قرة العيون
في تبسيط لغة بايثون

احمد سالم الصاعدي

مقدمة

الحمد لله الذي تتم بفضله الصالحات واصلي وأسلم على خير البشر نبينا محمد صلى الله عليه وسلم وعلى آله وصحبه الطيبين الطاهرين. اما بعد :

فلقد لاحظت خلال فترة تعلمي للغة بايثون افتقار المكتبة العربية الى كتاب مكتمل يشرح مبادئ هذه اللغة بشكل منظم وبسيط. فعقدت العزم على تأليف هذه الكتاب وذلك لما رأيته من الأهمية بمكان ان يتعلم القارئ العربي هذه اللغة والتي أصبحت اللغة البرمجية المحبوبة لدى الكثير من العلماء والباحثين والمهندسين. فمعظم جامعات العالم اليوم أصبحت تدرسها لطلابها لأنها لغة برمجية سهلة وقوية في نفس الوقت ويمكن استخدامها في مجالات عدة. لذلك اردت ان يكون هذا الكتاب لبنة أولى للمساهمة في تعليم هذه اللغة وحافزا الى تدريسها في مدارسنا الحكومية في مراحل مبكرة كالمتوسطة والثانوية مثلا وذلك لان الأجيال الحالية لديها شغف غير مسبوق للتعرف على كل ما هو جديد في عالم التقنية وخصوصا الكمبيوترية منها. فكل ما يحتاجه هذا الجيل هو استخدام طريقة سهلة وشيقة تعرفهم بهذه اللغة وتعزز شغفهم بالتقنية فيصبحوا قادرين على تعلمها والاستفادة منها دون مشقة او عناء. وبما ان هذا العمل بشري المصدر فانه لا يصل الى درجة الكمال لذلك ارجو ممن سنحت له الفرصة لقراءة هذا الكتاب ان يساهم في تحسين محتواه بإرسال ملاحظاته الى ايميل المؤلف ahmad.alsaadi@uj.edu.sa الذي يعدكم على اخذها في عين الاعتبار متى ما سنحت الفرصة لإصدار طبعة جديدة لهذا الكتاب.

المحتويات

الفصل الاول

نبذة عن لغة بايثون

تعريف لغة بايثون

بايثون (python) لغة برمجية مفتوحة المصدر سهلة التعلم يمكن الاعتماد عليها في كتابة الكثير من التطبيقات البرمجية القوية. وأكبر دليل على ذلك هو استخدام وكالة الارصاد الامريكية ناسا وشركتا قوقل وياهو وغيرها من الشركات الكبرى لهذه اللغة في بناء برامجهم المختلفة.

نشأة لغة بايثون

كانت بدايات نشأة هذه اللغة في هولندا على يد شخص يدعي جويدو فان روزم (Guido van Rossum) في نهاية الثمانيات الميلادية من القرن العشرين. حيث تم الاعلان عنها في عام ١٩٩١م. كما يعتبر فتح مصدر هذه اللغة من اهم الاسباب التي ادت الى زيادة شهرتها من خلال تكوين مجتمع برمجي نشط حولها أسهم في انشاء مكتبات كثيرة سهلت على المطورين الاخرين بناء تطبيقاتهم بسرعة وسهوله فائقة مقارنة باللغات البرمجية الأخرى.

مزايا لغة بايثون

للغة بايثون مزايا عدة جعلت منها اللغة المفضلة الاولى لدى كثير من المبرمجين ومن بين اهم هذه المزايا نذكر:

سهولة تراكيبها اللغوية: فأكوادها البرمجية تكتب بطريقة قريبة جدا من اللغة الانجليزية. لذلك نجدها لا تشكل أي عائق أمام أي مبرمج ان يفهم الأكواد المكتوبة من قبل مبرمجين اخرين عندما يستدعي الامر صيانة تلك الأكواد او تحديثها.

المرونة: يمكن تشغيل وتطوير البرامج المكتوبة بلغة بايثون على معظم انظمة التشغيل المعروفة. فالأكواد التي تم تطويرها على نظام ويندوز يمكن تشغيلها على نظام ماك ولينكس والعكس صحيح دون الحاجة الى اعادة بناء الأكواد (compiling).

كثرة المكتبات : يعتبر توفر المكتبات من اهم المزايا التي تقدمها اللغة للمبرمجين لتزيد من فعاليتهم في بناء التطبيقات. لذلك عند تنصيب اصدارة بايثون نجد انها تحتوي على مكتبات قياسية كثيرة بعضها يعتبر جزء لا يتجزأ من تراكيب اللغة كمكتبة الارقام والقوائم وبعضها الاخر يعمل على تسهيل التعامل مع انظمة التشغيل اما

الجزء الاكبر من هذه المكتبات فهو اختياري يتم استيراده متى ما دعت الحاجة لذلك. كما ان هناك مكتبات اخري تحتاج الى تنصيب قبل ان يتمكن المبرمج من استيرادها واستخدامها في برنامجه. وهذه المكتبات مجانية ويمكن تحميلها وتنصيبها اما من الموقع الخاص بالمطورين لهذه المكتبة او من موقع http://pypi.python.org والذي يحتوي حتى وقت كتابة هذا الكتاب على 69478 مكتبة مجانية جاهزة يمكن استخدامها في بناء التطبيقات المختلقة.

التكامل مع لغات برمجية أخرى: يمكن استخدام بايثون كلغة مساندة تمكن المستخدم لبرنامج مكتوب بلغة سي (C) او سي بلس بلس (C++) مثلا من زيادة او تعديل خصائص ذلك البرنامج ليتناسب مع احتياج المستخدم. ومن أقرب الامثلة على ذلك هو استخدام لغة بايثون في برنامج فري كاد (FreeCAD) كلغة برمجة نصية للتحكم بكافة خصائص البرنامج ووظائفه.

اصدارات لغة بايثون

هناك اصدارتان للغة بايثون. الإصدارة الأولى تعرف ببايثون 2 وهي الاقدم والاصدارة الاخرى تدعى بايثون 3 وهي الاحدث. تم اصدار بايثون 3 في عام 2008 لحل بعض المشاكل الجوهرية التي بنيت عليها الاصدارة 2. عدم توفر مكتبات تدعم الاصدارة 3 في بدايات الاصدار ادى الى تمسك المجتمع البرمجي بالإصدارة 2. لكن هذا العزوف لم يدم طويلا فالمكتبات المتوفرة للإصدارة 2 اصحبت متوفرة للإصدارة رقم 3 بنسبة 88% تقريبا. لذلك أصبح المجتمع البرمجي مجمعا على استخدام الاصدارة 3 لكل من أراد ان يبدأ في تعلم لغة بايثون خصوصا وان فريق التطوير للإصدارة 2 قد توقف عن عمل تحديثات لهذه الاصدارة مع حلول عام 2018. وأصبحت الاصدارة 2 جزء من التاريخ.

توزيعات بايثون

بالإضافة الى التوزيعة الرسمية التي يمكن تحميلها من ***www.python.org*** هناك توزيعات اخري تأتي محملة بمكتبات ومدير ادارة مكتبات تهدف الى اراحة المستخدم من عناء تحميل المكتبات واضافتها للإصدارة الرسمية. معظم هذه التوزيعات تأتي على شكل اصدارة مجانية واصدارة تجارية. ومن بين هذه التوزيعات ما يلي :

توزيعة Anaconda : يمكن تحميل هذه التوزيعة من ***www.continuum.io*** ويمكن تنصيب هذه التوزيعة على ويندوز وماك ولينكس. تحتوي التوزيعة على اكثر من 100 مكتبة و مدير ادارة مكتبات يدعى (***conda***). كما يوجد لهذه التوزيعة اصدارة مصغرة تدعى (***miniconda***) واصدارات تجارية اخرى.

توزيعة Enthought Canopy : يمكن تحميل هذه التوزيعة من ***www.enthought.com/products/canopy***.
تأتي هذه التوزيعة بإصدارة تجارية واصدارة مجانية لمدة سنه قابلة للتجديد للاكاديميين والطلاب و لكن بعد اجراء عملية التسجيل. تحتوي الاصدارة على مفسر بايثون 2 وأكثر من 450 مكتبة متخصصة للأغراض العلمية والتحليلية. كما ان عملية التسجيل تسمح للمستخدم بالاطلاع على الفيديوهات التعليمية المعمولة من قبل ***Enthought***.

توزيعة ActivePython :

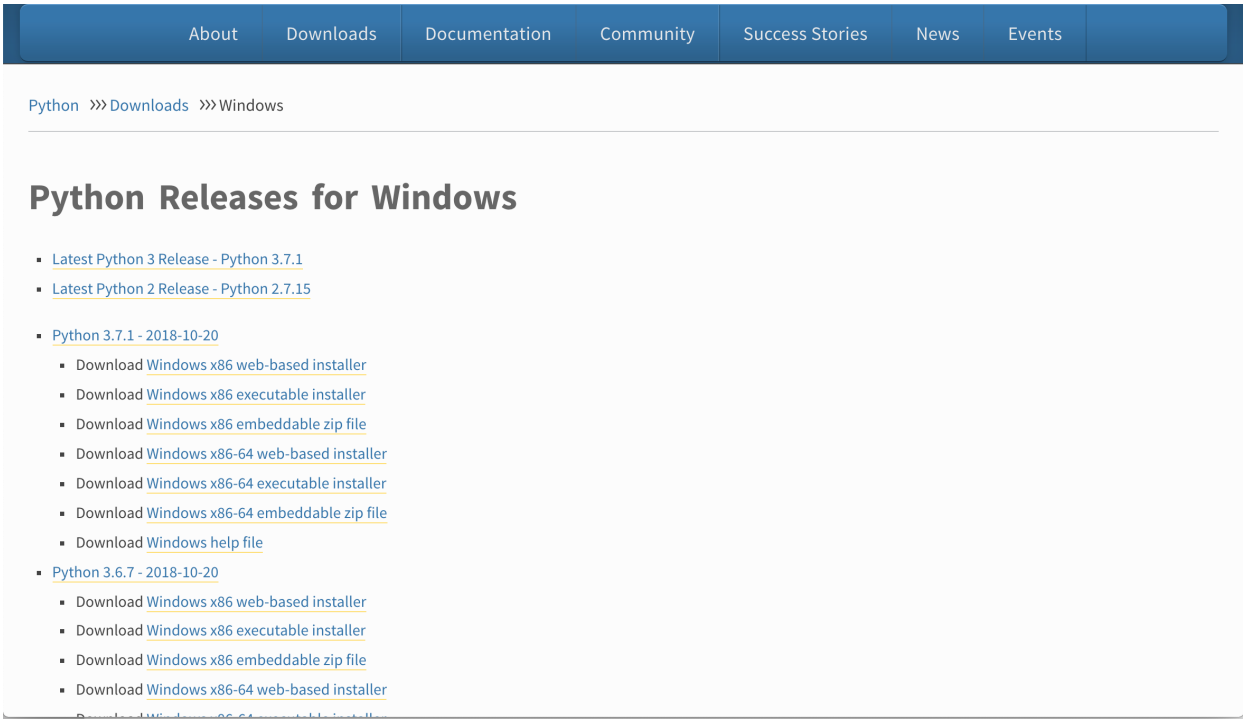
تنصيب بايثون

بما ان ميكروسوفت ويندوز مازال أكثر أنظمة التشغيل شيوعا وخاصة في الوطن العربي فإننا سنبدأ بشرح طريقة تنصيب مفسر لغة بايثون على هذا النظام اولا.

تنصيب بايثون على نظام ويندوز:

كما ذكرنا سابقا فان بايثون يأتي بتوزيعة رسمية من موقع ***www.python.org*** وتوزيعات أخرى تحتوي على بعض الاضافات مثل ***anaconda***, ***canopy*** وغيرها من التوزيعات وفي هذا المقام فإننا سوف نقوم بشرح طريقة تنصيب مفسر بايثون 3 من الموقع الرسمي وكذلك توزيعة ***anaconda*** لشهرتها الواسعة بين العلماء والباحثين. لذلك إذا كنت تستخدم اي من انظمة ويندوز سواء القديم منها او الجديد فإنك تحتاج الى تنصيب مفسر لغة بايثون على هذا النظام. واليك الخطوات التالية التي تساعدك على عمل ذلك:

اذهب الى الموقع التالي http://www.python.org/downloads :
اضغط على الرابط "***windows***" للانتقال الي خيرات التحميل الخاصة بنظام ويندوز.



هناك ثلاث خيرات لتنصيب أحدث اصدارة من مفسر بايثون 3 على ويندوز. أذا كان نظام الويندوز لديك 32 بت اضغط على الرابط Windows x86 executable installer لتحميل مفسر بايثون والتنصيب عن طريق ملف تنفيذي كما هو معلم في الصورة السابقة.

أما اذا كان نظام الويندوز 64 بت فقم بالضغط على الرابط Windows x86-64 executable installer

قم بتشغيل الملف الذي قمت بتحميله لبدأ عملية التنصيب.

ضع علامة صح على الخيار "*Add Python 3.7 to PATH*" ثم اضغط على"*Install Now*"

قد تظهر لك ملاحظة كما في الصورة التالية تسألك عن رغبتك في الاستمرار في عملية التنصيب فاضغط على "*Yes*". تستغرق عملية التنصيب قرابة الدقيقة فكن صبورا.

وعند اكتمال التنصيب سوف تظهر لك ملاحظة ان عملية التنصيب تمت بنجاح عندها اضغط على "*Close*".

لتشغيل مفسر بايثون هناك طريقتان فبحسب نظام ويندوز 10 مثلا يمكنك الذهاب الى قائمة ابدأ ثم الضغط على كافة البرامج كما في الصورة التالية :

وسوف تجد ان عملية التنصيب اضافت اربعة ملفات داخل قائمة البرامج في مجلد بايثون. اول هذه الملفات يمكنك من تشغيل مفسر بايثون باستخدام "IDEL" وهو بيئة تطوير خاصة ببايثون كما هو موضح بالشكل التالي :

فعند الضغط على هذا الخيار تظهر لك نافذة "***IDEL***" التفاعلية كما هو موضح بالشكل التالي :

وعند كتابة اي امر بعد العلامات الثلاث "*>>>*" والضغط على زر الادخال تظهر لك النتائج مباشرة في نفس النافذة لذلك سميت بالنافذة التفاعلية.

تمرين ١

اذا كنت تعمل على نظام ويندوز فقم باتباع الخطوات السابقة لتنصيب مفسر بايثون 3 ثم قم بتشغيل مفسر بايثون من خلال "***IDEL***" وجرب كتابة الامر التالي *2+4* ولاحظ النتيجة بعد الضغط على زر الادخال؟

اما الطريقة الثانية لتشغيل مفسر بايثون فتتم بالضغط على الملف الثاني من مجلد بايثون في قائمة البرامج كما يظهر في الشكل التالي :

فعند الضغط على هذا الخيار يظهر لك مفسر بايثون 3 في محرر الاوامر الخاص بنظام ويندوز كما في الشكل التالي :

تمرين ٢

قم بتشغيل مفسر بايثون 3 داخل محرر اوامر ويندوز و جرب كتابة الامر *2+4* لترى ناتج الجمع بعد الضغط على زر الادخال؟

تنصيب بايثون على نظام ماك ولينكس:

يأتي نظاما تشغيل ماك ولينكس وقد نصب عليهما مفسر بايثون ذو الإصدارة رقم 2. ويمكنك التأكد من ذلك باتباع الخطوات التالية:

- قم بتشغيل محرر الاوامر في الماك او لينكس والذي يدعى "***Terminal***"

اكتب في محرر الاوامر الامر ***python*** بعد رمز الدولار "*$*" واضغط على زر الادخال .

سوف تجد ان مفسر بايثون قد ظهر في محرر الاوامر ومن خلال البيانات التي كتبت عن اصدارة المفسر سوف تجدها الاصدارة الثانية كما في الشكل التالي:

لتنصيب الاصدارة 3 على ماك اتبع الخطوات التالية:

- اذهب الى الموقع التال ***www.python.org/downloads***.

اضغط على الرابط "***Mac OSX***" للانتقال الي خيرات التحميل الخاصة بنظام ماك

الفصل الثاني

المبادئ الأساسية للغة بايثون

بعد ان تأكدنا من أن نظام التشغيل الذي نعمل عليه يحتوي على احدى اصدارات لغة بايثون 3 يمكننا الان ان نبدأ رحلة التعلم والتي اتمنى ان تكون حافلة بالمتعة والفائدة .

أهداف الباب

عند اتمام هذا الباب يجب ان يكون لديك المام بعدة مبادئ اساسية عن لغة بايثون والتي من اهمها :

- طريقة كتابة المتغيرات
- طريقة تدوين الملاحظات على الكود البرمجي
- كيفية طباعة نص على شاشة الكمبيوتر
- الفرق بين استخدام الاحرف الصغيرة والكبيرة في كتابة الكود البرمجي.
- اجراء العمليات الحسابية الاساسية
- انواع البيانات الاساسية في لغة بايثون.
- القاعدة التي تحكم عدد المسافات الفارغة المتروكة قبل بداية كل سطر برمجي

المتغيرات

المتغيرات هي اسماء تستخدم للدلالة على قيم بيانات موجودة في ذاكرة الكمبيوتر. واستخدام المتغيرات في كتابة الأكواد البرمجية ذو اهمية قصوى بحيث لا يكاد يخلو برنامج من وجود متغير واحد او أكثر وذلك لأنها تسهل على المبرمج تذكر البيانات بأسماء يسهل حفظها بدلا من استخدام قيم البيانات ذاتها. لإسناد قيمة الى متغير فإننا نختار اسماً مناسب للمتغير وليكن x ومن ثم نضع علامة يساوي (=) واخيراً نضع القيمة المراد اسنادها للمتغير كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x=5
>>>
```

تمرين ١

قم بتشغيل مفسر بايثون التفاعلي كما تعلمت في الفصل السابق واسند القيمة 5 للمتغير x ثم اضغط زر الادخال.

في المثال السابق علامة (=) تسمى معامل الاسناد. لذلك يصبح الان بمقدورنا استخدام x عوضا عن القيمة 5 في اي عملية حسابية كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x=5
>>> x+4
9
>>>
```

لاحظ انه عند اسناد قيمة الى متغير فان مفسر بايثون لا يطلب منك تحديد نوع البيانات المدخلة وذلك لأنه يقوم بتحديد نوع البيانات بشكل تلقائي بمجرد إدخالها لذلك يطلق على لغة بايثون بانها ديناميكية. هذه الخاصية غير متوفرة في بعض لغات البرمجة الأخرى كلغة C مثلا التي تجبر المبرمج على تحديد نوع البيانات عند القيام بعمليات الاسناد كما في المثال التالي:

```
كتابة المتغيرات بلغة C تتطلب من المبرمج تحديد نوع البيان المطلوب اسناده
للمتغير حيث تمثل int هنا نوع البيانات وهو عدد صحيح
int x=5;
```

تمرين

١-شغل محرر بايثون التفاعلي واسند القيمة 5 للمتغير x. في سطر ثاني استخدم الدالة type(x) للتعرف على نوع البيان المسند للمتغير x؟

٢-كرر العملية مع العدد 5.0 ماذا تلاحظ؟

٣-كرر العملية مرة أخرى و اسند نص بين علامة تنصيص للمتغير x مثلا (x="how are you")؟ ماذا تلاحظ؟

٤-حاول ان تستخدم محرك البحث قوقل للتعرف على وظيفة الدالة type() في بايثون وماهي أنواع البيانات التي استخدمناها في التمرين؟

كما يجب التنويه الى ان قيمة x قابلة للتغيير. فعند اسناد القيمة 10 الى المتغير x فان x الآن تصبح تشير للقيمة 10 عوضا عن القيمة 5 كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x=5
>>> x+4
9
>>> x=10
>>> x+4
14
```

```
>>>
```

تمرين ٢

قم بتشغيل مفسر بايثون التفاعلي واسند القيمة 100 للمتغير n. في سطر ثاني قم بتغير قيمة n الى القيمة 50. في سطر ثالث حاول ان تعرف القيمة النهائية للمتغير n ؟

كما تمكن لغة بايثون من اجراء عملية اسناد متعددة في سطر برمجي واحد كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a,b,c=2,4,6
>>> a
2
>>> b
4
>>> c
6
>>>
```

ان لغة بايثون تتطلب التقيد بقواعد اساسية عند كتابة أسماء المتغيرات وتتلخص هذه القواعد فيما يلي :

أولاً: المتغيرات يجب ان تبدأ بحرف انجليزي او شرطة سفلية. عدا ذلك فان مفسر بايثون يعطي رسالة بوجود خطأ لغوي (Syntax Error). فمثلا إذا حاولنا كتابة متغير اسمه 2y وقمنا بإسناد القيمة 4 اليه فان محرر بايثون سوف يمنعنا من القيام بذلك ويعطينا رسالة تفيد بوجود خطأ في التركيب اللغوي كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2y=4
  File "<stdin>", line 1
    2y=4
     ^
SyntaxError: invalid syntax
>>>
```

ثالثاً: أسماء المتغيرات يمكن ان تكون حرف او كلمة او مجموعة كلمات بدون ترك مسافة بين الكلمات او تكون هذه الكلمات مربوطة بشرطة سفلية كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> carNumber=1234      اسم المتغير هنا مكتوب بطريقة صحيحة
>>> car_number=12345    اسم المتغير هنا مكتوب بطريقة صحيحة
```

```
عند تكوين اسم متغير من كلمتين مفصولة بمسافة سوف يرفض مفسر بايثون العملية
ويشير الي وجود خطأ في التركيب اللغوي كما يلي:
>>> car number=3456
  File "<stdin>", line 1
    car number=3456
             ^
SyntaxError: invalid syntax
>>>
```

رابعاً: يمكن استخدام الارقام في كتابة أسماء المتغيرات ولكن لا يمكن استخدامها في بداية اسم المتغير كما وضحنا في مثال سابق. وهنا مثال اخر للتوضيح أكثر:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x1=5        هنا اسم المتغير صحيح
>>> x2y=10      هنا اسم المتغير أيضا صحيح
عندما يكون اسم المتغير مبدوء برقم فان مفسر بايثون يرفض العلمية ويشير الي وجود
                                        خطأ في التركيب اللغوي كما في المثال التالي:
>>> 1x=8
  File "<stdin>", line 1
    1x=8
     ^
SyntaxError: invalid syntax
>>>
```

خامساً: لا يمكن استخدام اي رمز في كتابة المتغيرات عدا الاحرف والأرقام الانجليزية والشرطة السفلية. وسوف ينبهك محرر بايثون الى وجود خطأ لغوي (Syntax Error) عند استخدام رمز اخر كالدولار ($) مثلا في اسم المتغير كما انه سوف يشير الى مكان الخطأ بالعلامة (^) تحت رمز الدولار كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> $x=5
  File "<stdin>", line 1
    $x=5
    ^
SyntaxError: invalid syntax
>>>
```

يجب ملاحظة ان استخدام الرموز المحظورة لا يقتصر على بداية اسم المتغير فقط بل ان جود الرمز في أي موضع سواء في البداية او الوسط والنهاية من اسم المتغير سوف يعطي خطأ لغوي كما وضحنا في المثال السابق.

تمرين ٣

حاول التعرف على بعض الرموز المحظور استخدامها في أسماء المتغيرات؟ هل الاحرف العربية تنتمي الى هذه المجموعة من الرموز؟

سادساً: المتغيرات المكتوبة بالأحرف الكبيرة يتعامل معها مفسر بايثون على انها مختلفة عن المتغيرات المكتوبة بالأحرف الصغيرة. فمثلا المتغير car يختلف عن المتغير Car فمحرر بايثون يتعامل مع المتغيرين السابقين على انهما متغيرين منفصلين. لتوضيح هذه النقطة أكثر قم بإجراء التمرين التالي:

تمرين ٤

قم بتشغيل محرر بايثون التفاعلي واسند القيمة `5` للمتغير `car` واضغط زر الادخال. كرر العملية الان للمتغير `CAR` بإدخال القيمة `10`. حاول الان ان تتعرف على قيمة المتغير `car` ماذا تلاحظ؟ هل تغيرت قيمة المتغير `car` من `5` الى `10` ؟ لماذا؟ تعرف على قيمة المتغير `CAR` ؟

الحل:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> car = 5
>>> CAR = 10
>>> car
5
>>> CAR
10
>>>
```

سابعاً: أسماء المتغيرات يجب ان تكون مختلفة عن الكلمات المستخدمة في التركيب اللغوي للغة بايثون والجدول التالي يبين الكلمات المحجوزة من قبل لغة بايثون:

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

ولتمثيل على حظر استخدام كلمات بايثون المحجوزة كأسماء متغيرات انظر المثال التالي ادناه:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> global=5
  File "<stdin>", line 1
```

```
    global=5
          ^
SyntaxError: invalid syntax
>>> pass=5
  File "<stdin>", line 1
    pass=5
         ^
SyntaxError: invalid syntax
>>>
```

تدوين الملاحظات (comments)

تسمح لغة بايثون كغيرها من لغات البرمجة للمبرمج بأن يكتب ملاحظاته داخل الكود البرمجي من اجل ان تساعده على تذكر وظيفة الكود البرمجي او من اجل اعطاء شروحات وافيه للمبرمجين الاخرين الذين قد يعملون على صيانة وتطوير الكود البرمجي في المستقبل. ويمكن كتابة ملاحظة من سطر واحد في اي مكان من الكود البرمجي ولكن بعد ان يسبق الملاحظة علامة الوسم (#). فعند كتابة علامة (#) في بداية السطر فان بايثون يعتبر جميع محتويات هذا السطر ملاحظات للمستخدم وليست كود برمجي كما في المثال التالي:

```
# تكتب الملاحظات والتعليقات هنا باي لغة بهدف مساعدة المبرمج على فهم البرنامج
x =10
```

في المثال السابق كانت الملاحظات في سطر مستقل لكن ما يجب ملاحظته هو ان الملاحظات يمكن كتابتها في نفس السطر البرمجي كما في المثال التالي:

```
x =10   # تكتب الملاحظات هنا
```

فمفسر بايثون هنا يتعامل مع الجزء الذي يسبق علامة (#) على انه كود برمجي يحتاج الى تفسير والجزء الذي يقع بعد علامة (#) يعتبره ملاحظات وتعليقات للمبرمج لا تحتاج الى تفسير فيتجاهلها.

كما يمكن كتابة ملاحظة متعددة السطور باستخدام علامة التنصيص الثلاثية كما في المثال التالي:

```
'''
    comments ملاحظات
    notes  تعليقات
    description  شروحات
'''
x =10
```

تسمى طريقة استخدام علامة التنصيص الثلاثية في بايثون DocString وتستخدم بشكل خاص في كتابة تعليقات وشروحات داخل الدوال. وسوف نتحدث بشيء من التفصيل عن انواع علامات التنصيص واستخداماتها ولكن ليس الان بل عند بدء الحديث عن البيانات النصية لاحقاً.

الطباعة على شاشة الكمبيوتر

ان من المعتاد عند تعلم اي لغة برمجة جديدة ان يتم البدء بتعلم كيفية الطباعة على شاشة الكمبيوتر وذلك لأنها تعتبر وظيفة اساسية في جميع لغات البرمجة. ونحن سوف نسير على هذا العرف هنا على الرغم من تأخيرنا لها قليلا. فامر الطباعة على الشاشة في بايثون يكون باستخدام الدالة ()print متبوعا بما يراد طباعته داخل قوسي الدالة. وبما اننا في بداية المشوار ولم نتطرق للنصوص وطريقة كتابتها فسوف نبدأ بطباعة قيم رقمية اولا. فالسطر البرمجي التالي يقوم بطباعة القيمة 5 على الشاشة:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print(5)
5
>>>
```

تمرين

قم باسناد القيمة 25 للمتغير m وفي سطر ثاني قم بطباعة قمية m ؟

الاحرف الكبيرة ≠ الأحرف الصغيرة:

لغتنا العربية الجميلة لا تحوي على مفهوم الأحرف الصغيرة والكبيرة بعكس ما هو موجود في اللغة الانجليزية. وبما ان لغة بايثون مكتوبة باللغة الانجليزية فان هذه اللغة تهتم بما إذا كان الكود البرمجي او جزء منه مكتوب بالأحرف الصغيرة او الكبيرة. فالمتغير b يتعامل معه مفسر بايثون على انه مختلف عن B. فعندما نسند القيمة 5 مثلا للمتغير b فان اسناد القيمة 10 للمتغير B لا يغير من قيمة b لان مفسر بايثون يتعامل معهما على انهما متغيرين مختلفين كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> b=5
>>> B=10
>>> print(b)
5
>>>
```

اجراء العمليات الحسابية

كما هو المعتاد مع لغات البرمجة الاخرى فان العمليات الحسابية الأساسية في بايثون يمكن القيام بها كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
عملية الجمع
>>> 1+1
2
عملية الطرح
>>> 1-1
0
عملية الضرب
>>> 1*1
1
عملية القسمة
>>> 3/2
1.5
الاس
>>> 2**3
8
باقي القسمة
>>> 5%2
1
ناتج القسمة بعد اهمال الباقي
>>> 5//2
2
>>>
```

على الرغم من اننا لا ننصح باستخدام اصدارة بايثون 2 الا اننا نجد من الأفضل التنبيه الى انه يوجد اختلاف بسيط في عملية القسمة بين اصدارة بايثون 2 و 3. فعند اجراء عملية القسمة في اصدارة بايثون 2 على اعداد طبيعية فان ناتج القسمة يكون عدد صحيحا. بمعنى انه إذا أردنا قسمة العدد 3 على العدد 2 مثلا فان ناتج القسمة يكون 1 وليس 1.5 هذه المشكلة غير موجودة في اصدارة بايثون 3. ولتصحيح هذه المشكلة لمن يستخدمون الاصدارة 2 يمكنهم استخدام الاعداد العشرية في عملية القسمة سواء في البسط او المقام او كليهما كما في المثال التالي:

```
Python 2.7.10 (default, Jul 15 2017, 17:16:57)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 3/2
1
>>> 3/2.0
1.5
```

```
>>> 3.0/2
1.5
```

كما هو متعارف عليه في علم الرياضيات فان عمليتي القسمة والضرب تسبق عملية الطرح والجمع وعملية الاس تسبق الضرب والقسمة الا إذا استخدمت الاقواس لتحديد اسبقية العمليات الحسابية. وهذه امثلة توضح هذا المفهوم:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 10/2-1
4.0
>>> 10/(2-1)
10.0
>>> 10**2/5+4
24.0
>>> 10**2/(5+4)
11.11111111111111
>>> 2*(4+3)
14
>>> (3-6)/15
-0.2
>>> 3+4*2
11
>>>
```

أنواع البيانات

تنقسم البيانات الأساسية في لغة بايثون الي ثلاثة اقسام رئيسية وهي البيانات العددية والبيانات النصية والبيانات المنطقية. وسوف نقوم في هذه الجزئية من الفصل بإعطاء شيء من التفصيل عن كل نوع من هذه الانواع:

البيانات العددية: تنقسم البيانات العددية الى ثلاثة اقسام رئيسية:

1 - الاعداد الصحيحة (Integer numbers) او بشكل مختصر (int) وهي الاعداد التي تعبر عن عدم التجزئة وتشمل الاعداد الموجبة والسالبة والصفر {...,-3,-2,-1,0,1,2,3,....}.

2 - الأعداد العائمة (float numbers) او العشرية وهي الاعداد التي تحتوي على فاصلة عشرية سواء كانت موجبة او سالبة.

3 - الاعداد المركبة (complex numbers) وهي الاعداد التي تحتوي على اعداد حقيقية واعداد خيالية فمثلا العدد 2+3j يعتبر عدد مركب يتكون من جزء صحيح وهو العدد 2 وجزء خيالي هو العدد 3 ويمثل الحرف j الجذر التربيعي للعدد 1- حيث يكتب في لغة بايثون بالطريقة التالية:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3j
(2+3j)
>>>
```

ويمكن استخدام الدالة type() لمعرفة أنواع البيانات كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> type(4)
<class 'int'>
>>> type(4.1)
<class 'float'>
>>> type(2+3j)
<class 'complex'>
>>>
```

البيانات *المنطقية*: (Boolean data) وهي البيانات التي تحوى على احد القيمتين صح (True) او خطأ (False) لاحظ ان هاتين القيمتين تبتدأ بحرف كبير في لغة بايثون بعكس بعض اللغات الاخرى. وغالبا ما تنتج هذه القيم بعد اجراء عمليات المقارنة بين قيمتين او أكثر. فعند مقارنة عددين مثلاً في بايثون فإننا نستخدم علامتي يساوي متتابعتين "==" بين العددين كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 4==9
False
>>>
```

فعند عدم تساوي الرقمين مثلا فإننا نحصل على القيمة False اي خطأ. وعندما تكون القيمتين متساويتين فان الناتج يكون "True" اي صحيح كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 23==23
True
>>>
```

البيانات النصية: كل ما يكتب بين علامتي تنصيص سواء كانت احادية ('z') او ثنائية ("z") او ثلاثية ('''z''') يطلق عليه بيان نصي string أو بشكل مختصر .str فالقيمة "123" يتعامل معها بايثون على انها سلسلة من الرموز النصية مختلفة عن القيمة العددية 123. ويمكن التأكد من ذلك باستخدام الدالة type() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> type(123)
<class 'int'>
>>> type("123")
<class 'str'>
>>>
```

وهناك عدة قواعد يجب اتباعها عند كتابة البيانات النصية والتي يمكن تلخيصها في النقاط التالية:

اولاً: علامتي التنصيص لكل بيان نصي يجب ان تكون من نفس النوع. فعند البدء بعلامة تنصيص احادية يجب ان ينتهي البيان النصي بعلامة تنصيص احادية وكذلك الحال مع علامة التنصيص الثنائية والثلاثية كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 'Hello'
'Hello'
>>> "Hello"
'Hello'
>>> '''Hello'''
'Hello'
>>>
```

لاحظ انه عندما نقوم باستخدام علامتي تنصيص مختلفة فان ذلك سوف يجعل مفسر لغة بايثون يظهر رسالة تفيد بوجود خطأ كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> "Hello world'
  File "<stdin>", line 1
    "Hello world'
                 ^
SyntaxError: EOL while scanning string literal
```

علامة التنصيص الاحادية والثنائية تستخدم لكاتبة نصوص من سطر واحد بينما علامة التنصيص الثلاثية تسمح بكتابة أكثر من سطر. فعند استخدام علامة التنصيص الاحادية والثنائية لطباعة بيان نصي متعدد السطور فان مفسر بايثون التفاعلي يظهر لنا رسالة تفيد بوجود خطأ بمجرد النزول الى السطر التالي كما في المثالين التاليين:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> "Hello
  File "<stdin>", line 1
    "Hello
          ^
SyntaxError: EOL while scanning string literal
>>> 'Hello
  File "<stdin>", line 1
    'Hello
          ^
SyntaxError: EOL while scanning string literal
>>>
```

فالعبارة EOL التي يظهرها لنا محرر بايثون تفيد بان المحرر وصل الى نهاية السطر ولم يجد العلامة النصية التي ينتهي بها البيان النصي.

اما عند استخدام علامة التنصيص الثلاثية فان مفسر بايثون يتمم عملية الطباعة بدون اظهار اي رسالة خطأ كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('''Welcome
... to
... python''')
Welcome
to
python
>>>
```

يمكن استخدام رمز التجاهل "\" لتمكين علامة التنصيص الاحادية والثنائية من طباعة أكثر من سطر على الشاشة. لكن هناك فرق بين الطريقتين فعلامة التنصيص الثلاثية تبقي البيان النصي كما كتب من ناحية تعدد السطور بينما استخدام رمز التجاهل يقوم بكتابة البيان النصي كسطر واحد كما في المثالين التاليين:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> print("Welcome\
... to \
... python")
Welcometo python
>>>

>>> print('''Welcome
... to
... python''')
Welcome
to
python
>>>
```

يمكن استخدام علامة تنصيص او أكثر داخل علامتي تنصيص اخرى ولكن بعد التأكد من ان علامة التنصيص الداخلية مختلفة عن علامة التنصيص المستخدمة في بداية ونهاية النص كما هو موضح في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("My name is 'Omar' what is yours?")
My name is 'Omar' what is yours?
>>> print('Say "Hello" to your friends')
Say "Hello" to your friends
>>>
```

كما يمكن استخدام رمز التجاهل المشار اليه سابقا لأداء نفس الوظيفة إذا كانت علامة التنصيص الداخلية مشابهة لعلامتي التنصيص الخارجية. بحيث توضع علامة التجاهل قبل علامة التنصيص المراد تجاهلها كما ما هو موضح في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('My brother\'s car is new')
My brother's car is new
>>>
```

في المثال السابق استخدمنا علامة التجاهل (\) لتساعدنا على كتابة بيان نصي يحتوي بداخله على علامة تنصيص. فمفسر بايثون قام بطباعة علامة التنصيص بدون ان يظهر لنا علامة التجاهل في النص. لكن ماذا لو اردنا ان تكون علامة التجاهل جزء من النص المراد طباعته؟. ببساطة يتم اظهار علامة التجاهل في النص باستخدام علامة تجاهل أخرى بجانبها كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> print("The symbol \\ can be shown in this example")
The symbol \ can be shown in this example
>>>
```

وهناك علامات تجاهل أخرى خاصة تؤدي وظائف خاصة فمثلا \t يعتبرها مفسر بايثون على انها حقل فارغ مكون من اربع مسافات كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("1\t2\t3\t4")
1       2       3       4
>>>
```

فمفسر بايثون في المثال السابق قام بتحويل كل زمر تجاهل \t الى حقل فارغ مكون من اربع مسافات. وكذلك الامر مع رمز التجاهل \n فانه يخبر مفسر بايثون بالنزول الى سطر جديد كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("1\n2\n3\n4")
1
2
3
4
>>>
```

ويمكن التعرف على بقية رموز التجاهل من خلال الجدول التالي:

رمز التجاهل	المعنى
\newline	Ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)

\ooo	ASCII character with octal value *ooo*
\xhh...	ASCII character with hex value *hh...*

اذا اردنا ان نخبر مفسر بايثون بأن يتجاهل جميع رموز التجاهل الموجودة في النص فإننا نسبق النص بالحرف r ليخبر مفسر بايثون ان هذا النص لا يحتاج الى تفسير الرموز بداخله كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("1\t2\t3\t4")
1       2       3       4
>>> print("1\n2\n3\n4")
1
2
3
4
>>> print(r"1\t2\t3\t4")
1\t2\t3\t4
>>> print(r"1\n2\n3\n4")
1\n2\n3\n4
>>>
```

كما يجب الإشارة الى ان البيانات النصية ليس لها حد في قيمها فممكن ان تتكون من سلسلة من الحروف والأرقام والرموز بقدر ما تستوعبه ذاكرة الكمبيوتر المستخدم. ويكمن كذلك ان يكون البيان النصي فارغاً كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("")

>>>
```

الفصل الثالث

التعامل مع البيانات التجميعية (Collection Data)

في الفصل السابق تعلمنا كيفية التعامل مع البيانات الأساسية وكان كل بيان من تلك البيانات يمثل معلومة مستقلة بذاتها. اما في هذا الفصل فسوف نوسع مفهوم البيانات الى نطاق أوسع بحيث يتم ترتيب هذه البيانات الأساسية بطريقة خاصة في مجموعات فتصبح هذه البيانات مستقلة بذاتها.

أهداف الفصل

عند اتمام هذا الفصل يجب ان يكون لديك المام بالآتي :

- التعرف على البيانات التجمعية وأنواعها الرئيسية.
- معرفة التعامل مع الانواع الأساسية للبيانات التجميعية.
- التمييز بين الانواع الأساسية للبيانات التجميعية من خلال معرفة خصائص كل نوع.
- التوسع في معرفة البيانات النصية وكيفية التعامل معها.

البيانات التجميعية

المقصود بالبيانات التجميعية في لغة بايثون هي بيانات أساسية (سواء كانت عددية او نصية او منطقية) اندرجت تحت مجموعة لها خصائص معينه ومحددة فأصبحت بيانات مستقلة بذاتها يتعامل معها مفسر بايثون بطريقة خاصة. وتنقسم هذه البيانات التجميعية الى أربعة أنواع رئيسية:

1 - القوائم (lists)
2 - الصفوف (tuples).
3 - القواميس (dictionaries).
4 - المجموعات (sets).

القوائم (Lists)

القوائم في لغة بايثون هي عبارة عن مجموعة من البيانات توضع بين قوسين مربعين يفصل بين كل بيان واخر بفاصلة ويشار اليها بمتغير واحد. وتعتبر القوائم وسيلة سهلة لتخزين البيانات قبل اجراء أي عمليات تحليلية عليها. ويمكن كتابة قائمة بأشهر السنه كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
months=['Jan','Feb','Mar','Apr','Jun','jul','Aug','Sep','Oct','Nov','Dec']
>>>
```

كما يمكن انشاء قائمة فارغة بأكثر من طريقة. فمثلا يمكن انشاء قائمة فارغة باستخدام قوسين مربعين فارغين بعد اسم المتغير كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> cars=[]
>>>
```

كما يمكن انشاء قائمة فارغة باستخدام الدالة list() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
cars=list()
>>>
```

ولإضافة بيانات لقائمة فارغة او قائمة تحتوي على بيانات مسبقة يمكن استخدام الامر append() بحيث يكون البيان المضاف دائما في اخر القائمة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> list1=[]
>>> list1.append('Sunday')
>>> list1
['Sunday']
>>> list1.append('Monday')
>>> list1
['Sunday', 'Monday']
>>> list1.append('Tuesday')
>>> list1
['Sunday', 'Monday', 'Tuesday']
>>>
```

لاحظ استخدام النقطة بعد اسم القائمة لتنفيذ الامر. فهي تعنى قم بإضافة القيمة "Sunday" مثلا للقائمة list1 .

كما يجب التنويه الى ان القوائم يمكن ان تحتوي على اكثر من نوع من البيانات فالقائمة التالية تحتوي على بيانات نصية وبيانات عددية وبيانات منطقية:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> list2=['Ahmad',1990,True,23.5]
>>>
```

ولمعرفة عدد البيانات في قائمة فإننا نستخدم الدالة ()len كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> list2=['Ahmad', 1990, 'Hello', True, 23.5]
>>> len(list2)
5
>>>
```

ويتم الاشارة الى مكان البيانات في القائمة بأرقام صحيحة تسمى Index تبدأ من الصفر. فالقائمة السابقة يشار الى البيان الاول فيها بالرقم 0 والبيان الثاني بالرقم 1 وهكذا الى نهاية البيانات في القائمة.

مؤشر البيانات (list index)	0	1	2	3
البيانات	“Ahmad”	1990	True	23.5

وللحصول على بيان معين من قائمة فإننا نقوم بكتابة اسم القائمة متبوعا بقوسين مربعين يوضع بينها رقم مؤشر البيان. فعلى سبيل المثال عندما نريد البيان الاول من قائمة list2 السابقة فإننا نكتب الكود التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> list2[0]
'Ahmad'
>>>
```

وعندما نريد البيان الثالث مثلا فإننا نكتب الامر التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> list2[2]
True
>>>
```

كما يمكن الحصول على بيانات في قائمة ابتداءً من اليمين الى اليسار او بمعنى اخر من اخر بيان في القائمة. وللقيام بذلك نقوم باستخدام مؤشر سالب بين القوسين المربعين كما هو موضح في الجدول التالي:

مؤشر البيانات (list index)	-4	-3	-2	-1
البيانات	“Ahmad”	1990	True	23.5

وللحصول على أخر بيان في القائمة list2 السابقة باستخدام المؤشر الاعتيادي او المؤشر العكسي نقوم التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> list2[3]
23.5
>>> list2[-1]
23.5
>>>
```

للحصول على البيان قبل الأخير في قائمة فإننا نستخدم المؤشر 2- داخل القوسين المربعين و 3- للبيان الذي بعده وهكذا لباقي البيانات في القائمة.

كما يمكننا ان نغير قيم البيانات داخل قائمة باستخدام معامل الاسناد (=) كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> list2=["Ahmad",1990,True,23.5]
>>> list2[0]="Omar"
>>> list2
['Omar', 1990, True, 23.5]
>>>
```

ولمعرفة مؤشر بيان ما داخل قائمة فإننا نستخدم الدالة index() بحيث نضع البيان المراد معرفة مؤشره داخل قوسي الدالة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> list2=["Ahmad",1990,True,23.5]
>>> list2.index(1990)
1
>>> list2.index(23.5)
3
>>> list2.index("Ahmad")
0
```

```
>>>
```

في مثال سابق تعلمنا كيف نضيف بيان الى قائمة باستخدام الامر append() وعلمنا ان هذا الامر يقوم بإضافة البيان الى اخر القائمة. لكن ماذا لو أردنا ان نضيف بياناً في مكان معين من قائمة. لأداء هذه المهمة نستخدم الامر insert() وكيفية استخدام هذا الامر تتطلب أولا ادخال مؤشر البيان داخل القائمة التي سوف يحتلها البيان المراد إدخاله ومن ثم قيمة البيان كما هو موضح في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> list2.insert(2,"Hello")
>>> list2
['Ahmad', 1990, 'Hello', True, 23.5]
>>>
```

تمرين ١

١- اكتب قائمة فارغة وسمها cars؟
٢- أضف ثلاثة أنواع من السيارات الى هذه القائمة؟
٣- أفترض الان اننا نسينا ان نضيف نوع رابع من السيارات الى هذه القائمة ونود اضافته لأول القائمة كيف نقوم بذلك؟

ولإزالة بيان من قائمة فان هنا عدة دوال تساعدنا على القيام بذلك بطرق مختلفة. فعندما نريد إزالة اخر بيان في قائمة فإننا نستخدم الدالة pop() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> list2=['Ahmad', 1990, 'Hello', True, 23.5]
>>> list2.pop()
23.5
>>> list2
['Ahmad', 1990, 'Hello', True]
>>>
```

هنا الدالة pop() قامت بحذف اخر بيان من القائمة وطباعة البيان الذي تم حذفه بحيث انه بإمكاننا اسناد ما تم حذفه الى متغير كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> list2=['Ahmad', 1990, 'Hello', True]
>>> last_value=list2.pop()
>>> last_value
True
>>>
```

وكذلك يمكن إزالة عنصر معين من القائمة باستخدام الدالة ()remove وذلك بتحديد البيان المراد حذفه كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> list2=['Ahmad', 1990, 'Hello']
>>> list2.remove("Ahmad")
>>> list2
[1990, 'Hello']
>>>
```

كما انه بالإمكان استخدام الدالة العامة del لإزالة البيان المراد كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> list2=['Ahmad', 1990, 'Hello', True, 23.5]
>>> del list2[1]
>>> list2
['Ahmad', 'Hello', True, 23.5]
>>>
```

وأخيرا يمكن افراغ القائمة من البيانات بالكامل باستخدام الدالة ()clear بعد اسم القائمة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> list2=['Ahmad', 'Hello', True, 23.5]
>>> list2.clear()
>>> list2
[]
>>>
```

يجب ملاحظة ان القائمة تسمح بتكرار العناصر بمعنى انه يمكن تخزين نفس البيان في قائمة اكثر من مرة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> greatings=["hello","hello","hello"]
>>> greatings
['hello', 'hello', 'hello']
>>>
```

ولمعرفة عدد العناصر المكررة في قائمة نستخدم الدالة count() بعد تحدد البيان المراد معرفة عدد مرات تكراره كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> greatings=['hello', 'hello', 'hello', 'Hi', 'Hi']
>>> greatings.count("hello")
3
>>> greatings.count("Hi")
2
>>>
```

في مثال سابق تعلمنا كيف نستخدم الدالة append() لإضافة بيان واحد الى اخر القائمة. ولان نريد ان نوسع معرفتنا بهذه الدالة ونتعرف على ان هذه الدالة باستطاعتها إضافة قائمة الى اخر القائمة بحيث تظهر القائمة كقائمة جزئية داخل قائمة رئيسية كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> numbers=[1,2,3,4]
>>> numbers.append(['a','b','c'])
>>> numbers
[1, 2, 3, 4, ['a', 'b', 'c']]
>>>
```

لاحظ ان القائمة الرئيسية هي القائمة التي تسبق append() والقائمة الفرعية هي التي بداخل قوسي append().

لكن ماذا لو أردنا ان ندمج عناصر القائمة الفرعية مع عناصر القائمة الرئيسية بحيث تصبح كأنها قائمة رئيسية واحدة. للقيام بذلك يمكننا استخدام علامة الجمع بين القائمتين كما في المثال التالي:

```
 Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

```
numbers=[1, 2, 3, 4]
>>> numbers+['a','b','c']
[1, 2, 3, 4, 'a', 'b', 'c']
>>> ['a','b','c']+numbers
['a', 'b', 'c', 1, 2, 3, 4]
>>>
```

لاحظ ان ترتيب البيانات يعتمد على الطريقة التي تمت بها عملية الجمع.

كما يمكننا ان نستخدم الدالة extend() لأداء نفس المهمة بحيث نضع القائمة الفرعية المراد دمجها داخل قوسي الدالة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> numbers=[1,2,3,4]
>>> numbers.extend(['a','b','c'])
>>> numbers
[1, 2, 3, 4, 'a', 'b', 'c']
>>>
```

يمكن اجراء عملية ضرب على قائمة فتكون النتيجة كالتالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> numbers=[1,2,3]*4
>>> numbers
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> greetings=['Hi']*3
>>> greetings
['Hi', 'Hi', 'Hi']
>>>
```

لاحظ ان عملية الضرب هنا هي مجرد تكرار للبيانات داخل القائمة.

ويمكن عكس ترتيب البيانات في قائمة باستخدام الدالة reverse() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> numbers=[1, 2, 3, 4, 'a', 'b', 'c']
>>> numbers.reverse()
>>> numbers
```

```
['c', 'b', 'a', 4, 3, 2, 1]
>>>
```

ولإعادة ترتيب البيانات داخل قائمة فإننا نستخدم الدالة sort() بحيث تقوم هذه الدالة بترتيب البيانات تصاعديا بشكل تلقائي على حسب ترتيب الحروف الابجدية او الاعداد كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> cars = ['Ford', 'BMW', 'Volvo']
>>> cars.sort()
>>> cars
['BMW', 'Ford', 'Volvo']
>>>
```

ولعكس الترتيب بحيث يكون تنازليا نستخدم الخيار reverse=True داخل قوسي الدالة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> cars = ['Ford', 'BMW', 'Volvo']
>>> cars.sort(reverse=True)
>>> cars
['Volvo', 'Ford', 'BMW']
>>>
```

ويمكن أيضا تحدد طريقة الترتيب بحسب الكيفية التي ترغب بها ولكن هذه الخاصية متقدمة قليلا ولا نستطيع الحديث عنها هنا الان.

ولمعرفة ما اذا كان بيان معين موجود في قائمة فإننا نستخدم in بين البيان المراد البحث عنه واسم القائمة ليكون معنى التركيب اللغوي "هل البيان x موجود في القائمة y" فيكون ناتج العملية اما بصح True او خطأ False كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> colors=['green','red','blue']
>>> 'green' in colors
True
>>> 'yellow' in colors
False
>>>
```

ولعمل نسخه من القائمة نستخدم الدالة copy() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> cars = ['Ford', 'BMW', 'Volvo']
>>> cars2=cars.copy()
>>> cars2
['Ford', 'BMW', 'Volvo']
>>>
```

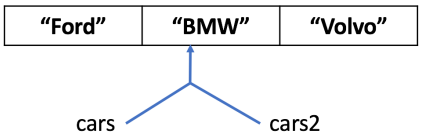
قد يتبادر الى الاذهان ان عملية النسخ باستخدام الدالة copy() غير ضروري لان اجراء عملية اسناد قائمة لمتغير جديد سوف تفي بالغرض كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> cars = ['Ford', 'BMW', 'Volvo']
cars2=cars
>>> cars2
['Ford', 'BMW', 'Volvo']
```

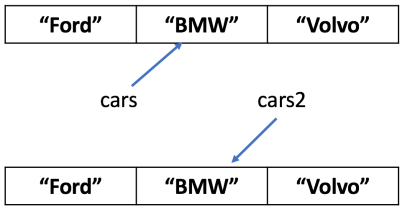
لكن في حقيقة الامر هناك اختلاف جوهري بين استخدام الدالة copy() وعملية الاسناد. فعملية الاسناد لا تعني تكوين قائمة مستقلة عن القائمة الاولى. بل ان اجراء أي تغيير في أي متغير سوف يؤثر على القائمة الاخرى كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> cars = ['Ford', 'BMW', 'Volvo']
cars2=cars                    عملية الاسناد الى متغير جديد
>>> cars2
['Ford', 'BMW', 'Volvo']
>>> cars.append("GMC")         عملية التغيير على القائمة الرئيسية
>>> cars
['Ford', 'BMW', 'Volvo', 'GMC']
>>> cars2                              ظهور التأثير على المتغير الجديد
['Ford', 'BMW', 'Volvo', 'GMC']
>>>
```

لاحظ انه عند إضافة بيان الى القائمة cars تم حدوث التغيير في أيضا في المتغير cars2 . ولذلك لأنه في حقيقة الامر لا يوجد قائمتين في ذاكرة الكمبيوتر أصلا. ان ذاكرة الكمبيوتر تحتفظ بقائمة واحدة فقط وعملية الاسناد قامت فقط بتكوين متغير يشير الى نفس القائمة كما في الشكل التالي:



اما عملية النسخ بالدالة ()copy فانها تقوم بإنشاء قائمة مستقلة عن القائمة الاولى في داكرة الكمبيوتر كما في الشكل التالي:



لذلك عند اجراء تغيير على القائمة التي نسخة بالدالة ()copy فان هذا التغيير لا يؤثر على القائمة الاصلية والعكس صحيح كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> cars = ['Ford', 'BMW', 'Volvo']
>>> cars2=cars.copy()           عملية النسخ
>>> cars.append([1,2,3])        عملية التغيير على القائمة الاصلية
>>> cars
['Ford', 'BMW', 'Volvo', [1, 2, 3]]
>>> cars2                       القائمة المنسوخة لم تتأثر
['Ford', 'BMW', 'Volvo']
>>> cars2.pop()                 عملية التغيير على القائمة المنسوخة
'Volvo'
>>> cars2
['Ford', 'BMW']
>>> cars                        القائمة الاصلية لم تتأثر
['Ford', 'BMW', 'Volvo', [1, 2, 3]]
>>>
```

ويمكن تجزئة قائمة (list slicing) الى قوائم أصغر بعدة طرق. فمثلا اذا اردنا اخذ الثلاثة البيانات الاولى من القائمة letters=[‘a’ , ’b’ , ’c’ , ’d’ , ’e’ , ’f” , ’g’] فإننا نقوم بالتالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> letters=['a','b','c','d','e','f','g']
>>> letters[0:3]
['a', 'b', 'c']
>>>
```

ما قمنا به في المثال السابق هو اننا استخدمنا مؤشر البيان الذي نريد ان تبتدأ به القائمة الجزئية داخل قوسين مربعين ثم وضعنا نقطتين فوق بعض (:) ثم وضعنا مؤشر اخر بيان لا نريد ان يكون في القائمة الجزئية. ففي المثال السابق اول بيان نريده في القائمة الجزئية هو ‘a’ ومؤشره 0 واخر بيان لا نريده ان يكون في القائمة الجزئية هو ‘d’ ومؤشره هو 3 . لذلك حصلنا القائمة الجزئية [‘a’ , ’b’ , ’c’] . ويكمن الحصول على نفس المهمة بترك مؤشر البيان الذي سوف تبتدأ به القائمة الجزئية فارغ لأن مفسر بايثون سوف يفترض انك تريد ان تبتدأ القائمة الجزئية من بداية القائمة الرئيسية كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> letters=['a','b','c','d','e','f','g']
>>> letters[:3]
['a', 'b', 'c']
>>>
```

وللحصول على قائمة جزئية تبتدأ مثلاً من الحرف ‘e’ وتنتهي باخر بيان في القائمة السابقة نكتفي بتحديد مؤشر العدد الذي نريد ان تبتدأ به القائمة الجزئية ونترك المؤشر الاخر فارغ لأن مفسر بايثون يفترض انك تريد بقية البيانات في القائمة الرئيسية كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> letters=['a','b','c','d','e','f','g']
>>> letters[4:]
['e', 'f', 'g']
>>>
```

تمرين

١-استكشف ماذا ينتج عن استخدام `letters[:]` للقائمة السابقة؟

٢-استكشف ماذا ينتج عن استخدام `letters[::]` للقائمة السابقة؟

٣-استكشف ماذا ينتج عن استخدام `letters[::-1]` للقائمة السابقة؟ حاول تذكر أي دالة من الدوال التي تعلمناها سابقا تقوم بنفس المهمة؟

٤-استكشف ماذا ينتج عن استخدام `letters[::1]` للقائمة السابقة؟

٥-استكشف ماذا ينتج عن استخدام `letters[::2]` للقائمة السابقة؟

الصفوف (tuples)

الصفوف في لغة بايثون هي عبارة عن مجموعة من البيانات غير قابلة للتغيير توضع بين قوسين منحنيين يفصل بين كل بيان واخر بفاصلة ويشار اليها بمتغير واحد كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> fruits=("banana","orange","apple")
>>>
```

حتى لو لم نكتب القوسين المنحنيين عند كتابتنا للبيانات التي تم فصلها عن بعض بفاصله فان مفسر بايثون سوف يفترض انك تريد كتابة صف من البيانات ويظهر لك النتيجة بقوسين منحنيين كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> colors='blue','red','orange'
>>> colors
('blue', 'red', 'orange')
>>>
```

واهم خاصية تميز الصفوف عن القوائم هي ان البيانات في الصفوف لا يمكن تغييرها ولو حاولنا ذلك نجد ان مفسر بايثون يعطينا رسالة بوجود خطأ مفاده ان الصفوف غير قابلة للتغيير كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> fruits=("banana","orange","apple")
>>> fruits[0]="grapes"
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

وللحصول على صف فارغ يمكننا ان نستخدم الطريقة التالية:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> houses=()
```

وكما هو الحال في القوائم فان البيانات في الصفوف يتم الحصول عليها بتحديد مؤشر البيان داخل قوسين مربعين بعد اسم متغير الصف كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> fruits=("banana","orange","apple")
>>> fruits[0]
'banana'
>>> fruits[2]
'apple'
>>>
```

وكذلك عملية تجزئة الصفوف تتم بنفس الطريقة التي تعلمناها في تجزئة القوائم كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> colors=('red','green','blue','yellow','orange')
>>> colors[:]
('red', 'green', 'blue', 'yellow', 'orange')
>>> colors[2:4]
('blue', 'yellow')
>>> colors[:4]
('red', 'green', 'blue', 'yellow')
>>> colors[2:]
('blue', 'yellow', 'orange')
>>> colors[::-1]
('orange', 'yellow', 'blue', 'green', 'red')
>>> colors[::2]
('red', 'blue', 'orange')
>>>
```

وكذلك عملية الجمع والضرب بين صفين تتم بنفس الطريقة التي تعلمناها مع القوائم كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> colors=('red','green','blue','yellow','orange')
>>> colors+(1,2,3)
('red', 'green', 'blue', 'yellow', 'orange', 1, 2, 3)
>>> letters=('a','b','c')
>>> letters*3
('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
>>>
```

بشكل مشابه للقوائم يمكن معرفة عدد البيانات في صف باستخدام الدالة len() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> colors=('red','green','blue','yellow','orange')
>>> len(colors)
5
>>>
```

وللسؤال عما إذا كان بيان ما موجود في صف فإننا نتبع نفس الطريقة المستخدمة مع القوائم باستخدام in كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> colors=('red','green','blue','yellow','orange')
>>> 'blue' in colors
True
>>> 'pink' in colors
False
>>>
```

تمرين

١-حاول ان تستخدم احد الدوال التي استخدمناها مع القوائم لاحداث تغيير على صف كاستخدام الدالة `remove()` او `append()` مثلا ماذا تلاحظ ولماذا؟

٢-حاول ان تستخدم الدوال التي لا تحدث تعديلا على بيانات الصف كالدالة `count()` و الدالة `index()` ماذا تلاحظ؟

٣-جرب استخدام الامر `del` على صف ماذا تلاحظ؟

٤-جرب استخدام الامر `dir()` واضعا بين قوسيه اسما لصف و تعرف على الدوال التي يمكن استخدامها مع الصف؟

(dictionaries)

القواميس في بايثون هي القواميس عبارة عن مجموعة من البيانات توضع بين قوسين متعرجين ويفصل بين بين كل بيان واخر بفاصلة بحيث يتكون كل بيان من جزأين. الجزء الاول يسمى المفتاح (key) والجزء الاخر يسمى القيمة (value) ويفصل بين المفتاح والقيمة بنقطتين فوق بعض كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name':'Hassan','Age':22,'Job':'student'}
>>>
```

المفتاح في القاموس يجب ان يكون مميزا عن غيره من المفاتيح في القاموس وإذا حاولنا كتابة قاموس يحتوي على مفتاحين متشابهين فان القيمة الاولى سوف يتم حذفها وتبقى القيمة الأخيرة داخل القاموس كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name':'Ali','Age':22,'Job':'student','name':'Waleed'}
>>> info
{'name': 'Waleed', 'Age': 22, 'Job': 'student'}
>>>
```

اما قيم المفاتيح فيمكن ان تتشابه ولا يعترض مفسر بايثون على تشابه القيم طالما المفاتيح مختلفة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> example={1:'odd',2:'even',3:'odd'}
>>> example
{1: 'odd', 2: 'even', 3: 'odd'}
>>>
```

مفاتيح القاموس يجب ان تكون بيانات غير قابلة للتغير فيمكن ان تكون نص او رقما او صفا ولكن لا يمكن ان تكون قائمة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
```

```
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> keys_types={23:'number','Hello':'string',(1,2,3):'tuples'}
>>> keys_types
{23: 'number', 'Hello': 'string', (1, 2, 3): 'tuples'}
>>> list_dict={[1,2]:'list'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

لاحظ انه عندما استخدمنا القائمة كمفتاح في قاموس اظهر لنا مفسر بايثون رسالة بوجود خطأ من النوع TypeError والذي يفيد بوجود خطأ في نوعية البيانات المستخدمة.

ويمكن كتابة قاموس فارغ بإحدى الطريقتين التاليتين:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={}
>>> info2=dict()
```

ويمكن الحصول على قيمة لمفتاح ما في قاموس باستخدام قوسين مربعين توضع بعد اسم القاموس ويكتب داخلها اسم المفتاح كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name':'Ahmad','Age':22,'Job':'student'}
>>> info['Age']
22
>>> info['name']
'Ahmad'
>>>
```

القواميس مثل القوائم يمكن تعديل البيانات الموجودة بداخلها فمثلا إذا اردنا تعديل Age في القاموس السابق فإننا نقوم بالاتي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name':'Ahmad','Age':22,'Job':'student'}
```

```
>>> info['Age']=40
>>> info
{'name': 'Ahmad', 'Age': 40, 'Job': 'student'}
>>>
```

كما يمكن إضافة بيانات جديدة الى قاموس بالطريقة التالية:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name':'Ahmad','Age':22,'Job':'student'}
>>> info['car']='Ford'
>>> info
{'name': 'Ahmad', 'Age': 22, 'Job': 'student', 'car': 'Ford'}
>>>
```

لمعرفة عدد البيانات في قاموس فاننا نستخدم الدالة ()len كما فعلنا في القوائم والصفوف:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name':'Ahmad','Age':22,'Job':'student'}
>>> len(info)
3
>>>
```

ولمعرفة ما اذا كان هناك بيان ما موجود في قاموس فاننا نستخدم in كما تعلمنا سابقا في القوائم والصفوف:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name':'Ahmad','Age':22,'Job':'student'}
>>> 'Age' in info
True
>>> 'car' in info
False
>>>
```

لحذف بيان من قاموس فإننا نستخدم الامر del كما فعلنا في القوائم:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name':'Ahmad','Age':22,'Job':'student'}
>>> del info['Job']
```

```
>>> info
{'name': 'Ahmad', 'Age': 40}
>>>
```

ولتفريغ القاموس من البيانات بالكامل نستخدم الدالة clear() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name':'Ahmad','Age':22}
>>> info.clear()
>>> info
{}
>>>
```

ولمعرفة كافة المفاتيح في قاموس ما نستخدم الدالة keys() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name': 'Ahmad', 'Age': 22, 'Job': 'student'}
>>> info.keys()
dict_keys(['name', 'Age', 'Job'])
>>>
```

ولمعرفة كافة القيم في القاموس نستخدم الدالة values() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name': 'Ahmad', 'Age': 22, 'Job': 'student'}
>>> info.values()
dict_values(['Ahmad', 22, 'student'])
>>>
```

ولدمج بيانات قاموسين نستخدم الدالة update() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name': 'Ahmad', 'Age': 22, 'Job': 'student'}
info2={'title':'Mr','phone':'00000'}
>>> info.update(info2)
>>> info
```

```
{'name': 'Ahmad', 'Age': 22, 'Job': 'student', 'title': 'Mr', 'phone':
'00000'}
>>>
```

لاحظ ان القاموس الذي يسبق الدالة هو القاموس الأساسي والقاموس الذي بداخل قوسي الدالة هو القاموس الذي تم دمج البيانات منه. يجب ملاحظ ان بالإمكان وضع قاموس فرعي داخل قاموس رئيسي كما يلي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name': 'Ahmad', 'Age': 22, 'Job': 'student'}
info2={'title':'Mr','phone':'00000'}
>>> info['more_data']=info2
>>> info
{'name': 'Ahmad', 'Age': 22, 'Job': 'student', 'more_data': {'title': 'Mr',
'phone': '00000'}}
>>>
```

في المثال السابق تم إضافة القاموس الفرعي كما هو تحت مفتاح جديد اسميناه more_data كما تعلمنا سابقا عن كيفية إضافة بيان جديد الى قاموس.

ويمكن استخدام الدالة items() لعرض جميع البيانات في قاموس كقائمة كل بيان يظهر كصف مكون من مفتاح وقيمة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> info={'name': 'Ahmad', 'Age': 22, 'Job': 'student'}
>>> info.items()
dict_items([('name', 'Ahmad'), ('Age', 22), ('Job', 'student')])
>>>
```

المجموعات (sets)

المجموعات في لغة بايثون هي عبارة عن تجمع لبيانات غير قابلة للتكرار توضع بين قوسين متعرجين {} يفصل بين كل بيان واخر بفاصلة ويشار اليها بمتغير واحد كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> numbers={1,2,3,4,1,3}
```

```
>>> numbers
{1, 2, 3, 4}
>>>
```

لاحظ في المثال السابق اننا حاولنا تكوين مجموعة بيانات متكررة لكن محرر بايثون حذف البيانات المتكررة من المجموعة لان المجموعات في بايثون لا تسمح بتكرار البيانات المتشابهة. لاحظ أيضا التشابه بين المجموعات والقواميس من حيث الاقواس. الاختلاف بين القواميس والمجموعات يكمن في طريقة كتابة البيانات فالقواميس كل بيان يتكون من مفتاح وقيمة مفصولين بنقطتين فوق بعض اما المجموعات فهي بيانات اعتيادية.

ويمكن انشاء مجموعة اما بالطريقة التي استخدمناها في المثال السابق او باستخدام الدالة set() وإدخال قائمة من البيانات داخل قوسي الدالة كما في المثال التالي:

```
 Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> letters=set(['a','b','c'])
>>> letters
{'a', 'c', 'b'}
>>>
```

واذا اردنا ان ننشئ مجموعة فارغة فإننا لا نستطيع ان نستخدم قوسين فارغين لأن هذا التركيب اللغوي محجوز للقواميس ولكن بوسعنا ان نستخدم الدالة set() بدون ان ندخل أي شيء بين قوسيها كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> numbers={}                  استخدام القوسين المتعرجين محجوز للقواميس
>>> type(numbers)
<class 'dict'>
>>>
 >>> letters=set()
>>> letters
set()
>>>
```

ولإضافة بيان الي مجموعة فارغة او مجموعة محتوية على بيانات سابقة نستخدم الدالة set() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> letters=set()
```

```
>>> letters.add('Ahmad')
>>> letters.add('Waleed')
>>> letters
{'Ahmad', 'Waleed'}
>>> letters.add('Hassan')
>>> letters
{'Hassan', 'Ahmad', 'Waleed'}
>>>
```

لاحظ ان ترتيب البيانات في المجموعات ليس بالضرورة ان يكون بنفس الترتيب الذي ادخلنا فيها البيانات وذلك لان المجموعات لا تستخدم مؤشر البيانات للإشارة الى البيانات بداخلها واذا حاولنا القيام بذلك فان مفسر بايثون سوف يظهر لنا رسالة تفيد بان المجموعات لا تستخدم مؤشر البيانات كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.


>>> letters={'Hassan', 'Ahmad', 'Waleed'}
>>> letters[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>>
```

ولحذف بيان من مجموعة هناك عدة خيارات لاداء هذه المهمة. فيمكن استخدم الدالة ()remove كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.


>>> letters={'Hassan', 'Ahmad', 'Waleed'}
>>> letters.remove('Ahmad')
>>> letters
{'Hassan', 'Waleed'}
>>>
```

ويمكن ايضا استخدام الدالة ()discard لحذف بيان من مجموعة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> letters={2, (3, 4), 'Hassan', 'Waleed'}
>>> letters.discard(2)
```

```
>>> letters
{(3, 4), 'Hassan', 'Waleed'}
>>>
```

الفرق بين استخدام الدالة remove() والدالة discard() هو ان الدالة discard() لا تعطينا رسالة بوجود خطأ عندما نطلب حذف بيان ليس موجود داخل مجموعة ما ام الدالة remove() فسوف تظهر لنا رسالة خطأ KeyError عند عندم وجود البيان داخل المجموعة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> letters={(3, 4), 'Hassan', 'Waleed'}
>>> letters.discard('Ahmad')   لم تظهر لنا أي رسالة رغم ان البيان غير موجود
>>> letters
{(3, 4), 'Hassan', 'Waleed'}
>>> letters.remove('Ahmad')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Ahmad'
>>>
```

كما يمكن أيضا استخدام الدالة pop() لحذف بيان عشوائي من مجموعة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> letters={(3, 4), 'Hassan', 'Waleed'}
>>> letters.pop()
(3, 4)
>>> letters
{'Hassan', 'Waleed'}
>>>
```

تمرين

استخدم الدالة `pop()` لحذف بيان عشوائي من مجموعة فارغة؟ ماذا تلاحظ؟

كما يمكن استخدام الدالة clear() كما فعلنا بالقوائم والصفوف لحذف جميع البيانات من مجموعة كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> letters={'Hassan', 'Waleed'}
```

```
>>> letters.clear()
>>> letters
set()
>>>
```

ذكرنا سابقا اننا نستطيع إضافة بيان الى مجموعة وكذلك باستطاعتنا حذفه لكن ليس باستطاعتنا إعادة تعيين قيم البيانات داخل المجموعات كما كنا نفعل بالبيانات داخل القوائم. لذلك يجب ان تكون البيانات المدخلة للمجموعة غير قابلة لإعادة التعيين كالبيانات النصية والصفوف والأرقام. وإذا حاولنا إضافة بيان قابل للتغيير كالقوائم مثلا فان بايثون يظهر لنا رسالة بوجود خطأ كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.


>>> letters={'Hassan', 'Waleed'}
>>> letters.add(2)
>>> letters.add((3,4))
>>> letters
{2, (3, 4), 'Hassan', 'Waleed'}
>>> letters.add([2,4,6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

تمرين

استكشف إمكانية إضافة مجموعة الى مجموعة ؟

ويكمن التعرف على ما اذا كان هناك بيان موجود في مجموعة باستخدام الامر in كما فعلنا سابقا في القوائم والصفوف وهذا مثال عليه هنا:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> set_1={1,2,'a','b'}
>>> 1 in set_1
True
>>> 3 in set_1
False
>>>
```

ويمكن اجراء عملية التقاطع بين مجموعتين للحصول على جميع البيانات الموجودة في كلا المجموعتين بعد حذف التكرار باستخدام الدالة union() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> set_1={1,2,'a','b'}
>>> set_2={'c','d',1,2}
>>> set_1.union(set_2)
{1, 2, 'd', 'a', 'b', 'c'}
>>>
```

اما اذا اردنا ان نحصل على البيانات المتشابهة فقط فاننا نستخدم الدالة intersection() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> set_1={1,2,'a','b'}
>>> set_2={'c','d',1,2}
>>> set_1.intersection(set_2)
{1, 2}
>>>
```

وللحصول على البيانات الموجود في مجموعة شريطة ان هذا البيانات غير موجودة في مجموعة ثانية نستخدم الدالة difference() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> set_1={1,2,'a','b'}
>>> set_2={'c','d',1,2}
>>> set_1.difference(set_2)
{'a', 'b'}
>>>
```

تمرين

استخدم المجموعتين في المثال السابق للتعرف على الفرق بين استخدم `set_1.difference(set_2)` واستخدام `set_2.difference(set_1)` ؟

وللحصول على جميع البيانات داخل مجموعتين بعد حذف العناصر المتشابهة نستخدم الدالة symmetric_difference() كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> set_1={1,2,'a','b'}
>>> set_2={'c','d',1,2}
>>> set_1.symmetric_difference(set_2)
{'d', 'c', 'a', 'b'}
>>>
```

تمرين

استخدم المجموعتين في المثال السابق للتعرف على الفرق بين استخدام الدالة `union()` والدالة `symmetric_difference()`؟

كما يمكن استخدام الدالة issubset() للسؤال عن ما اذا كانت مجموعة ما هي مجموعة جزئية من مجموعة أخرى كما في المثال التالي:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> set_1={1,2,'a','b'}
>>> set_2={1, 'd', 2, 'c'}
>>> set_3={'a','b'}
>>>
>>> set_3.issubset(set_1)
True
>>> set_2.issubset(set_1)
False
>>>
```

الفصل الرابع

حلقات التكرار

ان من أكثر بواعث الملل ان يضطر الانسان الى اداء الاعمال ذاتها مرارا وتكرارا. وذلك لأنه يرى في هذه الاعمال هدرا لوقته وجهده. فنجده قد يبدأ بتحرير نفسه من قيود هذه الاعمال بإسنادها الى الات من صنعة. ومن أشهر الآلات التي اعتمد عليها لأداء هذا الغرض جهاز الكمبيوتر. لذلك نجد ان حلقات التكرار المسؤولة عن اداء المهام المتكررة قد اصبحت جزء رئيسي وجوهري من اجزاء لغات البرمجة المشهورة. وفي لغة بايثون يتم عمل المهام المتكررة بواسطة تركيبين لغويين اساسيين هما حلقة for وحلقة while واللتان تعتبران المحوران الرئيسيان لهذه الفصل.

أهداف الفصل

عند اتمام هذا الفصل يجب ان يكون لديك المام بالآتي :

- التعرف على كيفية استخدام حلقة التكرار for.
- التعرف على كيفية استخدام حلقة التكرار while
- التعرف على مواضع التشابه والاختلاف بين مهام كل حلقة

حلقة *for*

تستخدم حلقة for التكرارية عندما يراد اداء مهمة ما عددا معلوماً من المرات. وأفضل طريقة لتفصيل التركيب اللغوي لهذه الحلقة هي اعطاء امثله توضح طريقة عملها فالمثال التالي يقوم بطباعة بيانات موجودة داخل قائمة واحدا تلو الاخر:

code:

```
for x in [1,2,3,4]:
    print(x)
```

output:

```
1
2
3
4
[Finished in 0.1s]
```

من المثال السابق يمكن ملاحظة الاتي:

- تبدأ الحلقة التكرارية for بكلمة for .
- ينتهي السطر الاول من الحلقة بنقطتين فوق بعض ":". ويعتبر نسيان كتابة النقطتين من أشهر الاخطاء التي يقع فيها المبرمجين المبتدئين في لغة بايثون. لذلك احرص على تذكر كتابتها دائماً.
- ترك فراغ في بداية الاسطر التي يراد تكرارها بمقدار حرف واحد على الأقل (Indentation) لإعلام مفسر بايثون بالمهام التي يراد تكرارها ومن المتعارف عليه في لغة بايثون ان يترك فراغ بمقدار 4 أحرف.
- يتم تحديد عدد مرات تكرار الطباعة بعدد البيانات الموجودة في القائمة.

يمكن كتابة المعنى الحرفي للتركيب اللغوي السابق كالاتي: "لكل قيمة يرمز اليها بالمتغير x موجودة في القائمة [1,2,3,4] قم بطباعة قيمة x على الشاشة".

اسم المتغير في هذا المثال اختياري فيمكننا ان نستبدله باي متغير نحب فيمكننا مثلا ان نسميه i فنحصل على نفس النتيجة كما في المثال التالي:

In [3]: code:

```
for i in [1,2,3,4]:
    print(i)
```

output:

```
1
2
3
4
[Finished in 0.1s]
```

إذا أردنا ان نكرر أكثر من مهمة فانه يجب مراعاة ان يكون عدد الفراغات المتروكة في بداية كل سطر متساويا كما في المثال التالي:

code:

```
for x in [1,2,3]:
    print(x)
    print(x**2)
```

output:

```
1
1
2
4
3
9
[Finished in 0.0s]
```

لاحظ انه عند اختلاف عدد المسافات المتروكة في بداية السطر فان بايثون يعطي ملاحظة بوجود خطأ كما في المثال التالي:

code:

```
for x in [1,2,3]:
    print(x)
  print(x**2)
```

output:

```
print(x**2)
          ^
IndentationError: unindent does not match any outer indentation level
[Finished in 0.1s with exit code 1]
```

في المثال السابق استخدمنا عدد البيانات الموجودة في قائمة لتحديد عدد مرات التكرار. وهذه الطريقة ليست الوحيدة لتحديد عدد مرات التكرار فهناك البيانات التجميعية التي ذكرناها في الفصل الثالث يمكن ان نستخدمها ايضا لتحديد عدد مرات التكرار كما في الأمثلة التالية:

code:

```
for x in (1,2,3):       استخدام الصفوف لإجراء عملية التكرار على عناصرها
    print(x)
```

output:

```
1
2
3
[Finished in 0.0s]
```

code:

```
d={1:"a",2:"b",3:"c"}    استخدام القاموس لإجراء عملية التكرار على بياناته
for x,i in d.items():
    print(x,i)
```

output:

```
1 a
2 b
3 c
[Finished in 0.0s]
```

code:

```
name="Ali"              استخدام النص لإجراء عملية التكرار على حروفه
for letter in name:
    print(letter)
```

output:

```
A
l
i
[Finished in 0.0s]
```

كما يمكن استخدام الدالة range() لتحديد عدد مرات التكرار وذلك لأنها تقوم بإنشاء قائمة من اعداد صحيحة كما في المثال التالي:

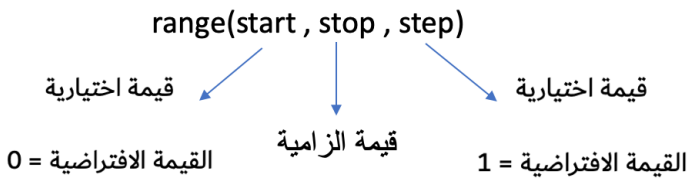
code:

```
for i in range(4):
        print(i)
```

output:

```
0
1
2
3
[Finished in 0.0s]
```

فالدالة range() في الاصل تأخذ ثلاث قيم من الاعداد الصحيحة اثنان منها اختياريان يمكن حذفهما كما في المثال السابق و الثالث متطلب اساسي لأنه يحدد الرقم الذي يجب ان تقف عنده قيم القائمة. فالعددان الاختياريان هما العددان اللذان يحددان بداية القيمة التي تبدأ عندها القائمة والاخر يحدد عدد الخطوات التي تكون بين قيم القائمة فعند حذف القيم الاختيارية يفترض بايثون أنك تريد قائمة من اعداد صحيحة تبدأ من الصفر وتزيد بمقدار العدد 1.



فالمثال السابق قام بايثون بإنتاج قائمة من الاعداد الصحيحة بدأت من الصفر وتزايدت بمقدار 1 وتوقفت قبل الرقم 4. لاحظ ان العدد الذي يحدد توقف القائمة لا يكون مشمولا في القائمة. ويمكن استخدام الدالة range() بثلاث قيم كما في المثال التالي:

code:

```
for i in range(4,10,2):
        print(i)
```

output:

```
4
6
8
[Finished in 0.0s]
```

فالقائمة السابقة بدأت بالعدد 4 وانتهت قبل العدد 10 وكان عدد الخطوات بمقدار 2. كما يمكن ان يكون عدد الخطوات ذو قيمة سالبة لذلك يجب ان تكون قيمة البداية أكبر من النهاية كما في المثال التالي:

code:

```
for i in range(16,4,-3):
        print(i)
```

output:

```
16
13
10
7
[Finished in 0.0s]
```

عند استخدام قيمتين في الدالة ()range فان بايثون يفترض ان القيمة التي حذفت هي قيمة عدد الخطوات كما في المثال التالي:

code:

```
for i in range(2,5):
        print(i)
```

output:

```
2
3
4
[Finished in 0.0s]
```

تمارين

1 - اكتب برانامج يقوم بطباعة الأرقام من 0 الي 10 ؟

2 - اكتب برنامج يقوم بطباعة الاعداد الزوجية من 50 الى 100 ؟

3 - اكتب برنامج يقوم بجمع الاعداد من 1 الى 100 ؟

4 - اكتب برنامج يقوم بطباعة كلمة "Hello" معكوسة؟

5 - اكتب برنامج يقوم بطباعة الاعداد تنازلياً من 20 الي 0 ؟

حلقة *while*

الحلقة التكرارية while تستخدم عندما يكون عدد مرات التكرار التي يريد المبرمج القيام بها غير معروف مسبقا. لذلك تستخدم هذه الحلقة التكرارية عندما نريد ان نقوم بعمليات تكرار غير منتهية او عندما لا يتم معرفة إمكانية الخروج من الحلقة التكرارية الا بعد اجراء بعض العمليات داخل الحلقة التكرارية. ويكون التركيب اللغوي لهذه الحلقة مشابها لحد ما لحلقة for كما في المثال التالي:

code:

```
x=1
while x==1:
        print("hello")
```

output:

```
hello
hello
hello
hello
hello
hello
hello
hello
hello   لإلغاء عملية الطباعة اضغط في آن واحد على ctrl+ C
```

من المثال السابق يمكن ملاحظة الاتي:

- تبدأ حلقة التكرار while بكلمة while.
- استمرارية حلقة التكرار يكون محكوماً بالشرط الذي يتلو كلمة while.
- ينتهي السطر الاول من الحلقة بنقطتين فوق بعض ":".
- يترك مسافة بمقدار حرف واحد على الأقل قبل كتابة الاوامر التي يراد تكرارها.

يمكن كتابة المعنى الحرفي للمثال السابق على النحو التالي: "طالما قيمة x مساوية للعدد واحد فقم بتكرار عملية طباعة كلمة "hello". وبما ان قيمة x لا تتغير داخل حلقة التكرار فان عملية طباعة كلمة "hello" سوف يقوم الكمبيوتر بتنفيذها بعدد غير منتهي من المرات. ولإنهاء عملية الطباعة بشكل اجباري يمكن الضغط على زر ctrl وزر الحرف C في وقت واحد.

ويمكن تحديد وقت الخروج من حلقة while من داخلها كما في المثال التالي:

code:

```
x=1
while x<5:
        print("hello")
        x=x+1
```

output:

```
hello
hello
hello
hello
[Finished in 0.0s]
```

لاحظ اننا قمنا في المثال السابق بتغيير قيمة المتغير x من داخل الحلقة التكرارية وذلك بزيادة قيمته بواحد في كل مرة تتم فيها عملية الطباعة لكلمة "hello". لذلك بعد ان تصبح قيمة المتغير x تساوي 5 عندها يصبح شرط التكرار غير متحقق ويتم الخروج من الحلقة التكرارية.

كما يمكن جعل عمل حلقة تكرار while غير منتهية بجعل شرط الحلقة التكرارية يساوي True او 1 كما في المثالين التاليين:

code:

```
while True:
    print("hello")
```

output:

```
hello
hello
hello
hello
```

code:

```
while 1:
    print("hello")
```

output:

```
hello
hello
hello
hello
```

الفصل الخامس

اتخاذ القرارات

ان من الاشياء الجميلة في لغة البرمجة ان المبرمج يستطيع ان يعطي الكمبيوتر قدرة على التفكير "ان صح التعبير" والتي من خلالها يستطيع الكمبيوتر اتخاذ قراراته. لكن في حقيقة الامر هذه القرارات ماهي الا شروط يفرضها المبرمج على البرنامج ليحقق مبتغاه. في هذا الفصل سوف نتحدث عن كيفية اتخاذ القرارات في بايثون باستخدام اداة الشرط if .

أهداف الفصل

عند اتمام هذا الفصل يجب ان يكون لديك المام بالآتي :

- التعرف على كيفية استخدام أداة الشرط if.
- التعرف على كيفية استخدامelse و elif مع أداة الشرط if.
- التعرف على طريقة كتابة أداة شرط داخل أداة شرط أخرى.

يبتدئ التركيب اللغوي لأداة الشرط بكلمة if متبوعا بالشرط المراد التأكد منه ومن ثم وضع نقطتين فوق بعض في نهاية السطر كما في المثال التالي:

code:

```
name="Ahmad"
if len(name)<10: print(name)
```

output:

```
Ahmad
[Finished in 0.1s]
```

ففي المثال السابق يتم التحقق من صحة كون عدد الاحرف الموجودة في كلمة "Ahmad" اصغر من العد 10. فاذا كانت العبارة صحيحة فان مفسر بايثون يقوم بتنفيذ العبارة التي تتلو النقطتين فقوق بعض. اما إذا كانت العبارة خاطئة فان مفسر بايثون يتجاهل تنفيذ العبارة التي تتلو النقطتين فوق بعض. وبما ان هذه العبارة صحيحة فانه يتم تنفيذ الكود print(name) الذي يتلو النقطتين فوق بعض.

ان كتابة العبارة المراد تنفيذها بعد تحقق الشرط يفضل ان تكون في سطر مستقل بدلا من ان تكون في نفس السطر ولكن إذا جعلت في سطر مستقل فانه يتوجب ترك مسافة بمقدار حرف واحد على الأقل في بداية السطر

الجديد (بفضل ترك مسافة بمقدار أربعة أحرف كما هو متعارف عليه بين مبرمجي بايثون) لكي يتمكن مفسر بايثون من التفرقة بين العبارة المشروطة والعبارة الاعتيادية التي قد تقع بعدها كما في المثال التالي:

code:

```
name="Ahmad"
if len(name)<10:
    print(name)
x=5
print(x)
```

عبارة يتطلب تنفيذها تحقق شرط الجملة الشرطية

عبارة غير داخلة في الجملة الشرطية

عبارة غير داخلة في الجملة الشرطية

output:

```
Ahmad
5
[Finished in 0.1s]
```

لاحظ في المثال السابقة ان ترك مسافة في العبارة المشروطة كان ضروري لتمكين مفسر بايثون من التفرقة بين العبارة الشرطية وما يليها من عبارات أخرى غير مرتبطة بالجملة الشرطية. لذلك كل العبارات التي يتطلب تنفيذها تحقق الشرط يجب ان تترك مسافة في بداية السطر وتكون هذه المسافة هي نفسها لجميع العبارات التي تتطلب تحقق الشرط. المثال التالي يوضح تنفيذ أكثر من عبارة تتطلب تحقق شرط معين:

```
name="Ahmad"
if len(name)<10:

    print(name)
    print(len(name))
    y=0

x=7
print(x)
```

عبارات شرطية

عبارات اعتيادية

تمرين

١-قم بكتابة المثال السابق واجعل واحدة من العبارات المشروطة تبتدأ بعدد مختلف من الفراغات عن العبارتين الأخريين ماذا تلاحظ؟

٢- قم بكتابة جملة شرطية للتعرف على ماذا كان العدد `6` موجود في القائمة `[1,2,3,4]` بحيث يقوم بطباعة هذا الرقم اذا كان الرقم موجود فعلا؟ ماذا تلاحظ؟

لنرفض اننا نريد ان نقوم بعمل ما إذا تحقق شرط معين ونقوم بعمل اخر إذا لم يتحقق الشرط. للقيام بهذه المهمة نحتاج الى استخدام else في الجملة الشرطية لتخبر بايثون بالمراد فعله في حين عدم تحقق الشرط المطلوب كما هو موضح في المثال التالي:

code:

```
 name="Ahmad"
if len(name)<4:
        print(name)
        print(len(name))
        y=0
else:
        print("the name is greater than 4 letters")
```

output:

```
the name is greater than 4 letters
[Finished in 0.0s]
```

لاحظ ان else تخضع لجميع شروط الجملة الشرطية السابقة من حيث انها تنتهي بنقطتين وتتوجب ترك مسافة في بداية العبارات التي يتطلب تنفيذها عدم تحقق الشرط.

تمرين

قم بكتابة جملة شرطية للتعرف على ماذا كان العدد 6 موجود في القائمة [1,2,3,4] بحيث يقوم بطباعة هذا الرقم اذا كان الرقم موجود فعلا؟ واذا لم يكن موجودا اطبع عبارة تفيد ان الرقم غير موجود؟

لنفرض اننا نريد ان نتحقق من توفر شرطين قبل ان نقوم بعمل ما ونقوم بعمل آخر في حال عدم توفر الشرطين. ماذا بإمكاننا ان نصنع في هذه الحالة؟ هناك حلول كثيرة فمنها مثلا ان نستخدم الجملة الشرطية if...else للتحقق من الشرط الاول والثاني باستخدام أداة الربط المنطقي AND وفي حال عدم تحقق الشرطين يتم تنفيذ العبارة التي تلي else كما في المثال التالي:

code:

```
name="Ahmad"
if len(name)<10 and name=="Ahmad":
        print(name)
else:
        print("the name is wrong or big")
```

output:

```
Ahmad
[Finished in 0.0s]
```

كما يمكن القيام بنفس المهمة باستخدام الجملة الشرطية البسيطة if للتحقق من الشرط الاول ومن ثم التحقق من الشرطة الثاني باستخدام الجملة الشرطية if ... else كما في المثال التالي:

code:

```
name="Ahmad"
if len(name)<10:
        if name=="Ahmad":
                print(name)
        else:
```

```
                print("the name is wrong or big")
```

output:

```
Ahmad
[Finished in 0.0s]
```

هذا النوع من الاستخدام للأدوات الشرطية يسمى الأدوات الشرطية الفرعية. بحيث يتم استخدام أداة شرطية داخل أداة شرطية أخرى.

ويمكن استخدام التركيب elif للتحقق من شرط اخر في حال عدم تحقق شرط سابق كما في المثال التالي:

code:

```
name="Ahmad"
if len(name)>10:
    print(name)
elif name=="Ahmad":
    print(name)
else:
    print("the name is wrong or big")
```

output:

```
Ahmad
[Finished in 0.0s]
```

الفصل السادس

كتابة الدوال

في هذا الفصل سوف نتطرق الى معرفة الدوال والتي هي عبارة عن أكود برمجية مستقلة يمكن استدعاءها متى واينما شاء المبرمج. الغرض الاساسي من استخدامها يكمن في تجزئة البرنامج الى اجزاء صغيرة تساعد المبرمج على عدم تكرار الأكواد داخل البرنامج وتسهل عليه عمليه اصلاح الاخطاء.

أهداف الفصل

عند اتمام هذا الفصل يجب ان يكون لديك المام بالآتي :

- ❖ كيفية كتابة دالة وطريقة استدعائها.
- ❖ التعرف على أنواع مدخلات الدوال وكيفية استخدامها.
- ❖ التعرف على مخرجات الدوال.
- ❖ التعرف على طريقة الصحيحة لكتابة تعليقات ارشادية بكيفية استخدام الدوال.

انشاء الدوال

لكتابة دالة في بايثون فإننا نستخدم def في بداية السطر لتبليغ مفسر بايثون باننا نرغب في تعريف دالة جديدة ثم نتبعها باسم الدالة الذي ينتهي بقوسين يوضع بداخلهما مدخلات الدالة وفي نهاية السطر نكتب نقطتين فوق بعض كما في المثال التالي:

```
def function_name():
    pass
```

لاحظ ان في الدالة السابقة كتبنا اسم الدالة ليكون function_name وبإمكانك ان تستخدم أي اسم آخر شريطة ان تنطبق عليه شروط كتابة اسم المتغيرات التي تعرفنا عليها في الفصل الثاني. كما اننا في المثال السابق لم نقم بوضع أي مدخلات داخل قوسي الدالة لأن ذلك الامر اختياري بحسب احتياج الدالة للمدخلات. وأخيراً فان كلمة pass وهي تعنى ان الدالة تم تعريفها فقط و لا تقوم بعمل اي شيء حتى الان. ولكي نجعل الدالة تقوم بعمل ما فإننا نقوم بكتابة الكود البرمجي ليحل محل كلمة pass كما في المثال التالي:

```
def hello():
    print("hello I am inside a function")
```

لاحظ ايضا ان الكود البرمجي داخل الدالة يكتب بعد ترك مسافة على الاقل بقيمة حرف والمتعارف عليه في لغة بايثون هو ترك فراغ بمقدار اربعة أحرف. جميع اسطر الكود البرمجي يجب ان تترك نفس الفراغ والا سوف يعطي مفسر بايثون اشعارا بوجود خطأ.

لا يتم تنفيذ الكود داخل الدالة الا بعد استدعائها وذلك بكتابة اسم الدالة متبوعاً بقوسين بداخلهما مدخلات الدالة ان كان هناك مدخلات اما إذا لم يكن هناك مدخلات فيكتفى بكتابة قوسين فارغين كما في المثال التالي:

code:

```
def hello():
    print("hello I am inside a function")

hello()
```

output:

```
hello I am inside a function
[Finished in 0.1s]
```

كما يجب ملاحظة انه يمكن كتابة تعريف الدالة في اي مكان من البرنامج بشرط ان لا يسبق امر استدعاءها تعريفها عند كتابة البرنامج. وعند اختلال هذا الشرط فان مفسر بايثون يعطي رسالة بوجود خطأ بعدم تعرفه على الدالة كما في المثال التالي:

code:

```
hello()

def hello():
    print("hello I am inside a function")
```

output:

```
Traceback (most recent call last):
    hello()
NameError: name 'hello' is not defined
[Finished in 0.1s with exit code 1]
```

لاحظ ايضا ان في الامثلة السابقة كتبنا قوسي الدالة فارغين لان الدالة لا تتطلب اي معلومات اضافية لتنفيذ الكود البرمجي داخلها. لنفرض الآن اننا نريد كتابة دالة أكثر ذكاء بحيث تقوم بإلقاء التحية على اي شخص ندخل اسمه للدالة. للقيام بهذه المهمة يمكن كتابة الدالة بالطريقة التالية:

```
def hello(name):
    print("Hello "+name)
```

لاحظ عند كتابتنا للدالة السابقة قمنا بإعطاء اسم الشخص المراد القاء التحية عليه كمتغير اسميناه name ووضعناه داخل قوسي الدالة. ومن ثم استخدمنا هذا المتغير في كتابة امر الطباعة. يسمى هذه المتغير مدخل الدالة. وعند استدعاء الدالة ما علينا القيام به هو ادخال الاسم المراد القاء التحية عليه كنص كما في المثال التالي:

code:

```
def hello(name):
    print("Hello "+name)

hello("Ali")
hello("Sara")
```

output:

```
Hello Ali
Hello Sara
[Finished in 0.0s]
```

كما يمكن كتابة دالة بأكثر من مدخل يفصل بينها بفاصلة داخل قوسي الدالة كما في المثال التالي:

code:

```
def addition(x,y):
    print(x+y)

addition(3,4)
```

output:

```
7
[Finished in 0.0s]
```

يجب الاشارة الى ان مدخلات الدالة تنقسم الى ثلاثة أقسام. القسم الاول يسمى مدخلات الزامية (positional arguments) بحيث انه متى تم كتابتها داخل قوسي الدالة عند تعريفها فان الدالة لا يمكن استدعاءها الا بعد تزويدها بهذه المدخلات. فلو قمنا باستدعاء الدالة السابقة بعدد اقل من المدخلات عما تم تعريف الدالة به فسوف يعطينا مفسر بايثون رسالة بوجود خطأ في طريقة استدعاء الدالة كما في المثال التالي:

code:

```
def addition(x,y):
    print(x+y)

addition()
```

output:

```
Traceback (most recent call last):
    addition()
TypeError: addition() missing 2 required positional arguments: 'x' and 'y'
```

كما يجب ملاحظة ان مفسر بايثون يعتمد الترتيب في طريقة كتابة المدخلات حين الاستدعاء لتنفيذ الدوال فمثلا عند استدعاء دالة الجمع السابقة وكتابة addition(3,4) فان مفسر بايثون يعطي x القيمة 3 ويعطي y القيمة 4 عند تنفيذ الدالة. لكن هذا الترتيب غير مهم عندما نستخدم اسم المدخل عند استدعاء الدالة كما في المثال التالي:

code:

```
def addition(x,y):
    print(x+y)

addition(y=5,x=8)
```

output:

```
13
[Finished in 0.0s]
```

والقسم الثاني يسمى مدخلات افتراضية. بحيث تعطى هذه المدخلات قيم افتراضية عند تعريف الدالة باستخدام علامة اليساوي "=". وعند استدعاء الدالة يمكن للمبرمج استبدال القيم الافتراضية بقيم جديدة والا قام مفسر بايثون باستخدام القيم الافتراضية إذا تم تجاهلها من قبل المبرمج وقت الاستدعاء. ولكي نوضح الفرق بين هذين النوعين لننظر للمثال التالي:

```
def addition(x,y=4):
    print(x+y)
```

يعتبر المدخل x الزامي بينما المدخل y اختياري وذلك لأننا قمنا بإعطائه قيمة افتراضية هي العدد 4.

الان إذا أردنا ان نستدعى الدالة السابقة بدون مدخلات فان مفسر بايثون سوف يعطينا اشعارا بوجود خطأ لأنه يوجد مدخل الزامي لم يتم إدخاله حين الاستدعاء كما في المثال التالي:

code:

```
def addition(x,y=4):
    print(x+y)

addition()
```

output:

```
Traceback (most recent call last):
    addition()
TypeError: addition() missing 1 required positional argument: 'x'
[Finished in 0.1s with exit code 1]
```

لنقوم الآن باستخدام مدخل واحد لاستدعاء الدالة كما في المثال التالي:

code:

```
def addition(x,y=4):
    print(x+y)

addition(2)
```

output:

```
6
[Finished in 0.0s]
```

المدخل الذي استخدمناه في المثال السابق هو المدخل الالزامي والذي يمثل قيمة x اما المدخل الاختياري فقد قام مفسر بايثون باستخدام قيمته الافتراضية (4) عند اهمالنا ادخاله وقت استدعائنا للدالة. إذا رغبنا في تغيير القيمة الافتراضية للمدخل y فإننا نقوم بإدخال القيمة وقت استدعاء الدالة بعد ادخال قيمة x اولا كما في المثال التالي:

code:

```
def addition(x,y=4):
    print(x+y)

addition(2,9)
```

output:

```
11
[Finished in 0.0s]
```

يجب ان نشير الى ان مفسر بايثون لا يسمح بتعريف دالة تحتوي على مدخلات الزامية ومدخلات اختيارية الا بترتيب معين. وهو البدء بالمتغيرات الالزامية اولاً ثم المدخلات الاختيارية. وعند تعريف دالة بعكس ذلك فان مفسر بايثون يعطي اشعارا بوجود خطأ كما في المثال التالي:

code:

```
def addition(y=4,x):
    print(x+y)
```

output:

```
def addition(y=4,x):
                ^
SyntaxError: non-default argument follows default argument
[Finished in 0.1s with exit code 1]
```

هناك نوع ثالث من مدخلات الدالة تسمى المدخلات المتغيرة وهي المدخلات التي لا نعرف عددها حين انشاء الدالة. لذلك لتعريف دالة بعدد متغير من القيم نقوم بوضع علامة النجمة (*) قبل اسم المتغير لتعريف مفسر بايثون بأن عدد قيم هذا المدخل غير معروف كما في المثال التالي:

code:

```
def addition(*number):
        total=0
        for i in number:
                total=total+i
        print(total)

addition()
addition(1)
addition(1,2,3,4)
```

output:

```
0
1
10
[Finished in 0.1s]
```

لاحظ ان في المثال السابق وضعنا نجمة قبل اسم المدخل لنخبر مفسر بايثون بان عدد المدخلات غير معروف. كما يجب ملاحظة ايضاً انه عند كتابة كود الدالة استخدمنا أسلوب معين لتعامل مع عدد متغير من المدخلات الا وهو حلقة for. لذلك عند استدعاء الدالة بدون مدخلات تعامل بايثون مع الدالة وأعطى النتيجة 0 وهي قيمة المتغير total. وكذلك قام بالتعامل مع مدخل واحد ومع أربعة مدخلات بدون ان يعطينا أي ملاحظة بوجود خطأ في عدد المدخلات.

لنفرض الان اننا لا نريد طباعة ما تقوم به الدالة على الشاشة ولكن فقط نريد ارجاع ناتج الدالة الى متغير بحيث نستطيع كتابة امر الطباعة متى وكيف ما شئنا. لكي نقوم بهذه المهمة فانه لابد لنا من استخدام الامر return لإخبار الدالة بإرجاع محصلة الكود البرمجي داخلها كما في المثال التالي:

```
def addition(x,y):
    return x+y

sum=addition(3,7)
```

في المثال السابق تم اسناد ناتج الدالة الى المتغير sum ولكي نتأكد ان العملية تمت بنجاح لنقوم بطباعة قيمة sum كما في المثال التالي:

code:

```
def addition(x,y):
    return x+y

sum=addition(3,7)
print(sum)
```

output:

```
10
[Finished in 0.0s]
```

في حقيقة الامر ان الدالة دائما تقوم بإرجاع محصلة الكود حتى لو لم نكتب ما نريد ارجاعه. فاذا لم نستخدم الامر return فان الدالة تقوم بإرجاع قيمة افتراضية هي None وتعني انه لم يتم ارجاع اي شيء. كما في المثال التالي:

code:

```
def addition(x,y):
    z=x+y

sum=addition(5,9)
print(sum)
```

output:

```
None
[Finished in 0.0s]
```

لاحظ ان الدالة السابقة قامت بجمع المدخلين واسناد النتيجة الى المتغير z وبما اننا لم نستخدم الامر return لإرجاع قيمة المتغير z فان الدالة قامت بإرجاع القيمة الافتراضية None. هذه النقطة تقودنا الى ما يسمى بمدى المتغيرات داخل الدوال. فالمتغير z في الدالة السابقة سوف يمسح من الذاكرة بمجرد الانتهاء من استدعاء الدالة لذلك تسمى هذه المتغيرات بالمتغيرات المحلية (local variables) اما المتغيرات التي تقع خارج الدوال فتسمى متغيرات عامة (global variables) بحيث تحتفظ بقيمتها في اي مكان من البرنامج حتى في داخل الدوال كما في المثال التالي:

code:

```
x=5
def addition(y):
    print(x+y)

addition(4)
```

output:

```
9
[Finished in 0.0s]
```

لكن هناك امر مهم يجب ان تعرفه وهو انه لا يمكننا تغيير قيم المتغيرات العامة داخل الدوال الا بعد استخدام كلمة global قبل اسم المتغير المراد تغيير قيمته كما في المثال التالي:

code:

```
x=5
def change_x(y):
    global x
```

```
    x=y

change_x(3)
print(x)
```

output:

```
3
[Finished in 0.0s]
```

لاحظ ان كلمة global في المثال السابق ابلغت مفسر بايثون ان هذا المتغير عام ويجب الاحتفاظ بقيمته حتى بعد الانتهاء من مهمة الدالة. اما إذا لم نستخدم كلمة global فان مفسر بايثون سوف يفترض ان x متغير محلى وسوف يمحو قيمته من الذاكرة بعد الانتهاء منها ولا يحدث اي تغيير للمتغير العام كما في المثال التالي:

code:

```
x=5
def change_x(y):
    x=y

change_x(4)
print(x)
```

output:

```
5
[Finished in 0.1s]
```

التعليقات الارشادية للدوال

ان اول خطوة يقوم بها المبرمج لتسهيل فهم الدالة التي يقوم بإنشائها هي اختيار اسم مناسب لها بحيث يكون هذا الاسم عنوانا لما تقوم به الدالة. لكن في بعض الأحيان يتطلب الامر كتابة اسما طويلا للدالة فيصعب بذلك عملية الاستدعائها لذلك قد يكتفى بكتابة اسم مختصر للدالة ومن ثم يتم كتابة تعليق بداخلها لشرح طريقة عملها وما تقوم به من مهام. فنجد ان دليل بايثون الارشادي رقم PEP 8 الخاص بتوحيد طريقة تنسيق وكتابة الأكواد البرمجية وجعلها سهلة القراءة لكافة المبرمجين ينص على انه يجب كتابة ملخص على شكل تعليق او ملاحظة في السطر الذي يتلو اسم الدالة بحيث يشير فيه المبرمج على الوظيفة التي تقوم بها الدالة كما في المثال التالي:

```
def add_two_numbers(x,y):
    '''تقوم هذه الدالة بحساب مجموع الرقمين المدخلين وارجاع حاصل جمعهما'''
    return x+y
```

يمكن التعرف على ملخص وظيفة الدالة إذا تم كتابته عند تعريف الدالة باستخدام الامر "__doc__" بعد كتابة اسم الدالة متبوعا بنقطة كما في المثال التالي:

code:

```
def add_two_numbers(x,y):
    '''تقوم هذه الدالة بحساب مجموع الرقمين المدخلين وارجاع حاصل جمعها'''
    return x+y
print(add_two_numbers.__doc__)
```

output:

```
تقوم هذه الدالة بحساب مجموع الرقمين المدخلين وارجاع حاصل جمعها
[Finished in 0.1s]
```

تمرين

1. تعرف على التعليمات الارشادية للدالة `len()` التي تعلمناها سابقاً لاعطائنا عدد البيانات في قائمة او عدد الحروف في نص؟
2. تعرف على التعليقات الارشادية الخاصة بالدالة `print()` ؟
3. تعرف على التعليقات الارشادية بالدالة `type()` ؟

التعامل مع الملفات

ان جميع البرامج التي قمنا بكتابتها حتى الان كانت تتطلب ادخال البيانات يدويا داخل الكود البرمجي قبل تشغيله. وهي طريقة بدائية الى حد ما وغير مفيدة خاصة عندما نرغب في ادخال بيانات جديدة غير التي ادخلناها قبل تشغيل البرنامج. لذلك استحدث المبرمجون تركيب لغوي يمكن من ادخال البيانات بعد تشغيل البرنامج وهو على النحو التالي:

code:

```
num=input("please enter a number: ")
print("you entered: ",num)
```

output:

```
please enter a number:
```

فالتركيب اللغوي هذا يطلب من المستخدم بعد تشغيله للبرنامج ان يقوم بإدخال قيمة المتغير num وذلك من خلال كتابة ملاحظة او رسالة يكتبها المبرمج بين قوسي الدالة على شكل نص لترشد المستخدم الى ما يجب عمله. ويبقى البرنامج منتظراً للمستخدم حتى يدخل قيمة ويضغط على زر الادخال ليكمل بعدها البرنامج تنفيذ ما تبقى من كود برمجي.

يجب ملاحظة ان كل ما يقوم بإدخاله المستخدم من بيانات في هذا التركيب اللغوي يتم التعامل معه كسلسلة نصية. ويمكن التأكد من هذه المعلومة باستخدام الدالة type() كما في المثال التالي:

code:

```
num=input("please enter a number: ")
print(type(num))
```

output:

```
please enter a number: 10
<class 'str'>
```

لاحظ انه بعد ادخال الرقم 10 والضغط على زر الادخال طبع لنا البرنامج نوع البيان الذي ادخلناه وهو من النوع النصي "str".

وكمثال آخر لنفرض اننا نريد كتابة برنامج يطلب من المستخدم ادخال قيمة رقمين x و y. لكي يقوم بحساب حاصل جمعهما. فمن خلال تعلمنا للتركيب اللغوي السابق يمكن كتابة البرنامج بالطريقة التالية:

code:

```
x=input("please enter a value for x: ")
y=input("please enter a value for y: ")
sum=x+y
print("The sum of x and y = ",sum)
```

output:

```
please enter a value for x: 5
please enter a value for y: 9
The sum of x and y = 59
```

عند تنفيذ البرنامج السابق وإدخال قيمة 5 للمتغير x وقيمة 9 للمتغير y نجد ان هناك مشكلة في البرنامج. فالبرنامج تعامل مع المدخلات على انها سلسلة نصية وتمت عملية الجمع كما تعلمنا سابقا بتجميع النصوص مع بعضها البعض. يمكن حل المشكلة هذه بتحويل سلسلة النصوص المدخلة الى ارقام كما في المثال التالي:

code:

```
x=input("please enter a value for x: ")
y=input("please enter a value for y: ")
sum=float(x)+float(y)
print("The sum of x and y = ",sum)
```

output:

```
please enter a value for x: 5
please enter a value for y: 9
The sum of x and y = 14.0
```

تمرين

طور البرنامج السابق باستخدام حلقة `while` بحيث تجعل البرنامج يسأل المستخدم بعد اجراء عملية الجمع عما إذا كان يرغب في اجراء عملية جمع أخرى او يرغب في انهاء البرنامج.

على الرغم من ان طريقة ادخال البيانات السابقة تعتبر مفيدة وتفاعلية الا انها تصبح مملة ومجهدة لأنها مازالت تتطلب التدخل البشري في أدائها وخصوصاً عندما يتطلب الامر ادخال بيانات كثيرة. لذلك تعتبر طريقة قراءة الملفات من أسهل وأشهر الطرق البرمجية استخداما. ولغة بايثون تستخدم الدالة open() للتعامل مع الملفات سواء كان الغرض من ذلك قراءة ملف او الكتابة عليه. وتحتاج هذه الدالة مدخل الزامي واحد وهو عنوان موقع الملف على القرص الصلب اما بقية المدخلات الأخرى فهي مدخلات افتراضية. فعند استدعاء الدالة بالمدخل الالزامي فقط فان مفسر بايثون سوف يفترض ان المبرمج ينوى القراءة من ملف كما في المثال التالي:

```
file=open("data.txt")
```

عند تنفيذ الكود البرمجي السابق يتكون في ذاكرة الكمبيوتر كائن برمجي يمثل قناة تواصل بين القرص الصلب الذي يحوي الملف ومفسر بايثون ويشار الى هذا الكائن بالمتغير file. لاحظ ان ذكر اسم الملف لا يكفي للوصول الى الملف الا إذا كان ملف البرنامج والملف المطلوب القراءة منه موجودان في نفس المكان. أما إذا كان اسم

الملف المراد القراءة منه موجود في مكان آخر فان مفسر بايثون يتطلب من المبرمج كتابة العنوان الكامل لموقع الملف على القرص الصلب. كما يجب ملاحظة ايضاً ان كتابة عنوان الملف دائما يكون على هيئة نص أي بمعنى انه يجب ان يكون بين علامتي تنصيص.

ان تنفيذ الكود البرمجي السابق لا يعنى ان عملية القراءة قد تمت وانما يعنى ان عملية تكوين الكائن البرمجي الذي يتعامل مع الملف المطلوب قد تمت. وإذا كانت هناك مشكلة في اسم او عنوان الملف عند تنفيذ الكود البرمجي السابق فان مفسر بايثون سوف يعطى رسالة بوجود خطأ في تكوين كائن الارتباط لعدم العثور على الملف كما في المثال التالي:

code:

```
file=open("data.txt")
```

output:

```
Traceback (most recent call last):
    file=open("data.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'data.txt'
[Finished in 0.0s with exit code 1]
```

للقيام بعملية القراءة بعد نجاح تكوين كائن الارتباط نستخدم الدالة ()read لقراءة كافة محتويات الملف بعد اسم كائن الارتباط مفصولا بنقطة كما في المثال التالي:

code:

```
file=open("data.txt")
content=file.read()
print(content)
```

output:

```
1,2,3,4,5
[Finished in 0.0s]
```

في المثال السابق قمنا بإنشاء ملف نصي في نفس المكان الذي يتواجد فيه البرنامج واسميناه data.txt وكتبنا بداخله بعض القيم النصية. ثم قمنا بتكوين كائن ربط عن طريق استخدام الدالة ()open واسمينا هذا الكائن file. بعد ذلك قمنا بعملية القراءة من الملف النصي وجعلنا محتويات الملف النصي في متغير اسميناه content. في الخطوة الأخيرة قمنا بطابعة قيمة المتغير content والتي تمثل محتويات الملف النصي.

كما يجب الإشارة الى ان هناك دوال أخرى تقوم بعملية القراءة من ملف ولكن بخصائص تختلف قليلاً عن الدالة ()read. فمثلاً يمكن استخدام الدالة ()readline لقراءة سطر واحد من الملف كما في المثال التالي:

code:

```
file=open("data.txt")
line=file.readline()
print(line)
```

output:

```
1,2,3,4,5
[Finished in 0.0s]
```

في المثال السابق قمنا بإنشاء ملف اسمه data.txt وكتبنا داخله ثلاثة اسطر كل سطر يحتوي على خمسة ارقام وقمنا بقراءة سطر احد فقط باستخدام الدالة readline() وطباعة نتيجة القراءة على الشاشة.

كما يمكن قراءة الملف السابق بأكمله باستخدام الدالة readlines() (لاحظ s الجمع للتفريق بين الدالة السابقة والدالة التي نحن بصددها الآن) بحيث يكون ناتج القراءة عبارة عن قائمة من البيانات كل بيان فيها يمثل محتوى كل سطر من الملف الذي تتم قراءته كما في المثال التالي:

code:

```
file=open("data.txt")
lines=file.readlines()
print(lines)
```

output:

```
['1,2,3,4,5\n', '6,7,8,9,10\n', '11,12,13,14,15']
[Finished in 0.1s]
```

تمرين

1. قم بكتابة برنامج يقرأ من ملف نصي اسمه `numbers.txt` موجود في نفس المكان الذي يتواجد فيه البرنامج ويحتوي على الاعداد من `1` الى `10` في السطر الاول و `11` الى `20` في السطر الثاني و `21` الى `30` في السطر الثالث. حاول ان تقرأ محتويات الملف باستخدام الدوال الثلاث السابقة وطباعتها على الشاشة؟
2. حاول التعرف على نوع البيانات الناتجة عملية القراءة بالدوال الثلاث السابقة باستخدام الدالة `type()`؟

لنفرض الان اننا نريد الكتابة على ملف ماذا بوسعنا ان نغير في الدالة open() للقيام بهذه المهمة؟

في حقيقة الامر ان الكتابة على ملف تتطلب القيام بتغيير قيمة النمط (mode) الافتراضية في الدالة open() والتي تمثل المدخل الثاني الذي يلي عنوان موقع الملف كما في المثال التالي:

```
file=open("data.txt",”w”)
```

عند كتابة ”w“ كمدخل ثاني في الدالة open() فان مفسر بايثون سوف يقوم بإنشاء كائن ارتباط للكتابة بدلاً من كائن القراءة الافتراضي. لكن يجب ملاحظة ان هناك اختلاف بسيط بين نمطي الكتابة والقراءة. فنمط الكتابة عند

عدم وجود الملف المحدد فان مفسر بايثون يقوم بإنشائه تلقائيا بينما نمط القراءة يعطي رسالة بوجود خطأ في حال عدم العثور على الملف.

ودالة الكتابة في بايثون هي ()write. ويتم التعامل معها تقريبا بنفس الطريقة التي استخدمناها مع دالة القراءة كما هو موضح في المثال التالي:

```
file=open("data.txt","w")
file.write("Hello python")
```

فبعد انشاء كائن الارتباط للكتابة نستخدم الكائن file مع الدالة ()write لتنفيذ عملية الكتابة على الملف والتي هي هنا عبارة عن نص وضع بين قوسي الدالة ()write. ويمكن كذلك استخدام الدالة ()writelines وذلك لكتابة قائمة من البيانات كما في المثال التالي:

```
file=open("data.txt","w")
file.writelines(["Welcome ","to ", "python ", "world!"])
```

تمرين

لتاكد ان عمليتي الكتابة في التمرينين السابقين تمتا بشكل صحيح. قم بتنفيذ كل كود ومن ثم قم بكتابة كود برمجي اخر يقوم بقراءة وطباعة ما تم كتابته؟

يجب ملاحظة ان فتح ملف بغرض الكتابة عليه يختلف بحسب النمط المدخل. فالنمط "w" يعنى مسح محتويات الملف السابقة ان وجدت والكتابة على الملف من بدايته. اما إذا استخدمنا النمط "a" كمدخل ثاني في الدالة ()open فان ذلك يعنى ان محتويات الملف السابقة لا يتم مسحها وانما تتم الكتابة من حيث ما انتهى محتوى الملف السابق. وهناك أنماط أخرى لا يسع الحديث عنها هنا ويمكن البحث عنها من خلال محركات البحث للاستزادة.

ان فتح ملف سواء للقراءة او الكتابة بالطريقة السابقة قد يسبب بعض إشكالية في نفاذ جزء من مساحة الذاكرة وذلك لان الملف الذي تم فتحه سوف يضل في ذاكرة الكمبيوتر حتى يتم اغلاقه. لذلك ان استخدام الطريقة السابقة للتعامل مع الملفات يتطلب تنظيف الذاكرة الانتهاء من عملية القراءة الو الكتابة باستخدام الدالة ()close كما في المثال التالي:

```
file=open("data.txt")
content=file.read()
file.close()
```

ولتلافي نسيان اغلاق الملفات بعد الانتهاء من اجراء العمليات عليها يمكن استخدام تركيب لغوي آخر في لغة بايثون يعتمد على استخدام البادئة with والذي من خلاله يقوم مفسر بايثون بإغلاق الملف المفتوح بمجرد الانتهاء من هذه المهمة كما في المثال التالي:

code:

```
with open("data.txt") as file:
    content=file.read()
    print(content)
```

output:

```
Welcome to python world!
[Finished in 0.0s]
```

لاحظ ان في المثال السابق تم استخدام الاتي:

- بدء التركيب اللغوي باستخدام كلمة with.
- استخدام كلمة as لإسناد كائن الربط الى المتغير file.
- استخدام النقطتين فوق بعض ":" عند نهاية السطر الاول.
- ترك مسافة في الاسطر التي تلي السطر الاول للقيام بالعمليات المطلوبة على الملف المفتوح.
- مجرد كتابة سطر برمجي لا يبتدأ بترك مسافة يخبر مفسر بايثون بأن عملية التعامل مع الملف قد انتهت فيقوم بإقفال الملف ومحوه من الذاكرة.

تمرين

قم باعادة كتابة تمارين القراءة والكتابة السابقة باستخدام التركيب اللغوي `with`؟

الفصل الثامن

التعامل مع المكتبات (modules)

في هذا الفصل سوف نتطرق الى الحديث عن المكتبات (modules) في لغة بايثون والتي تعتبر من أحد أهم العوامل التي ساهمت في ذياع صيت لغة بايثون على كافة الأصعدة. فعلى سبيل المثال نجد مكتبات numpy و scipy و matplotlib مشهورة الاستخدام في الاوساط العلمية والبحثية. اما مكتبتي Django و flask فهما مشهورتان لدى مطوري تطبيقات الويب. وتعتبر مكتبة pandas مشهورة لدى المهتمين بتحليل البيانات الى غير ذلك من المكتبات الاخرى.

سنحاول في هذا الفصل تبسط مفهوم المكتبات في بايثون لدى القارئ حيث سنبدأ بتعريف المكتبات وكيفية التعامل معها. وسنقوم أيضاً بتعريف القارئ ببعض المكتبات المبنية داخل اصدارة بايثون 3 والتي يكاد لا يستغنى عنها عند كتابة أي كود برمجي بلغة بايثون. كما اننا سوف نتعرف على كيفية تنصيب مكتبة خارجية الى مجلد مكتبات بايثون من اجل استخدامها في كتابة البرامج. وفي نهاية الفصل سوف نتطرق بشكل مبسط الى كيفية انشاء مكتبة خاصة بنا.

أهداف الفصل

عند اتمام هذا الفصل يجب ان يكون لديك المام بالآتي :

- تعريف المكتبات في لغة بايثون وكيفية استدعائها.
- استخدام بعض المكتبات المبنية داخل اصدارة بايثون.
- معرفة كيفية تنصيب مكتبة خارجية داخل مجلد مكتبات بايثون لاستخدامها في كتابة البرامج.
- التعرف على كيفية بناء مكتبة خاصة.

المكتبة في بايثون هي عبارة عن ملف بايثون اعتيادي يحتوي على كود برمجي تم من خلاله تعريف عدد من الكائنات البرمجية والدوال والمتغيرات بهدف تسهيل أداء مهام متعلقة بموضوع معين. فنجد على سبيل المثال المكتبة math والتي هي عبارة مكتبة داخلية مبنية في اصدارة بايثون تحوي على دوال رياضية عديدة كدالة الجذر التربيعي ()sqrt وادالة جيب الزاوية ()sin وجيب تمام الزاوية ()cos الى غيرها من الدوال الرياضية. ولكي نتمكن من استخدام الدوال الموجودة داخل أي مكتبة يجب ان يتم استدعاؤها من خلال الامر import كما في المثال التالي:

code:

```
import math
x=math.sqrt(4)
print(x)
```

output:

```
2.0
[Finished in 0.1s]
```

لاحظ ان عملية الاستدعاء تمت باستخدام الامر import متبوعاً باسم المكتبة. ومن ثم أصبح بإمكاننا استخدام الدوال داخل المكتبة math باستخدام اسم المكتبة أولاً ومن ثم اسم الدالة بعد الفصل بينها بنقطة. وطريقة الاستدعاء هذه ليست الوحيدة في بايثون. فهناك طرق استدعاء أخرى. فمثلا يمكن استدعاء المكتبة السابقة باستخدام التركيب اللغوي التالي:

code:

```
from math import *
x=sqrt(4)
print(x)
```

output:

```
2.0
[Finished in 0.1s]
```

لاحظ ان طريقة الاستدعاء في المثال السابق يمكن ترجمتها الى المعنى التالي: "من المكتبة math استورد كل الدوال والمتغيرات." وعلامة النجمة بعد كلمة import تعني استيراد كل شيء. كما يجب ملاحظة انه عند استخدام طريقة الاستدعاء هذه فإننا لا نستخدم اسم المكتبة قبل الدالة وانما نكتفى باستخدام الدالة مباشرة. لكن في حقيقة الامر فان مبرمجي بايثون لا يوصون باستخدام طريقة الاستدعاء هذه لسببين: اولها ان هذه الطريقة تستدعي كل شيء في المكتبة والتي قد لا نحتاج الا لجزء يسير من الدوال الموجود في المكتبة. فتؤدي هذه الطريقة الى حشو ذاكرة الكمبيوتر بما لا فائدة منه. ثاني هذه الاسباب هو انه قد يحصل تضارب بين اسماء الدوال الموجودة في المكتبة المستدعاه وأسماء دوال جديدة يمكن ان نقوم بإنشائها بأنفسنا. فمثلا عندما نستورد المكتبة math بالطريقة السابقة ثم نقوم بتعرف دالة جديدة تحمل اسم أحد الدوال الموجودة في المكتبة math فانه يحصل ارتباك لدى المبرمج حول اية دالة تم استخدامها عند تنفيذ البرنامج كما في المثال التالي:

code:

```
import math
def sqrt(number):
    return "Hello "+str(number)
x=sqrt(4)
print(x)
```

output:

```
Hello 4
[Finished in 0.0s]
```

على الرغم ان من يحدد ايتها دالة سوف يستدعيها مفسر بايثون عند تنفيذه الكود البرمجي السابق محكوم بمكان تعريف الدالة الجديدة التي قمنا بإنشائها داخل الكود البرمجي الا ان طريقة الاستدعاء هذه تسبب ارباك لكل من يقرأ الكود البرمجي السابق.

كما يمكن تحديد الدوال المراد استخدامها حين الاستدعاء بحيث لا يتم استدعاء دوال المكتبة بالكامل بل يكتفى باستدعاء الدوال المحددة فقط كما في المثال التالي:

code:

```
from math import sin, pi
print(sin(0.5*pi))
```

output:

```
1.0
[Finished in 0.0s]
```

كما يمكن تغيير اسم المكتبة حين الاستدعاء بغرض تبسيط كتابة الكود البرمجي كما في المثال التالي:

code:

```
import math as m
print(m.sin(0.5*m.pi))
```

output:

```
1.0
[Finished in 0.0s]
```

مكتبات بايثون الداخلية

تحتوي اصدارة بايثون 3 على العديد من المكتبات الداخلية (built-in modules). وسوف نتطرق في هذه الجزئية من الفصل الي ثلاثة من هذه المكتبات وهي os و sys و أخيرا time. اما بقية المكتبات فيمكن استخدام الدالة help() والتي تعطي تعريف وشروحات وافيه عن أي مكتبة داخلية تكتب داخل قوسي الدالة كما في المثال التالي:

code:

```
import os
help(os)
```

output:

```
Help on module math:

NAME
```

```
    math

MODULE REFERENCE
    https://docs.python.org/3.6/library/math

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the inverse hyperbolic cosine of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

    asinh(...)
        asinh(x)

        Return the inverse hyperbolic sine of x.

    atan(...)
        atan(x)

        Return the arc tangent (measured in radians) of x.
```

لاحظ اننا قمنا بحذف جزء كبير من ناتج الكود السابق وذلك لأننا نرغب فقط في توصيل فكرة انه يمكن استخدام الدالة help() مع أي مكتبة داخلية للتعرف على ما يمكن ان تقوم به المكتبة من مهام.

كما يمكن أيضا التعرف على محتويات أي مكتبة بعد استدعائها باستخدام الدالة dir() كما في المثال التالي:

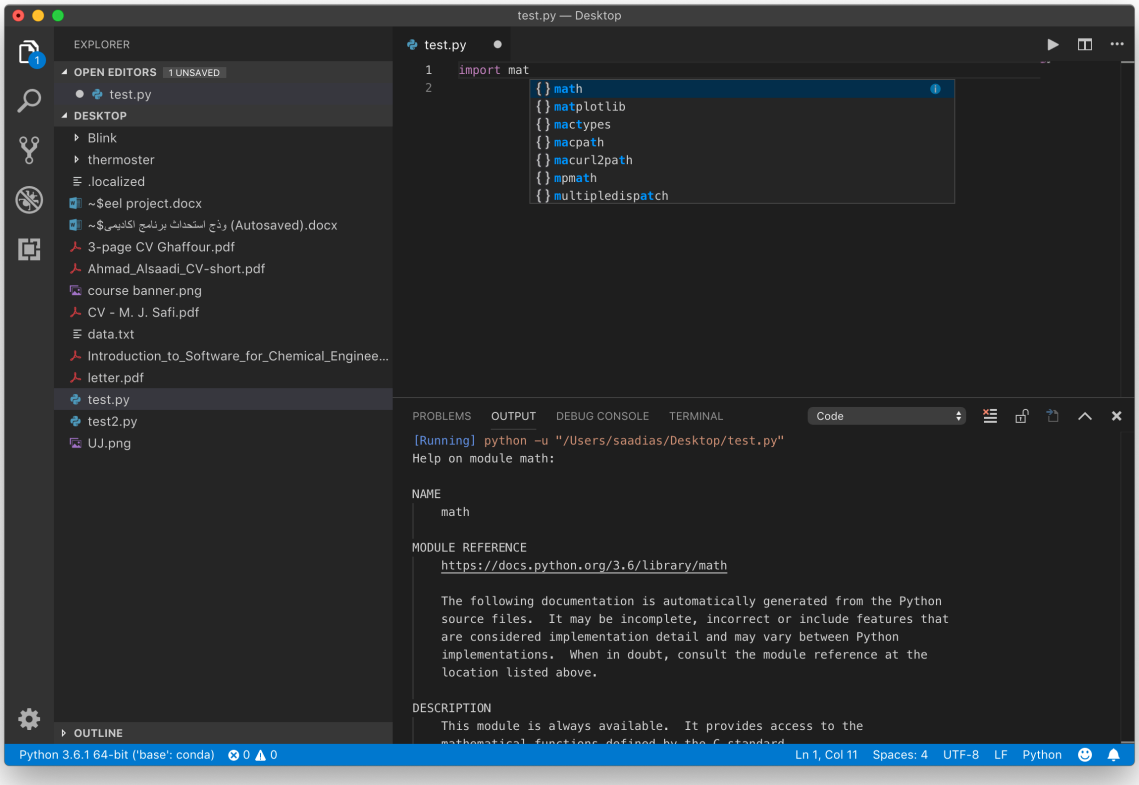
```
import math
print(dir(math))
```

output:

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd',
'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
[Finished in 0.1s]
```

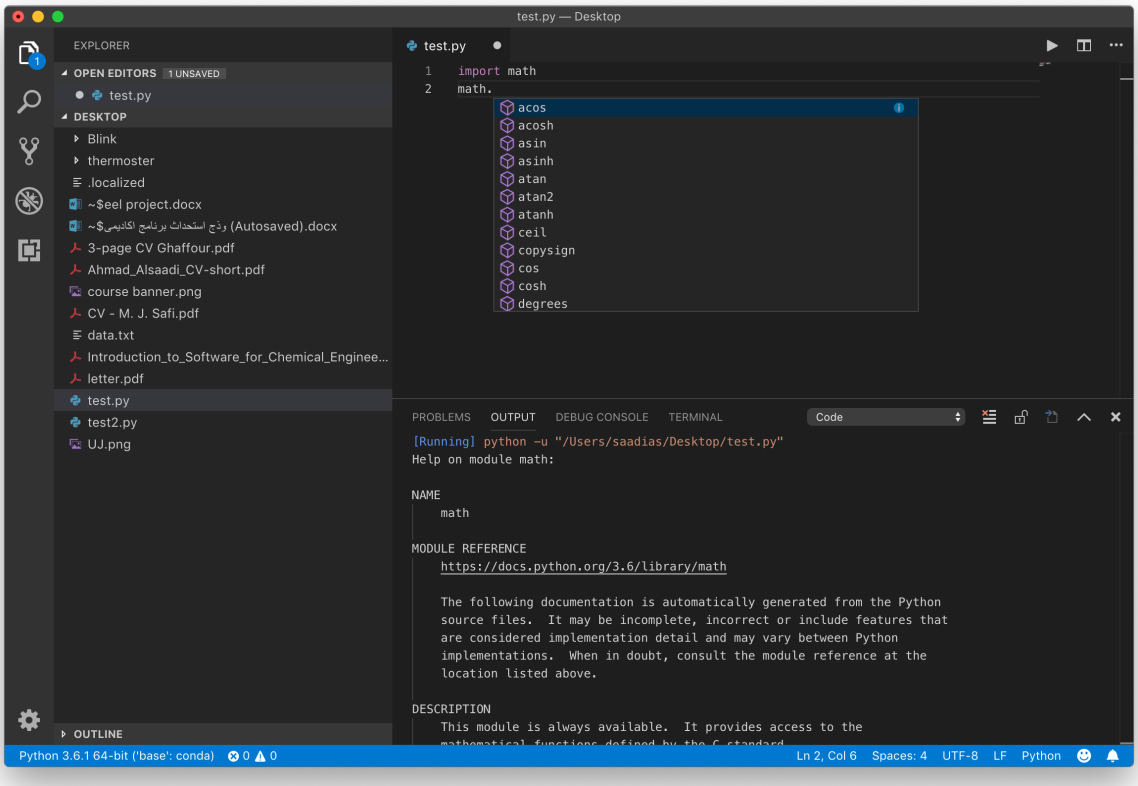
لاحظ ان ناتج الدالة dir() هو عبارة قائمة تحوي جميع دوال ومتغيرات المكتبة math. ولو دققت النظر في بيانات القائمة السابقة لوجدت ان الدوال التي تعلمناها من هذه المكتبة math موجودة بالفعل داخل هذه القائمة وقد قمنا بوضع خط تحت هذه الدوال في المثال السابق.

كما يمكن معرفة محتويات المكتبة بعد استدعائها بشكل تلقائي عندما نستخدم برنامج مختص بتطوير البرامج (Integrated Development Enviroment) كفجول ستوديو كود visual studio code او باي تشارم PyCharm كما في المثال التالي والمقتبس من برنامج "فجول ستوديو كود" بعد إضافة أدوات تطوير لغة بايثون اليه:



فعند كتابة الامر import في المكان المخصص لكتابة الكود البرمجي يقوم "فيجول ستوديو كود" باقتراح قائمة بالمكتبات التي يمكنك ان تستدعيها. وكلما كتبت أحرف أكثر من اسم المكتبة التي تريد استدعائها كلما كانت

اقتراحات البرنامج أكثر دقة لما تنوي استيراده. كما ان خاصية الاكمال الذاتي هذه تساعد المبرمج على مشاهدة محتويات المكتبة من دوال ومتغيرات. فبمجرد كتابة اسم المكتبة واضافة نقطة بعدها يبدأ البرنامج بعرض الدوال والمتغيرات في المكتبة لكي يساعدك على العثور على ما تبحث عنه كما في المثال التالي:



مكتبة os

تسهل مكتبة os التفاعل مع نظام التشغيل الذي يعمل عليه المبرمج سواء كان ذلك نظام ويندوز او ماك او لينكس. وتقدم هذه المكتبة الكثير من الدوال الخاصة بالتعامل مع نظام التشغيل فمثلا يمكن معرفة المجلد الحالي الذي يتواجد فيه البرنامج الذي نعمل على كتابة كوده باستخدام الدالة getcwd() كما في المثال التالي:

code:

```
import os
print(os.getcwd())
```

output:

```
/Users/saadias/Desktop
[Finished in 0.1s]
```

كما يمكن أيضا تنفيذ جميع الاوامر التي يمكن تنفيذها على محرر الاوامر في ويندوز او ماك او لينكس من خلال كود بايثون البرمجي باستخدام الدالة system() بحيث يوضع الامر المراد تنفيذه بين علامتي تنصيص داخل قوسي الدالة كما في المثال التالي:

code:

```
import os
os.system("echo Hello")
```

output:

```
Hello
[Finished in 0.0s]
```

وكذلك أيضا يمكن تكوين مجلد باستخدام الدالة mkdir() وكذلك حذف مجلد باستخدام rmdir() وايضاً يمكن تغيير موقع المجلد باستخدام الدالة chdir() وعرض قائمة بأسماء المجلدات باستخدام الدالة listdir() والكثير الكثير من الدوال الخاصة بالتعامل مع الملفات والمجلدات والتي لا يسع المقام لذكرها هنا ويمكن الاطلاع عليها من خلال وسائل المساعدة التي ذكرناه سابقاً.

مكتبة sys

تمكن مكتبة sys المبرمج بلغة بايثون من التعامل مع كل ما يخص مفسر بايثون. فمثلا يمكن معرفة رقم اصدارة بايثون عن طريق استخدام المتغير version كما في المثال التالي:

code:

```
import sys
print(sys.version)
```

output:

```
3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)]
[Finished in 0.0s]
```

كما يمكن معرفة المجلدات التي سوف يقوم مفسر بايثون بالبحث فيها عند اجراء اي عملية استدعاء سواءً كانت لمكتبة او دالة وذلك باستخدام المتغير path كما في المثال التالي:

code:

```
import sys
print(sys.path)
```

output:

```
['/Users/saadias/Desktop', '/Users/saadias/anaconda/lib/python36.zip',
'/Users/saadias/anaconda/lib/python3.6',
'/Users/saadias/anaconda/lib/python3.6/lib-dynload',
'/Users/saadias/anaconda/lib/python3.6/site-packages',
```

```
'/Users/saadias/anaconda/lib/python3.6/site-packages/Sphinx-1.5.6-py3.6.egg',
'/Users/saadias/anaconda/lib/python3.6/site-packages/aeosa']
[Finished in 0.0s]
```

يجب ملاحظة ان ناتج الكود السابق يختلف بحسب اختلاف إعدادات النظام لكل مستخدم وطريقة ونوعية تنصيبه لإصدارة بايثون. فأول مكان يقوم مفسر بايثون بالبحث فيه هو المجلد الذي تم تشغيل الكود البرمجي منه ثم مجلد المكتبات الداخلية (يختلف باختلاف إصدارة بايثون المستخدمة) ومن ثم مجلد المكتبات الخارجية site-packages.

كما يمكن استخدام المتغير argv في مكتبة sys للتعرف على المدخلات التي تصاحب ملف بايثون عند تشغيله من محرر أوامر النظام الذي يعمل عليه المستخدم. ولتوضح هذه الفكرة أكثر لنفرض ان لدينا ملف بايثون اسمه test.py وكتبنا بداخله الكود البرمجي التالي:

```
import sys
print(sys.argv)
```

لنقم الان بتشغيل الملف السابق من محرر الاوامر سواءً كان ذلك في ويندوز او في غيره من الانظمة. فمن خلال محرر الاوامر في ماك يمكن استخدام الطريقة التالية:

```
Ahmads-MacBook-Pro:Desktop saadias$ python test.py
['test.py']
```

بعد كتابة الامر python test.py والضغط على زر الادخال يتم تشغيل الملف الذي كتبناه بلغة بايثون والذي يقوم بطباعة قيمة المتغير argv. المتغير argv الذي تم طباعته هنا هو عبارة عن قائمة تحتوي على اسم الملف الذي قمنا بتشغيله. ويمكن إضافة بيانات الى المتغير argv وذلك بكتابتها بعد اسم الملف عند التشغيل كما في المثال التالي:

```
Ahmads-MacBook-Pro:Desktop saadias$ python test.py 1 2 3
['test.py', '1', '2', '3']
Ahmads-MacBook-Pro:Desktop saadias$
```

لا حظ ان كل شيء يكتب بعد اسم الملف حين تشغيله يتعامل معه مفسر بايثون على انه مدخل يتم اضافته الى قائمة argv ويجب ملاحظة ان الفصل بين البيانات يتم بواسطة ترك مسافة بين كل مدخل وآخر كما هو واضح في المثال السابق. وتعتبر طريقة الادخال هذه شائعة الاستخدام للبرامج التي يتم تشغيلها من محرر أوامر النظام.

تمرين

قم بكتابة برنامج يستغل خاصية ادخال البيانات عند التشغيل بحيث يقوم بجمع الاعداد المدخلة وطباعة ناتج الجمع على شاشة الكمبيوتر؟

مكتبة time

مكتبة time الداخلية تحتوي على العديد من الدوال التي تساعد المبرمج على القيام بمهام متعلقة بالوقت. فمثلا يمكن استخدام الدالة time() لمعرفة الوقت بعدد الثوان التي مضت منذ الساعة 12 صباحاً من اليوم الاول لشهر يناير لعالم 1970 كما في المثال التالي:

code:

```
import time
print(time.time())
```

output:

```
1546516361.497566
[Finished in 0.1s]
```

قد لا تكون صيغة الوقت هذه مفيدة للكثير لكن يمكن تحويلها تحويل هذه الثواني صيغة وقت مألوف باستخدام الدالة ctime() كما في المثال التالي:

code:

```
import time
t=time.time()
print(time.ctime(t))
```

output:

```
Thu Jan  3 14:58:50 2019
[Finished in 0.1s]
```

الدالة ctime() تعطي صيغة الوقت المألوفة عندما نزودها بعدد الثوان التي مرت منذ ذلك التاريخ لكن عندما نستخدمها بدون أي مدخلات فإنها تعطينا الوقت الحالي بالصيغة المألوفة كما في المثال التالي:

code:

```
import time
print(time.ctime())
```

output:

```
Thu Jan  3 14:58:50 2019
[Finished in 0.1s]
```

كما يمكن تأجيل تنفيذ أي سطر برمجي يقع بعد الدالة sleep() لعدد معين من الثوان توضع بين قوسي الدالة كما في المثال التالي:

code:

```
import time
print(time.ctime())
time.sleep(5)
print(time.ctime())
```

output:

```
Thu Jan  3 15:10:54 2019
Thu Jan  3 15:10:59 2019
[Finished in 5.1s]
```

كما يجب عليك ملاحظة ان معظم دوال المكتبة time تتعامل مع كائن برمجي للوقت ينتج من تنفيذ بعض الدوال بحيث يكون تكوين هذا الكائن البرمجي مجزأ الى أجزاء الوقت المعروفة من سنة وشهر ويوم وساعة الى ...اخ. بحيث تسهل على المبرمج التعامل مع أجزاء الوقت وكتابتها بالطريقة التي يرغب بها. فعند استدعاء الدالة localtime() يتم ارجاع كائن الوقت الخاص بالمكان الحالي وكذلك الدالة gmtime() تقوم بارجاع كائن خاص بوقت جرينتش كما في المثال التالي:

code:

```
import time
Makkah_time=time.localtime()
London_time=time.gmtime()
print(Makkah_time)
print(London_time)
print(Makkah_time.tm_hour)
```

output:

```
time.struct_time(tm_year=2019, tm_mon=1, tm_mday=3, tm_hour=15, tm_min=35,
tm_sec=51, tm_wday=3, tm_yday=3, tm_isdst=0)
time.struct_time(tm_year=2019, tm_mon=1, tm_mday=3, tm_hour=12, tm_min=35,
tm_sec=51, tm_wday=3, tm_yday=3, tm_isdst=0)
15
[Finished in 0.1s]
```

تمرين

قم بكتابة برنامج يقوم بحساب فارق التوقيت بين مكة ولندن وطباعة الناتج على شاشة الكمبيوتر؟

كما يمكن إعادة صياغة الوقت بالطريقة التي نرغب بها باستخدام الدالة strftime() كما في المثال التالي:

code:

```
import time

Makkah_time = time.localtime()
formatted_time = time.strftime("%m/%d/%Y, %H:%M:%S", Makkah_time)

print(formatted_time)
```

output:

```
01/03/2019, 15:41:50
[Finished in 0.1s]
```

في المثال السابق تم وضع صيغة الوقت المطلوبة على شكل نص كمدخل اول للدالة وكائن الوقت وضع كمدخل ثان. ولحل شفرة رموز شفرة الوقت السابقة يمكن الاستعانة بالجدول التالي:

الرمز	المعنى
%Y	السنة من 4 ارقام
%y	السنة في رقمين
%m	الشهر
%d	اليوم
%H	الساعة
%M	الدقيقة
%S	الثانية

طريقة تنصيب مكتبة خارجية

في الإصدارات الحديثة من بايثون تم دمج الأداة المستخدمة لإضافة المكتبات الخارجية مع مفسر بايثون بحيث ان هذه الأداة تكون متوفرة بمجرد تنصيب اصدارة بايثون. تختلف هذه الأداة بحسب التوزيعة المستخدمة. فالأداة pip تأتي مع الاصدارة الأساسية. اما توزيعة Anaconda فلها أداة تنصيب خاصة بها تسمى conda تؤدي نفس وظائف الأداة pip. ولشرح طريقة استخدام pip لنستعرض أولا المكتبات التي تم تنصيبها داخل اصدارة بايثون وذلك باستخدام الامر pip list من محرر أوامر النظام كما في المثال التالي بحسب نظام ماك:

```
Ahmads-MacBook-Pro:Desktop saadias$ pip list

Package                            Version
---------------------------------- -----------
alabaster                          0.7.10
anaconda-client                    1.6.3
anaconda-navigator                 1.6.2
anaconda-project                   0.6.0
appnope                            0.1.0
appscript                          1.0.1
asn1crypto                         0.22.0
```

فالمثال السابق قام بطباعة كافة المكتبات المنصبة على اصدارة بايثون وارقام اصداراتها.

فاذا كانت المكتبة التي ترغب باستخدامها ليست موجود في هذه القائمة فان بإمكانك تنصيبها عن طريق استخدام الامر pip install متبوعاً باسم المكتبة المطلوبة كما في المثال التالي:

```
Ahmads-MacBook-Pro:Desktop saadias$ pip install progressbar
Collecting progressbar
  Retrying (Retry(total=4, connect=None, read=None, redirect=None,
status=None)) after connection broken by
'ReadTimeoutError("HTTPSConnectionPool(host='pypi.org', port=443): Read timed
out. (read timeout=15)",)': /simple/progressbar/
  Downloading
https://files.pythonhosted.org/packages/a3/a6/b8e451f6cff1c99b4747a2f7235aa90
4d2d49e8e1464e0b798272aa84358/progressbar-2.5.tar.gz
Building wheels for collected packages: progressbar
  Running setup.py bdist_wheel for progressbar ... done
  Stored in directory:
/Users/saadias/Library/Caches/pip/wheels/c0/e9/6b/ea01090205e285175842339aa3b
491adeb4015206cda272ff0
Successfully built progressbar
Installing collected packages: progressbar
Successfully installed progressbar-2.5
Ahmads-MacBook-Pro:Desktop saadias$
```

في المثال السابق قمنا بتنصيب المكتبة progressbar والتي تعتبر من المكتبات الصغيرة بحيث تساعد المبرمج في تكوين كائن مرئ يوضح مدى التقدم في انجاز مهمة ما. وعند اكتمال التنصيب تعطي الأداة pip رسالة بنجاح عملية التنصيب للمكتبة كما هو موضح باللون الداكن أعلاه. المثال التالي يوضح طريقة استخدام هذه المكتبة بعد تنصيبها:

```
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from time import sleep
>>> from progressbar import ProgressBar
>>> bar=ProgressBar()
>>> for i in bar(range(50)):
...     sleep(0.5)
...
100%
|########################################################################|
>>>
```

اما إذا كانت المكتبة منصبة لديك من قبل وترغب فقط بتحديثها فانه بإمكانك استخدام خيار التحديث كما في المثال التالي:

```
Ahmads-MacBook-Pro:Desktop saadias$ pip install --upgrade progressbar
Requirement already up-to-date: progressbar in
/Users/saadias/anaconda/lib/python3.6/site-packages (2.5)
Ahmads-MacBook-Pro:Desktop saadias$
```

حيث اعطانا الامر السابق ان مكتبة progressbar المنصبة على هي أحدث اصدارة موجودة ولا تحتاج الى تحديث. اما إذا كانت الاصدارة قديمة فان pip سوف يقوم بتنصيب اصدارة المكتبة الاحدث من موقع https://pypi.org.

اما الأداة conda فإنها تعمل تقريبا بنفس الطريق التي تعمل بها الأداة pip. لذلك اعتقد انه لا داعي للحديث عنها هنا ويمكن استخدام محرك البحث قوقل لتعرف عليها أكثر.

طريقة انشاء مكتبة

كما ذكرنا في بداية الفصل مكتبات بايثون هي مجرد ملف بايثون اعتيادي ينتهي بالحرفين py. ويحتوي على كائنات برمجية ودوال ومتغيرات خاصة بموضوع معين. ولكي نوضح سهولة طريقة انشاء أي مكتبة لنفرض اننا بصدد انشاء مكتبة خاصة بتحويل الوحدات الطولية (متر، سنتيمتر، مليمتر، مايكرومتر) ولنفرض ان اسمها meter. للقيام بهذه المهمة سوف نفتح ملف جديد ونحفظه باسم meter.py ثم نبدأ بكتابة الكود البرمجي التالي بداخله:

```
'''This module has been built to convert between the different
   units of length'''
def to_cm(number):
        '''This function converts meter to centimeter'''
        return number*100

def to_mm(number):
        '''This function converts meter to millimeter'''
        return number*1000

def to_micro(number):
        '''This function converts meter to micrometer'''
        return number*10e6

def from_cm(number):
        '''This function converts centimeter to meter'''
        return number/100

def from_mm(number):
        '''This function converts millimeter to meter'''
        return number/1000
def from_micro(number):
        '''This function converts micrometer to meter'''
        return number/10e6
```

الآن لنفتح ملف بايثون جديد ونحاول استدعاء المكتبة meter التي قمنا بإنشائها. يجب ملاحظة ان ملف المكتبة التي قمنا بإنشائها والملف الذي سوف نستدعي من خلاله المكتبة يجب ان يكونا في مجلد واحد لكي يعثر مفسر بايثون على المكتبة كما تعلمنا سابقاً حين استعرضنا مكتبة sys ودالتها ()path.

code:

```
import meter
print(meter.to_cm(4))
```

output:

```
400
[Finished in 0.1s]
```

كما تلاحظ من المثال السابق ان المكتبة التي قمنا بإنشائها تعمل كأي مكتبة موجودة داخل اصدارة بايثون. فمثلا يمكن ان نتعرف على الشروحات التي كتبناها عن المكتبة بين علامة التنصيص الثلاثية باستخدام المتغير __doc__ كما يلي:

code:

```
import meter
print(meter.__doc__)
```

output:

```
This module has been built to convert between the differet
   units of length
[Finished in 0.0s]
```

كما يمكن استعراض محتويات المكتبة meter باستخدام الدالة ()dir كما يلي:

code:

```
import meter
print(dir(meter))
```

output:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'from_cm', 'from_micro', 'from_mm',
'to_cm', 'to_micro', 'to_mm']
[Finished in 0.0s]
```

لاحظ ان المتغيرات التي تبتدأ بشرطتين وتنتهي بشرطتين لم نقم بكتابتها داخل المكتبة ولكن بايثون قام بإنشائها بشكل تلقائي.

كما يجب ملاحظة ان ملف المكتبة التي انشأناها ليس بالضرورة ان يكون في نفس المجلد الذي سوف نستدعي منه المكتبة. فعندما نرغب في وضع المكتبة في مكان اخر يجب علينا ان نخبر مفسر بايثون بان يبحث في المجلد

الذي وضعت فيه المكتبة وذلك من خلال إضافة هذا المجلد الي القائمة التي سوف يبحث فيها مفسر بايثون عن المكتبة كما في المثال التالي:

code:

```
import sys
sys.path.append('/Users/saadias/Desktop')

import meter
print(meter.to_cm(5))
```

output:

```
500
[Finished in 0.1s
```

لاحظ اننا قمنا بإضافة المجلد الذي يحتوي على المكتبة meter الى قائمة الاماكن التي يبحث فيها مفسر بايثون عن المكتبات بواسطة الدالة ()append قبل ان نقوم باستدعاء المكتبة meter.

كما يمكن أيضا إضافة المكتبة الى مجلد خاص تم تحديده من قبل مفسر بايثون ليكون خاص بمكتبات المستخدم. ولتعرف على هذا المجلد نقوم باستدعاء المتغير USER_SITE من مكتبة site الداخلية كما في المثال التالي:

code:

```
import site
print(site.USER_SITE)
```

output:

```
/Users/saadias/.local/lib/python3.6/site-packages
[Finished in 0.1s]
```

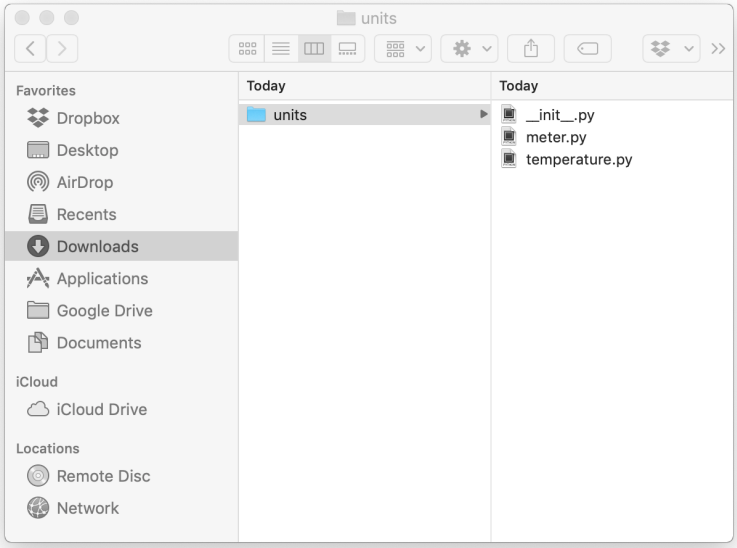
المجلد الذي ينتج عن تنفيذ الكود السابق يختلف باختلاف اصدارة بايثون المستخدمة ويختلف أيضا باختلاف نظام التشغيل المستخدم. وفي غالب الأحيان لا يكون هذا المجلد موجود ويجب علينا انشاءه ومن ثم وضع المكتبة التي انشأناها بداخله لكي يتم التعرف عليها.

تمرين

قم بتطوير مكتبة `meter` لكي تقوم بالتحويل من سنتيمتر الى ميليمتر والعكس ومن سنتيمتر الى مايكرومتر والعكس؟

وقبل ان نختم هذا الفصل نود نبين انه يمكن عمل رزمة من المكتبات package والتي هي عبارة عن مجموعة من المكتبات التي يحويها مجلد واحد. فعند انشاء مكتبة جديدة لتحويل درجات الحرارة وليكن اسمها مثلا temperature.py بحيث توضع هذه المكتبة الي جانب المكتبة meter السابقة في مجلد واحد وليكن اسم المجلد

units فان مفسر بايثون يشترط للتعرف على هذه المكتبات ان يقوم المبرمج بإنشاء ملف فارغ داخل هذا المجلد بحيث يكون اسم هذا الملف __init__.py كما هو موضح بالشكل التالي:



كما يجب ملاحظة ان الشروط اللازمة لعثور مفسر بايثون على مكتبة معينة تنطبق على رزمة المكتبات كما وضحنا سابقاً. ويمكن استدعاء المكتبة meter من رزمة المكتبات كما يلي:

code:

```
import sys
sys.path.append('/Users/saadias/Downloads')

from units import meter
print(meter.from_cm(100))
```

output:

```
1
[Finished in 0.0s
```

الفصل التاسع

البرمجة الكائنية في بايثون

الفصل العاشر

برمجة واجهة المستخدم

الملاحق

تحويل الاعداد الثنائية الى عشرية والعكس

الفصل الثامن

المنسقات

المنسقات هي عبارة عن دوال يمكنها استدعاء دوال وارجاع دوال كمخرجات لهذه الدوال. لاحظ المثال التالي للتوضيح

In [218]:

لنقوم الان بتعريف دالة بسيطة تقوم بجمع رقمين وارجاع ناتج الجمع. كما تحتوي هذه الدالة على ملاحظات تفيد وظيفة الدالة كما في المثال التالي:

In [164]:

لاحظ انه عند استخدام الامر __doc__ على الدالة add فإنها تقوم بإظهار الملاحظات تم كتابتها لتوضيح وظيفة الدالة كما في المثال التالي:

In [165]:

this function add two numbers and return their sum

In [166]:

In [167]:

this function add two numbers and return their sum

تم تنسيق الدالة بواسطة دالة منسقة

لاحظ ان الدالة المنسقة decorated_by اخذت الدالة add كمدخلات وقامت بتعديل الملاحظات الخاصة بالدالة بحيث اضافة عليه سطر اخر من الملاحظات واعادة الدالة بعد التعديل عليها. لذلك عندما قمنا بطباعة ملاحظات الدالة المعدلة وجدنا التعديل قد ظهر في امر الطابعة الثاني.

ان عملية تعديل دالة واعادة تسميتها بنفص الاسم قد يسبب بعض التشويش في ذهن المبرمج لذلك تم استحداث طريقة سلهة لاحداث التعديل على الدالة دون اللجوء الى تغيير اسمها وذلك باستخدم الرمز @ متعبوعا باسم الدالة المنسقة فوق الدالة المراد التعديل عليها كما في المثال التالي:

In [180]:

In [181]:

this function add two numbers and return their sum

تم تنسيق الدالة بصورة اسهل

In [ ]:

ترتيب المنسقات

يمكن تطبيق اكثر من دالة منسقة على دالة اخرى كما في المثال التالي:

In [185]:

this function add two numbers and return their sum

المنسقة الاولى

المنسقة الثانية

وما يجب ملاحظته من المثال السابق هو طريقة تطبيق المنسقات يبدا من الاسفل الى الاعلى. فلقد تم تطبيق الدالة المنسقة decorated_by_one اولا ثم تم تطبيق الدالة المنسقة الثانية .decorated_by_two لذلك يجب التنبه الى عملية الترتيب هذه.

تمرين: قم بعكس ترتيب المنسقات في المثال السابق بحيث تصبح المنسقة decorated_by_two تقع مباشرة فوق الدالة add ثم ضع المنسقة decorated_by_one فوق المنسقة الثانية ولاحظ الفرق بين نتيجة هذا التغيير ونتيجة المثال السابق؟

اين يمكننا استخدام المنسقات

مكتبة بايثون الاساسية تحتوي على منسقات فمثلا عندما نريد ان نستخدم دالة في مصنف لا يحتاج الى على عمل نسخة من المصنف فإننا نستخدم المنسقة @classmethod او .@staticmethod كما ان هناك اطارات عمل مثل flask تستخدم المنسقة @app.route للربط بين الدوال و عناوين الصفحات عندما تستدعى من المتصفح. وكذلك اطار العمل Django والذي يستخدم المنسقة @login_required للتأكد من تسجيل الدخول قبل الدخول الى صفحات الموقع. كما يمكن استخدام المنسقات عندما نرغب بالتأكد من ملاءمة مدخلات دالة ما من حيث القيمة والنوع كما في المثال التالي:

In [188]:

Out[188]:

{}

أهمية المنسقات

استخدام المنسقات يعتبر طريقة ذكية في كتابة الكود البرمجي فهي تساعد على سهولة قراءته واكتشاف الاخطاء به. كما ان استخدام المنسقات يجعل الكود البرمجي مستقل الاجزاء بحيث يمكن التعديل على الدالة باضافة او ازالة دون المساس بالكود البرمجي الموجود داخل الدالة المراد التعديل عليها.

In [ ]: