Patrick Nsukami

3/4/23

Contents

Introdution:

Python est un langage de programmation interprété, orienté objet et au typage dynamique. Python est un langage libre et open source, créé par Guido van Rossum. La première version, la 0.9.0, de Python est sortie en 1991.

Pourquoi?

1. Libre et open source

Python est libre et open source. Nous sommes libres d'utiliser, de copier, d'étudier, de modifier le code source de ce langage. Le code source a été ouvert de manière à encourager l'amélioration du langage par toute personne le souhaitant.

2. Typage dynamique:

Python est un langage dynamiquement typé[^1]. Il n'est pas obligatoire de préciser les types des objets que vous définssez. Et pendant le cycle de vie de votre programme, une variable peut posséder des valeurs de différents types.

4. Multi-paradigme

Python supporte la programmation impérative, fonctionnelle, orientée objet.

5. Un écosystème large:

Depuis un an maintenant, le langage occupe la première place de l'index Tiobe index. La communauté offre des milliers de bibliothèques nous permettant de développer n'importe quel genre d'application. Pour en savoir plus, merci de consulter ce lien

Configuration et installation

Pour les utilisateurs de Windows:

Si vous êtes sous Windows, le meilleur moyen d'installer Python est de:

- Se rendre sur le site web officiel de Python.
- Aller sur la page de téléchargement ici.
- Télécharger la version la plus récente du langage et adaptée à votre système.

Pour les utilisateurs de MaOS:

Si vous êtes sous MacOS, le meilleur moyen d'installer Python est de:

- Se rendre sur cette page
- Télécharger puis installer la version adaptée à votre système
- Sinon, consulter et suivre la démarche expliquée ici

Pour les utilisateurs de GNU/Linux:

Sur la plupart des systèmes de type Linux, Python est installé par défaut. Si ce n'est pas le cas:

- Se rendre sur cette page:
- Télécharger puis installer la version adaptée à votre système

Si l'installation s'est déroulée correctement, nous devrions être capables de démarrer une console interactive et voir un message d'accueil à peu égal à ce qui suit:

```
>> python3
Python 3.10.0 (default, Oct 21 2021, 16:51:22) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

S'il n'est pas possible d'installer Python sur votre ordinateur, voici une liste de liens qui vous permettrons d'avoir un interpreteur Python dans votre navigateur:

- Python
- Sympy
- PythonAnywhere
- Colab
- Pythonfiddle
- Trinket
- Replit

Bien qu'il soit tout à fait possible de suivre ce cours en utilisant n'importe quel système d'exploitation, nous vous recommendons vivement de faire votre apprentissage en utilisant un système d'exploitation de type GNU/Linux.

Il y aura des commandes à exécuter en mode texte, il y aura des fichiers de configuration à éditer. Mais surtout, les logiciels que nous développerons, les modèles que nous construirons, les scripts que nous écrirons, tous ces programmes s'exécuteront *très probablement* sur des serveurs de type GNU/Linux.

Environnement de développement:

L'apprentissage se fera dans un premier temps en utilisant Idle. Idle est un environnement facilitant notre découverte du langage. Lorsque nous serons un peu plus à l'aise avec l'écriture de code, nous travaillerons avec des outils beaucoup plus robustes tels que:

- IPython
- Spyder
- PyCharm
- VSCode
- Jupyter, JupyterLab

Pour les plus valeureux et les plus courageux:

- Emacs
- Vi

NB: Les environnements de développement, il y en a plusieurs, chacun avec ses forces et ses faiblesses. Testez en quelques un, puis choisissez celui avec lequel vous vous sentez le plus à l'aise et le plus productif.

Les opérateurs:

Les opérateurs sont des symbôles désignant un certain traitement, par exemple, le '+' pour additionner, le '*' pour multiplier. Ces opérateurs sont divisés en catégories:

Les opérateurs arithmétiques:

Pour effectuer des opérations mathématiques sur des nombres entiers ou des nombres réels:

```
+     Addition         x + y
-     Soustraction     x - y
*     Multiplication   x * y
/     Division         x / y
%     Modulo           x % y
**    Puissance        x ** y
//    Division entière x // y
```

Les opérateurs d'assignation:

Ces opérateurs sont utilisés pour "affecter", "assigner", une valeur à une variable:

```
=     x = 5           équivant à: x = 5
+=    x += 3          équivant à: x = x + 3
-=    x -= 3          équivant à: x = x - 3
*=    x *= 3          équivant à: x = x * 3
/=    x /= 3          équivant à: x = x / 3
%=    x %= 3          équivant à: x = x % 3
//=   x //= 3         équivant à: x = x // 3
**=   x **= 3         équivant à: x = x ** 3
```

Les opérateurs de comparaison:

Utilisés pour effectuer des comparaisons:

```
==    Equal                       x == y
!=    Not equal                   x != y
>     Greater than                x > y
<     Less than                   x < y
>=    Greater than or equal to    x >= y
<=    Less than or equal to       x <= y
```

Les opérateurs logique:

Utiliés pour combiner des expressions booléennes:

```
and    Returns True if both statements are true                  x < 5 and  x < 10
or     Returns True if one of the statements is true             x < 5 or x < 4
not    Reverse the result, returns False if the result is true   not(x < 5 and x < 10)
```

L'opérateur identité:

L'opérateur d'identité, compare deux variables et renvoie "Vrai" si ces deux variables référencent le même objet en mémoire. Cet opérateur ne compare pas les "valeurs", il compare les "identités".

```
# is       Renvoie Vrai si le même objet est référencé en mémoire      x is y
# is not   Renvoie Vrai si les objets référencés sont différents   x is not y

In [1]: ranks = [1, 2, 3]
   ...: rates = [1, 2, 3]

In [2]:

In [2]: ranks == rates # ces deux objets ont la même valeur
Out[2]: True

In [3]: ranks is rates # ces deux objets sont différents (par l'identité)
Out[3]: False

In [4]: id(ranks) # l'id de ranks...
Out[4]: 140411803459904

In [5]: id(rates) # est différent de l'id de rates
Out[5]: 140411803460544
```

L'opérateur d'appartenance:

Opérateur utilisé pour vérifier si un objet fait partie d'un groupe d'objet:

```
In [7]: 4 in [5, 6, 7]  # Returns True if an object x is present in the object y
Out[7]: False

In [8]: "a" in "patrick"
Out[8]: True
```

Valeurs et types:

Nous pouvons considérer un programme comme étant un ensemble **“d’objets”** que l’on manipule. Pour manipuler des “objets”, nous avons besoin:

- de les nommer, en utilisant des **“variables”**.
- de leur donner, de leur assigner une **“valeur”**.
- de savoir quel est le **“type”** de la valeur assignée à l’objet.

Les **valeurs** 100, 250, “Hello World!”, sont des valeurs de différents **types**. Par exemple, 2 est un nombre **entier**, “Hello, World!” est une **chaine de caractère**. Pour determiner le type d’un objet, nous utiliserons la fonction type.

Voici la liste des types standards en Python, ce qui sont installés par défaut dans notre système (*en gras, ceux sur lesquels nous nous concentrerons*):

- Nombres: **int, float, complex**
- Booléens: **bool**
- Séquences:
  - non mutables: **String, Tuple**, Byte
  - mutables: **List**, ByteArray
- Ensembles: **Set**, Frozenset
- Tableaux associatifs: **Dictionnaire**

Nous entendrons souvent parler de ce qu’on appelle les **types primitifs**. Les types primitifs sont: **int, float, str, complex, bool**. Ces types vont nous permettre de stocker *une seule* valeur dans notre variable:

```
In [10]: name = "patrick"

In [11]: age = 45

In [12]: height = 192.3

In [13]: location = 4 + 6j

In [14]: male = True

In [15]: type(name)
Out[15]: str

In [16]: type(age)
Out[16]: int

In [17]: type(location)
Out[17]: complex

In [18]: type(male)
Out[18]: bool
```

* ***Interlude: tracé de figures géométriques avec le module Turtle.***

Le contrôle de flux:

Toutes les instructions que nous avons écrites jusqu'à présent, ce sont exécutées **une par une et dans l'ordre d'apparition**. Nous aurons parfois besoin de modifier l'**ordre d'exécution** de nos instructions (contrôler le flux d'exécution). Python dispose d'instructions nous permettant de modifier le contrôle de flux en utilisant des *boucles et/ou des conditions*.

Les structures conditionnelles:

Nous aurons parfois besoin d'exécuter une instruction, seulement si certaines conditions sont remplies.

if:

Dans sa forme la plus simple, l'instruction if s'utilise pour ***exécuter*** quelque chose, lorsqu'une certaine **condition est vérifiée**:

```
x = int(input("please type a value for x: ")
if x < 0:  # si x est inférieur à 0
    print("x is negative")  # alors afficher un message
```

if, else:

Nous serons parfois dans la situation où nous voulons exécuter quelque chose lorsqu'une condition est vérifiée et ***exécuter autre chose*** si la condition ***n'est pas vérifiée***:

```
x = int(input("please type a value for x: ")
if x < 0:  # si x est inférieur à 0
    print("x is negative")  # alors afficher un message
else: # sinon (dans le cas contraire)
    print("x is positive")  # alors, afficher un autre message
```

if, elif, else:

Pour finir, certains enbranchements offrent, non pas une, non pas deux, mais plusieurs issues possibles. Parfois, suivant la valeur prise par une variable, il y aura plus de deux alternatives possibles pour la suite du programme:

```
x = int(input("please type a value for x: ")
if x < 0:  # si x est inférieur à 0
    print("x is negative")  # alors afficher un message
elif 5 < x < 10: # si x compris entre 5 et 10
    print("x is between 5 and 10")
elif 10 < x < 15:
    print("x is between 10 and 15")
else: # si aucune des conditions précédentes n'est vérifiée, alors
    print("x is greater than 15")  # afficher 1 message pour toutes les autres valeurs de x
```

Filtrage par motif:

Le filtrage par motif est un autre moyen de contrôler le *flux d'exécution* de notre programme. Nous n'en parlerons pas ici, mais nous pouvons retenir que filtrage par motif permet de faire correspondre différents ***motifs*** à ***différentes actions***. Ce mécanisme ressemble à ce que l'on pourrait le faire avec une série de ***if...elif...else.*** Mais attention, ce mécanisme est bien plus profond qu'il ne parait. Nous recommendons la documentation officielle pour en savoir d'avantage.

Les structures itératives:

Certains problèmes devront être résolus en exécutant une instructions plusieurs fois. Le nombre de répétions peut dépendre, soit d'une expression logique, soit du nombre d'élément dans une séquence.

Les boucles for:

Les boucles for nous permettent d'exécuter une ou plusieurs instructions *plusieurs fois*. La répétition est fonction de ce qu'on appelle en Python *un iterable* (un objet sur lequel on peut faire une itération), exemple d'itérables: **list, tuple, array, range, map, string, dictionary**. La répétition se fera autant de fois qu'il y a d'éléments dans la séquence:

```
>>> lst = [3, 1, 4, 1, 5]
>>> for e in lst: # parcours d'une liste
...     print(e) # affichage
```

```
>>> lst = [3, 1, 4, 1, 5]
>>> for i, e in enumerate(lst):  # parcours en récupérant index et élément
...     print(i, e)
```

```
>>> msg = 'Hello!'
>>> for k in msg:  # parcours d'une chaine de caractère
...     print(k)
```

```
>>> for k in range(3, 7):  # parcours d'un objet range
...     print(k)
```

```
>>> d = {4: 'foo', 'Bar': 3.14, (20, 19): [20, 23]}
>>> for k, v in d.items():  # parcours d'un dictionnaire (clé et valeur)
...     print('k: {}, v: {}'.format(k, v))
```

Les boucles while:

Les boucles while, comme les boucles for, permettent la répétition d'instructions. la répétion s'effectuera tant qu'une certaine condition sera vérifiée. Le corps de la boucle doit contenir une instruction modifiant la condition. Si ce n'est pas le cas, la boucle s'exécutera à l'infini. Exemple:

```
number = 23
running = True

while running:
    guess = int(input('Please, enter an integer : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        # instruction permettant à la boucle de s'arrêter éventuellement
        running = False
    elif guess < number:
```

```
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
```

Les expressions: break, continue, else

Lorsque l'on met en place une instruction itérative, nous avons la possibilité de:

- ***l'interrompre***, en utilisant un ***break***
- lui demander de ***passer à l'itération suivante***, en utilisant un ***continue***.
- personnaliser sa ***terminaison*** en cas de non interruption avec une ***clause else***.

Pour en savoir plus, voir ici.

Les fonctions:

En programmation, une fonction est un ensemble d'instructions effectuant une certaine tâche. Aulieu de réécrire les mêmes instructions encore et encore, on commence par définir une fonction. Une fois la fonction définie, on pourra l'exécuter autant de fois que nécessaire sans avoir à réécrire les instructions réalisant la tâche voulue.

Pour effecuter sa tâche, une fonction peut *potentiellement* avoir besoin de recevoir une ou plusieurs données en *entrée*. Lorsque la fonction a terminé de s'exécuter, elle peut *potentiellement* retourner une ou plusieurs valeurs.

Les fonctions prédéfinies:

Python nous offre plusieurs fonctions dites prédéfinies. C'est à dire, elles ont été définies par les développeurs du langage, nous nous contenterons de les utiliser. Les 4 fonctions que nous devons absolument bien connaitre, car nous les utiliserons souvent sont:

- help, pour consulter la documentation
- type, pour retrouver le type d'un objet
- dir, pour lister les attributs d'un objet
- print, pour afficher sur la sortie standard

Les fonctions définies par l'utilisateur:

* ***Voir le cours sur les fonctions.***

Les structures de données:

Les structure de données sont le moyen pour nous de regrouper, de manipuler, d'organiser un ensemble de données. Le tout afin de faciliter l'utilisation ou bien le traitement de ces données. Le tout afin d'écrire des programmes beaucoup plus interessants.

Il existe plusieurs types de structures de données. Les 4 structures de données que nous étudierons sont: **list, tuple, dictionary, set**. Pour aller un peu plus loin, nous parlerons brièvement des **deques** et des **tableaux**.

1. Les listes:

Les listes sont des collections ordonnées et modifiables d'éléments. Les éléments d'une liste peuvent être de types différents.

```
>>> arr = ["one", "two", "three"]
>>> arr[0]
'one'
>>> arr[1] = "hello"
>>> arr
['one', 'hello', 'three']
>>> del(arr[1])
>>> arr
['one', 'three']
>>> arr.append(23)
>>> arr
['one', 'three', 23]
```

2. Les tuples:

Les tuples sont des collections ordonnées et non modifiables d'éléments. Les éléments d'une liste peuvent être de types différents.

```
>>> arr = ("one", "two", "three")
>>> arr[0]
'one'
>>> arr
('one', 'two', 'three')
```

```
>>>
>>> arr[1] = "hello"  # nous ne pouvons modifier un tuple
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> del(arr[1]) # nous ne pouvons modifier un tuple
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

3. Les dictionnaires:

Les éléments d'une liste ou d'un tuple sont ordonnés. On récupère un élément grâce à son index. Les dictionnaires sont des collections qui vont nous permettre de rassembler des éléments mais ceux-ci seront identifiés par une clé. et non par un index. Les dictionnaires font partie des structures de données les plus utilisées en Python.

```
>>> car2 = {
...     "color": "blue",
...     "mileage": 40231,
...     "automatic": False,
... }
>>> car2
{'color': 'blue', 'automatic': False, 'mileage': 40231}
>>>
>>> car2["mileage"] # récupération d'une valeur
40231
>>>
>>> car2["mileage"] = 12  # modification du dictionnaire
>>> car2["windshield"] = "broken"
>>> car2
{'windshield': 'broken', 'color': 'blue',
 'automatic': False, 'mileage': 12}
```

4. Les sets:

Les sets sont des ensembles de données non ordonnées et non dupliquées. Les sets s'utilisent pour effectuer des opérations mathématiques telles que: **union, intersection, difference, etc...**

```
>>> vowels = {"a", "e", "i", "o", "u"}
>>> "e" in vowels
True
>>> letters = set("alice")
>>> letters.intersection(vowels)
{'a', 'e', 'i'}
>>> vowels.add("x")
>>> vowels
{'i', 'a', 'u', 'o', 'x', 'e'}
>>> len(vowels)
6
```

5. Les deques:

Les deques se comportent comme les objets de type séquence en Python, c'est à dire, tout ce que nous avons appris avec les listes *(indexing, slicing, concatenation, multiplication)* reste valable pour les deques.

La particularité des deques:

- Permettent l'ajout ou le retrait à partir de la tête
- Permettent de fixer la taille maximale de la séquence
- Consomment plus d'espace mémoire que les listes

```
In [2]: from collections import deque
In [3]: d1 = deque()

In [4]: type(d1)
Out[4]: collections.deque

In [6]: d1.append(4)

In [7]: print(d1)
deque([4])

In [8]: d1.extend([1, 2, 3])

In [9]: print(d1)
deque([4, 1, 2, 3])

In [10]: d1.popleft()  # retrait par la tête
Out[10]: 4

In [11]: d1.appendleft(6)  # ajout par la tête

In [12]: print(d1)
deque([6, 1, 2, 3])

In [14]: d1.extendleft([7, 8, 9])  # ajout par la tête

In [15]: print(d1)
deque([9, 8, 7, 6, 1, 2, 3])
```

6. Les tableaux:

Les objets de type tableau se comportent comme les objets de type séquence en Python, c'est à dire, tout ce que nous avons appris avec les listes *(indexing, slicing, concatenation, multiplication)* reste valable pour les tableaux.

La particularité des tableaux:

- Tous les éléments doivent avoir le même type
- Consomment moins d'espace mémoire que les listes

```
In [20]: from array import array

In [22]: a1 = array('d')  # création d'un tableau contenant des float

In [23]: type(a1)
```

```
Out[23]: array.array

In [24]: a1.append(1)

In [25]: print(a1)
array('d', [1.0])

In [27]: a1.extend([3, 4, 5])

In [28]: print(a1)
array('d', [1.0, 3.0, 4.0, 5.0])

In [29]: a1.append("foo")  # erreur si ajout d'un élément n'étant pas un float
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [29], line 1
----> 1 a1.append("foo")

TypeError: must be real number, not str

In [36]: print(a1)
array('d', [1.0, 3.0, 4.0, 5.0])

In [37]: a1.pop()
Out[37]: 5.0

In [38]: print(a1)
array('d', [1.0, 3.0, 4.0])

In [39]: a1.insert(1, 3.14)

In [40]: a1.pop(0)
Out[40]: 1.0

In [41]: print(a1)
array('d', [3.14, 3.0, 4.0])

In [42]: a1[:1]
Out[42]: array('d', [3.14])

In [43]: a1[1:]
Out[43]: array('d', [3.0, 4.0])
```

Pour aller plus loin:

En espérant que ce cours vous aura appris quelque chose, retenez que le sujet est vaste et qu'il y a encore beaucoup à découvrir. Pour en apprendre d'avantage, nous vous recommendons vivement de consulter les ressources suivantes.

À lire:

- Kaggle: intro to programming
- Kaggle: Python
- The official Python tutorial

À regarder:

- Variables et types
- Types numériques
- Strings 1
- Strings 2
- Conditions
- Conditions 2
- Boucles et fonctions
- Boucle while
- Sequences, index, slices
- Listes
- Tuples
- Dictionaires
- Sets

"On ne peut pas peindre du blanc sur du blanc, du noir sur du noir..." Proverbe