

الفصل الأول

مقدمة

1-1- مقدمة عن البايثون:

بايثون (بالإنجليزية: Python) هي لغة برمجة، عالية المستوى سهلة التعلم مفتوحة المصدر قابلة للتوسيع، تعتمد أسلوب البرمجة الكائنية (OOP). لغة بايثون هي لغة مُفسَّرة، ومُتعدِّدة الاستخدامات، وتستخدم بشكل واسع في العديد من المجالات، كبناء البرامج المستقلة باستخدام الواجهات الرسومية وفي تطبيقات الويب. بشكل عام، يمكن استخدام بايثون لعمل البرامج البسيطة للمبتدئين، ولإنجاز المشاريع الضخمة في الوقت نفسه. غالباً ما يُنصح المبتدئون في ميدان البرمجة بتعلم هذه اللغة لأنها من بين أسرع اللغات البرمجية تعلماً.

1-2- لمحة تاريخية عن البايثون:

طُوِّرت بايثون في معهد الرياضيات والمعلوماتية الهولندي (CWI) في مدينة أمستردام على يد جايدو فان روسم في أواخر ثمانينات القرن العشرين، وكان أول إعلان عنها في عام 1991م. كُتبت نواة اللغة باستعمال لغة سي. أطلق روسم الاسم «بايثون» على لغته تعبيراً عن إعجابه بفِرقة مسرحية هزلية شهيرة من بريطانيا، كانت تطلق على نفسها اسم مونتي بايثون.

تتميز بايثون بمجتمعها النشط، كما أن لها الكثير من المكتبات البرمجية ذات الأغراض الخاصة التي يرمجها أشخاص من ذلك المجتمع. مثلاً، هناك مكتبة باي جايم التي توفر مجموعة من الدوال من أجل برمجة الألعاب. يمكن لبايثون أيضاً التعامل مع العديد من أنواع قواعد البيانات مثل mysql وغير ذلك.

1-3- استخدامات لغة البايثون:

من الاستخدامات:

- تطوير الويب (من جانب الخادم).
- تطوير البرمجيات.
- الرياضيات.
- البرمجة النصية للنظام.

ماذا يمكن أن تفعل بايثون؟

- يمكن استخدام بايثون على الخادم لإنشاء تطبيقات الويب.
- يمكن استخدام Python جنبًا إلى جنب مع البرامج لإنشاء مهام سير العمل.
- يمكن لبايثون الاتصال بأنظمة قواعد البيانات. ويمكنه أيضًا قراءة الملفات وتعديلها.
- يمكن استخدام بايثون للتعامل مع البيانات الضخمة وإجراء العمليات الحسابية المعقدة.
- يمكن استخدام بايثون للنماذج الأولية السريعة، أو لتطوير البرمجيات الجاهزة للإنتاج.
- لماذا بايثون؟
- تعمل لغة Python على منصات مختلفة (Windows، Mac، Linux، و Raspberry Pi، وما إلى ذلك).
- لدى Python بناء جملة بسيط مشابه للغة الإنجليزية.
- لدى بايثون بناء جملة يسمح للمطورين بكتابة برامج ذات أسطر أقل من بعض لغات البرمجة الأخرى.
- يمكن التعامل مع بايثون بطريقة إجرائية، أو بطريقة موجهة للكائنات، أو بطريقة وظيفية.

انتهى الفصل الأول

الفصل الثاني

مقارنة البايثون مع لغات البرمجة المعاصرة

2-1- مميزات البايثون:

لغة البرمجة Python تتميز بعدة ميزات هامة تجعلها شديدة الشعبية ومفيدة لمجموعة واسعة من التطبيقات. أقدم لك بعض الميزات الهامة التي تميز Python عن باقي لغات البرمجة:

1. قابلية القراءة والكتابة:

Python تتمتع ببنية بسيطة وقواعد صارمة، مما يجعل الشفرة سهلة القراءة والكتابة. هذا يساعد في تسهيل التعاون بين المطورين وفهم الشفرة حتى للأشخاص الذين لا يعرفون Python بشكل كبير.

2. تعدد الاستخدامات:

يمكن استخدام Python في مجموعة واسعة من التطبيقات بما في ذلك تطوير الويب، والحوسبة العلمية، والتحليل البياني، والذكاء الاصطناعي، وتطوير تطبيقات السحابة، وأكثر من ذلك.

3. مجتمع نشط ومكتبات غنية:

يتمتع Python بمجتمع مطورين كبير ونشط، مما يعني وجود العديد من المكتبات والإطارات الجاهزة التي تسهل على المطورين إنشاء تطبيقاتهم بسرعة وكفاءة.

4. توافق مع معايير صناعية:

Python يلتزم بمعايير صناعية قياسية، مما يجعلها مناسبة للاستخدام في العديد من الصناعات والتطبيقات التجارية.

5. البرمجة الديناميكية:

Python تعتبر لغة ديناميكية، مما يعني أن المتغيرات لا تحتاج إلى تعريف أنواعها مسبقاً، ويمكن تغيير نوع المتغير أثناء تشغيل البرنامج. هذا يجعل البرمجة أكثر مرونة وسهولة.

6. التوجيه الكائني:

Python تدعم التوجيه الكائني (OOP)، مما يسمح للمطورين بتنظيم الشفرة بشكل أفضل وإعادة استخدام الأكواد بشكل فعال.

7. الواجهات البسيطة:

Python يسهل تعلمه واستخدامه، وهو مناسب للمبتدئين بسبب بنيته البسيطة والتي تشبه اللغة الإنجليزية.

8. ميزة التكامل:

Python يتكامل بشكل جيد مع لغات أخرى مثل C و ++C، ويوفر واجهات برمجة تطبيقات (API) للتفاعل مع العديد من التقنيات والخدمات.

9. متعدد المنصات:

Python يمكن تشغيله على معظم أنظمة التشغيل الرئيسية (Windows، Linux، macOS)، مما يعني أن التطبيقات المكتوبة بـ Python قابلة للتشغيل على مختلف البيئات.

10. التطوير السريع:

Python يسمح بتطوير سريع وفعال، حيث يمكن للمطورين بناء تطبيقاتهم بسرعة باستخدام المكتبات الجاهزة والإطارات.

تلك هي بعض الميزات الهامة التي تجعل لغة البرمجة Python مميزة وشائعة في مجال تطوير البرمجيات.

2-2- البايثون واللغات الأخرى:

لغة البرمجة Python تختلف عن العديد من لغات البرمجة الأخرى، مثل HTML, C++, JavaScript و Java، من حيث الصفات والميزات. فيما يلي بعض الفروق البارزة بين لغة البرمجة Python وبعض اللغات الأخرى:

1. التركيز على القراءة والكتابة:

Python تعتبر من لغات البرمجة سهلة القراءة والكتابة، مما يجعلها مثالية للمبتدئين وتسهل على المطورين فهم الشفرة.

2. الكفاءة في التطوير:

Python تعزز فعالية التطوير بفضل خصائصها الديناميكية وسهولة التفاعل مع الكود، مما يقلل من وقت التطوير بشكل عام.

3. النمط التفاعلي:

Python يقدم مفسراً تفاعلياً يتيح للمستخدمين تجربة الأوامر فوراً، وهو أمر مفيد خلال عملية التطوير والاختبار.

4. النموذج الدينامي:

Python هي لغة برمجة ديناميكية، حيث لا يلزم تعريف النوع مسبقاً، مما يسهل على المطورين كتابة الشفرة بشكل أسرع.

5. الإدارة الآلية للذاكرة:

Python يتمتع بإدارة ذاكرة آلية، مما يقلل من الحاجة إلى التحكم اليدوي في الذاكرة مقارنة بلغات مثل C++.

6. الوسوم والتنسيق:

HTML وتقنيات الويب الأخرى (مثل JavaScript و CSS) تستخدم وسوم لتحديد هيكل الصفحة وتنسيقها، بينما Python يستخدم تنسيق الهوية البيانية لتنظيم الشفرة.

7. البرمجة الشيئية:

C++ و Java تشجعان على البرمجة الشيئية، في حين أن Python تدعم البرمجة الشيئية وتسمح أيضاً بأسلوب برمجيات أخرى مثل البرمجة الإجرائية.

8. الأداء:

لغات مثل C++ توفر أداءً عالي المستوى، بينما Python تتمتع بأداء معقول ولكن ليس بنفس مستوى الأداء الذي يقدمه C++.

9. المجتمع والدعم:

Python لديها مجتمع كبير ونشط من المطورين، مما يعزز من توفر مكتبات وأدوات مفيدة.

10. الاستخدامات:

Python يستخدم على نطاق واسع في مجالات مثل التطوير الويب والذكاء الاصطناعي، بينما C++ يستخدم في تطوير البرامج ذات الأداء العالي مثل الألعاب.

تلك هي بعض الفروق الرئيسية بين Python وبعض لغات البرمجة الشهيرة الأخرى. يجب على المطور اختيار اللغة التي تناسب احتياجات مشروعه بناءً على المتطلبات والأهداف المحددة.

2-2-1- البايثون والـ C++ :

هناك العديد من الفروق بين لغة البرمجة Python ولغة السي بلس بلس (C++). فيما يلي بعض الفروق الرئيسية بينهما:

1. النمط البرمجي:

Python هي لغة برمجة تفسيرية (interpreted) وتدعم البرمجة الديناميكية، حيث يتم تحليل وتنفيذ الشفرة خطوة بخطوة. بينما C++ هي لغة برمجة تجميعية (compiled) وتدعم البرمجة الثابتة وتحتاج إلى ترجمة قبل تنفيذها.

2. إدارة الذاكرة:

في ++C، يتحكم المبرمج تمامًا في إدارة الذاكرة، مما يعني أنه يجب عليه القيام بعمليات تخصيص وتحرير الذاكرة يدويًا. بينما في Python، يتمتع جاز التخزين الضمني (garbage collector) بالقدرة على إدارة تلك العمليات تلقائيًا، مما يقلل من خطأ الإدارة اليدوية للذاكرة.

3. الكتابة والقراءة:

Python تُعتبر أكثر سهولة في الكتابة والقراءة مقارنة ب ++C، حيث تتيح للمبرمجين كتابة كميات أقل من الشفرة لتحقيق نفس الوظائف.

4. التوجيه الكائني:

كلاهما يدعم مفاهيم البرمجة الشيئية (OOP)، ولكن الدعم في ++C قوي ومبني بشكل أساسي حول التوجيه الكائني، بينما يُعتبر التوجيه الكائني في Python جزءًا من اللغة ولكن ليس بنفس القوة.

5. الأداء:

++C عادةً ما تكون أدائها أفضل من Python، خاصة في تطبيقات تتطلب أداءً عاليًا، مثل الألعاب أو البرامج ذات الحسابات الكبيرة.

6. استخدامات مختلفة:

++C تستخدم على نطاق واسع في تطبيقات النظم، البرمجة المضمنة، وتطوير الألعاب، بينما يستخدم Python بشكل شائع في التطوير الويب، الحوسبة العلمية، والتطبيقات الخفيفة.

7. مكتبات وإطارات العمل:

Python تتمتع بمجموعة واسعة من المكتبات والإطارات الجاهزة التي تساعد في تسريع عملية التطوير، بينما ++C يتطلب في بعض الأحيان كتابة المزيد من الشفرة لتحقيق نفس الغرض.

8. سهولة التعلم:

Python عادةً ما تكون أسهل للتعلم والاستفادة منها للمبتدئين مقارنة ب ++C، حيث تقدم بنية بسيطة وصفورية. يُلاحظ أن اختيار اللغة يعتمد على احتياجات المشروع وتفضيلات المبرمج، وكل لغة لها ميزاتها واستخداماتها الملائمة.

2-2-2- البايثون و ال JavaScript :

هناك العديد من الفروق بين لغة البرمجة Python ولغة الجافا سكريبت (JavaScript). فيما يلي بعض الفروق الرئيسية بينهما:

1. الاستخدام:

Python غالبًا ما يُستخدم في مجالات مثل التطوير الويب (خاصة باستخدام إطار العمل Django)، الحوسبة العلمية، والتطبيقات العامة. بينما يُستخدم JavaScript أساسًا لبرمجة تفاعل المستخدم في صفحات الويب وتطوير الواجهات الرسومية.

2. الموقع الذي يتم تنفيذ الكود فيه:

Python يُعتبر لغة خادم (Server-side language) حيث يتم تنفيذ الكود على الخادم. في المقابل، JavaScript هو لغة عميل (Client-side language) تنفذ على متصفح الويب لديك.

3. النموذج البرمجي:

Python تدعم البرمجة الشيئية (OOP) والبرمجة الإجرائية، بينما JavaScript تدعم أيضًا البرمجة الشيئية وتعتبر أساسًا للبرمجة الشيئية في سياق تطوير الويب.

4. التنفيذ:

Python يحتاج إلى مفسر (interpreter) لتنفيذ الشفرة، بينما يتم ترجمة وتنفيذ شفرة JavaScript مباشرة على متصفح الويب.

5. مكتبات وإطارات العمل:

Python تحظى بمجموعة كبيرة من المكتبات والإطارات الجاهزة مثل Django و JavaScript. Flask. يستفيد من العديد من المكتبات والإطارات لتطوير واجهات المستخدم وتفاعل المستخدم، مثل React و Angular.

6. الاستخدام خارج المتصفح:

Python يمكن استخدامه في تطبيقات السطح المكتبية والحوسبة العلمية وأغراض أخرى، بينما يُستخدم JavaScript بشكل أساسي داخل المتصفح ولكن أصبح أيضًا يُستخدم في مجالات مثل تطوير تطبيقات الخوادم باستخدام Node.js.

تلك هي بعض الفروق الرئيسية بين لغة البرمجة Python ولغة الجافا سكريبت. يتعلم المطورون استخدام اللغات المناسبة وفقًا لاحتياجات مشروعهم وسياق العمل.

2-2-3- البايثون والجافا :

هناك العديد من الفروق بين لغة البرمجة Python ولغة Java. فيما يلي بعض الفروق الرئيسية بينهما:

1. النمط البرمجي:

Python تدعم البرمجة الشيئية (OOP) والبرمجة الإجرائية، بينما Java تركز أساسًا على البرمجة الشيئية.

2. إدارة الذاكرة:

في Python، يتمتع "Automatic Garbage Collector" جار التخزين الضمني بالقدرة على إدارة تخصيص وتحرير الذاكرة تلقائيًا، بينما في Java يتم تحكم المبرمج في تلك العمليات.

شرح متعمق:

في لغة البرمجة Python، يتم تنظيف الذاكرة تلقائيًا باستخدام آلية تسمى "جار التخزين الضمني" أو "garbage collector". الذاكرة الضمنية تتيح للبرنامج تخصيص مساحة في الذاكرة لتخزين المتغيرات والكائنات أثناء تشغيل البرنامج، وعندما لا يكون هناك حاجة لهذه المتغيرات أو الكائنات، يقوم جار التخزين الضمني بتحرير الذاكرة المخصصة لها.

الفائدة الرئيسية هنا هي أن المبرمج لا يحتاج إلى القلق بشكل مباشر حول تحرير الذاكرة بنفسه. في Python، يمكن للمبرمج كتابة الشفرة دون الحاجة إلى دعم الذاكرة بطرق تفصيلية، وذلك لأن جار التخزين الضمني يدير هذه العملية بشكل تلقائي.

بالمقابل، في لغة البرمجة Java، يُكَلِّف المبرمج بشكل أكبر بإدارة الذاكرة. يجب عليه أن يتأكد من تحرير الذاكرة التي تم تخصيصها يدويًا بمرور الوقت.

3. الأداء:

Java عادةً ما تكون أداؤها أفضل من Python، خاصة في التطبيقات الكبيرة والمعقدة. يُعتبر Java أكثر فعالية في الأداء وأداءً على المدى الطويل.

4. نظام التشغيل:

Java تعتبر من لغات البرمجة المتعددة المنصات (cross-platform)، مما يعني أن التطبيقات المكتوبة في Java يمكن تشغيلها على أنظمة تشغيل مختلفة دون الحاجة إلى إعادة كتابة الشفرة. Python أيضًا تدعم هذه الميزة بشكل جزئي، ولكن Java تفوز في هذا السياق.

5. تنفيذ الشفرة:

Python تعتبر لغة مفسرة (interpreted) حيث يتم تنفيذ الشفرة خطوة بخطوة. في المقابل، Java تعتبر لغة مترجمة (compiled) حيث يتم ترجمة الشفرة إلى لغة بينية تعتبر محمولة بين الأنظمة.

6. الكتابة والقراءة:

Python تعتبر أكثر سهولة في الكتابة والقراءة مقارنة بـ Java، حيث تقدم بنية بسيطة وقواعد أقل صرامة.

7. التعامل مع الخطأ:

Java يتطلب التعامل الصريح مع استثناءات البرمجة (exceptions)، بينما Python يتيح التعامل مع الأخطاء بطريقة أكثر تساهلاً.

9. مكتبات وإطارات العمل:

كلاهما يتمتع بمجموعة واسعة من المكتبات والإطارات الجاهزة، ولكن Python تشهد على وجود مجتمع كبير من المطورين يساهم في إنشاء وصيانة مكتبات وإطارات قوية.

تلك هي بعض الفروق الرئيسية بين لغة البرمجة Python ولغة Java. يجب على المبرمجين اختيار اللغة التي تناسب احتياجات مشروعهم والسياق الذي يعملون فيه.

2-2-4- البايثون وال- php :

هناك العديد من الفروق بين لغة البرمجة Python ولغة PHP. فيما يلي بعض الفروق الرئيسية بينهما:

1. الاستخدام الرئيسي:

Python غالباً ما يُستخدم في تطوير البرمجيات العامة، الحوسبة العلمية، وتطبيقات الويب. بينما PHP أنشئت أصلاً لتطوير تطبيقات الويب وتفاعلها مع قواعد البيانات.

2. النموذج البرمجي:

Python تدعم البرمجة الشيئية (OOP) والبرمجة الإجرائية، بينما PHP تركز أساساً على البرمجة الإجرائية، ولكن تدعم أيضاً البرمجة الشيئية.

3. الأداء:

عمومًا، PHP يتميز بأداء جيد في تطوير تطبيقات الويب والصفحات الديناميكية. Python أيضًا يُستخدم في تطوير الويب، ولكن PHP يكون غالبًا أكثر فعالية في هذا السياق.

4. تفاعل مع قواعد البيانات:

PHP تم تصميمها بشكل أساسي للتفاعل مع قواعد البيانات، وهي متكاملة بشكل أفضل مع MySQL وقواعد البيانات الأخرى. Python أيضًا يدعم تفاعل ممتاز مع قواعد البيانات، ويستخدم بشكل شائع مع SQLite و PostgreSQL.

5. الكتابة والقراءة:

Python تُعتبر أكثر سهولة في الكتابة والقراءة مقارنة بـ PHP، وتقدم بنية بسيطة وصفرية.

6. إدارة الذاكرة:

PHP تقوم بإدارة الذاكرة تلقائيًا بشكل أفضل من Python، خاصة في سياق تطوير تطبيقات الويب.

7. نظام التشغيل:

Python تعتبر من لغات البرمجة المتعددة المنصات (cross-platform)، مما يعني أن تطبيقات Python يمكن تشغيلها على أنظمة متعددة دون تعديل. PHP أيضًا تعمل على معظم خوادم الويب، ولكن يجب تكوينها بشكل مناسب لتشغيلها على نظام معين.

8. المكتبات والإطارات:

كلاهما يحظون بدعم قوي من المكتبات والإطارات الجاهزة. Python يشتهر بإطارات مثل Django و Flask، في حين أن PHP يستفيد من إطارات مثل Laravel و Symfony.

9. التوجيه الكائني:

Python تعتبر لغة مفتوحة ومرنة تدعم التوجيه الكائني بفعالية، بينما PHP تشجع على التركيز على البرمجة الإجرائية.

2-2-5- البايثون والـ C# :

هناك العديد من الفروق بين لغة البرمجة Python ولغة C#. فيما يلي بعض الفروق الرئيسية بينهما:

1. النمط البرمجي:

Python تدعم البرمجة الشيئية (OOP) والبرمجة الإجرائية، بينما C# تركز بشكل أساسي على البرمجة الشيئية.

2. النظام النموذجي:

C# تستند إلى نظام النموذج الصارم، حيث يجب تحديد نوع كل متغير بوضوح، بينما Python تستخدم نموذج النموذج الضعيف (dynamically-typed)، مما يعني أنه يمكن تعيين نوع المتغير تلقائياً دون الحاجة إلى تعيينه.

3. البيئة والمنصات:

C# تعتبر جزءاً من بيئة تطوير Microsoft وتدعم بشكل أساسي نظام التشغيل Windows. بينما Python يمكن تشغيله على أنظمة التشغيل المختلفة بما في ذلك Windows وLinux وMac OS.

4. مجتمع المطورين والمكتبات:

Python تتمتع بمجتمع كبير ونشط من المطورين، مما يعني توفر مكتبات وإطارات واسعة النطاق. C# أيضاً لديه مجتمع قوي ويستفيد من تكامله مع منصة .NET، مما يوفر مجموعة كبيرة من المكتبات والأدوات.

5. أداء التطبيقات:

عموماً، C# يُعتبر أكثر كفاءة في الأداء من Python، خاصة في تطبيقات الويب الكبيرة والمعقدة وتطبيقات سطح المكتب.

6. التطوير المتكامل:

C# يتمتع بدعم متكامل للتطوير مع Visual Studio، بينما Python يمكن تطويره باستخدام محررات النص العادية وبيئات تطوير متكاملة مثل PyCharm وVS Code.

7. الاستخدامات الرئيسية:

C# يستخدم على نطاق واسع في تطوير تطبيقات Windows، تطبيقات سطح المكتب، وتطبيقات الألعاب، بينما Python يُستخدم بشكل شائع في تطوير الويب، الحوسبة العلمية، والتطبيقات العامة.

تلك هي بعض الفروق الرئيسية بين Python وC#. يجب على المطور اختيار اللغة التي تناسب مع احتياجات مشروعه والتفضيلات الشخصية.

2-3- الأشخاص القادرين على تعلم البايثون:

لغة البرمجة Python تعتبر مناسبة لشريحة واسعة من الأشخاص، ويمكن لمجموعة متنوعة من الأفراد التعلم واستخدام Python بشكل فعال. إليك قائمة بالأشخاص الذين يمكن أن يستفيدوا من تعلم لغة البرمجة Python:

1. المبتدئين:

Python تُعتبر إحدى أفضل لغات البرمجة للمبتدئين. بفضل بنيتها البسيطة وسهولة قراءة الشفرة، يمكن للمبتدئين التعلم بسرعة وفهم مفاهيم البرمجة.

2. المطورين:

سواء كانوا مطورين ويب، مطورين حوسبة علمية، أو مطورين تطبيقات، يستخدم العديد من المحترفين Python في أعمالهم اليومية.

3. الطلاب:

Python تُستخدم على نطاق واسع في المؤسسات التعليمية والجامعات. الطلاب يمكن أن يستخدموا Python في مشاريعهم وأعمالهم الدراسية.

4. الباحثين العلميين:

يستخدم العديد من الباحثين العلميين Python في تحليل البيانات، والمحاكاة، والحوسبة العلمية بسبب العديد من المكتبات المتقدمة المتاحة.

5. المحللين البياناتيين:

Python يعتبر أداة فعالة لتحليل البيانات والإحصائيات، ويستخدمه المحللون البياناتيون لاستخراج الأنماط والتحليلات من البيانات.

6. المهندسين:

يمكن للمهندسين استخدام Python في تطوير البرمجيات والتحكم في الأجهزة، وذلك بفضل إمكانياتها في التفاعل مع الأجهزة الإلكترونية.

7. مطوري الذكاء الاصطناعي والتعلم الآلي:

Python تعتبر لغة محبوبة بين مطوري الذكاء الاصطناعي والتعلم الآلي، حيث توفر العديد من المكتبات المتخصصة مثل TensorFlow وPyTorch.

8. المطورين العاملين في مجال التطوير الويب:

Python تُستخدم بشكل شائع في تطوير الويب، سواء كان ذلك باستخدام إطارات ومكتبات مثل Flask و Django، أو في تطوير الجزء الخلفي للتطبيقات.

بشكل عام، Python تقدم بيئة تعلم مفتوحة وميسرة لمجموعة واسعة من الأفراد، من المبتدئين إلى المحترفين، ومن مختلف المجالات الفنية.

انتهى الفصل

الفصل الثالث

بيئة البايثون

3-1- بناء جملة بايثون:

تم تصميم بايثون لسهولة القراءة، ولها بعض أوجه التشابه مع اللغة الإنجليزية مع تأثير الرياضيات. تستخدم بايثون أسطرًا جديدة لإكمال الأمر، على عكس لغات البرمجة الأخرى التي غالبًا ما تستخدم الفواصل المنقوطة أو الأقواس. تعتمد بايثون على المسافة البادئة، باستخدام المسافة البيضاء، لتحديد النطاق؛ مثل نطاق الحلقات والوظائف والفئات. غالبًا ما تستخدم لغات البرمجة الأخرى الأقواس المتعرجة لهذا الغرض.

مثال:

```
print("Hello, World!")
```

يطبع العبارة Hello, World! .

3-2- تثبيت بايثون:

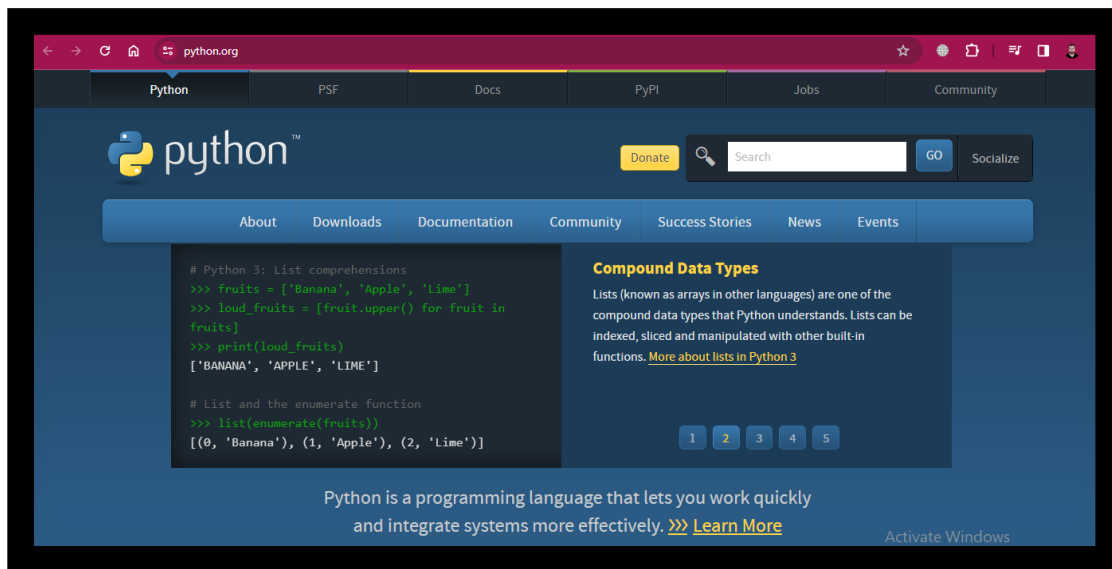
سيتم تثبيت لغة python بالفعل على العديد من أجهزة الكمبيوتر الشخصية وأجهزة Mac. للتحقق مما إذا كان لديك python مثبتًا على جهاز كمبيوتر يعمل بنظام Windows، ابحث في شريط البداية عن Python أو قم بتشغيل ما يلي في سطر الأوامر (cmd.exe):

```
C:\Users\Your Name>python -version
```

للتحقق مما إذا كان لديك python مثبتًا على نظام Linux أو Mac، ثم على Linux افتح سطر الأوامر أو على Mac افتح Terminal واكتب:

python -version

إذا وجدت أنه ليس لديك لغة Python مثبتة على جهاز الكمبيوتر الخاص بك، فيمكنك تنزيلها مجانًا من الموقع التالي: <https://www.python.org>



الشكل (1-3) موقع البايثون الأساسي

3-3- بداية سريعة :

بايثون هي لغة برمجة مفسرة، وهذا يعني أنك كمطور تكتب ملفات بايثون (.py) في محرر نصوص ثم تضع تلك الملفات في مترجم بايثون ليتم تنفيذها. طريقة تشغيل ملف بايثون هي كالتالي في سطر الأوامر:

C:\Users\Your Name>python helloworld.py

حيث "helloworld.py" هو اسم ملف python الخاص بك.

لنكتب أول ملف بايثون لدينا، يسمى helloworld.py، والذي يمكن إجراؤه باستخدام أي محرر نصوص.

helloworld.py

print("Hello, World!")

احفظ الملف الخاص بك. افتح سطر الأوامر، وانتقل إلى الدليل الذي قمت بحفظ ملفك فيه، وقم بتشغيل:

```
C:\Users\Your Name>python helloworld.py
```

يجب أن يكون الناتج كما يلي:

Hello, World!

تهانينا، لقد قمت بكتابة وتنفيذ أول برنامج بايثون لك.

3-4- سطر أوامر بايثون:

لاختبار كمية قصيرة من التعليمات البرمجية في بايثون، في بعض الأحيان يكون من الأسرع والأسهل عدم كتابة التعليمات البرمجية في ملف. أصبح هذا ممكناً لأنه يمكن تشغيل Python كسطر أوامر بحد ذاته.

اكتب ما يلي في سطر أوامر Windows أو Mac أو Linux:

```
C:\Users\Your Name>python
```

أو، إذا لم يعمل الأمر "python"، يمكنك تجربة: "py"

```
C:\Users\Your Name>py
```

من هناك يمكنك كتابة أي لغة بايثون، بما في ذلك مثالنا helloworld:

```
C:\Users\Your Name>python
```

Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license" for more information.

```
>>> print("Hello, World!")
```

Hello, World!

عندما تنتهي من سطر أوامر بايثون، يمكنك ببساطة كتابة ما يلي للخروج من واجهة سطر أوامر بايثون:

```
exit()
```

3-5- تنفيذ بناء جملة بايثون Python Syntax:

كما تعلمنا في الفقرة السابقة، يمكن تنفيذ بناء جملة بايثون عن طريق الكتابة مباشرة في سطر الأوامر:

```
>>> print("Hello, World!")
```

Hello, World!

أو عن طريق إنشاء ملف python على الخادم، باستخدام ملحق الملف py. ، وتشغيله في سطر الأوامر:

```
C:\Users\Your Name>python myfile.py
```

3-6- مسافة بادئة بايثون:

تشير المسافة البادئة إلى المسافات في بداية سطر التعليمات البرمجية.

بينما في لغات البرمجة الأخرى تكون المسافة البادئة في التعليمات البرمجية مخصصة لسهولة القراءة فقط، فإن المسافة البادئة في بايثون مهمة جدًا.

تستخدم بايثون المسافة البادئة للإشارة إلى كتلة من التعليمات البرمجية.

مثال:

```
if 5 > 2:
```

```
    print("Five is greater than two!")
```

• سوف تعطيك بايثون خطأ إذا تخطيت المسافة البادئة.

مثال:

Syntax Error:

```
if 5 > 2:
```

```
print("Five is greater than two!")
```

عدد المسافات متروك لك كمبرمج، والاستخدام الأكثر شيوعًا هو أربعة، ولكن يجب أن تكون واحدة على الأقل.

مثال:

```
if 5 > 2:  
    print("Five is greater than two!")  
if 5 > 2:  
    print("Five is greater than two!")
```

يجب عليك استخدام نفس عدد المسافات في نفس كتلة التعليمات البرمجية، وإلا فسوف تعطيك بايثون خطأ:

مثال:

```
if 5 > 2:  
    print("Five is greater than two!")  
        print("Five is greater than two!")
```

هنا سيعطي خطأ.

انتهى الفصل

الفصل الرابع

التعليقات في البايثون

4-1- مقدمة:

تتمتع Python بإمكانية التعليق لغرض التوثيق داخل التعليمات البرمجية.

تبدأ التعليقات بـ #، وستعرض بايثون بقية السطر كتعليق:

مثال:

```
#This is a comment.  
print("Hello, World!")
```

- يمكن استخدام التعليقات لشرح كود بايثون.
- يمكن استخدام التعليقات لجعل التعليمات البرمجية أكثر قابلية للقراءة.
- يمكن استخدام التعليقات لمنع التنفيذ عند اختبار التعليمات البرمجية.

4-2- إنشاء تعليق:

تبدأ التعليقات بـ #، وسوف تتجاهلها بايثون:

مثال:

```
#This is a comment  
print("Hello, World!")
```

يمكن وضع التعليقات في نهاية السطر، وستجاهل بايثون بقية السطر:

مثال:

```
print("Hello, World!") #This is a comment
```

ليس من الضروري أن يكون التعليق نصاً يشرح الكود، ويمكن استخدامه أيضاً لمنع بايثون من تنفيذ الكود:

مثال:

```
#print("Hello, World!")  
print("Cheers, Mate!")
```

3-4- تعليقات متعددة الأسطر:

لإضافة تعليق متعدد الأسطر، يمكنك إدراج # لكل سطر.

مثال:

```
#This is a comment  
#written in more than just one line  
print("Hello, World!")
```

بما أن بايثون ستجاهل القيم الحرفية للسلسلة التي لم يتم تعيينها لمتغير، فيمكنك إضافة سلسلة متعددة الأسطر (علامات الاقتباس الثلاثية) في التعليقات البرمجية الخاصة بك، ووضع تعليقك بداخلها:

مثال:

```
"""This is a comment  
written in more than just one line"""  
print("Hello, World!")
```

طالما لم يتم تعيين السلسلة لمتغير، ستقرأ بايثون الكود، لكنها تتجاهله بعد ذلك، وتكون قد قمت بعمل تعليق متعدد الأسطر.

انتهى الفصل

الفصل الخامس

المتغيرات في البايثون

5-1- مقدمة:

في بايثون، يتم إنشاء المتغيرات عندما تقوم بتعيين قيمة لها:

مثال:

```
x = 5
```

```
y = "Hello, World!"
```

ليس لدى بايثون أمر للإعلان عن متغير. والمتغيرات عبارة عن حاويات لتخزين قيم البيانات. ويتم إنشاء المتغير في اللحظة التي تقوم فيها بتعيين قيمة له لأول مرة.

مثال:

```
x = 5
```

```
y = "John"
```

```
print(x)
```

```
print(y)
```

لا يلزم التصريح عن المتغيرات بأي نوع معين، بل ويمكن تغيير نوعها بعد تعيينها.

مثال:

```
x = 4     # x is of type int
```

```
x = "Sally" # x is now of type str
```

```
print(x)
```

5-2- التحويل القسري Casting:

إذا كنت تريد تحديد نوع بيانات المتغير، فيمكن القيام بذلك عن طريق الإرسال.

مثال:

```
x = str(3) # x will be '3'
y = int(3) # y will be 3
z = float(3) # z will be 3.0
```

5-3- معرفة نوع المتغير:

يمكنك الحصول على نوع بيانات المتغير باستخدام الدالة type().

مثال:

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

سوف تتعلم المزيد عن أنواع البيانات والإرسال لاحقاً في هذا الكتاب.

5-4- اقتباسات مفردة أم مزدوجة:

يمكن الإعلان عن متغيرات السلسلة إما باستخدام علامات الاقتباس المفردة أو المزدوجة:

مثال:

```
x = "John"
# is the same as
x = 'John'
```

5-5- حساسية اللغة للمتحويلات:

أسماء المتغيرات حساسة لحالة الأحرف.

مثال:

```
a = 4
A = "Sally"
#A will not overwrite a
```

5-6- أسماء المتغيرات:

يمكن أن يكون للمتغير اسم قصير (مثل x و y) أو اسم أكثر وصفًا (العمر، اسم السيارة، الحجم الإجمالي).

5-6-1- قواعد متغيرات بايثون:

- يجب أن يبدأ اسم المتغير بحرف أو بشرطة سفلية ولا يمكن أن يبدأ اسم المتغير برقم.
- يمكن أن يحتوي اسم المتغير فقط على أحرف أبجدية رقمية وشرطات سفلية (A-Z، 0-9، و _).
- أسماء المتغيرات حساسة لحالة الأحرف (العمر والعمر والعمر هي ثلاثة متغيرات مختلفة).
- لا يمكن أن يكون اسم المتغير أيًا من كلمات Python الأساسية.

مثال: أسماء متحولات صحيحة:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

مثال: أسماء متحولات غير صحيحة:

```
2myvar = "John"
my-var = "John"
my var = "John"
```

تذكر أن أسماء المتغيرات حساسة لحالة الأحرف.

5-6-2- أسماء متغيرة متعددة الكلمات:

قد يكون من الصعب قراءة الأسماء المتغيرة التي تحتوي على أكثر من كلمة واحدة. هناك العديد من التقنيات التي يمكنك استخدامها لجعلها أكثر قابلية للقراءة:

5-6-2-1- حالة الجمل:

كل كلمة، باستثناء الأولى، تبدأ بحرف كبير:

```
myVariableName = "John"
```


5-2-2-6-2- قضية باسكال:

تبدأ كل كلمة بحرف كبير:

```
MyVariableName = "John"
```

5-2-2-6-3- حالة الثعبان:

يتم فصل كل كلمة بحرف سفلي:

```
my_variable_name = "John"
```

5-7- الإسناد لمتغيرات بايثون:**5-7-1- تعيين قيم متعددة:**

العديد من القيم لمتغيرات متعددة. تتيح لك لغة Python تعيين قيم لمتغيرات متعددة في سطر واحد.

مثال:

```
x, y, z = "Orange", "Banana", "Cherry"
```

```
print(x)
```

```
print(y)
```

```
print(z)
```

• ملاحظة: تأكد من أن عدد المتغيرات يطابق عدد القيم، وإلا فسوف تحصل على خطأ.

5-7-2- تعيين قيمة واحدة لمتغيرات متعددة:

يمكنك تعيين نفس القيمة لمتغيرات متعددة في سطر واحد:

مثال:

```
x = y = z = "Orange"
```

```
print(x)
```

```
print(y)
```

```
print(z)
```

5-7-3- فك المجموعة:

إذا كان لديك مجموعة من القيم في قائمة، في Tuple وما إلى ذلك يسمح لك Python باستخراج القيم إلى متغيرات. وهذا ما يسمى التفريغ.

مثال:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

5-8- طباعة (إخراج) المتغيرات:

تُستخدم الدالة print() غالبًا لإخراج المتغيرات.

مثال:

```
x = "Python is awesome"
print(x)
```

في الدالة print()، يمكنك إخراج متغيرات متعددة، مفصولة بفاصلة:

مثال:

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

يمكنك أيضًا استخدام عامل التشغيل + لإخراج متغيرات متعددة:

مثال:

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

لاحظ حرف المسافة بعد "Python" و "is"، فبدونهما ستكون النتيجة "Pythonisawesome".

بالنسبة للأرقام، يعمل الحرف + كعامل رياضي.

مثال:

```
x = 5
y = 10
print(x + y)
```

في الدالة print()، عندما تحاول دمج سلسلة ورقم باستخدام عامل التشغيل +، ستعطيك Python خطأ:

مثال:

```
x = 5
y = "John"
print(x + y)
```

أفضل طريقة لإخراج متغيرات متعددة في الدالة print() هي الفصل بينها بفواصل، والتي تدعم أنواعًا مختلفة من البيانات.

مثال:

```
x = 5
y = "John"
print(x, y)
```

الخرج:

5John

5-9- المتغيرات العامة:

تُعرف المتغيرات التي يتم إنشاؤها خارج الوظيفة (كما في جميع الأمثلة أعلاه) بالمتغيرات العامة. يمكن للجميع استخدام المتغيرات العامة، سواء داخل الوظائف أو خارجها.

مثال:

قم بإنشاء متغير خارج الدالة، واستخدمه داخل الدالة

```
x = "awesome"
def myfunc():
    print("Python is " + x)
myfunc()
```

الخرج:

Python is awesome

إذا قمت بإنشاء متغير بنفس الاسم داخل دالة، فسيكون هذا المتغير محليًا، ولا يمكن استخدامه إلا داخل الدالة. سيبقى المتغير الشامل الذي يحمل نفس الاسم كما كان، عامًا وبالقيمة الأصلية.

مثال:

قم بإنشاء متغير داخل دالة بنفس اسم المتغير العام

```

x = "awesome"
def myfunc():
    x = "fantastic"
    print("Python is " + x)
myfunc()
print("Python is " + x)

```

الخرج:

```

Python is fantastic
Python is awesome

```

10-5- الكلمة المحجوزة global:

عندما تقوم بإنشاء متغير داخل دالة، يكون هذا المتغير محلياً، ولا يمكن استخدامه إلا داخل تلك الدالة. لإنشاء متغير عام داخل دالة، يمكنك استخدام الكلمة الأساسية العالمية.

مثال: إذا كنت تستخدم الكلمة الأساسية العمومية، فإن المتغير ينتمي إلى النطاق العام:

```

def myfunc():
    global x
    x = "fantastic"
myfunc()
print("Python is " + x)

```

الخرج:

```

Python is fantastic

```

استخدم أيضاً الكلمة الأساسية العامة إذا كنت تريد تغيير متغير عام داخل دالة.

مثال:

لتغيير قيمة متغير عام داخل دالة، قم بالإشارة إلى المتغير باستخدام الكلمة الأساسية العامة:

```

x = "awesome"
def myfunc():
    global x
    x = "fantastic"
myfunc()
print("Python is " + x)

```

الخرج:

```

Python is fantastic

```

انتهى الفصل

الفصل السادس

أنواع البيانات في البايثون

1-6- أنواع البيانات المضمنة:

في البرمجة، نوع البيانات هو مفهوم مهم. يمكن للمتغيرات تخزين بيانات من أنواع مختلفة، ويمكن لأنواع المختلفة القيام بأشياء مختلفة.

تحتوي لغة Python على أنواع البيانات التالية المضمنة افتراضياً، في هذه الفئات:

Text Type:	str	نوع النص
Numeric Types:	int, float, complex	الأنواع الرقمية
Sequence Types:	list, tuple, range	أنواع التسلسل
Mapping Type:	dict	نوع التعيين
Set Types:	set, frozenset	أنواع المجموعة
Boolean Type:	bool	النوع المنطقي
Binary Types:	bytes, bytearray, memoryview	الأنواع الثنائية
None Type:	NoneType	لا يوجد نوع

جدول (1-6) الأنواع للبيانات في البايثون

2-6- الحصول على نوع البيانات:

يمكنك الحصول على نوع البيانات لأي كائن باستخدام الدالة `type()` :

مثال: اطبع نوع بيانات المتغير `x`:

```
x = 5
print(type(x))
```

3-6- تحديد نوع البيانات:

في بايثون، يتم تعيين نوع البيانات عندما تقوم بتعيين قيمة لمتغير.
أمثلة:

```
x = "Hello World"
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
Hello World
<class 'str'>
```

أمثلة:

```
x = 20
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
20
<class 'int'>
```

أمثلة:

```
x = 20.5
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
20.5
<class 'float'>
```

أمثلة:

```
x = 1j
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
1j
<class 'complex'>
```

أمثلة:

```
x = ["apple", "banana", "cherry"]
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
['apple', 'banana', 'cherry']
<class 'list'>
```

أمثلة:

```
x = ("apple", "banana", "cherry")
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
('apple', 'banana', 'cherry')
<class 'tuple'>
```

أمثلة:

```
x = ("apple", "banana", "cherry")
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
('apple', 'banana', 'cherry')
<class 'tuple'>
```

أمثلة:

```
x = range(6)
#display x:
print(x)
print(type(x))
```

الخرج:

```
range(0, 6)
<class 'range'>
```

أمثلة:

```
x = {"name" : "John", "age" : 36}
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
{'name': 'John', 'age': 36}
<class 'dict'>
```

أمثلة:

```
x = {"apple", "banana", "cherry"}
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
{'cherry', 'apple', 'banana'}
<class 'set'>
```

أمثلة:

```
x = frozenset({"apple", "banana", "cherry"})
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
frozenset({'cherry', 'apple', 'banana'})
<class 'frozenset'>
```

أمثلة:

```
x = True
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
True
<class 'bool'>
```


أمثلة:

```
x = b"Hello"
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
b'Hello'
<class 'bytes'>
```

أمثلة:

```
x = bytearray(5)
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
bytearray(b'\x00\x00\x00\x00\x00')
<class 'bytearray'>
```

أمثلة:

```
x = memoryview(bytes(5))
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
<memory at 0x01368FA0>
<class 'memoryview'>
```

أمثلة:

```
x = None
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
None
<class 'NoneType'>
```

4-6- ضبط نوع البيانات المحدد:

إذا كنت تريد تحديد نوع البيانات، يمكنك استخدام وظائف المنشئ التالية:

أمثلة:

```
x = str("Hello World")
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
Hello World
<class 'str'>
```

أمثلة:

```
x = int(20)
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
20
<class 'int'>
```

أمثلة:

```
x = float(20.5)
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
20.5
<class 'float'>
```

أمثلة:

```
x = complex(1j)
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
1j
<class 'complex'>
```

أمثلة:

```
x = list(("apple", "banana", "cherry"))
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
['apple', 'banana', 'cherry']
<class 'list'>
```

أمثلة:

```
x = tuple(("apple", "banana", "cherry"))
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
('apple', 'banana', 'cherry')
<class 'tuple'>
```

أمثلة:

```
x = range(6)
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
range(0, 6)
<class 'range'>
```

أمثلة:

```
x = dict(name="John", age=36)
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
{'name': 'John', 'age': 36}
<class 'dict'>
```

أمثلة:

```
x = set(("apple", "banana", "cherry"))
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
{'apple', 'cherry', 'banana'}
<class 'set'>
```

أمثلة:

```
x = frozenset(("apple", "banana", "cherry"))
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
frozenset({'apple', 'banana', 'cherry'})
<class 'frozenset'>
```

أمثلة:

```
x = bool(5)
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
True
<class 'bool'>
```

أمثلة:

```
x = bytes(5)
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
b'\x00\x00\x00\x00\x00'
<class 'bytes'>
```

أمثلة:

```
x = bytearray(5)
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
bytearray(b'\x00\x00\x00\x00\x00')
<class 'bytearray'>
```

أمثلة:

```
x = memoryview(bytes(5))
#display x:
print(x)
#display the data type of x:
print(type(x))
```

الخرج:

```
<memory at 0x0368AFA0>
<class 'memoryview'>
```

انتهى الفصل

الفصل السابع

الأعداد في البايثون

7-1- مقدمة :

هناك ثلاثة أنواع رقمية في بايثون:

- `int`
- `float`
- `complex`

يتم إنشاء متغيرات الأنواع الرقمية عندما تقوم بتعيين قيمة لها:

مثال:

```
x = 1 # int
y = 2.8 # float
z = 1j # complex
```

للتحقق من نوع أي كائن في بايثون، نستخدم الدالة `type()`:

مثال:

```
print(type(x))
print(type(y))
print(type(z))
```

7-2- الأعداد الصحيحة `int` :

`Int`، أو عدد صحيح، هو عدد صحيح، موجب أو سالب، بدون كسور عشرية، بطول غير محدود.

مثال:

```
x = 1
y = 35656222554887711
z = -3255522
```

```
print(type(x))
print(type(y))
print(type(z))
```

7-3- الأعداد العشرية float :

العدد العشري، أو "رقم النقطة العائمة" هو رقم، موجب أو سالب، يحتوي على واحد أو أكثر من الكسور العشرية.

مثال:

```
x = 1.10
y = 1.0
z = -35.59
print(type(x))
print(type(y))
print(type(z))
```

العدد العشري float : يمكن أيضًا أن يكون أرقامًا علمية بحرف "e" للإشارة إلى قوة الرقم 10.

مثال:

```
x = 35e3
y = 12E4
z = -87.7e100
print(type(x))
print(type(y))
print(type(z))
```

7-4- الأعداد العقدية أو المركبة complex :

تتم كتابة الأعداد المركبة بحرف "j" باعتباره الجزء التخيلي:

مثال:

```
x = 3+5j
y = 5j
z = -5j
print(type(x))
print(type(y))
print(type(z))
```

7-5- التحويل للأنواع casting :

يمكنك التحويل من نوع إلى آخر باستخدام الطرق int() و float() و complex():

مثال:

```
x = 1 # int
y = 2.8 # float
z = 1j # complex
a = float(x)
b = int(y)
c = complex(x)
print(a)
print(b)
print(c)
print(type(a))
print(type(b))
print(type(c))
```

• ملاحظة: لا يمكنك تحويل الأرقام المركبة إلى نوع أرقام آخر.

7-6- الرقم عشوائي :

ليس لدى بايثون دالة لإنشاء أرقام عشوائية، لكن بايثون لديها وحدة (module) تدعى random والتي يمكن استخدامها لإنشاء أرقام عشوائية:

مثال:

```
import random
print(random.randrange(1, 10))
```

7-7- تحديد نوع متغير:

قد تكون هناك أوقات تريد فيها تحديد نوع لمتغير. بايثون هي لغة موجهة للكائنات، وبالتالي فهي تستخدم الفئات لتحديد أنواع البيانات، بما في ذلك أنواعها الأساسية. لذلك يتم إجراء عملية تحديد النوع في بايثون باستخدام وظائف:

Int(): إنشاء رقم صحيح من عدد صحيح حرفي، أو عدد عشري حرفي (عن طريق إزالة جميع الكسور العشرية) ، أو سلسلة حرفية (بشرط أن تمثل السلسلة عددًا صحيحًا)

Float(): ينشئ رقمًا عائماً من عدد صحيح حرفي أو عدد عشري حرفي أو سلسلة حرفية (بشرط أن تمثل السلسلة عددًا عشريًا أو عددًا صحيحًا)

Str(): إنشاء سلسلة من مجموعة واسعة من أنواع البيانات، بما في ذلك السلاسل والأعداد الصحيحة والحرفية العائمة

مثال:

Integers:

```
x = int(1) # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

مثال:

Floats:

```
x = float(1) # x will be 1.0
y = float(2.8) # y will be 2.8
z = float("3") # z will be 3.0
w = float("4.2") # w will be 4.2
```

مثال:

Strings:

```
x = str("s1") # x will be 's1'
y = str(2) # y will be '2'
z = str(3.0) # z will be '3.0'
```

انتهى الفصل

الفصل الثامن

السلاسل في البايثون

8-1- مقدمة :

السلاسل النصية في لغة بايثون محاطة بعلامات اقتباس مفردة أو علامات اقتباس مزدوجة.

يمكنك عرض سلسلة حرفية باستخدام الدالة `print()`.

مثال:

```
print("Hello")
print('Hello')
```

8-2- إسناد سلسلة إلى متغير:

يتم تعيين سلسلة إلى متغير باستخدام اسم المتغير متبوعاً بعلامة يساوي والسلسلة:

مثال:

```
a = "Hello"
print(a)
```

8-3- السلاسل متعددة الأسطر :

يمكنك تعيين سلسلة متعددة الأسطر لمتغير باستخدام ثلاث علامات اقتباس.

مثال:

```
a = """Lorem ipsum dolor sit amet,
ut labore et dolore magna aliqua."""
print(a)
```

أو ثلاثة علامات اقتباس واحدة:

مثال:

```
a = "Lorem ipsum dolor sit amet,
ut labore et dolore magna aliqua."
print(a)
```

ملحوظة: في النتيجة، يتم إدراج فواصل الأسطر في نفس الموضع كما في الكود.

4-8- السلاسل هي المصفوفات:

مثل العديد من لغات البرمجة الشائعة الأخرى، السلاسل النصية في بايثون عبارة عن صفائف من البايتات تمثل أحرف يونيكود.

ومع ذلك، لا تحتوي لغة Python على نوع بيانات للأحرف، فالحرف الواحد هو ببساطة سلسلة بطول 1. يمكن استخدام الأقواس المربعة للوصول إلى عناصر السلسلة.

مثال: احصل على الحرف في الموضع 1 (تذكر أن الحرف الأول له الموضع 0):

```
a = "Hello, World!"
print(a[1])
```

5-8- حلقات مع السلسلة:

نظرًا لأن السلاسل عبارة عن مصفوفات، فيمكننا تكرار الأحرف الموجودة في سلسلة باستخدام حلقة for.

مثال: قم بالمرور على الحروف الموجودة في كلمة "banana":

```
for x in "banana":
    print(x)
```

6-8- طول السلسلة:

للحصول على طول السلسلة، استخدم الدالة len().

مثال: طباعة طول السلسلة a :

```
a = "Hello, World!"
print(len(a))
```

7-8- التحقق من السلسلة:

للتحقق من وجود عبارة أو حرف معين في سلسلة، يمكننا استخدام الكلمة الأساسية in.

مثال: تحقق مما إذا كانت كلمة "free" موجودة في النص التالي:

```
"The best things in life are free!"
txt = "The best things in life are free!"
print("free" in txt)
```

للتحقق من عدم وجود عبارة أو حرف معين في سلسلة ما، يمكننا استخدام الكلمة الأساسية not in.

مثال: تحقق مما إذا كانت كلمة "expensive" غير موجودة في النص التالي:

```
"The best things in life are free!"
txt = "The best things in life are free!"
print("expensive" not in txt)
```

استخدم مفهوم التحقق من السلاسل في عبارة if.

مثال: اطبع فقط في حالة عدم وجود كلمة "expensive":

```
txt = "The best things in life are free!"
if "expensive" not in txt:
    print("No, 'expensive' is NOT present.")
```

8-8- تقطيع السلاسل:

يمكنك إرجاع نطاق من الأحرف باستخدام بناء سلسلة جزئية.

حدد دليل البداية ودليل النهاية، مفصولين بنقطتين، لإرجاع جزء من السلسلة.

مثال: اطبع الأحرف من الموضع 2 إلى الموضع 5 (غير متضمن):

```
b = "Hello, World!"
print(b[2:5])
```

- ملاحظة: الحرف الأول يحتوي على فهرس 0.

8-8-1- سلسلة جزئية من البداية:

من خلال ترك دليل البداية فارغ، سيبدأ المجال عند المحرف الأول:

مثال: اطبع المحارف من البداية إلى الموضع 5 (غير متضمنة):

```
b = "Hello, World!"
print(b[:5])
```

8-8-2- سلسلة جزئية حتى النهاية:

من خلال ترك فهرس النهاية فارغ، سينتقل المجال إلى النهاية:

مثال: اطبع المحارف من الموضع 2، وحتى النهاية:

```
b = "Hello, World!"
print(b[2:])
```

8-8-3- الفهرسة السلبية:

استخدم الفهارس السالبة لبدء الشريحة من نهاية السلسلة:

مثال: اطبع المحارف:

من: "o" في "World!" (الموضع -5)

إلى: "d" في "World!" (الموضع -2):

```
b = "Hello, World!"
print(b[-5:-2])
```

8-9- التعديل على السلاسل:

لدى بايثون مجموعة من الأساليب المضمنة التي يمكنك استخدامها على السلاسل.

8-9-1- الأحرف الكبيرة:

مثال: تقوم الطريقة `Upper()` بإرجاع السلسلة المكتوبة بأحرف كبيرة:

```
a = "Hello, World!"
print(a.upper())
```

8-9-2- الأحرف الصغيرة:

مثال: تقوم الطريقة `lower()` بإرجاع السلسلة بأحرف صغيرة:

```
a = "Hello, World!"
print(a.lower())
```

8-9-3- إزالة المسافة البيضاء:

المسافة البيضاء هي المسافة قبل و/أو بعد النص الفعلي، وفي كثير من الأحيان تريد إزالة هذه المسافة.

مثال: يقوم التابع `strip()` بإزالة أي مسافة بيضاء (فارغة) من البداية أو النهاية:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

8-9-4- استبدال السلسلة:

يستبدل تابع `replace()` سلسلة بسلسلة أخرى:

مثال:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

8-9-5- تقسيم السلسلة:

تقوم طريقة `split()` بإرجاع قائمة حيث يصبح النص الموجود بين الفاصل المحدد هو عناصر القائمة.

مثال: تقوم الطريقة `split()` بتقسيم السلسلة إلى سلاسل فرعية إذا وجدت مثيلات للفاصل:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

سنتعرف على المزيد حول القوائم في فصل قوائم بايثون.

8-10- دمج السلاسل:

لربط سلسلتين أو دمجهما، يمكنك استخدام عامل `+`.

مثال: دمج المتغير `a` مع المتغير `b` في المتغير `c`:

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

ولإضافة مسافة بينهما، أضف " " :

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

8-11- تنسيق السلاسل:

كما تعلمنا في فصل متغيرات بايثون، لا يمكننا الجمع بين السلاسل والأرقام.

مثال: سيعطي خطأ

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

ولكن يمكننا الجمع بين السلاسل والأرقام باستخدام تابع format()!

يأخذ التابع format() الوسائط التي تم تمريرها، وينسقها، ويضعها في السلسلة حيث تكون العناصر النائبة {} هي:

مثال: استخدم طريقة format() لإدراج الأرقام في السلاسل:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

يأخذ التابع format() عددًا غير محدود من الوسائط، ويتم وضعه في العناصر النائبة المعنية:

مثال:

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

يمكنك استخدام أرقام الفهرس {} 0 للتأكد من وضع الوسائط في العناصر النائبة الصحيحة:

مثال:

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

8-12- إدراج أحرف غير قانونية في سلسلة:

- لإدراج أحرف غير قانونية في سلسلة، استخدم حرف هروب.
 - حرف الهروب هو شرطة مائلة عكسية \ متبوعة بالحرف الذي تريد إدراجه.
- مثال على الحرف غير القانوني هو علامة الاقتباس المزدوجة داخل سلسلة محاطة بعلامات اقتباس مزدوجة.
- مثال: سوف تحصل على خطأ إذا استخدمت علامات الاقتباس المزدوجة داخل سلسلة محاطة بعلامات اقتباس مزدوجة:

```
txt = "We are the so-called "Vikings" from the north."
```

لإصلاح هذه المشكلة، استخدم حرف الهروب \".

مثال: يسمح لك حرف الهروب باستخدام علامات الاقتباس المزدوجة عندما لا يُسمح لك بذلك عادةً:

```
txt = "We are the so-called \"Vikings\" from the north."
```

• أحرف الهروب الأخرى المستخدمة في بايثون:

- \ ' Single Quote
- \\ Backslash
- \n New Line
- \t Tab

8-13- طرق السلسلة:

لدى بايثون مجموعة من الطرق المضمنة التي يمكنك استخدامها على السلاسل.
ملاحظة: تقوم كافة طرق السلسلة بإرجاع قيم جديدة. لا يغيرون السلسلة الأصلية.

الطريقة	الوصف
capitalize()	تحويل الحرف الأول إلى أحرف كبيرة. مثال:
<pre>txt = "hello, and welcome to my world." x = txt.capitalize() print (x)</pre>	القاعدة العامة :
	مثال 2:
<pre>txt = "python is FUN!" x = txt.capitalize() print (x)</pre>	
casefold()	تحويل السلسلة إلى حالة صغيرة مثال:
<pre>txt = "Hello, And Welcome To My World!" x = txt.casefold() print(x)</pre>	التعريف والاستخدام:
	تقوم طريقة casefold() بإرجاع سلسلة حيث تكون جميع الأحرف صغيرة. تشبه هذه الطريقة الطريقة Lower()، لكن طريقة casefold() أقوى ، مما يعني أنها ستحول المزيد من الأحرف إلى أحرف صغيرة، وستجد المزيد من التطابقات عند مقارنة سلسلتين ويتم تحويل كليهما باستخدام casefold() طريقة.
	القاعدة العامة:
	<code>string.casefold()</code>

center()

إرجاع سلسلة مركزية.

مثال: اطبع كلمة "banana" بمساحة 20 حرفاً، مع وضع كلمة "banana" في المنتصف:

```
txt = "banana"
x = txt.center(20)
print(x)
```

القاعدة العامة :

```
string.center(length, character)
```

length إجباري. طول السلسلة التي تم إرجاعها*Character* اختياري. الحرف لملء المساحة المفقودة على كل جانب. الافتراضي هو " " (مسافة)

مثال : استخدام الحرف "O" كحرف الحشو:

```
txt = "banana"
x = txt.center(20, "O")
print(x)
```

count()

إرجاع عدد المرات التي تحدث فيها قيمة محددة في سلسلة.

مثال: إرجاع عدد المرات التي تظهر فيها القيمة "apple" في السلسلة:

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple")
print(x)
```

القاعدة العامة :

```
string.count(value, start, end)
```

value إجباري. سلسلة. السلسلة المراد البحث عنها*start* اختياريًا. عدد صحيح. الموقف لبدء البحث. الافتراضي هو 0*end* اختياريًا. عدد صحيح. الموقف من إنهاء البحث. الافتراضي هو نهاية السلسلة

مثال: البحث من الموضع 10 إلى 24:

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple", 10, 24)
print(x)
```

endswith()

يُرجع true إذا انتهت السلسلة بالقيمة المحددة.

مثال: سيرجع true

```
txt = "Hello, welcome to my world."
x = txt.endswith(".")
print(x)
```

القاعدة للتابع:

```
string.endswith(value, start, end)
```

value مطلوبة. القيمة المراد التحقق من انتهاء السلسلة بها*start* اختياريًا. عدد صحيح يحدد عند أي موضع لبدء البحث*end* اختياريًا. عدد صحيح يحدد الموضع الذي سيتم إنهاء البحث عنده

مثال: تحقق مما إذا كان الموضع من 5 إلى 11 ينتهي بعبارة "my world."

```
txt = "Hello, welcome to my world."
x = txt.endswith("my world.", 5, 11)
print(x)
```

expandtabs()

يضبط حجم علامة التبويب للسلسلة.

find()

يبحث في السلسلة عن قيمة محددة ويعيد الموضع الذي تم العثور عليه فيه.

مثال:

أين توجد في النص كلمة "welcome" ؟:

```
txt = "Hello, welcome to my world."
x = txt.find("welcome")
print(x)
```

التعريف والاستخدام:

تعثر طريقة find() على التواجد الأول للقيمة المحددة.

تقوم طريقة find() بإرجاع -1 إذا لم يتم العثور على القيمة.

طريقة find() هي تقريباً نفس طريقة index() والفرق الوحيد هو أن طريقة index() تثير استثناءً إذا لم يتم العثور على القيمة.

القاعدة العامة :

`string.find(value, start, end)`

value إجباري. القيمة للبحث عنها

start اختياريًا. من أين تبدأ البحث. الافتراضي هو 0

end اختياريًا. حيث لإنهاء البحث. الافتراضي هو نهاية السلسلة

مثال: أين يوجد في النص أول ظهور للحرف "e" ؟:

```
txt = "Hello, welcome to my world."
x = txt.find("e")
print(x)
```

مثال: أين يوجد في النص أول ظهور للحرف "e" عندما تبحث فقط بين الموضع 5 و 10 ؟:

```
txt = "Hello, welcome to my world."
x = txt.find("e", 5, 10)
print(x)
```

مثال:

إذا لم يتم العثور على القيمة، فإن طريقة find() ترجع -1، لكن طريقة index() ستثير استثناءً:

```
txt = "Hello, welcome to my world."
print(txt.find("q"))
print(txt.index("q"))
```

format()

تنسيق القيم المحددة في سلسلة.

format_map()

تنسيق القيم المحددة في سلسلة.

index()

يبحث في السلسلة عن قيمة محددة ويعيد الموضع الذي تم العثور عليه فيه.
مثال:

أين توجد في النص كلمة "welcome"؟:

```
txt = "Hello, welcome to my world."
x = txt.index("welcome")
print(x)
```

التعريف والاستخدام:

تعثر طريقة **index()** على التواجد الأول للقيمة المحددة.

تثير طريقة **index()** استثناءً إذا لم يتم العثور على القيمة.

طريقة **index()** هي تقريباً نفس طريقة **find()** والفرق الوحيد هو أن طريقة **find()** ترجع -1 إذا لم يتم العثور على القيمة.

القاعدة:

```
string.index(value, start, end)
```

value إجبارية. القيمة للبحث عنها

start اختياريًا. من أين تبدأ البحث. الافتراضي هو 0

end اختياريًا. حيث لإنهاء البحث. الافتراضي هو نهاية السلسلة

مثال: أين يوجد في النص أول ظهور للحرف "e"؟:

```
txt = "Hello, welcome to my world."
x = txt.index("e")
print(x)
```

مثال:

أين يوجد في النص أول ظهور للحرف "e" عندما تبحث فقط بين الموضع 5 و 10؟:

```
txt = "Hello, welcome to my world."
x = txt.index("e", 5, 10)
print(x)
```

مثال: إذا لم يتم العثور على القيمة، فإن طريقة **find()** ترجع -1، لكن طريقة **index()** ستثير استثناءً:

```
txt = "Hello, welcome to my world."
print(txt.find("q"))
print(txt.index("q"))
```

isalnum()

إرجاع True إذا كانت جميع الأحرف في السلسلة عبارة عن حروف أبجدية رقمية.

isalpha()

إرجاع True إذا كانت جميع الأحرف في السلسلة بالأحرف الأبجدية.

isascii()

إرجاع True إذا كانت جميع الأحرف في السلسلة عبارة عن أحرف **ascii**.

isdecimal()

إرجاع True إذا كانت جميع الأحرف في السلسلة عبارة عن أرقام عشرية.

isdigit()

إرجاع True إذا كانت جميع الأحرف في السلسلة عبارة عن أرقام.

isidentifier()

إرجاع True إذا كانت السلسلة عبارة عن معرف.

islower()

إرجاع True إذا كانت كافة الأحرف في السلسلة صغيرة.

مثال:

تحقق مما إذا كانت جميع الأحرف الموجودة في النص مكتوبة بأحرف صغيرة:

```
txt = "hello world!"
x = txt.islower()
print(x)
```

التعريف والاستخدام:

تقوم الدالة islower() بإرجاع True إذا كانت جميع الأحرف مكتوبة بأحرف صغيرة، وإلا فستُرجع False.

لا يتم التحقق من الأرقام والرموز والمسافات، فقط الحروف الأبجدية.
القاعدة العامة:`string.islower()`**مثال:**

تحقق مما إذا كانت جميع الأحرف الموجودة في النصوص مكتوبة بأحرف صغيرة:

```
a = "Hello world!"
b = "hello 123"
c = "mynameisPeter"
print(a.islower())
print(b.islower())
print(c.islower())
```

isnumeric()

إرجاع صحيح إذا كانت كافة الأحرف في السلسلة رقمية.

isprintable()

إرجاع True إذا كانت كافة الأحرف الموجودة في السلسلة قابلة للطباعة.

isspace()

إرجاع True إذا كانت جميع الأحرف في السلسلة عبارة عن مسافات بيضاء.

istitle()

تُرجع القيمة True إذا كانت السلسلة تتبع قواعد العنوان.

isupper()

تُرجع القيمة True إذا كانت جميع الأحرف في السلسلة مكتوبة بأحرف كبيرة.

join()

يربط عناصر التكرار حتى نهاية السلسلة.

مثال: قم بربط جميع العناصر الموجودة في tuple في سلسلة، باستخدام حرف التجزئة كفاصل:

```
myTuple = ("John", "Peter", "Vicky")
x = "#".join(myTuple)
print(x)
```

الخرج:`John#Peter#Vicky`**التعريف والاستخدام :**

تأخذ طريقة join() جميع العناصر الموجودة في كائن قابل للتكرار وتجمعها في سلسلة واحدة. يجب تحديد سلسلة كفاصل.

القاعدة العامة:`string.join(iterable)`*iterable* /جباري. أي كائن قابل للتكرار حيث تكون كافة القيم التي تم إرجاعها عبارة عن سلاسل

مثال:

قم بربط جميع العناصر الموجودة في القاموس في سلسلة، باستخدام الكلمة "TEST" كفاصل:

```
myDict = {"name": "John", "country": "Norway"}
mySeparator = "TEST"
x = mySeparator.join(myDict)
print(x)
```

الخرج:

nameTESTcountry

[ljust\(\)](#)

إرجاع نسخة مضبوطة على اليسار من السلسلة.

[lower\(\)](#)

تحويل سلسلة إلى حالة صغيرة.

[rstrip\(\)](#)

إرجاع نسخة القطع اليسرى من السلسلة.

مثال:

إزالة المسافات على يسار السلسلة:

```
txt = "  banana  "
x = txt.rstrip()
print("of all fruits", x, "is my favorite")
```

الخرج:

of all fruits banana is my favorite

التعريف والاستخدام:

تقوم الطريقة **rstrip()** بإزالة أي أحرف بادئة (المسافة هي الحرف البادئ الافتراضي المراد إزالته) القاعدة العامة:

```
string.rstrip(characters)
```

اختيارية. مجموعة من الأحرف المراد إزالتها كأحرف بادئة

مثال:

إزالة الأحرف الرائدة:

```
txt = ".,,.,,ssaaww....banana"
x = txt.rstrip(".,asw")
print(x)
```

الخرج:

banana

[maketrans\(\)](#)

إرجاع جدول الترجمة لاستخدامه في الترجمات.

[partition\(\)](#)

تقوم بإرجاع tuple حيث يتم تقسيم السلسلة إلى ثلاثة أجزاء.

مثال:

ابحث عن كلمة "bananas" وأرجع tuple يحتوي على ثلاثة عناصر:

1- كل شيء قبل "المحددة"

2- "المحددة"

3- كل شيء بعد "المحددة"

```
txt = "I could eat bananas all day"
x = txt.partition("bananas")
print(x)
```

الخرج:

('I could eat ', 'bananas', ' all day')

التعريف والاستخدام :

يبحث تابع **partition()** عن سلسلة محددة، ويقسم السلسلة إلى tuple يحتوي على ثلاثة عناصر. يحتوي العنصر الأول على الجزء الموجود قبل السلسلة المحددة. يحتوي العنصر الثاني على السلسلة المحددة. العنصر الثالث يحتوي على الجزء بعد السلسلة. ملاحظة: تبحث هذه الطريقة عن التواجد الأول للسلسلة المحددة.

القاعدة العامة:

```
string.partition(value)
```

value مطلوبة. السلسلة المطلوب البحث عنها

مثال:

إذا لم يتم العثور على القيمة المحددة، فإن تابع **partition()** يُرجع tuple يحتوي على:
1 - السلسلة بأكملها، 2 - سلسلة فارغة، 3 - سلسلة فارغة:

```
txt = "I could eat bananas all day"
x = txt.partition("apples")
print(x)
```

الخرج:

```
('I could eat bananas all day', '', '')
```

replace()

إرجاع سلسلة حيث يتم استبدال قيمة محددة بقيمة محددة.

مثال: استبدال كلمة "banana"

```
txt = "I like bananas"
x = txt.replace("bananas", "apples")
print(x)
```

التعريف والاستخدام:

تستبدل طريقة **replace()** عبارة محددة بعبارة أخرى محددة. ملاحظة: سيتم استبدال كافة تكرارات العبارة المحددة، إذا لم يتم تحديد أي شيء آخر.

القاعدة العامة:

```
string.replace(oldvalue, newvalue, count)
```

oldvalue اجباري السلسلة المطلوب البحث عنها

newvalue اجباري السلسلة المراد استبدال القيمة القديمة بها

count اختياري. رقم يحدد عدد تكرارات القيمة القديمة التي تريد استبدالها.

مثال:

استبدال كل تكرار لكلمة "one" :

```
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three")
print(x)
```

مثال:

يستعاض عن أول ظهورين لكلمة "one" :

```
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three", 2)
print(x)
```

[rfind\(\)](#)

يبحث في السلسلة عن قيمة محددة ويعيد آخر موضع تم العثور عليه فيه.

مثال: أين يوجد في النص آخر تواجد للسلسلة "casa"؟:

```
txt = "Mi casa, su casa."
x = txt.rfind("casa")
print(x)
```

التعريف والاستخدام:

- تبحث طريقة **rfind()** عن التواجد الأخير للقيمة المحددة.
- تقوم طريقة **rfind()** بإرجاع -1 إذا لم يتم العثور على القيمة.
- طريقة **rfind()** هي تقريباً نفس طريقة **rindex()**.

القاعدة العامة:

```
string.rfind(value, start, end)
```

value مطلوبة. القيمة للبحث عن

start اختياريًا. من أين تبدأ البحث. الافتراضي هو 0

end اختياريًا. حيث لإنهاء البحث. الافتراضي هو نهاية السلسلة

مثال: أين يوجد في النص آخر ظهور للحرف "e"؟:

```
txt = "Hello, welcome to my world."
x = txt.rfind("e")
print(x)
```

مثال: أين يوجد في النص آخر ظهور للحرف "e" عندما تبحث فقط بين الموضع 5 و 10؟:

```
txt = "Hello, welcome to my world."
x = txt.rfind("e", 5, 10)
print(x)
```

مثال: إذا لم يتم العثور على القيمة، فإن طريقة **rfind()** ترجع -1، لكن طريقة **rindex()** ستثير استثناءً:

```
txt = "Hello, welcome to my world."
print(txt.rfind("q"))
print(txt.rindex("q"))
```

[rindex\(\)](#)

يبحث في السلسلة عن قيمة محددة ويعيد آخر موضع تم العثور عليه فيه.

[rjust\(\)](#)

إرجاع نسخة صحيحة ومضبوطة من السلسلة.

[rpartition\(\)](#)

تقوم بإرجاع tuple حيث يتم تقسيم السلسلة إلى ثلاثة أجزاء.

مثال: ابحث عن آخر تواجد لكلمة "bananas"، وقم بإرجاع صف يحتوي على ثلاثة عناصر:

1- كل شيء قبل "المباراة"

2- "المباراة"

3- كل شيء بعد "المباراة"

```
txt = "I could eat bananas all day, bananas are my favorite fruit"
x = txt.rpartition("bananas")
print(x)
```

الخرج:

```
('I could eat bananas all day, ', 'bananas', ' are my favorite fruit')
```

التعريف والاستخدام:

- يبحث الأسلوب **rpartition()** عن آخر تواجد لسلسلة محددة، ويقسم السلسلة إلى صف يحتوي على ثلاثة عناصر.
- يحتوي العنصر الأول على الجزء الموجود قبل السلسلة المحددة.
- يحتوي العنصر الثاني على السلسلة المحددة.
- العنصر الثالث يحتوي على الجزء بعد السلسلة.

القاعدة العامة:

`string.rpartition(value)`

value مطلوبة. السلسلة المطلوب البحث عنها

مثال: إذا لم يتم العثور على القيمة المحددة، تقوم طريقة `rpartition()` بإرجاع صف يحتوي على:

1 - سلسلة فارغة، 2 - سلسلة فارغة، 3 - السلسلة بأكملها:

```
txt = "I could eat bananas all day, bananas are my favorite fruit"
x = txt.rpartition("apples")
print(x)
```

[rsplit\(\)](#)

يقسم السلسلة عند الفاصل المحدد، ويعيد قائمة `list`.

[rstrip\(\)](#)

إرجاع نسخة القطع الصحيحة من السلسلة.

مثال: قم بإزالة أي مسافات بيضاء في نهاية السلسلة:

```
txt = "   banana   "
x = txt.rstrip()
print("of all fruits", x, "is my favorite")
```

الخرج:

of all fruits banana is my favorite

القاعدة العامة:

`string.rstrip(characters)`

characters اختيارية. مجموعة من الأحرف المطلوب إزالتها كأحرف لاحقة

مثال: قم بإزالة الأحرف اللاحقة إذا كانت عبارة عن فاصلات أو نقاط أو s أو q أو w:

```
txt = "banana,,,,,ssqqqww....."
x = txt.rstrip(",.qsw")
print(x)
```

[split\(\)](#)

يقسم السلسلة عند الفاصل المحدد، ويعيد قائمة.

مثال: قم بتقسيم سلسلة إلى قائمة حيث تكون كل كلمة عبارة عن عنصر قائمة:

```
txt = "welcome to the jungle"
x = txt.split()
print(x)
```

الخرج:

['welcome', 'to', 'the', 'jungle']

التعريف والاستخدام:

تقوم طريقة `Split()` بتقسيم السلسلة إلى قائمة.

يمكنك تحديد الفاصل، الفاصل الافتراضي هو أي مسافة بيضاء.

ملاحظة: عند تحديد `maxsplit`، ستحتوي القائمة على العدد المحدد من العناصر بالإضافة إلى عنصر واحد.
القاعدة العامة:

```
string.split(separator, maxsplit)
```

`separator` اختياري. يحدد الفاصل الذي سيتم استخدامه عند تقسيم السلسلة. بشكل افتراضي، أي مسافة بيضاء هي فاصل

`maxsplit` اختياري. يحدد عدد الانقسامات التي يجب القيام بها. القيمة الافتراضية هي -1، وهي "كافة الأحداث"

مثال: قم بتقسيم السلسلة باستخدام الفاصلة، متبوعة بمسافة، كفاصل:

```
txt = "hello, my name is Peter, I am 26 years old"
x = txt.split(", ")
print(x)
```

الخرج:

```
['hello', 'my name is Peter', 'I am 26 years old']
```

مثال: استخدم حرف التجزئة كفاصل:

```
txt = "apple#banana#cherry#orange"
x = txt.split("#")
print(x)
```

الخرج:

```
['apple', 'banana', 'cherry', 'orange']
```

مثال: قم بتقسيم السلسلة إلى قائمة تحتوي على عنصرين كحد أقصى:

```
txt = "apple#banana#cherry#orange"
# setting the maxsplit parameter to 1, will return a list with 2 elements!
x = txt.split("#", 1)
print(x)
```

الخرج:

```
['apple', 'banana#cherry#orange']
```

`splitlines()`

يقسم السلسلة عند فواصل الأسطر ويعيد القائمة.

مثال: قم بتقسيم سلسلة إلى قائمة حيث يكون كل سطر عنصرًا في القائمة:

```
txt = "Thank you for the music\nWelcome to the jungle"
x = txt.splitlines()
print(x)
```

الخرج:

```
['Thank you for the music', 'Welcome to the jungle']
```

القاعدة العامة:

```
string.splitlines(keeplinebreaks)
```

`keeplinebreaks` اختياري. يحدد ما إذا كان يجب تضمين فواصل الأسطر (صحيح) أم لا (خطأ). القيمة الافتراضية خاطئة

مثال: قم بتقسيم السلسلة، ولكن احتفظ بفواصل الأسطر:

```
txt = "Thank you for the music\nWelcome to the jungle"
x = txt.splitlines(True)
print(x)
```

startswith()

يُرجع True إذا كانت السلسلة تبدأ بالقيمة المحددة.
مثال: تحقق مما إذا كانت السلسلة تبدأ بـ "Hello":

```
txt = "Hello, welcome to my world."
x = txt.startswith("Hello")
print(x)
```

التعريف والاستخدام:

تُرجع الدالة **startswith()** True إذا كانت السلسلة تبدأ بالقيمة المحددة، وإلا فستُرجع False.
القاعدة العامة:

`string.startswith(value, start, end)`

value مطلوبة. القيمة المطلوب التحقق مما إذا كانت السلسلة تبدأ بها
start اختياريًا. عدد صحيح يحدد عند أي موضع لبدء البحث
end اختياريًا. عدد صحيح يحدد الموضع الذي سيتم إنهاء البحث عنده
مثال: تحقق مما إذا كان الموضع من 7 إلى 20 يبدأ بالحرف "wel":

```
txt = "Hello, welcome to my world."
x = txt.startswith("wel", 7, 20)
print(x)
```

strip()

إرجاع نسخة مشذبة مقطعة من السلسلة.
مثال: إزالة المسافات في بداية ونهاية السلسلة:

```
txt = " banana "
```

```
x = txt.strip()
```

```
print("of all fruits", x, "is my favorite")
```

التعريف والاستخدام:

يقوم التابع **strip()** بإزالة أي مسافات بيضاء بادئة وزائدة.
البادئة تعني في بداية السلسلة، واللاحقة تعني في النهاية.
يمكنك تحديد الحرف (الحروف) المراد إزالتها، وإذا لم يكن الأمر كذلك، فستتم إزالة أي مسافات بيضاء.
القاعدة العامة :

`string.strip(characters)`

characters اختيارية. مجموعة من الأحرف المراد إزالتها كأحرف بادئة/لاحقة
مثال: إزالة الأحرف البادئة والزائدة:

```
txt = ".,rrrtgg.....banana.....rrr"
```

```
x = txt.strip(".,grt")
```

```
print(x)
```

الخرج:

banana

swapcase()

يتم تبديل الحالات، حيث تصبح الأحرف الصغيرة كبيرة والعكس صحيح.
مثال: جعل الحروف الصغيرة كبيرة والحروف الكبيرة صغيرة:

```
txt = "Hello My Name Is PETER"
```

```
x = txt.swapcase()
```

```
print(x)
```

التعريف والاستخدام:

تقوم طريقة **swapcase()** بإرجاع سلسلة حيث تكون كافة الأحرف الكبيرة صغيرة والعكس صحيح.

القاعدة العامة:

`string.swapcase()`

[title\(\)](#)

تحويل الحرف الأول من كل كلمة إلى أحرف كبيرة.
مثال: اجعل الحرف الأول في كل كلمة كبيراً:

```
txt = "Welcome to my world"
x = txt.title()
print(x)
```

التعريف والاستخدام

تقوم طريقة title() بإرجاع سلسلة حيث يكون الحرف الأول في كل كلمة بأحرف كبيرة. مثل رأس أو عنوان. إذا كانت الكلمة تحتوي على رقم أو رمز، فسيتم تحويل الحرف الأول بعد ذلك إلى أحرف كبيرة.
القاعدة العامة:

```
string.title()
```

[translate\(\)](#)

إرجاع سلسلة مترجمة.
مثال: استبدل أي أحرف "S" بحرف "P":

```
#use a dictionary with ascii codes to replace 83 (S) with 80 (P):
mydict = {83: 80}
txt = "Hello Sam!"
print(txt.translate(mydict))
```

الخرج:

```
Hello Pam!
```

التعريف والاستخدام:

- تقوم طريقة Translator () بإرجاع سلسلة حيث يتم استبدال بعض الأحرف المحددة بالحرف الموصوف في القاموس، أو في جدول التعيين.
- استخدم طريقة maketrans () لإنشاء جدول التعيين.
- إذا لم يتم تحديد حرف في القاموس/الجدول، فلن يتم استبدال الحرف.
- إذا كنت تستخدم قاموساً، فيجب عليك استخدام رموز ascii بدلاً من الأحرف.

```
string.translate(table)
```

table مطلوب. إما قاموس، أو جدول تعيين يصف كيفية إجراء الاستبدال
مثال: استخدم جدول التعيين لاستبدال "S" بـ "P":

```
txt = "Hello Sam!"
mytable = str.maketrans("S", "P")
print(txt.translate(mytable))
```

مثال: استخدم جدول التعيين لاستبدال العديد من الأحرف:

```
txt = "Hi Sam!"
x = "mSa"
y = "eJo"
mytable = str.maketrans(x, y)
print(txt.translate(mytable))
```

[upper\(\)](#)

تحويل سلسلة إلى حالة كبيرة.

[zfill\(\)](#)

يملاً السلسلة بعدد محدد من القيم 0 في البداية.

انتهى الفصل

الفصل التاسع

القيم البوليانية (المنطقية) في البايثون

9-1- مقدمة:

تمثل القيم المنطقية إحدى القيمتين: True أو False. يمكنك تقييم أي تعبير في بايثون، والحصول على إحدى إجابتين، True أو False.

عند مقارنة قيمتين، يتم تقييم التعبير وترجع بايثون الإجابة المنطقية.

مثال:

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

عند استخدام شرط في عبارة if، تُرجع بايثون True أو False.

مثال: اطبع رسالة بناءً على ما إذا كان الشرط True أم False :

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

9-2- تقييم القيم والمتغيرات:

تتيح لك الدالة bool() تقييم أي قيمة وترجع True أو False .

مثال: تقييم سلسلة ورقم:

```
print(bool("Hello"))
print(bool(15))
```

مثال: تقييم متغيرين:

```
x = "Hello"
y = 15
print(bool(x))
print(bool(y))
```

معظم القيم صحيحة. يتم تقييم أي قيمة تقريباً إلى True إذا كانت تحتوي على نوع ما من المحتوى.

- أي سلسلة صحيحة، باستثناء السلاسل الفارغة.
- أي رقم صحيح، باستثناء 0.
- أي قائمة، و tuple، و set، وقاموس صحيحة، باستثناء تلك الفارغة.

مثال: الخرج سيكون True :

```
bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
```

بعض القيم False

في الواقع، لا توجد العديد من القيم التي يتم تقييمها على أنها False ، باستثناء القيم الفارغة، مثل () ، [] ، {} ،
"" ، والرقم 0 ، والقيمة None. وبالطبع يتم تقييم القيمة False إلى False.

مثال:

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

9-3- إرجاع قيمة منطقية:

يمكنك إنشاء وظائف تُرجع قيمة منطقية.

مثال: اطبع القيمة المنطقية عن طريق التابع :

```
def myFunction() :
    return True
print(myFunction())
```

ملحوظة: يمكنك تنفيذ التعليمات البرمجية بناءً على الإجابة المنطقية للدالة.

مثال: اطبع YES اذا التابع ارجع True وإلا اطبع NO

```
def myFunction() :  
    return True  
if myFunction():  
    print("YES!")  
else:  
    print("NO!")
```

تحتوي بايثون أيضاً على العديد من الوظائف المضمنة التي تُرجع قيمة منطقية، مثل دالة `isinstance()` والتي يمكن استخدامها لتحديد ما إذا كان الكائن من نوع بيانات معين:

مثال: تأكد فيما إذا الكائن نوعه `int` أو لا:

```
x = 200  
print(isinstance(x, int))
```

انتهى الفصل

الفصل العاشر

المعاملات في البايثون

10-1- مقدمة:

يتم استخدام العوامل لإجراء العمليات على المتغيرات والقيم. وفي المثال أدناه، نستخدم المعامل + لجمع قيمتين معًا:

مثال:

```
print(10 + 5)
```

تُقسّم البايثون العوامل إلى المجموعات التالية:

- معاملات العمليات الحسابية
- معاملات التعيين
- معاملات المقارنة
- معاملات المنطقية
- معاملات الهوية
- معاملات العضوية
- معاملات Bitwise
- معاملات بايثون الحسابية

10-2- المعاملات الحسابية:

تُستخدم المعاملات الحسابية مع القيم الرقمية لإجراء العمليات الحسابية الشائعة:

المعامل	الاسم	مثال
+	الجمع	$x + y$
-	الطرح	$x - y$
*	الضرب	$x * y$
/	القسمة	x / y
%	باقي القسمة	$x \% y$
**	أس العدد	$x ** y$
//	قسمة مع تدوير	$x // y$

10-3- المعاملات الإسنادية:

يتم استخدام عوامل التعيين لتعيين قيم للمتغيرات:

المعامل	التعبير	تعبير مشابه
=	$x = 5$	$x = 5$
+=	$x += 3$	$x = x + 3$
-=	$x -= 3$	$x = x - 3$
*=	$x *= 3$	$x = x * 3$
/=	$x /= 3$	$x = x / 3$
%=	$x \% = 3$	$x = x \% 3$
//=	$x //= 3$	$x = x // 3$
**=	$x ** = 3$	$x = x ** 3$
&=	$x \& = 3$	$x = x \& 3$
=	$x = 3$	$x = x 3$
^=	$x ^ = 3$	$x = x ^ 3$

10-4- معاملات المقارنة:

يتم استخدام عوامل المقارنة لمقارنة قيمتين:

المعامل	الاسم	مثال
==	المساواة	$x == y$
!=	عدم المساواة	$x != y$
>	أكبر من	$x > y$
<	أقل من	$x < y$
>=	أكبر أو يساوي	$x >= y$
<=	أصغر أو يساوي	$x <= y$

10-5- المعاملات المنطقية:

يتم استخدام العوامل المنطقية لدمج العبارات الشرطية:

المعامل	الوصف	مثال
and	يرجع True اذا كلا التعبيرين صحيحين	$x < 5 \text{ and } x < 10$
or	يرجع True اذا كان إحدى التعبيرين صحيحين	$x < 5 \text{ or } x < 4$
not	يعكس النتيجة اذا كانت True تصبح False	$\text{not}(x < 5 \text{ and } x < 10)$

10-6- معاملات الهوية:

يتم استخدام معاملات الهوية لمقارنة الكائنات، ليس إذا كانت متساوية، ولكن إذا كانت في الواقع نفس الكائن، مع نفس موقع الذاكرة:

المعامل	الوصف	مثال
is	يرجع True إذا كان كلا المتغيرين هما نفس الكائن	x is y
is not	يرجع True إذا كان كلا المتغيرين ليسا نفس الكائن	x is not y

10-7- معاملات العضوية:

تُستخدم معاملات العضوية لاختبار ما إذا كان هناك تسلسل موجود في كائن:

المعامل	الوصف	مثال
in	تُرجع True في حالة وجود تسلسل بالقيمة المحددة في الكائن	x in y
not in	تُرجع True في حالة عدم وجود تسلسل بالقيمة المحددة في الكائن	x not in y

8-10- معاملات Bitwise:

يتم استخدام عوامل Bitwise لمقارنة الأرقام (الثنائية):

المعامل	الاسم	الوصف	مثال
&	AND	يسند الى كل بت القيمة 1 اذا كانت البتات كلها 1 في $x \& y$	$x \& y$
	OR	يسند الى كل بت القيمة 1 اذا كان إحدى البتات 1 في $y x$	$x y$
^	XOR	يضببط كل بت على 1 إذا كان واحد فقط من البتتين هو 1	$x \wedge y$
~	NOT	يعكس كل البتات	$\sim x$
<<	Zero fill left shift	انتقل إلى اليسار عن طريق دفع الأصفار من اليمين واترك البتات الموجودة في أقصى اليسار تسقط	$x << 2$
>>	Signed right shift	قم بالتحريك لليمين عن طريق دفع نسخ من أقصى اليسار للداخل من اليسار، واترك البتات الموجودة في أقصى اليمين تسقط	$x >> 2$

10-9- أسبقية المشغل:

- تصف أسبقية عامل التشغيل الترتيب الذي يتم به تنفيذ العمليات.
- الأقواس لها الأسبقية القصوى، مما يعني أنه يجب تقييم التعبيرات الموجودة داخل الأقواس أولاً.
- يتم توضيح ترتيب الأسبقية في الجدول أدناه، بدءاً من الأسبقية الأعلى ثم الأدنى:

المعامل	الوصف
()	بين قوسين
**	الأس
+x -x ~x	أحادي زائد، أحادي ناقص، و bitwise NOT
* / // %	الضرب والقسمة والتقسيم الأرضي والمعامل
+ -	جمع وطرح
<< >>	التحويلات باتجاه اليسار واليمين
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
== != > >= < <= is in , not in, is not	المقارنات والهوية ومشغلي العضوية
not	منطقية NOT
and	AND
or	OR

إذا كان هناك عاملين لهما نفس الأسبقية، فسيتم تقييم التعبير من اليسار إلى اليمين.

انتهى الفصل

الفصل الحادي عشر

القوائم في البايثون

11-1- مجموعات بايثون (المصفوفات) :

هناك أربعة أنواع من بيانات المجموعة في لغة برمجة بايثون:

القائمة عبارة عن مجموعة مرتبة وقابلة للتغيير. يسمح بأعضاء مكررة.

Tuple عبارة عن مجموعة مرتبة وغير قابلة للتغيير. يسمح بأعضاء مكررة.

المجموعة عبارة عن مجموعة غير مرتبة وغير قابلة للتغيير * وغير مفهرسة. لا يوجد أعضاء مكررة.

القاموس عبارة عن مجموعة مرتبة ** وقابلة للتغيير. لا يوجد أعضاء مكررة.

* عناصر المجموعة غير قابلة للتغيير، ولكن يمكنك إزالة و/أو إضافة عناصر وقتما تشاء.

** اعتبارًا من إصدار Python 3.7، يتم ترتيب القواميس. في Python 3.6 والإصدارات الأقدم، القواميس غير مرتبة.

عند اختيار نوع المجموعة، من المفيد فهم خصائص هذا النوع. إن اختيار النوع المناسب لمجموعة

بيانات معينة قد يعني الاحتفاظ بالمعنى، وقد يعني زيادة في الكفاءة أو الأمان.

11-2- مقدمة عن القوائم:

تُستخدم القوائم لتخزين عناصر متعددة في متغير واحد.

مثال:

```
mylist = ["apple", "banana", "cherry"]
```

القوائم هي واحدة من أنواع البيانات الأربعة المضمنة في لغة Python المستخدمة لتخزين مجموعات من البيانات، أما الأنواع الثلاثة الأخرى فهي Tuple و Set و Dictionary، وكلها ذات خصائص واستخدامات مختلفة.

11-3- إنشاء القوائم:

- يتم إنشاء القوائم باستخدام الأقواس المربعة.

مثال:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

- من الممكن أيضًا استخدام تابع list () عند إنشاء قائمة جديدة.

مثال: استخدم باني القائمة list() لإنشاء قائمة:

```
thislist = list(("apple", "banana", "cherry"))
# note the double round-brackets
print(thislist)
```

11-4- خصائص القائمة:

- عناصر القائمة مرتبة وقابلة للتغيير وتسمح بقيم مكررة.

ملحوظة: تتم فهرسة عناصر القائمة، العنصر الأول يحتوي على فهرس [0]، والعنصر الثاني يحتوي على فهرس [1] وما إلى ذلك.

Ordered

- عندما نقول أن القوائم مرتبة، فهذا يعني أن العناصر لها ترتيب محدد، ولن يتغير هذا الترتيب.
 - إذا قمت بإضافة عناصر جديدة إلى القائمة، فسيتم وضع العناصر الجديدة في نهاية القائمة.
- ملاحظة: هناك بعض طرق القائمة التي من شأنها تغيير الترتيب، ولكن بشكل عام: ترتيب العناصر لن يتغير.

Changeable قابل للتغيير

- القائمة قابلة للتغيير، مما يعني أنه يمكننا تغيير العناصر وإضافتها وإزالتها في القائمة بعد إنشائها.

Allow Duplicates السماح بالتكرارات

- بما أن القوائم مفهرسة، يمكن أن تحتوي القوائم على عناصر بنفس القيمة.

مثال: تسمح القوائم بقيم مكررة:

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

11-5- طول القائمة:

لتحديد عدد العناصر الموجودة في القائمة، استخدم الدالة len():

مثال: طباعة عدد العناصر في القائمة:

```
thislist = ["apple", "banana", "cherry"]  
print(len(thislist))
```

11-6- أنواع البيانات لعناصر القائمة:

يمكن أن تكون عناصر القائمة من أي نوع بيانات:

أمثلة: أنواع البيانات String و int و Boolean داخل القائمة:

```
list1 = ["apple", "banana", "cherry"]  
list2 = [1, 5, 7, 9, 3]  
list3 = [True, False, False]
```

يمكن أن تحتوي القائمة على أنواع بيانات مختلفة.

مثال: قائمة تحتوي على سلاسل وأعداد صحيحة وقيم منطقية:

```
list1 = ["abc", 34, True, 40, "male"]
```

11-7- تحديد النوع للقوائم type():

من وجهة نظر بايثون، يتم تعريف القوائم على أنها كائنات بنوع البيانات "قائمة":

```
<class 'list'>
```

مثال: ما هو نوع بيانات القائمة؟

```
mylist = ["apple", "banana", "cherry"]  
print(type(mylist))
```

11-8- الوصول إلى العناصر:

يتم فهرسة عناصر القائمة ويمكنك الوصول إليها بالرجوع إلى رقم الفهرس.

مثال: طباعة العنصر الثاني من القائمة:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist[1])
```

ملاحظة: يحتوي العنصر الأول على فهرس 0.

11-8-1- الفهرسة السلبية:

الفهرسة السلبية تعني البدء من النهاية (- 1) يشير إلى العنصر الأخير، (- 2) يشير إلى العنصر

الأخير الثاني وما إلى ذلك.

مثال: طباعة العنصر الأخير في القائمة:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

11-8-2- نطاق الفهارس:

يمكنك تحديد نطاق من الفهارس عن طريق تحديد مكان البدء ومكان نهاية النطاق. عند تحديد نطاق، ستكون القيمة المرجعة عبارة عن قائمة جديدة تحتوي على العناصر المحددة. مثال: إرجاع العنصر الثالث والرابع والخامس:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

ملاحظة: سيبدأ البحث عند الفهرس 2 (متضمن) وينتهي عند الفهرس 5 (غير مضمن). تذكر أن العنصر الأول يحتوي على فهرس 0. من خلال ترك قيمة البداية، سيبدأ النطاق عند العنصر الأول. مثال: قم بإرجاع العناصر من البداية إلى "kiwi" ولكن لا تتضمنها:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

من خلال ترك القيمة النهائية، سيستمر النطاق حتى نهاية القائمة. مثال: قم بإرجاع العناصر من "cherry" إلى النهاية:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

11-8-3- مجموعة من المؤشرات السلبية:

حدد الفهارس السالبة إذا كنت تريد بدء البحث من نهاية القائمة.

مثال: يقوم هذا المثال بإرجاع العناصر من "orange" (4-) إلى "mango" (1-) ولكن لا يشملها:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

11-8-4- تحقق من وجود العنصر:

لتحديد ما إذا كان هناك عنصر محدد موجود في القائمة، استخدم الكلمة الأساسية **in**.

مثال: تحقق مما إذا كانت كلمة "apple" موجودة في القائمة:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")
```

11-9- تغيير قيمة عنصر:

لتغيير قيمة عنصر معين، راجع رقم الفهرس.

مثال: قم بتغيير العنصر الثاني:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

11-10- تغيير نطاق من قيم عنصر:

لتغيير قيمة العناصر ضمن نطاق معين، حدد قائمة بالقيم الجديدة، وقم بالإشارة إلى نطاق أرقام الفهرس الذي تريد إدراج القيم الجديدة فيه.

مثال: قم بتغيير قيمتي "banana" و "cherry" بقيمتي "blackcurrant" و "watermelon":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

الخرج:

```
['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

إذا قمت بإدراج عناصر أكثر مما قمت باستبداله، فسيتم إدراج العناصر الجديدة في المكان الذي حددته، وسيتم نقل العناصر المتبقية وفقاً لذلك.

مثال: قم بتغيير القيمة الثانية عن طريق استبدالها بقيمتين جديدتين:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

الخرج:

```
['apple', 'blackcurrant', 'watermelon', 'cherry']
```

ملاحظة: سيتغير طول القائمة عندما لا يتطابق عدد العناصر المدرجة مع عدد العناصر المستبدلة.

إذا قمت بإدراج عناصر أقل مما قمت باستبداله، فسيتم إدراج العناصر الجديدة في المكان الذي حددته، وسيتم نقل العناصر المتبقية وفقاً لذلك:

مثال: قم بتغيير القيمة الثانية والثالثة باستبدالها بقيمة واحدة:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

الخرج:

```
['apple', 'watermelon']
```

11-11- إدراج عناصر:

لإدراج عنصر قائمة جديد، دون استبدال أي من القيم الموجودة، يمكننا استخدام طريقة `insert()`.

تقوم طريقة `insert()` بإدراج عنصر في الفهرس المحدد:

مثال: أدخل "watermelon" كعنصر ثالث:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

الخرج:

```
['apple', 'banana', 'watermelon', 'cherry']
```

ملاحظة: نتيجة للمثال أعلاه، ستحتوي القائمة الآن على 4 عناصر.

11-12- إضافة عناصر:

لإضافة عنصر إلى نهاية القائمة، استخدم طريقة `append()`:

مثال: استخدام طريقة `append()` لإلحاق عنصر:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

11-13- توسيع القائمة:

لإلحاق عناصر من قائمة أخرى بالقائمة الحالية، استخدم طريقة `extend()`.

مثال: أضف العناصر `tropical` إلى `thislist`:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

الخرج:

```
['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']
```

سيتم إضافة العناصر إلى نهاية القائمة.

11-14- إضافة أي Iterable:

ليس من الضروري أن تقوم طريقة `extend()` بإلحاق القوائم، حيث يمكنك إضافة أي كائن قابل للتكرار (الصفوف والمجموعات والقواميس وما إلى ذلك).

مثال: إضافة عناصر Tuple إلى القائمة:

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)
```

11-15- إزالة العنصر المحدد:

تقوم طريقة الإزالة `remove()` العنصر المحدد.

مثال: إزالة "banana":

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

إذا كان هناك أكثر من عنصر واحد بالقيمة المحددة، فسيقوم التابع `remove()` بإزالة التكرار الأول.

مثال: إزالة أول ظهور لكلمة "banana":

```
thislist = ["apple", "banana", "cherry", "banana", "kiwi"]
thislist.remove("banana")
print(thislist)
```

11-16- إزالة العنصر المحدد:

• تقوم طريقة `pop()` بإزالة الفهرس المحدد.

مثال: إزالة العنصر الثاني:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

• إذا لم تقم بتحديد الفهرس، فإن طريقة `pop()` تزيل العنصر الأخير.

مثال: إزالة العنصر الأخير:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

• تقوم الكلمة الأساسية `del` أيضًا بإزالة الفهرس المحدد.

مثال: إزالة العنصر الأول:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

- يمكن للكلمة الأساسية del أيضًا حذف القائمة بالكامل.

مثال: إزالة القائمة كاملة:

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

11-17- مسح القائمة:

- تقوم الطريقة clear() بإفراغ القائمة.
- القائمة لا تزال قائمة، ولكن ليس لديها محتوى.

مثال: مسح محتوى القائمة:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

11-18- حلقة على القائمة:

يمكنك المرور على عناصر القائمة باستخدام حلقة for:

مثال: طباعة جميع العناصر الموجودة في القائمة، واحدًا تلو الآخر:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
    print(x)
```

تعرف على المزيد حول حلقات for في فصل Python For Loops.

11-19- حلقة من خلال أرقام الفهرس:

- يمكنك أيضًا المرور على عناصر القائمة بالإشارة إلى رقم الفهرس الخاص بها.
- استخدم الدالتين range() و len() لإنشاء كائن قابل للتكرار مناسب.

مثال: طباعة جميع العناصر من خلال الإشارة إلى رقمها الفهرس:

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
    print(thislist[i])
```

التكرار الذي تم إنشاؤه في المثال أعلاه هو [0، 1، 2].

11-20- استخدام حلقة while:

- يمكنك تكرار عناصر القائمة باستخدام حلقة while.
- استخدم الدالة len() لتحديد طول القائمة، ثم ابدأ من 0 وانتقل عبر عناصر القائمة بالرجوع إلى فهرسها.
- تذكر زيادة الفهرس بمقدار 1 بعد كل تكرار.

مثال: اطبع جميع العناصر باستخدام حلقة while لاستعراض جميع أرقام الفهرس

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
    print(thislist[i])
    i = i + 1
```

تعرف على المزيد حول حلقات while في فصل Python while Loops.

11-21- التكرار باستخدام فهم القائمة:

يقدم فهم القائمة أقصر بناء جملة للتكرار عبر القوائم.

مثال: يد قصيرة (short hand) للحلقة for التي ستطبع جميع العناصر في القائمة:

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

11-22- فهم قائمة List Comprehension:

يوفر فهم القائمة بناء جملة أقصر عندما تريد إنشاء قائمة جديدة بناءً على قيم القائمة الموجودة.

مثال: بناءً على قائمة الفواكه، تريد قائمة جديدة تحتوي فقط على الفواكه التي تحتوي على الحرف "a" في الاسم.

بدون فهم القائمة، سيتعين عليك كتابة عبارة for مع اختبار شرطي بداخلها.

مثال:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
    if "a" in x:
        newlist.append(x)
print(newlist)
```

باستخدام فهم القائمة، يمكنك القيام بكل ذلك باستخدام سطر واحد فقط من التعليمات البرمجية.

مثال:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

القاعدة العامة:

```
newlist = [expression for item in iterable if condition == True]
```

القيمة المرجعة هي قائمة جديدة، مع ترك القائمة القديمة دون تغيير. يشبه الشرط عامل التصفية الذي يقبل فقط العناصر التي يتم تقييمها على أنها True.

مثال: قبول فقط العناصر التي ليست "apple":

```
newlist = [x for x in fruits if x != "apple"]
```

الشرط إذا كان `x != "apple"` سيرجع True لجميع العناصر بخلاف "apple"، مما يجعل القائمة الجديدة تحتوي على جميع الفواكه باستثناء "apple".

• **الشرط اختياري ويمكن حذفه:**

مثال: مع عدم وجود عبارة if:

```
newlist = [x for x in fruits]
```

• **متوقعة:**

يمكن أن يكون الكائن القابل للتكرار أي كائن قابل للتكرار، مثل قائمة أو صف أو مجموعة وما إلى ذلك. مثال: يمكنك استخدام الدالة `range()` لإنشاء كائن قابل للتكرار:

```
newlist = [x for x in range(10)]
```

• **نفس المثال لكن بشرط:**

مثال: قبول الأرقام الأقل من 5 فقط:

```
newlist = [x for x in range(10) if x < 5]
```

• **تعبير:**

التعبير هو العنصر الحالي في التكرار، ولكنه أيضاً النتيجة، والتي يمكنك معالجتها قبل أن ينتهي بها الأمر كعنصر قائمة في القائمة الجديدة:

مثال: قم بتعيين القيم في القائمة الجديدة إلى أحرف كبيرة:

```
newlist = [x.upper() for x in fruits]
```

يمكنك ضبط النتيجة على ما تريد.

مثال: اضبط جميع القيم في القائمة الجديدة على "hello":

```
newlist = ['hello' for x in fruits]
```

يمكن أن يحتوي التعبير أيضاً على شروط، ليس مثل المرشح، ولكن كوسيلة لمعالجة النتيجة.

مثال: إرجاع "orange" بدلاً من "banana":

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

يقول **التعبير** في المثال أعلاه:

"أرجع السلعة إذا لم تكن موزة، وإذا كانت موزة قم بإرجاعها برتقالة".

11-23- فرز القائمة:

تحتوي كائنات القائمة على طريقة sort() والتي ستقوم بفرز القائمة أبجدياً رقمياً، بشكل تصاعدي افتراضياً.

مثال: ترتيب القائمة أبجدياً:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

مثال: فرز القائمة عددياً:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

11-23-1- ترتيب تنازلي:

للفرز تنازلياً، استخدم الكلمة الأساسية reverse = True.

مثال: ترتيب القائمة تنازلياً:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

مثال 2: ترتيب القائمة تنازلياً:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```

11-23-2- تخصيص وظيفة الفرز:

يمكنك أيضاً تخصيص وظيفتك الخاصة باستخدام الكلمة الأساسية الوسيطة key = function.

ستعيد الدالة رقماً سيتم استخدامه لفرز القائمة (الرقم الأقل أولاً).

مثال: قم بفرز القائمة بناءً على مدى قرب الرقم من 50:

```
def myfunc(n):
    return abs(n - 50)
thislist = [100, 50, 65, 82, 23, 1]
thislist.sort(key = myfunc)
print(thislist)
```

الخرج:

[50, 65, 23, 82, 1, 100]

11-23-3- فرز غير حساس لحالة الأحرف:

افتراضياً، تكون الدالة `sort()` حساسة لحالة الأحرف، مما يؤدي إلى فرز جميع الأحرف الكبيرة قبل الأحرف الصغيرة.

مثال: يمكن أن يؤدي الفرز الحساس لحالة الأحرف إلى نتيجة غير متوقعة:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```

الخرج:

['Kiwi', 'Orange', 'banana', 'cherry']

لحسن الحظ، يمكننا استخدام الوظائف المضمنة كوظائف رئيسية عند فرز القائمة.

لذا، إذا كنت تريد وظيفة فرز غير حساسة لحالة الأحرف، فاستخدم `str.lower` كوظيفة رئيسية.

مثال: قم بإجراء نوع غير حساس لحالة الأحرف من القائمة:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

الخرج:

['banana', 'cherry', 'Kiwi', 'Orange']

11-23-4- ترتيب عكسي:

ماذا لو كنت تريد عكس ترتيب القائمة، بغض النظر عن الأبجدية؟ تعمل الطريقة `reverse()` على عكس ترتيب الفرز الحالي للعناصر.

مثال: عكس ترتيب عناصر القائمة:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

11-24- نسخ القائمة:

لا يمكنك نسخ قائمة ببساطة عن طريق كتابة `list2 = list1`، لأن: `list2` ستكون مرجعًا فقط إلى `list1`، وسيتم إجراء التغييرات التي تم إجراؤها في `list1` تلقائيًا في `list2` أيضًا. هناك طرق لإنشاء نسخة، إحدى الطرق هي استخدام نسخة طريقة القائمة المضمنة. مثال: قم بعمل نسخة من القائمة باستخدام طريقة `Copy()`:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

هناك طريقة أخرى لعمل نسخة وهي استخدام الطريقة المضمنة `list()`.

مثال: قم بعمل نسخة من القائمة باستخدام طريقة `list()`:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

11-25- ربط قائمتين:

هناك عدة طرق لربط أو ربط قائمتين أو أكثر في بايثون. إحدى أسهل الطرق هي استخدام عامل التشغيل `+`.

مثال: اربط قائمتين:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
```

هناك طريقة أخرى لضم قائمتين وهي إلحاق كافة العناصر من القائمة 2 إلى القائمة 1، واحدًا تلو الآخر.

مثال: إلحاق القائمة 2 بالقائمة 1:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
for x in list2:
    list1.append(x)
print(list1)
```

أو يمكنك استخدام التابع `extend()`، حيث يكون الغرض هو إضافة عناصر من قائمة إلى قائمة أخرى.

مثال: استخدم طريقة `extend()` لإضافة القائمة 2 في نهاية القائمة 1:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)
```

11-26- طرق القائمة:

لدى Python مجموعة من الأساليب المضمنة التي يمكنك استخدامها في القوائم.

الطريقة	الوصف
<u>append()</u>	يضيف عنصرا في نهاية القائمة
<u>clear()</u>	يزيل كافة العناصر من القائمة
<u>copy()</u>	إرجاع نسخة من القائمة
<u>count()</u>	إرجاع عدد العناصر بالقيمة المحددة
<u>extend()</u>	أضف عناصر القائمة (أو أي عنصر قابل للتكرار) إلى نهاية القائمة الحالية
<u>index()</u>	إرجاع فهرس العنصر الأول بالقيمة المحددة
<u>insert()</u>	يضيف عنصرا في الموضع المحدد
<u>pop()</u>	يزيل العنصر في الموضع المحدد
<u>remove()</u>	إزالة العنصر بالقيمة المحددة
<u>reverse()</u>	يعكس ترتيب القائمة
<u>sort()</u>	فرز القائمة

انتهى الفصل

الفصل الثاني عشر

ال Tuples في البايثون

1-12- مقدمة:

هناك أربعة أنواع من بيانات المجموعة في لغة برمجة بايثون:

1. القائمة عبارة عن مجموعة مرتبة وقابلة للتغيير. يسمح بأعضاء مكررة.
 2. Tuple عبارة عن مجموعة مرتبة وغير قابلة للتغيير. يسمح بأعضاء مكررة.
 3. المجموعة عبارة عن مجموعة غير مرتبة وغير قابلة للتغيير* وغير مفهرسة. لا يوجد أعضاء مكررة.
 4. القاموس عبارة عن مجموعة مرتبة وقابلة للتغيير. لا يوجد أعضاء مكررة.
- *عناصر المجموعة غير قابلة للتغيير، ولكن يمكنك إزالة و/أو إضافة عناصر وقتما تشاء. يتم استخدام الصفوف لتخزين عناصر متعددة في متغير واحد.

```
mytuple = ("apple", "banana", "cherry")
```

Tuple: هو أحد أنواع البيانات الأربعة المضمنة في لغة Python المستخدمة لتخزين مجموعات من البيانات، أما الأنواع الثلاثة الأخرى فهي List و Set و Dictionary، وكلها ذات خصائص واستخدامات مختلفة.

Tuple عبارة عن مجموعة مرتبة وغير قابلة للتغيير.

تتم كتابة الصفوف بأقواس مستديرة.

مثال: إنشاء Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

2-12- عناصر الصفوف:

عناصر Tuple مرتبة وغير قابلة للتغيير وتسمح بقيم مكررة.

تتم فهرسة عناصر Tuple، العنصر الأول يحتوي على فهرس [0]، والعنصر الثاني يحتوي على فهرس [1] وما إلى ذلك.

• Ordered

عندما نقول أن الصفوف مرتبة، فهذا يعني أن العناصر لها ترتيب محدد، ولن يتغير هذا الترتيب.

• غير قابل للتغيير Unchangeable

الصف غير قابل للتغيير، مما يعني أنه لا يمكننا تغيير العناصر أو إضافتها أو إزالتها بعد إنشاء الصف.

• السماح بالتكرارات Allow Duplicates

نظرًا لأن الصفوف مفهرسة، فمن الممكن أن تحتوي على عناصر لها نفس القيمة.

مثال: تسمح Tuples بقيم مكررة:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

12-3- طول الصفوف:

لتحديد عدد العناصر الموجودة في الصف، استخدم الدالة `len()`.

مثال: اطبع عدد العناصر في الصف:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

12-4- إنشاء Tuple بعنصر واحد:

لإنشاء صف يحتوي على عنصر واحد فقط، عليك إضافة فاصلة بعد العنصر، وإلا فلن تتعرف عليه Python على أنه صف.

مثال: صف عنصر واحد، تذكر الفاصلة:

```
thistuple = ("apple",)
print(type(thistuple))
#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

12-5- عناصر Tuple - أنواع البيانات:

يمكن أن تكون عناصر Tuple من أي نوع بيانات.

مثال: أنواع البيانات `String` و `int` و `boolean`:

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

يمكن أن يحتوي الصف على أنواع بيانات مختلفة.

مثال: صف يحتوي على سلاسل وأعداد صحيحة وقيم منطقية:

```
tuple1 = ("abc", 34, True, 40, "male")
```

• في حال استخدام type() :

من وجهة نظر بايثون، يتم تعريف الصفوف على أنها كائنات من نوع البيانات "tuple":

```
<class 'tuple'>
```

مثال: ما هو نوع البيانات من Tuple؟

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

12-6- باني ال Tuple:

من الممكن أيضًا استخدام باني tuple() لإنشاء صف.

مثال: باستخدام طريقة tuple() لإنشاء صف:

```
thistuple = tuple(("apple", "banana", "cherry"))
# note the double round-brackets
print(thistuple)
```

12-7- الوصول إلى عناصر Tuple:

يمكنك الوصول إلى عناصر الصف من خلال الرجوع إلى رقم الفهرس الموجود داخل الأقواس المربعة.

مثال: اطبع العنصر الثاني في الصف:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

• ملاحظة: يحتوي العنصر الأول على فهرس 0.

12-8- الفهرسة السلبية في Tuple:

الفهرسة السلبية تعني البدء من النهاية.

(- 1) يشير إلى العنصر الأخير، (- 2) يشير إلى العنصر الأخير الثاني وما إلى ذلك.

مثال: طباعة العنصر الأخير من tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

12-9- نطاق الفهارس في Tuple:

يمكنك تحديد نطاق من الفهارس عن طريق تحديد مكان البدء ومكان نهاية النطاق.

عند تحديد نطاق، ستكون القيمة المرجعة عبارة عن **صف جديد** يحتوي على العناصر المحددة.

مثال: إرجاع العنصر الثالث والرابع والخامس:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi")
print(thistuple[2:5])
```

- ملاحظة: سيبدأ البحث عند الفهرس 2 (متضمن) وينتهي عند الفهرس 5 (غير مضمن).
- تذكر أن العنصر الأول يحتوي على فهرس 0.
- من خلال ترك قيمة البداية، سيبدأ النطاق عند العنصر الأول.

مثال: يقوم هذا المثال بإرجاع العناصر من البداية إلى "kiwi"، بحيث لا يتم تضمينها:

```
thistuple = ("apple", "orange", "kiwi", "melon")
print(thistuple[:4])
```

من خلال ترك القيمة النهائية، سيستمر النطاق حتى نهاية الصف.

مثال: يقوم هذا المثال بإرجاع العناصر من "cherry" إلى النهاية:

```
thistuple = ("apple", "banana", "cherry", "kiwi")
print(thistuple[2:])
```

10-12- مجموعة من المؤشرات السلبية في Tuple:

حدد الفهارس السالبة إذا كنت تريد بدء البحث من نهاية المجموعة.

مثال: يقوم هذا المثال بإرجاع العناصر من الفهرس -4 (مضمن) إلى الفهرس -1 (مستبعد)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi")
print(thistuple[-4:-1])
```

11-12- تحقق من وجود العنصر في Tuple:

لتحديد ما إذا كان هناك عنصر محدد موجود في الصف، استخدم الكلمة الأساسية **in**.

مثال: تحقق مما إذا كانت كلمة "apple" موجودة في الصف:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
    print("Yes, 'apple' is in the fruits tuple")
```

12-12- تحديث الـ Tuple:

الصف غير قابل للتغيير، مما يعني أنه لا يمكنك تغيير العناصر أو إضافتها أو إزالتها بمجرد إنشاء الصف.

ولكن هناك بعض الحلول:

يمكنك تحويل الصف إلى قائمة وتغيير القائمة وتحويل القائمة مرة أخرى إلى صف.

مثال: قم بتحويل المجموعة إلى قائمة لتتمكن من تغييرها:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

12-13- إضافة عناصر إلى Tuple:

نظرًا لأن الصفوف غير قابلة للتغيير، فهي لا تحتوي على طريقة `append()` مدمجة، ولكن هناك طرق أخرى لإضافة عناصر إلى الصف.

1. **التحويل إلى قائمة:** تمامًا مثل الحل البديل لتغيير الصف، يمكنك تحويله إلى قائمة وإضافة العنصر (العناصر) وتحويله مرة أخرى إلى صف.

مثال: تحويل الصف إلى قائمة، وإضافة "orange"، وتحويله مرة أخرى إلى صف:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

2. **إضافة صف إلى صف:** يُسمح لك بإضافة صفوف إلى صفوف، لذا إذا كنت تريد إضافة عنصر واحد (أو عدة عناصر)، فقم بإنشاء صف جديد مع العنصر (العناصر)، وأضفه إلى الصف الموجود. مثال: قم بإنشاء صف جديد بالقيمة "orange"، وأضف هذا الصف:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y
print(thistuple)
```

ملاحظة: عند إنشاء صف يحتوي على عنصر واحد فقط، تذكر تضمين فاصلة بعد العنصر، وإلا فلن يتم تعريفه على أنه صف.

12-14- إزالة العناصر من Tuple:

🔧 ملاحظة: لا يمكنك إزالة العناصر الموجودة في صف.

الصف غير قابل للتغيير، لذلك لا يمكنك إزالة العناصر منه، ولكن يمكنك استخدام نفس الحل البديل الذي استخدمناه لتغيير عناصر الصف وإضافتها.

مثال: تحويل الصف إلى قائمة، وإزالة "apple"، وتحويله مرة أخرى إلى صف:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

أو يمكنك حذف المجموعة بالكامل.

مثال: يمكن للكلمة الأساسية **del** حذف الصف بالكامل:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

12-15- Tuple: تفريغ

عندما نقوم بإنشاء صف، فإننا عادةً ما نقوم بتعيين قيم له. وهذا ما يسمى "التعبئة" للصف.

مثال: التعبئة لـ Tuple:

```
fruits = ("apple", "banana", "cherry")
```

لكن في بايثون، يُسمح لنا أيضًا باستخراج القيم مرة أخرى إلى متغيرات. وهذا ما يسمى "التفريغ".

مثال: تفريغ الصفوف:

```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)
```

ملاحظة: يجب أن يتطابق عدد المتغيرات مع عدد القيم في الصف، وإذا لم يكن الأمر كذلك، فيجب عليك استخدام علامة النجمة لجمع القيم المتبقية كقائمة.

استخدام العلامة النجمية*: إذا كان عدد المتغيرات أقل من عدد القيم، يمكنك إضافة * إلى اسم المتغير وسيتم تخصيص القيم للمتغير كقائمة.

مثال: قم بتعيين بقية القيم كقائمة تسمى "red":

```
fruits = ("apple", "banana", "cherry", "strawberry")
(green, yellow, *red) = fruits
print(green)
print(yellow)
print(red)
```

إذا تمت إضافة العلامة النجمية إلى اسم متغير آخر غير الاسم الأخير، فسوف تقوم بايثون بتعيين قيم للمتغير حتى يتطابق عدد القيم المتبقية مع عدد المتغيرات المتبقية.

مثال: أضف قائمة قيم المتغير "tropic":

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
(green, *tropic, red) = fruits
print(green, tropic, red)
```

16-12- حلقة من خلال Tuple:

يمكنك إجراء حلقات عبر عناصر المجموعة باستخدام حلقة **for**.

مثال: قم بالتكرار عبر العناصر وطباعة القيم:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
    print(x)
```


17-12- حلقة من خلال أرقام الفهرس في Tuple:

يمكنك أيضاً تكرار عناصر المجموعة من خلال الإشارة إلى رقم الفهرس الخاص بها. استخدم الدالتين `len()` و `range()` لإنشاء كائن قابل للتكرار مناسب.

مثال: طباعة جميع العناصر من خلال الإشارة إلى رقمها الفهرس:

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
    print(thistuple[i])
```

18-12- استخدام حلقة while في Tuple:

يمكنك التكرار خلال عناصر المجموعة باستخدام حلقة `while`. استخدم الدالة `len()` لتحديد طول الصف، ثم ابدأ من 0 ثم كرر طريقك عبر عناصر الصف من خلال الرجوع إلى فهرسها.  تذكر زيادة الفهرس بمقدار 1 بعد كل تكرار.

مثال: اطبع جميع العناصر باستخدام حلقة `while` لاستعراض جميع أرقام الفهرس:

```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
    print(thistuple[i])
    i = i + 1
```

19-12- ربط الصفوف Tuple:

لضم صفين أو أكثر، يمكنك استخدام عامل التشغيل `+`.
مثال: ربط صفين:

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

20-12- مضاعفة الصفوف Tuple:

إذا كنت تريد مضاعفة محتوى الصف لعدد معين من المرات، فيمكنك استخدام عامل التشغيل `*`.

مثال: اضرب fruits في 2:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
```

21-12- طرق الصفوف:

لدى بايثون طريقتان مدمجتان يمكنك استخدامهما في الصفوف.

الوصف	الطريقة
<p>إرجاع عدد المرات التي تحدث فيها قيمة محددة في صف</p> <p>مثال: قم بإرجاع عدد المرات التي تظهر فيها القيمة 5 في الصف:</p> <pre>thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5) x = thistuple.count(5) print(x)</pre> <p>الخرج: 2</p> <p>التعريف والاستخدام:</p> <p>تُرجع طريقة count() عدد المرات التي تظهر فيها قيمة محددة في الصف.</p> <p>القاعدة العامة:</p> <p><code>tuple.count(value)</code></p> <p><i>value</i> مطلوبة. العنصر المراد البحث عنه</p>	<p>count()</p>
<p>يبحث في الصف عن قيمة محددة ويعيد الموضع الذي تم العثور عليه فيه</p> <p>مثال: ابحث عن التواجد الأول للقيمة 8، وقم بإرجاع موضعه:</p> <pre>thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5) x = thistuple.index(8) print(x)</pre> <p>الخرج: 3</p> <p>التعريف والاستخدام</p> <p>تعثر طريقة <code>index()</code> على التواجد الأول للقيمة المحددة.</p> <p>تثير طريقة <code>index()</code> استثناءً إذا لم يتم العثور على القيمة.</p> <p>القاعدة العامة:</p> <p><code>tuple.index(value)</code></p> <p><i>value</i> مطلوبة. العنصر المراد البحث عنه</p>	<p>index()</p>

انتهى الفصل

الفصل الثالث عشر

المجموعات في البايثون

13-1- مقدمة:

المجموعة:

تُستخدم المجموعات لتخزين عناصر متعددة في متغير واحد.

مثال: الشكل العام:

```
myset = {"apple", "banana", "cherry"}
```

Set: هو أحد أنواع البيانات الأربعة المضمنة في لغة Python المستخدمة لتخزين مجموعات من البيانات، والأنواع الثلاثة الأخرى هي List و Tuple و Dictionary، وكلها ذات خصائص واستخدامات مختلفة.

المجموعة عبارة عن مجموعة غير مرتبة وغير قابلة للتغيير * وغير مفهومة.

* ملاحظة: عناصر المجموعة غير قابلة للتغيير، ولكن يمكنك إزالة العناصر وإضافة عناصر جديدة.

13-2- إنشاء المجموعة:

تتم كتابة المجموعات بأقواس مجعدة { }.

مثال: أنشئ مجموعة:

```
thisset = {"apple", "banana", "cherry"}
```

```
print(thisset)
```

من الممكن أيضًا استخدام مُنشئ set() لإنشاء مجموعة.

مثال: استخدم الباني set() لإنشاء مجموعة :

```
thisset = set(("apple", "banana", "cherry"))
```

```
# note the double round-brackets
```

```
print(thisset)
```


13-3- خصائص المجموعة:

ملاحظة: المجموعات غير مرتبة، لذا لا يمكنك التأكد من الترتيب الذي ستظهر به العناصر.

تعيين العناصر Set Items

عناصر المجموعة غير مرتبة وغير قابلة للتغيير ولا تسمح بقيم مكررة.

غير مرتبة Unordered

غير مرتبة يعني أن العناصر الموجودة في المجموعة ليس لها ترتيب محدد.

يمكن أن تظهر عناصر المجموعة بترتيب مختلف في كل مرة تستخدمها، ولا يمكن الإشارة إليها بواسطة الفهرس أو المفتاح.

غير قابل للتغيير Unchangeable

عناصر المجموعة غير قابلة للتغيير، يعني أنه لا يمكننا تغيير العناصر بعد إنشاء المجموعة. بمجرد إنشاء المجموعة، لا يمكنك تغيير عناصرها، ولكن يمكنك إزالة وإضافة عناصر.

التكرارات غير مسموح بها Duplicates Not Allowed

لا يمكن أن تحتوي المجموعات على عنصرين بنفس القيمة.

مثال: مضاعفة القيم يتم تجاهله:

```
thisset = {"apple", "banana", "cherry", "apple"}  
print(thisset)
```

ملاحظة: تعتبر القيمتان True و 1 نفس القيمة في المجموعات، ويتم التعامل معها على أنها مكررة.

مثال:

```
thisset = {"apple", "banana", "cherry", True, 1, 2}  
print(thisset)
```

ملاحظة: تعتبر القيمتان False و 0 نفس القيمة في المجموعات، ويتم التعامل معها على أنها مكررة.

مثال:

```
thisset = {"apple", "banana", "cherry", False, True, 0}  
print(thisset)
```

13-4- طول المجموعة:

لتحديد عدد العناصر الموجودة في المجموعة، استخدم الدالة len().

مثال: أوجد عدد عناصر المجموعة:

```
thisset = {"apple", "banana", "cherry"}  
print(len(thisset))
```

13-5- أنواع البيانات في المجموعة:

يمكن أن تكون عناصر المجموعة من أي نوع بيانات.

مثال:

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

يمكن أن تحتوي المجموعة على أنواع بيانات مختلفة.

مثال:

```
set1 = {"abc", 34, True, 40, "male"}
```

13-6- نوع المجموعة:

من وجهة نظر بايثون، يتم تعريف المجموعات ككائنات بنوع البيانات "set":

```
<class 'set'>
```

مثال:

```
myset = {"apple", "banana", "cherry"}
print(type(myset))
```

13-7- مجموعات بايثون (المصفوفات) (تذكيرة):

هناك أربعة أنواع من بيانات المجموعة في لغة برمجة بايثون:

- القائمة عبارة عن مجموعة مرتبة وقابلة للتغيير. يسمح بأعضاء مكررة.
 - Tuple عبارة عن مجموعة مرتبة وغير قابلة للتغيير. يسمح بأعضاء مكررة.
 - المجموعة عبارة عن مجموعة غير مرتبة وغير قابلة للتغيير* وغير مفهرسة. لا يوجد أعضاء مكررة.
 - القاموس عبارة عن مجموعة مرتبة وقابلة للتغيير. لا يوجد أعضاء مكررة.
- *عناصر المجموعة غير قابلة للتغيير، ولكن يمكنك إزالة العناصر وإضافة عناصر جديدة.

13-8- الوصول إلى عناصر المجموعة:

لا يمكنك الوصول إلى العناصر الموجودة في مجموعة بالإشارة إلى فهرس أو مفتاح. ولكن يمكنك تكرار عناصر المجموعة باستخدام حلقة for، أو السؤال عما إذا كانت هناك قيمة محددة موجودة في مجموعة، باستخدام الكلمة الأساسية in.

مثال: استخدم حلقة لطباعة عناصر المجموعة :

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
    print(x)
```

مثال: افحص فيما إذا كلمة الـ "banana" موجودة بالمجموعة:

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

9-13- تغيير العناصر المجموعة:

بمجرد إنشاء المجموعة، لا يمكنك تغيير عناصرها، ولكن يمكنك إضافة عناصر جديدة.

9-13-1- إضافة عنصر الى المجموعة:

بمجرد إنشاء المجموعة، لا يمكنك تغيير عناصرها، ولكن يمكنك إضافة عناصر جديدة.

لإضافة عنصر واحد إلى مجموعة استخدم طريقة `add()`.

مثال:

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

9-13-2- إضافة مجموعات:

لإضافة عناصر من مجموعة أخرى إلى المجموعة الحالية، استخدم طريقة `update()`.

مثال: أضف عناصر من مجموعة الى أخرى:

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)
```

9-13-3- إضافة أي Iterable:

ليس من الضروري أن يكون الكائن الموجود في طريقة التحديث `update()` ، بل يمكن أن يكون أي كائن قابل للتكرار (صفوف، قوائم، قواميس وما إلى ذلك).

مثال: أضف عناصر في قائمة إلى مجموعة:

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)
```

10-13- حذف العناصر في المجموعة:

لإزالة عنصر في مجموعة، استخدم طريقة `remove()` أو طريقة `discard()`.

مثال: احذف كلمة "banana" باستخدام تابع `remove()`:

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

ملاحظة: إذا كان العنصر المطلوب إزالته غير موجود، فستؤدي عملية `remove()` إلى ظهور خطأ.

مثال: احذف كلمة "banana" باستخدام تابع `discard()` :

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

ملاحظة: إذا كان العنصر المطلوب إزالته غير موجود، فإن يؤدي التابع `discard()` إلى ظهور خطأ.

يمكنك أيضاً استخدام طريقة `pop()` لإزالة عنصر ما، ولكن هذه الطريقة ستزيل عنصراً عشوائياً، لذلك لا يمكنك التأكد من العنصر الذي سيتم إزالته.

القيمة المرجعة لطريقة `pop()` هي العنصر الذي تمت إزالته.

مثال: احذف عنصر من المجموعة بشكل عشوائي باستخدام طريقة الـ `pop()` :

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

ملاحظة: المجموعات غير مرتبة، لذلك عند استخدام طريقة `pop()`، فإنك لا تعرف العنصر الذي سيتم إزالته.

مثال: قم بإفراغ جميع العناصر في المجموعة باستخدام `clear()` :

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

مثال: قم بحذف المجموعة بشكل كامل باستخدام الكلمة المحجوزة `del` :

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

11-13- عناصر المجموعة باستخدام الحلقة:

يمكنك تكرار العناصر المحددة باستخدام حلقة `for`.

مثال: قم بالمرور عبر المجموعة وطباعة القيم:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
    print(x)
```

12-13- ربط المجموعات:

- هناك عدة طرق لضم مجموعتين أو أكثر في بايثون.
- يمكنك استخدام طريقة `union()` التي تُرجع مجموعة جديدة تحتوي على جميع العناصر من كلتا المجموعتين، أو طريقة `update()` التي تُدرج جميع العناصر من مجموعة إلى أخرى.

مثال: تُرجع الطريقة `union()` مجموعة جديدة تحتوي على جميع العناصر من كلتا المجموعتين:

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

الخرج:

```
{'b', 2, 3, 1, 'c', 'a'}
```

- يمكنك استخدام | عامل التشغيل بدلاً من طريقة `union()` ، وستحصل على نفس النتيجة.

مثال:

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1 | set2
print(set3)
```

الخرج:

```
{'b', 2, 'a', 'c', 3, 1}
```

يمكن استخدام جميع طرق الانضمام وعوامل التشغيل للانضمام إلى مجموعات متعددة. عند استخدام إحدى الطرق، ما عليك سوى إضافة المزيد من المجموعات بين قوسين، مفصولة بفواصل.

مثال:

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = {"John", "Elena"}
myset = set1.union(set2, set3)
print(myset)
```

الخرج:

```
{b, Elena, John, 'c', 'a', 2, 3, 1}
```

- عند استخدام | المشغل، قم بفصل المجموعات بالمزيد | العاملين.

مثال:

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = {"John", "Elena"}
set4 = {"apple", "bananas", "cherry"}
myset = set1 | set2 | set3 | set4
print(myset)
```

الخرج:

```
{Elena, 'a', John, apple, 'b', cherry, 3, banana, 'c', 2, 1}
```

مثال: يقوم تابع `update()` بإدراج العناصر الموجودة في `set2` في `set1`:

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set1.update(set2)
print(set1)    ##{'b', 'a', 1, 3, 'c', 2}
```

ملحوظة: كل من union() و update() سوف يستبعدان أي عناصر مكررة.

• الاحتفاظ فقط بالنسخ المكررة.

مثال: اربط set1 و set2، لكن احتفظ بالنسخ المكررة فقط:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.intersection(y)
print(z)
```

الخرج:

```
{'apple'}
```

ستحتفظ طريقة intersection_update() أيضاً بالنسخ المكررة فقط، ولكنها ستغير المجموعة الأصلية بدلاً من إرجاع مجموعة جديدة.

مثال: احتفظ بالعناصر الموجودة في كل من set1 و set2:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set1.intersection_update(set2)
print(set1)
```

الخرج:

```
{'apple'}
```

تعتبر القيمتين True و 1 نفس القيمة. وينطبق الشيء نفسه على خطأ و 0.

مثال: اربط المجموعات التي تحتوي على القيم True و False و 1 و 0، وشاهد ما يعتبر تكراراً:

```
set1 = {"apple", 1, "banana", 0, "cherry"}
set2 = {"False", "google", 1, "apple", 2, True}
set3 = set1.intersection(set2)
print(set3)
```

الخرج:

```
{'False', True, 'apple'}
```

يمكنك استخدام عامل التشغيل & بدلاً من طريقة intersection() وستحصل على نفس النتيجة.

مثال: استخدم & للانضمام إلى مجموعتين:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1 & set2
print(set3)
```

ملاحظة: يسمح لك عامل التشغيل & فقط بربط المجموعات مع المجموعات، وليس مع أنواع البيانات الأخرى مثلما يمكنك باستخدام طريقة intersection().

سأشرح طريقة difference() مجموعة جديدة تحتوي فقط على العناصر من المجموعة الأولى غير الموجودة في المجموعة الأخرى.

مثال: احتفظ بجميع العناصر من المجموعة 1 غير الموجودة في المجموعة 2:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1.difference(set2)
print(set3)
```

الخرج:

```
{'banana', 'cherry'}
```

يمكنك استخدام عامل التشغيل - بدلاً من طريقة difference() وستحصل على نفس النتيجة.

مثال: استخدم - لإيجاد الفرق:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1 - set2
print(set3)
```

الخرج:

```
{'banana', 'cherry'}
```

ملاحظة: يسمح لك عامل التشغيل - فقط بربط المجموعات مع المجموعات، وليس مع أنواع البيانات الأخرى

كما يمكنك باستخدام طريقة الفرق.

ستحتفظ طريقة difference_update() أيضاً بالعناصر من المجموعة الأولى غير الموجودة في المجموعة الأخرى، ولكنها ستغير المجموعة الأصلية بدلاً من إرجاع مجموعة جديدة.

مثال: استخدم طريقة difference_update() للاحتفاظ بالعناصر غير الموجودة في كلتا المجموعتين:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set1.difference_update(set2)
print(set1)
```

● احتفظ بكل شيء، ولكن ليس التكرارات.

ستحتفظ طريقة sysmatic_difference_update() فقط بالعناصر غير الموجودة في كلتا المجموعتين.

مثال: احتفظ بالعناصر غير الموجودة في المجموعتين:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.symmetric_difference_update(y)
print(x)
```

الخرج:

```
{'google', 'banana', 'microsoft', 'cherry'}
```

يمكنك استخدام عامل التشغيل \wedge بدلاً من طريقة `symmetric_difference()`، وستحصل على نفس النتيجة.

مثال: استخدم \wedge لربط مجموعتين:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1 ^ set2
print(set3)
```

الخرج:

```
{'google', 'banana', 'microsoft', 'cherry'}
```

ملاحظة: يسمح لك عامل التشغيل \wedge فقط بربط المجموعات مع المجموعات، وليس مع أنواع البيانات الأخرى كما يمكنك باستخدام طريقة `symmetric_difference()`.

ستحتفظ الطريقة `symmetric_difference_update()` أيضاً بكل شيء ما عدا التكرارات، ولكنها ستغير المجموعة الأصلية بدلاً من إرجاع مجموعة جديدة.

مثال: استخدم طريقة `symmetric_difference_update()` للاحتفاظ بالعناصر غير الموجودة في المجموعتين:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set1.symmetric_difference_update(set2)
print(set1)
```

سيُرجع التابع `symmetric_difference()` مجموعة جديدة تحتوي فقط على العناصر غير الموجودة في كلتا المجموعتين. ملاحظة: القيمتان `True` و `1` تعتبران نفس القيمة في المجموعات، ويتم معاملتهما كقيم مكررة.

مثال:

```
x = {"apple", "banana", "cherry", True}
y = {"google", 1, "apple", 2}
z = x.symmetric_difference(y)
print(z)
```

الخرج:

```
{'google', 2, 'banana', 'cherry'}
```


13-13- طرق المجموعة:

لدى بايثون مجموعة من الطرق المضمنة التي يمكنك استخدامها في المجموعات.

الطرق	الوصف
add()	إضافة عنصر الى المجموعة
clear()	حذف العناصر جميعها من المجموعة
copy()	إعادة نسخة من المجموعة
difference()	إرجاع مجموعة تحتوي على الفرق بين مجموعتين أو أكثر تحتوي المجموعة التي تم إرجاعها على عناصر موجودة فقط في المجموعة الأولى، وليس في كلتا المجموعتين. كاختصار، يمكنك استخدام عامل التشغيل - بدلاً من ذلك.
difference_update()	إزالة العناصر الموجودة في هذه المجموعة والمضمنة أيضاً في مجموعة أخرى محددة
discard()	احذف قيمة معينة تقوم طريقة التجاهل بإزالة العنصر المحدد من المجموعة. تختلف هذه الطريقة عن طريقة الإزالة remove() لأن طريقة الإزالة discard() ستثير خطأ في حالة عدم وجود العنصر المحدد، ولن يحدث ذلك في طريقة التجاهل discard().
intersection()	تُرجع مجموعة، وهي تقاطع مجموعتين أخريين تقوم طريقة التقاطع بإرجاع مجموعة تحتوي على التشابه بين مجموعتين أو أكثر. المجموعة التي تم إرجاعها تحتوي فقط على العناصر الموجودة في كلتا المجموعتين، أو في جميع المجموعات إذا تم إجراء المقارنة مع أكثر من مجموعتين. كاختصار، يمكنك استخدام عامل التشغيل & بدلاً من ذلك.

[intersection_update\(\)](#) إزالة العناصر الموجودة في هذه المجموعة والتي لا توجد في مجموعة (مجموعات) أخرى محددة
تقوم طريقة `intersection_update()` بإزالة العناصر غير الموجودة في كلتا المجموعتين (أو في جميع المجموعات إذا تم إجراء المقارنة بين أكثر من مجموعتين).
تختلف طريقة `intersection_update()` عن طريقة `intersection()`، لأن طريقة `intersection()` تُرجع مجموعة جديدة، دون العناصر غير المرغوب فيها، بينما تقوم طريقة `intersection_update()` بإزالة العناصر غير المرغوب فيها من المجموعة الأصلية.
كاختصار، يمكنك استخدام عامل التشغيل `&=` بدلاً من ذلك.

[isdisjoint\(\)](#) إرجاع ما إذا كانت مجموعتان لهما تقاطع أم لا
يُرجع الأسلوب `isdisjoint()` [True] في حالة عدم وجود أي من العناصر في كلتا المجموعتين، وإلا فإنه يُرجع [False].
مثال: ماذا لو لم تكن هناك عناصر في كلا المجموعتين؟
إرجاع False في حالة وجود عنصر واحد أو أكثر في كلتا المجموعتين:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.isdisjoint(y)
print(z)
```

الخرج:

Flase

[issubset\(\)](#) إرجاع ما إذا كانت مجموعة أخرى تحتوي على هذه المجموعة أم لا
تقوم الطريقة `issubset()` بإرجاع True إذا كانت جميع العناصر الموجودة في المجموعة الأصلية موجودة في المجموعة المحددة، وإلا فإنها ترجع False.
كاختصار، يمكنك استخدام عامل التشغيل `=>` بدلاً من ذلك، انظر المثال أدناه.

مثال:

استخدم `=>` كاختصار بدلاً من `issubset()`:

```
x = {"a", "b", "c"}
y = {"f", "e", "d", "c", "b", "a"}
z = x <= y #z = x.issubset(y)
print(z)
#True
```

issuperset()

إرجاع ما إذا كانت هذه المجموعة تحتوي على مجموعة أخرى أم لا. تقوم الطريقة `issuperset()` بإرجاع `True` إذا كانت جميع العناصر الموجودة في المجموعة المحددة موجودة في المجموعة الأصلية، وإلا فإنها ترجع `False`. باختصار، يمكنك استخدام عامل التشغيل `<=` بدلاً من ذلك، انظر المثال أدناه. مثال: يُرجع `True` إذا كانت جميع عناصر المجموعة `y` موجودة في المجموعة `x`:

```
x = {"f", "e", "d", "c", "b", "a"}
y = {"a", "b", "c"}
z = x.issuperset(y)
print(z)
#True
```

pop()

احذف عنصر من المجموعة. تقوم طريقة `pop()` بإزالة عنصر عشوائي من المجموعة. تقوم هذه الطريقة بإرجاع العنصر الذي تمت إزالته. مثال: إرجاع العنصر المحذوف:

```
fruits = {"apple", "banana", "cherry"}
fruits.pop()
print(fruits) # {'banana', 'cherry'}
```

remove()

احذف عنصر محدد. تقوم طريقة `remove()` بإزالة العنصر المحدد من المجموعة. تختلف هذه الطريقة عن طريقة `discard()` لأن طريقة `remove()` ستثير خطأ في حالة عدم وجود العنصر المحدد، ولن تكون طريقة `discard()` كذلك. مثال: إزالة `"banana"` من المجموعة:

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")
print(fruits) # {'cherry', 'apple'}
```

symmetric_difference()

إرجاع مجموعة ذات فروق متماثلة بين مجموعتين. يُرجع الأسلوب `symmetric_difference()` مجموعة تحتوي على كافة العناصر من كلتا المجموعتين، ولكن ليس العناصر الموجودة في كلتا المجموعتين. تحتوي المجموعة المرتجعة على مزيج من العناصر غير الموجودة في كلتا المجموعتين. باختصار، يمكنك استخدام عامل التشغيل `^` بدلاً من ذلك، انظر المثال أدناه. مثال:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x ^ y # z = x.symmetric_difference(y)
print(z)
```

[symmetric_difference_update\(\)](#)

يُدرج الاختلافات المتماثلة من هذه المجموعة وأخرى.

يقوم الأسلوب `symmetric_difference_update()` بتحديث المجموعة الأصلية عن طريق إزالة العناصر الموجودة في كلتا المجموعتين، وإدراج العناصر الأخرى.

كاختصار، يمكنك استخدام عامل التشغيل `=^` بدلاً من ذلك، انظر المثال أدناه.

مثال:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x ^= y # x.symmetric_difference_update(y)
print(x)
# {'cherry', 'banana', 'google', 'microsoft'}
```

[union\(\)](#)

إرجاع مجموعة تحتوي على اتحاد المجموعات

تقوم طريقة `union()` بإرجاع مجموعة تحتوي على كافة العناصر من المجموعة الأصلية، وجميع العناصر من المجموعة (المجموعات) المحددة. يمكنك تحديد أي عدد تريده من المجموعات، مفصولة بفواصل. ليس من الضروري أن تكون مجموعة، يمكن أن تكون أي كائن قابل للتكرار. إذا كان العنصر موجوداً في أكثر من مجموعة واحدة، فستحتوي النتيجة على مظهر واحد فقط لهذا العنصر.

كاختصار، يمكنك استخدام `|` المشغل بدلاً من ذلك، انظر المثال أدناه.

مثال:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x | y # z = x.union(y)
print(z)
# {'microsoft', 'banana', 'google', 'cherry', 'apple'}
```

مثال:

```
x = {"a", "b", "c"}
y = {"f", "d", "a"}
z = {"c", "d", "e"}
result = x.union(y, z) # result = x | y | z
print(result)
# {'c', 'f', 'b', 'd', 'e', 'a'}
```

update()

قم بتحديث المجموعة باتحاد هذه المجموعة وغيرها.
يقوم أسلوب update() بتحديث المجموعة الحالية عن طريق إضافة عناصر من مجموعة أخرى (أو أي مجموعة أخرى قابلة للتكرار).
إذا كان هناك عنصر موجود في كلتا المجموعتين، فسيكون هناك مظهر واحد فقط لهذا العنصر في المجموعة المحدثة.
كاختصار، يمكنك استخدام عامل التشغيل |= بدلاً من ذلك، انظر المثال أدناه.

مثال:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x |= y #x.update(y)
print(x)
#{'cherry', 'banana', 'apple', 'google', 'microsoft'}
```

مثال:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = {"cherry", "micra", "bluebird"}
x |= y | z #x.update(y, z)
print(x)
#{'cherry', 'google', 'apple', 'microsoft', 'bluebird', 'micra', 'banana'}
```

انتهى الفصل

الفصل الرابع عشر

القواميس في البايثون

1-14- مقدمة:

تُستخدم القواميس لتخزين قيم البيانات في أزواج المفتاح: القيمة. القاموس عبارة عن مجموعة مرتبة* وقابلة للتغيير ولا تسمح بالتكرارات. تتم كتابة القواميس بأقواس معقوفة، ولها مفاتيح وقيم. مثال:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
```

مثال: قم بإنشاء قاموس وطباعة عناصره:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(thisdict)
```

مثال: اطبع قيمة الـ "brand" في القاموس:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(thisdict["brand"])
```

مرتب أم غير مرتب؟

اعتبارًا من الإصدار 3.7 من Python، تم ترتيب القواميس. في Python 3.6 والإصدارات الأقدم، القواميس غير مرتبة.

عندما نقول أن القواميس مرتبة، فهذا يعني أن العناصر لها ترتيب محدد، وهذا الترتيب لن يتغير.

• **Unordered غير مرتبة**

يعني أن العناصر ليس لها ترتيب محدد، ولا يمكنك الرجوع إلى عنصر باستخدام الفهرس.

• **Changeable قابل للتغيير**

القواميس قابلة للتغيير، مما يعني أنه يمكننا تغيير العناصر أو إضافتها أو إزالتها بعد إنشاء القاموس.

• **Duplicates Not Allowed التكرارات غير مسموح بها**

لا يمكن أن تحتوي القواميس على عنصرين بنفس المفتاح.

مثال: ستحل القيم المكررة محل القيم الموجودة:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964,
    "year": 2020
}
print(thisdict)
```

14-2- طول القاموس:

لتحديد عدد العناصر الموجودة في القاموس، استخدم الدالة len().

مثال: اطبع عدد العناصر في القاموس:

```
print(len(thisdict))
```

14-3- عناصر القاموس - أنواع البيانات:

يمكن أن تكون القيم الموجودة في عناصر القاموس من أي نوع بيانات.

مثال: أنواع بيانات String ، int ، boolean ، و list :

```
thisdict = {
    "brand": "Ford",
    "electric": False,
    "year": 1964,
    "colors": ["red", "white", "blue"]
}
```

• استخدام الطريقة type()

من وجهة نظر بايثون، يتم تعريف القواميس ككائنات بنوع البيانات "dict".

```
<class 'dict'>
```

مثال: طباعة نوع بيانات القاموس:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(type(thisdict))
```

14-4- الباني للقاموس:

من الممكن أيضًا استخدام المُنشئ dict() لإنشاء قاموس.

مثال:

```
thisdict = dict(name = "John", age = 36, country = "Norway")
print(thisdict)
```

14-5- الوصول إلى العناصر:

يمكنك الوصول إلى عناصر القاموس من خلال الإشارة إلى اسمه الرئيسي، داخل قوسين مربعين.

مثال: اجلب القيمة التي مفتاحها "model":

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = thisdict["model"]
```

• هناك أيضًا طريقة تسمى get() والتي ستعطيك نفس النتيجة.

مثال:

```
x = thisdict.get("model")
```

• احصل على المفاتيح:

ستعيد طريقة keys() قائمة بجميع المفاتيح الموجودة في القاموس.

مثال:

```
x = thisdict.keys()
```

قائمة المفاتيح هي عرض للقاموس، مما يعني أن أي تغييرات يتم إجراؤها على القاموس سوف تنعكس في

قائمة المفاتيح.

مثال: أضف عنصرًا جديدًا إلى القاموس الأصلي، وتأكد من تحديث قائمة المفاتيح أيضًا:

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.keys()
print(x) #before the change
car["color"] = "white"
print(x) #after the change
```

• الحصول على القيم:

سُيُرجع تابع values() قائمة بجميع القيم الموجودة في القاموس.

مثال:

```
x = thisdict.values()
```

قائمة القيم هي عرض للقاموس، مما يعني أن أي تغييرات يتم إجراؤها على القاموس سوف تنعكس في قائمة القيم.

مثال:

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.values()
print(x) #before the change
car["year"] = 2020
print(x) #after the change
```

مثال: أضف عنصرًا جديدًا إلى القاموس الأصلي، وتأكد من تحديث قائمة القيم أيضًا:

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.values()
print(x) #before the change
car["color"] = "red"
print(x) #after the change
```

• الحصول على العناصر:

سيُعيد التابع items() كل عنصر في القاموس على شكل صفوف في القائمة.

مثال: احصل على قائمة بأزواج المفتاح:القيمة

```
x = thisdict.items()
```

القائمة التي تم إرجاعها هي عرض لعناصر القاموس، مما يعني أن أي تغييرات يتم إجراؤها على القاموس سوف تنعكس في قائمة العناصر.

مثال: قم بإجراء تغيير في القاموس الأصلي، وتأكد من تحديث قائمة العناصر أيضاً:

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.items()
print(x) #before the change
car["year"] = 2020
print(x) #after the change
```

مثال: أضف عنصراً جديداً إلى القاموس الأصلي، وتأكد من تحديث قائمة العناصر أيضاً:

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.items()
print(x) #before the change
car["color"] = "red"
print(x) #after the change
```

• تحقق من وجود المفتاح:

لتحديد ما إذا كان هناك مفتاح محدد موجود في القاموس، استخدم الكلمة الأساسية `in`.

مثال: تحقق مما إذا كان "model" موجوداً في القاموس:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
if "model" in thisdict:
    print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

14-6- تغيير القيم:

يمكنك تغيير قيمة عنصر معين من خلال الإشارة إلى اسمه الرئيسي.

مثال: تغيير "year" إلى 2018:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
thisdict["year"] = 2018
```

14-7- تحديث القاموس:

سيقوم أسلوب update() بتحديث القاموس بالعناصر الموجودة في الوسيطة المحددة.

يجب أن تكون الوسيطة عبارة عن قاموس، أو كائن قابل للتكرار مع أزواج المفاتيح: القيمة.

مثال: قم بتحديث "year" السيارة باستخدام طريقة update() :

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
thisdict.update({"year": 2020})
```

14-8- إضافة العناصر:

تتم إضافة عنصر إلى القاموس باستخدام مفتاح فهرس جديد وتعيين قيمة له.

مثال:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
thisdict["color"] = "red"
print(thisdict)
```

14-9- إزالة العناصر:

هناك عدة طرق لإزالة العناصر من القاموس.

مثال: تقوم طريقة pop() بإزالة العنصر الذي يحمل اسم المفتاح المحدد:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
thisdict.pop("model")
print(thisdict)
```

مثال: يزيل التابع popitem() آخر عنصر تم إدراجه:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
thisdict.popitem()
print(thisdict)
```

مثال: تقوم الكلمة الأساسية del بإزالة العنصر الذي يحمل اسم المفتاح المحدد:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
del thisdict["model"]
print(thisdict)
```

مثال: يمكن للكلمة الأساسية del أيضاً حذف القاموس بالكامل:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
del thisdict
print(thisdict)
#this will cause an error because "thisdict" no longer exists.
```

مثال: تقوم الطريقة clear() بإفراغ القاموس:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
thisdict.clear()
print(thisdict)
```

10-14- حلقة من خلال القاموس:

يمكنك تكرار عبور العناصر في القاموس باستخدام حلقة `for`.
عند التكرار عبر القاموس، تكون القيمة المرجعة هي مفاتيح القاموس، ولكن هناك طرق لإرجاع القيم.
مثال: اطبع جميع أسماء المفاتيح في القاموس، واحدًا تلو الآخر:

```
for x in thisdict:
    print(x)
```

مثال: اطبع جميع القيم الموجودة في القاموس واحدة تلو الأخرى:

```
for x in thisdict:
    print(thisdict[x])
```

مثال: يمكنك أيضًا استخدام التابع `values()` لإرجاع قيم القاموس:

```
for x in thisdict.values():
    print(x)
```

مثال: يمكنك استخدام طريقة `keys()` لإرجاع مفاتيح القاموس:

```
for x in thisdict.keys():
    print(x)
```

مثال: قم بالتكرار عبر المفاتيح والقيم باستخدام طريقة `items()`:

```
for x, y in thisdict.items():
    print(x, y)
```

11-14- نسخ القاموس:

لا يمكنك نسخ قاموس ببساطة عن طريق كتابة `dict2 = dict1`، لأن `dict2` سيكون مرجعًا فقط إلى `dict1`، والتغييرات التي تم إجراؤها في `dict1` سيتم إجراؤها تلقائيًا في `dict2` أيضًا.
هناك طرق لإنشاء نسخة، إحدى الطرق هي استخدام طريقة `copy()`.
مثال:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
mydict = thisdict.copy()
print(mydict)
```

هناك طريقة أخرى لعمل نسخة وهي استخدام الدالة `dict()`.

مثال:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
mydict = dict(thisdict)
print(mydict)
```

14-12- قواميس متداخلة:

يمكن أن يحتوي القاموس على قواميس، وهذا ما يسمى القواميس المتداخلة.

مثال: إنشاء قاموس يحتوي على ثلاثة قواميس:

```
myfamily = {
    "child1" : {
        "name" : "Emil",
        "year" : 2004
    },
    "child2" : {
        "name" : "Tobias",
        "year" : 2007
    },
    "child3" : {
        "name" : "Linus",
        "year" : 2011
    }
}
```

أو، إذا كنت تريد إضافة ثلاثة قواميس إلى قاموس جديد.

مثال: قم بإنشاء ثلاثة قواميس، ثم قم بإنشاء قاموس واحد يحتوي على القواميس الثلاثة الأخرى:

```
child1 = { "name" : "Emil", "year" : 2004 }
child2 = { "name" : "Tobias", "year" : 2007 }
child3 = { "name" : "Linus", "year" : 2011 }
myfamily = { "child1" : child1, "child2" : child2, "child3" : child3 }
```

• الوصول إلى العناصر في القواميس المتداخلة:

للوصول إلى عناصر من قاموس متداخل، يمكنك استخدام اسم القواميس، بدءًا من القاموس الخارجي.

مثال: طباعة اسم الطفل 2:

```
print(myfamily["child2"]["name"])
```

مثال: لنفترض أنك تدير نظامًا لإدارة معلومات الطلاب في مدرسة. يحتوي النظام على قواميس

متداخلة لتخزين معلومات كل طالب، مثل اسم الطالب، صفه الدراسي، ودرجاته في المواد المختلفة:

قاموس يحتوي على بيانات الطلاب

```
students={
    "student_1":{
        "name":"Ali Mohammad",
        "grade":90,
        "scores":{"math": 95 , "science":85 , "English": 90 }
    },
}
```

```

"student_2":{
    "name":"Ali Mohammad2",
    "grade":90,
    "scores":{"math": 95 , "science":85 , "English": 90 }
},
"student_3":{
    "name":"Ali Mohammad3",
    "grade":90,
    "scores":{"math": 95 , "science":85 , "English": 90 }
}
}

```

طباعة كل معلومات الطالب

```

for student_id,student_info in students.items() :
    print("student ID:", student_id )
    print("Name:", student_info["name"] )
    print("Grade:", student_info["grade"] )
    for subject,score in student_info["scores"].items():
        print(subject,score)

```

الوصول الى درجة معينة

```

mark_english2=students["student_2"]
print(mark_english2)

```

```

mark_english2=students["student_2"]["scores"]
print(mark_english2)

```

```

mark_english2=students["student_2"]["scores"]["English"]
print(mark_english2)

```

13-14- طرق القاموس:

لدى بايثون مجموعة من الأساليب المضمنة التي يمكنك استخدامها في القواميس.

الوصف	التابع
يزيل كافة العناصر من القاموس.	clear()
إرجاع نسخة من القاموس.	copy()
إرجاع قاموس بالمفاتيح والقيمة المحددة. مثال: قم بإنشاء قاموس بثلاثة مفاتيح، جميعها بالقيمة 0:	fromkeys()

```
x = ('key1', 'key2', 'key3')
y = 0
thisdict = dict.fromkeys(x, y)
print(thisdict)
# {'key1': 0, 'key2': 0, 'key3': 0}
```

التعريف والاستخدام:

تقوم طريقة `fromkeys()` بإرجاع قاموس يحتوي على المفاتيح المحددة والقيمة المحددة.
القاعدة العامة:

```
dict.fromkeys(keys, value)
```

مثال: نفس المثال المذكور أعلاه، ولكن دون تحديد القيمة:

```
x = ('key1', 'key2', 'key3')
thisdict = dict.fromkeys(x)
print(thisdict)
```

[`get\(\)`](#)

إرجاع قيمة المفتاح المحدد.

[`items\(\)`](#)

تقوم بإرجاع قائمة تحتوي على صف لكل زوج من قيم المفاتيح.

[`keys\(\)`](#)

إرجاع قائمة تحتوي على مفاتيح القاموس.

[`pop\(\)`](#)

إزالة العنصر بالمفتاح المحدد.

[`popitem\(\)`](#)

يزيل آخر زوج من قيمة-المفتاح الذي تم إدراجه.

مثال: إزالة العنصر الأخير من القاموس:

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
car.popitem()
print(car)
```

تقوم طريقة `popitem()` بإزالة العنصر الذي تم إدراجه آخر مرة في القاموس. في الإصدارات قبل 3.7، يقوم التابع بإزالة عنصر عشوائي.

العنصر المحذوف هو القيمة المرجعة لطريقة `popitem()`، على شكل صف، انظر المثال أدناه.

القاعدة العامة :

```
dictionary.popitem()
```

مثال:

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.popitem()
print(x)
```

[setdefault\(\)](#) إرجاع قيمة المفتاح المحدد. إذا كان المفتاح غير موجود: أدخل المفتاح بالقيمة المحددة
مثال: احصل على قيمة عنصر "model":

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964 }
x = car.setdefault("model", "Bronco")
print(x)
```

التعريف والاستخدام:

تقوم طريقة `setdefault()` بإرجاع قيمة العنصر بالمفتاح المحدد.
في حالة عدم وجود المفتاح، أدخل المفتاح بالقيمة المحددة، انظر المثال أدناه

القاعدة العامة:

```
dictionary.setdefault(keyname, value)
```

مثال: احصل على قيمة العنصر "color"، إذا كان العنصر "color" غير موجود، أدخل "color"
بالقيمة "white":

```
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.setdefault("color", "white")
print(x)
```

[update\(\)](#) يقوم بتحديث القاموس بأزواج القيمة الرئيسية المحددة

[values\(\)](#) إرجاع قائمة بجميع القيم الموجودة في القاموس

انتهى الفصل

الفصل الخامس عشر

الشرط في البايثون

15-1- مقدمة:

تدعم بايثون الشروط المنطقية المعتادة من الرياضيات:

- يساوي: $B == A$
- لا يساوي: $B != A$
- أقل من: $B > A$
- أقل من أو يساوي: $B <= A$
- أكبر من: $B < A$
- أكبر من أو يساوي: $B <= A$

يمكن استخدام هذه الشروط بعدة طرق، الأكثر شيوعاً في "عبارات if" والحلقات.

مثال:

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

في هذا المثال، نستخدم متغيرين، a و b، يتم استخدامهما كجزء من عبارة if لاختبار ما إذا كان b أكبر من a.

15-2- المسافة الفارغة:

تعتمد بايثون على المسافة البادئة (مسافة بيضاء في بداية السطر) لتحديد النطاق في الكود. غالباً ما تستخدم لغات البرمجة الأخرى الأقواس المتعرجة لهذا الغرض.

مثال: إذا كانت العبارة، بدون مسافة بادئة (ستؤدي إلى حدوث خطأ):

```
a = 33
b = 200
if b > a:
    print("b is greater than a") # هنا سيكون الخطأ بسبب عدم وجود فراغ بداية التعليمة
```

elif -3-15:

الكلمة الأساسية elif هي طريقة بايثون للقول "إذا لم تكن الشروط السابقة صحيحة، فجرب هذا الشرط".

مثال:

```
a = 33
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
```

في هذا المثال، a يساوي b، وبالتالي فإن الشرط الأول ليس صحيحًا، ولكن شرط elif صحيح، لذلك نطبع على الشاشة أن "a و b متساويان".

else -4-15:

تلتقط الكلمة الأساسية else أي شيء لم يتم اكتشافه بالشروط السابقة.

مثال:

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

في هذا المثال a أكبر من b، وبالتالي فإن الشرط الأول غير صحيح، وكذلك شرط elif غير صحيح، لذلك ننتقل إلى الشرط else ونطبع على الشاشة أن "a أكبر من b".

مثال:

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

5-15- اليد القصيرة للشرط Short Hand If:

إذا كان لديك عبارة واحدة فقط تريد تنفيذها، فيمكنك وضعها في نفس سطر عبارة if.

مثال:

```
if a > b: print("a is greater than b")
```

15-6- اليد القصيرة للشرط Short Hand If .. else :

إذا كان لديك عبارة واحدة فقط تريد تنفيذها، وواحدة لـ if، وواحدة لـ else، فيمكنك وضعها كلها في نفس السطر.
مثال: سطر واحد إذا عبارة أخرى:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

تُعرف هذه التقنية باسم عوامل التشغيل الثلاثية أو التعبيرات الشرطية.
يمكنك أيضاً الحصول على عدة عبارات أخرى في نفس السطر.
مثال:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

15-7- المعامل and مع عبارة ال if:

تعد الكلمة الأساسية and عاملاً منطقيًا، وتُستخدم للجمع بين العبارات الشرطية.
مثال: اختبر ما إذا كان a أكبر من b، وإذا كان c أكبر من a:

```
a = 200
b = 33
c = 500
if a > b and c > a:
    print("Both conditions are True")
```

15-8- المعامل or مع عبارة ال if:

الكلمة الأساسية أو هي عامل منطقي، وتستخدم للجمع بين العبارات الشرطية.
مثال: اختبار إذا كان a أكبر من b، أو إذا كان a أكبر من c:

```
a = 200
b = 33
c = 500
if a > b or a > c:
    print("At least one of the conditions is True")
```

15-9- المعامل Not مع عبارة ال if:

الكلمة الأساسية not هي عامل منطقي، وتستخدم لعكس نتيجة العبارة الشرطية.
مثال: اختبار ما إذا كان a ليس أكبر من b:

```
a = 33
b = 200
if not a > b:
    print("a is NOT greater than b")
```

10-15- الشروط المتداخلة:

يمكن أن يكون لديك عبارات if داخل عبارات if، وهذا ما يسمى عبارات if المتداخلة.
مثال:

```
x = 41
if x > 10:
    print("Above ten,")
    if x > 20:
        print("and also above 20!")
    else:
        print("but not above 20.")
```

11-15- بيان المرور:

إذا لم يكن من الممكن أن تكون العبارات فارغة، ولكن إذا كان لديك لسبب ما عبارة if بدون محتوى،
فضع عبارة pass لتجنب الحصول على خطأ.
مثال:

```
a = b = 33
if b > a:
    pass
```

انتهى الفصل

الفصل السادس عشر

الحلقات في البايثون

16-1- مقدمة:

لدى بايثون أمرين بدائيين للحلقة:

- حلقات while
- حلقات for

16-2- حلقة الـ while:

باستخدام حلقة while يمكننا تنفيذ مجموعة من العبارات طالما كان الشرط صحيحًا.

مثال: اطبع i طالما أنني أقل من 6:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

ملحوظة: تذكر أن تزيد i، وإلا ستستمر الحلقة إلى الأبد.

تتطلب حلقة while المتغيرات ذات الصلة لتكون جاهزة، في هذا المثال نحتاج إلى تعريف متغير الفهرسة، i، الذي قمنا بتعيينه على 1.

16-3- جملة الـ break مع الـ while:

باستخدام عبارة Break يمكننا إيقاف الحلقة حتى لو كان شرط while صحيحًا.

مثال: الخروج من الحلقة عندما يكون I أصغر من 3:

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

16-4- جملة الـ continue مع الـ while:

باستخدام عبارة continue يمكننا إيقاف التكرار الحالي والاستمرار في التالي.

مثال: تابع إلى التكرار التالي إذا كان $i = 3$:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

16-5- جملة الـ else مع الـ while:

باستخدام عبارة else يمكننا تشغيل كتلة من التعليمات البرمجية مرة واحدة عندما يصبح الشرط غير صحيح.

مثال: اطبع رسالة عندما يكون الشرط خاطئاً:

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

16-6- جملة الـ For:

يتم استخدام حلقة for للتكرار عبر تسلسل (إما قائمة، أو صف، أو قاموس، أو مجموعة، أو سلسلة). هذا أقل تشابهاً مع الكلمة الأساسية for في لغات البرمجة الأخرى، ويعمل بشكل أشبه بطريقة التكرار كما هو موجود في لغات البرمجة الموجهة للكائنات الأخرى.

باستخدام حلقة for، يمكننا تنفيذ مجموعة من العبارات، مرة واحدة لكل عنصر في القائمة، أو صف، أو مجموعة، وما إلى ذلك.

مثال: اطبع كل فاكهة في قائمة الفاكهة:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

- لا تتطلب حلقة for تعيين متغير فهرسة مسبقاً.
 - حلقات من خلال سلسلة
 - حتى السلاسل النصية هي كائنات قابلة للتكرار، فهي تحتوي على سلسلة من الأحرف.
- مثال: قم بالتمرير على الحروف الموجودة في كلمة "banana":

```
for x in "banana":
    print(x)
```

16-7- جملة الـ break مع الـ for:

باستخدام عبارة Break يمكننا إيقاف الحلقة قبل أن يتم تكرارها عبر جميع العناصر.

مثال: اخرج من الحلقة عندما تكون x هي "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

مثال: اخرج من الحلقة عندما تكون x هي "banana"، ولكن هذه المرة يأتي الفاصل قبل الطباعة:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        break
    print(x)
```

16-8- جملة الـ continue مع الـ for:

باستخدام عبارة continue يمكننا إيقاف التكرار الحالي للحلقة، والاستمرار في العبارة التالية.

مثال: لا تطبع banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

16-9- التابع range():

للتكرار عبر مجموعة من التعليمات البرمجية لعدد محدد من المرات، يمكننا استخدام الدالة range().

ترجع الدالة range() سلسلة من الأرقام، تبدأ من 0 افتراضياً، وتزيد بمقدار 1 (افتراضياً)، وتنتهي عند رقم محدد.

مثال:

```
for x in range(6):
    print(x)
```

لاحظ أن range(6) ليس القيم من 0 إلى 6، بل القيم من 0 إلى 5.

القيمة الافتراضية للدالة range() هي 0 كقيمة البداية، ومع ذلك فمن الممكن تحديد قيمة البداية عن طريق

إضافة معلمة: range(2,6)، والتي تعني القيم من 2 إلى 6 (ولكن لا تشمل 6).

مثال:

```
for x in range(2, 6):
    print(x)
```

تقوم الدالة `range()` افتراضياً بزيادة التسلسل بمقدار 1، ومع ذلك فمن الممكن تحديد قيمة الزيادة عن طريق إضافة معلمة ثالثة `range(2, 30, 3)` :
مثال: قم بزيادة التسلسل بـ 3 (الافتراضي هو 1):

```
for x in range(2, 30, 3):
    print(x)
```

10-16- جملة الـ else مع الـ for:

تحدد الكلمة الأساسية `else` في حلقة `for` كتلة التعليمات البرمجية التي سيتم تنفيذها عند انتهاء الحلقة.
مثال: اطبع جميع الأرقام من 0 إلى 5، واطبع رسالة عند انتهاء الحلقة:

```
for x in range(6):
    print(x)
else:
    print("Finally finished!")
```

ملاحظة: لن يتم تنفيذ الكتلة `else` إذا تم إيقاف الحلقة بواسطة عبارة `break`.
مثال: اكسر الحلقة عندما تكون `x` تساوي 3، وانظر ماذا يحدث مع الكتلة `else`:

```
for x in range(6):
    if x == 3: break
    print(x)
else:
    print("Finally finished!")
```

11-16- حلقات متداخلة مع الـ for:

الحلقة المتداخلة هي حلقة داخل حلقة. سيتم تنفيذ "الحلقة الداخلية" مرة واحدة لكل تكرار للحلقة الخارجية.
مثال: اطبع كل صفة لكل فاكهة:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
    for y in fruits:
        print(x, y)
```

12-16- جملة الـ pass مع الـ for:

لا يمكن أن تكون حلقات `for` فارغة، ولكن إذا كان لديك لسبب ما حلقة `for` بدون محتوى، فقم بإدخال عبارة `pass` لتجنب حدوث خطأ.

مثال:

```
for x in [0, 1, 2]:
    pass
```

انتهى الفصل

الفصل السابع عشر

التوابع في البايثون

17-1- مقدمة:

الوظيفة عبارة عن كتلة من التعليمات البرمجية التي تعمل فقط عند **استدعائها**. يمكنك تمرير البيانات، المعروفة باسم **المعلمات**، إلى دالة. يمكن للوظيفة **إرجاع البيانات** نتيجة لذلك.

17-2- إنشاء وظيفة:

يتم تعريف الدالة في لغة Python باستخدام الكلمة الأساسية def. **مثال:**

```
def my_function():
    print("Hello from a function")
```

17-3- استدعاء الوظيفة:

لاستدعاء دالة، استخدم اسم الدالة متبوعًا بقوسين. **مثال:**

```
def my_function():
    print("Hello from a function")
my_function()
```

17-4- الوسيطات Arguments:

يمكن تمرير المعلومات إلى الوظائف **كوسائط**. يتم تحديد الوسائط بعد اسم الوظيفة، داخل الأقواس. يمكنك إضافة أي عدد تريده من الوسائط، ما عليك سوى الفصل بينها بفاصلة. يحتوي المثال التالي على دالة ذات **وسيط** واحدة (fname). عندما يتم استدعاء الدالة، نقوم بتمرير الاسم الأول، والذي يتم استخدامه داخل الدالة لطباعة الاسم الكامل.

مثال:

```
def my_function(fname):
    print(fname + " Refsnes")
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

المعلومات (البارامترات) أو الوسيطات؟

يمكن استخدام المعامل والوسيط لنفس الشيء: المعلومات التي يتم تمريرها إلى دالة.

من منظور الوظيفة:

- المعامل **parameter** هي المتغير المدرج داخل الأقواس في تعريف الدالة.
- الوسيطة **argument** هي القيمة التي يتم إرسالها إلى الدالة عند استدعائها.

عدد الوسيطات Number of Arguments:

بشكل افتراضي، يجب استدعاء الدالة بالعدد الصحيح من الوسائط. وهذا يعني أنه إذا كانت وظيفتك تتوقع وسيطتين، فيجب عليك استدعاء الدالة باستخدام وسيطتين، وليس أكثر ولا أقل.

مثال: تتوقع هذه الدالة وسيطتين، وتحصل على وسيطتين:

```
def my_function(fname, lname):
    print(fname + " " + lname)
my_function("Emil", "Refsnes")
```

إذا حاولت استدعاء الدالة باستخدام وسيطة واحدة أو ثلاث وسيطات، فسوف تحصل على خطأ.

مثال: تتوقع هذه الدالة وسيطتين، ولكنها تحصل على وسيطة واحدة فقط:

```
def my_function(fname, lname):
    print(fname + " " + lname)
my_function("Emil")
```

الوسطاء التعسفية، *args, Arbitrary Arguments:

إذا كنت لا تعرف عدد الوسائط التي سيتم تمريرها إلى وظيفتك، أضف * قبل اسم المعلمة في تعريف الوظيفة. بهذه الطريقة ستتلقى الدالة مجموعة من الوسائط، ويمكنها الوصول إلى العناصر وفقاً لذلك.

مثال: إذا كان عدد الوسائط غير معروف، أضف * قبل اسم المعلمة:

```
def my_function(*kids):
    print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

وسيطات الكلمات الرئيسية Keyword Arguments

يمكنك أيضاً إرسال الوسائط باستخدام صيغة المفتاح = القيمة. بهذه الطريقة لا يهم ترتيب الوسيطات.
مثال:

```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

وسيطات الكلمات الرئيسية التعسفية:

إذا كنت لا تعرف عدد وسائط الكلمات الرئيسية التي سيتم تمريرها إلى وظيفتك، أضف علامتين نجميتين: ** قبل اسم المعلمة في تعريف الوظيفة. بهذه الطريقة ستتلقى الدالة قاموساً للوسائط، ويمكنها الوصول إلى العناصر وفقاً لذلك.

مثال: إذا كان عدد وسيطات الكلمات الرئيسية غير معروف، أضف علامة مزدوجة ** قبل اسم المعلمة:

```
def my_function(**kid):
    print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")
```

قيمة المعلمة الافتراضية Default Parameter Value

يوضح المثال التالي كيفية استخدام قيمة المعلمة الافتراضية. إذا قمنا باستدعاء الدالة بدون وسيطة، فإنها تستخدم القيمة الافتراضية.

مثال:

```
def my_function(country = "Norway"):
    print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

تمرير القائمة كوسيط Passing a List as an Argument

يمكنك إرسال أي نوع من أنواع البيانات إلى دالة (سلسلة أو رقم أو قائمة أو قاموس وما إلى ذلك)، وسيتم التعامل معها على أنها نفس نوع البيانات داخل الوظيفة. على سبيل المثال إذا أرسلت قائمة كوسيط، فستظل قائمة عندما تصل إلى الوظيفة.

مثال:

```
def my_function(food):
    for x in food:
        print(x)
fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

17-5- إرجاع القيم Return Values:

للسماح للدالة بإرجاع قيمة، استخدم عبارة الإرجاع.

مثال:

```
def my_function(x):
    return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

17-6- بيان المرور The pass Statement:

لا يمكن أن تكون تعريفات الدالة فارغة، ولكن إذا كان لديك تعريف دالة بدون محتوى لسبب ما، فقم بإدخال عبارة المرور لتجنب حدوث خطأ.

مثال:

```
def myfunction():
    pass
```

17-7- الوسيطات الموضعية فقط Positional-Only Arguments:

يمكنك تحديد أن الدالة يمكن أن تحتوي على وسيطات موضعية فقط، أو وسيطات كلمات رئيسية فقط. لتحديد أن الدالة يمكن أن تحتوي على وسائط موضعية فقط، أضف / بعد الوسائط.

مثال:

```
def my_function(x, /):
    print(x)
my_function(3)
```

بدون ، / يُسمح لك فعليًا باستخدام وسيطات الكلمات الرئيسية حتى لو كانت الوظيفة تتوقع وسيطات موضعية.

مثال:

```
def my_function(x):
    print(x)
my_function(x = 3)
```

ولكن عند إضافة /، سوف تحصل على خطأ إذا حاولت إرسال وسيطة الكلمة الرئيسية.

مثال:

```
def my_function(x, /):
    print(x)
my_function(x = 3) ##error
```

وسيطات الكلمات الرئيسية فقط Keyword-Only Arguments

لتحديد أن الدالة يمكن أن تحتوي على وسائط الكلمات الرئيسية فقط، أضف * قبل الوسائط.

مثال:

```
def my_function(*, x):  
    print(x)  
my_function(x = 3)
```

بدون *, يُسمح لك باستخدام وسيطات الموضع حتى إذا كانت الدالة تتوقع وسيطات الكلمات الرئيسية.

مثال:

```
def my_function(x):  
    print(x)  
my_function(3)
```

ولكن عند إضافة *, / سوف تحصل على خطأ إذا حاولت إرسال وسيطة موضعية.

مثال:

```
def my_function(*, x):  
    print(x)  
my_function(3)## error
```

: Combine Positional-Only and Keyword-Only

يمكنك الجمع بين نوعي الوسيطة في نفس الوظيفة. أي وسيطة قبل / تكون موضعية فقط، وأي وسيطة بعد * تكون مخصصة للكلمات الرئيسية فقط.

مثال:

```
def my_function(a, b, /, *, c, d):  
    print(a + b + c + d)  
my_function(5, 6, c = 7, d = 8)
```

8-17- العودية:

تقبل بايثون أيضًا عودية الوظيفة، مما يعني أن الوظيفة المحددة يمكنها استدعاء نفسها.

العودية هي مفهوم رياضي وبرمجي شائع. وهذا يعني أن وظيفة تستدعي نفسها. وهذا له فائدة أنه يمكنك تكرار البيانات للوصول إلى نتيجة.

يجب أن يكون المطور حذرًا للغاية فيما يتعلق بالتكرار لأنه قد يكون من السهل جدًا الانزلاق إلى كتابة وظيفة لا تنتهي أبدًا، أو وظيفة تستخدم كميات زائدة من الذاكرة أو طاقة المعالج. ومع ذلك، عند كتابتها بشكل صحيح، يمكن أن تكون العودية طريقة فعالة للغاية وأنيقة رياضياً للبرمجة.

في هذا المثال، `tri_recursion()` هي دالة قمنا بتعريفها لتسمي نفسها ("`recurse`"). نحن نستخدم المتغير `k` كالبينات، والذي يتناقص (-1) في كل مرة نكرر فيها. تنتهي العودية عندما لا يكون الشرط أكبر من 0 (أي عندما يكون 0).

بالنسبة للمطور الجديد، قد يستغرق الأمر بعض الوقت لمعرفة كيفية عمل ذلك بالضبط، وأفضل طريقة لمعرفة ذلك هي اختباره وتعديله.

مثال:

```
def tri_recursion(k):
    if(k > 0):
        result = k + tri_recursion(k - 1)
        print(result)
    else:
        result = 0
    return result
print("\n\nRecursion Example Results")
tri_recursion(6)
```

الخرج:

```
Recursion Example Results
1
3
6
10
15
21
```

مثال: حساب المضروب (Factorial) باستخدام التكرار:

```
def factorial(n):
    # الحالة الأساسية: إذا كان n يساوي 0 أو 1، نرجع 1
    if n == 0 or n == 1:
        return 1
    # استدعاء التابع بشكل تعاودي
    else:
        return n * factorial(n - 1)
    # اختبار التابع
print(factorial(5)) # الناتج 120
```

مثال: حساب متتالية فيبوناتشي (Fibonacci) باستخدام التكرار:

```
def fibonacci(n):
    # الحالة الأساسية: إذا كان n يساوي 0 نرجع 0، وإذا كان يساوي 1 نرجع 1
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # استدعاء التابع بشكل تعاودي
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
    # اختبار التابع
print(fibonacci(6))    ##output: 8
```

17-9- وظيفة لامدا:

وظيفة لامدا هي وظيفة مجهولة صغيرة. يمكن لدالة لامدا أن تأخذ أي عدد من الوسائط، ولكن يمكن أن تحتوي

على تعبير واحد فقط.

في لغة البرمجة بايثون، `lambda` هي طريقة لإنشاء توابع صغيرة ومؤقتة تُعرف أحياناً بـ "التوابع المجهولة" (anonymous functions) لأنها لا تحتاج إلى تعريف اسم محدد. تُستخدم التوابع `lambda` عادةً في الحالات التي تحتاج فيها إلى تابع بسيط ومؤقت، خاصةً داخل تعبيرات أو كمدخلات لتوابع أخرى.

القاعدة العامة:

`lambda arguments : expression`

- يتم تنفيذ التعبير وإرجاع النتيجة.

مثال: أضف 10 إلى الوسيطة `a`، وقم بإرجاع النتيجة:

```
x = lambda a : a + 10
print(x(5))# 15
```

- يمكن لوظائف `Lambda` أن تأخذ أي عدد من الوسائط.

مثال: اضرب الوسيطة `a` بالوسيطة `b` وأرجع النتيجة:

```
x = lambda a, b : a * b
print(x(5, 6))##30
```

مثال: اجمع الوسيطات `a` و `b` و `c` وأرجع النتيجة:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))##13
```

لماذا نستخدم وظائف `Lambda`؟

تظهر قوة لامدا بشكل أفضل عند استخدامها كوظيفة مجهولة داخل وظيفة أخرى. لنفترض أن لديك تعريف دالة يأخذ وسيطة واحدة، وسيتم ضرب هذه الوسيطة برقم غير معروف.

```
def myfunc(n):
    return lambda a : a * n
```

استخدم تعريف الوظيفة هذا لإنشاء دالة تضاعف دائماً الرقم الذي ترسله.

مثال:

```
def myfunc(n):
    return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))##22
```

أو استخدم نفس تعريف الدالة لإنشاء دالة تضاعف دائماً الرقم الذي ترسله ثلاث مرات.

مثال:

```
def myfunc(n):
    return lambda a : a * n
mytripler = myfunc(3)
print(mytripler(11))##33
```

أو استخدم نفس تعريف الوظيفة لإجراء كلتاوظيفتين في نفس البرنامج.

مثال:

```
def myfunc(n):
    return lambda a : a * n
mydoubler = myfunc(2)
mytripler = myfunc(3)
print(mydoubler(11))##22
print(mytripler(11))##33
```

استخدم وظائف لامدا عندما تكون هناك حاجة إلى وظيفة مجهولة لفترة قصيرة من الزمن.

مثال: جمع عددين باستخدام lambda:

```
sum = lambda a, b: a + b
print(sum(5, 3))
# الناتج 8
```

استخدام lambda في تابع map لتحويل قائمة من الأعداد بمضاعفة كل عنصر:

```
numbers = [1, 2, 3, 4, 5]
doubled = list(map(lambda x: x * 2, numbers))
print(doubled)
# الناتج [2, 4, 6, 8, 10]
```

استخدام lambda في تابع filter لترشيح قائمة من الأعداد والحصول فقط على الأعداد الزوجية:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens) # الناتج [2, 4, 6, 8, 10]
```

استخدام lambda في تابع sorted لترتيب قائمة من القواميس بناءً على قيمة مفتاح محدد:

```
students = [
    {'name': 'Ahmed', 'age': 21}, {'name': 'Sara', 'age': 19}, {'name': 'Huda', 'age': 22}
]
sorted_students = sorted(students, key=lambda student: student['age'])
print(sorted_students)
# الناتج [{'name': 'Sara', 'age': 19}, {'name': 'Ahmed', 'age': 21}, {'name': 'Huda', 'age': 22}]
```


مميزات وعيوب التوابع λ :

المميزات:

- البساطة والاختصار: يمكن تعريف التوابع البسيطة بسرعة دون الحاجة إلى تعريفات متعددة الأسطر.
- المرونة: تُستخدم بشكل كبير مع التوابع التي تتطلب مدخلات من نوع تابع مثل: `map` و `filter` و `sorted`.

العيوب:

- محدودية الوظائف: يمكن أن تحتوي فقط على تعبير واحد وليس سلسلة من الأوامر.
 - صعوبة القراءة: قد تكون أقل وضوحًا من التوابع المسماة عندما تكون التعابير معقدة.
 - إعادة الاستخدام: غير مريحة للاستخدام في حال الحاجة إلى إعادة استخدامها في مواضع متعددة.
- بشكل عام، تعد التوابع λ أداة مفيدة جدًا في بايثون، خاصةً عندما تحتاج إلى حلول سريعة وبسيطة ضمن سياقات محددة.

انتهى الفصل

الفصل الثامن عشر

المصفوفات في البايثون

18-1-مقدمة:

لا تحتوي لغة Python على دعم مدمج للمصفوفات، ولكن يمكن استخدام قوائم Python بدلاً من ذلك. يوضح هذا الفصل كيفية استخدام القوائم كمصفوفات، ومع ذلك، للعمل مع المصفوفات في Python، سيتعين عليك استيراد مكتبة، مثل مكتبة NumPy. تُستخدم المصفوفات لتخزين قيم متعددة في متغير واحد.

18-2-إنشاء المصفوفة:

مثال: قم بإنشاء مصفوفة تحتوي على أسماء السيارات:

```
cars = ["Ford", "Volvo", "BMW"]
```

ما هي المصفوفة؟

المصفوفة عبارة عن متغير خاص يمكنه الاحتفاظ بأكثر من قيمة واحدة في المرة الواحدة. إذا كانت لديك قائمة بالعناصر (قائمة بأسماء السيارات، على سبيل المثال)، فقد يبدو تخزين السيارات في متغيرات فردية كما يلي.

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
```

ومع ذلك، ماذا لو كنت تريد التنقل بين السيارات والعثور على سيارة معينة؟ وماذا لو لم يكن لديك 3 سيارات بل 300؟

الحل هو مصفوفة!

يمكن للمصفوفة أن تحتوي على العديد من القيم تحت اسم واحد، ويمكنك الوصول إلى القيم عن طريق الإشارة إلى رقم الفهرس.

18-3- الوصول إلى عناصر المصفوفة:

يمكنك الرجوع إلى عنصر صفيف بالإشارة إلى رقم الفهرس.

مثال: احصل على قيمة عنصر المصفوفة الأول:

```
x = cars[0]
```

مثال: تعديل قيمة عنصر المصفوفة الأول:

```
cars[0] = "Toyota"
```

18-4- طول المصفوفة:

استخدم طريقة len() لإرجاع طول المصفوفة (عدد العناصر في المصفوفة).

مثال: إرجاع عدد العناصر في مجموعة cars:

```
x = len(cars)
```

ملاحظة: يكون طول المصفوفة دائماً أكبر من مؤشر المصفوفة الأعلى بمقدار واحد.

18-5- تكرار عناصر المصفوفة:

يمكنك استخدام حلقة for in للتنقل عبر جميع عناصر المصفوفة.

مثال: طباعة كل عنصر في مجموعة cars:

```
for x in cars:  
    print(x)
```

18-6- إضافة عناصر المصفوفة:

يمكنك استخدام طريقة append() لإضافة عنصر إلى المصفوفة.

مثال: أضف عنصراً آخر إلى مصفوفة cars:

```
cars.append("Honda")
```

18-7- إزالة عناصر المصفوفة:

يمكنك استخدام طريقة pop() لإزالة عنصر من المصفوفة.

مثال: احذف العنصر الثاني من مصفوفة cars:

```
cars.pop(1)
```

يمكنك أيضاً استخدام طريقة remove() لإزالة عنصر من المصفوفة.

مثال: احذف العنصر الثاني احذف العنصر الذي له القيمة "Volvo": من مصفوفة cars:

```
cars.remove("Volvo")
```

ملاحظة: تقوم طريقة remove() الخاصة بالقائمة بإزالة التواجد الأول للقيمة المحددة فقط.

18-8- طرق المصفوفة:

لدى Python مجموعة من الأساليب المضمنة التي يمكنك استخدامها في القوائم/المصفوفات.

الطريقة	الوصف
clear()	يزيل كافة العناصر من القائمة
copy()	إرجاع نسخة من القائمة
count()	يُرجع عدد العناصر ذات القيمة المحددة
extend()	أضف عناصر القائمة (أو أي عنصر قابل للتكرار) إلى نهاية القائمة الحالية
index()	يُرجع فهرس العنصر الأول بالقيمة المحددة
insert()	يضيف عنصراً في الموضع المحدد
reverse()	يعكس ترتيب القائمة
sort()	يفرز القائمة

ملاحظة: لا تحتوي لغة Python على دعم مدمج للمصفوفات، ولكن يمكن استخدام قوائم Python بدلاً من ذلك.

انتهى الفصل

الفصل التاسع عشر

الأصناف والكائنات في البايثون

19-1- مقدمة:

البايثون هي لغة برمجة كائنية التوجه. كل شيء تقريبًا في بايثون هو كائن، له خصائصه وأساليبه. تشبه الفئة منشئ الكائن، أو "مخطط" لإنشاء الكائنات.

19-2- إنشاء صنف:

لإنشاء صنف ، استخدم الكلمة الأساسية `class` .
مثال: أنشئ صنف اسمه `MyClass` وخاصية `x`:

```
class MyClass:
    x = 5
```

19-3- إنشاء كائن:

يمكننا الآن استخدام الفئة المسماة `MyClass` لإنشاء كائنات.
مثال: أنشئ كائن اسمه `p1` واطبع قيمة `x`:

```
p1 = MyClass()
print(p1.x)
```

19-4- الدالة `__init__()`:

الأمثلة المذكورة أعلاه هي فئات (أصناف) وكائنات في أبسط أشكالها، وليست مفيدة حقًا في تطبيقات الحياة الواقعية. لفهم معنى الفئات علينا أن نفهم وظيفة `__init__()` المضمنة. تحتوي كافة الفئات على وظيفة تسمى `__init__()` ، والتي يتم تنفيذها دائمًا عند بدء الصنف. استخدم الدالة `__init__()` لتعيين قيم لخصائص الكائن، أو العمليات الأخرى الضرورية التي يجب القيام بها عند إنشاء الكائن.
مثال: أنشئ فئة باسم "Person" ، واستخدم الدالة `__init__()` لتعيين قيم للاسم والعمر:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

ملاحظة: يتم استدعاء الدالة `__init__()` تلقائيًا في كل مرة يتم فيها استخدام الصنف لإنشاء كائن جديد.

19-5- الدالة __str__():

- تتحكم الدالة __str__() فيما يجب إرجاعه عندما يتم تمثيل كائن الفئة كسلسلة.
 - إذا لم يتم تعيين الدالة __str__()، فسيتم إرجاع تمثيل السلسلة للكائن.
- مثال: تمثيل السلسلة لكائن بدون الدالة __str__():

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
p1 = Person("John", 36)
print(p1)
```

مثال: تمثيل السلسلة لكائن باستخدام الدالة __str__():

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return f"{self.name}({self.age})"
p1 = Person("John", 36)
print(p1)
```

19-6- طرق الكائن:

يمكن أن تحتوي الكائنات أيضاً على طرق. الأساليب في الكائنات هي وظائف تنتمي إلى الكائن. دعونا ننشئ طريقة في فئة الشخص.

مثال: أدخل دالة تطبع تحية وقم بتنفيذها على الكائن p1:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)
p1 = Person("John", 36)
p1.myfunc()
```

ملاحظة: تعد المعلمة self مرجعاً للكائن الحالي للفئة، ويتم استخدامها للوصول إلى المتغيرات التي تنتمي إلى الفئة.

19-7- المعلمة الذاتية The self Parameter:

تعد المعلمة self مرجعاً للمثيل الحالي للفئة، وتستخدم للوصول إلى المتغيرات التي تنتمي إلى الفئة. ليس من الضروري أن يتم تسميتها self، يمكنك تسميتها كما تريد، ولكن يجب أن تكون المعلمة الأولى لأي دالة في الصنف.

مثال: استخدم الكلمات mysillyobject و abc بدلاً من self:

```
class Person:
    def __init__(mysillyobject, name, age):
        mysillyobject.name = name
        mysillyobject.age = age
    def myfunc(abc):
        print("Hello my name is " + abc.name)
p1 = Person("John", 36)
p1.myfunc()
```

19-8- تعديل خصائص الكائن:

يمكنك تعديل الخصائص على كائنات مثل هذا.

مثال: أسند القيمة 40 على متحول العمر للكائن p1 :

```
p1.age = 40
```

19-9- حذف خصائص الكائن:

يمكنك حذف خصائص الكائنات باستخدام الكلمة الأساسية del.

مثال: احذف خاصية العمر من الكائن p1:

```
del p1.age
```

19-10- حذف الكائنات:

يمكنك حذف الكائنات باستخدام الكلمة الأساسية del.

مثال: حذف الكائن p1:

```
del p1
```

19-11- كلمة pass:

لا يمكن أن تكون تعريفات الفئة فارغة، ولكن إذا كان لديك تعريف فئة بدون محتوى لسبب ما.

مثال: قم بإدخال عبارة المرور لتجنب حدوث خطأ:

```
class Person:
    pass
```

19-12- مثال نموذجي عن استخدام الأصناف:

سنقوم بإنشاء فئات (classes) كائنا (objects) لتمثيل العناصر المختلفة في لعبة PUBG (PlayerUnknown's Battlegrounds). سننشئ فئات للاعبين (Player)، الأسلحة (Weapon)، المركبات (Vehicle)، والخريطة (Map).

1- تعريف فئة اللاعب (Player) :

```

class Player:
    def __init__(self, name, health=100):
        self.name = name
        self.health = health
        self.inventory = []
        self.position = (0, 0)
    def move(self, x, y):
        self.position = (x, y)
        print(f"{self.name} moved to position {self.position}")

    def take_damage(self, damage):
        self.health -= damage
        if self.health <= 0:
            print(f"{self.name} has been eliminated!")
        else:
            print(f"{self.name} took {damage} damage, {self.health} health remaining")
    def pick_up_weapon(self, weapon):
        self.inventory.append(weapon)
        print(f"{self.name} picked up {weapon.name}")

```

2- تعريف فئة الأسلحة (Weapon) :

```

class Weapon:
    def __init__(self, name, damage):
        self.name = name
        self.damage = damage
    def __str__(self):
        return self.name
# مثال عن استخدام
akm = Weapon("AKM", 49)
m416 = Weapon("M416", 43)

```

3- تعريف فئة المركبات (Vehicle) :

```

class Vehicle:
    def __init__(self, type, max_speed, health=100):
        self.type = type
        self.max_speed = max_speed
        self.health = health
    def drive(self, destination):
        print(f"Driving {self.type} to {destination} at max speed of {self.max_speed} km/h")
    def take_damage(self, damage):
        self.health -= damage
        if self.health <= 0:
            print(f"The {self.type} has been destroyed!")
        else:
            print(f"The {self.type} took {damage} damage, {self.health} health remaining")
# مثال عن استخدام
buggy = Vehicle("Buggy", 90)
buggy.drive("Pochinki")
buggy.take_damage(50)

```


4- تعريف فئة الخريطة (Map) :

```
class Map:
    def __init__(self, name, size):
        self.name = name
        self.size = size
        self.players = []
        self.vehicles = []
    def add_player(self, player):
        self.players.append(player)
        print(f"{player.name} joined the map {self.name}")
    def add_vehicle(self, vehicle):
        self.vehicles.append(vehicle)
        print(f"{vehicle.type} added to the map {self.name}")
# مثال عن استخدام Map
erangel = Map("Erangel", "8x8 km")
erangel.add_player(player1)
erangel.add_vehicle(buggy)
```

• استخدام هذه الفئات معًا في سياق اللعبة:

هذه الأمثلة تُظهر كيفية استخدام الفئات (classes) والأشياء (objects) لتمثيل العناصر المختلفة في لعبة PUBG. يمكن توسيع هذه الفئات بإضافة المزيد من الخصائص والأساليب لجعل النموذج أكثر تفصيلاً وتعقيداً بما يتناسب مع متطلبات اللعبة.

```
from classesPubg.player import Player
from classesPubg.weapon import Weapon
from classesPubg.vehicle import Vehicle
from classesPubg.map import Map
```

```
# إنشاء لاعبين
# مثال عن استخدام Player
player1 = Player("Player1")
player1.move(10, 20)
player1.take_damage(30)
player1.pick_up_weapon("AKM")
```

```
player2 = Player("Player2")
player3 = Player("Player3")
```

```
# إنشاء أسلحة
akm = Weapon("AKM", 49)
m416 = Weapon("M416", 43)
```

```
# إنشاء مركبات
buggy = Vehicle("Buggy", 90)
jeep = Vehicle("Jeep", 70)
```

```
# إنشاء خريطة
erangel = Map("Erangel", "8x8 km")
```

إضافة لاعبين إلى الخريطة

```
erangel.add_player(player1)  
erangel.add_player(player2)  
erangel.add_player(player3)
```

إضافة مركبات إلى الخريطة

```
erangel.add_vehicle(buggy)  
erangel.add_vehicle(jeep)
```

تحريك اللاعبين وأخذ أسلحة

```
player1.move(10, 20)  
player1.pick_up_weapon(akm)  
player2.move(15, 30)  
player2.pick_up_weapon(m416)
```

قيادة المركبات

```
buggy.drive("School")  
jeep.drive("Military Base")
```

إلحاق الأضرار

```
player1.take_damage(40)  
buggy.take_damage(70)
```

انتهى الفصل

الفصل العشرون

الوراثة في البايثون

20-1- مقدمة:

- يتيح لنا الوراثة تحديد فئة ترث جميع الأساليب والخصائص من فئة أخرى.
- الفئة الأصلية هي الفئة الموروثة منها، وتسمى أيضاً الفئة الأساسية (الأب).
- الفئة التابعة هي الفئة التي ترث من فئة أخرى، وتسمى أيضاً الفئة المشتقة (الابن).

20-2- إنشاء فئة الأب:

يمكن لأي فئة أن تكون فئة أصل، وبالتالي فإن بناء الجملة هو نفس إنشاء أي فئة أخرى.
مثال:

قم بإنشاء فئة باسم "شخص"، مع خصائص الاسم الأول واسم العائلة، وطريقة اسم الطباعة:

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

#Use the Person class to create an object, and then execute the printname method:

```
x = Person("John", "Doe")
x.printname()
```

20-3- إنشاء فئة الابن:

لإنشاء فئة ترث الوظيفة من فئة أخرى، أرسل الفئة الأصل كمعلمة عند إنشاء الفئة الفرعية.
مثال: قم بإنشاء فئة باسم Student، والتي سوف ترث الخصائص والأساليب من فئة الشخص:

```
class Student(Person):
    pass
```

الآن تتمتع فئة الطالب بنفس خصائص وأساليب فئة الشخص.

مثال: استخدم فئة الطالب لإنشاء كائن، ثم قم بتنفيذ طريقة اسم الطباعة:

```
x = Student("Mike", "Olsen")
x.printname()
```

20-3-1- أضف الدالة __init__ :

لقد قمنا حتى الآن بإنشاء فئة فرعية ترث الخصائص والأساليب من الفئة الأب. نريد إضافة الدالة __init__ إلى الفئة الفرعية (بدلاً من الكلمة الأساسية pass).

ملاحظة: يتم استدعاء الدالة __init__ تلقائياً في كل مرة يتم فيها استخدام الفصل لإنشاء كائن جديد.

مثال: أضف الدالة __init__ إلى فئة الطالب (الابن):

```
class Student(Person):
    def __init__(self, fname, lname):
        #add properties etc.
```

عند إضافة الدالة __init__، لن ترث الفئة الفرعية (الابن) الدالة __init__ الخاصة بالأب بعد ذلك.

ملاحظة: تتجاوز وظيفة __init__ الخاصة بالابن وراثته وظيفة __init__ الخاصة بالأب.

للمحافظة على وراثته الدالة __init__ الخاصة بالأب، قم بإضافة استدعاء إلى الدالة __init__ الخاصة بالأب.

مثال:

```
class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
```

لقد نجحنا الآن في إضافة الدالة __init__ واحتفظنا بوراثته الفئة الأصلية (الأب).

20-3-2- استخدم الدالة super() :

لدى بايثون أيضاً دالة super() التي ستجعل الفئة الفرعية ترث جميع التوابع والخصائص من الفئة الأب.

مثال:

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
```

باستخدام الدالة super()، لا يتعين عليك استخدام اسم العنصر الأصلي، حيث سيرث تلقائياً الأساليب والخصائص من العنصر الأصلي.

20-4- إضافة خصائص:

مثال: أضف خاصية تسمى Graduateyear إلى فئة الطالب:

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
        self.graduationyear = 2019
```

في المثال أدناه، يجب أن يكون عام 2019 متغيرًا، ويتم تمريره إلى فئة الطالب عند إنشاء كائنات الطالب. للقيام بذلك، قم بإضافة معلمة أخرى في الدالة `__init__()`.
مثال: أضف معلمة سنة، وقم بتمرير السنة الصحيحة عند إنشاء الكائنات:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
x = Student("Mike", "Olsen", 2019)
```

20-5- إضافة طرق:

مثال: أضف طريقة تسمى الترحيب في فصل الطالب:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

    def welcome(self):
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

إذا قمت بإضافة طريقة في الفئة الفرعية بنفس اسم وظيفة في الفئة الأصل، فسيتم تجاوز وراثة الطريقة الأصلية.

20-6- مثال نموذجي بسيط:

لنفترض أننا نريد إنشاء نموذج للحيوانات. سنبدأ بإنشاء فئة أساسية تسمى `Animal`، ثم سننشئ فئة فرعية تسمى `Dog` ترث من `Animal`.

الفئة الأساسية Animal

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")
    def move(self):
        print(f"{self.name} is moving")
```

الفئة الفرعية Dog

```
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed
    def speak(self):
        print(f"{self.name} says Woof!")
    def fetch(self):
        print(f"{self.name} is fetching the ball")
```

استخدام الوراثة

```
# إنشاء كائن من الفئة Dog
my_dog = Dog("Buddy", "Golden Retriever")
# استخدام الأساليب الموروثة والمخصصة
my_dog.move() # Buddy is moving
my_dog.speak() # Buddy says Woof!
my_dog.fetch() # Buddy is fetching the ball
```

شرح المثال

1. الفئة الأساسية Animal:

- تحتوي على مُنشئ (__init__) يقوم بتعيين اسم الحيوان.
- تحتوي على تابع speak كطريقة مجردة (abstract method) يجب على الفئات الفرعية أن تنفذها.

- تحتوي على تابع move يقوم بطباعة رسالة تشير إلى أن الحيوان يتحرك.

2. الفئة الفرعية Dog:

- ترث من الفئة Animal باستخدام class Dog(Animal).
- تحتوي على مُنشئ (__init__) يقوم بتعيين اسم وسلالة الكلب.
- يستخدم super().__init__(name) لاستدعاء مُنشئ الفئة الأساسية لتعيين الاسم.
- تنفذ تابع speak المطلوب وتخصصه لطباعة صوت الكلب.
- تحتوي على تابع جديد fetch الذي لا يوجد في الفئة الأساسية.

20-7- مثال نموذجي أكثر تعقيداً:

لنفترض أننا نريد إضافة المزيد من الأنواع من الحيوانات وإظهار كيفية استخدام الوراثة المتعددة المستويات.

الفئة الأساسية Animal

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")

    def move(self):
        print(f"{self.name} is moving")
```

الفئة الفرعية Mammal() ثدييات:

```
class Mammal(Animal):
    def __init__(self, name, fur_color):
        super().__init__(name)
        self.fur_color = fur_color
    def describe(self):
        print(f"{self.name} is a mammal with {self.fur_color} fur")
```

الفئة الفرعية Dog ترث من (Mammal):

```
class Dog(Mammal):
    def __init__(self, name, breed, fur_color):
        super().__init__(name, fur_color)
        self.breed = breed
    def speak(self):
        print(f"{self.name} says Woof!")
    def fetch(self):
        print(f"{self.name} is fetching the ball")
```

استخدام الوراثة المتعددة المستويات:

```
# إنشاء كائن من الفئة Dog
my_dog = Dog("Buddy", "Golden Retriever", "golden")
# استخدام الأساليب الموروثة والمخصصة
my_dog.move()      # Buddy is moving
my_dog.describe()  # Buddy is a mammal with golden fur
my_dog.speak()     # Buddy says Woof!
my_dog.fetch()     # Buddy is fetching the ball
```

شرح المثال المعقد:

1. الفئة الأساسية: Animal

- نفسها كما في المثال البسيط.

2. الفئة الفرعية: Mammal

- ترث من Animal.
- تحتوي على مُنشئ يقوم بتعيين الاسم ولون الفراء.
- تحتوي على تابع describe الذي يصف الثدييات.

3. الفئة الفرعية: Dog

- ترث من Mammal.
- تحتوي على مُنشئ يقوم بتعيين الاسم، السلالة، ولون الفراء.
- تستخدم super().__init__(name, fur_color) لاستدعاء مُنشئ الفئة الأساسية.
- تنفذ تابع speak وتخصصه لطباعة صوت الكلب.
- تحتوي على تابع fetch الذي لا يوجد في الفئة الأساسية أو الفرعية.

20-8- مزايا الوراثة:

- إعادة الاستخدام: يمكن إعادة استخدام الكود في الفئات الفرعية مما يقلل من التكرار.
- التوسعة: يمكن إضافة خصائص وأساليب جديدة في الفئات الفرعية دون تعديل الفئة الأساسية.
- التنظيم: يسهل تنظيم الكود وتصنيفه بناءً على العلاقات بين الكائنات.

20-9- نصائح حول الوراثة:

- استخدم الوراثة بحكمة. لا تستخدم الوراثة فقط لتقليل الكود، بل استخدمها عندما يكون هناك علاقة منطقية بين الفئات.
 - تجنب الوراثة العميقة جداً (الكثير من المستويات) لأنها قد تجعل الكود معقداً ويصعب صيانته.
 - استفد من تعدد الأشكال (polymorphism) لجعل الكود أكثر مرونة وقابلية للتوسع.
- باستخدام الوراثة بشكل صحيح، يمكن تحسين هيكلية البرامج وجعلها أكثر قابلية لإعادة الاستخدام والصيانة.

انتهى الفصل

الفصل الحادي والعشرين

تعدد الأشكال polymorphism

21-1- مقدمة:

كلمة "تعدد الأشكال" تعني "أشكال متعددة"، وفي البرمجة تشير إلى الأساليب/الوظائف/المعاملات الذين يحملون نفس الاسم والتي يمكن تنفيذها على العديد من الكائنات أو الفئات. مثال على دالة بايثون التي يمكن استخدامها على كائنات مختلفة هي الدالة len(). حيث بالنسبة للسلاسل النصية، تُرجع len() عدد الأحرف.

مثال:

```
x = "Hello World!"
print(len(x))
```

21-2- الصف tuple:

بالنسبة إلى tuples، تقوم len() بإرجاع عدد العناصر الموجودة في المجموعة.

مثال:

```
mytuple = ("apple", "banana", "cherry")
print(len(mytuple))
```

21-3- قاموس dictionary:

بالنسبة للقواميس، تقوم len() بإرجاع عدد أزواج المفاتيح/القيمة في القاموس.

مثال:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(len(thisdict))
```

21-4- تعدد الأشكال الطبقي:

يمكن أن يكون لدينا فئات متعددة بنفس اسم الطريقة. وعلى سبيل المثال، نفترض أن لدينا ثلاث فئات: السيارة، والقارب، والطائرة، ولديهم طريقة move().
مثال: فئات مختلفة بنفس الطريقة:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Drive!")

class Boat:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Sail!")

class Plane:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Fly!")

car1 = Car("Ford", "Mustang")    #Create a Car class
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class
plane1 = Plane("Boeing", "747")  #Create a Plane class

for x in (car1, boat1, plane1):
    x.move()
```

انظر إلى الحلقة في النهاية. بسبب تعدد الأشكال يمكننا تنفيذ نفس الطريقة لجميع الفئات الثلاثة.

21-5- تعدد الأشكال فئة الميراث:

إذا استخدمنا المثال أعلاه وقمنا بإنشاء فئة رئيسية تسمى مركبة، وقمنا بإنشاء فئات فرعية منها Car و Boat و Plane، فإن الفئات الفرعية ترث توابع المركبة، ولكن يمكنها تجاوزها.

مثال: قم بإنشاء فئة تسمى مركبة وقم بإنشاء فئات فرعية من المركبات مثل السيارة والقارب والطائرة:

```
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Move!")

class Car(Vehicle):
    pass

class Boat(Vehicle):
    def move(self):
        print("Sail!")

class Plane(Vehicle):
    def move(self):
        print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
    print(x.brand)
    print(x.model)
    x.move()
```

ترث الفئات الفرعية الخصائص والأساليب من الفئة الأصل.

نستنتج من المثال أعلاه:

- يمكنك أن ترى أن فئة السيارة فارغة، ولكنها **ترث** العلامة التجارية والطراز و move() من السيارة.
- ترث فئتا Boat و Plane أيضاً العلامة التجارية والطراز و move() من المركبة، لكن كلاهما يتجاوز طريقة move().
- بسبب **تعدد الأشكال** يمكننا تنفيذ نفس الطريقة لجميع الفئات.

انتهى الفصل

الفصل الثاني و العشرين

النطاق scope

22-1- مقدمة:

المتغير متاح فقط من داخل المنطقة التي تم إنشاؤه فيها. وهذا ما يسمى النطاق.

22-2- النطاق المحلي:

ينتمي المتغير الذي تم إنشاؤه داخل دالة إلى النطاق المحلي لتلك الوظيفة، ولا يمكن استخدامه إلا داخلها.
مثال: المتغير الذي تم إنشاؤه داخل دالة متاح داخل تلك الوظيفة:

```
def myfunc():
    x = 300
    print(x)
myfunc()
```

22-3- الوظيفة داخل الوظيفة:

كما هو موضح في المثال أعلاه، فإن المتغير x غير متوفر خارج الدالة، ولكنه متاح لأي دالة داخل الدالة.
مثال: يمكن الوصول إلى المتغير المحلي من دالة داخل الدالة:

```
def myfunc():
    x = 300
    def myinnerfunc():
        print(x)
    myinnerfunc()
myfunc()
```

22-4- النطاق العام:

المتغير الذي تم إنشاؤه في النص الرئيسي لكود Python هو متغير global وينتمي إلى النطاق العالمي global. فالمتغيرات العالمية متاحة من داخل أي نطاق، عالمي ومحلي.
مثال: المتغير الذي تم إنشاؤه خارج الوظيفة يكون عالمياً ويمكن لأي شخص استخدامه:

```
x = 300
def myfunc():
    print(x)
myfunc()
print(x)
```

22-5- تسمية المتغيرات:

إذا كنت تعمل بنفس اسم المتغير داخل الدالة وخارجها، فستعاملهما بايثون كمتغيرين منفصلين، أحدهما متاح في النطاق العام (خارج الدالة) والآخر متاح في النطاق المحلي (داخل الدالة).
مثال: ستقوم الدالة بطباعة x المحلي، ثم سيقوم الكود بطباعة x العالمي:

```
x = 300
def myfunc():
    x = 200
    print(x)
myfunc()
print(x)
```

22-6- الكلمة الرئيسية العالمية:

إذا كنت بحاجة إلى إنشاء متغير عام، ولكنك عالق في النطاق المحلي، فيمكنك استخدام الكلمة الأساسية العامة. الكلمة الأساسية العالمية `global` تجعل المتغير عالمياً.
مثال: إذا كنت تستخدم الكلمة الأساسية العمومية، فإن المتغير ينتمي إلى النطاق العام:

```
def myfunc():
    global x
    x = 300
myfunc()
print(x)
```

استخدم أيضاً الكلمة الأساسية العامة إذا كنت تريد إجراء تغيير على متغير عام داخل دالة.
مثال: لتغيير قيمة متغير عام داخل دالة، قم بالإشارة إلى المتغير باستخدام الكلمة الأساسية العامة:

```
x = 300
def myfunc():
    global x
    x = 200
myfunc()
print(x)
```

22-7- الكلمة الرئيسية غير المحلية:

- يتم استخدام الكلمة الأساسية غير المحلية للعمل مع المتغيرات داخل الوظائف المتداخلة.
- الكلمة الأساسية غير المحلية تجعل المتغير ينتمي إلى الوظيفة الخارجية.

مثال: إذا كنت تستخدم الكلمة الأساسية **غير المحلية**، فسوف ينتمي المتغير إلى الوظيفة الخارجية:

```
def myfunc1():  
    x = "Jane"  
  
def myfunc2():  
    nonlocal x  
    x = "hello"  
    myfunc2()  
    return x  
print(myfunc1())
```

انتهى الفصل

الفصل الثالث و العشرين

الوحدات modules

23-1- مقدمة:

ما هي الوحدة؟ هي نفس مكتبة التعليمات البرمجية. وهي ملف يحتوي على مجموعة من الوظائف التي تريد تضمينها في التطبيق الخاص بك.

23-2- إنشاء وحدة:

لإنشاء وحدة نمطية، ما عليك سوى حفظ الكود الذي تريده في ملف بامتداد الملف `.py`.
مثال: احفظ هذا الكود في ملف اسمه `mymodule.py`

```
def greeting(name):
    print("Hello, " + name)
```

23-3- استخدم الوحدة النمطية:

يمكننا الآن استخدام الوحدة التي أنشأناها للتو، باستخدام عبارة الاستيراد.
مثال: قم باستيراد الوحدة المسماة `mymodule`، واستدعاء وظيفة الترحيب:

```
import mymodule
mymodule.greeting("Jonathan")
```

● ملاحظة: عند استخدام دالة من وحدة نمطية، استخدم الصيغة:

Module name.function name.

23-4- المتغيرات في الوحدة النمطية:

يمكن أن تحتوي الوحدة على وظائف، كما هو موضح بالفعل، ولكن أيضًا متغيرات من جميع الأنواع (المصفوفات والقواميس والكائنات وما إلى ذلك).

مثال: احفظ هذا الكود في الملف `mymodule.py`

```
person1 = { "name": "John", "age": 36, "country": "Norway" }
```

مثال: قم باستيراد الوحدة المسماة `mymodule`، وقم بالوصول إلى قاموس `person1`:

```
import mymodule
a = mymodule.person1["age"]
print(a)
```

23-5- تسمية الوحدة النمطية:

- يمكنك تسمية ملف الوحدة النمطية كما تريد، ولكن يجب أن يكون له امتداد الملف `.py`.
 - يمكنك إعادة تسمية الوحدة النمطية.
 - يمكنك إنشاء اسم مستعار عند استيراد وحدة نمطية، باستخدام الكلمة الأساسية `as`.
- مثال: قم بإنشاء اسم مستعار لـ `mymodule` يسمى `mx`:

```
import mymodule as mx
a = mx.person1["age"]
print(a)
```

23-6- وحدات مدمجة:

هناك العديد من الوحدات المضمنة في لغة بايثون، والتي يمكنك استيرادها وقتما تشاء.

مثال: استيراد واستخدام وحدة النظام الأساسي:

```
import platform
x = platform.system()
print(x)
```

باستخدام الدالة `dir()`:

توجد وظيفة مضمنة لسرد كافة أسماء الوظائف (أو أسماء المتغيرات) في الوحدة النمطية. الدالة `dir()`.

مثال: قم بإدراج كافة الأسماء المحددة التي تنتمي إلى وحدة النظام الأساسي:

```
x = dir(platform)
print(x)
```

ملاحظة: يمكن استخدام الدالة `dir()` في كافة الوحدات، بما في ذلك تلك التي تقوم بإنشائها بنفسك.

23-7- استيراد من الوحدة النمطية:

يمكنك اختيار استيراد أجزاء فقط من الوحدة النمطية، باستخدام الكلمة الأساسية `from`.

مثال: تحتوي الوحدة المسماة `mymodule` على وظيفة واحدة وقاموس واحد:

```
def greeting(name):
    print("Hello, " + name)
person1 = { "name": "John", "age": 36, "country": "Norway" }
```

مثال: قم باستيراد قاموس `person1` فقط من الوحدة النمطية:

```
from mymodule import person1
print (person1["age"])
```

ملاحظة: عند الاستيراد باستخدام الكلمة الأساسية `from`، لا تستخدم اسم الوحدة عند الإشارة إلى العناصر الموجودة في الوحدة.

مثال:

`mymodule.person1["age"]` وليس `person1["age"]`

انتهى الفصل

الفصل الرابع و العشرين

الرياضيات math

24-1- مقدمة:

تحتوي لغة Python على مجموعة من الوظائف الرياضية المضمنة، بما في ذلك وحدة رياضية واسعة النطاق، والتي تتيح لك أداء المهام الرياضية على الأرقام.

24-2- وظائف الرياضيات المضمنة:

يمكن استخدام الدالتين `min()` و `max()` للعثور على أدنى أو أعلى قيمة في كائن قابل للتكرار.

مثال:

```
x = min(5, 10, 25)
y = max(5, 10, 25)
print(x, y)
```

تقوم الدالة `abs()` بإرجاع القيمة المطلقة (الإيجابية) للرقم المحدد.

مثال:

```
x = abs(-7.25)
print(x)
```

• تقوم الدالة `pow(x, y)` بإرجاع قيمة `x` إلى قوة `(x^y)`.

مثال: أعد قيمة 4 للقوة 3 (مثل $4 * 4 * 4$):

```
x = pow(4, 3)
print(x)
```

24-3- وحدة الرياضيات:

تحتوي بايثون أيضاً على وحدة مدمجة تسمى الرياضيات، والتي تعمل على توسيع قائمة الوظائف الرياضية. لاستخدامها، يجب عليك استيراد وحدة الرياضيات:

```
import math
```

عندما تقوم باستيراد وحدة الرياضيات، يمكنك البدء في استخدام أساليب وثوابت الوحدة.

على سبيل المثال، تقوم طريقة `math.sqrt()` بإرجاع الجذر التربيعي لرقم:

```
import math
x = math.sqrt(64)
print(x)
```

يقوم الأسلوب `math.ceil()` بتقريب الرقم لأعلى إلى أقرب عدد صحيح، ويقوم الأسلوب `math.floor()` بتقريب الرقم للأسفل إلى أقرب عدد صحيح له، ويعيد النتيجة.

مثال:

```
import math
x = math.ceil(1.4)
y = math.floor(1.4)
print(x) # returns 2
print(y) # returns 1
```

يُرجع ثابت `math.pi` قيمة π (3.14...).

مثال:

```
import math
x = math.pi
print(x)
```

انتهى الفصل

الفصل الخامس و العشرين

المحاولة والاستثناء Try Except

25-1- مقدمة:

تتيح لك كتلة المحاولة (try) اختبار كتلة من التعليمات البرمجية بحثاً عن الأخطاء.
تتيح لك كتلة الاستثناء (except) معالجة الخطأ.
تتيح لك الكتلة (else) تنفيذ التعليمات البرمجية في حالة عدم وجود خطأ.
تتيح لك الكتلة (finally) تنفيذ التعليمات البرمجية، بغض النظر عن نتيجة كتلة المحاولة والاستثناء.

25-2- معالجة الاستثناء:

- عند حدوث خطأ، أو استثناء كما نسميه، ستتوقف لغة بايثون عادةً وتولد رسالة خطأ.
- يمكن معالجة هذه الاستثناءات باستخدام عبارة المحاولة try.

مثال: ستولد كتلة المحاولة استثناءً، لأن x لم يتم تعريفه:

```
try:
    print(x)
except:
    print("An exception occurred")
```

نظرًا لأن كتلة المحاولة تثير خطأً، فسيتم تنفيذ كتلة الاستثناء. بدون كتلة المحاولة، سوف يتعطل البرنامج ويظهر خطأ.

مثال: سيؤدي هذا البيان إلى ظهور خطأ، لأنه لم يتم تعريف x:

```
print(x)
```

25-3- العديد من الاستثناءات:

يمكنك تحديد أي عدد تريده من كتل الاستثناء، على سبيل المثال. إذا كنت تريد تنفيذ كتلة خاصة من التعليمات البرمجية لنوع خاص من الأخطاء.

مثال: اطبع رسالة واحدة إذا أدت كتلة المحاولة إلى ظهور خطأ `NameError` وأخرى للأخطاء الأخرى:

```
try:
    print(x)
except NameError:
    print("Variable x is not defined")
except:
    print("Something else went wrong")
```

25-4- else مع الاستثناءات:

يمكنك استخدام الكلمة الأساسية `else` لتحديد كتلة من التعليمات البرمجية التي سيتم تنفيذها في حالة عدم ظهور أي أخطاء.

مثال: في هذا المثال، لا تولد كتلة المحاولة أي خطأ:

```
try:
    print("Hello")
except:
    print("Something went wrong")
else:
    print("Nothing went wrong")
```

25-5- finally مع الاستثناءات:

سيتم تنفيذ الكتلة الأخيرة، إذا تم تحديدها، بغض النظر عما إذا كانت كتلة المحاولة تثير خطأ أم لا.

مثال:

```
try:
    print(x)
except:
    print("Something went wrong")
finally:
    print("The 'try except' is finished")
```

• قد يكون هذا مفيداً لإغلاق الكائنات وتنظيف الموارد.

مثال: حاول فتح ملف غير قابل للكتابة والكتابة فيه (سنتوسع في التفاصيل في الفصل القادم):

```
try:
    f = open("demofile.txt")
    try:
        f.write("Lorum Ipsum")
    except:
        print("Something went wrong when writing to the file")
    finally:
        f.close()
except:
    print("Something went wrong when opening the file")
```

• يمكن أن يستمر البرنامج دون ترك كائن الملف مفتوحاً.

25-6- رفع استثناء:

- باعتبارك مطور لغة Python، يمكنك اختيار طرح استثناء في حالة حدوث شرط ما.
 - لرمي (أو رفع) استثناء، استخدم الكلمة الأساسية الرفع `raise`.
- مثال: يظهر خطأ ويوقف البرنامج إذا كانت x أقل من 0:

```
x = -1
if x < 0:
    raise Exception("Sorry, no numbers below zero")
```

- يمكنك تحديد نوع الخطأ الذي يجب رفعه، والنص الذي سيتم طباعته للمستخدم.

مثال: قم برفع الخطأ `TypeError` إذا لم يكن x عددًا صحيحًا:

```
x = "hello"
if not type(x) is int:
    raise TypeError("Only integers are allowed")
```

انتهى الفصل

الفصل السادس و العشرين

التعامل مع الملفات File Handling

26-1- مقدمة:

- تعد معالجة الملفات جزءاً مهماً من أي تطبيق ويب.
- لدى Python العديد من الوظائف لإنشاء الملفات وقراءتها وتحديثها وحذفها.

26-2- التعامل مع الملف:

- الوظيفة الأساسية للعمل مع الملفات في بايثون هي الدالة open().
- تأخذ الدالة open() معلمتين؛ اسم الملف، والوضع.

هناك أربع طرق (أوضاع) مختلفة لفتح ملف:

- "r" - القراءة - القيمة الافتراضية. يفتح ملفاً للقراءة، خطأ إذا كان الملف غير موجود.
 - "a" - إلحاق - يفتح ملفاً للإلحاق، وينشئ الملف إذا لم يكن موجوداً.
 - "w" - كتابة - يفتح ملفاً للكتابة، وينشئ الملف إذا لم يكن موجوداً.
 - "x" - إنشاء - إنشاء الملف المحدد، وإرجاع خطأ في حالة وجود الملف.
- بالإضافة إلى ذلك، يمكنك تحديد ما إذا كان يجب التعامل مع الملف كوضع ثنائي أو نصي:

- "t" - نص - القيمة الافتراضية. وضع النص.
- "b" - ثنائي - الوضع الثنائي (مثل الصور).

26-3- بناء الجملة:

لفتح ملف للقراءة يكفي تحديد اسم الملف:

```
f = open("demofile.txt")
```

الكود أعلاه هو نفسه:

```
f = open("demofile.txt", "rt")
```

نظرًا لأن "r" للقراءة و "t" للنص هما القيمتان الافتراضيتان، فلن تحتاج إلى تحديدهما.

ملاحظة: تأكد من وجود الملف، وإلا فسوف تحصل على خطأ.

تقوم الدالة open() بإرجاع كائن ملف، والذي يحتوي على طريقة read() لقراءة محتوى الملف.
مثال:

```
f = open("demofile.txt", "r")
print(f.read())
```

إذا كان الملف موجودًا في موقع مختلف، فسيتمكن عليك تحديد مسار الملف.

مثال: افتح ملفًا في موقع مختلف:

```
f = open("D:\\myfiles\\welcome.txt", "r")
print(f.read())
```

26-4- قراءة أجزاء فقط من الملف:

افتراضيًا، تقوم الدالة read() بإرجاع النص بأكمله، ولكن يمكنك أيضًا تحديد عدد الأحرف التي تريد إرجاعها.
مثال: قم بإرجاع الأحرف الخمسة الأولى من الملف:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

• قراءة السطور:

يمكنك إرجاع سطر واحد باستخدام التابع readline().

مثال: اقرأ سطرًا واحدًا من الملف:

```
f = open("demofile.txt", "r")
print(f.readline())
```

من خلال استدعاء readline() مرتين، يمكنك قراءة السطرين الأولين.

مثال: اقرأ سطرين من الملف:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

من خلال تكرار سطور الملف، يمكنك قراءة الملف بأكمله سطرًا تلو الآخر.

مثال: قم بالتكرار عبر سطر الملف سطرًا:

```
f = open("demofile.txt", "r")
for x in f:
    print(x)
```

26-5- إغلاق الملفات:

من الممارسات الجيدة إغلاق الملف دائمًا عند الانتهاء منه.

مثال: أغلق الملف عند الانتهاء منه:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

ملاحظة: يجب عليك دائماً إغلاق ملفاتك، في بعض الحالات، بسبب التخزين المؤقت، قد لا تظهر التغييرات التي تم إجراؤها على الملف حتى تقوم بإغلاق الملف.

26-6- الكتابة إلى ملف موجود:

للكتابة إلى ملف موجود، يجب عليك إضافة معلمة إلى الدالة open():

"a" - إلحاق - سيتم إلحاقه بنهاية الملف.

"w" - الكتابة - سوف تحل محل أي محتوى موجود.

مثال: افتح الملف "demofile2.txt" وألحق المحتوى بالملف:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()
#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

مثال: افتح الملف "demofile3.txt" واستبدل المحتوى:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()
#open and read the file after the overwriting:
f = open("demofile3.txt", "r")
print(f.read())
```

ملحوظة: الطريقة "w" ستحل محل الملف بأكمله.

26-7- إنشاء ملف جديد:

لإنشاء ملف جديد في بايثون، استخدم طريقة open()، مع أحد المعلمات التالية:

"x" - إنشاء - سيتم إنشاء ملف، وإرجاع خطأ في حالة وجود الملف.

"a" - إلحاق - سيقوم بإنشاء ملف إذا كان الملف المحدد غير موجود.

"w" - كتابة - سيقوم بإنشاء ملف إذا كان الملف المحدد غير موجود.

مثال: قم بإنشاء ملف يسمى "myfile.txt":

```
f = open("myfile.txt", "x")
```

النتيجة: يتم إنشاء ملف فارغ جديد!

مثال: أنشئ ملفًا جديدًا إذا لم يكن موجودًا:

```
f = open("myfile.txt", "w")
```

26-8- حذف ملف:

لحذف ملف، يجب عليك استيراد وحدة نظام التشغيل وتشغيل وظيفة `os.remove()` الخاصة بها.

مثال: إزالة الملف "demofile.txt":

```
import os
os.remove("demofile.txt")
```

26-9- التحقق من وجود الملف:

لتجنب الحصول على خطأ، قد ترغب في التحقق من وجود الملف قبل محاولة حذفه.

مثال: تحقق من وجود الملف ثم احذفه:

```
import os
if os.path.exists("demofile.txt"):
    os.remove("demofile.txt")
else:
    print("The file does not exist")
```

26-10- حذف المجلد:

لحذف مجلد بأكمله، استخدم الطريقة `os.rmdir()`.

مثال: إزالة المجلد "myfolder":

```
import os
os.rmdir("myfolder")
```

ملاحظة: يمكنك فقط إزالة المجلدات الفارغة.

انتهى الفصل

الفصل السابع و العشرين

التعامل مع JSON في البايثون

27-1- مقدمة:

الـ (JSON (JavaScript Object Notation هو تنسيق خفيف لتبادل البيانات. يُستخدم بشكل واسع في تطبيقات الويب لتمثيل البيانات بشكل قابل للقراءة البشرية وسهل التحليل بواسطة الآلات. يتميز JSON بتركيبه البسيط والمعتمد على زوج المفتاح والقيمة.

JSON في Python : يحتوي Python على حزمة مدمجة تسمى json، والتي يمكن استخدامها للعمل مع بيانات JSON.

27-2- استخدامات JSON :

- نقل البيانات: يمكن استخدام JSON لنقل بيانات معينة بين الخادم (server) والعميل (client) .
- تخزين البيانات: يمكن تخزين بيانات معينة في قاعدة بيانات وتحديثها أو جلبها عند الطلب.
- عرض البيانات: يمكن استخدام البيانات الممثلة بـ JSON لعرض تفاصيل البيانات على واجهة المستخدم.

27-3- فوائد JSON:

- سهولة القراءة والكتابة JSON: سهل الفهم والكتابة سواء للبشر أو الآلات.
- قابلية النقل: يمكن نقل بيانات JSON بسهولة بين الأنظمة المختلفة.
- مدعوم بشكل واسع JSON: مدعوم في معظم لغات البرمجة ويستخدم بشكل واسع في تطوير التطبيقات الحديثة.

4-27- تحليل JSON:**1-4-27 التحويل من JSON إلى Python:**

إذا كان لديك سلسلة JSON، فيمكنك تحليلها باستخدام طريقة json.loads(). ستكون النتيجة عبارة عن قاموس Python.

مثال: التحويل من JSON إلى Python:

```
import json
# some JSON:
x = '{ "name":"John", "age":30, "city":"New York"}'
# parse x:
y = json.loads(x)
# the result is a Python dictionary:
print(y["age"])
```

2-4-27 التحويل من Python إلى JSON :

إذا كان لديك كائن Python، فيمكنك تحويله إلى سلسلة JSON باستخدام طريقة json.dumps().

مثال: التحويل من Python إلى JSON:

```
import json
# a Python object (dict):
x = {
    "name": "John",
    "age": 30,
    "city": "New York"
}
# convert into JSON:
y = json.dumps(x)
# the result is a JSON string:
print(y)
```

• يمكنك تحويل كائنات Python من الأنواع التالية إلى سلاسل JSON :

Dict – list – tuple – string – int – float – True – False - None

مثال: تحويل كائنات Python إلى سلاسل JSON وطباعة القيم:

```
import json
print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

عند التحويل من Python إلى JSON ، يتم تحويل كائنات Python إلى ما يعادلها من JSON (JavaScript) :

البايثون	JSON
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

مثال: تحويل كائن بايثون يحتوي على جميع أنواع البيانات القانونية:

```
import json
x = {
    "name": "John",
    "age": 30,
    "married": True,
    "divorced": False,
    "children": ("Ann", "Billy"),
    "pets": None,
    "cars": [
        {"model": "BMW 230", "mpg": 27.5},
        {"model": "Ford Edge", "mpg": 24.1}
    ]
}
print(json.dumps(x))
```

27-5- تنسيق النتيجة:

يطبع المثال أعلاه سلسلة JSON ، ولكن ليس من السهل قراءتها، بدون مسافات بادئة وفواصل أسطر.

تحتوي طريقة json.dumps() على معلمات (بارامترات) لتسهيل قراءة النتيجة.

مثال: استخدم معلمة المسافة البادئة لتحديد عدد المسافات البادئة:

```
json.dumps(x, indent=4)
```

الخرج:

```
{
    "name": "John",
    "age": 30,
    "married": true,
    "divorced": false,
    "children": [
        "Ann",
        "Billy"
    ],
    "pets": null,
    "cars": [
        {
            "model": "BMW 230",
            "mpg": 27.5
        },
        {
            "model": "Ford Edge",
            "mpg": 24.1
        }
    ]
}
```

يمكنك أيضاً تحديد الفواصل، والقيمة الافتراضية هي (", ", ": "), مما يعني استخدام فاصلة ومسافة لفصل كل كائن، وعلامة نقطتين ومسافة لفصل المفاتيح عن القيم.
مثال: استخدم معلمة الفواصل لتغيير الفاصل الافتراضي:

```
json.dumps(x, indent=4, separators=(".", " = "))
```

الخرج:

```
{
  "name" = "John".
  "age" = 30.
  "married" = true.
  "divorced" = false.
  "children" = [
    "Ann".
    "Billy"
  ].
  "pets" = null.
  "cars" = [
    {
      "model" = "BMW 230".
      "mpg" = 27.5
    }.
    {
      "model" = "Ford Edge".
      "mpg" = 24.1
    }
  ]
}
```

27-6- ترتيب النتيجة:

تحتوي طريقة json.dumps() على معلمات لترتيب المفاتيح في النتيجة.

مثال: استخدم معلمة sort_keys لتحديد ما إذا كان يجب فرز النتيجة أم لا:

```
json.dumps(x, indent=4, sort_keys=True)
```

الخرج:

```
{
  "age": 30,
  "cars": [
    {
      "model": "BMW 230",
      "mpg": 27.5
    },
    {
      "model": "Ford Edge",
      "mpg": 24.1
    }
  ],
  "children": [
    "Ann",
    "Billy"
  ],
  "divorced": false,
  "married": true,
  "name": "John",
  "pets": null
}
```

انتهى الفصل

انتهى الكتاب

تم بعونه تعالى بتاريخ 2025/4/1