Le langage Python

Quelques références parmi d'autres :

https://docs.python.org/3/reference/index.html

https://docs.python.org/3/library/index.html

https://docs.python.org/fr/3/tutorial/index.html

https://courspython.com/introduction-python.html

https://jakevdp.github.io/WhirlwindTourOfPython/index.html

https://github.com/OrkoHunter/python-easter-eggs

1. La philosophie zen de Python

Dans Thonny : **import this**

2. Objets et variablespi

Quelques particularités de Python :

En Python, tout est objet, y compris les variables.

Python utilise le typage dynamique : c'est la valeur qui donne son type à la variable et non la variable qui impose le type de la valeur. En conséquence, on ne doit pas déclarer le type de la variable puisque celle-ci est automatiquement du type de l'objet stocké en mémoire. Pour le suivi des différentes variables utilisées dans un programme, Python emploie un mécanisme qui associe étiquette (la variable), la valeur stockée et l'emplacement mémoire. La variable Python n'est donc pas un contenant comme le sont les variables C ou les primitives Java.

Si une variable est un objet, cela signifie qu'elle dispose d'attributs et de méthodes. Comment donc savoir de quelles méthodes on dispose pour cette variable ? En Python, les méthodes associées à un objet dépendent de son type. Si différents types implémentent des méthodes spécifiques, certaines fonctionnalités sont communes à de nombreux types. Par exemple, la fonction len() peut être utilisée avec différents types comme les Strings, les listes, les tuples, les sets, les objets de classes définies par l'utilisateur, etc. Pour qu'un objet de n'importe quel type puisse utiliser la fonction len(), il lui suffit de la prévoir explicitement dans sa définition[1]. On a ainsi non seulement une illustration du

[1] Sous la forme __len__(..)

polymorphisme, mais aussi d'un concept propre à Python, le duck typing[2] : pour calculer la longueur d'un objet à l'aide de la fonction len(), Python vérifie uniquement l'existence de la méthode dans la classe de l'objet, et non le type de cet objet. Si la méthode est implémentée, alors Python exécute correctement le calcul de longueur; dans le cas contraire, il lève une exception TypeError.

Gestion de la mémoire : en tant qu'objet, la donnée créée est accompagnée de métadonnées telle que la référence à son type ou un compteur des références d'autres objets vers lui-même : un ramasse-miettes (garbage collector) se charge de libérer la mémoire de la plupart des objets qui ne sont plus référencés.

Optimisation de la mémoire

Lorsque deux variables sont associées à la même valeur, Python ne crée pas d'objets distincts mais fait pointer les deux variables vers cette valeur.

Exemple : a et b sont toutes deux associées à la valeur 20; elles pointent donc vers la même adresse mémoire.

```
a = 20
b = 20
print (a is b) # true
```

Les méthodes des objets

Pour connaître les méthodes accessibles à un type, utiliser la fonction built-in **dir** :

```
print (dir(str))
"""
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
"""
```

[2] *if it looks like a duck, swims like a duck and quacks like a duck, it's a duck*

NB. """ … """ commentent les lignes entourées.

Exemples de manipulation de variables :

```
A = 10
```

La variable A référence le type entier puisqu'elle référence un objet entier.

```
A = "chaîne"
```

La variable A référence le type chaîne. Elle ne référence plus un objet entier, mais un objet String.

On peut vérifier le type courant d'une variable par les méthodes built-in type(<variable>) ou isinstance(<variable>,<type>).

```
print (type(A))                 # <class 'int'>  ou  <class 'str'>
print (isinstance(a,str))       # True
```

3. Quelques opérateurs

Les particularités :

- Quotient entier : //

  ```
  print ( 9 // 2)          # 4
  ```

- Puissance : **

  ```
  print ( 5 ** 3 )         # 125
  ```

- Affectation – opération (comme en C)

  ```
  A = 3
  A +=10
  print ( A )              # 13
  ```

  N.B. - L'opérateur d'in- ou décrémentation -- et ++ n'existent pas → utiliser **A += 1**

  - on peut additionner des listes de valeurs avec +=

4. Afficher et lire

Pour afficher : print

Formatage simple avec la f-string : les variables python sont notées entre accolades

```
d = 7
print (f"d a pour valeur {d}")        # d a pour valeur 7
```

Affichage par concaténation, pour produire une String :

```
a = 4
b = 10
somme = a + b
print("Somme : ",a," + ", b, "= ",somme)
```

Pour lire : input

```
cp = input ("Quel est votre code postal ? ")
print ( cp )          # 7000
```

NB. Input lit des strings. Il est nécessaire de convertir la donnée si on la destine à un usage numérique:

```
age = input ("Quel est votre âge ? ")
print(f"Dans 10 ans, vous aurez {int(age) + 10} ans ")
```

5. Les séquences

Il s'agit d'ensembles de données homogènes ou non, indicées à partir de 0. Certaines séquences sont mutables (modifiables "en place[3]") ou immuables.

Séquences mutables	Séquences immuables
Les listes	Les chaînes Les tuples

[3] "en place" (in place) : dans l'exact emplacement mémoire d'origine

5.1. Fonctions communes à toutes les séquences

```
print(dir(list))
"""
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append',
'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
"""
```

5.2. Les slices de séquences

Le slice, découpage, peut s'utiliser, par exemple, dans une boucle.

[indice1 : indice2] : depuis indice1 inclus jusqu'à indice2 exclu (pour autant que les indices existent) :

```
seq = "Le langage Python"
print(seq[2:10])            # langage
```

[: indice2] : depuis l'indice 0 inclus jusqu'à indice2 exclus

```
seq = "Le langage Python"
print(seq[:10])             # Le langage
```

[indice1 :] : depuis indice1 inclus jusqu'à la fin

```
seq = "Le langage Python"
print(seq[11:])
```

[: - indice2] : jusqu'à l'indice2 exclu, à rebours

```
seq = "Le langage Python"
print(seq[:-7])             # Le langage
```

[-indice1 :] : jusqu'à indice1 inclus, à rebours

```
seq = "Le langage Python"
print(seq[-7:])              # python
```

4.3. Les chaînes

Les chaînes sont des types immuables, c'est-à-dire non modifiables en place. Si la chaîne référence une nouvelle valeur, elle entraîne la création d'un nouvel objet situé à un emplacement mémoire différent :

```
ch = "Langage"
print(ch)
print("Adresse de la chaîne ch : ",id(ch))          # 71216736
ch = "Python"
print(ch)
print("Adresse de la chaîne ch :  ",id(ch))         # 25817056
```

➔ Renseignez-vous sur l'usage de quelques fonctions prédéfinies et testez-les dans un programme

➔ Tester une chaîne à partir d'une expression régulière

```
import re          # re = regular expression

regex = "[0-9]*"
tester = "123456"
match = re.match(regex,tester)
print(match)       # match retourne le match ou rien
```

4.4. Les listes [ ]

Les listes sont mutables, c'est-à-dire qu'on peut les modifier "en place" sans entraîner la création d'un nouvel objet.

```
liste2 = [10,5,2,7,3,9,6,4,2,1]
print(liste2)                # [10, 5, 2, 7, 3, 9, 6, 4, 2, 1]
liste2.sort()
print(liste2)                # [1, 2, 2, 3, 4, 5, 6, 7, 9, 10]
```

4.5. Les tuples ( )

Les tuples sont des ensembles de données immuables. On choisira ce type de séquence lorsque les données ne devront pas être modifiées.

Dans la création d'un tuple, les parenthèses sont facultatives :

```
tuple1 = 5,'abc','z'
print(tuple1)                # (5, 'abc', 'z')
tuple2 = (5,'abc','z')
print(tuple2)                # (5, 'abc', 'z')
```

5. Les conditions

5.1. if

```
liste = [1,7,3,4,5,6]
if liste[1] == 2:
    print("2")
elif liste[1] is 1:
    print("1");
else:
    print("ni 1, ni 2")
```

5.2. match …case

```
a = 5
match a:
    case 1:
        print("1")
    case 2:
        print("2")
```

```
    case 3:
        print("3")
    case 4:
        print("4")
    case other:
        print("Aucun nombre de 1 à 4")
```

5.3. Test d'appartenance

Le test d'appartenance retourne True ou False :

```
print('an' in 'banane')                # True
```

```
if 'an' in 'banane':
    print("Le motif se trouve dans la chaîne")
else:
    print("Le motif n'a pas été trouvé")
```

```
liste_noms = ["France", "Belgique", "Suisse", "Espagne"]
print('Autriche' in liste_noms)              # False
```

6. Les boucles

Boucle while

```
i = 0
som = 0
while(som < 10):
    i = i + 3
    som += i

print("Som = ",som)
```

Boucle for

```
for i in range(10):
    print (i,end=" ")                    # 0 1 2 3 4 5 6 7 8 9
```

```
for i in range(1,10):
    print (i)                            # (affichage vertical) 1 2 3 4 5 6 7 8 9 10
```

```
liste = [1,2,3,7,8,9]
for i in liste:
    print(i ** 3, end = " ")             # (affichage horizontal) 1 8 27 343 512 729 1728

for i in liste:
    if i % 2:
        print(i, end=" ")                # 1 3 7 9
```

7. Compréhension de listes

Il s'agit de constitution d'une liste par **filtrage** d'une autre liste.

A la manière du C / Java

```
liste2=list()          # ou liste2 = []
print("Comme en C : ")
for i in liste:
    if i < 14:
        liste2.append(i)

print(liste2)
```

La même chose, de manière pythonique → à privilégier

```
liste = [5,17,9,3,23,12,19,18,26,6]
liste2 = [i for i in liste if i < 14]
print(liste2)
```

```
liste3 = [i ** 2 for i in liste if i < 10]
print(liste3)
```

8. Les dictionnaires et les ensembles

Ces deux types utilisent des tables de hash. Les dictionnaires et les ensembles sont constitués de paires clés – valeurs.
En Python, le hash fonctionne avec un tableau de hash et une fonction de hash. Cette dernière calcule un indice à partir de la clé et place la valeur à cet indice dans le tableau. Si l'indice est déjà occupé, l'indice suivant est sélectionné. Pour récupérer la valeur, la fonction recalcule l'indice puis s'y rend directement pour récupérer la valeur associée. L'efficacité de manipulation des données ne dépend que de la vitesse de travail de la fonction, et non plus du volume de données.

Les dictionnaires et les ensembles sont particulièrement indiqués pour les données auxquelles on souhaite accéder rapidement et pour lesquelles l'ordre est sans importance.

8.1. Les dictionnaires { }

Seules les valeurs sont mutables, non les clés.

```
dict1 = {'a':12, 'g':8, 'c':14}
print(dict1)              # {'a': 12, 'g': 8, 'c': 14}
print(dict1['g'])         # 8
dict1['g'] = 10           # sera modifié
```

Compréhension de dictionnaire (usage moins fréquent) :

```
dict2 = {x for x in dict1 if x == 'a' or x == 'c'}
print(dict2)              # {'a', 'c'}
```

8.2. Les ensembles : set {}

Les ensembles servent à stocker des clés uniques (sans valeurs associées). Il ne peut y avoir de doublon dans un ensemble.

```
s1 = {'a','h','c','d','k'}
```

```
s2 = {'l','d','b','h','p','s'}
print(s1 & s2)          # éléments communs → {'h', 'd'}
print(s1 - s2)          # sans les éléments communs → {'k', 'c', 'a'}
print(s1 | s2)          # Réunion → {'l', 'c', 'a', 'h', 'p', 's', 'k', 'b', 'd'}
```

9. Programmation objet

Voici une classe type

```
class Exemple_classe:
    premier = ''                                                    (0) et (1)

    def __init__(self,parametre):  # initialisation                 (2)
        self.premier = parametre
        self.deuxieme = None  # Initialisé à None ou à une valeur par défaut
        self.troisieme = [1, 2, 3, 4, 5, 6]
        self.quatrieme = []  # Initialisé à une liste vide ou autre valeur appropriée

    def methode_1(self):
        print (f"Le paramètre reçu est {self.premier}")

    def methode_2(self,nouvelle_valeur):
        self.premier = nouvelle_valeur
        self.deuxieme = "Bob"                                       (3)

    def methode_3(self):
        return self.deuxieme                                        (4)

    def __repr__(self):                                             (5)
        return f"premier = {self.premier} et deuxieme = {self.deuxieme}"

    def __len__(self):
        return len(self.quatrieme)
```

```
exemple_objet = Exemple_classe('Marcel')
exemple_objet.methode_1()
exemple_objet.methode_2('Louise')
print("Le nouveau nom est ", exemple_objet.methode_3())
print("appel meth 4 : ",exemple_objet.methode_4() )
print(exemple_objet)
print("Autrement : ",exemple_objet.__repr__())
print(len(exemple_objet))
print ("Autrement : ",exemple_objet.__len__())
```

(0) *self* : ce mot désigne l'objet courant. Il doit accompagner chaque utilisation d'un attribut de la classe et est le premier paramètre obligatoire de chaque méthode

(1) (3) et (4) Les déclarations globales sont facultatives. Une déclaration d'attribut peut être effectuée dans une classe; <u>après sa création avec self</u>, sa portée devient globale

(2) __init__(self, …) : Cette méthode est appelée lors de l'instanciation de la classe. Elle agit *après la création* de l'objet et son allocation en mémoire. La preuve en est qu'elle reçoit l'objet courant en paramètre.
Le constructeur __new__(cls), dont le paramètre est la classe (cls) est quant à lui implicite dans l'exemple ci-dessus. Cette méthode construit l'objet (lui alloue de la mémoire) et retourne celui-ci.

(5) La méthode __repr__ retourne une chaîne des attributs de la classe (cf toString() en java).

Associations – Compositions

Exemple de classe utilisant un objet d'une autre classe en attribut :

Source : ChatGPT, modèle de langage développé par OpenAI

```
class Car:
    def __init__(self, model, color):
        self.model = model
        self.color = color
        self.engine = Engine()                    # instanciation de la classe Engine
```

```
    def start(self):
        self.engine.start()

class Engine:
    def start(self):
        print("Engine started.")

my_car = Car("Tesla", "red")
my_car.start() # Output: Engine started.
```

L'objet de type Engine aurait également pu être communiqué en paramètre lors de l'instanciation de la classe Car.