Python 101

Une introduction au langage Python

Python ? Comme le reptile ?

- Euh, non… pas comme le reptile
- Le logo en est inspiré mais…
- Son nom fut choisi en hommage aux Monty Python et leur émission à la radio : "The Monty Python's Flying Circus"

Python ? Comme le reptile ?

- Python a été créé au début des années 1990 par Guido van Rossum
- Successeur d'un langage appelé "ABC"
- Guido reste, à ce jour, l'auteur principal bien que le langage intègre les contributions de bien d'autres
- En mai 2000, Guido et l'équipe de développement principale intègrent "BeOpen" pour former l'équipe "BeOpen PythonLabs"
- En octobre 2000, l'équipe "PythonLabs" part chez "Digital Creations" ("Zope Corporation" aujourd'hui)
- En 2001, la "Python Software Foundation" est créée pour gérer tous les aspects de la propriété intellectuelle de Python

Python ? Comme le reptile ?

- Python est un langage de développement puissant et facile à apprendre
- Il intègre des structures de données de haut niveau
- Il à une approche simple mais efficace de la programmation orientée objet (POO)
- Sa syntaxe élégante, son typage dynamique et sa nature interprétée en font un langage idéal pour le scripting et le développement rapide d'applications, dans de nombreux domaines et sur de multiples plateformes

Python ? Comme le reptile ?

- L'interpréteur Python et sa vaste bibliothèque standard sont disponibles librement, sous forme de sources ou de binaires, pour toutes les plateformes majeures, depuis le site Internet http://www.python.org/ et peuvent être librement redistribués.
- Le même site distribue et contient des liens vers des modules, des programmes et des outils tiers ainsi que vers de la documentation supplémentaire.
- L'interpréteur Python peut être facilement étendu par de nouvelles fonctions et types de données implémentés en C ou C++ (ou tout autre langage appelable depuis le C).
- Python est également adapté comme langage d'extension pour personnaliser des applications.

Le "Zen" de Python

- Le beau est préférable au laid.
- L'explicite est préférable à l'implicite.
- Le simple est préférable au complexe.
- Le complexe est préférable au compliqué.
- L'horizontal est préférable à l'imbriqué.
- L'aéré est préférable au dense.
- La lisibilité compte. Les cas spéciaux ne le sont pas assez pour transgresser les règles.
- Sauf si le cas pratique bat le cas théorique.
- Les erreurs ne devraient jamais arriver silencieusement.
- Sauf si on les a explicitement rendues silencieuses.
- En cas de doute, ne tentez pas de deviner.
- Il devrait y avoir une, et de préférence une seule, manière évidente de le faire.
- Même si cette manière peut ne pas sembler évidente au premier abord sauf si vous êtes néerlandais.
- Ce qui est fait maintenant est préférable à ce qui ne sera jamais fait.
- Même si jamais est souvent mieux que tout de suite.
- Si l'implémentation est difficile à expliquer, c'est que c'est une mauvaise idée.
- Si l'implémentation est facile à expliquer, c'est que c'est peut-être une bonne idée.
- Les espaces de noms sont une brillante idée, créons-en plus !

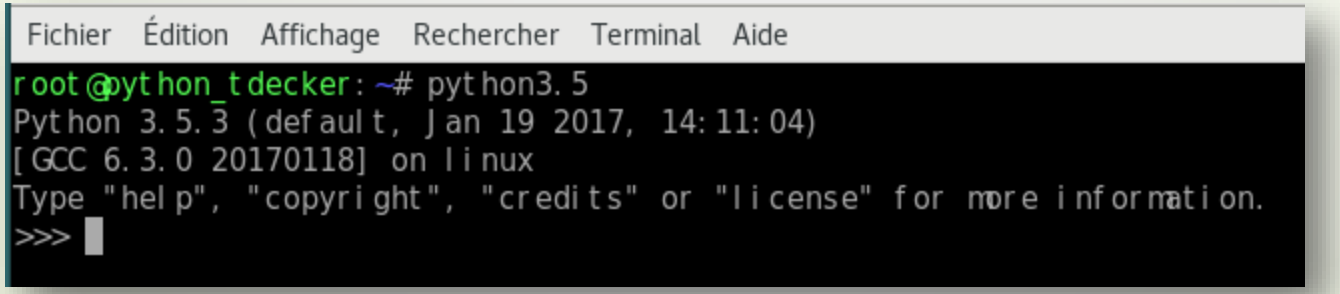
Le "Zen" de Python (*import this*)

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

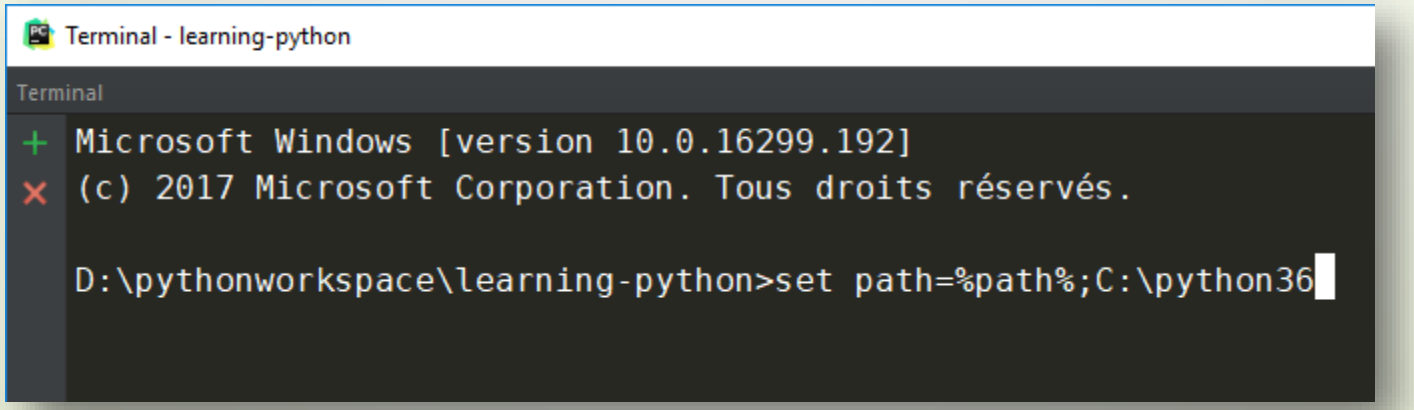
Invoquer l'interpréteur Python

- L'interpréteur Python se trouve en général dans /usr/local/bin/python3.6 sur les machines où il est disponible
- Ajouter /usr/local/bin au chemin de recherche de votre Shell Unix rend possible de le lancer en tapant la commande :



Invoquer l'interpréteur Python

- Sur les machines Windows, l'installation Python est habituellement placée dans C:\Python36, même si vous pouvez changer cela lorsque vous lancez l'installateur
- Pour ajouter ce dossier à votre chemin de recherche, vous pouvez taper la commande suivante dans un prompt de commande d'une machine DOS



Invoquer l'interpréteur Python

- Taper un caractère de fin de fichier (***Ctrl-D*** sous Unix, ***Ctrl-Z*** sous Windows) à la suite d'une invite de commande primaire provoque la fermeture de l'interpréteur avec un statut d'erreur nul
- Si cela ne fonctionne pas, vous pouvez fermer l'interpréteur en tapant la commande ***quit()***
- L'interpréteur opère de façon similaire au Shell Unix
- Lorsqu'il est appelé avec l'entrée standard connectée à un périphérique tty, il lit et exécute les commandes de façon interactive
- Lorsqu'il est appelé avec un nom de fichier en argument ou avec un fichier comme entrée standard, il lit et exécute un script depuis ce fichier
- Quand un fichier de script est utilisé, il est parfois utile de pouvoir lancer le script puis d'entrer dans le mode interactif après coup. Cela est possible en passant ***-i*** avant le script

Passage d'arguments

- Lorsqu'ils sont connus de l'interpréteur, le nom du script et les arguments additionnels sont représentés sous forme d'une liste assignée à la variable ***argv*** du module ***sys***
- Vous pouvez y accéder en exécutant ***import sys***
- La liste contient au minimum un élément
- Quand aucun script ni aucun arguments ne sont donnés, ***sys.argv[0]*** est une chaine vide
- Quand '**-**' (qui représente l'entrée standard) est passé comme nom de script, ***sys.argv[0]*** contient '**-**'
- Quand -c commande est utilisé, ***sys.argv[0]*** contient '***-c***'
- Enfin, quand ***-m*** module est utilisé, le nom complet du module est assigné à ***sys.argv[0]***
- Les options trouvées après ***-c*** commande ou ***-m*** module ne sont pas lues comme options de l'interpréteur Python mais laissées dans sys.argv pour être utilisée par le module ou la commande

Mode interactif

- Lorsque des commandes sont lues depuis un tty, l'interpréteur est dit être en mode interactif
- Dans ce mode, il demande la commande suivante avec le prompt primaire, en général trois signes plus-grand-que (>>>)
- Pour les lignes de continuation, il affiche le prompt secondaire, par défaut trois points (...)
- L'interpréteur affiche un message de bienvenue indiquant son numéro de version et une notice de copyright avant d'afficher le premier prompt

```
D:\pythonworkspace\learning-python>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> la_terre_est_plate = True
>>> if la_terre_est_plate:
...     print("Attention à ne pas tomber!")
...
Attention à ne pas tomber!
>>>
```

Encodage du code source

- Par défaut Python considère que ses fichiers source sont encodés en UTF-8
- Dans cet encodage, les caractères de la plupart des langues peuvent être utilisés ensemble dans les chaînes de caractères, identifiants, et commentaires, bien que la bibliothèque standard n'utilise que des caractères ASCII dans ses identifiants, une bonne habitude que tout code portable devrait suivre
- Pour afficher correctement tous ces caractères, votre éditeur doit reconnaître que le fichier est en UTF-8, et utiliser une fonte de caractère qui comprend tous les caractères utilisés dans le fichier
- Pour annoncer un encodage différent de l'encodage par défaut, une ligne de commentaire particulière doit être ajoutée à la première ligne du fichier
- Sa syntaxe est la suivante : ***# -*- coding: encoding -*-*** , ou encoding est un des codecs supportés par Python (par exemple: ***cp-1252*** pour Windows)

Encodage du code source

- Une exception à la règle précédente est lorsque la première ligne est un shebang UNIX
- Dans ce cas, la déclaration de l'encodage doit être placé sur la seconde ligne du fichier..
- Par exemple :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

Introduction informelle

- Dans les exemples qui suivent, les entrées et sorties se distinguent par la présence ou l'absence d'invite (>>> et ...)
- Pour reproduire les exemples, vous devez taper tout ce qui est après l'invite, au moment où celle-ci apparaît
- Les lignes qui n'affichent pas d'invite sont les sorties de l'interpréteur
- Notez qu'une invite secondaire (...) affichée seule sur une ligne dans un exemple indique que vous devez entrer une ligne vide
- Ceci est utilisé pour terminer une commande multi-lignes

Introduction informelle

- Beaucoup d'exemples, même ceux saisis à l'invite de l'interpréteur, incluent des commentaires
- Les commentaires en Python commencent avec un caractère dièse, #, et s'étendent jusqu'à la fin de la ligne
- Un commentaire peut apparaître au début d'une ligne ou à la suite d'un espace ou de code, mais pas à l'intérieur d'une chaîne de caractères littérale
- Un caractère dièse à l'intérieur d'une chaîne de caractères est juste un caractère dièse
- Comme les commentaires ne servent qu'à expliquer le code et ne sont pas interprétés par Python, ils peuvent être ignorés lorsque vous tapez les exemples

```
>>>
>>> # C'est le premier commentaire
... spam = 1 # Et c'est le second commentaire
>>> text ="# Ce n'est pas un commentaire... C'est entre quotes!"
>>>
```

Les nombres

- L'interpréteur agit comme une simple calculatrice
- Vous pouvez lui entrer une expression et il vous affiche la valeur. La syntaxe des expressions est simple
- Les opérateurs `+`, `-`, `*` et `/` fonctionnent comme dans la plupart des langages
- Les parenthèses peuvent être utilisées pour faire des regroupements

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5 # La division retourne toujours un nombre décimal
1.6
>>>
```

Les nombres

- Les nombre entiers (comme 2, 4, 20) sont de type ***int***, alors que les décimaux (comme 5.0, 1.6) sont de type ***float***
- Les divisions (/) donnent toujours des float
- Utilisez l'opérateur // pour effectuer des divisions entières, et donc obtenir un résultat entier
- Pour obtenir le reste de cette division entière, utilisez l'opérateur %
- Avec Python il est possible de calculer des puissances avec l'opérateur **

```
>>> 17 / 3 # La division classique retourne un nombre décimal (float)
5.666666666666667
>>> 17 // 3 # La division entière retourne la partie entière
5
>>> 17 % 3 # L'opérateur % retourne le reste de la division
2
>>> 5 * 3 + 2 # partie entière * diviseur + reste
17
>>>
```

```
>>> 5 ** 2 # 5 au carré
25
>>> 2 ** 7 # 2 puissance 7
128
>>>
```

Les nombres

- Le signe égal (=) est utilisé pour affecter une valeur à une variable
- Après cela, aucun résultat n'est affiché avant l'invite suivante
- Si une variable n'est pas « définie » (si aucune valeur ne lui a été affecté), l'utiliser engendrera une erreur
- Il y a un support complet des nombres à virgule flottante
- Les opérateurs avec des types d'opérandes mélangés convertissent l'opérande entier en virgule flottante

```
>>> largeur = 20
>>> hauteur = 5 * 9
>>> largeur * hauteur
900
>>> une_variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'une_variable' is not defined
>>> 4 * 3.75 - 1
14.0
```

Les nombres

- En mode interactif, la dernière expression affichée est affectée à la variable _
- Ce qui signifie que lorsque vous utilisez Python comme calculatrice, il est parfois plus simple de continuer des calculs
- Cette variable doit être considérée comme une variable en lecture seule par l'utilisateur
- Ne lui affectez pas de valeur explicitement
- Vous créeriez ainsi une variable locale indépendante avec le même nom qui masquerait la variable native et son fonctionnement magique
- En plus des ***int*** et des ***float***, il existe les ***Decimal*** et les ***Fraction***
- Python gère aussi les nombre complexes, en utilisant le suffixe j ou J pour indiquer la partie imaginaire (tel que: 3+5j)

```
>>> taxe = 12.5 / 100
>>> prix = 100.50
>>> prix * taxe
12.5625
>>> prix + _
113.0625
>>> round(_,2)
113.06
```

Les chaines de caractères

- Au delà des nombres, Python peut aussi manipuler des chaînes de caractères, qui peuvent être exprimés de différentes manières
- Elles peuvent être écrites entre guillemets simples ('...') ou entre guillemets ("...") sans distinction
- \ peut être utilisé pour protéger un guillemet

```
>>> 'spam eggs' # Simples quotes
'spam eggs'
>>> 'L\'arrivée des problèmes' # Utilisez \' pour échapper la simple quote
"L'arrivée des problèmes"
>>> "L'arrivée des problèmes" # Ou utilisez les doubles quotes
"L'arrivée des problèmes"
>>> 'Il a dit "oui" !'
'Il a dit "oui" !'
>>> "Il a dit \"oui\" !"
'Il a dit "oui" !'
>>>
```

Les chaines de caractères

- En mode interactif, l'interpréteur affiche les chaînes de caractères entre guillemets et en protégeant les guillemets et autres caractères spéciaux avec des antislash
- Bien que cela puisse paraître différent de ce qui a été donné (les guillemets peuvent changer) La chaîne est affichée entre guillemets si elle contient un guillemet simple mais aucun guillemet, sinon elle est affichée entre guillemets simples
- La fonction ***print()*** affiche les chaînes de manière plus lisible, en retirant les guillemets et en affichant les caractères spéciaux qui étaient protégées par un antislash

```
>>> "Il a dit \"oui\" !"
'Il a dit "oui" !'
>>> print("Il a dit \"oui\" !")
Il a dit "oui" !
>>> s = "Première ligne\nSeconde ligne"
>>> s # Sans print(), \n est inclu dans la sortie
'Première ligne\nSeconde ligne'
>>> print(s) # Avec print() \n produit un retour à la ligne
Première ligne
Seconde ligne
>>>
```

Les chaines de caractères

- Pour éviter que les caractères précédés d'un \ ne soient interprétés comme étant spéciaux, utilisez les chaînes brutes (raw strings) en préfixant la chaîne d'un r
- Les chaînes de caractères peuvent s'étendre sur plusieurs lignes
- On peut utiliser les triples guillemets, simples ou doubles: '''...''' ou """..."""
- Les retours à la ligne sont automatiquement inclus, mais on peut l'en empêcher en ajoutant \ à la fin de la ligne

```
>>> print('C:\le\nom') # Ici, \n signifie newline
C:\le
om
>>> print(r'C:\le\nom') # Notez le r avant la quote
C:\le\nom
>>> print("""\
... Usage: blah [OPTIONS]
...      -h              Affiche ce message
...      -H hostname     Hostname a connecter
... """)
Usage: blah [OPTIONS]
        -h              Affiche ce message
        -H hostname     Hostname a connecter
```

Les chaines de caractères

- Les chaînes peuvent être concaténées (collées ensemble) avec l'opérateur +, et répétées avec l'opérateur *
- Plusieurs chaînes de caractères, écrites littéralement (c'est à dire entre guillemets), côte à côte, sont automatiquement concaténées
- Cette fonctionnalité est surtout intéressante pour couper des chaînes trop longues
- Cela ne fonctionne cependant qu'avec les chaînes littérales, pas les variables ni les expressions
- Pour concaténer des variables, ou des variables avec des chaînes littérales, utilisez l'opérateur +

```
>>> # 3 fois 'un', suivi par 'inum'
... 3 * 'un' + 'inum'
'unununinum'
>>> 'Py' 'thon'
'Python'
>>> text = ('Mettre plusieurs chaines entre parathèses '
...         'Pour les joindre ensembles')
>>> text
'Mettre plusieurs chaines entre parathèses Pour les joindre ensembles'
>>>
```

```
>>> prefixe + 'thon'
'Python'
>>>
```

Les chaines de caractères

- Les chaînes de caractères peuvent être indexées (accéder aux caractères par leur position), le premier caractère d'une chaîne est à la position 0
- Il n'existe pas de type distinct pour les caractères, un caractère est simplement une chaîne de longueur 1
- Les indices peuvent également être négatifs, pour effectuer un décompte en partant de la droite
- Notez que puisque -0 égal 0, les indices négatifs commencent par -1

```
>>> mot = 'Python'
>>> mot[0] # Caractère à la position 0
'P'
>>> mot[5] # Caractère à la position 5
'n'
>>>
```

```
>>> mot[-1] # Dernier caractère
'n'
>>> mot[-2] # Avant dernier caractère
'o'
>>> mot[-6]
'P'
>>>
```

Les chaines de caractères

- En plus d'accéder à un élément par son indice, il est aussi possible de trancher une liste
- Accéder à une chaîne par un indice permet d'obtenir un caractère, alors que la trancher permet d'obtenir une sous-chaîne
- On parle de ***slicing***
- Notez que le début est toujours inclus et la fin toujours exclue
- Cela assure que s[:i] + s[i:] est toujours égal à s

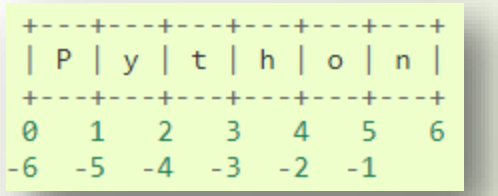
```
>>> # Caractères de la position 0 (inclue) à la position 2 (exclue)
... mot[0:2]
'Py'
>>> # Caractères de la position 2 (inclue) à la position 5 (exclue)
... mot[2:5]
'tho'
>>>
```

```
>>> mot[:2] + mot[2:]
'Python'
>>> mot[:4] + mot[4:]
'Python'
>>>
```

Les chaines de caractères

- Les indices par tranches ont des valeurs par défaut utiles
- Le premier indice lorsqu'il est omis équivaut à zéro, le second à la taille de la chaîne de caractères
- Une façon de mémoriser la façon dont les tranches fonctionnent est de penser que les indices pointent entre les caractères, le côté gauche du premier caractère ayant la position 0
- Le côté droit du dernier caractère d'une chaîne de n caractères a alors pour indice n
- Pour des indices non négatifs, la longueur d'une tranche est la différence entre ces indices, si les deux sont entre les bornes
- Par exemple, la longueur de mot[1:3] est 2

```
>>> mot[:2]
'Py'
>>> mot[4:]
'on'
>>> mot[-2:]
'on'
>>>
```



Les chaines de caractères

- Utiliser un indice trop grand générera une erreur
- Cependant, les indices hors bornes sont gérés silencieusement lorsqu'ils sont utilisés dans des tranches
- Les chaînes de caractères, en Python ne peuvent pas être modifiées, on dit qu'elles sont immuable
- Affecter une nouvelle valeur à un indice dans une chaîne produit une erreur

```
>>> mot[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> mot[4:42]
'on'
>>> mot[42:]
''
>>>
```

```
>>> mot[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> mot[2:] = 'Py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Les chaines de caractères

- Si vous avez besoin d'une chaîne différente, vous devez en créer une autre
- La fonction native ***len()*** renvoie la longueur d'une chaîne

```
>>> 'J' + mot[1:]
'Jython'
>>> mot[:2] + 'Py'
'PyPy'
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
>>>
```

Les listes

- Python connaît différents types de données combinés, utilisés pour regrouper plusieurs valeurs
- La plus souple est la liste, qui peut être écrite comme une suite de valeurs (éléments) séparés par des virgules placée entre crochets
- Les éléments d'une liste ne sont pas obligatoirement tous du même type, bien qu'à l'usage ce soit souvent le cas
- Comme les chaînes de caractères (et toute autre types de séquence), les listes peuvent être indicées et découpées

```
>>> carres = [1, 4, 9, 16, 25]
>>> carres
[1, 4, 9, 16, 25]
>>>
```

```
>>> carres[0] # L'indexation retourne l'élément
1
>>> carres[-1]
25
>>> carres[-3:] # Le slicing retourne une nouvelle liste
[9, 16, 25]
>>>
```

Les listes

- Toutes les opérations par tranches (slicing) renvoient une nouvelle liste contenant les éléments demandés
- Cela signifie que l'opération suivante renvoie une copie superficielle de la liste (shallow copy)
- Les listes gèrent aussi les opérations comme les concaténations

```
>>> carres_bis = carres[:]
>>> carres_bis
[1, 4, 9, 16, 25]
>>>
>>> carres + carres_bis
[1, 4, 9, 16, 25, 1, 4, 9, 16, 25]
>>>
```

Les listes

- Mais à la différence des chaînes qui sont immuables, les listes sont muables
- Il est possible de changer leur contenu
- Il est aussi possible d'ajouter des éléments à la fin d'une liste avec la méthode append()

```
>>> cubes = [1, 8, 27, 65, 125] # Une erreur dans la liste
>>> 4 ** 3 # Le cube de 4 est 64, pas 65
64
>>> cubes[3] = 64
>>> cubes
[1, 8, 27, 64, 125]
>>>
```

```
>>> cubes.append(216)
>>> cubes.append(7 ** 3)
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
>>>
```

Les listes

- Des affectations de tranches sont également possibles, ce qui peut même modifier la taille de la liste ou la vider complètement

```
>>> lettres = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> lettres
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # Remplacer certaines valeurs
... lettres[2:5] = ['C', 'D', 'E']
>>> lettres
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # Les enlever
... lettres[2:5] = []
>>> lettres
['a', 'b', 'f', 'g']
>>> # Vider la liste en remplaçant tous les élément par une liste vide
... lettres[:] = []
>>> lettres
[]
>>>
```

Les listes

- Il est possible d'imbriquer des listes (de créer des listes contenant d'autres listes)
- La primitive ***len()*** s'applique aussi aux listes

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
>>>
```

```
>>> lettres = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> len(lettres)
7
>>>
```

Vers la programmation

- Bien entendu, on peut utiliser Python pour des tâches plus compliquées que d'additionner deux et deux
- Par exemple, on peut écrire une sous-séquence initiale de la suite de Fibonacci

```
>>> # Suite de Fibonacci
... # La somme de deux éléments donne le suivant
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
>>>
```

Vers la programmation

- L'exemple précédent introduit plusieurs nouvelles fonctionnalités
- La première ligne contient une affectation multiple
  - les variables a et b se voient affecter simultanément leurs nouvelles valeurs 0 et 1
  - Cette méthode est encore utilisée à la dernière ligne, pour démontrer que les expressions sur la partie droite de l'affectation sont toutes évaluées avant que les affectations ne soient effectuées
  - Ces expressions en partie droite sont toujours évaluées de la gauche vers la droite

Vers la programmation

- La boucle ***while*** s'exécute tant que la condition (ici : b < 10) reste vraie
  - En Python, comme en C, tout entier différent de zéro est vrai et zéro est faux
  - La condition peut aussi être une chaîne de caractères, une liste, ou en fait toute séquence
  - Une séquence avec une valeur non nulle est vraie, une séquence vide est fausse
  - Le test utilisé dans l'exemple est une simple comparaison
  - Les opérateurs de comparaison standards sont écrits comme en C : **<** (inférieur), **>** (supérieur), **==** (égal), **<=** (inférieur ou égal), **>=** (supérieur ou égal) et ***!=*** (non égal)

Vers la programmation

- Le corps de la boucle est ***indenté***
  - l'indentation est la méthode utilisée par Python pour regrouper des instructions
  - En mode interactif, vous devez saisir une tabulation ou des espaces pour chaque ligne indentée
  - En pratique, vous aurez intérêt à utiliser un éditeur de texte pour les saisies plus compliquées
  - Tous les éditeurs de texte dignes de ce nom disposent d'une fonction d'auto-indentation
  - Lorsqu'une expression composée est saisie en mode interactif, elle doit être suivie d'une ligne vide pour indiquer qu'elle est terminée (car l'analyseur ne peut pas deviner que vous venez de saisir la dernière ligne)
  - Notez bien que toutes les lignes à l'intérieur d'un bloc doivent être indentées au même niveau

Vers la programmation

- La fonction ***print()*** écrit les valeur des paramètres qui lui sont fournis
  - Ce n'est pas la même chose que d'écrire l'expression que vous voulez afficher (comme nous l'avons fait dans l'exemple de la calculatrice), dû à la manière de print de gérer les paramètres multiples, les nombres décimaux, et les chaînes
  - Les chaînes sont affichées sans apostrophes et un espace est inséré entre les éléments de telle sorte que vous pouvez facilement formater les choses

- Le paramètre nommé ***end*** peut servir pour enlever le retour à la ligne, ou terminer la ligne par une autre chaîne

```
>>> i = 256 * 256
>>> print('La valeur de i est', i)
La valeur de i est 65536
>>>
```

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=',')
...     a, b = b, a + b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
>>>
```

L'instruction *if*

- Sans doute la plus connue
- Il peut y avoir un nombre quelconque de parties ***elif***, et la partie ***else*** est facultative. Le mot clé "***elif***" est un raccourci pour "else if", mais permet de gagner un niveau d'indentation. Une séquence if … elif … elif … est par ailleurs équivalente aux instructions switch ou case disponibles dans d'autres langage

```
>>> x = int(input("Entrez un enter SVP : "))
Entrez un enter SVP : 42
>>> if x < 0:
...     x = 0
...     print("Entier négatif changé en zéro")
... elif x == 0:
...     print("Zéro")
... elif x == 1:
...     print("Un")
... else:
...     print("Plus")
...
Plus
>>>
```

L'instruction *for*

- L'instruction ***for*** que propose Python est un peu différente de celle que l'on peut trouver en C ou en Pascal
- Au lieu de toujours itérer sur une suite arithmétique de nombres (comme en Pascal), ou de donner à l'utilisateur la possibilité de définir le pas d'itération et la condition de fin (comme en C), l'instruction ***for*** en Python itère sur les éléments d'une séquence (qui peut être une liste, une chaîne de caractères…), dans l'ordre dans lequel ils apparaissent dans la séquence

```
>>> # Longueurs de chaines
... mots = ['Chien', 'Chat', 'Poule', 'Trouvez l\'intrus']
>>> for mot in mots:
...     print(mot, len(mot))
...
Chien 5
Chat 4
Poule 5
Trouvez l'intrus 16
>>>
```

L'instruction *for*

- Si vous devez modifier la séquence sur laquelle s'effectue l'itération à l'intérieur de la boucle (par exemple pour dupliquer ou supprimer un élément), il est plus que recommandé de commencer par en faire une copie, celle-ci n'étant pas implicite
- La notation « par tranches » rend cette opération particulièrement simple
- Avec ***for mot in mots:***, l'exemple tenterait de créer une liste infinie, en insérant 'Trouvez l'intrus" indéfiniment

```
>>> for mot in mots[:]:
...     if len(mot) > 6:
...             mots.insert(0, mot)
...
>>> mots
["Trouvez l'intrus", 'Chien', 'Chat', 'Poule', "Trouvez l'intrus"]
>>>
```

La fonction *range()*

- Si vous devez itérer sur une suite de nombres, la fonction intégrée range() est faite pour cela
- Elle génère des suites arithmétiques
- Le dernier élément fourni en paramètre ne fait jamais partie de la liste générée ; range(10) génère une liste de 10 valeurs, dont les valeurs vont de 0 à 9

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

La fonction *range()*

- Il est possible de spécifier une valeur de début et/ou une valeur d'incrément différente(s) (y compris négative pour cette dernière, que l'on appelle également parfois le "pas")

```
>>> for i in range(5, 10):
...     print(i)
...
5
6
7
8
9
```

```
>>> for i in range(0, 10, 3):
...     print(i)
...
0
3
6
9
```

```
>>> for i in range(-10, -100, -30):
...     print(i)
...
-10
-40
-70
```

La fonction *range()*

- Pour itérer sur les indices d'une séquence, on peut combiner les fonctions *range()* et len()

```
>>> a = ['Thierry', 'DECKER', 'enseigne', 'Python']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Thierry
1 DECKER
2 enseigne
3 Python
```

- Cependant, dans la plupart des cas, il est plus pratique d'utiliser la fonction *enumerate()*

La fonction *range()*

- Une chose étrange se produit lorsqu'on affiche un range
- Les objets donnés par ***range()*** se comportent presque comme des listes, mais n'en sont pas
- Ce sont des objets qui génèrent les éléments de la séquence au fur et à mesure de leur itération, économisant ainsi de l'espace.
- On appelle de tels objets des itérables, c'est à dire des objets qui conviennent à des itérateurs, des fonctions ou constructions qui s'attendent à quelque-chose duquel ils peuvent tirer des éléments, successives successivement, jusqu'à épuisement
- On a vu que l'instruction ***for*** est un itérateur
- La fonction ***list()*** en est un autre, qui créé des listes à partir d'itérables

```
>>> print(range(20))
range(0, 20)
>>>
```

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>>
```

Les instructions *break, continue* et les clauses *else* au sein des boucles

- L'instruction break, comme en C, interrompt la boucle for ou while la plus profonde
- Les boucles peuvent également disposer d'une instruction else ; celle-ci est exécutée lorsqu'une boucle se termine alors que tous ses éléments ont été traités (dans le cas d'un for) ou que la condition devient fausse (dans le cas d'un while), mais pas lorsque la boucle est interrompue par une instruction break
- L'exemple suivant, qui effectue une recherche de nombres premiers, en est une démonstration

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...             if n % 2 == 0:
...                     print(n, 'égal', x, '*', n//x)
...                     break
...     else:
...             print(n, 'est premier')
...
2 est premier
3 est premier
4 égal 2 * 2
5 est premier
6 égal 2 * 3
7 est premier
8 égal 2 * 4
9 est premier
>>>
```

Les instructions *break, continue* et les clauses *else* au sein des boucles

- L'instruction continue, également empruntée au C, fait passer la boucle à son itération suivante

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...             print("Nombre pair trouvé", num)
...             continue
...     print("Nombre trouvé", num)
...
Nombre pair trouvé 2
Nombre trouvé 3
Nombre pair trouvé 4
Nombre trouvé 5
Nombre pair trouvé 6
Nombre trouvé 7
Nombre pair trouvé 8
Nombre trouvé 9
>>>
```

L'instruction *pass*

- L'instruction pass ne fait rien
- Elle peut être utilisée lorsqu'une instruction est nécessaire pour fournir une syntaxe correcte, mais qu'aucune action ne doit être effectuée
- L'exemple suivant est une boucle infinie que l'on interrompra par Ctrl+c
- Un autre cas d'utilisation du pass est de réserver un espace en phase de développement pour une fonction ou un traitement conditionnel, vous permettant ainsi de construire votre code à un niveau plus abstrait
- L'instruction pass est alors ignorée silencieusement

```
>>> while True:
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
>>>
```

```
>>> def ma_fonction():
...     # A implémenter rapidement
...     pass
...
>>>
```

Définir des fonctions

- On peut créer une fonction qui écrit la suite de Fibonacci jusqu'à une limite imposée

```
>>> def fibo(n):
...     """Affiche la suite de fibonacci jusqu'à n."""
...     a, b = 0, 1
...     while a < n:
...             print(a, end=" ")
...             a, b = b, a+b
...     print()
... # Appel de la fonction définie juste avant
...
>>> fibo(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
>>>
```

Définir des fonctions

- Le mot-clé ***def*** introduit une définition de fonction
- Il doit être suivi du nom de la fonction et d'une liste entre parenthèses de ses paramètres
- L'instruction qui constitue le corps de la fonction débute à la ligne suivante, et doit être indentée
- La première instruction d'une fonction peut, de façon facultative, être une chaîne de caractères littérale
- Cette chaîne de caractères sera alors la chaîne de documentation de la fonction, ou ***docstring***
- Il existe des outils qui utilisent ces chaînes de documentation pour générer automatiquement une documentation en ligne ou imprimée, ou pour permettre à l'utilisateur de naviguer de façon interactive dans le code ; prenez-en l'habitude, c'est une bonne habitude que de documenter le code que vous écrivez !

Définir des fonctions

- L'exécution d'une fonction introduit une nouvelle table de symboles utilisée par les variables locales de la fonction
- Plus précisément, toutes les affectations de variables effectuées au sein d'une fonction stockent la valeur dans la table de symboles locale ; tandis que les références de variables sont recherchées dans la table de symboles locale, puis dans la table de symboles locale des fonctions englobantes, puis dans la table de symboles globale et finalement dans la table de noms des primitives
- Par conséquent, il est impossible d'affecter une valeur à une variable globale (sauf en utilisant une instruction ***global***), bien qu'elles puissent être référencée

Définir des fonctions

- Les paramètres effectifs (arguments) d'une fonction sont introduits dans la table de symboles locale de la fonction appelée lorsqu'elle est appelée
- Par conséquent, les passages de paramètres se font par valeur, la valeur étant toujours une référence à un objet, et non la valeur de l'objet lui-même.
- Lorsqu'une fonction appelle une autre fonction, une nouvelle table de symboles locale est créée pour cet appel

Définir des fonctions

- Une définition de fonction introduit le nom de la fonction dans la table de symboles courante
- La valeur du nom de la fonction est un type qui est reconnu par l'interpréteur comme une fonction utilisateur
- Cette valeur peut être affectée à un autre nom qui pourra alors être utilisé également comme une fonction
- Ceci fournit un mécanisme de renommage général

```
>>> fibo
<function fibo at 0x0000020859819AE8>
>>> f = fibo
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
>>>
```

Définir des fonctions

- Si vous venez d'autres langages, vous pouvez penser que ***fibo()*** n'est pas une fonction mais une procédure, puisqu'elle ne renvoie pas de résultat
- En fait, même les fonctions sans instruction return renvoient une valeur, quoique ennuyeuse
- Cette valeur est appelée ***None*** (c'est le nom d'une primitive)
- Écrire la valeur None est normalement supprimé par l'interpréteur lorsqu'il s'agit de la seule value écrite
- Vous pouvez le voir si vous y tenez vraiment en utilisant ***print()***

```
>>> print(fibo(0))

None
>>>
```

Définir des fonctions

- Il est facile d'écrire une fonction qui renvoie une liste de la série de Fibonacci au lieu de l'imprimer

```
>>> def fibo(n): # Retourne une suite de Fibonacci jusqu'à n
...     """Retourne une liste contenant la suite"""
...     resultat = []
...     a, b = 0, 1
...     while a < n:
...             resultat.append(a)
...             a, b = b, a+b
...     return resultat
...
>>> f100 = fibo(100) # Appel de la fonction fibo()
>>> f100             # Affichage de la liste retournée
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>>
```

Définir des fonctions

- L'instruction ***return*** provoque la sortie de la fonction en renvoyant une valeur
- return sans expression en paramètre renvoie ***None***
- Arriver à la fin d'une fonction renvoie également None
- L'instruction ***resultat.append(a)*** appelle une méthode de l'objet ***resultat*** qui est une liste
- Une méthode est une fonction qui "appartient" à un objet et qui est nommée ***obj.methodname***, où ***obj*** est un objet (il peut également s'agir d'une expression), et ***methodname*** est le nom d'une méthode définie par le type de l'objet
- Différents types définissent différentes méthodes. Des méthodes de différents types peuvent porter le même nom sans qu'il n'y ait d'ambigüité (vous pouvez définir vos propres types d'objets et leurs méthodes en utilisant des classes)
- La méthode ***append()*** donnée dans cet exemple est définie pour les listes ; elles ajoute un nouvel élément à la fin de la liste. Dans cet exemple, elle est l'équivalent de ***resultat = resultat + [a]***, mais elle est plus efficace

Fonctions : *Valeurs par défaut des arguments*

- La forme la plus utile consiste à indiquer une valeur par défaut pour certains arguments
- Ceci crée une fonction qui pourra être appelée avec moins d'arguments que ceux présents dans sa définition

```
>>> def demander_ok(prompt, tentatives=4, rappel='Essayez à nouveau !'):
...     while True:
...             ok = input(prompt)
...             if ok in ('y', 'ye', 'yes'):
...                     return True
...             if ok in ('n', 'no', 'nope'):
...                     return False
...             tentatives = tentatives - 1
...             if tentatives < 0:
...                     raise ValueError('Réponse utilisateur invalide')
...             print(rappel)
...
>>>
```

Fonctions : *Valeurs par défaut des arguments*

- Cette fonction peut être appelée de plusieurs façons
  - en ne fournissant que les arguments obligatoires
    - *demander_ok('Voulez-vous vraiment quitter ?')*
  - en fournissant une partie des arguments facultatifs :
    - *demander_ok('OK pour écraser le fichier ?', 2)*
  - en fournissant tous les arguments
    - *demander_ok('OK pour écraser le fichier ?', 2, 'Allez, seulement oui ou non !')*
- Cet exemple présente également le mot-clé ***in***.
- Celui-ci permet de tester si une séquence contient une certaine valeur

Fonctions : *Valeurs par défaut des arguments*

- Les valeurs par défaut sont évaluées lors de la définition de la fonction dans la portée de définition

```
>>> i = 50
>>>
>>> def f(arg=i):
...     print(arg)
...
>>> i = 60
>>> f()
50
>>>
```

Fonctions : *Valeurs par défaut des arguments*

- ***Avertissement important*** : La valeur par défaut n'est évaluée qu'une seule fois
- Ceci fait une différence lorsque cette valeur par défaut est un objet muable tel qu'une liste, un dictionnaire ou des instances de la plupart des classes
- Par exemple, la fonction suivante accumule les arguments qui lui sont passés au fil des appels successifs
- Si vous ne voulez pas que cette valeur par défaut soit partagée entre des appels successifs, vous pouvez écrire la fonction de cette façon

```
>>> def f(a, L=[]):
...     L.append(a)
...     return L
...
>>> print(f(1))
[1]
>>> print(f(2))
[1, 2]
>>> print(f(3))
[1, 2, 3]
>>>
```

```
>>> def f(a, L=None):
...     if L is None:
...         L = []
...     L.append(a)
...     return L
...
>>> print(f(1))
[1]
>>> print(f(2))
[2]
>>> print(f(3))
[3]
>>>
```

Fonctions : *Arguments nommés*

- Les fonctions peuvent également être appelées en utilisant des arguments nommés sous la forme ***kwarg=value***
- ***ma_fonction()*** accepte un argument obligatoire (***a_pos***), et trois arguments facultatifs (***a_nom_a***, ***a_nom_b*** et ***a_nom_c***)
- Cette fonction peut être appelée de n'importe laquelle de ces façons :
  - *ma_fonction(1000)*
  - *ma_fonction(a_pos=1000)*
  - *ma_fonction(a_pos=1000, a_nom_a='AAA')*
  - *ma_fonction(a_pos=1000, a_nom_a='AAA')*
  - *ma_fonction(a_nom_b='BBB', a_pos=100, a_nom_c='CCC')*

```
>>> def ma_fonction(a_pos, a_nom_a='A', a_nom_b='B', a_nom_c='C'):
...     print('a_pos vaut ', a_pos)
...     print('a_nom_a vaut ', a_nom_a)
...     print('a_nom_b vaut ', a_nom_b)
...     print('a_nom_c vaut ', a_nom_c)
...
```

Fonctions : *Arguments nommés*

- Dans un appel de fonction, les arguments nommés doivent suivre les arguments positionnés
- Tous les arguments nommés doivent correspondre à l'un des arguments acceptés par la fonction, mais leur ordre n'est pas important
- Ceci inclut également les arguments facultatifs
- Aucun argument ne peut recevoir une valeur plus d'une fois, comme l'illustre cet exemple incorrect du fait de cette restriction

```
>>> def fonction(a):
...     pass
...
>>> fonction(0,a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fonction() got multiple values for argument 'a'
>>>
```

Fonctions : *Arguments nommés*

- Quand un dernier paramètre formel est présent sous la forme ****name***, il reçoit un dictionnaire contenant tous les arguments nommés à l'exception de ceux correspondant à un paramètre formel
- Ceci peut être combiné à un paramètre formel sous la forme *name qui lui reçoit un tuple contenant les arguments positionnés au-delà de la liste des paramètres formels (****name*** doit être présent avant *****name***)
- Il est garanti que l'ordre d'affichage des arguments est le même que l'ordre dans lesquels ils sont fournis lors de l'appel à la fonction

```
>>> def fromagerie(fromage, *arguments, **mots_cles):
...     print("-- Avez vous du", fromage, "?")
...     print("-- Désolé nous n'avons plus de", fromage)
...     for arg in arguments:
...         print(arg)
...     print("-" * 40)
...     for kw in mots_cles:
...         print(kw, ":", mots_cles[kw])
...
>>>
>>> fromagerie("Camembert", "Il est très demandé.",
...            "Il est vraiment très, très demandé.",
...            fromager="Thierry DECKER",
...            client="Raymond Depardon",
...            sketch="Sketch du fromager")
-- Avez vous du Camembert ?
-- Désolé nous n'avons plus de Camembert
Il est très demandé.
Il est vraiment très, très demandé.
----------------------------------------
fromager : Thierry DECKER
client : Raymond Depardon
sketch : Sketch du fromager
>>>
```

Fonctions : *Liste d'arguments arbitraires*

- Pour terminer, l'option la moins fréquente consiste à indiquer qu'une fonction peut être appelée avec un nombre arbitraire d'arguments
- Ces arguments sont intégrés dans un tuple
- Avant le nombre variable d'arguments, zéro arguments normaux ou plus peuvent apparaître
- Normalement, ces arguments variadiques sont les derniers paramètres, parce qu'ils agrègent toutes les valeurs suivantes. Tout paramètre placé après le paramètre ***args** ne pourra être utilisé que par son nom

```
>>> def ma_fonctions(*args):
...     total = 0
...     for nbr in args:
...         total += nbr
...     return total
...
>>>
>>> print(ma_fonctions(10, 20, 30))
60
>>> print(ma_fonctions(1, 2, 3, 4, 5))
15
>>>
```

Fonctions : *Séparation des listes d'arguments*

- La situation inverse intervient lorsque les arguments sont déjà dans une liste ou un tuple mais doivent être séparés pour un appel de fonction nécessitant des arguments positionnés séparés
- Par exemple, la primitive range() attend des arguments ***start*** et ***stop*** distincts
- S'ils ne sont pas disponibles séparément, écrivez l'appel de fonction en utilisant l'opérateur * pour séparer les arguments présents dans une liste ou un tuple
- De la même façon, les dictionnaires peuvent fournir des arguments nommés en utilisant l'opérateur **

```
>>> list(range(3, 6))
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))
[3, 4, 5]
>>>
```

```
>>> def ma_fonction(dividende, diviseur):
...     return dividende / diviseur
...
>>> kwargs = {"diviseur":12, "dividende": 144}
>>> print("La division donne", ma_fonction(**kwargs))
La division donne 12.0
>>>
```

Fonctions : *Fonctions anonymes*

- Avec le mot-clé ***lambda***, on peut créer de petites fonctions anonymes
- Voilà une fonction qui renvoie la somme de ses deux arguments : lambda a, b: a+b
- Les fonctions lambda peuvent être utilisées partout où un objet fonction est attendu
- Elles sont syntaxiquement restreintes à une seule expression
- Sémantiquement, elles ne sont qu'une raccourci syntaxique pour une définition de fonction normale
- Comme les fonctions imbriquées, les fonctions lambda peuvent référencer des variables de la portée englobante

```
>>> def incrementeur(n):
...     return lambda x: x + n
...
>>> f = incrementeur(42)
>>> f(0)
42
>>> f(2)
44
>>> f
<function incrementeur.<locals>.<lambda> at 0x00000202B1889BF8>
>>>
```

Fonctions : *Fonctions anonymes*

- L'exemple précédent utilise une fonction anonyme pour renvoyer une fonction
- Un autre usage typique est de donner une fonction minimaliste directement en temps que paramètre

```
>>> pairs = [(1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre')]
>>> pairs
[(1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(2, 'deux'), (4, 'quatre'), (3, 'trois'), (1, 'un')]
>>>
```

Chaines de documentation *(docstrings)*

- Voici quelques conventions concernant le contenu et le format des chaînes de documentation
- La première ligne devrait toujours être courte, un résumé concis de l'utilité de l'objet
- Pour être bref, nul besoin de rappeler le nom de l'objet ou son type, qui sont accessibles par d'autres moyens (sauf si le nom est un verbe qui décrit une opération)
- Cette ligne devrait commencer avec une majuscule et se terminer par un point
- Si il a d'autres lignes dans la chaîne de documentation, la seconde ligne devrait être vide, pour la séparer visuellement du reste de la description
- Les autres lignes peuvent alors constituer un ou plusieurs paragraphes décrivant le mode d'utilisation de l'objet, ses effets de bord, etc.

Chaines de documentation *(docstrings)*

- L'analyseur de code Python ne supprime pas l'indentation des chaînes de caractères littérales multi-lignes, donc les outils qui utilisent la documentation doivent si besoin faire cette opération eux-mêmes
- La convention suivante s'applique : la première ligne non vide après la première détermine la profondeur d'indentation de l'ensemble de la chaîne de documentation (on ne peut pas utiliser la première ligne qui est généralement accolée aux guillemets d'ouverture de la chaîne de caractères et dont l'indentation n'est donc pas visible)
- Les espaces « correspondant » à cette profondeur d'indentation sont alors supprimés du début de chacune des lignes de la chaîne
- Aucune ligne ne devrait présenter un niveau d'indentation inférieur mais si cela arrive, tous les espaces situés en début de ligne doivent être supprimés
- L'équivalent des espaces doit être testé après expansion des tabulations (normalement remplacés par 4 espaces)

Chaines de documentation *(docstrings)*

- Voici un exemple de chaîne de documentation multi-lignes

```
>>>
>>> def ma_fonction():
...     """Ne fait rien, mais est documenté
...
...     Non, cette fonction ne fait vraiment rien
...     """
...     pass
...
>>> print(ma_fonction.__doc__)
Ne fait rien, mais est documenté

        Non, cette fonction ne fait vraiment rien

>>>
```

Annotation des fonctions

- Les annotations sont des métadonnée optionnelles décrivant les types utilisées par une fonction définie par l'utilisateur (Voir la PEP 484 pour plus d'informations)
- Les annotations sont stockées dans l'attribut __annotations__ de la fonction, sous forme d'un dictionnaire, et n'ont aucun autre effet
- Les annotations sur les paramètres sont définis par deux points (:) après le nom du paramètre suivi d'une expression donnant la valeur de l'annotation
- Les annotations de retour sont définies par -> suivi d'une expression, entre la liste des paramètres et les deux points de fin de l'instruction def
- L'exemple suivant a un paramètre positionnel, un paramètre nommé, et une valeur de retour annotée

Annotation des fonctions

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>>
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'eggs': <class 'str'>, 'return': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
>>>
```

Style de codage

- Maintenant que vous êtes prêt à écrire des programmes plus longs et plus complexes, il est temps de parler du style de codage
- La plupart des langages peuvent être écrits (ou plutôt formatés) selon différents styles
- certains sont plus lisibles que d'autres
- Rendre la lecture de votre code plus facile aux autres est toujours une bonne idée, et adopter un bon style de codage peut énormément vous y aider
- En Python, la ***PEP 8*** a émergé comme étant un guide auquel la plupart des projets adhèrent ; elle met en avant un style de codage très lisible et agréable à l'œil
- Chaque développeur Python devrait donc la lire et s'en inspirer autant que possible

Style de codage

- Utilisez des indentations de 4 espaces, et pas de tabulation.
- 4 espaces constituent un bon compromis entre une indentation courte (qui permet une profondeur d'imbrication plus importante) et une longue (qui rend le code plus facile à lire). Les tabulations introduisent de la confusion, et doivent être proscrites autant que possible.
- Faites des retours à la ligne, de telle sorte qu'elles n'excèdent pas 79 caractères.
- Ceci aide les utilisateurs ne disposant que de petits écrans, et permet sur de plus grands de disposer plusieurs fichiers côte à côte sans difficulté.
- Utilisez des lignes vides pour séparer les fonctions et les classes, ou pour scinder de gros blocs de code à l'intérieur de fonctions.
- Lorsque c'est possible, placez les commentaires sur leur propres lignes.
- Utilisez les chaînes de documentation

Style de codage

- Utilisez des espaces autour des opérateurs et après les virgules, mais pas directement à l'intérieur des parenthèses : a = f(1, 2) + g(3, 4).
- Nommez toujours vos classes et fonctions de la même manière ; la convention est d'utiliser une notation ***CamelCase*** pour les classes, et ***minuscules_avec_trait_bas*** pour les fonctions, méthodes et variables. Utilisez toujours ***self*** comme nom du premier argument des méthodes (voyez une première approche des classes pour en savoir plus sur les classes et les méthodes).
- N'utilisez pas d'encodages exotiques dès lors que votre code est sensé être utilisé dans des environnements internationaux. Par défaut, Python travaille en UTF-8, ou sinon du simple ASCII fonctionne dans la plupart des cas.
- De la même manière, n'utilisez que des caractères ASCII pour vos noms de variables si vous soupçonnez qu'un personne parlant une autre langue lira ou devra modifier votre code.

Style de codage

- Le document PEP 257
- Google Python Style Guide

Références

- Python.org : https://www.python.org/
- Learning Python : https://github.com/thierrydecker/learning-python
- …

Outils

- IDE Pycharm Community : https://www.jetbrains.com/pycharm/
- Analyse en ligne de code Python : http://www.pythontutor.com/
- …