا يحدث حقًا من وجهة نظر البرمجة كائنية التوجه، لا تقلق إذا شعرت أن الفقرات التالية بلا أي معنى عند قراءتها للمرة الأولى فأنت لم تتعرف على جميع المفاهيم بعد.

يبني السطر الأول كائنا من نوع قائمة List في حين يستدعي السطر الثاني والثالث تابع append() لهذا الكائن، ثم استدعينا في السطر الرابع التابع sort() وفي السطر الخامس نحصل على أول عنصر في القائمة.

ننتقل إلى السطر السادس حيث نستدعي تابع __getitem__() في القائمة stuff بمعامل صفري (لنستعيد العنصر ذو الفهرس صفر في القائمة).

```
print (stuff.__getitem__(0))
```

السطر السابع هو مجرد طريقة مطولة لاسترجاع العنصر الصفري في القائمة.

```
print (list.__getitem__(stuff,0))
```

في هذه الشيفرة استدعينا التابع __getitem__ من الصنف List (class) ومررنا القائمة والعنصر الذي نريد استرجاعه من القائمة كمعامل.

إن الأسطر الثلاث الأخيرة من البرنامج متكافئة، لكن من الأنسب استخدم الأقواس المربعة [ ] للبحث عن عنصر محدد في قائمة.

يمكننا التعرف على قدرات الكائن عبر النظر إلى خرج التابع dir() :

```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

سيتضح لك في بقية هذا الفصل كل المصطلحات المبهمة في الأعلى لذلك أحرص على العودة عندما تنهي الفصل وأعد قراءة الفقرات السابقة كي تتحقق من فهمك.

4.14 البدء مع البرامج

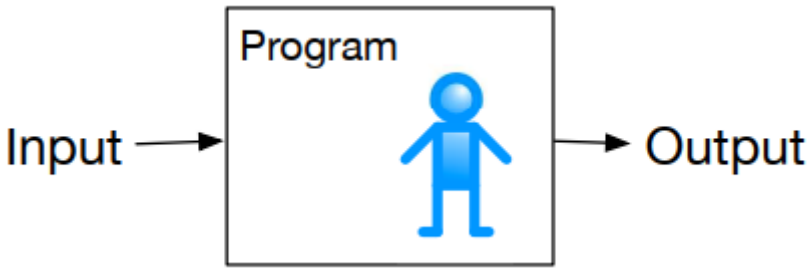
تعلمنا سابقًا أن البرنامج في أبسط أشكاله يأخذ بعض المدخلات، ليعالجها، ثم يُنتج بعض من المخرجات. فلننظر إلى برنامج تحويل أرقام الطوابق في المصاعد القصير جدًا لكنه كامل ويُظهر كلا من تلك الخطوات الثلاث.

```
usf = input('Enter the US Floor Number: ')
wf = int(usf) - 1
print('Non-US Floor Number is',wf)


Code: http://www.py4e.com/code3/elev.py #
```

إذا تأملنا هذا البرنامج بتمعن فسنرى البرنامج وما يمكن أن نسميه بالعالم الخارجي حيث يتفاعل البرنامج مع العالم الخارجي فيستقبل منه ويرسل إليه، وفي قلب البرنامج نفسه لدينا شيفرة وبيانات لإنجاز المهمة التي صمم البرنامج لحلها.

ولقد أخذت البرمجة كائنية التوجه هذا المفهوم وطورته أكثر فهي تقسم برنامجنا إلى نطاقات متعددة، ولكل نطاق شيفراته وبياناته (كأنه برنامج مستقل) وتفاعلاته المحددة جيدًا مع العالم الخارجي والنطاقات الأخرى ضمن البرنامج الرئيسي. إذا نظرنا مجددًا إلى تطبيق استخراج الرابط التشعبي حين استخدمنا مكتبة BeautifulSoup

سنرى بوضوح مثالًا لبرنامج مُنشأ عبر ربط الكائنات المختلفة سويًا لتحقيق المهمة.



الشكل 17 : البرنامج

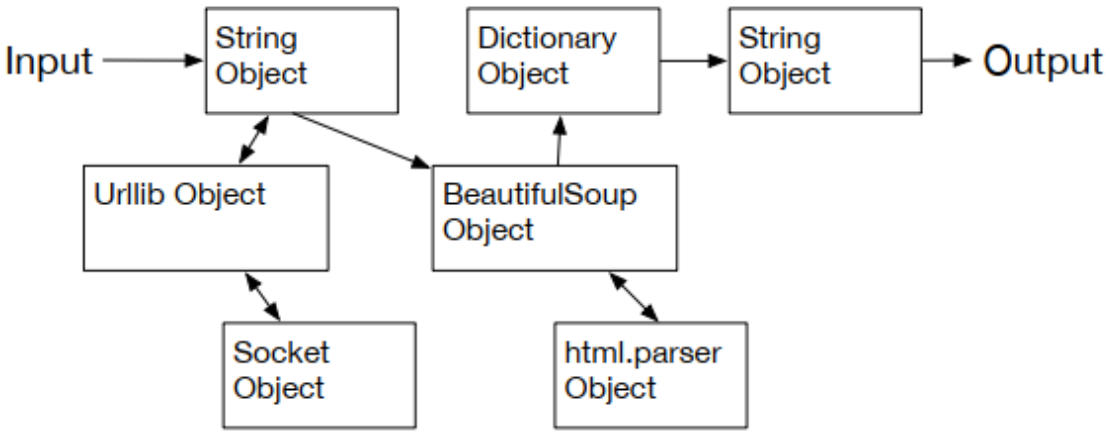
```
# To run this, download the BeautifulSoup zip file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl
# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')
# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: http://www.py4e.com/code3/urllinks.py
```

لقد خزّنا الرابط في متغير من نوع سلسلة نصية ثم مررناه إلى urllib لجلب البيانات من الويب، حيث تستخدم مكتبة urllib مكتبة socket لإنشاء اتصال الشبكة الفعلي لاستعادة البيانات. بعد ذلك نأخذ النص الناتج عن urllib ونسلمه إلى BeautifulSoup لتحليله، حيث تستعين BeautifulSoup بالكائن html.parser لتعيد كائنًا.

نستدعي التابع tags() على ذلك الكائن ليعيد قاموس من الوسوم، ثم نمر على عناصر القاموس باستخدام حلقة ونستدعي التابع get() لكل وسم لطباعة الخاصية href.

بإمكاننا رسم مخطط لهذا البرنامج وكيف تعمل هذه الكائنات معا.

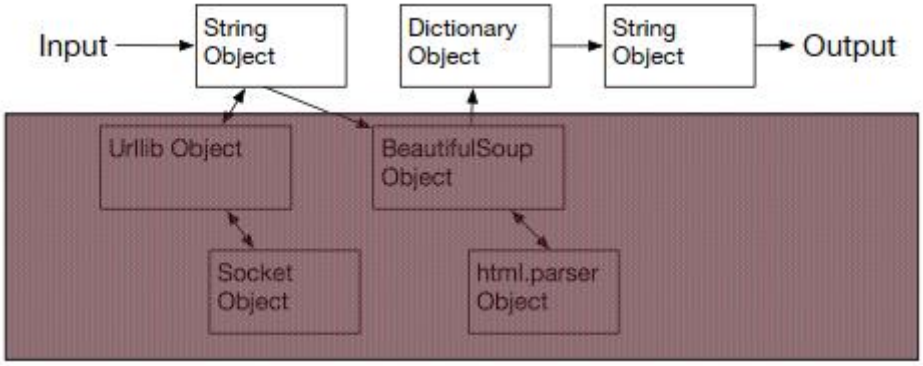
الغرض من هذا المثال ليس معرفة كيف نستخرج رابطًا من صفحة ويب بل رؤية كيف نبني شبكة كائنات متفاعلة وكيفية انسياب المعلومات بين تلك الكائنات لإنشاء البرنامج، لعلك لاحظت عندما استخدمت هذا البرنامج في فصل سابق من هذا الكتاب أمكنك فهم ماذا يقوم به بدون أن تدرك كيف كان ينسق البرنامج حركة البيانات بين الكائنات، فما هي إلا أسطر من الشيفرة التي تؤدي المطلوب.



الشكل 18: البرنامج كشبكة من الكائنات

5.14 تقسيم المشكلة

يتميز أسلوب البرمجة كائنية التوجه أنه بإمكانه إخفاء التعقيد، فمثلًا عندما نحتاج لمعرفة كيف نستخدم مكتبتي urllib وBeautifulSoup فنحن لا نحتاج لمعرفة كيف تعمل هذه المكتبات داخليًا، مما يسمح لنا بالتركيز على المشكلة الذي نريد حلها وتجاهل ما سواها.



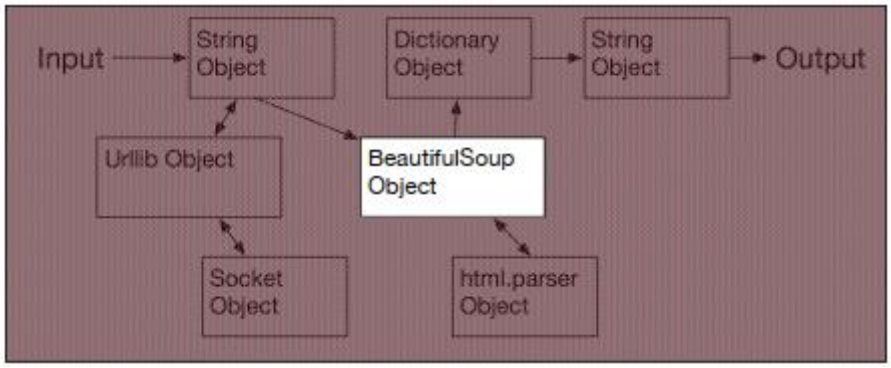
الشكل 19: تجاهل التفصيل عند استخدام الكائن

تعد القدرة على التركيز على جزء من البرنامج وتجاهل ما سواه مفيدًا أيضًا لمطوري الكائنات التي نستخدمها، فمثلًا لا يحتاج المبرمجون الذين يطورون BeautifulSoup لمعرفة أو للاهتمام كيف نحصل على صفحة HTML أو ماهية الصفحات التي نريد قراءتها أو ما الذي نخطط لفعله مع البيانات التي نستخرجها من صفحة ويب.

6.14 إنشاء كائن في لغة بايثون

الكائن ببساطة هو جزء صغير من البرنامج يحتوي على بعض الشيفرات بالإضافة إلى بعض من بنى البيانات.

لنتذكر مفهوم التابع الذي يسمح لنا بتخزين بعض الشيفرة ويمنحها اسم معين، ثم يمكننا استحضار تلك الشيفرة لاحقًا بكتابة اسم التابع فقط.



الشكل 20: تجاهل التفاصيل عند إنشاء الكائن

قد يحوي الكائن العديد من التوابع تسمى (methods) إلى جانب البيانات التي تستخدمها هذه التوابع، ونسمي عناصر البيانات التي هي جزء من الكائن بالخواص (properties).

نستخدم الكلمة المفتاحية class لنحدد البيانات والشيفرة التي سنستخدمها لإنشاء الكائنات حيث

يأتي بعد الكلمة المفتاحية اسم الصنف يليها الشيفرة المتضمنة للخواص (البيانات) والتوابع (الشيفرة).

```
class PartyAnimal:
    x = 0
   def party(self) :
      self.x = self.x + 1
      print("So far",self.x)


an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)


# Code: http://www.py4e.com/code3/party2.py
```

هذا الكائن يحوي خاصية واحدة x وتابع واحد هو party، يحوي التابع على عامل خاص نطلق عليه self. كما أن الكلمة المفتاحية def لا تؤدي إلى تنفيذ شيفرة التابع كذلك فالكلمة المفتاحية class لا تصنع كائنًا، فالصنف يمثل القالب الذي يتضمن البيانات والشيفرة التي سيتكون منها كل كائن من نوع PartyAnimal. للتوضيح، اعتبر الصنف كقالب صنع الكعك والكائنات المنشأة منه هي الكعك، فنحن لا نضع الزينة على القالب بل على الكعك، وبإمكانك وضع زينة مختلفة على كل قطعة.

الشكل 21 :صنف وكائنان

لنتابع الآن شرح البرنامج ونأتي للسطر التالي

```
an = PartyAnimal()
```

نأمر هنا لغة بايثون ببناء كائن أو نموذج من الصنف PartyAnimal ويبدو الأمر كأنه استدعاء تابع للصنف نفسه.

تبني لغة بايثون الكائن ببياناته وتوابعه ثم تعيده إلى المتغير an، ما سبق يشبه الأمر التالي الذي كنا نستخدمه في الفصول السابقة:

```
counts = dict()
```

حيث نأمر بايثون ببناء كائن باستخدام قالب القاموس dict (الجاهز في بايثون) ونسنده إلى متغير counts.

تذكر عندما نستخدم صنف PartyAnimal لبناء كائن فإن المتغير an يستخدم ليشير إلى ذلك الكائن، ويحوي كل كائن أو نموذج من Partyanimal المتغيرx وتابع يدعى party.

نستدعي التابع party في هذا السطر

```
an.party()
```

عند استدعاء التابع party فإن العامل (الذي نسميه اصطلاحا self) يشير إلى الكائن بذاته من بين كائنات الصنف PartyAnimal والذي تم استدعاء التابع عبره.

في التابع party نرى السطر:

```
self.x = self.x + 1
```

يستخدم هذا السطر عامل النقطة التي تعني (استخدم x التي تنتمي للكائن) وفي كل مرة تستدعى فيها التابع party فإن قيمة x الداخلية تزداد بمقدار 1 ثم تعرض النتيجة على الخرج.

يمثل السطر التالي طريقة أخرى لاستدعاء التابع party عبر الكائن an:

```
PartyAnimal.party(an)
```

الاختلاف هنا أننا نستدعي الشيفرة من داخل الصنف نفسه ثم نمرر مؤشر الكائن an كمعامل (أي المعامل المسمى self ضمن التابع)، وبإمكاننا التفكير أن an.party هي اختصار للسطر أعلاه.

عندما ينفذ البرنامج فإنه سيعطي الخرج التالي:

```
So far 1
```

```
So far 2
So far 3
So far 4
```

فالكائن بُني ثم استدعى التابع party أربع مرات، بحيث يزيد بمقدار 1 ويطبع القيمة x الموجودة في الكائن an.

7.14 الصنف كنوع بيانات

في لغة بايثون كل المتغيرات لها نوع محدد. استخدم التابع الجاهز dir لمعرفة قدرات أي متغير وأيضا بإمكاننا استخدام type وdir مع الأصناف التي انشأناها

```
class PartyAnimal:
    x = 0
    def party(self) :
        self.x = self.x + 1
        print("So far",self.x)


an = PartyAnimal()
print ("Type", type(an))
print ("Dir ", dir(an))
print ("Type", type(an.x))
print ("Type", type(an.party))


# Code: http://www.py4e.com/code3/party3.py
```

عندما ينفذ البرنامج سينتج الخرج التالي:

```
Type <class '__main__.PartyAnimal'>
Dir ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

يمكنك القول إننا باستخدام الكلمة المفتاحية class صنعنا نوع بيانات جديد.

يمكن باستخدام التابع dir رؤية كل من خواص العدد الصحيح x والتابع party في الكائن.

8.14 دورة حياة الكائن

في الأمثلة السابقة عرّفنا صنف (أي قالب) واستخدمناه لإنشاء نموذج منه (كائن) ثم استخدمنا هذا الكائن.

عندما ينتهي البرنامج تُهمل كل المتغيرات. عادة لا نفكر كثيرًا في عملية إنشاء وهدم المتغيرات، لكن عندما يصبح كائننا أكثر تعقيدًا نحتاج لاتخاذ إجراءات معينة أثناء إنشاء الكائن، وحذف الأشياء عند إهماله.

إذا أردنا تبيان عمليات البناء والهدم نضيف توابع خاصة لكائنا.

```
class PartyAnimal:
    x = 0

    def __init__(self):
        print('I am constructed')

    def party(self) :
        self.x = self.x + 1
        print('So far',self.x)

    def __del__(self):
        print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)

# Code: http://www.py4e.com/code3/party4.py
```

عندما ينفذ البرنامج فإنه يعطي الخرج التالي:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

عندما تُنشِأ لغة بايثون الكائن فإنها تستدعي تابع __init__ لتعطينا فرصة لضبط القيم الابتدائية للكائن.

عندما تصادف لغة بايثون السطر: an = 42

فإن بايثون تتخلص من الكائن عن طريق إعادة استخدام المتغير an لتخزين القيمة 42، وعند تدمير الكائن تستدعى شيفرة الهادم __del__.

لا يمكننا حماية المتغيرات في الكائن من عملية الهدم، كل ما هنالك أننا نقوم بالعمليات الضرورية قبل أن يختفي الكائن نهائيًا.

تذكر دائمًا عند تطوير الكائنات فمن الشائع جدًا إضافة التابع الباني للكائن لضبط القيم الابتدائية له، ونادرًا ما نحتاج تابع الهادم للكائن.

9.14 تعدد الكائنات

حتى الآن عرّفنا ما هو الصنف وبنينا كائن وحيد واستخدمناه ثم تخلصنا منه.

تظهر القوة الحقيقية للبرمجة كائنية التوجه عندما نبني كائنات متعددة من صنفٍ واحد، حيث نعطي قيم ابتدائية مختلفة لكل من الكائنات.

هنا سنقوم بتمرير البيانات إلى الباني لإعطاء كل كائن قيمة ابتدائية مختلفة:

```
class PartyAnimal:
    x = 0
    name = ''
    def __init__(self, nam):
        self.name = nam
        print(self.name,'constructed')
```

```
  def party(self) :
    self.x = self.x + 1
    print(self.name,'party count',self.x)

s = PartyAnimal('Sally')
j = PartyAnimal('Jim')

s.party()
j.party()
s.party()

# Code: http://www.py4e.com/code3/party5.py
```

تضم عوامل الباني العامل self الذي يشير إلى حالة العنصر وعوامل إضافية تمرر عبر الباني أثناء إنشاء الكائن:

```
s = PartyAnimal('Sally')
```

أثناء عملية الإنشاء فإن السطر الثاني ينسخ العامل (nam) إلى خاصية name للكائن عبر self

```
self.name = nam
```

إن خرج البرنامج يظهر أن كل من تلك الكائنات s و j تحوي نسخ مستقلة من قيم x و name

```
Sally constructed
Jim constructed
Sally party count 1
Jim party count 1
Sally party count 2
```

10.14 الوراثة

تُعد القدرة على إنشاء صنف جديد عبر توسيع صنف موجود ميزة أخرى للبرمجة كائنية التوجه، فعند توسيع الصنف ندعو الصنف الأصلي بالصنف الأب (parent class) والصنف الجديد بالصنف الابن (child class)

فلننقل صنف PartyAnimal إلى ملف منفصل، لاستيراده في ملف جديد وتوسيعه كما يلي:

```
from party import PartyAnimal


class CricketFan(PartyAnimal):
    points = 0
    def six(self):
        self.points = self.points + 6
        self.party()
        print(self.name,"points",self.points)


s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))


# Code: http://www.py4e.com/code3/party6.py
```

نشير عند تعريف الصنف CricketFan إلى الصنف PartyAnimal، هذا يعني أن كل من المتغيرات x والتوابع party في الصنف PartyAnimal ستورث إلى الصنف CricketFan، فعلى سبيل المثال استدعينا التابع party من صنف PartyAnimal في تابع six للصنف CricketFan.

عند تنفيذ البرنامج ننشأ s وj ككائنات مستقلة من الصنفين PartyAnimal وCricketFan

لاحظ أن الكائن j لديه قدرات إضافية تفوق الكائن s

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 6
['__class__', '__delattr__', ... '__weakref__',
```

```
'name', 'party', 'points', 'six', 'x']
```

في خرج التابع dir للكائن j (ذو الصنف CricketFan) نرى أن له خواص وتوابع من الصنف الأب وأيضًا الخواص والتوابع التي أضفناها عندما وسعنا الصنف لإنشاء CricketFan.

11.14 ملخص

هذه مقدمة مختصرة للبرمجة كائنية التوجه التي تركز بصورة أساسية على قواعد تعريف واستخدام الكائنات.

لنراجع الشيفرة التي رأيناها في بداية الفصل، الآن لن تجد أدنى صعوبة في فهمها.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))


# Code: http://www.py4e.com/code3/party1.py
```

يُنشئ في السطر الأول كائن من الصنف list، عندها تستدعي بايثون التابع الباني وهو (__init__) لضبط البيانات الداخلية التي سوف تُستخدم لتخزين قائمة من البيانات.

لاحظ أننا لم نمرر أي عوامل إلى هذا الباني، وعندما ينتهي الباني من عمله نستخدم المتغير stuff للإشارة إلى الكائن الناتج من الصنف list.

في السطرين الثاني والثالث نستدعي التابع append لإضافة عنصر جديد إلى نهاية القائمة عبر تحديث خواص الكائن stuff. ثم في السطر الرابع نستدعي تابع sort بلا أي عوامل لترتيب بيانات الكائن.

نستخدم عندما نريد طباعة أول عنصر في القائمة الأقواس المربعة [] والتي تعد بديلًا مختصرًا لاستدعاء __getitem__ باستخدام stuff، وهذا يكافئ استدعاء تابع __getitem__ على صنف List وتمرير كائن stuff كمعامل أول والفهرس التي نرغب بعرض محتواه كمعامل ثاني.

نستدعي في نهاية البرنامج التابع الهادم (والذي يسمى __del__) ليتمكن الكائن من التعامل مع أي بيانات سائبة قبل هدم الكائن stuff. هذه هي أساسيات البرمجة كائنية التوجه وهناك تفاصيل إضافية تُتّبع عند تطوير تطبيقات ضخمة أو مكتبات لكنها خارج نطاق هذا الفصل.

12.14 فهرس المصطلحات

- **الخاصية (attribute):** متغير جزء من الصنف.
- **الصنف (class):** قالب يستخدم لبناء كائن، ويحدد الخواص والتوابع التي تشكل الكائن.
- **الصنف الابن (child class):** صنف جديد ينشأ من توسيع الصنف الأب، ويرث جميع الخواص والتوابع من الصنف الأب.
- **التابع الباني (constructor):** تابع اختياري يسمى (__init__) يستدعى لحظة بناء الكائن ويستخدم عادة لضبط القيم الابتدائية.
- **التابع الهادم (destructor):** تابع اختياري يسمى (__del__) يستدعى في اللحظة قبل إزالة الكائن، نادر الاستخدام.
- **الوراثة (inheritance):** عندما يتم إنشاء صنف جديد (الابن child) عند توسيع صنف موجود (أب parent) يحصل الصنف الابن على جميع الخواص والتوابع من الصنف الأب بالإضافة لخواص وتوابع محددة له.
- **التابع (method):** تابع موجود في صنف والكائنات المبنية من الصنف، وبعض المنهجيات في البرمجة كائنية التوجه تستخدم عبارة رسالة message بدلًا من method للتعبير عن هذا المفهوم.
- **الكائن (object):** نموذج يتم إنشاؤه من الصنف، يحوي كل من الخواص والتوابع المعرّفة في الصنف، وبعض وثائق البرمجة كائنية التوجه تستخدم مصطلح instance بدلًا من object.
- **الصنف الأب (parent class):** الصنف الذي تم توسيعه لصنع صنف ابن جديد، يشارك الصنف الأب كل الخواص والتوابع مع الصنف الابن.

الفصل الخامس عشر

استخدام قواعد البيانات ولغة SQL

15 استخدام قواعد البيانات ولغة SQL

1.15 ما هي قاعدة البيانات؟

إن قاعدة البيانات هي ملف منظّم لتخزين البيانات، وتكون معظم هذه القواعد منظمة بطريقة مشابهة للقواميس حيث تربط بين مفاتيح وقيم، لكن الاختلاف الرئيسي بينهما أن قاعدة البيانات موجودة على القرص الصلب (أو أي أداة تخزين دائمة أخرى)، أي تحتفظ بالبيانات حتى بعد انتهاء البرنامج، وبإمكانها تخزين بيانات أكثر من القواميس بسبب وجودها على وحدة تخزين دائمة، بينما يكون حجم القاموس مرتبط بحجم ذاكرة الحاسوب. صُممت برمجيات قواعد البيانات بشكل يجعل إدخال البيانات والوصول إليها سريعًا جدًا مهما كِبَر حجمها تمامًا كالقواميس، وتحافظ هذه البرمجيات على الأداء السريع بإنشاء فهارس تزامنًا مع إدخال بيانات جديدة سامحةً للحاسوب بالوصول إلى مُدخَل معين بسرعة.

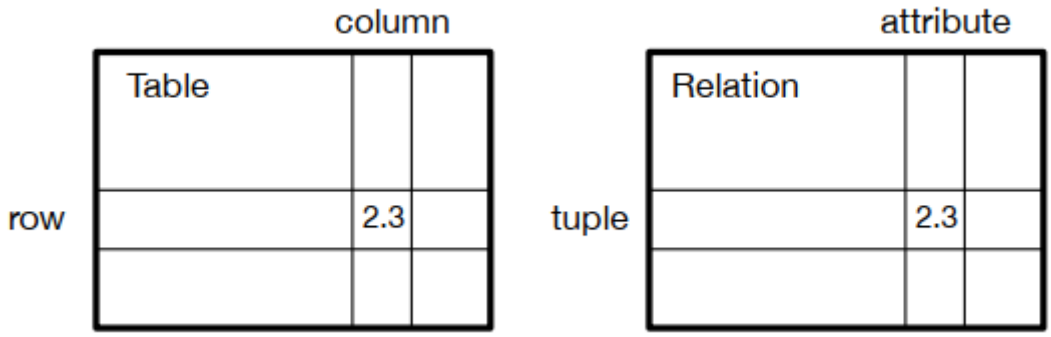
توجد أنظمة مختلفة لقواعد البيانات تستخدم لأهداف متعددة ومنها Oracle و MySQL و Microsoft SQL Server و PostgreSQL و SQLite .

سنركز في هذا الكتاب على SQLite لكونه قاعدة بيانات شائعة الاستخدام ومضمَّن في بايثون، كما صممت بطريقة تسمح بتضمينها في التطبيقات المختلفة لتأمين دعم قواعد البيانات فيها. على سبيل المثال يستخدم متصفح Firefox قاعدة بيانات SQLite داخليًا كحال تطبيقات أخرى وموقعها http://sqlite.org/ وهي مناسبة للتعامل مع مسائل التلاعب بالبيانات كالتي نشهدها في مجال المعلوماتية مثل استكشاف (spidering) موقع تويتر (آلية تستخدم في محركات البحث للاكتشاف والوصول إلى جميع صفحات الويب على شبكة الانترنت لفهرستها في هذا البرنامج، يكتشف البرنامج قائمة أصدقاء حساب ما على تويتر ويستخدمه لكشف أصدقائهم وهكذا) الذي سنتحدث عنه في هذا الفصل.

2.15 مفاهيم في قواعد البيانات

للوهلة الأولى، تشبه قاعدة البيانات الجدول المليء بالحقول، حيث أن بنى البيانات الرئيسية فيها هي: الجداول والأسطر والأعمدة. بينما يشار إليهم في قواعد البيانات العلائقية كـ: العلاقة (relation) والصف (tuple) والسمة (attribute) على الترتيب، لذلك سنستخدم الكلمات الشائعة في هذا

الكتاب.



الشكل 22: قاعدة بيانات علائقية

3.15 متصفح قاعدة البيانات في SQLite

سنركز في هذا الفصل على استخدام بايثون للتعامل مع البيانات في ملفات قاعدة بيانات SQLite، يسهل برنامج يدعى متصفح قاعدة البيانات في SQLite العديد من العمليات المتاح مجانًا على الموقع http://sqlitebrowser.org/

بإمكانك إنشاء الجداول بسهولة بواسطة المتصفح وإدخال بيانات أو التعديل عليها أو إجراء استعلام بسيط حول البيانات الموجودة في تلك القاعدة، أي أن متصفح قاعدة البيانات يشبه محرر النصوص عند التعامل مع الملفات النصية. فعندما تحتاج لتنفيذ عملية أو عدة عمليات على الملف النصي ستفتحه باستخدام محرر النصوص وتجري التعديلات التي ترغب بها، إلا إن كان لديك العديد من التعديلات والعمليات لتجريها فستنشئ برنامج بايثون يساعدك في تنفيذ هذا، وهذا مشابه لما يتعلق بقواعد البيانات حيث نجري عمليات بسيطة عبر مدير قاعدة البيانات بينما نفضل بايثون للعمليات الأكثر تعقيدًا.

4.15 إنشاء جدول قاعدة بيانات

تتطلب قواعد البيانات بنية محددة أكثر من قوائم وقواميس بايثون (تتيح SQL مرونة أكبر فيما يتعلق بنوع البيانات المخزنة ضمن عمود ما، ولكن سنبقي أنماط البيانات التي سنستخدمها محددة بحيث تنطبق المبادئ بشكل مشابه على أنظمة قواعد بيانات أخرى مثل MySQL ، يجب علينا تحديد أسماء أعمدة الجدول قبل إنشائه إضافةً إلى نوع البيانات التي ننوي تخزينها في تلك الأعمدة ليتمكن البرنامج من اختيار الطريقة الأكثر فعالية لتخزين المعلومة والبحث عنها، كما يمكنك الاطلاع على أنماط البيانات التي تدعمها SQLite عبر الرابط الآتي: http://www.sqlite.org/datatypes.html

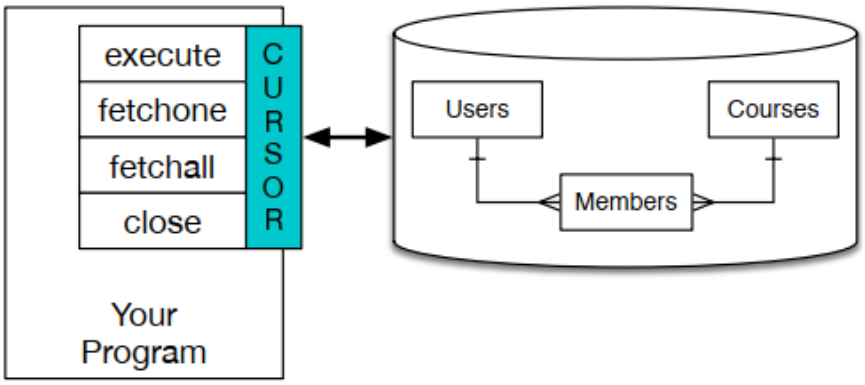
قد يبدو لك تحديد بنية بياناتك مسبقًا أمرًا شاقًا في بداية الأمر ولكن النتيجة تكون الوصول السريع إلى تلك البيانات بالرغم من احتواء قاعدة بياناتك على حجوم كبيرة من المعلومات.

لإنشاء ملف قاعدة بيانات مع جدول اسمه Tracks ذو عمودين بلغة بايثون نكتب الشيفرة التالية:

```
import sqlite3

conn = sqlite3.connect('music.sqlite')

cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks')

cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()


# Code: http://www.py4e.com/code3/db1.py
```

تنشئ تعليمة connect اتصالًا مع قاعدة البيانات الموجودة ضمن الملف music.sqlite في المجلد الحالي، وسيُنشأ الملف في حال عدم وجوده مسبقًا وسبب تسمية هذه العملية بالاتصال فهو بسبب احتمال وجود قاعدة البيانات على مخدم قواعد بيانات منفصل وهو المخدم الذي شغلنا تطبيقنا عن طريقه، أما في أمثلتنا القادمة فستكون قاعدة البيانات ملف محلي موجود بنفس مجلد برنامج بايثون الذي سننفذه.

يشبه المؤشر (cursor) معرَف الملف (file handle) حيث نستخدمه لتنفيذ عمليات معينة على قاعدة البيانات، استدعاء التابع cursor () يشبه من حيث المبدأ استدعاء التابع open () عند التعامل مع الملفات النصية.



الشكل 23: مؤشر قاعدة البيانات

نستطيع بدء تنفيذ الأوامر على محتويات قاعدة البيانات مستخدمين التابع excute() بمجرد حصولنا على المؤشر، ويُعبَّر عن تلك الأوامر بلغة خاصة موحدة من قبل مجموعة من شركات قواعد البيانات، الأمر الذي يتيح لنا تعلم لغة واحدة وتسمى لغة الاستعلام البنيوية أو اختصارًا SQL ويمكنك الاطلاع على معلومات عنها عبر الرابط: http://en.wikipedia.org/wiki/SQL

نفذنا تعليمتين من تعليمات قواعد البيانات في مثالنا السابق حيث سنكتب الكلمات المفتاحية لهذه اللغة بأحرف كبيرة والأجزاء التي سنضيفها على الأوامر المستخدمة بأحرف صغيرة (كأسماء الجداول والأعمدة).

تحذف التعليمة الأولى الجدول Tracks من قاعدة البيانات إذا كان موجود مسبقًا مما يسمح لنا بإنشاء الجدول مجددًا في كل مرة تشغيل بدون حدوث أخطاء، مع ملاحظة أن تعليمة DROP TABLE تحذف الجدول مع جميع محتوياته من قاعدة البيانات (لا يمكن التراجع عن العملية).

```
cur.execute('DROP TABLE IF EXISTS Tracks')
```

بينما تنشئ التعليمة الثانية جدول اسمه Tracks ذو عمود باسم title محتوياته نصية وعمود باسم play محتوياته أعداد صحيحة.

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

نستطيع بعد إنشاء الجدول إضافة بعض البيانات إليه باستخدام عملية INSERT وهذا بعد إنشاء اتصال جديد مع قاعدة البيانات والحصول على المؤشر لنتمكن من تنفيذ أوامر SQL باستخدام ذلك المؤشر.

يشير الأمر INSERT إلى الجدول الذي نستخدمه ثم يعرّف سطرًا جديدًا بتحديد الحقول التي نريد تضمينها (title, plays) متبوعة بالقيم التي نريد إدخالها إلى السطر الجديد، كما نكتب القيم كعلامات استفهام (?,?) لنبين أن تلك القيم الفعلية ستُدخل لاحقًا كصف ('My way', 15) في المعامل الثاني للتابع execute ()

```
import sqlite3
conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()
cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)', ('Thunderstruck', 20))
cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)', ('My Way', 15))
```

```
conn.commit()

print('Tracks:')

cur.execute('SELECT title, plays FROM Tracks')

for row in cur:

    print(row)

cur.execute('DELETE FROM Tracks WHERE plays < 100')

conn.commit()

cur.close()


# Code: http://www.py4e.com/code3/db2.py
```

Tracks

title	plays
Thunderstruck	20
My Way	15

الشكل24 : الأسطر في الجدول

نضيف بدايةً سطرين إلى الجدول باستخدام INSERT ثم نستخدم commit() لكتابة البيانات ضمن قاعدة البيانات، ثم نستخدم الأمر SELECT لاستدعاء الصفوف التي أدخلناها سابقًا إلى الجدول ونحدد فيه الأعمدة التي نريد (title, plays) كما نحدد أي جدول نريد استخراج البيانات منه، وبعد تنفيذ SELECT يصبح بإمكاننا المرور على محتويات المؤشر باستخدام حلقة for، مع العلم أن المؤشر لا يقرأ جميع محتويات قاعدة البيانات عند استخدام تعليمة SELECT وذلك لزيادة الكفاءة حيث يقرأ البيانات التي نحتاجها وفق تكرارات حلقة for ويكون خرج البرنامج كما يأتي:

```
Tracks:

('Thunderstruck', 20)

('My Way', 15)
```

تستخرج الحلقة سطرين عبارة عن صفوف بحيث تكون القيمة الأولى هي العنوان title والقيمة

الثانية عبارة عن عدد مرات تشغيل الأغنية plays.

ملاحظة: قد تجد سلاسل نصية تبدأ بحرف 'u' في كتب أخرى أو على الإنترنت حيث كان هذا دليلًا في إصدار بايثون 2 على كون السلاسل مرمزة بترميز Unicode أي سلاسل تتضمن مجموعة المحارف غير اللاتينية لكن في بايثون الإصدار 3 جميع السلاسل هي Unicode افتراضيًا.

ننفذ في نهاية البرنامج أمر الحذف DELETE لإزالة الصفوف التي أنشأناها لنتمكن من إعادة تشغيل البرنامج عدة مرات، كما يظهر هذه الأمر استخدام عبارة WHERE والتي تسمح لنا بتحديد الصفوف التي ستنفذ عليها تعليمة الحذف، ولكن صدف في هذا المثال أن طابق معيار التحديد جميع صفوف قاعدة البيانات لإخلاء الجدول وتشغيل البرنامج بشكل متكرر، ونستدعي commit() بعد تنفيذ DELETE لتأكيد حذف البيانات من قاعدة البيانات.

5.15 ملخص عن لغة الاستعلام البنيويّة SQL

استخدمنا لحد الآن لغة SQL في برامج بايثون السابقة وذكرنا العديد من أوامرها الأساسية، لذا سنركز عليها بشكل أكبر في هذا القسم وسنقدم نظرة عامة على قواعد هذه اللغة، وباعتبار وجود شركات عدة مختصة بقواعد البيانات فقد حُددت SQL كلغة معيارية لنتمكن من التواصل باستخدام مختلف أنظمة قواعد البيانات التابعة لتلك الشركات.

إن قواعد البيانات العلائقية مكونة من جداول وسطور وأعمدة حيث تكون الأعمدة ذات نوع معين نصي أو رقمي أو تواريخ وعند إنشاء جدول نصرح بأسماء وأنواع الأعمدة مثال:

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

ولإضافة سطر جديد ضمن الجدول نستخدم الأمر INSERT في لغة SQL مثال:

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

تحدد هذه التعليمة اسم الجدول ومجموعة من الحقول (الأعمدة) التي تود إضافتها للسطر الجديد، كما تضيف الكلمة المفتاحية VALUES مجموعة من القيم الموافقة لكل حقل.

يُستخدم الأمر SELECT لاستدعاء السطور والأعمدة التي نحتاجها من قاعدة البيانات أما عبارة WHERE تظهر السطور المطلوبة، كما يمكن استخدام عبارة ORDER BY عند الحاجة للتحكم بترتيب الصفوف المستدعاة مثال:

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

يدل استخدام رمز النجمة * على استدعاء جميع أعمدة لكل الأسطر التي تحقق شرط عبارة WHERE، وعلى عكس بايثون نستخدم هنا إشارة يساوي واحدة في هذا الشرط بدلًا من إشارتين، بينما تكون بقية العمليات المنطقية المتاحة مع WHERE هي نفسها <، >، <=، >=، =!، إضافة إلى الأقواس واستخدام AND وOR.

تستطيع تحديد ترتيب الصفوف اعتمادًا على أحد الحقول مثال:

```
SELECT title,plays FROM Tracks ORDER BY title
```

ولحذف سطر ما تحتاج لاستخدام تعليمة DELETE وعبارة WHERE لتحديد أي السطور تريد حذفها مثال:

```
DELETE FROM Tracks WHERE title = 'My Way'
```

يمكن أيضًا تحديث قيمة أحد الأعمدة أو كلها ضمن سطر أو أكثر في جدول ما باستخدام تعليمة UPDATE كما يأتي:

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

تُحدد هذه التعليمة جدول معين ثم مجموعة القيم المطلوب تغييرها بعد تعليمة SET أما عبارة WHERE فلتحديد الصفوف المراد تحديثها، وتُعدل تعليمة UPDATE كافة السطور التي توافق شرط WHERE أو جميع سطور الجدول في حال عدم استخدام WHERE.

تسمح تعليمات SQL الأربعة هذه (INSERT و SELECT و UPDATE و DELETE) بتنفيذ عمليات إنشاء وتعديل البيانات.

6.15 استكشاف تويتر باستخدام قواعد البيانات

سنكتب في هذا القسم برنامج استكشاف أو تعقب يمر على حسابات تويتر ويسجل بياناتهم ضمن قاعدة بيانات (كن حذرًا عند تشغيل هذا البرنامج كي لا تتسبب في إلغاء وصولك إلى تويتر إن استخرجت كم كبير من البيانات أو استخدمت البرنامج كثيرًا).

إحدى مشاكل هذه البرامج احتياجنا لإعادة تشغيله عدة مرات دون فقد البيانات التي قد استخرجتها وإعادة استدعائها مرة أخرى من البداية لذلك نود تخزين تلك البيانات ليتمكن برنامجنا من إكمال العمل من نقطة توقفه.

سنبدأ باستخراج قائمة أصدقاء حساب تويتر معين ومنشوراته (تغريداته) ثم سنمر باستخدام الحلقات على قائمة الأصدقاء لنضيفهم إلى قاعدة بيانات لاستدعائهم مستقبلًا، بعد ذلك نأخذ اسم حساب أحد الأصدقاء لنستدعي قائمة أصدقائه ونضيف أسماء الأصدقاء التي لم ترد في قاعدة البيانات سابقًا ونتابع هذه العملية للصديق التالي وهكذا، ثم نحسب عدد مرات تكرار اسم أحد الأصدقاء كتعبير عن شعبيته، ويصبح بإمكاننا بعد انتهاء هذه العملية إعادة تشغيل البرنامج مرارًا وتكرارًا، كما يعتبر هذا البرنامج معقدًا بعض الشيء حيث يعتمد على البرامج المكتوبة في تمارين سابقة والتي تستخدم واجهة برمجية API لتويتر، وتكون الشيفرة المصدرية له كالآتي:

```
from urllib.request  import urlopen
import urllib.error
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute( ' ' '
            CREATE TABLE IF NOT EXISTS Twitter
            (name TEXT, retrieved INTEGER, friends INTEGER)' ' ')

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
```

```
while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == quit): break
    if (len(acct) < 1) :
        cur.execute('SELECT name FROM Twitter WHERE retrieved = 0      LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print('No unretrieved Twitter accounts found')
            continue
    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'} )
    print('Retrieving', url)
    connection = urlopen(url, context=ctx)
    data = connection.read().decode()
    headers = dict(connection.getheaders())


    print('Remaining', headers['x-rate-limit-remaining'])
    js = json.loads(data)
    # Debugging
    # print json.dumps(js, indent=4)
    cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ? ', (acct, ))
    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
```

```
        print(friend)
        cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
                    (friend, ))
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET friends = ? WHERE name
                =?',  (count+1,friend))
            countold = countold + 1
        except:
            cur.execute(''' INSERT INTO Twitter (name, retrieved, friends) VALUES
                        (?, 0, 1) ''',  (friend, ))
            countnew = countnew + 1
    print('New accounts=', countnew, 'revisited=', countold)
    conn.commit()
cur.close()


# Code: http://www.py4e.com/code3/twspider.py
```

إن قاعدة بياناتنا مخزنة في الملف spider.sqlite وتحتوي جدول واحد اسمه Twitter، حيث كل سطر في هذا الجدول يحوي عمود باسم name لاسم الحساب وعمود باسم retrieved يدل إن كنا قد استخرجنا قائمة أصدقائه وعمود باسم friends يمثل عدد مرات إضافته كصديق.

سنطلب من المستخدم في الحلقة الرئيسية للبرنامج إدخال حساب تويتر أو كتابة كلمة quit للخروج من البرنامج، فإن أدخل حساب تويتر سنستخرج قائمة أصدقائه ونخزنها في قاعدة البيانات في حال عدم وجودها مسبقًا، وإن كان اسم الصديق موجود مسبقًا فنزيد بمقدار 1 خانة friends في السطر المخصص في قاعدة البيانات، وفي حال ضغط المستخدم مفتاح enter -بدون كتابة أي شيء- نبدأ باستخراج قائمة أصدقاء الحساب التالي - من القائمة التي استخرجناها مسبقًا- لنضيفهم إلى قاعدة البيانات أو نحدث بياناتهم ونزيد عدّاد أصدقائهم (أي الخانة friends)، وبمجرد الانتهاء من عملية الاستخراج نمر على كامل عناصر القاموس users في شيفرة JSON المعادة لنستخرج اسم الحساب

(المخزن في المتغيرscreen_name) لكل مستخدم، ثم نستخدم تعليمة SELECT لنتحقق فيما إذا كان الاسم مخزنًا في قاعدة البيانات وإن كان مُسجلًا نستعيد عدّاد الأصدقاء ونحدث قيمته.

```
countnew = 0
countold = 0
for u in js['users'] :
    friend = u['screen_name']
    print(friend)
    cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
                (friend, ) )
    try:
        count = cur.fetchone()[0]
        cur.execute('UPDATE Twitter SET friends = ? WHERE name = ? ',
                    (count+1,friend) )
        countold = countold + 1
    except:
        cur.execute('INSERT INTO Twitter (name, retrieved, friends) VALUES ( ?,
                    0, 1 ) ', ( friend, ) )
        countnew = countnew + 1
print('New accounts'=,countnew, 'revisited=',countold)
conn.commit()
```

ويجب أن نستعيد السطور بمجرد تنفيذ تعليمة SELECT باستخدام حلقة for ولكن من الأفضل كوننا سنستعيد سطر واحد (LIMIT 1) استخدام التابع fetchone() لجلب السطر الأول والوحيد الناتج عن الأمر SELECT، وبما أن هذا التابع يعيد السطر كصف (بالرغم من وجود خانة واحدة) نأخذ أول قيمة من الصف [0] لنحصل على قيمة عدّاد الأصدقاء الحالي ونضعها في المتغير count، وفي حال نجاح هذه العملية نستعمل الأمر Update مع عبارة WHERE لزيادة 1 إلى قيمة عامود عداد الأصدقاء friends في السطر الموافق لحساب الصديق المطلوب، مع ملاحظة وجود عنصرين نائبين (علامات الاستفهام) في شيفرة SQL حيث المعامل الثاني للتابع execute () هو صف ذو عنصرين

يحتوي القيم البديلة لعلامات الاستفهام.

إن حدث فشل في تنفيذ تعليمات try فغالبًا السبب هو عدم تطابق أي سجل مع العبارة WHERE name=? ضمن تعليمة SELECT لذلك نستخدم ضمن تعليمة except تعليمة INSERT لإضافة الاسم الجديد screen_name إلى الجدول مع وضع قيمة 0 في خانة retrived لتشير إلى عدم استدعاء الاسم بعد ثم نسند قيمة 1 إلى خانة عداد الأصدقاء.

كخلاصة عند تشغيل البرنامج للمرة الأولى ندخل اسم حساب تويتر، فيعمل كالآتي:

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20 revisited= 0
Enter a Twitter account, or quit: quit
```

وباعتبارها المرة الأولى لتشغيل البرنامج تكون قاعدة البيانات فارغة وننشئها ضمن الملف spider.sqlite مع إضافة جدول باسم Twitter، ثم نستخرج بعض أسماء الأصدقاء ونضيفهم إلى قاعدة البيانات، ولربما ترغب بكتابة تعليمات تظهر لك مضمون الملف بعد تنفيذ ما سبق:

```
import sqlite3
conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur:
    print(row)
    count = count + 1
print(count, 'rows. ')
cur.close()

# Code: http://www.py4e.com/code3/twdump.py
```

ببساطة يفتح هذا البرنامج قاعدة البيانات ويحدد جميع الأعمدة لكل الأسطر ضمن جدول Twitter، ثم يمر على كل سطر ويعرضه، ويكون خرجه عند تنفيذ أول عملية استكشاف كما يأتي:

```
('opencontent', 0, 1)
('lhawthorn', 0, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
20 rows.
```

نلاحظ وجود سطر واحد لكل اسم screen_name لم نستخرج بياناته حيث يوجد صديق واحد لكلٍّ منهم ضمن قاعدة البيانات، حيث تعكس هذه القاعدة عملية استخراج أصدقاء حساب تويتر معين للمرة الأولى، كما باستطاعتنا إعادة تشغيل البرنامج لنستخرج أصدقاء الحساب التالي، وذلك عن طريق الضغط على مفتاح enter بدلًا من إدخال اسم حساب جديد كما يأتي:

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18 revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17 revisited= 3
Enter a Twitter account, or quit: quit
```

وتكون الشيفرة البرمجية المنفذة هي :

```
if (len(acct) < 1) :
    cur.execute('SELECT name FROM Twitter WHERE
                retrieved = 0   LIMIT  1')
    try:
        acct = cur.fetchone()[0]
```

```
    except:
        print('No unretrieved twitter accounts found')
        continue
```

ثم نستخدم الأمر SELECT لاستخراج اسم المستخدم الأول (LIMT 1) صاحب القيمة الصفرية في عامود retrived، كما نستخدم العبارة fetchone()[0] ضمن كتلة try/except لاستخراج الاسم screen_name من البيانات المستعادة أو العودة للمرور على بقية المستخدمين، وفي حال نجحنا باستدعاء الاسم لحساب غير مكتشف فنستخرج بياناته كما يلي:

```
url=twurl.augment(TWITTER_URL,{ 'screen_name': acct, 'count': '20'})
print('Retrieving', url)
connection = urllib.urlopen(url)
data = connection.read()
js = json.loads(data)


cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ? ',(acct, ))
```

نستخدم تعليمة UPDATE بمجرد نجاح العملية السابقة، وذلك لتغيير قيمة العمود retrieved إلى الواحد للإشارة إلى انتهاء عملية استخراج قائمة أصدقاء المستخدم لعدم استخراج نفس البيانات مرارًا وتكرارًا والسماح لنا بالتقدم عبر شبكة الأصدقاء في تويتر.

عند تشغيل البرنامج والضغط على مفتاح enter مرتين لاستخراج أصدقاء الصديق نحصل على الخرج الآتي:

```
('opencontent', 1, 1)
('lhawthorn', 1, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
```

```
('cnxorg', 0, 2)
('knoop', 0, 1)
('kthanos', 0, 2)
('LectureTools', 0, 1)
...
55 rows.
```

كما نرى فقد وثّقنا زيارة الحسابين lhawthorn و opencontent ونلاحظ وجود متابعين لكل من الحسابين cnxorg وkthanos، وبما أننا استخرجنا قائمة أصدقاء ثلاثة أشخاص حتى الآن وهم drchuck و opencontent وlhawthorn فإن جدولنا الأن يحتوي على 55 سطرٍ من الأصدقاء لاستخراج بياناتهم، وفي كل مرة تشغيل سيختار البرنامج الحساب التالي غير المزار بعد ضغط enter (على سبيل المثال الحساب التالي الواجب معالجته هوsteve_coppin) ثم نستخرج قائمة الأصدقاء لهذا الحساب ونضيفهم لنهاية قاعدة البيانات أو نحدث عداد أصدقائهم إن كانوا موجودين ضمن الجدول مسبقًا، وباعتبار أن بيانات البرنامج مخزنة على قرص في قاعدة بيانات فيمكن إيقاف عملية الاكتشاف مؤقتًا وإكمالها بعدد المرات الذي نحتاجه دون فقد البيانات.

7.15 نمذجة البيانات

تكمن قوة قاعدة البيانات العلائقية الحقيقية في إمكانية إنشاء عدة جداول وربطها ببعضها البعض، وتدعى عملية تقسيم البيانات إلى عدد من الجداول وإنشاء روابط بينها بنمذجة البيانات (data modeling)، ويدعى المخطط الذي يظهر الجداول والعلاقات بينها بنموذج البيانات.

تتطلب عملية نمذجة البيانات مهارات وخبرة كبيرة نسبيًا لذلك سنتطرق إلى أساسيات نمذجة قواعد البيانات العلائقية في هذا القسم من الفصل، وللحصول على مزيد من المعلومات حول هذا الموضوع بإمكانك زيارة الرابط الآتي: http://en.wikipedia.org/wiki/Relational_model

فلنفترض أننا أردنا إنشاء قائمة تبين جميع علاقات الأصدقاء ببعضهم البعض في البرنامج السابق بدلًا من إحصاء أصدقاء كل شخص فحسب بهدف الحصول على قائمة جميع الأشخاص المتابعين لحساب تويتر معين، وبما أن كل حساب قد يكون مُتابع من عدة حسابات أخرى فلا نستطيع الاكتفاء بإضافة عمود واحد إلى جدول Twitter، لذا ننشئ جدول جديد منفصل لنحدد طرفي الصداقة،

ونوضح في الشيفرة التالية طريقة التنفيذ:

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

وفي كل مرة نضيف شخص يُتابعه drchuck نضيف سطر كالتالي:

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

وبما أننا سنتعامل مع 20 صديق من أصدقاء حساب تويتر الخاص بـ " drchuck " أي سنضيف 20 مرة اسم " drchuck " باعتباره المعامل الأول مما يعني إعادة ذكر اسم الحساب عدة مرات في قاعدة البيانات، ينتهك هذا التكرار أهم معايير قواعد البيانات والذي ينص على ألا نكرر سلسلة نصية نفسها أكثر من مرة، فإن احتجنا تلك السلسلة أكثر من مرة استعضنا عنها برقم، حيث عمليًا تشغل السلاسل النصية حجمًا أكبر من الأرقام على قرص التخزين وفي ذاكرة الحاسوب وتتطلب زمن معالجة أكبر في عمليات المقارنة والترتيب، لكن قد يكون زمن المعالجة ومساحة التخزين غير مهمين في حال وجود بضعة مئات فقط من المدخلات في قاعدة البيانات، أما إذا كان لدينا مليون شخص في قاعدة البيانات مع احتمال وجود رابط مع 100 مليون صديق فتكون لسرعة عملية البحث في البيانات أهمية كبرى.

سنخزن حسابات تويتر المستخرجة في جدول اسمه People بدلًا من Twitter المستخدم في المثال السابق، ويحتوي هذا الجدول على عمود إضافي لتخزين المفتاح الرقمي المرتبط بالسطر الخاص بحساب تويتر محدد، مع الأخذ بعين الاعتبار أن SQLite تملك ميزة إضافة قيمة المفتاح تلقائيًا لأي سطر ندخله للجدول باستخدام عمود ذي نوع بيانات مخصص (INTEGER PRIMARY KEY).

ننشئ جدول People مع عمود إضافي يسمى id كما يأتي:

```
CREATE TABLE People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

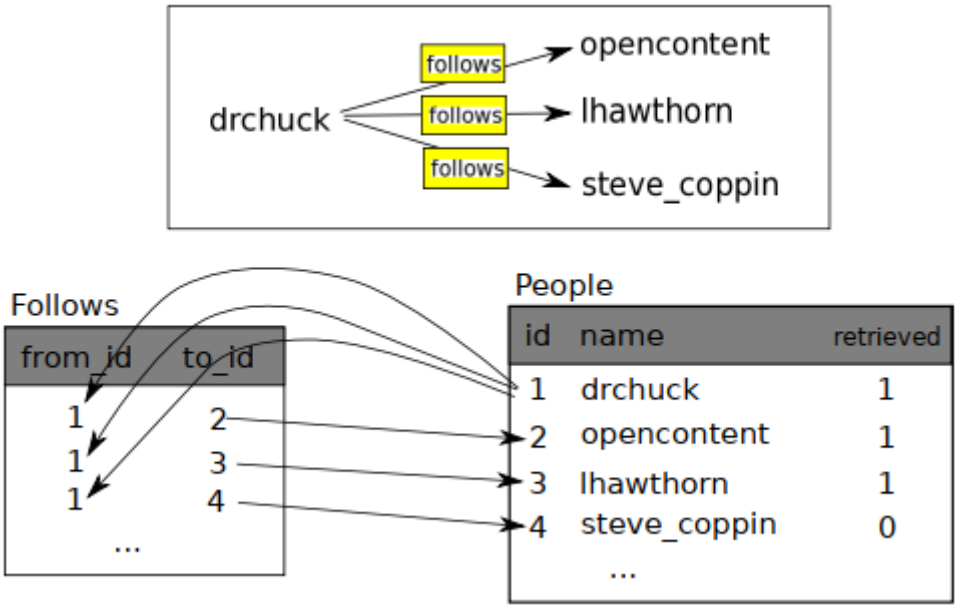
ونلاحظ أننا لم نعد نحتفظ بعدّاد الأصدقاء في كل سطر ضمن جدول People، وعند تحديد نوع العمود id كـ (INTEGER PRIMARY KEY) فهذا يعني أننا نرغب في أن تسند SQLite رقم فريد لكل سطر ندخله تلقائيًا، كما أضفنا الكلمة المفتاحية UNIQUE للإشارة إلى عدم سماحنا بإدخال سطرين بنفس القيمة للخانة name، وبدلًا من إنشاء جدول Pals كما فعلنا أعلاه سننشئ جدول يدعى Follows ذي عمودين كلاهما من نوع عدد صحيح وسندعوهما from_id و to_id ونركز على كون كل زوج من هاتين الخانتين فريد في الجدول (أي لا نستطيع تكرار السطر) في قاعدة البيانات.

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id) )
```

نحددعند إضافة عبارة UNIQUE للجداول مجموعة قواعد نطلب من قاعدة البيانات تنفيذها أثناء إدخالنا السجلات، وغرضنا من وضع هذه القواعد هو جعل برنامجنا أسهل ومريح أكثر كما سنرى لاحقًا فهذه القواعد تمنعنا من ارتكاب الأخطاء وتسهل كتابة جزء من برنامجنا، وباختصار فإن إنشاء جدول Follows يمثل نموذجًا للعلاقة بين الأشخاص عندما يتابع أحدهم شخصًا آخر وذلك من خلال زوج من الأرقام التي تشير إلى وجود علاقة بين هؤلاء الأشخاص واتجاه هذه العلاقة.

8.15 برمجة قاعدة البيانات ذات الجداول المتعددة

سنعدل على برنامج اكتشاف تويتر بحيث يحوي جدولين للمفاتيح الأساسية وعلاقات المفاتيح كما وضحنا أعلاه، ويكون البرنامج كما يأتي:



الشكل 25 : العلاقات بين الجداول

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
```

```
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS People  (id INTEGER
                PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows (from_id INTEGER,
                to_id INTEGER, UNIQUE(from_id, to_id))''')

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute('SELECT id, name FROM People WHERE retrieved=0
                    LIMIT 1')
        try:
            (id, acct) = cur.fetchone()
        except:
```

```
        print('No unretrieved Twitter accounts found')
        continue
else:
    cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
                (acct, ))
    try:
        id = cur.fetchone()[0]
    except:
        cur.execute('''INSERT OR IGNORE INTO People     (name, retrieved)
                    VALUES (?, 0)''', (acct, ))
        conn.commit()
        if cur.rowcount != 1:
            print('Error inserting account:', acct)
            continue
        id = cur.lastrowid

url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '100'})
print('Retrieving account', acct)
try:
    connection = urllib.request.urlopen(url, context=ctx)
except Exception as err:
    print('Failed to Retrieve', err)
    break

data = connection.read().decode()
headers = dict(connection.getheaders())
```

```
print('Remaining', headers['x-rate-limit-remaining'])


try:
    js = json.loads(data)
except:
    print('Unable to parse json')
    print(data)
    break

# Debugging
# print(json.dumps(js, indent=4))

if 'users' not in js:
    print('Incorrect JSON received')
    print(json.dumps(js, indent=4))
    continue

cur.execute('UPDATE People SET retrieved=1 WHERE name = ?', (acct, ))

countnew = 0
countold = 0
for u in js['users']:
    friend = u['screen_name']
    print(friend)
```

```
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
                    (friend, ))
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                        VALUES (?, 0)''', (friend, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', friend)
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Follows (from_id, to_id)
                    VALUES (?, ?)''', (id, friend_id))
    print('New accounts=', countnew, ' revisited=', countold)
    print('Remaining', headers['x-rate-limit-remaining'])
    conn.commit()
cur.close()


# Code: http://www.py4e.com/code3/twfriends.py
```

يبدو البرنامج وكأنه معقد بعض الشيء إلا أنه يوضح الأنماط البرمجية التي نحتاج لاستخدامها عند الاعتماد على مفاتيح رقمية لربط الجداول حيث تكون تلك الأنماط كما يأتي:

١. إنشاء جداول ذات مفاتيح أساسية (primary keys) وقيود.

٢ . عند وجود مفتاح منطقي (logical key) لشخص ما (مثلًا اسم الحساب) واحتجنا لمعرفة قيمة id الخاصة به، مع الأخذ بعين الاعتبار وجود الشخص في جدول people من عدمه فسنحتاج إما للبحث عن الشخص في جدول people لاستخراج قيمة id الخاصة به أو إضافته إلى الجدول والحصول على id السطر المضاف الجديد.

٣ .إدخال السطر الذي يحدد علاقة الصداقة follows.

وسنغطي كل من هذه النقاط على حدة في الفقرات التالية.

1.8.15 القيود في قواعد البيانات

يمكننا في مرحلة تصميم الجداول تطبيق مجموعة قواعد على قاعدة البيانات، لتجنب ارتكاب الأخطاء أو إدخال بيانات غير صحيحة، ويتم هذا بالآلية التالية:

```
cur.execute('''CREATE TABLE IF NOT EXISTS People   (id INTEGER PRIMARY
KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows (from_id INTEGER,
                to_id INTEGER, UNIQUE(from_id, to_id))''')
```

نشترط أن كل من عمود الاسم name في جدول people وزوج الرقمين (from_id, to_id) في كل سطر من جدول follows فريد (غير مكرر) حيث تمنعنا هذه القيود أو القواعد من ارتكاب الأخطاء كإضافة نفس علاقة الصداقة في مثالنا السابق عدة مرات، ونستطيع الاستفادة منها من خلال التعليمات الآتية:

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved) VALUES ( ?, 0)''', (
friend, ) )
```

أضفنا عبارة OR IGNORE إلى تعليمة INSERT ليتم تجاهل التعليمة INSERT إن انتهكت قاعدة استخدام اسم فريد غير مكرر، أي أننا نستخدم تلك القواعد كشبكة أمان لنتأكد من عدم ارتكاب أخطاء بدون قصد، وبشكل مشابه نستخدم التعليمات التالية للتأكد من عدم إضافة نفس العلاقة لجدول follows :

```
cur.execute('''INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)''',
(id, friend_id) )
```

أي ببساطة طلبنا من قاعدة البيانات تجاهل تعليمة الإدخال إن كانت تنتهك قاعدة عدم التكرار في

أسطر الجدول Follows.

2.8.15 استعادة أو إضافة سجل في قاعدة البيانات

عندما نطلب من المستخدم إدخال اسم حساب تويتر معين ونجده ضمن قاعدة البيانات علينا البحث عن قيمة معرّفه id، بينما إن لم يكن موجود في جدول people فعلينا إضافة السجل والحصول على قيمة المعرّف في السطر المضاف، وهذا أسلوب شائع جدًا ونُفذ مرتين في البرنامج السابق أعلاه، حيث يظهر البرنامج كيفية البحث عن معرّف حساب صديق ما بعد استخراج الاسم screen_name من عقدة user في ملف JSON المستعاد، وباعتبار وجود احتمالية لوجود الحساب في قاعدة البيانات مسبقًا لذا علينا أولًا تفقد وجوده مسبقًا في جدول people عن طريق تعليمة SELECT، فإن لم نواجه أي أخطاء في بنية (try/except) نستخرج السجل باستخدام fetchone () ثم نستخرج أول عنصر (وهو العنصر الوحيد) من الصف المستعاد ونخزنه في المتحول friend_id، فإن فشلت تعليمة SELECT ستفشل تعليمات fetchone () [0] وسينتقل التنفيذ إلى قسم except

```
friend = u['screen_name']
cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1'
        (friend, ) )
try:
    friend_id = cur.fetchone()[0]
    countold = countold + 1
except:
    cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( friend, ) )
    conn.commit()
    if cur.rowcount != 1 :
    print('Error inserting account:',friend)
        continue
    friend_id = cur.lastrowid
```

```
    countnew = countnew + 1
```

وإن نُفذت تعليمات قسم except فهذا يعني أننا لم نعثر على السطر لذلك سيتوجب علينا إضافة السطر، أي نستخدم INSERT OR IGNORE لتجنب الأخطاء ثم نستدعي commit() لإجبار قاعدة البيانات لإجراء عملية تحديث للبيانات، وبعد انتهاء عملية الكتابة يصبح بإمكاننا تفقد قيمة cur.rowcount لمعرفة عدد السطور المتأثرة وإن كان عدد الصفوف المتأثرة لا يساوي الواحد فهذا خطأ حيث أننا نعمل على إدخال صف واحد فقط، وإن نجحت عملية INSERT نستطيع تفقد قيمة cur.lastrowid لمعرفة القيمة التي أسندتها قاعدة البيانات لعمود id في السطر الجديد المُنشَأ.

3.8.15 تخزين علاقة الصداقة بين مستخدمي تويتر

بمجرد معرفة قيمة المفتاح لكلٍّ من مستخدم تويتر وصديقه في ملفات JSON تصبح عملية إدخال القيم إلى جدول Follow عملية بسيطة وذلك عن طريق التعليمة الآتية:

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES
            (?, ?)', (id, friend_id) )
```

كما يجب ملاحظة أن قاعدة البيانات تمنعنا من إدخال نفس العلاقة مرتين عبر إضافة القيود وإضافة OR IGNORE إلى تعليمة INSERT، ونعرض هنا نتيجة تنفيذ هذا البرنامج:

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20 revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17 revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17 revisited= 3
```

```
Enter a Twitter account, or quit: quit
```

كما نرى، فقد بدأنا بمعالجة الحساب drchuck (اسم حساب المؤلف في تويتر) ثم ندع البرنامج يختار الحسابين التاليين لاستخراج بياناتهما وإضافتها إلى قاعدة البيانات تلقائيًا، ونعرض هنا بعض الصفوف الأولى من جدولي people وfollows الناتجة بعد اكتمال تشغيل البرنامج:

```
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

نجد هنا من جدول people رقم المعرّف ID والاسم Name والخانة الأخيرة visited تشير فيما إذا كان الحساب قد تم معالجته أم لا، ونلاحظ أنه تمت معالجة الحسابات الثلاثة الأولى، أما في جدول follows فنجد أرقام توضح العلاقة بين الأصدقاء حيث تبين البيانات الظاهرة أن المستخدم الأول drchuck صديق لجميع الأشخاص الظاهرة أسماؤهم في الصفوف الخمسة الأولى، وهذا منطقي فقد استخرجنا أولًا قائمة أصدقاء drchuck وخزناها وهكذا إن استطعت إظهار صفوف أكثر من جدول follows فسترى أصدقاء المستخدمين 2 و3 أيضًا.

9.15 أنواع المفاتيح الثلاثة

بما أننا بدأنا ببناء نموذج بيانات من خلال إضافة بياناتنا إلى عدد من الجداول المرتبطة ببعضها البعض، حيث ربطنا السطور باستخدام المفاتيح فلابد من التعرف على بعض المصطلحات المتعلقة بهذا الموضوع، حيث يوجد بشكل عام ثلاثة أنواع من المفاتيح المستخدمة في نمذجة البيانات:

- **المفتاح المنطقي (logical key):** هو مفتاح يستخدم علميًا للبحث عن سطر معين، وهو في مثالنا خانة name الذي يمثل اسم المستخدم وقد بحثنا اعتمادًا عليه لنحصل على بيانات سطر خاصة بأحد المستخدمين، لاحظنا سابقًا أنه من المفيد فرض قيود على المفتاح المنطقي بحيث يكون غير مكرر UNIQUE ولكن بما أنه -أي المفتاح- يعبّر عن كيفية بحثنا عن سطر معين من وجهة نظر العالم الخارجي فمن المنطقي أحيانًا السماح لعدة سطور ضمن الجدول بالحصول على نفس القيمة.
- **المفتاح الرئيسي (primary key):** هو رقم تولده قاعدة البيانات تلقائيًا، يستخدم لربط عدة سطور من جداول مختلفة فلا فائدة لوجوده خارج البرنامج، وتكون الطريقة الأسرع لإيجاد سطر ما ضمن أحد الجداول بالبحث عنه عن طريق المفتاح الرئيسي باعتباره رقم صحيح حيث لا يشغل مساحة تخزين كبيرة ويمكن مقارنته وترتيبه بسرعة، وكمثال عنه في نموذجنا الخانة id .
- **المفتاح الخارجي (foreign key):** وهو رقم يشير إلى المفتاح الرئيسي لسطر بجدول أخر، وكمثال عليه في نموذجنا المفتاح from_id.

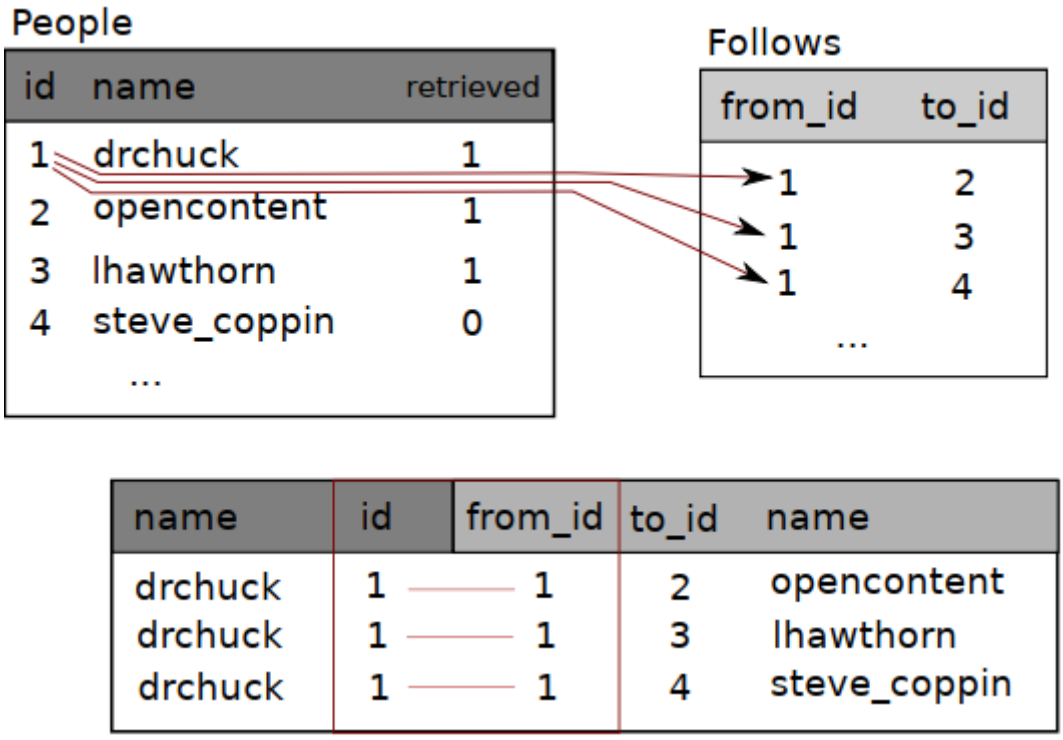
مع الأخذ بعين الاعتبار أننا استخدمنا نظام تسميات معين حيث أشرنا إلى المفتاح الرئيسي بـ id بينما أشرنا إلى المفتاح الخارجي باسم ينتهي باللاحقة _id .

10.15 استخدام عبارة JOIN لاستعادة البيانات

الآن وقد قطعنا شوطًا اتبعنا فيه معايير تصميم قواعد البيانات وفصلنا البيانات ضمن جدولين وربطنا بينهما باستخدام المفاتيح الرئيسية والخارجية أصبح بالإمكان أن نستخدم تعليمة SELECT لجمع البيانات من الجدولين. تستخدم لغة SQL عبارة JOIN لربط تلك الجداول ببعضها حيث تسمح لنا بتحديد الخانات اللازمة لربط السطور بين الجداول المختلفة، كما في المثال الآتي:

```
SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 1
```

وتشير JOIN إلى أننا نسترجع الخانات المتقاطعة بين الجدولين Follows وPeople وتشير عبارة ON إلى شرط تحقيق هذا التقاطع، نفسر التعليمة السابقة كما يلي: نستعيد السطور من جدول Follows ونضيف إليه السطر من جدول People حيث تتطابق قيمة from_id في جدول follows مع قيمة id في جدول People .



الشكل 26 : ربط الجداول باستخدام JOIN

باختصار إن مهمة JOIN هي إنشاء سطور معدلة طويلة تحتوي على خانات من جدول People والخانات الموافقة لها في جدول Follows حيث يوجد أكثر من حالة توافق ما بين حقل id من جدول People وحقل from_id من جدول Follows، أي تنشئ عبارة JOIN سطر معدل لكلٍّ من أزواج الصفوف المتوافقة مكررةً البيانات بالقدر الذي نحتاجه.

ويوضح البرنامج التالي البيانات التي ستخزن في قاعدة البيانات بعد تشغيل برنامج استكشاف تويتر عدة مرات:

```
import sqlite3

conn = sqlite3.connect('friends.sqlite')
```

```
cur = conn.cursor()
cur.execute('SELECT * FROM People')
count = 0
print('People:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')
cur.execute('SELECT * FROM Follows')
count = 0
print('Follows:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')
cur.execute('''SELECT * FROM Follows JOIN People
                ON Follows.to_id = People.id
                 WHERE Follows.from_id = 2''')
count = 0
print('Connections for id=2:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')
cur.close()
# Code: http://www.py4e.com/code3/twjoin.py
```

عرضنا بيانات الجدولين في بداية البرنامج ثم عرضنا مجموعة جزئية من البيانات المتقاطعة بين الجداول المترابطة ببعضها ويكون خرج البرنامج كما يلي:

```
python twjoin.py
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, 'drchuck', 1)
(2, 28, 28, 'cnxorg', 0)
(2, 30, 30, 'kthanos', 0)
(2, 102, 102, 'SomethingGirl', 0)
(2, 103, 103, 'ja_Pac', 0)
20 rows.
```

لاحظ أننا نجد في الخرج أعمدة الجدولين People و Follows ومجموعة الأسطر الظاهرة في النهاية هي نتيجة استخدام تعليمة SELECT مع عبارة JOIN حيث نفحص في عملية SELECTالأخيرة

حسابات أصدقاء"opencontent" – اسم أحد الحسابات- حيث (People.id=2)، ففي كل سطر معدل من تلك العملية يظهر أول عمودين من جدول Follows متبوعًا بالأعمدة بدءًا من العمود الثالث حتى الخامس من جدول People، وتجد أيضًا أن العمود الثاني (Follows.to_id) يطابق العمود الثالث (People.id) في كلٍّ من السطور المعدلة المضافة.

11.15 الملخص

تناول هذا الفصل أساسيات استخدام قواعد البيانات في لغة بايثون، مع ملاحظة أن كتابة برنامج لاستخدام قاعدة بيانات لتخزين البيانات أكثر تعقيدًا من استخدام قواميس بايثون أو ملفات عادية مما يجعل الميل نحو قواعد البيانات قليلًا نسبيًا إلا عند الحاجة الحقيقية لها وللإمكانيات التي توفرها، ونلخص هنا الحالات التي تكون فيها قواعد البيانات مفيدة جدًا:

1- إن كان برنامجك يحتاج إلى إجراء العديد من التحديثات العشوائية الصغيرة ضمن مجموعة بيانات هائلة الحجم.
2- في حال كانت البيانات ضخمة جدًا بحيث لا تسَع ضمن قاموس ولديك حاجة للبحث عن معلومات معينة بشكل متكرر.
3- عند وجود عملية معالجة تستهلك وقت طويل وتحتاج خلالها إلى وقف وإعادة تشغيل البرنامج وحفظ البيانات من عملية تشغيل إلى أخرى.

باختصار، تستطيع إنشاء قاعدة بيانات بسيطة تحتوي جدول بسيط لتناسب احتياجات تطبيقات مختلفة، إلا أن معظم القضايا ستتطلب عدة جداول وعلاقات بين السطور في مختلف الجداول، ومن المهم التمعن بشكل جيد بالتصميم واتباع معايير قواعد البيانات عند البدء بإنشاء الروابط بين الجداول للاستفادة أقصى ما يمكن من الإمكانيات التي توفرها قاعدة البيانات، وبما أن الهدف الرئيسي لاستخدام قواعد البيانات هو التعامل مع كمية كبيرة من البيانات فمن المهم نمذجة البيانات بطريقة فعّالة ليتمكن البرنامج من العمل أسرع ما يمكن.

12.15 التنقيح

أحد أكثر الإجراءات شيوعًا عند تطوير برامج بلغة بايثون للاتصال بقاعدة بيانات SQLite هو تشغيل برنامج وتفحص النتائج باستخدام متصفح قواعد بيانات الخاص بـ SQLite، حيث يسمح لك المتصفح بتفقد عمل البرنامج، كما يجب أن تكون حذرًا لأن SQLite تمنع برنامجين مختلفين من

تعديل نفس البيانات في نفس الوقت. على سبيل المثال، إن فتحت قاعدة بيانات معينة في متصفح قواعد بيانات وأجريت تعديل ما عليها بدون الضغط على زر "حفظ" فسيحجب المتصفح ملف قاعدة البيانات وسيمنع أي برنامج آخر من الوصول إلى الملف، أي لن يكون برنامج بايثون قادرًا على الوصول الملف في هذه الحالة، وكحل لتلك المشكلة نتأكد من إغلاق متفصح قاعدة البيانات أو استخدام قائمة ملف لإغلاق قاعدة البيانات في المتصفح قبل محاولة الوصول إلى قاعدة البيانات باستخدام بايثون، وبهذا نتجنب مشكلة فشل برنامج بايثون بسبب حجب قاعدة البيانات.

13.15 فهرس المصطلحات

- **السمة (attribute):** إحدى قيم صفوف بايثون يُعرف باسم عمود أو خانة.
- **قاعدة أو قيد (constraint):** عندما نطلب من قاعدة البيانات تطبيق قاعدة ما على أحد الحقول أو السطور في جدول ما، وأشهرها عدم تكرار القيم في حقل معين (مثلًا: يجب أن تكون جميع القيم فريدة).
- **المؤشر (cursor):** يسمح لك بتنفيذ أوامر SQL في قاعدة بيانات معينة واستخراج البيانات منها، كما يُشابه مأخذ الشبكة (socket) أو معرف الملفات (file handle) الخاصين باتصالات الشبكة والملفات.
- **متصفح قواعد البيانات (database browser):** برنامج يسمح لك بالاتصال بشكل مباشر مع قواعد البيانات والتعديل عليها دون الحاجة إلى كتابة برنامج.
- **المفتاح الخارجي (foreign key):** مفتاح رقمي يشير إلى المفتاح الرئيسي الخاص بسطر ما في جدول آخر، وتنشئ هذه المفاتيح الروابط بين السطور المخزنة في جداول مختلفة.
- **الفهرس (index):** بيانات إضافية يحتويها برنامج قاعدة البيانات كالسطور والإدخالات إلى جدول ما بهدف إجراء عمليات البحث بسرعة.
- **المفتاح المنطقي (logical key):** مفتاح نستخدمه للبحث عن سطر معين، على سبيل المثال قد يشكل عنوان البريد الإلكتروني لشخص ما في جدول حسابات المستخدمين قيمة مناسبة ليكون المفتاح المنطقي الخاص ببيانات المستخدم.
- **المعايرة (normalization):** تصميم نموذج بيانات يمنع تكرارها حيث نخزن كل عنصر من

البيانات في مكان واحد ضمن قاعدة البيانات ونربطه بمكان آخر عن طريق المفتاح الخارجي.

- **المفتاح الرئيسي (primary key):** مفتاح رقمي خاص بكل سطر ويستخدم للإشارة إلى سطر ما في جدول مختلف، وعادةً ما تكون قاعدة البيانات مضبوطة بحيث تسنِد قيم هذه المفاتيح بشكل تلقائي أثناء إدخال السطور.
- **العلاقة (relation):** جزء من قاعدة البيانات تحتوي على صفوف (tuples) وسمات (attributes) وتُعرف باسم "جدول".
- **الصف (tuple):** مدخل وحيد ضمن قاعدة بيانات معينة يتألف من مجموعة من السمات، ويُعرف باسم "سطر".

الفصل السادس عشر

العرض المرئيّ للبيانات

16 العرض المرئيّ للبيانات

تعلّمنا إلى حدّ الآن أساسيّات لغة بايثون، وكيفيّة استخدامها مع الشبكات، وقواعد البيانات، وطرائق التعامل مع البيانات.

سندرس في هذا الفصل ثلاث تطبيقات تشمل جميع تلك المفاهيم معًا؛ بهدف إدارة وعرض البيانات مرئيًّا. ويمكنك اعتبار هذه التطبيقات نماذجَ تُعينُكَ عندما تشرع بحلّ مسائل حقيقيّة.

تتوفّر هذه التطبيقات كملفٍّ مضغوط بصيغة "ZIP"، يمكنك تحميله وفكّ ضغطه على حاسوبك الشخصيّ لتشغيله.

1.16 عرض خريطة باستخدام بيانات جغرافيّة من غوغل

سنستخدم في هذا المشروع واجهة غوغل البرمجيّة للترميز الجغرافيّ (Google geocoding API)؛ للبحث عن مواقع جغرافيّة لبعض أسماء جامعات مُدخلة من المستخدم، ثمّ إدراج هذه المواقع على خريطة غوغل.

حمّل التطبيق من هنا: http://www.py4e.com/code3/geodata.zip

أولى القضايا الواجب حلّها هي أنّ النسخة المجّانيّة من واجهة غوغل البرمجيّة للترميز الجغرافيّ محدودة من ناحية عدد الطلبات في اليوم، فإن كان لديك الكثير من البيانات، ترتَّب عليك إيقاف عمليّة البحث وإعادتها عدّة مرّات، لذلك سنحلّ هذه المشكلة على مرحلتين.

في المرحلة الأولى، نقرأ بيانات أسماء الجامعات الموجودة في ملفّ where.data كلّ سطر على حدة، ثمّ نسترجع المعلومات المرمَّزة جغرافيًّا من غوغل لكلّ سطر، ونخزّنها في قاعدة البيانات geodata.sqlite. لكن، قبل استخدام الواجهة البرمجيّة مع كلّ موقع مُدخل من قبل المستخدم، علينا التحقّق من عدم توفّر تلك البيانات لدينا مسبقًا؛ للتأكّد من عدم تكرار الطلب لذات البيانات من غوغل، حيث تعمل قاعدة البيانات بمثابة ذاكرة تخزين مؤقّت (cache) محلّيّة لبيانات الترميز الجغرافيّ.

يمكنك إعادة تنفيذ العمليّة في أيّ وقت بحذف الملفّ geodata.sqlite.

سيقرأ برنامج geoload.py المدخلات من الملفّ where.data، ويتحقّق فيما إذا كان كلّ سطر متوفّرًا مسبقًا في قاعدة البيانات، أم لا. في حال عدم توفّر بيانات الموقع، سنستدعي الواجهة البرمجيّة

للترميز الجغرافيّ لاسترجاع البيانات، وتخزينها ضمن قاعدة البيانات.

الشكل 27 : خريطة غوغل

نبيّن فيما يلي الخرج بعد التشغيل، وذلك بوجود بعض البيانات في قاعدة البيانات:

```
Found in database Northeastern University

Found in database University of Hong Kong, ...

Found in database Viswakarma Institute, Pune, India

Found in database UMD

Found in database Tufts University


Resolving Monash University

Retrieving http://maps.googleapis.com/maps/api/

geocode/json?address=Monash+University

Retrieved 2063 characters { "results" : [

{'status': 'OK', 'results': ... }
```

```
Resolving Kokshetau Institute of Economics and Management
Retrieving http://maps.googleapis.com/maps/api/
geocode/json?address=Kokshetau+Inst ...
Retrieved 1749 characters { "results" : [
{'status': 'OK', 'results': ... }
...
```

المواقع الخمسة الأولى موجودة مسبقًا في قاعدة البيانات، لذلك تمّ تخطيها. يستمرّ البرنامج حتّى يجد مواقع جديدة، ثمّ يبدأ في استرجاعها.

يمكن إيقاف برنامج groload.py في أيّ وقت تريد، بالإضافة إلى وجود عدّاد يمكن استخدامه للحدّ من استدعاءات الواجهة البرمجيّة للترميز الجغرافيّ في كلّ مرّة تشغيل؛ لأنّه لا يجب الوصول إلى المعدّل الأقصى للبيانات اليوميّة، لأنّ where.data لا يحتوي إلّا على بضع مئات من عناصر البيانات، فلن يكون هناك مشكلة. لكنْ، في حال وجود المزيد من البيانات، سيتطلّب ذلك التشغيل عدّة مرّات على مدار أيّام عدّة حتّى تحصل قاعدة البيانات على جميع البيانات الجغرافيّة المرمّزة للمواقع المطلوبة.

بمجرّد توفّر البيانات في geodata.py، يمكنك عرضها باستخدام برنامج geodump.py، حيث يقرأ هذا البرنامج محتوى قاعدة البيانات، وينشئ الملفّ "where.js"؛ ليعرض الموقع وخطوط الطول والعرض على شكل ملفّ Javascript تنفيذيّ.

ويكون ناتج تنفيذ برنامج geodump.py كالتالي:

```
Northeastern University, ... Boston, MA 02115, USA 42.3396998 -71.08975
Bradley University, 1501 ... Peoria, IL 61625, USA 40.6963857 -89.6160811
...
Monash University Clayton ... VIC 3800, Australia -37.9152113 145.134682
Kokshetau, Kazakhstan 53.2833333 69.3833333
...
```

```
12 records written to where.js

Open where.html to view the data in a browser
```

يتألّف الملفّ "where.html" من كود HTML، وJavaScript لمعاينة وعرض خريطة غوغل.

يقرأ أحدث البيانات في "where.js" لعرضها. ويكون محتوى هذا الملفّ على الشكل التالي:

```
myData = [

[42.3396998,-71.08975, 'Northeastern Uni ... Boston, MA 02115'],

[40.6963857,-89.6160811, 'Bradley University, ... Peoria, IL 61625, USA'],

...

];
```

يحتوي هذا المتغيّر المكتوب بلغة JavaScript على قائمة من القوائم. ومن المفترض أن تكون هذه الصيغة مألوفة بالنسبة لك، حيث إنّ الصيغة البرمجيّة لكتابة القوائم بلغة JavaScript تشبه إلى حدٍّ كبير الصيغة البرمجيّة المستخدَمة في بايثون لأجل هذا الغرض.

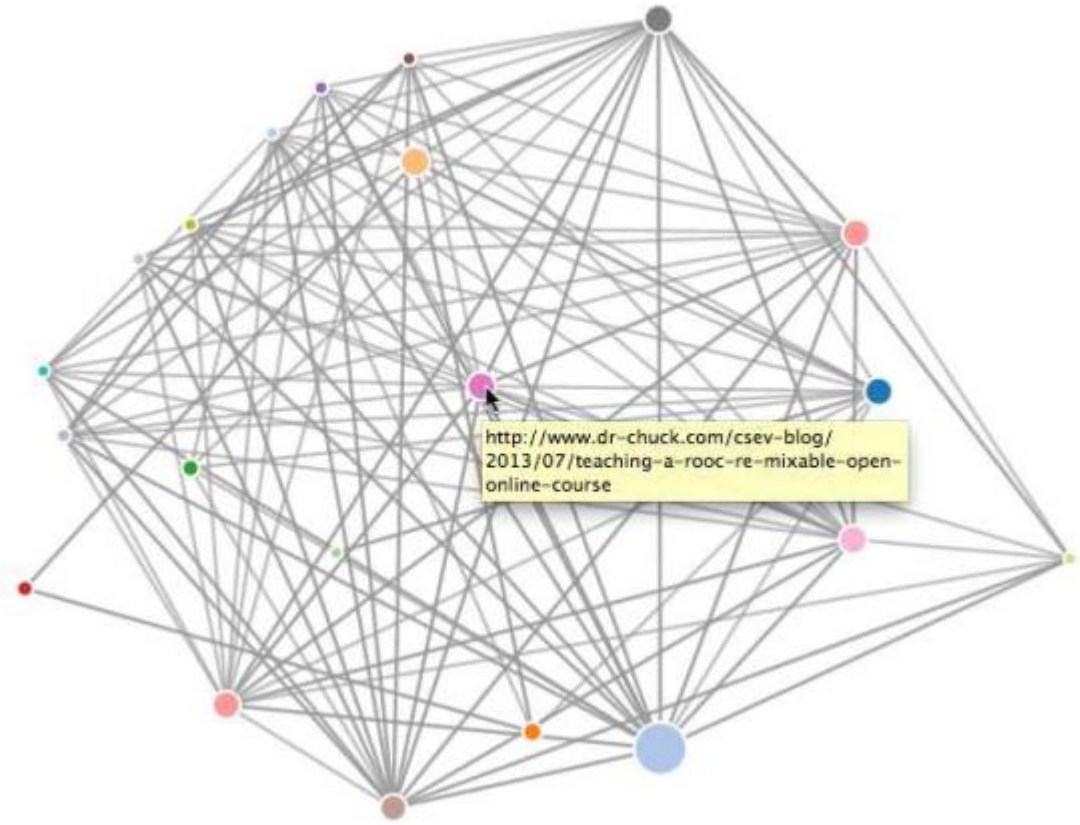
افتح "where.html" على المتصفّح لمعاينة المواقع. يمكنك التحرّك بين المواقع على الخريطة لمعرفة الموقع الذي أوجدته واجهة الترميز الجغرافيّ. وفي حال عدم وجود أيّة بيانات عند فتح ملفّ "where.html"، يجب التحقّق من JavaScript، أو وحدة المطوّرين (developer console) على المتصفّح.

2.16 العرض المرئيّ للشبكات والارتباطات

سننفّذ في هذا التطبيق إحدى وظائف محرّكات البحث. سنكتشف جزءًا صغيرًا من شبكة الإنترنت، ونشغّل نسخة مُبسّطة من خوارزميّة غوغل لترتيب الصفحات؛ بهدف التعرّف على أكثر الصفحات ارتباطًا، ثمّ سنعرض ترتيب وارتباطات الصفحات التي اكتشفناها.

سنستخدم مكتبة D3 JavaScript Visualization (مزيد من المعلومات في الرابط http://d3js.org/) لإعداد عرض الخرج.

يمكن تحميل وفكّ ضغط التطبيق عبر الرابط: www.py4e.com/code3/pagerank.zip



الشكل 28 : رتبة الصفحات

يكتشف البرنامج الأوّل spider.py أحد مواقع الشبكة، ويستخرج منه عدّة صفحات ليخزّنها في قاعدة البيانات spider.sqlite وفقًا للروابط بين هذه الصفحات.

يمكنك إعادة تشغيل هذه العمليّة في أيّ وقت بحذف الملفّ spider.sqlite، وإعادة تشغيل spider.py.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:2
1 http://www.dr-chuck.com/ 12
2 http://www.dr-chuck.com/csev-blog/ 57
How many pages:
```

في المثال السابق، بحث البرنامج في أحد المواقع، وأرجع صفحتين منه. في حال أعدنا تشغيل البرنامج لتعقّب المزيد من الصفحات، فلن يتتبَّع الصفحات التي حُفظَت مسبقًا في قاعدة البيانات.

حالما تعيد تشغيله، يتّجه إلى صفحاتٍ جديدة لم تُكتَشَف بعد؛ ليبدأ من عندها. بالتالي، يمكن القول إنّ مع كلّ عمليّة تشغيل ناجحة للبرنامج spider.py، تضاف صفحات جديدة.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:3
3 http://www.dr-chuck.com/csev-blog 57
4 http://www.dr-chuck.com/dr-chuck/resume/speaking.htm 1
5 http://www.dr-chuck.com/dr-chuck/resume/index.htm 13
How many pages:
```

يمكنك الحصول على عدّة نقاط مرجعيّة في قاعدة البيانات داخل البرنامج، وتدعى "webs". يختار البرنامج إحدى الصفحات التي لم تُكتشف بعد عشوائيًّا ليكتشفها تاليًا.

إذا أردت عرض محتويات ملفّ spider.sqlite، يمكنك تشغيل spdump.py.

```
(5, None, 1.0, 3, 'http://www.dr-chuck.com/csev-blog')
(3, None, 1.0, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, None, 1.0, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, None, 1.0, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

يعرض المثال السابق عدد الروابط الجديدة المضافة، والترتيب القديم، ثمّ الجديد للصفحة، ورمز تعريف الصفحة، ورابطها بالترتيب. يعرض برنامج spdump.py فقط الصفحات التي تملك على الأقلّ رابط يشير لها.

بمجرّد توفّر عدّة صفحات في قاعدة البيانات، يمكنك تنفيذ عمليّة الترتيب لهذه الصفحات باستخدام برنامج sprank.py، حيث يمكنك إخبار البرنامج بعدد مرّات تكرار خوارزميّة ترتيب الصفحات.

```
How many iterations:2
1 0.546848992536
2 0.226714939664
[(1, 0.559), (2, 0.659), (3, 0.985), (4, 2.135), (5, 0.659)]
```

يمكنك عرض محتوى قاعدة البيانات بتنفيذ pdump.py لترى أنّ قيمة ترتيب الصفحة قد حُدّثت لجميع المواقع:

```
(5, 1.0, 0.985, 3, 'http://www.dr-chuck.com/csev-blog')
(3, 1.0, 2.135, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, 1.0, 0.659, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, 1.0, 0.659, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

يمكنك تشغيل البرنامج sprank.py بقدر ما تشاء، وسيظهر لك في كلّ مرّة ترتيب الصفحات. أو يمكنك تشغيله بضع مرّات، ثمّ إضافة صفحات جديدة بتشغيل البرنامج spider.py، وثمّ تشغيل sprank.py، والاطّلاع على الترتيب الجديد.

تقوم محركات البحث بهاتين العمليّتين (تعقّب صفحات جديدة وترتيبها) طوال الوقت.

إذا أردت ترتيب الصفحات من جديد، شغّل برنامج spreset.py، ثمّ أعد تشغيل sprank.py.

```
How many iterations:50
1 0.546848992536
2 0.226714939664
3 0.0659516187242
4 0.0244199333
5 0.0102096489546
6 0.00610244329379
...
```

```
42 0.000109076928206
43 9.91987599002e-05
44 9.02151706798e-05
45 8.20451504471e-05
46 7.46150183837e-05
47 6.7857770908e-05
48 6.17124694224e-05
49 5.61236959327e-05
50 5.10410499467e-05


[(512, 0.0296), (1, 12.79), (2, 28.93), (3, 6.808), (4, 13.46)]
```

يُعرض التغيّر الوسطيّ الحاصل في ترتيب الصفحات عند كلّ تكرار لخوارزميّة ترتيب الصفحات. في البداية، تكون الشبكة غير متوازنة، لذا تتباين قيم ترتيب الصفحات مع تكرار الخوارزميّة، وعندما نصل إلى عدد معيّن من التكرارات للخوارزميّة، تصبح تلك القيم متقاربة أكثر، لذا عليك تشغيل برنامج sprank.py عددًا كافيًا من المرّات حتّى تتقارب قيم ترتيب الصفحات.

إذا أردت عرض ترتيب الصفحات الحاليّ للمواقع التي تعاملنا معها أعلاه، شغّل البرنامج spjson.py لقراءة قاعدة البيانات وتخزين البيانات الخاصة بالصفحات الأكثر ارتباطًا وفق صيغة JSON؛ لتُعرض في المتصفّح.

```
Creating JSON output on spider.json...
How many nodes? 30
Open force.html in a browser to view the visualization
```

بإمكانك عرض هذه البيانات بفتح ملفّ "force.html" في المتصفّح؛ لتشاهد الروابط والعقد مولّدة تلقائيًا، حيث يمكنك السحب والنقر على أيّة عقدة، كما يمكنك أيضًا إيجاد الرابط الذي تمثّله العقدة عبر النقر المزدوج عليها.

إذا أعدت تشغيل البرامج المساعدة الأخرى، مثل برنامج spjson.py، وحدّثت الصفحة في المتصفّح، ستحصل على بيانات جديدة من ملفّ spider.json.

3.16 تحليل وعرض البيانات الواردة في البريد الإلكترونيّ

بعد أن أصبح التعامل مع الملفّين "mbox-short.txt" و"mbox.txt" مألوفًا بالنسبة لنا، حان الوقت لتحسين مهارتنا في تحليل البيانات الواردة في البريد الإلكترونيّ.

في التطبيقات العمليّة، قد تحتاج أحيانًا إلى سحب البيانات من الخوادم. قد تكون هذه البيانات مليئة بالأخطاء، وغير متناسقة، وتتطلّب الكثير من التعديل والترتيب، لذا قد تتطلّب العمليّة وقتًا طويلًا.

سنتطرّق في هذا القسم إلى أعقد تطبيق تعاملنا معه إلى حدّ الآن، حيث سوف نسحب 1 جيجابايت من البيانات لتحليلها واستعراضها.

يمكنك تحميل التطبيق من خلال الرابط التالي: https://www.py4e.com/code3/gmane.zip

سوف نستخدم خدمة مجّانيّة لأرشفة قوائم البريد الإلكترونيّ للحصول على البيانات، تدعى http://www.gmane.org، حيث تعدّ هذه الخدمة شائعة الاستخدام في التطبيقات مفتوحة المصدر؛ إذ تؤمّن أرشيف يمكنك البحث فيه عن نشاط بريدهم الإلكترونيّ. علمًا أنّه لا يوجد حدود للتحميل، إذ يمكنك تحميل قدر ما تشاء من البيانات، لكن يفضّل ألّا تسرف وتسبّب ضغطًا على

الشكل 29: مجموعة كلمات موزّعة على شكل غيمة مولّدة من قائمة شركة Sakai

خوادمهم. يمكنك الاطّلاع على شروط الاستخدام من خلال زيارة الصفحة التالية: http://www.gmane.org/export.php

من المهمّ استخدام هذه الخدمة بمسؤوليّة، وذلك من خلال إضافة تأخير زمنيّ لطلبات الوصول للخوادم، وتوزيع المهام التي تحتاج إلى وقت معالجة طويل على أطول فترة ممكنة. لذلك، احرص على عدم إساءة استخدام هذه الخدمة.

لدى تعقُّب بيانات البريد الإلكترونيّ لـ Sakai باستخدام هذا البرنامج، أُنتِج قرابة 1 غيغابايت من البيانات، واستغرق ذلك عمليّات عديدة على مدار عدّة أيّام.

يحتوي الملفّ README.txt في المجلّد المضغوط أعلاه على إرشادات حول كيفيّة تحميل نسخة من ملفّ content.sqlite مجموعة من رسائل البريد الإلكترونيّ لSakai الجاهزة، حيث لا تضطرّ بذلك إلى التعقّب لمدّة خمسة أيّام متواصلة من أجل تشغيل البرامج. لكن، حتّى وإن حمّلت المحتوى الجاهز مسبقًا، فلا يزال عليك إجراء عمليّة تعقّب لمتابعة أحدث الرسائل.

تتجلّى الخطوة الأولى في تعقّب أرشيف gmane، حيث إنّ عنوان URL الأساسيّ مضاف بشكل مباشر في شيفرة gmane.py، وفي قائمة مطوّري Sakai. يمكنك تعقّب أرشيف آخر عن طريق تغيير عنوان URL الأساسيّ. تأكّد من حذف ملفّ content.sqlite عند تبديل عنوان URL الأساسيّ.

يعمل ملفّ gmane.py بطريقة مسؤولة، حيث يعمل ببطء، ويستردّ رسالة بريد إلكترونيّ واحدة في الثانية، وذلك كي لا يعلَق في أرشيف gmane. كما يخزّن جميع بياناته في قاعدة بيانات تتيح المقاطعة وإعادة التشغيل كلّما دعت الحاجة. قد يستغرق تحميل جميع البيانات عدّة ساعات، لذلك قد تحتاج إلى إعادة التشغيل عدّة مرّات.

فيما يأتي ناتج عمليّة تشغيل ملف gmane.py، حيث يستردّ الرسائل الخمسة الأخيرة من قائمة مطوّري Sakai:

```
How many messages:10
http://download.gmane.org/gmane.comp.cms.sakai.devel/51410/51411 9460
nealcaidin@sakaifoundation.org 2013-04-05 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51411/51412 3379
samuelgutierrezjimenez@gmail.com 2013-04-06 re: [building ...
```

```
http://download.gmane.org/gmane.comp.cms.sakai.devel/51412/51413 9903
da1@vt.edu 2013-04-05 [building sakai] melete 2.9 oracle ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51413/51414 349265
m.shedid@elraed-it.com 2013-04-07 [building sakai] ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51414/51415 3481
samuelgutierrezjimenez@gmail.com 2013-04-07 re: ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51415/51416 0

Does not start with From
```

يتصفّح البرنامج قاعدة البيانات content.sqlite من البداية حتّى رقم الرسالة الأولى التي لم يتمّ تعقّبها مسبقًا، ويبدأ بعد ذلك في تعقّب تلك الرسالة، إذ يستمرّ في التعقّب حتّى يصل إلى العدد المطلوب من الرسائل، أو يصل إلى صفحة لا تبدو أنّها رسالة منسّقة بشكل صحيح.

أحيانًا تُفقد رسالة في gmane.org، ويعود السبب في ذلك إلى إمكانيّة حذفها من قبل المسؤولين، أو احتماليّة ضياع إحدى الرسائل. إذا توقّف المتعقِّب بشكل يوحي أنّه وصل إلى رسالة مفقودة، عندها انتقلْ إلى SQLite Manager، وأضف صفًّا مع كتابة رقم المعرّف المفقود، مع إبقاء الحقول الأخرى فارغة، ثمّ أعد تشغيل gmane.py. سيؤدّي هذا إلى تحرير ملفّ التعقّب؛ ممّا يتيح له استمراريّة التعقّب دون أن يعلق عند الرسالة المفقودة. أمّا الرسائل الفارغة، فسيجري تجاهلها في المرحلة التالية من العمليّة.

يمكنك تشغيل gmane.py مرّة أخرى للحصول على رسائل جديدة عند إرسالها إلى القائمة، وذلك بمجرّد أن تتعقّب جميع الرسائل وتضعها في content.sqlite، حيث تعدّ هذه الخاصيّة من الخصائص المفيدة.

تعدّ بيانات content.sqlite بيانات أوّليّة، حيث يعدّ نموذج بياناتها غير فعّال، كما أنّها غير مضغوطة. هذا متعمّد، لأنّه يسمح لك بالاطّلاع على content.sqlite في SQLite Manager لتصحيح مشاكل عمليّة التعقّب. من غير المحبّذ إجراء طلبات على قاعدة البيانات هذه، لأنّ العمليّة ستكون بطيئة للغاية.

أمّا الخطوة الثانية، فهي تشغيل برنامج gmodel.py، إذ يقرأ هذا البرنامج البيانات الأوّليّة من content.sqlite، وينتج نسخة منظّمة ومنسّقة من البيانات في ملفّ index.sqlite. سيكون هذا الملفّ أصغر بكثير (غالبًا ما يكون أصغر بعشر مرّات) من content.sqlite؛ لأنّه يضغط كلًّا من الترويسة والنصّ الأساسيّ.

في كلّ مرّة تشغيل لـgmodel.py، يحذف index.sqlite ويعيد تكوينه، ممّا يتيح إمكانيّة ضبط معاملاته، وتحرير جداول الربط في content.sqlite لتعديل عمليّة تنظيم البيانات. فيما يلي ناتج تشغيل برنامج gmodel.py، حيث يعرض سطر في كلّ مرّة تعالَج 250 رسالة بريد، لتتمكّن من ملاحظة التغيير. وبعد فترة من عمل البرنامج، يكون قد عالج قرابة 1 جيجابايت من بيانات البريد.

```
Loaded allsenders 1588 and mapping 28 dns mapping 1
1 2005-12-08T23:34:30-06:00 ggolden22@mac.com
251 2005-12-22T10:03:20-08:00 tpamsler@ucdavis.edu
501 2006-01-12T11:17:34-05:00 lance@indiana.edu
751 2006-01-24T11:13:28-08:00 vrajgopalan@ucmerced.edu
...
```

يتحمّل برنامج gmodel.py عبء معالجة عدد من مهمّات تنظيم البيانات. على سبيل المثال: اقتطاع أسماء النطاقات إلى مستويين أو إلى ثلاثة مستويات. على سبيل المثال، يصبح si.umich.edu بالشكل umich.edu ، ويصبح caret.am.ac.uk بالشكل cam.ac.uk، وتُحوَّل عناوين البريد الإلكترونيّ أيضًا إلى حالة الأحرف الصغيرة، كما تُحوَّل بعض العناوين التي تنتهي بـ gmane.org@، مثل العناوين الآتية:

arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.org،

إلى العنوان الحقيقيّ كلّما عثِرَ على عنوان بريد إلكترونيّ حقيقيّ مطابق في مكان آخر ضمن الرسالة.

يوجد في قاعدة البيانات mapping.sqlite جدولان يسمحان لك بربط أسماء النطاقات وعناوين البريد الإلكترونيّ الفرديّة التي تتغيّر خلال مدّة توفّر قائمة البريد الإلكترونيّ. على سبيل المثال، استخدم ستيف غيثينس Steve Githens عناوين البريد الإلكترونيّ أدناه مع تغيّر عمله:

s-githens@northwestern.edu

sgithens@cam.ac.uk

swgithen@mtu.edu

يمكننا إضافة اثنين من المدخلات إلى جدول الربط في mapping.sqlite حتّى يربط gmodel.py الإيميلات الثلاثة بعنوان واحد:

s-githens@northwestern.edu -> swgithen@mtu.edu

sgithens@cam.ac.uk -> swgithen@mtu.edu

يمكنك أيضًا إضافة مدخلات مماثلة في جدول DNSMapping إذا كان هناك العديد من أسماء DNS التي تريد ربطها إلى DNS واحد. على سبيل المثال، أُضيف الربط التالي إلى بيانات Sakai:

iupui.edu -> indiana.edu

وبذلك، تكون جميع الحسابات من جميع أنحاء حرم جامعة إنديانا قد تمّ تعقّبها.

يمكنك إعادة تشغيل gmodel.py مرارًا وتكرارًا، وإضافة عمليّات ربط لجعل البيانات أكثر تنظيمًا ودقّة. وعند الانتهاء، ستكون لديك نسخة منظّمة من البريد الإلكترونيّ في index.sqlite. يؤمّن هذا الملفّ آليّة سريعة لتحليل البيانات.

إنّ أوّل وأبسط تحليل للبيانات هو تحديد "من الذي أرسل أكبر عدد من الرسائل؟"، و"ما هي المنظّمة التي أرسلت أكبر عدد من رسائل البريد؟". يتمّ ذلك باستخدام gbasic.py:

```
How many to dump? 5
Loaded messages= 51330 subjects= 25033 senders= 1584


Top 5 Email list participants
steve.swinsburg@gmail.com 2657
azeckoski@unicon.net 1742
ieb@tfd.co.uk 1591
```

```
csev@umich.edu 1304
david.horwitz@uct.ac.za 1184

Top 5 Email list organizations
gmail.com 7339
umich.edu 6243
uct.ac.za 2451
indiana.edu 2258
unicon.net 2055
```

لاحظ مدى سرعة تشغيل gbasic.py مقارنةً بـgmane.py أو حتّى gmodel.py. يعملون جميعًا على البيانات ذاتها، لكن gbasic.py هو الأسرع، لأنّه يستخدم البيانات المضغوطة والمنظّمة في index.sqlite. إذا كان لديك الكثير من البيانات لإدارتها، فقد تتطلّب العمليّة الموجودة في هذا التطبيق وقتًا أطول للتطوير، نظرًا إلى أنّها عمليّة متعدّدة الخطوات، ولكنّها ستوفّر لك الكثير من الوقت عندما تبدأ فعليًّا في عمليّة استكشاف وعرض البيانات.

يمكنك إجراء تمثيل بسيط للبيانات الخاصّة بتكرار الكلمات في سطور الموضوع بتشغيل الملفّ gword.py:

```
Range of counts: 33229 129
Output written to gword.js
```

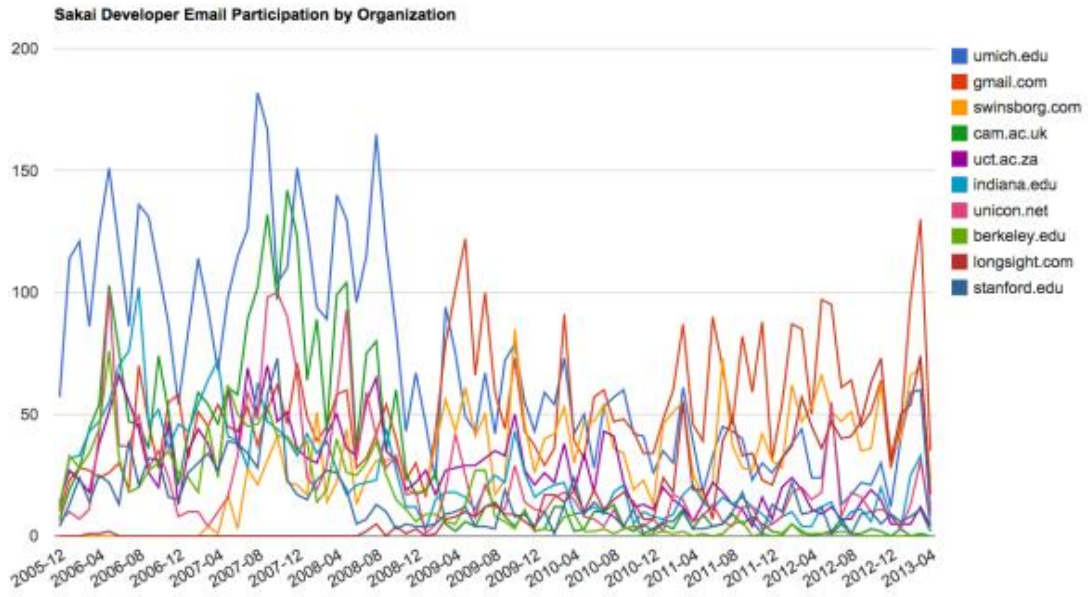
ينتج عن هذا التمثيل الملفّ gword.js الذي يمكنك عرضه باستخدام gword.htm لإنتاج مجموعة كلمات (ذات أشكال وأحجام مختلفة) مشابهة لتلك الموجودة في (الشكل-3) في بداية هذا القسم. ينتج التمثيل الثاني عند تشغيل gline.py؛ إذ يحسب عدد الإيميلات تبعًا للمنظّمة:

```
Loaded messages= 51330 subjects= 25033 senders= 1584
Top 10 Oranizations
['gmail.com', 'umich.edu', 'uct.ac.za', 'indiana.edu',
```

```
'unicon.net', 'tfd.co.uk', 'berkeley.edu', 'longsight.com',
'stanford.edu', 'ox.ac.uk']
Output written to gline.js
```

تكتب مخرجاته في gline.js التي تعرض باستخدام gline.htm.

يعدّ هذا التطبيق من التطبيقات المعقَّدة، والمتطوّرة نسبيًّا، وله ميّزات لإنجاز بعض عمليّات استرداد البيانات الحقيقيّة، وتنظيمها، وتمثيلها.



الشكل 30 : توزّع الإيميلات بالنسبة للمنظّمة