Introduction à Python

Ahmed Ammar et Hassen Ghalila

Faculté des Sciences de Tunis, Université de Tunis El Manar

Contents

```
#Juste pour savoir la dernière fois que cela a été exécuté:
import time
print(time.ctime())
```

0.1 Utilisation de Python comme calculatrice

L'utilisation de la fonction `print()` n'est pas nécessaire pour obtenir un résultat. Il suffit de taper certaines opérations et le résultat est obtenu avec les touches `<SHIFT> + <ENTRER>`.

```
2 + 22
```

```
(2+3)*(3+4)/(5*5)
```

Python aime l'utilisation des espaces pour rendre les scripts plus lisibles

```
(2+3) * (3+4.) / (5*5)
```

L'art d'écrire un bon code python est décrit dans le document suivant: http://legacy.python.org/dev/peps/pep-0008/

0.2 Affectations

Comme tout autre langage de programmation, vous pouvez affecter une valeur à une variable. Ceci est fait avec le symbole "="

```
a = 4
```

```
a
```

```
a = a + 1
a
```

```
a *= 4 # semblable à a = a * 4
a
```

```
a, b = 1, 3
a, b
```

Certains noms de variables ne sont pas disponibles, ils sont réservés à python lui-même:

and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

```
lambda_ = 2
file = 3
```

0.3 Les commentaires

```
a = 2 # ceci est un commentaire
```

```
"""C'est un grand commentaire
sur plusieurs lignes
la fin est comme il a commencé"""
```

0.4 Les types

Les types utilisés dans Python sont:integers, long integers, floats (double prec.), complexes, strings, booleans.

La fonction `type()` donne le type de son argument:

```
type(2)
```

```
type(2.3)
```

```
int(0.8) # Tronquer
```

```
round(0.8765566777) # Valeur la plus proche

a = 1.5 + 0.5j # Nombres complexes

a**2.

(1+2j)*(1-2j)

a.real

(a**3).imag

a.conjugate() # C'est une fonction, il faut ()
```

0.4.1 Booleans

Les opérateurs de comparaison sont: <, >, <=, >=, ==, !=

```
5 < 7

a = 5
b = 7

b < a

c = 2

c < a < b

a < b and b < c

res = a < 7
print(res, type(res))

print (int(res))

not res is True

a = True
print (a)
```

0.5 Formatage des chaînes

```
print("Hello world!")
```

```
print ('Hello world!')
```

```
print ("Hello I'm here") # ' à l'intérieur ""
```

```
# C'est l'ancien mode de mise en forme des sorties (style C)
a = 7.5
b = 'tralala'
c = 8.9e-33
print('a = %f, b = %s, c = %e' % (a, b, c))
```

```
# La nouvelle façon est d'utiliser la méthode format() de l'objet chaîne et {}
# pour définir la valeur à imprimer et à utiliser quel format
print('a = {}, b = {}, c = {}'.format(a,b,c))
print('a = {0}, b = {1}, c = {2}'.format(a**2,b,c))
print('a = {:f}, b = {:20s}, c = {:15.3e}'.format(a,b,c))
```

0.6 Chaînes

```
a="C'est une chaîne"
```

```
len(a)
```

Beaucoup de commandes peuvent fonctionner sur des chaînes. Les chaînes, comme TOUT en python, sont des objets. Les méthodes sont exécutées sur des objets par points:

```
a.upper()
```

```
a.title()
```

```
a.split()
```

```
a.split()[1]
```

```
a = "C'est une chaîne. Avec différentes phrases."
```

```
a.split('.')
```

```
a.split('.')[1].strip() # Nous définissons ici le caractère utilisé pour se divi
                        # Le défaut est l'espace (une combinaison d'espaces)
```

```
a = 'tra'
b = 'la'
print (' '.join((a,b,b)))
print ('-'.join((a,b,b)))
print (''.join((a,b,b)))
print (' '.join((a,b,b)).split())
print (' & '.join((a,b,b)) + '\\\\')
```

0.7 Conteneurs: Tuples, listes et dictionnaires

0.7.1 Liste:

Une collection d'objets. Peut être de différents types. Elle a un ordre.

```
L = ['red','green','blue'] # Les crochets sont utilisés pour définir des listes
```

```
type(L) # Affiche le type de L
```

```
L[1]
```

```
L[0] # Les indices commencent à 0 !!!
```

```
L[-1] # Dernier élément
```

```
L[-3]
```

```
L = L + ['black', 'white'] # Le symbole d'addition sert à agréger les valeurs d'
```

```
print(L)
```

```
L[1:3] # L [start: stop]: éléments d'index i, où start <= i <stop !! l'élément s
```

```
L[2:] # Les limites peuvent être omises
```

```
L[-2:]
```

```
L[::2] # L [start: stop: step] tous les 2 éléments
```

```
L[::-1]
```

Les listes sont mutables: leur contenu peut être modifié.

```
L[2] = 'yellow'
L
```

```
L.append('pink') # Ajouter une valeur à la fin
L
```

```
L.insert(2, 'blue')    #L.insert (index, objet) -- insérer l'objet avant indice
L
```

```
L.extend(['magenta', 'purple'])
L
```

```
L.append(['magenta', 'azul'])
L
```

```
L.append(2)
L
```

```
L = L[::-1] # ordre inverse
L
```

```
L2 = L[:-3] # Coupant les 3 derniers éléments
print (L)
print (L2)
```

```
L[25] # Out of range conduit à une erreur
```

```
print(L)
print (L[20:25]) # Mais NON ERREUR lors du tranchage.
print(L[20:])
print (L[2:20])
```

```
print (L.count('yellow'))
```

On peut µutiliser TAB pour rechercher les méthodes (fonctions qui s'appliquent à un objet)

```
a = [1,2,3]
b = [10,20,30]
```

```
print(a+b) # Peut-être pas ce que vous attendiez, mais plutôt logique
```

```
print(a*b) # Ne multiplie pas l'élément par élément. Numpy fera ce travail.
```

```
L = range(4) # Créez une liste. Notez que le paramètre est le nombre d'éléments,
L
```

```
L = range(2, 20, 2) # Tous les 2 entiers
L
```

0.7.2 Tuples:

comme des listes, mais immuables

```
T = (1,2,3)
T
```

```
T2 = 1, 2, 3
print (T2)
type(T2)
```

```
T[1]
```

Les tuples sont immuables

```
T[1] = 3 # Ne marche pas!
```

0.7.3 Dictionnaries

Un dictionnaire est essentiellement une table efficace qui mappe les clés des valeurs. C'est un conteneur non ordonné

```
D = {'Christophe': 12, 'Antonio': 15} # Défini par {key : value}
```

```
D['Christophe'] # Accéder à une valeur par la clé
```

```
D.keys() # Liste des clés du dictionnaire
```

```
D['Julio'] = 16 # Ajout d'une nouvelle entrée
```

```
print(D)
```

0.8 Structures de contrôle

0.8.1 Exécution conditionnelle if - elif - else

L'exemple ci-dessous illustre la forme complète de cette structure. Les parties `elif...` et `else...` sont facultatives. Pour des tests multiples, on peut bien entendu cascader plusieurs parties `elif...`.

Notez bien la présence du caractère : (double point) précédant le début de chaque bloc !

```
a, b = 4, 5
if a > b:
    print("%f est supérieur à %f" % (a,b) )
elif a == b:
    print("%f est égal à %f" % (a,b) )
else:
    print("%f est inférieur à %f" % (a,b) )
```

0.8.2 Boucle for

La boucle for permet d'itérer les valeurs d'une liste, d'un tuple, d'une chaîne ou de tout objet itérable. Comme dans les autres structures de contrôle, le caractère : (double point) définit le début du bloc d'instruction contrôlé par for.

```
# Sur listes
lst = [10, 20, 30]
for n in lst:
    print(n, end=' ')

# Sur tuples
for index in range(len(lst)):
    print(index, lst[index])

for index,val in enumerate(lst):
    print(index, val) # => même affichage que ci-dessus
```

```
# Sur chaînes
voyelles = 'aeiouy'
for car in 'chaine de caracteres':
    if car not in voyelles:
        print(car, end='')
```

Pour itérer sur une suite de nombres entiers, on utilise souvent la fonction range (objet itérable) présentée plus haut.

```
# Sur dictionnaires
carres = {}
for n in range(1,4):
    carres[n] = n**2
carres
```

0.8.3 Boucle while

La boucle while permet d'exécuter un bloc d'instruction aussi longtemps qu'une condition (expression logique) est vraie.

Notez aussi la présence du caractère : (double point) définissant le début du bloc d'instruction contrôlé par while.

```
nb = 1 ; stop = 5
# Affiche le carré des nombres de nb à stop
while nb <= stop:
    print(nb, nb**2)
    nb += 1
```

0.9 Fonctions, modules, packages, scripts

0.9.1 Fonctions

De façon générale, on implémente une fonction lorsqu'un ensemble d'instructions est susceptible d'être utilisé plusieurs fois dans un programme. Cette décomposition en petites unités conduit à du code plus compact, plus lisible et plus efficace.

L'exemple ci-dessous illustre les principes de base de définition d'une fonction en Python : - La première ligne `def nomFonction(arguments...):` définit le nom avec lequel on invoquera la fonction, suivi entre parenthèses de son(ses) éventuel(s) arguments (paramètres d'entrée) séparés par des virgules ; cette ligne doit se terminer par : - Les instructions de la fonction (corps de la fonction) constituent ensuite un bloc qui doit donc être indenté à droite - Au début du corps on peut définir, sous forme de chaîne/commentaire multi-ligne, le texte d'aide en-ligne (appelé docstrings) qui sera affiché avec `help(fonction)` - Avec l'expression `return` on sort de la fonction en renvoyant optionnellement des données de retour sous forme d'un objet de n'importe quel type ; si l'on ne passe pas d'argument à return, la fonction retournera alors None (objet nul) ; dans l'exemple ci-contre, on retourne 2 valeurs que l'on a choisi d'emballer sous forme de tuple

```
def somProd(n1, n2):
    """Fonction calculant somme et produit de n1 et n2
    Résultat retourné dans un tuple (somme, produit)"""
    return (n1+n2, n1*n2)

help(somProd)

somProd(3,10)

somProd()  # erreur "somProd() takes exactly 2 args"
           # et même erreur si on passe 1 ou >2 args.
```

Une fonction étant un objet, on peut l'assigner à une variable, puis utiliser celle-ci comme un "alias" de la fonction !

```
sp = somProd
sp(3,10)
```

Lors de l'appel à la fonction, il est aussi possible de passer les paramètres de façon nommée avec `paramètre=valeur`. Dans ce cas, l'ordre dans lequel on passe ces paramètres n'est pas significatif !

On peut en outre définir, dans la déclaration def, des paramètres optionnels. Si l'argument n'est pas fourni lors de l'appel de la fonction, c'est la valeur par défaut indiquées dans la définition qui sera utilisée par la fonction.

```
def fct(p1, p2=9, p3='abc'):
    # 1 param. obligatoire, et 2 param. optionnels
    return (p1, p2, p3)

print(fct())          # => erreur (1 param. oblig.)

print(fct(1))

print(fct(1, 2))

print(fct(1, 2, 3))

print(fct(p3=3, p1='xyz'))
```

0.9.2 Modules

Un module Python (parfois appelé bibliothèque ou librairie) est un fichier rassemblant des fonctions et classes relatives à un certain domaine. On implémente un module lorsque ces objets sont susceptibles d'être utilisés par plusieurs programmes.

Pour avoir accès aux fonctions d'un module existant, il faut charger le module avec la commande `import`, ce qui peut se faire de différentes manières, notamment :

a) `from module import *` : on obtient l'accès direct à l'ensemble des fonctions du module indiqué sans devoir les préfixer par

```
from math import *

dir()    # => objets dans namespace, notamment ces fcts

sin(pi/2)

cos(pi)
```

b) `from module import fct1, fct2...` : on ne souhaite l'accès qu'aux fonctions fct1, fct2... spécifiées

```
from math import pi, sin

dir()    # => objets dans namespace, notamment ces fcts

sin(pi/2)

cos(pi)
```

c) `import module1, module2...` : toutes les fonctions de(s) module(s) spécifié(s) seront accessibles, mais seulement en les préfixant du nom du module

```
import math

# math.<tab> => liste les fonctions du module
dir(math)         # => objets dans namespace

help(math)        # => affiche l'aide sur ces fonctions

help(math.sin)    # => aide sur la fct spécifiée (sin)

math.sin(math.pi/2)

cos(pi)           # => erreur (non préfixés par module)

math.cos(math.pi)
```

d) `import module as nomLocal` : toutes les fonctions du module sont accessible en les préfixant du nomLocal que l'on s'est défini

```
import math as mt
```

```
# mt.<tab>  => liste les fonctions du module

dir(mt)           # => objets dans namespace

help(mt)          # => affiche l'aide sur ces fonctions

mt.sin(mt.pi/2)

math.sin(math.pi/2) # => erreur "name math not defined"

mt.cos(mt.pi)
```