

لغة البايثون لتحليل البيانات

ما هي لغة بايثون؟

بايثون هي لغة برمجة عالية المستوى موجهة للકائنات وتستخدم لمجموعة واسعة من المشكلات ذات النطاق والتعقيد. على عكس العديد من اللغات المماثلة، من السهل إتقانها وتعلمها وهي مثالية للمبتدئين. لكن هذه السهولة ليست السبب الوحيد لأهميتها، هي قوية بما يكفي حتى للمستخدمين المتقدمين. بالإضافة إلى ذلك، تعد بايثون أكثر أدوات علوم البيانات استخداماً، وهي مدرجة كشرط في معظم قوائم إعلانات وظائف علوم البيانات.

ما هو سبب اختيار بايثون في علم البيانات؟

- البساطة: بايثون هي واحدة من أسهل اللغات للبدء بها. أيضا، هذه البساطة لا تحد من الميزات التي تحتاجها.
- المكتبات والأطر: نظراً لشعبيتها، تمتلك بايثون المئات من المكتبات والأطر المختلفة التي تساعده بشكل كبير في عملية التطوير الخاصة بك وتتوفر الكثير من الوقت. بصفتك عالم بيانات، ستجد أن العديد من هذه المكتبات تركز على علم البيانات والتعلم الآلي.
- مجتمع هائل: أحد أسباب شهرة بايثون هو أنها تضم مجتمعاً كبيراً من المهندسين والعلماء.
- أهميةها في التعلم العميق: تحتوي بايثون على العديد من الحزم مثل keras و Tensorflow و PyTorch التي تساعده علماء البيانات على تطوير خوارزميات التعلم العميق.
- تمثيل مرئي أفضل للبيانات: التمثيل المرئي للبيانات هو مفتاح لعلماء البيانات لأنه يساعدهم على فهم البيانات بشكل أفضل.

المكتبات وإدارتها في بايثون

المكتبات في بايثون عبارة عن أجزاء من التعليمات البرمجية يمكن إعادة استخدامها بما في ذلك مجموعة من الدوال والطرق تسمح للمستخدم بأداء العديد من المهام دون أن يضطر المستخدم إلى كتابتها.

➤ NumPy

NumPy هي مكتبة أساسية للحسابات الرياضية والعلمية. تدعم المكتبة المصفوفات والمصفوفات الكبيرة متعددة الأبعاد، بالإضافة إلى مجموعة كبيرة من الدوال الرياضية عالية المستوى للعمل على هذه المصفوفات.

➤ Keras

Keras هي مكتبة بايثون مفتوحة المصدر تُستخدم على نطاق واسع لتعليم نماذج التعلم العميق.

TensorFlow ➤

هي واحدة من مكتبات بايثون الأكثر استخداماً لمعالجة البيانات والنموذج ، بالإضافة إلى أنها مكتبة مهمة للتعلم الآلي في بايثون.

PyTorch ➤

هو إطار عمل تعلم آلي وتعلم عميق مفتوح المصدر طوره باحثو Facebook AI. يستخدم العديد من علماء البيانات حول العالم PyTorch على نطاق واسع لمعالجة مسائل اللغة الطبيعية ورؤية الحاسوب.

Scrapy ➤

تعد Scrapy واحدة من أكثر مكتبات بايثون شيوعاً لاستخراج البيانات من مواقع الويب. تساعد هذه المكتبة في استرداد البيانات من مواقع الويب بطريقة فعالة.

► Pandas

تعد Pandas واحدة من أهم مكتبات علم البيانات المستخدمة لإنشاء هيكل البيانات. يوفر Pandas مرونة قوية في إنشاء هيكل البيانات لعلم البيانات.

► Scikit-Learn

يستخدمها علماء البيانات لنمذجة البيانات إحصائياً بما في ذلك التصنيف، وتقليل الأبعاد والتجميع والتوقع.

► Matplotlib

يعد رسم المخططات أحد الخطوات الأساسية أثناء تحليل البيانات وإدارتها. تعد Matplotlib واحدة من أكثر المكتبات شيوعاً في مجتمع بايثون للرسم والتمثيل المرئي للبيانات الثابتة والمتحركة والتفاعلية.

تثبيت وحذف وتحديث المكتبات

يجب عليك استخدام Pip لإدارة المكتبات.

Pip هي أداة أساسية تتيح لك تنزيل الحزم التي تحتاجها وتحديثها وحذفها. بالإضافة إلى ذلك، يمكن استخدامه للتحقق من التبعيات المناسبة والتوافق بين الإصدارات.

> **pip install NumPy**

• ثم أدخل الأمر التالي في سطر الأوامر:

• للتأكد من تثبيت المكتبة، قم بتشغيل سطر أوامر بایثون واكتب الأمر التالي:

>>> **import NumPy**

• يتم استخدام الأمر التالي لحذف مكتبة على سبيل المثال، NumPy :

> **pip uninstall NumPy**

• في بعض الأحيان، تجد نفسك في موقف يتquin عليك فيه ترقية مكتبة. نظراً لأن تثبيت مكتبة أخرى يتطلب إصداراً أحدث من المكتبة مثبتاً على جهاز الكمبيوتر الخاص بك، فقد ترغب في الاستفادة من الإصدار المحدث، الذي يحتوي على ميزات إضافية.

> **pip install --upgrade numpy**

البرمجة بلغة بايثون

تركيب الجملة في بايثون

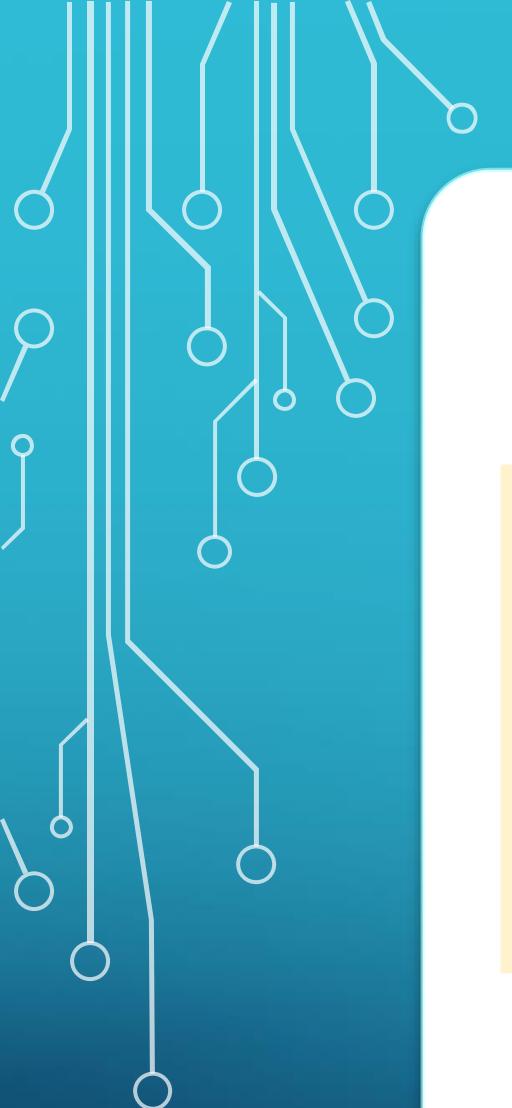
يشير تركيب الجملة في لغة البرمجة إلى بنية اللغة. بمعنى آخر، التركيب اللغوي هو مجموعة من القواعد التي تحدد كيفية كتابة البرمجة بلغة ما.

الكلمات المحوزة

✓ الكلمات المفتاحية هي بعض الكلمات المحوزة في بايثون والتي لها معانٍ خاصة. تستخدم الكلمات المفتاحية لتعريف القواعد النحوية وهيكل اللغة.

- ✓ لا يمكن استخدام الكلمة المفتاحية كمعرف أو وظيفة أو اسم متغير.
- ✓ الكلمات المفتاحية في بايثون حساسة لحالة الأحرف، لذا يجب كتابتها كما هي.
- ✓ جميع الكلمات المفتاحية في بايثون، باستثناء `True` و`False` و`None`، هي أحرف صغيرة.

يمكنك دائمًا الحصول على قائمة بالكلمات المفتاحية في إصدارك الحالي



```
In [1]: import keyword  
keyword.kwlist  
  
Out [1]: ['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await',  
'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except',  
'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',  
'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while',  
'with', 'yield']
```

المعرف

المعرف هو الاسم الذي نستخدمه لتحديد متغير، أو فئة، أو دالة، أو وحدة نمطية، أو كائن. هذا يساعد على تمييز كيان واحد عن الآخر.

قواعد كتابة المعرف

هناك بعض القواعد لكتابة المعرفات. نحتاج إلى معرفة أن لغة بايثون حساسة لحالة الأحرف. هذا يعني أن `name` و `Name` هما معرفان مختلفان في بايثون. المدرجة أدناه هي بعض القواعد لكتابة المعرفات في بايثون:

١. يمكن أن تكون المعرفات مزيجاً من الأحرف الصغيرة من a إلى z أو الأحرف الكبيرة (من A إلى Z) أو الأرقام (من 0 إلى 9) وأو الشرطة السفلية (_).
٢. لا يمكن أن يبدأ المعرف برقم.
٣. لا يمكن استخدام رموز معينة مثل !، @، # ، \$ ، % كمعرفات .
٤. يمكن أن يكون المعرف بأي طول.

المتغير

المتغير هو موقع يستخدم لتخزين البيانات في الذاكرة. هذا يعني أنه عند إنشاء متغير، فإنك تشغّل بعض المساحة في الذاكرة. يتم تعين اسم لكل متغير بحيث يمكن تحديده والوصول إليه من متغيرات أخرى.

اسناد قيم للمتغيرات

يعد إنشاء المتغيرات في بايثون أمراً بسيطاً، ما عليك سوى كتابة اسم المتغير على اليسار = وقيمة المتغير على اليمين:

```
In [1]: num = 5  
In [2]: str = "Python"
```

لا تحتاج إلى تحديد نوع المتغير، تستنتج بايثون النوع بناءً على القيمة التي نخصصها.

الاسناد المتعدد

تسمح لنا ببايثون بتعيين قيمة لمتغيرات متعددة في الذاكرة، يسمى الاسناد المتعدد. يمكننا تطبيق اسناد متعدد بطريقتين، إما عن طريق تعين قيمة واحدة لمتغيرات متعددة أو عن طريق تعين قيم متعددة لمتغيرات متعددة. تأمل الأمثلة التالية:

```
In [3]: a=b=c=20  
      print("a:",a)  
      print("b:",b)  
      print("c:",c)  
  
Out [3]: a: 20  
          b: 20  
          c: 20  
  
In [4]: a,b,c=1,2.54,"python"
```

أنواع البيانات

في البرمجة، يعد نوع البيانات مفهوماً مهماً. يمكن للمتغيرات تخزين البيانات من أنواع مختلفة.

الاعداد

- ✓ في بايثون، يشير نوع البيانات الرقمية إلى البيانات التي تحتوي على قيمة رقمية.
- ✓ يمكن أن تكون القيم الرقمية أعداداً صحيحة أو أرقاماً فاصلة عائمة أو حتى أرقاماً معقدة.
- ✓ يتم تعريف هذه القيم على أنها فئات `int` و `float` و `complex` في بايثون.
- ✓ يتم استخدام وظيفة `type` لتحديد نوع البيانات.

```
In [1]: a = 6 b = 7.0 c = 4 + 3j  
print("Type of a: ", type(a))  
print("Type of b: ", type(b))  
print("Type of c: ", type(c))  
  
Out [1]: Type of a: <class 'int'>  
Type of b: <class 'float'>  
Type of c: <class 'complex'>
```

السلسل النصية

- ✓ في بايثون، السلسل عبارة عن مصفوفات من البايتات التي تمثل أحرف Unicode.
- ✓ السلسلة النصية عبارة عن مجموعة من حرف واحد أو أكثر.
- ✓ لا توجد بيانات حرفية في بايثون، فالحرف عبارة عن سلسلة بطول واحد.

```
In [2]: Str_1 = 'Python Data Types'
```

```
Str_1
```

```
Out [2]: 'Python Data Types'
```

```
In [3]: Str_2 = "data science"
```

```
Str_2
```

```
Out [3]: "data science"
```

```
In [3]: type(Str_2)
```

```
Out [3]: str
```

```
In [3]: type("c")
```

```
Out [3]: str
```

العوامل

تُستخدم عوامل بايثون عموماً لإجراء عمليات على القيم والمتغيرات. إنها رموز قياسية تُستخدم لأداء العمليات المنطقية والحسابية والمقارنة.

العوامل الحسابية

تُستخدم العوامل الحسابية لإجراء عمليات حسابية مثل الجمع، والطرح، والضرب، والقسمة.

العامل	شرح العامل	العملية
+	الجمع	$x + y$
-	الطرح	$x - y$
*	الضرب	$x * y$
/	القسمة (قسمة x على y)	x / y
//	القسمة (حاصل القسمة التحتية x على y)	$x // y$
%	باقي القسمة	$x \% y$
**	القوة	$x ** y$

In [1]:

```
x=10
y=3
add = x+y
sub = x - y
mul = x * y
div1 = x / y
div2 = x // y
mod = x % y
p = x ** y
```

```
print("Addition:",add)
print("Subtraction:",sub)
print("Multiplication:",mul)
print("Division(float):",div1)
print("Division(floor):",div2)
print("Modulo:",mod)
print("Power:",p)
```

Out [1]:

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division(float): 3.333333333333335
Division(floor): 3
Modulo: 1
Power: 1000
```

العوامل المنطقية

يتم تطبيق العوامل المنطقية **or**، **and** و **not** على التعبيرات المنطقية وتكون النتيجة صواب أو خطأ.
يتم استخدام هذه العوامل للشروط المعقدة.

العامل	شرح العامل	العملية
and	إذا كان كلا المعاملين True	$x \text{ and } y$
or	إذا كان أحد المعاملات True	$x \text{ or } y$
not	إذا كان True فالمعامل False	$\text{not } x$

```
In [1]: X = True  
Y = False  
print(X and Y)  
print(X or Y)  
print(not X)  
  
Out [1]: False  
True  
False
```

عوامل المقارنة

تقارن هذان العاملان قيم جانبيهما ويحددان العلاقة بينهما. تسمى عوامل المقارنة أيضًا بالعوامل الارتباطية.

In [1]: $a = 11$ $b = 26$

```
print(a > b)
```

```
print(a < b)
```

```
print(a == b)
```

```
print(a != b)
```

```
print(a >= b)
```

```
print(a <= b)
```

Out [1]: `False`

`True``False``True``False``True`

العامل	شرح العامل	العملية
$>$	أكبر من: True عندما يكون المعامل الأيسر أكبر من المعامل الأيمن.	$x > y$
$<$	أصغر من: True عندما يكون المعامل الأيسر أصغر من المعامل الأيمن.	$x < y$
$==$	يساوي: إذا كان كلا المعاملين متساوين.	$x == y$
$!=$	غير مساوي: True إذا كان كلا المعاملين غير متساوين.	$x != y$
$>=$	أكبر من او يساوي: إذا كان المعامل الأيسر أكبر من او يساوي المعامل الأيمن.	$x >= y$
$<=$	أصغر من يساوي: إذا كان المعامل الأيسر أقل من او يساوي المعامل الأيمن.	$x <= y$

هياكل البيانات

- ✓ يعد تنظيم البيانات وإدارتها وتخزينها أمرًا مهمًا لأنه يتيح الوصول السهل والتغيير الفعال.
- ✓ تسمح لك هياكل البيانات بتنظيم بياناتك بطريقة يمكنك من خلالها تخزين مجموعة من البيانات وربطها معاً وتنفيذ العمليات بناءً عليها.
- ✓ هياكل البيانات هي الكتل الأساسية لحل المشكلات بكفاءة في العالم الحقيقي.
- ✓ إنها أدوات مثبتة ومحسنة تمنحك إطاراً سهلاً لتنظيم التطبيقات.

□ تدعم بايثون ضمنياً هياكل البيانات التي تسمح لك ب تخزين البيانات والوصول إليها.

□ يوجد إجمالي أربعة هياكل بيانات داخلية في لغة برمجة بايثون. تتضمن هياكل البيانات هذه القوائم والصفوف والقواميس والمجموعات.

□ هياكل بيانات بايثون بسيطة، ولكنها قوية جدًا.

تساعد هياكل البيانات فيما يلي:

- إدارة واستخدام مجموعة البيانات.
- البحث السريع عن بيانات محددة في قاعدة البيانات.
- إنشاء اتصالات هرمية أو علاقات بين نقاط البيانات.
- تبسيط وتسريع معالجة البيانات.

يتم تعريف القائمة `list` على أنها مجموعة مرتبة من العناصر. بمعنى آخر، يحتوي على قائمة متسلسلة من العناصر.
ترتيب العناصر هو خاصية متصلة تظل ثابتة طوال عمر القائمة.

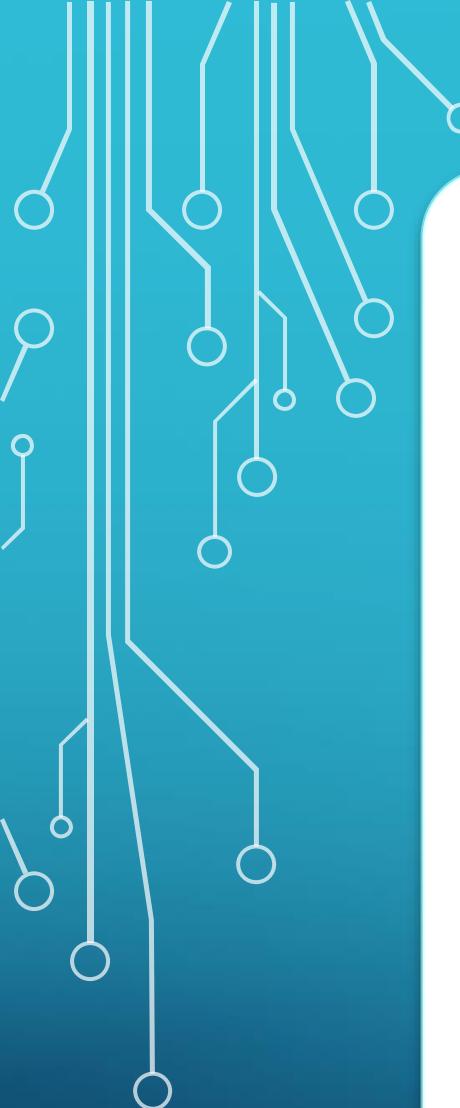
نظرًا لأن كل شيء في بايثون عبارة عن كائن، فإن إنشاء قائمة يؤدي بشكل أساسي إلى إنشاء كائن بايثون من نوع معين.
عند إنشاء قائمة، يجب وضع جميع العناصر في القائمة ومفصولة بفواصل لإخطار بايثون بأن القائمة قد تم إنشاؤها.
يمكن إنشاء قائمة في بايثون على النحو التالي:

```
List_A = [item 1, item 2, item 3, ...., item n]
```

```
In [1]: my_list = [1, 2, 3, 4]
```

```
In [2]: my_list
```

```
Out [2]: [1, 2, 3, 4]
```



إذا لم تضع أي عناصر داخل القوس، فستتلقى قائمة فارغة كإخراج:

```
In [1]: my_list1 = []
```

```
In [2]: my_list1
```

```
Out [2]: []
```

يمكن أن تكون كل قائمة عدداً من العناصر بأنواع مختلفة من البيانات:

```
In [3]: my_list = [1, 'example', 5.45]
```

```
In [4]: my_list
```

```
Out [4]: [1, 'example', 5.45]
```

بالإضافة إلى ذلك، يمكن أن تحتوي القائمة على قائمة أخرى كعنصر. تُعرف هذه القائمة بالقائمة المتداخلة:

```
In [5]: my_list1 = [1.56, 'python']
```

```
In [6]: my_list2 = ['example', 1]
```

```
In [7]: my_list = [5, my_list1 , 'data scientist', my_list2]
```

```
In [8]: my_list
```

```
Out [6]: [5, [1.56, 'python'], 'data scientist', ['example', 1]]
```

اضافة عنصر

يمكن إضافة عنصر إلى قائمة في بايثون باستخدام الدوال `append()` و `extend()` و `insert()`:

- الدالة `append()` يضيف كل العناصر المنقولة إلى القائمة كعنصر واحد.
- الدالة `extend()` يضيف عناصر إلى القائمة واحدة تلو الأخرى.
- الدالة `insert()` يضيف عنصراً في فهرس معين إلى القائمة.

```
In [7]: my_list = [7, 2, 1]
In [8]: my_list
Out [8]: [7, 2, 1]
In [9]: my_list.append([44, 15,'python'])
In [10]: my_list
Out [10]: [7, 2, 1, [44, 15, 'python']]
In [11]: my_list.extend(['example',2])
In [12]: my_list
Out [12]: [7, 2, 1, [44, 15, 'python'], 'example', 2]
In [13]: my_list.insert(1, 'insert_example1')
In [14]: my_list.insert(6, 'insert_example2')
In [15]: my_list
Out [15]: [7, 'insert_e1', 2, 1, [44, 15, 'python'], 'example',
'insert_e2', 2]
```



دوال اخرى

هناك العديد من الدوال الأخرى التي يمكن استخدامها عند العمل مع القوائم:

- ترجع الدالة `len()` طول القائمة.
- ترجع الدالة `index()` فهرس أحد العناصر (ملاحظة: إذا ظهر عنصر في القائمة عدة مرات ، فسيتم إرجاع الفهرس الأول المطابق).
- باستخدام الدالة `sort()` يتم فرز القائمة بترتيب صعودي.

```
In [33]: my_list1 = [4, 7, 5, 1, 4, 12]
```

```
In [34]: len(my_list)
```

```
Out [34]: 6
```

```
In [35]: my_list.index(5)
```

```
Out [35]: 2
```

```
In [36]: my_list.sort()
```

```
In [37]: my_list
```

```
Out [37]: [1, 4, 4, 5, 7, 12]
```

الصفوف (Tuples) هي سلسلة ثابتة وغير قابلة للتغيير من العناصر في بايثون. أهم فرق بينها وبين القوائم هو ثباتها. في حين أن القوائم قابلة للتغيير، لا تتوفر هذه الميزة في الصنف.

أسهل طريقة لإنشاء صنف هي كما يلي:

```
tuple_A = item 1, item 2, item 3,..., item n
```

يعد استخدام الأقواس في تكوين المجموعة أمرًا اختياريًّا ، لكن يوصى بالتمييز بين بداية المجموعة ونهايتها:

```
tuple_A = (item 1, item 2, item 3,..., item n)
```

```
In [1]: my_tuple = (1, 2, 3)
```

```
In [2]: my_tuple
```

```
Out [2]: (1, 2, 3)
```

باستدعاء tuple ، يمكنك تحويل كل سلسلة إلى عدة صنف:

```
In [3]: tuple([1, 3, 8])
```

```
Out [3]: (1, 3, 8)
```

```
In [4]: tup_A = tuple('Python')
```

```
In [5]: tup_A
```

```
Out [5]: ('P', 'y', 't', 'h', 'o', 'n')
```

```
In [6]: tup_A = tuple('Python')
```

```
In [7]: tup_A.index('y')
```

```
Out [7]: 1
```

```
In [8]: my_tuple = (8, 1, 4, 5, 3)
```

```
In [9]: max (my_tuple)
```

```
Out [9]: 8
```

```
In [9]: min (my_tuple)
```

```
Out [9]: 1
```

عند كتابة صنف بعنصر واحد، يجب استخدام فاصلة بعد العنصر. يمكن إنشاء صنف تحتوي على عنصر واحد على النحو التالي:

```
tuple_A = (item 1,)
```

لإنشاء صنف فارغ، يجب على المستخدم إنشاء زوج من الأقواس الفارغة على النحو التالي:

```
tuple_A = ()
```

```
In [10]: Empty_tuple=()
```

```
In [11]: Empty_tuple
```

```
Out [11]: ()
```

هناك العديد من الدوال التي يمكن استخدامها عند العمل مع الصنفوف:

- ترجع `len()` الدالة طول المجموعة.
- الدالة `index()` ترجع فهرس العنصر.
- تُرجع الدالة `max()` أكبر قيمة في المجموعة.
- تُرجع الدالة `min()` أصغر قيمة في المجموعة.

لماذا يفضل الصنف Tuple على القائمة List؟

- تعتبر الصنفوف أسرع من القوائم. يتم تخزين المجموعة في كتلة واحدة من الذاكرة. تعتبر المجموعات غير قابلة للتغيير ، لذلك ليست هناك حاجة إلى مساحة إضافية لتخزين العناصر الجديدة.
- يفضل الصنف عندما لا يرغب المستخدم في تغيير البيانات. في بعض الأحيان ، يريد المستخدم إنشاء كائن يظل سليماً طوال حياته. تعتبر المجموعات غير قابلة للتغيير، لذا يمكن استخدامها لمنع الإضافة أو التعديل أو الحذف العرض للبيانات.

القواميس

القاموس (Dictionary) في بایثون هو مجموعة غير مرتبة من القيم والتي على عكس هياكل البيانات الأخرى التي تحتوي على قيمة واحدة فقط كعنصر، يتم استخدامها لتخزين أزواج القيمة والمفتاح. يتم توفير قيمة المفتاح في القاموس لمزيد من التحسين في القاموس، يجب أن تكون المفاتيح فريدة.

```
my_dictionary= {key 1 : value 1, key 2 : value 2}
```

```
In [1]: my_dict = {'First': 'Python', 'Second': 'Julia'}
```

```
In [2]: my_dict
```

```
Out [2]: {'First': 'Python', 'Second': 'Julia'}
```

لتغيير قيم القاموس، عليك القيام بذلك باستخدام المفاتيح. لذلك، قم أولاً بالوصول إلى المفتاح
ثم قم بتغيير قيمته:

```
In [3]: my_dict['Second'] = 'golang'  
In [4]: my_dict  
Out [4]: {'First': 'Python', 'Second': 'golang'}
```

لإضافة قيم، ما عليك سوى إضافة زوج قيم - مفتاح آخر وفقاً للأمر التالي:

```
In [3]: my_dict['Third'] = 'Rust'  
In [4]: my_dict  
Out [4]: {'First': 'Python', 'Second': 'golang', 'Third': 'Rust'}
```

تُستخدم الدالة `pop()` لحذف قيمة (ترجع هذه الدالة أيضاً القيمة المحذوفة):

```
In [5]: my_dict.pop('Third')  
Out [5]: Rust  
In [6]: my_dict  
Out [6]: {'First': 'Python', 'Second': 'golang'}
```

يتم استخدام الدالة `clear()` لمسح القاموس بأكمله:

```
In [7]: my_dict.clear()  
In [8]: my_dict  
Out [8]: {}
```

- ✓ يتم تعريف المجموعة (Set) على أنها مجموعة من العناصر الفريدة التي لا تتبع ترتيباً معيناً.
- ✓ تُستخدم المجموعات عندما يكون وجود كائن في مجموعة من الكائنات أكثر أهمية من عدد المرات التي تظهر فيها الكائنات أو ترتيبها في المجموعات، إذا تكررت البيانات أكثر من مرة، يتم إدخالها في المجموعة مرة واحدة فقط.
- ✓ على عكس الصنفوف، فإن المجموعات قابلة للتغيير؛ أي أنه يمكن تعديلها، أو إضافتها، أو استبدالها، أو إزالتها.

يمكن عرض مجموعة مثال على النحو التالي:

```
set_a = {"item 1", "item 2", "item 3",....., "item n"}
```

```
In [1]: my_set = {2, 2, 3, 1, 4, 5, 5, 5}
```

```
In [2]: my_set
```

```
Out [2]: {1, 2, 3, 4, 5}
```

يمكنك استخدام الدالة add() لإضافة عنصر:

```
In [3]: my_set = {8, 1, 5}
```

```
In [4]: my_set.add(6)
```

```
In [5]: my_set
```

```
Out [5]: {1, 5, 6, 8}
```

هناك عمليات تطبق على مجموعات الرياضيات مثل الاتحاد (Union) والاشراك (Intersection) وما إلى ذلك. يوضح المثال التالي المجموعة المكونة من اتحاد مجموعتين:

```
In [3]: a = {1, 2, 3, 4, 5}
```

```
In [4]: b = {6, 4, 5, 1, 3, 8, 7}
```

```
In [5]: a.union(b)
```

```
Out [5]: {1, 2, 3, 4, 5, 6, 7, 8}
```

هيكل التحكم والحلقات

الأوامر الشرطية

نريد تنفيذ بعض الأوامر فقط في حالة استيفاء الشروط المحددة. يتم إنشاء البيانات الشرطية، المعروفة أيضاً باسم بيانات القرار، للقيام بذلك والتصرف بناءً على ما إذا كان شرط معين صحيحًا أم خطأ. في بايثون يمكننا اتخاذ القرارات باستخدام الأوامر التالية:

- امر `if`
- امر `if-else`
- امر `elif`
- امر `if-else` المتداخلة

امر if

في أوامر التحكم ، تكون عبارة if هي أبسط أشكالها. ينفذ الشرط ويقيمه على أنها صحيحة أو خاطئة:

```
if condition:  
    statement 1  
    statement 2  
    statement n
```

مثال:

```
In [1]: num = 3  
      if (num < 7):  
          print("Num is smaller than 7")  
Out [1]: Num is smaller than 7  
  
In [2]: a = 3  
      b = 2  
      if (a > b):  
          print("a is greater than b")  
Out [2]: a is greater than b
```

امر if-else

باستخدام تعليمة if-else ، إذا كان شرط معين صحيحاً ، فسيتم تنفيذ العبارات الموجودة داخل بلوك if ، وإذا كان الشرط خاطئاً ، فسيتم تنفيذ بلوك else:

```
if condition:  
    statement 1  
else:  
    statement 2
```

مثال:

```
In [1]: a = 1  
      b = 2  
      if (a > b):  
          print("a is greater than b")  
      else:  
          print("b is greater than a")  
  
Out [1]: b is greater than a
```

امر elif

بمساعدة أمر elif ، يمكننا اتخاذ قرار معقد. يتحقق الأمر elif من عدة شروط واحدة تلو الأخرى وينفذ بلوك الكود هذه إذا تم استيفاء الشرط:

```
if condition-1:  
    statement 1  
elif condition-2:  
    stetement 2  
elif condition-3:  
    stetement 3  
...  
else:  
    statement
```

مثال:

```
In [1]: num = -1  
      if (num > 0):  
          print("Number is positive")  
      elif (num < 0):  
          print("Number is negative")  
      else:  
          print("Number is Zero")  
Out [1]: Number is negative
```

الحلقات التكرارية

تسمح لنا الحلقات بتكرار جزء من الكود متى أردنا؛ طالما تم استيفاء الشرط الذي حددناه. تساعدنا الحلقات في تقليل تكرار الكود الخاص بنا، حيث يتيح لنا تشغيل عملية عدة مرات.

حلقة for

باستخدام حلقة for ، يمكن تمرير أي تسلسل تكراري أو متغير. يمكن أن يكون التسلسل عبارة عن سلسلة أو قائمة أو قاموس أو صفوف أو مجموعة. كيفية استخدام حلقة for للتكرار والتنقل على النحو التالي:

```
for iterator_var in sequence:  
    statements(s)
```

مثال:

```
In [1]: n = 4  
      for i in range(n):  
          print(i)
```

```
Out [1]: 0  
        1  
        2  
        3
```

```
In [2]: Str = 'Persian'  
      for i in Str:  
          print(i)
```

```
Out [2]: P  
        e  
        r
```

حلقة while

في بايثون ، تُستخدم حلقة while لتنفيذ مجموعة من الأوامر بشكل متكرر حتى يتم استيفاء شرط معين ، وعندما يكون الشرط خاطئاً ، يتم تنفيذ السطر مباشرة بعد حلقة البرنامج. كيفية استخدام حلقة while هو كما يلي:

```
while expression:  
    statement(s)
```

مثال:

```
In [1]: num = 10  
sum = 0  
i = 1  
while i <= num:  
    sum = sum + i  
    i = i + 1  
print("Sum of first 10 number is:", sum)  
Out [1]: Sum of first 10 number is: 55  
In [2]: count = 0  
while (count < 3):  
    count = count + 1  
    print("Hello")  
Out [2]: Hello  
Hello  
Hello
```

الدوال

- ✓ تُستخدم الدوال Functions في البرمجة لمجموعة من التعليمات التي تريد استخدامها بشكل متكرر، أو بسبب تعقيدتها، فمن الأفضل وضعها في روتين فرعي آخر والاتصال بها عند الضرورة.
- ✓ تُعد بر الدوال جزءاً مهماً من أي لغة برمجة لسبعين. أولاً، يسمحون لك بإعادة استخدام الكود الذي كتبته. على سبيل المثال، إذا كنت تعمل باستخدام قاعدة بيانات، فيجب عليك دائمًا التوacial مع قاعدة البيانات وإعلامها بالجدول الذي تريد الوصول إليه.
- ✓ عن طريق كتابة دالة، يمكنك القيام بذلك عن طريق كتابة سطر من التعليمات البرمجية في أي برنامج يحتاج إلى الوصول إلى قاعدة البيانات.
- ✓ ميزة أخرى لاستخدام دالة للقيام بذلك هي أنه إذا كنت بحاجة إلى تغيير نوع قاعدة البيانات التي تستخدمها، أو إذا وجدت عيّناً في المنطق الذي كتبته فيه الدالة لأول مرة، فيمكنك ببساطة عمل نسخة واحدة من الدالة ويمكن للتطبيقات الأخرى استخدام النسخة المعدلة ليتم تحديثها على الفور.

تعريف الدالة

فيما يلي أربع خطوات لتعريف دالة في بايثون:

1. استخدم الكلمة الأساسية `def` للإعلان عن الدالة ، ثم اختر اسمًا للدالة.
2. أضف معاملات إلى الدالة. ضعهم بين قوسين وقم بإنتهاء السطر بنقطتين (:) .
3. أضف التعبيرات التي يجب أن تنفذها الدالة.
4. إذا كان يجب أن تعرض الدالة شيئاً ما في الإخراج ، فقم بإنتهاء الدالة بعبارة `return`.
بدون عبارة `return` ، تقوم الدالة بإرجاع كائن `None`.

يمكن تطبيق الخطوات أعلاه في بايثون كما يلي:

```
def function_name(parameters):  
    statement(s)  
    return expression
```

مثال:

```
In [1]: def greet(name):
    print("Hello, " + name + ". Good morning!")
In [2]: greet("ali")
```

```
Out [2]: Hello, ali. Good morning!
```

```
In [3]: def absolute_value(num):
    if num >= 0:
        return num
    else:
        return -num
```

```
In [3]: absolute_value(5)
```

```
Out [3]: 5
```

```
In [4]: absolute_value(-8)
```

```
Out [4]: 8
```

```
In [5]: def evenOdd(x):
    if (x % 2 == 0):
        print("even")
    else:
        print("odd")
```

```
In [6]: evenOdd(5)
```

```
Out [6]: odd
```

```
In [7]: evenOdd(8)
```

```
Out [7]: even
```

العمل مع مكتبة NumPy

✓ NumPy هي مكتبة بايثون تستخدم للعمل مع المصفوفات. السبب في أهمية علم البيانات باستخدام بايثون هو أن معظم المكتبات في التعلم الآلي والتعلم العميق تعتمد على NumPy كأحد كتلها الأساسية، لأن السرعة والموارد مهمة جدًا بالنسبة لها.

✓ قد تتساءل عن سبب استخدامنا لمصفوفات NumPy عند وجود قوائم بايثون. في قائمة بايثون، يُعرض الغرض من المصفوفات. ومع ذلك، فهي بطيئة في المعالجة، وبطئها يمكن في كيفية تخزين الكائن في الذاكرة.

✓ كائن بايثون هو في الواقع مؤشر إلى موقع ذاكرة يخزن كل تفاصيل كائن، مثل البايت وقيمه. في حين أن هذه المعلومات الإضافية هي ما يجعل بايثون لغة ديناميكية، إلا أنها تأتي أيضًا بتكلفة. للتغلب على هذه المشكلة، نستخدم مصفوفات NumPy التي تحتوي على عناصر متجانسة فقط، أي العناصر التي لها نفس نوع البيانات.

✓ هذا يجعل تخزين المصفوفات ومعالجتها أكثر كفاءة. يهدف NumPy إلى توفير كائن مصفوفة أسرع بما يصل إلى 50 مرة من قوائم بايثون التقليدية. على عكس القوائم، يتم تخزين مصفوفات NumPy في مكان مستمر في الذاكرة، بحيث يمكن للعمليات الوصول إليها والتعامل معها بشكل فعال. يسمى هذا السلوك الإحالة المحلية في علوم الحاسوب.

✓ هذا هو السبب الرئيسي وراء كون NumPy أسرع من القوائم. أيضًا، يمكن إجراء العمليات الأولية باستخدام مصفوفات NumPy، وهو أمر غير ممكן مع قوائم بايثون! هذا هو سبب تفضيل مصفوفات NumPy على قوائم بايثون عند إجراء عمليات حسابية على كميات كبيرة من البيانات.

استيراد NumPy

عندما ترييد استخدام حزمة أو مكتبة في التعليمات البرمجية الخاصة بك، يجب عليك أولاً إتاحتها. لبدء استخدام NumPy وجميع الدوال في NumPy، يجب عليك استيرادها. يمكن القيام بذلك بسهولة باستخدام أمر الاستيراد:

```
In [1]: import numpy as np
```

يرمز np إلى NumPy الذي يستخدمه مجتمع علم البيانات. نقوم بتقصير NumPy إلى np لتوفير الوقت والحفظ أيضًا على الكود القياسي بحيث يمكن لأي شخص يعمل باستخدام الكود الخاص بنا فهمه وتنفيذ بسهولة.

إنشاء مصفوفة NumPy

تُستخدم طريقة np.array() لإنشاء مصفوفة أساسية في NumPy. الشيء الوحيد الذي يجب تضمينه هو قيم المصفوفة كقائمة:

```
In [1]: np.array([1,2,3,4])
```

```
Out [1]: array([1, 2, 3, 4])
```

تحتوي هذه المصفوفة على قيم عدد صحيح. يمكنك تحديد نوع البيانات في وسيطة :dtype

```
In [2]: np.array([1,2,3,4], dtype=np.float32)
```

```
Out [2]: array([1., 2., 3., 4.], dtype=float32)
```

باستخدام الأقواس المربعة ([]) يمكننا الوصول إلى عناصر المصفوفة. عند الوصول إلى العناصر، يضع في اعتبارك أن الفهرسة في NumPy تبدأ من 0. هذا يعني أنه إذا كنت تريد الوصول إلى العنصر الأول في المصفوفة، في يمكنك الوصول إليه باستخدام: 0

```
In [3]: a = np.array([5 , 1, 3, 7])  
Out [3]: 5
```

يمكن أيضًا أن تكون مصفوفات NumPy متعددة الأبعاد:

```
In [4]: a = np.array([[1 , 5, 2], [6, 8, 1], [10, 3, 11]])  
a  
Out [4]: array([[ 1,  5,  2],  
                 [ 6,  8,  1],  
                 [10,  3, 11]])  
In [5]: a[0]  
  
Out [5]: array([1, 5, 2])  
In [6]: a[2]  
  
Out [6]: array([10,  3, 11])  
In [7]: a[0][0]  
  
Out [7]: 1
```

المصفوفة الطرفية

يتيح لك NumPy إنشاء مصفوفة من الأصفار باستخدام طريقة `np.zeros()`. كل ما عليك فعله هو إدخال شكل¹ المصفوفة المطلوب:

```
In [1]: np.zeros(7)
```

```
Out [1]: array([0., 0., 0., 0., 0., 0., 0.])
```

المصفوفة السابقة عبارة عن مصفوفة ذات بعد واحد. لإنشاء مصفوفة ثنائية الأبعاد، قم بما يلي:

```
In [2]: np.zeros((2,6))
```

```
Out [2]: array([[0., 0., 0., 0., 0., 0.],  
[0., 0., 0., 0., 0., 0.]])
```

المصفوفة الواحدية

يمكنك أيضاً إنشاء مصفوفة من الوحدات باستخدام طريقة `np.ones()`:

```
In [1]: np.ones(6)
```

```
Out [1]: array([1., 1., 1., 1., 1., 1.])
```

إضافة وحذف وفرز العناصر

يمكنك إضافة عناصر إلى المصفوفة الخاصة بك باستخدام طريقة `np.append()`:

```
In [1]: a = np.array([5, 1, 2, 3, 9, 4, 7])
```

```
a
```

```
Out [1]: array([5, 1, 2, 3, 9, 4, 7])
```

```
In [2]: np.append(a, [12,2,1])
```

```
Out [2]: array([5, 1, 2, 3, 9, 4, 7, 12, 2, 1])
```

يتم استخدام طريقة `np.delete()` لحذف عنصر في موقع معين:

```
In [3]: a = np.array([5, 1, 2, 3, 9, 4, 7])
```

```
a
```

```
Out [3]: array([5, 1, 2, 3, 9, 4, 7])
```

```
In [4]: np.delete(a, 0)
```

```
Out [4]: array([1, 2, 3, 9, 4, 7])
```

تحديد شكل وحجم المصفوفة

باستخدام `ndim`, يمكن الحصول على عدد محاور أو أبعاد المصفوفة:

```
In [1]: a = np.array([[8,5,7,4,1,6],  
                   [9,2,3,7,5,1]])  
      a.ndim  
Out [1]: 2
```

`size` يخبرك بالعدد الإجمالي لعناصر المصفوفة:

```
In [2]: a = np.array([[8,5,7,4,1,6],  
                   [9,2,3,7,5,1]])  
      a.size  
Out [2]: 12
```

يستخدم `shape` لإيجاد شكل المصفوفة:

```
In [3]: a = np.array([[8,5,7,4,1,6],  
                   [9,2,3,7,5,1]])  
      a.shape  
Out [3]: (2, 6)
```

الخلاصة

- بايثون هي واحدة من أكثر اللغات قيمة وإثارة للاهتمام لتحليل البيانات.
- بايثون هي واحدة من أسهل اللغات للبدء بها. أيضا، هذه البساطة لا تحد من التسهيلات التي تحتاجها.
- **Jupyter Notebook** هو أداة قوية بشكل لا يصدق لتطوير وتقديم مشاريع علم البيانات التفاعلية التي يمكن أن تتضمن نص، أو صورة أو صوتاً أو فيديو بالإضافة إلى تنفيذ التعليمات البرمجية.
- **NumPy** هي مكتبة بايثون تستخدم للعمل مع المصفوفات.
- يهدف **NumPy** إلى توفير كائن مصفوفة أسرع بما يصل إلى 50 مرة من قوائم بايثون التقليدية