Mise à niveau

python

CPGE Lycée Omar Ibn Abdelaziz

Oujda

MPSI 2-3

Plan

- Présentation de Python
- Les types et les opérations de base
- Les structures de contrôle
- Les listes/tuples/dictionnaire/set/chaine de caractères
- Les fonctions
- Les fichiers
- Les classes
- Les exceptions
- Les modules, Ressources

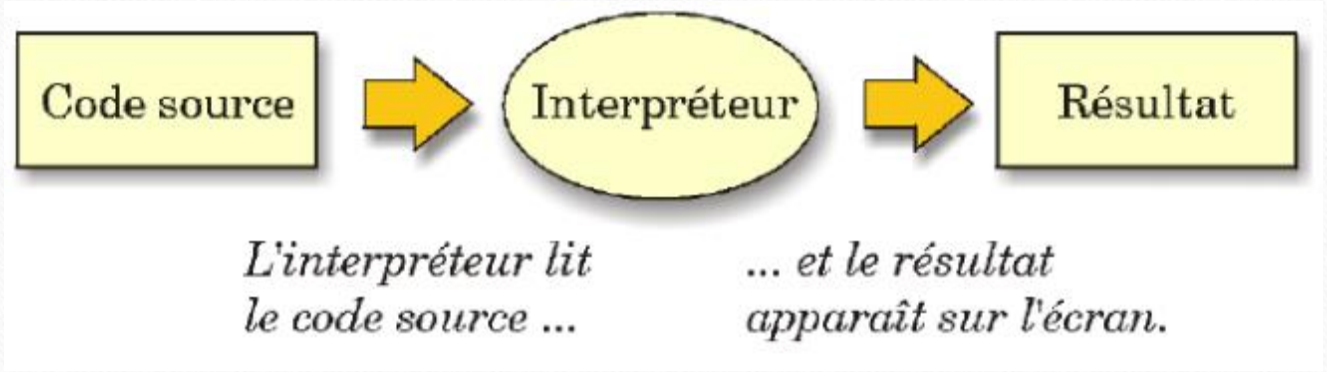
Le langage Python

- Développé en 1989 par Guido van Rossum (v1: 1991)
- Open-source
- Portable
- Orienté objet
- Dynamique
- Extensible
- Support pour l'intégration d'autres langages

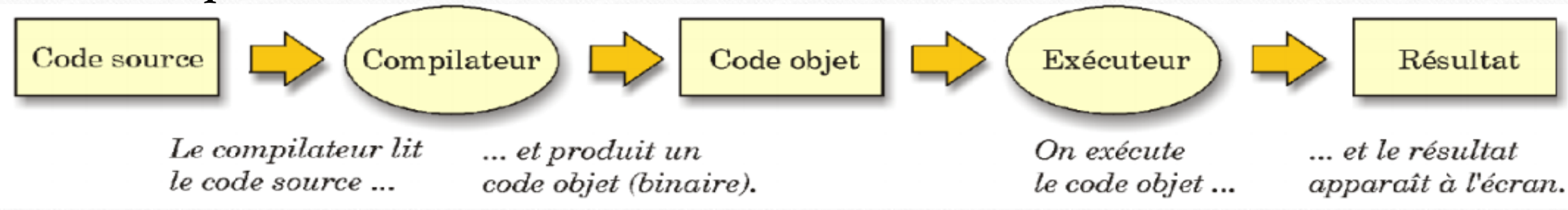
Comment faire fonctionner mon code source ?

Il existe 2 techniques principales pour effectuer la traduction en langage machine de mon code source :

- **Interprétation**:



- **Compilation:**



Et Python ?

- **Avantages :** interpréteur permettant de tester n'importe quel petit bout de code, compilation transparentes.

- **Inconvénients :** peut être lent.

Les différentes versions

- Il existe 2 versions de Python : 2.x et 3.x.
- Python 3.x n'est pas une simple amélioration ou extension de Python 2.x.
- Tant que les auteurs de librairies n'auront pas effectué la migration, les deux versions devront coexister.
- Nous nous intéresserons uniquement à Python 3.x.

[1]: https://wiki.python.org/moin/Python2orPython3

Comment lancer un script Python

Il y a différentes méthodes pour lancer un script Python

– Ecrire le code dans un fichier “script.py”. Et “python script.py”

– Coder directement dans l’interpréteur.

– Utiliser un IDE (eg. Pyzo, PyCharm, Spyder,…)

- Utiliser Anaconda(choix recommandé)

Opérations de base

affectation

```
In [1]: i = 3 # i vaut 3
In [2]: a, pi = True, 3.14159
In [3]: k = r = 2.15
```

affichage dans l'interpréteur

```
In [4]: i
Out[4]: 3
In [5]: print(i)
Out[5]: 3
```

Input/Output

Récupérer la données ???

- fonction **input** : affiche un message (optionnel) à l'utilisateur et récupère la données saisie par l'utilisateur:

Syntaxe : **variable=input("message")**

Exemple

```
# demandez une valeur à l'utilisateur
nom = input(" Donnez votre nom : ")
prenom = input(" Donnez votre prenom : ")

# Affichage du nom complet
print( " Bienvenue :", nom, prenom)
```

Input/Output

Récupérer la données ???

- Fonction **input** : retourne une valeur de type **texte(‘3’ est différente de 3)**
- Fonction **eval** : évaluer et convertir en une valeur numérique une valeur contenu dans un texte.(on peut utiliser aussi les fonctions **int** et **float**)

  Exemple :

  - eval("34.5") retourne 34.5
  - eval("345") retourne 345
  - eval("3 + 4") retourne 7
  - eval("51 + (54 * (3 + 2))") retourne 321.

Les opérateurs

```
>>> 10 + 3
13
>>> 10 - 3
7
>>> 10 * 3
30
>>> 10 / 3
3
>>> 10 // 3
3
>>> 10 % 3
1
>>> 10**3
1000
>>> 10 + 3 * 5  # *,/ then +,-
25
>>> (10 + 3) * 5
65
>>> -1**2  # -(1**2)
-1
```

```
>>> 10.0 + 3.0
13.0
>>> 10.0 - 3.0
7.0
>>> 10.0 * 3
30.0
>>> 10.0 / 3
3.3333333333333335
>>> 10.0 // 3
3.0
>>> 10.0 % 3.0
1.0
>>> 10.0**3
1000.0
>>> 4.2 + 3.14
7.3399999999999999
>>> 4.2 * 3.14
13.188000000000001
```

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0
**	Exponent - Performs exponential (power) calculation on operators	a**b will give 10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

a=10

b=20

Et '+=' ?

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.

a=10

b=20

Operator	Description	Example a,b = 10,02	Example a,b = 0,02
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	(a and b) is true.	(a and b) is false.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(a or b) is true.	(a or b) is true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not(a and b) is false.	not(a and b) is true.

L’ordre des opérateurs

Parenthesis
Power
Division
Multiplication
Soustraction
Addition
Left to Right

1. 1 + 2 ** 3 / 4 * 5
2. 1 + 8 / 4 * 5
3. 1 + 2 * 5
4. 1 + 10
5. 11

Les chaînes de caractères

```
s = 'hi'
print(s[1])          ## i
print(len(s) )       ## 2
print(s + ' there' ) ## hi there

multi = """It was the best of times.
 It was the worst of times."""
```

Concatenation

```
pi = 3.14
##text = 'The value of pi is ' + pi      ## NO, does not work
text = 'The value of pi is '  + str(pi)  ## yes
```

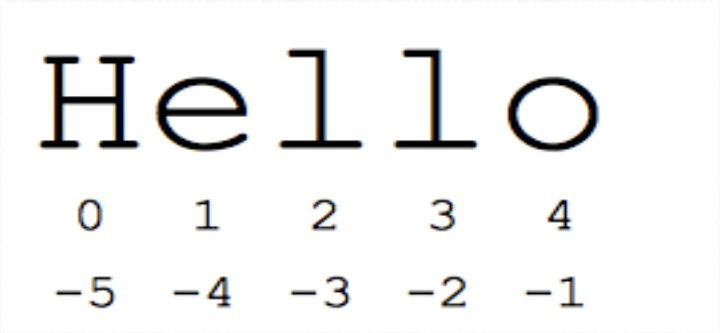
- **s.lower(), s.upper()** -- returns the lowercase or uppercase version of the string
- **s.strip()** -- returns a string with whitespace removed from the start and end
- **s.isalpha()/s.isdigit()/s.isspace()...** -- tests if all the string chars are in the various character classes
- **s.startswith('other'), s.endswith('other')** -- tests if the string starts or ends with the given other string
- **s.find('other')** -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
- **s.replace('old', 'new')** -- returns a string where all occurrences of 'old' have been replaced by 'new'
- **s.split('delim')** -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
- **s.join(list)** -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

Le reste: https://docs.python.org/2/library/stdtypes.html#string-methods

Manipulations des chaînes de caractères

Accès au caractères

Supposons on a s = "Hello”



- **s[1:4] is 'ell'** -- chars starting at index 1 and extending up to but not including index 4
- **s[1:] is 'ello'** -- omitting either index defaults to the start or end of the string
- **s[:] is 'Hello'** -- omitting both always gives us a copy of the whole thing (this is the pythonic way to copy a sequence like a string or list)
- **s[1:100] is 'ello'** -- an index that is too big is truncated down to the string length

- **s[-1] is 'o'** -- last char (1st from the end)
- **s[-4] is 'e'** -- 4th from the end
- **s[:-3] is 'He'** -- going up to but not including the last 3 chars.
- **s[-3:] is 'llo'** -- starting with the 3rd char from the end and extending to the end of the string.

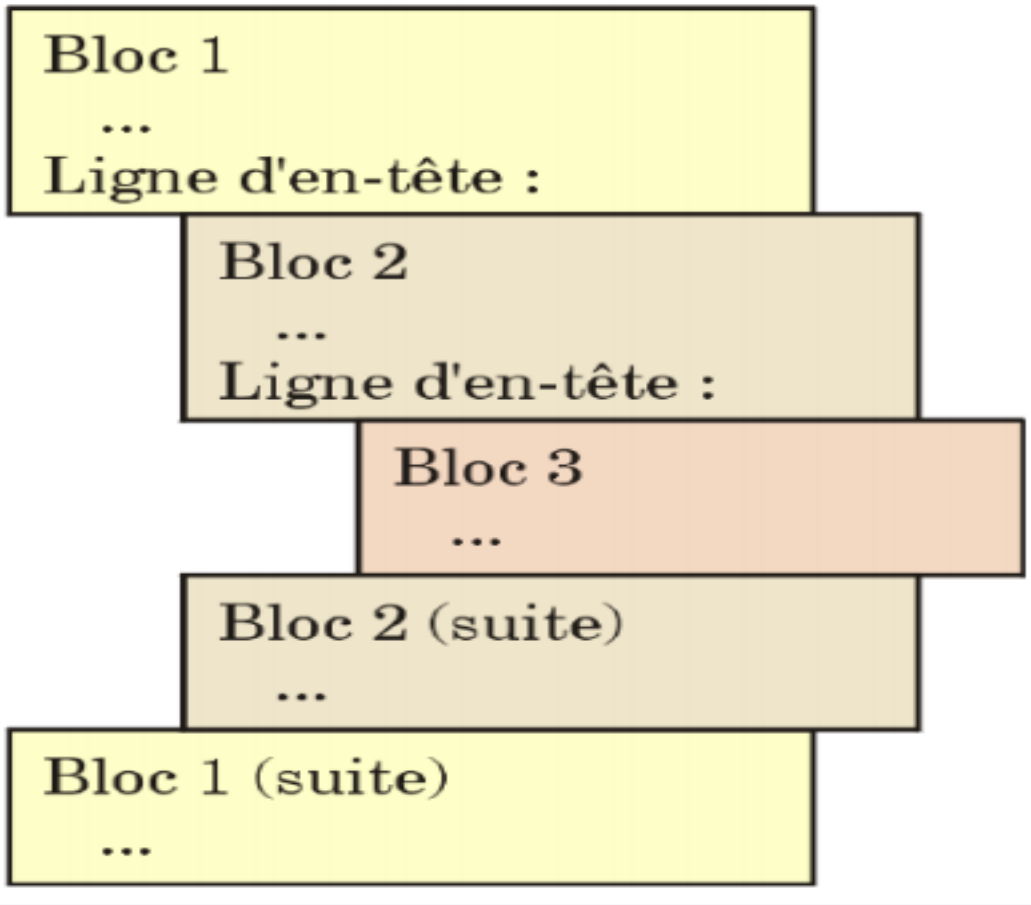
Manipulations des chaînes de caractères

Pour le formatage des chaînes de caractères on utilise l'opérateur %

```
# % operator
text = "%d little pigs come out or I'll %s and %s and %s" % (3, 'huff', 'puff', 'blow down')
```

Les structures de contrôle IF

Python does not use { } to enclose blocks of code for if/loops/function etc.. Instead, Python uses the colon (:) and indentation/whitespace to group statements.



Les structures de contrôle IF

Syntaxe général

```
if <test1>:
    <blocs d'instructions 1>
elif <test2>:
    <blocs d'instructions 2>
else:
    <blocs d'instructions 3>
```

Les structures de contrôle IF

```
if speed >= 80:
    print('License and registration please‘)
    if mood == 'terrible' or speed >= 100:
        print('You have the right to remain silent.‘)
    elif mood == 'bad' or speed >= 90:
        print("I'm going to have to write you a ticket.“)
        write_ticket()
else:
    print("Let's try to keep it under 80 ok?“)
```

Les structures de contrôle `**while**`

Syntaxe général:

```
while <test1>:
    <blocs d'instructions 1>
    if <test2>: break
    if <test3>: continue
else:
    <blocs d'instructions 2>
```

- **break** : sort de la boucle sans passer par else,
- **continue** : remonte au début de la boucle,
- **pass** : ne fait rien,
- **else** : lancé si et seulement si la boucle se termine normalement.

Les structures de contrôle `**while**`

Exemples:

boucle infinie

```
while 1:
    pass
```

y est-il premier ?

```
x = y / 2
while x > 1:
    if y % x == 0:
        print(y, 'est facteur de', x)
        break
    x = x-1
else:
    print(y, 'est premier')
```

Les structures de contrôle `**for**`

Format général:

```
for <cible> in <objet>:
    <blocs d'instructions>
    if <test1>: break
    if <test2>: continue
else:
    <blocs d'instructions>
```

- **break** : sort de la boucle sans passer par else,
- **continue** : remonte au début de la boucle,
- **pass** : ne fait rien,
- **else** : lancé si et seulement si la boucle se termine normalement.

Les listes

- Tous les langages de programmation offrent, d'une manière ou d'une autre, la possibilité de manipuler des collections d'éléments, souvent appelées tableaux.
- Un tableau est généralement une collection d'éléments de même type. Par exemple en langage C, on déclare un tableau d'entiers de la manière suivante : int monTableau[10] ;
- l'accès aux éléments du tableau se fait ensuite en utilisant le nom du tableau et en fournissant un indice entre crochets :
  - monTableau[5] = 150 ;
  - val = monTableau[0] ;

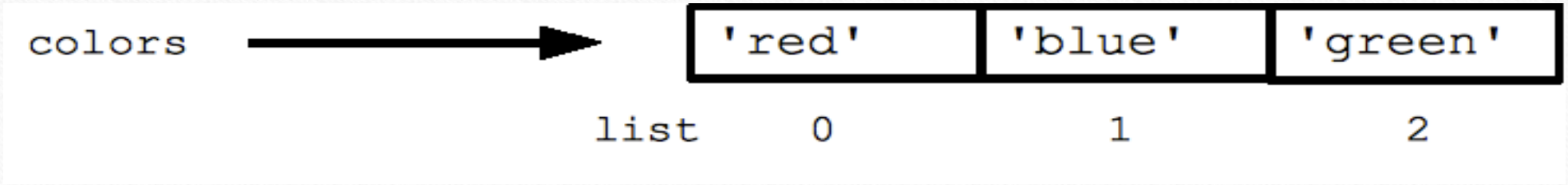
Les listes

- En langage Python, les listes fonctionnent un peu comme les tableaux du langage C mais avec quelques différences notables.
- Comme pour toute autre variable en Python, on créé une liste sans déclaration préalable, sans préciser le type des éléments contenus dans la liste.
- maListe = [1,2,3,4,5,6,7,8,9]
- On peut aussi créé une liste vide au départ : maListe = [ ]
- ou encore : maListe = list()

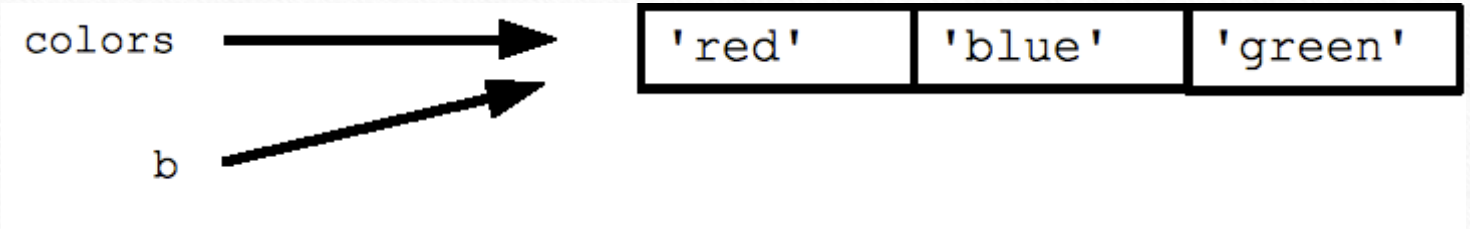
Les listes

- colors = ['red', 'blue', 'green']

```
print(colors[0])   ## red
print(colors[2])   ## green
print(len(colors))  ## 3
```



- b = colors ## Does not copy the list



Les listes

ATTENTION !

```
>>> L = ['Dans', 'python', 'tout', 'est', 'objet']
>>> T = L
>>> T[4] = 'bon'
>>> T
 ['Dans', 'python', 'tout', 'est', 'bon']
>>> L = T[:]
>>> L[4] = 'objet'
>>> T; L
['Dans', 'python', 'tout', 'est', 'bon'] ['Dans', 'python', 'tout', 'est', 'objet']
```

Opérations sur les listes

```
>>> sept_zeros = [0]*7; sept_zeros
[0, 0, 0, 0, 0, 0, 0]
>>> L1, L2 = [1, 2, 3], [4, 5]
>>> L1 + L2
[1, 2, 3, 4, 5]
```

Une liste est une séquence comme pour les chaînes de caractères

Gérer les listes

- **list.append(elem)** -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- **list.insert(index, elem)** -- inserts the element at the given index, shifting elements to the right.
- **list.extend(list2)** adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().
- **list.index(elem)** -- searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).
- **list.remove(elem)** -- searches for the first instance of the given element and removes it (throws ValueError if not present)
- **list.sort()** -- sorts the list in place (does not return it). (The sorted() function shown below is preferred.)
- **list.reverse()** -- reverses the list in place (does not return it)
- **list.pop(index)** -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

Gérer les listes

```
list = ['larry', 'curly', 'moe']
list.append('shemp')         ## append elem at end
list.insert(0, 'xxx')        ## insert elem at index 0
list.extend(['yyy', 'zzz'])  ## add list of elems at end
print(list)  ## ['xxx', 'larry', 'curly', 'moe', 'shemp', 'yyy', 'zzz']
print(list.index('curly'))    ## 2

list.remove('curly')         ## search and remove that element
list.pop(1)                  ## removes and returns 'larry'
print(list)  ## ['xxx', 'moe', 'shemp', 'yyy', 'zzz']
```

Gérer les listes

```
list = ['a', 'b', 'c', 'd']
print(list[1:-1])   ## ['b', 'c']
list[0:2] = 'z'     ## remplacer['a', 'b'] par ['z']
print(list)         ## ['z', 'c', 'd']
```

Les fonctions

Definition

```
def <nom_fonction>(arg1, arg2,... argN):
    ...
    bloc d'instructions
    ...
    return <valeur(s)>
```

Les fonctions

Example

Fonction sans paramètres

```
def table7():
    n = 1
    while n < 11:
        print(n*7)
        n += 1
```

Remarque Une fonction qui n'a pas de return renvoie par défaut **None**.

Les fonctions

Example

Fonction avec paramètre

```
def table(base):
    n = 1
    while n < 11:
        print(n*base)
        n += 1
```

Les fonctions

Example

Fonction avec plusieurs paramètres

```
def table(base, debut=0, fin=11):
    print ('Fragment de la table de multiplication par'\ , base, ':' )
    n = debut
    l = []
    while n < fin:
        print (n*base)
        l.append(n*base)
        n += 1
        return l
```

Les fonctions

Example

Déclaration d'une fonction sans connaître ses paramètres

```
>>> def f(*args, **kwargs):
... Print(args)
... Print(kwargs)
>>> f(1, 3, 'b', j = 1)
(1, 3, 'b')
'j': 1
```

Les fonctions

Lambda

Python permet une syntaxe intéressante qui vous laisse définir des mini-fonctions d'une ligne à la volée. Empruntées à Lisp, ces fonctions dites lambda peuvent être employées partout où une fonction est nécéssaire.

Definition:

lambda argument1,... argumentN : expression utilisant les arguments

Exemple:

```
>>> f = lambda x, i : x**i
>>> f(2, 4)
16
>>> val_abs=lambda x: x if x>=0 else –x
>>> val_abs(-5)
5
```

Les tuples

Initialisation

T=() | tuple() | (1,) | 'a', 'b', 'c', 'd' | ('a', 'b', 'c', 'd')

Concaténation

```
>>> (1, 2)*3
(1, 2, 1, 2, 1, 2)
>>> t1, t2 = (1, 2, 3), (4, 5)
>>> t1 + t2
(1, 2, 3, 4, 5)
```

Un tuple est aussi une séquence.

Les tuples

Un tuple n'est pas modifiable

```
>>> t = 'a', 'b', 'c', 'd'
>>> t[0] = 'alpha'
        Traceback
        (most recent call last):
            File "", line 1, in ?
        TypeError: object does not support item assignment
>>> t= ('alpha',) + t[1:]
>>> t ('alpha', 'b', 'c', 'd')
```

Les dictionnaires

Initialisation

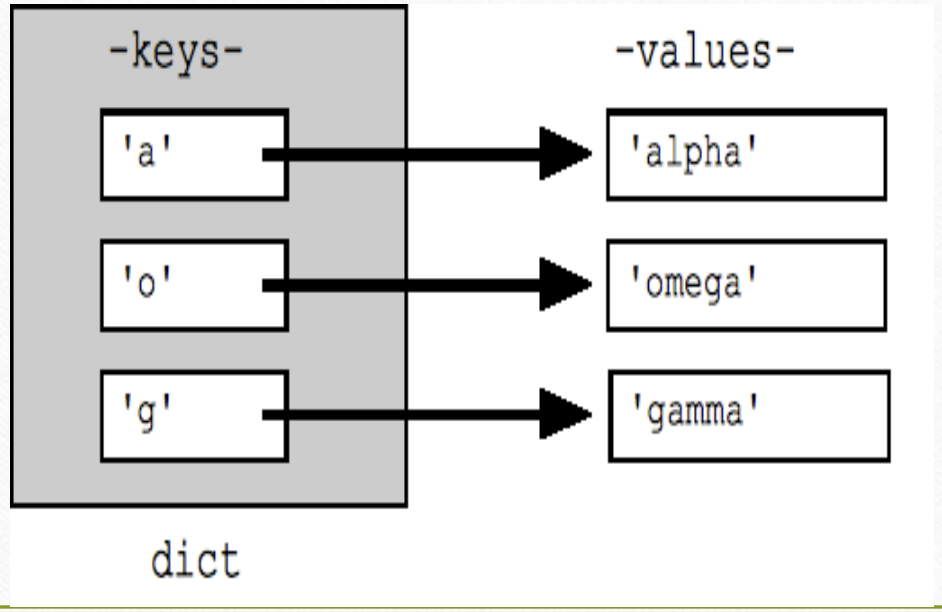
D={} | dict() | {'point': 1, 'ligne': 2, 'triangle': 3}

Remarques

- un dictionnaire est constitué de clés et de valeurs
- on ne peut pas concaténer un dictionnaire avec un autre

Ajout d'une clé ou modification d'une valeur

```
>>> dico['quad'] = 4
>>> dico
{'quad': 4, 'ligne': 2, 'triangle': 3, 'point': 1}
>>> dico['point'] = 3
{'quad': 4, 'ligne': 2, 'triangle': 3, 'point': 3}
```



Les dictionnaires

Copie d'un dictionnaire

```
>>> dico = {'computer':'ordinateur', 'mouse':'souris', 'keyboard':'clavier'}
>>> dico2 = dico
>>> dico3 = dico.copy()
>>> dico2['printer'] = 'imprimante'
>>> dico2
{'computer': 'ordinateur', 'mouse': 'souris', 'printer': 'imprimante', 'keyboard': 'clavier'}
>>> dico
{'computer': 'ordinateur', 'mouse': 'souris', 'printer': 'imprimante', 'keyboard': 'clavier'}
>>> dico3
 {'computer': 'ordinateur', 'mouse': 'souris', 'keyboard': 'clavier'}
```

Manipuler les dictionnaires

Un dictionnaire a ses propres méthodes (help(dict))

- len(dico) : taille du dictionnaire
- dico.keys() : renvoie les clés du dictionnaire sous forme de liste
- dico.values() : renvoie les valeurs du dictionnaire sous forme de liste
- dico.has_key : renvoie True si la clé existe, False sinon
- dico.get : donne la valeur de la clé si elle existe, sinon une valeur par défaut

Les structures de contrôle `zip` et `map`

- **zip** : permet de parcourir plusieurs séquences en parallèle
- **map** : applique une méthode sur une ou plusieurs séquences

Remarque

map peut être beaucoup plus rapide que l'utilisation de for

Les structures de contrôle `zip` et `map`

Exemples

Utilisation de zip

```
L1 = [1, 2, 3]
L2 = [4, 5, 6]

for (x, y) in zip(L1, L2):
    print(x, y, '--', x + y)
```

Utilisation de map

S = '0123456789'

print(list(map(int, S)))

Les structures de contrôle `zip` et `map`

Autre Exemple

```
def addition(n):
    return n + n

numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

FICHIER SOUS PYTHON

- **Introduction :**

Dans tous les langages de programmation, on utilise la notion de fichier. Qui désigne un ensemble d'informations enregistrées sur un support (clé usb, disque dur, etc.).

Dans la majorité des langages de programmation, on distingue deux types de fichiers :

- Les fichiers **textes** : les informations sont sous un format texte (.txt, .docs, …) qui est lisible par n'importe quel éditeur de texte,

Les fichiers **binaires**: les informations ne sont lisibles que par le programme qui les a conçus (.pdf, .jpg, …).

FICHIER SOUS PYTHON

- **Introduction :**

Les principales manipulations sur un fichier sont :

- **L'ouverture** du fichier
- La **lecture** ou **l'écriture** d'un élément dans un fichier
- La **fermeture** du fichier

- **Ouverture d'un fichier :**

Tout fichier doit être ouvert avant de pouvoir accéder à son contenu en **lecture** et **écriture**. L'ouverture d'un fichier est réalisée par la fonction **open** selon la syntaxe suivante :

Variable = open(chemin, mode)

Mode : 'r' : mode lecture seule 'w' : mode écriture seule 'a' : mode ajout 'b' : binaire

FICHIER SOUS PYTHON

Exemples :

```
>>> f1 = open (‘c:\\test1.txt’, "r")
 #ouvrir le fichier test1 en mode lecture seulement
>>> f2 = open ("c:\\test2.txt" , ‘wb’)
 #ouvrir le fichier test1 en mode écriture binaire
>>> f3 = open ("c:\\ test3.txt" , ‘a’)
#ouvrir le fichier test1 en mode ajout (append)
>>> f4 = open ("c:\\ test3.txt" , ‘r+’)
 #ouvrir le fichier test1 en mode lecture/écriture
```

FICHIER SOUS PYTHON

Remarques :

- En mode **lecture**, le fichier doit **exister** sinon une exception de type ***FileNotFoundError*** sera levée.
- En mode écriture, Si le fichier n'existe pas, il est créé sinon, son contenu est **perdu**.
- En mode ajout, si le fichier existe déjà, il sera étendu. sinon, il sera **créé**.
- N'oublie pas à la fin de fermer le fichier afin que d'autres programmes puissent le lire via la commande :

```
f1.close() # fermeture du fichier
```

FICHIER SOUS PYTHON

Ecriture dans un fichier

La méthode **write(message)** permet d'écrire la chaîne de caractère *message* dans un fichier. elle retourne le nombre de caractères écrits dans le fichier

```
f = open (" test.txt ", 'w')
f.write ('DUT IDSD1\n ESTE 2020\n.')
f.close ( )
```

FICHIER SOUS PYTHON

Lecture d'un fichier

- **méthode read(t) :** Pour lire la quantité **t** à partir de la position déjà atteinte dans le fichier et les retourne en tant que chaîne de caractères. Quand **t** est omise ou négative, le contenu **tout entier** du fichier est lu et retourné.

- **méthode readline( ) :** lit une seule ligne à partir du fichier.

- **méthode readlines( ) :** retourne une liste contenant toutes les lignes du fichier

FICHIER SOUS PYTHON

Exeemple :

- Quel est le résultat de l'exécution du programme ci-dessus ?

```
f = open ("essai.txt", "w")
f.write ("Bonjour INFO1, ceci n'est pas évident !!!")
f.close ()
f = open ("essai.txt", "r")
data1 = f.read()
data2 = f.read()
print(data1 , "****" ,data2)
f.close ()
```

Bonjour INFO1, ceci n'est pas évident !!! **

- Si on change data1 = f.read() par data1 = f.read(**13**), que sera-t-il le résulta de l'exécution du programme ?

Bonjour INFO1 ** , ceci n'est pas évident !!!**

FICHIER SOUS PYTHON

Remarque : De façon pragmatique, l'instruction **with** permet d'écrire un code plus court sans utiliser l'instruction close. Les deux bouts de code suivant sont équivalents :

```
f = open("abc.txt", "r")
for ligne in f:
    print(ligne)
f.close()
```

```
with  open("abc.txt", "r")    as f :
      for ligne in f:
          print(ligne)
```

FICHIER SOUS PYTHON

Exercice : On désire stocker les notes des étudiant d'une classe dans un fichier. Ecrire un programme dans lequel l'utilisateur rentre :

- Le nom du fichier (on exige l'extension '.txt')
- Le nombre des étudiants
- La saisie des noms et

notes (0<= note <= 20)

```
file=input("donner le nom du fichier : ")
file=file+".txt"
n=int(input("donner le nombre des étudiants : "))
with open(file,"a") as f :
    for i in range(n):
        name=input("donner le nom de l'étudiant : ")
        note=float(input("donner sa note : "))
        while not(0<=note<=20):
            note=float(input("donner sa note : "))
        f.write(name+'\t'+str(note)+'\n')
```

FICHIER SOUS PYTHON

Enregistrer des objets dans des fichiers

Grâce au module **pickle**, on peut enregistrer n'importe quel objet et le récupérer par la suite, au prochain lancement du programme.

Tout d'abord, il faut importer le module pickle.

```
import pickle
```

- Dans la suite, nous allons utiliser deux classes incluses dans ce module : la classe **Pickler** et la classe **Unpickler**.

FICHIER SOUS PYTHON

Classe Pickler

Pour créer un objet **Pickler**, nous allons l'appeler en passant en paramètre le fichier dans lequel nous allons enregistrer nos objets.

```
with open('donnees', 'wb') as fichier :
    mon_pickler = pickle.Pickler(fichier)
    # enregistrement ...
```

Remarque

- Les objets seront enregistrés dans le fichier ‘donnees’. J’ai pas donné d'extension, vous pouvez le faire.
- Notez le mode d'ouverture est le mode écriture binaire.
- Le fichier créer ne sera pas très lisible si vous essayez de l'ouvrir, mais ce n'est pas le but.

FICHIER SOUS PYTHON

Enregistrer un objet dans un fichier

On utilise la méthode **dump** du **Pickler** pour enregistrer l'objet. Son emploi est :

```
with open('donnees.dat', 'wb') as fichier :
    mon_pickler = pickle.Pickler(fichier)
    mon_pickler.dump(objet)
```

```
import pickle
score = {"joueur 1":5,"joueur 2":35,"joueur 3":20,"joueur 4":2,}
L=[i for i in range(4)]
with open('donnees.dat', 'wb') as fichier:
    mon_pickler = pickle.Pickler(fichier)
    mon_pickler.dump(score)
    mon_pickler.dump(L)
```

FICHIER SOUS PYTHON

Récupérer nos objets enregistrés

Classe Unpickler

- Commençons par créer notre objet **Unpickler**. À sa création, on lui passe le fichier dans lequel on va lire les objets. Notons qu'on doit changer le mode, on repasse en mode **rb** puisque le fichier est binaire.

```
with open('donnees.dat', rb") as fichier :
    mon_depickler = pickle.Unpickler(fichier)
    # Lecture des objets contenus dans le fichier...
```

- Pour lire les objets contenues dans notre fichier, nous utilisons la méthode **load** de notre **depickler**. Elle renvoie le premier objet qui a été lu (s'il y en a plusieurs, il faut l'appeler plusieurs fois).

FICHIER SOUS PYTHON

Récupérer nos objets enregistrés

```
with open('donnees.dat', 'rb') as fichier:
    mon_depickler = pickle.Unpickler(fichier)
    score_recupere = mon_depickler.load()
    print(score_recupere)
    L=mon_depickler.load()
    print(L)
```

{'joueur 1': 5, 'joueur 2': 35, 'joueur 3': 20, 'joueur 4': 2}
[0, 1, 2, 3]

FICHIER SOUS PYTHON

Exemple : L=[0,1,2,3] et formule $f(x)=x^2/3$

On veut écrire un programme qui permet de stocker les données dans un fichiers nommé ‘save.p’

```
import pickle
L=[i for i in range(4)]
def f(x):
    return x**2/3
with open(‘save.p', 'wb') as fichier:
    mon_pickler = pickle.Pickler(fichier)
    mon_pickler.dump(f)
    mon_pickler.dump(L)
```

```
with open(‘save.p', 'rb') as fichier:
    mon_depickler = pickle.Unpickler(fichier)
    g=mon_depickler.load()
    L=mon_depickler.load()
    for e in L:
        print(g(e))
```

Module Os

Opérations sur les fichiers et dossiers

- En Python, on peut changer le répertoire de travail courant. Pour cela, vous devez utiliser une fonction du module **os**

- **Se placer dans un dossier**

```
import os
os.chdir('C:/Users/windows 7 fr/Desktop/abc')
```

- **Récupérer le répertoire de travail**

```
chemin = os.getcwd()
```

- **Lister le contenu d'un dossier**

```
liste = os.listdir('C:/Users/windows 7 fr/Desktop/abc ')
```

Module Os

Opérations sur les fichiers et dossiers

- Renommer un fichier ou un dossier

```
os.rename('ancien_nom.txt', 'nouveau_nom.txt')
```

- Supprimer un fichier

```
os.remove('fichier.txt')
```

- Créer un dossier

```
os.mkdir('/chemin/dossier_vide/')
```

- Vérifier si le fichier ou dossier existe

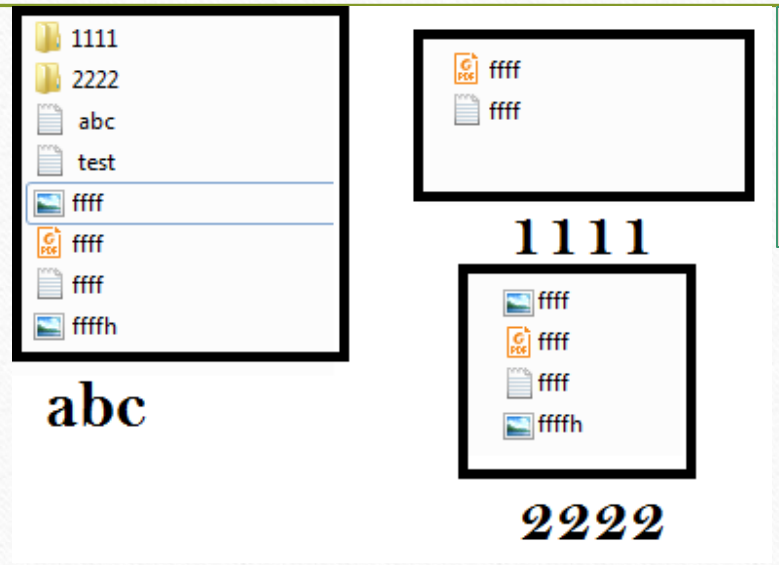
```
if os.path.exists('/chemin/fichier.txt'):
    print('Le fichier existe.')
```

- Vérifier s'il s'agit d'un fichier

```
if os.path.isfile('/chemin/fichier.txt'):
    print("L'élément est un fichier.")
```

Exemple :

On cherche à lister tous les fichiers d'une extension **ext** donnée d'un répertoire ayant un chemin absolue **rep**. La recherche doit aussi portée sur les sous dossiers du répertoire d'une manière récursive.




```
rep='C:/Users/windows 7 fr/Desktop/abc'
ext="txt"
search_file(rep,ext)
```

```
import os
def search_file(rep,ext):
    liste=os.listdir(rep)
    for e in liste:
        if os.path.isfile(e):
            if e[-3:]==ext:
                print(rep+"/"+e)
        else:
            search_file(rep+"/"+e,ext)
```

C:/Users/windows 7 fr/Desktop/abc/ abc.txt
C:/Users/windows 7 fr/Desktop/abc/ test.txt
C:/Users/windows 7 fr/Desktop/abc/1111/ffff.txt
C:/Users/windows 7 fr/Desktop/abc/2222/ffff.txt
C:/Users/windows 7 fr/Desktop/abc/ffff.txt