Introduction à l'algorithmique et à la programmation Python 3

Table des matières

Introduction

Programmer, c'est réaliser des programmes informatiques. Les programmes sont omniprésents sur l'ordinateur, et remplissent beaucoup de fonctions : jeux vidéo, logiciel de traitement de texte, .. L'ordinateur fonctionne avec un langage binaire. Pour qu'il comprenne ce que l'on souhaite faire, il est nécessaire d'écrire le programme avec un langage de programmation choisi. Ce langage sera alors traduit, donnant un programme en binaire compréhensible par l'ordinateur.

Le langage Python est portable : il peut fonctionner sous différents systèmes d'exploitations : Linux, Windows ou Mac. De plus il existe une multiude de bibliothèques, afin d'étendre les possibilités des programmes. Dans Python, l'indentation du code est importante. La modifier peut modifier le comportement du programme.

Ce cours a pour but de rassembler succintement toutes les fonctionnalités, la syntaxe et plus globalement le fonctionnement de Python 3.

Variables

Les valeurs ont un type, qui va déterminer ce qui se passe lorsque l'on fait une opération sur ces valeurs.

Type	En Python
entier	int
flottant	float
booléen	bool
indéfini	None
chaîne de caractères	str

Transtypage

Connaître le type d'une variable : `type()`
Changer le type d'une variable : fonction du nom du type

Calculs

Opérations basiques (`+,-,*`) : le type du résultat dépend des types d'entrée (s'il y a un `float`, la sortie est un `float`)
La division (/) donne toujours un `float`
Division euclidienne : `x // y` (reste : `x % y`)
Puissance : `x ** y`

Opérations sur les booléens

`and`
`or`
`not(True) = False`
supérieur : `>`
inférieur : `<`

égalité : `==`
différence : `!=`
supérieur ou égal : `>=`
inférieur ou égal : `<=`

Comparaisons

Possibilité de comparer des `str` (par ordre alphabétique)

```
"a" < "b" // True
"m" < "o" < "y" // True
"pomme" < "poire" // False
```

Concaténation

+ pour concaténer (exemple : `'IUT' + 'Info'` → `'IUTInfo'`)
* pour répéter une chaîne plusieurs fois (exemple : `'IUT' * 3` → `'IUTIUTIUT'`)

Fonctions

```
def fonction(parametre1, parametre2):
    # code de la fonction
    return var
```

```
def fonction(*parametres):
    # parametres est un tuple avec la liste des params

def fonction(nom, prenom, *params):
    # code
```

Ensembles

Listes

```
lst = list()
lst = []
lst = [0] * 5
lst.append(x)
lst.insert(i, x)
lst.remove(x)
```

```
for i, element in enumerate(lst):
    pass
```

```
liste = range(0,6)
liste_new = [nb * nb for nb in liste]
liste_new = [0, 1, 4, 9, 16, 25]
```

Tuples

Les tuples sont des objets non modifiables.

```
mon_tuple = ()
mon_tuple = (1,)
mon_tuple = (1,2)
```

Set

Pas de doublons, pas d'ordre

```
L = [1,1,2,1]
x = set(L)
x = {1,2}
```

```
x.add(1)
x.remove(1)
x.discard(1) # pas d'erreur si 1 n'est pas dans l'ensemble
x.pop() # retire un élément et le renvoie
frozenset() : ensembles pas modifiables
```

Opérations sur les ensembles

```
s ∪ t : s | t
s Δ t : s ^ t
s ∩ t : s & t
```

```
s.update(t) # union
s.intersection_update(t) # intersection
```

Vérifier une inclusion :

```
if s <= t # si s inclus dans t
```

Dictionnaires

```
dico = {}
dico[3.5] = 1
dico = {3.5:1,-1:None}
```

```
for cle in dico.keys(): OU for cle in dico:
for valeur in dico.values():
for cle, valeur in dico.items(): #
```

```
if x in dico: # dans les clés
if x un dico.values() # dans les valeurs
```

```
liste_points = [(1,2),(3,4),…]
for x,y in liste_points:
    print("x=",x,",y=",y)
```

Manipulation de chaînes de caractères

```
>>> s = "bonjour tout le monde"
>>> s.split()
['bonjour', 'tout', 'le', 'monde']
```

join(iterable) : colle les chaînes dans iterable dans un string

```
>>> ':'.join(['ab','cd','efg'])
'ab:cd:efg"
```

Concaténation avancée

```
>>> str = "Bonjour, je m'appelle {0} {1}, j'ai {2} ans."
>>> str.format("Paul","Dupont","23")
"Bonjour, je m'appelle Paul Dupont, j'ai 23 ans."
```

Les slices

Syntaxe :

```
t[debut:fin:pas]
```

Par défaut : `debut` vaut `0`
`fin` vaut `len(t)`
`pas` vaut `1`

```
>>> str = "salut"
>>> str[:2]
"sa"
>>> str[2:]
"lut"
>>> str[2:4]
"lu
>>> str[-1]
"t"
>>> str[1:-2]
"al"
>>> str[::2]
"slt"
```

Itérables en compréhension

Syntaxe

```
gauche expression for element in iterable droite
```

Les délimiteurs `gauche` et `droite` dépendent du type utilisé :

- Ensemble : { }
- Liste : [ ]

```
E = {2 ** i for i in range(11)}
F = {i for i in range(1, 11, 2)}
```

Compréhensions conditionnelles

```
gauche expression for element in iterable if conditions droite
```

```
F = {i for i in range(11) if i % 2}
```

Compréhensions multiples

```
C = [(i,j) for i in range(5) for j in range(4)]
```

Lecture et écriture dans un fichier

Manipulation basique d'un fichier

```
fichier = open(chemin, mode)
# manipulations
fichier.close()
```

`chemin` : chemin vers le fichier à ouvrir
`mode` : décrit le mode d'utilisation du fichier

- `r` (par défaut) : lecture
- `w` : écriture : le fichier est créé s'il n'existe pas, sinon il est effacé pour pouvoir y écrire
- `a` : mode ajout

`file` est un objet modifiable. Il connaît le fichier et a une position courante, qui est modifiée au fur à mesure de la lecture ou de l'écriture.

Lecture dans un fichier

```
fichier = open('fichier.py')
ligne = None
while ligne != '':
    ligne = fichier.readline()
    print(ligne)
fichier.close()
```

Un fichier peut aussi être vu comme une structure itérable.

```
fichier = open('fichier.py')
for ligne in fichier:
    print(ligne)
fichier.close()
```

Pour retirer le dernier caractère (\n) : `ligne = ligne [:-1]`

Ecriture

```
fichier = open('fichier.py', 'w')
fichier.write('bonjour\n')
fichier.close()
```

enumerate() : numérote les éléments d'un itérable

```
mots = ["bonjour", "tout", "le", "monde"]
for position, chaine in enumerate(mots):
    print("mot", position, ":", chaine)
```

Arguments en ligne de commande

```
import sys
print("Je ne fais rien d'autre qu'afficher mes paramètres:\n")
for position, parametre in enumerate(sys.argv):
    print("\tparamètre", position, ":", parametre)
```

Les exceptions

```
try:
    # code à essayer
except type_exception as exception:
    # code exécuté en cas d'erreur
finally:
    # code exécuté quelque soit l'issue du bloc
```

Déclenche une exception si la condition est fausse :

```
assert var == 5
```

Lever une exception :

```
raise TypeException("message")
```

Fonctions lambda

On déclare à la volée une fonction anonyme à l'aide de :

```
lambda x: expression(x)
```

qui équivaut à :

```
def f(x):
    return expression(x)
```

```
sorted(lst) # si a < b
sorted(lst, key=len) # si len(a) < len(b)
sorted(lst, key=lambda mot: mot[0])
sorted(lst, key=lambda mot: mot[-1])
```

Ce concept convient quand :

- La définition de la fonction est courte
- On n'a pas besoin de cette fonction ailleurs

La fonction map

Appliquer une transformation à tous les éléments d'un itérable

```
map(f, I) # applique la fonction f à tous les éléments de l'itérable I
```

Attention : map renvoie un itérable de type map. On consomme moins d'espace mémoire, et on a du code plus rapide.

```
L = [random() for i in range(10)]
M = list(map(str, L)) # réels -> chaînes
N = list(map(int, L)) # réels -> entiers
P = list(map(lambda x: x * 10, L)) # multiplier tous les éléments par 10
liste_chaines = ["au revoir", ...]
R = map(str.split, liste_chaines)
```

La fonction filter

Filtrer des éléments d'un itérable

```
filter(f, I) # récupère tous les éléments de l'itérable I pour lesquels la fonction f renvoie True
```

Attention : filter renvoie un itérable de type filter.

```
L = [random() for i in range(10)]
M = list(filter(lambda x: x > 0.5, L))
N = list(filter(lambda x: x <= 0.5, L))
P = [randint(0,1000) for I in range(20)]
Q = list(filter(lambda x: x % 2, P))
R = list(filter(lambda x: not x % 2, P))
```