



PROGRAMMATION WEB AVANCÉE

PYTHON



- Qu'est-ce que Python ?
- Installer Python
- Interpréteur de commandes Python
- Variables
- Structures conditionnelles
- Boucles
- Exceptions
- Listes, Tuples et Dictionnaires
- Programmation Orientée Objet
- Travaux Pratiques

QU'EST-CE QUE PYTHON ?

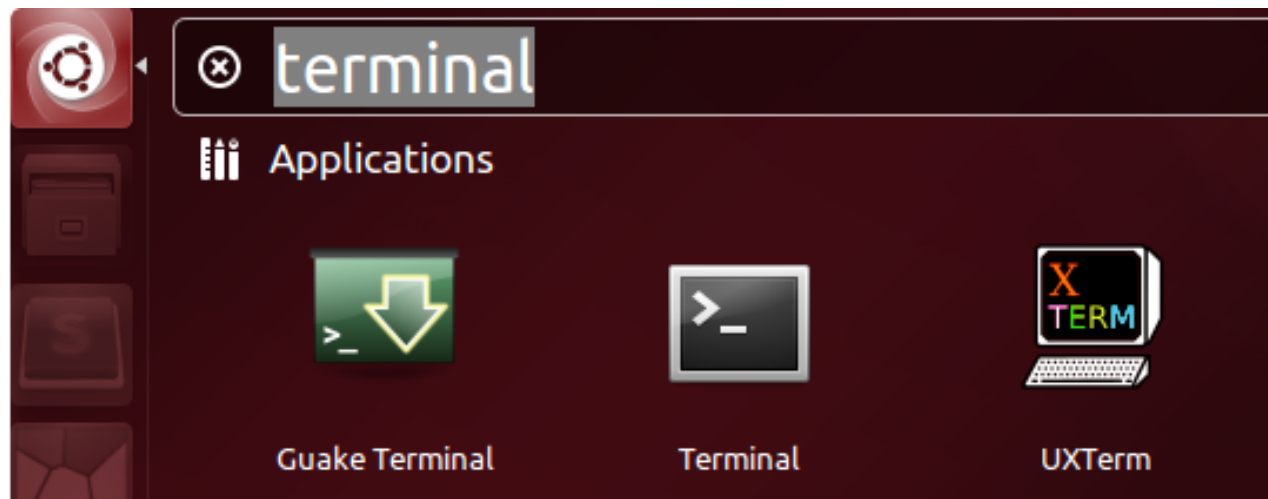
- ▶ Langage de programmation objet interprété
- ▶ Guido Van Russom, 1991
- ▶ Python 2.x, Python 3.x
- ▶ Langage de script pour automatiser des tâches
- ▶ Calcul numérique
- ▶ Développement web
- ▶ Instagram, YouTube, Dropbox, Spotify, Pinterest, ...

INSTALLER PYTHON

- ▶ <https://www.python.org/>
- ▶ Windows : Downloads
- ▶ Linux : « python -V »
- ▶ Mac OS X

INTERPRETEUR DE COMMANDES PYTHON

- ▶ Lancer le terminal, puis lancer la commande « python »



```
olivier@bigone: ~  
olivier@bigone:~$ python  
Python 2.7.6 (default, Mar 22 2014, 22:59:56)  
[GCC 4.8.2] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 
```

```
olivier@bigone: ~  
olivier@bigone:~$ python  
Python 2.7.6 (default, Mar 22 2014, 22:59:56)  
[GCC 4.8.2] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 3+1  
4  
>>> 
```

VARIABLES

- ▶ Une variable est une donnée de votre programme, stockée dans votre ordinateur.
- ▶ En python, il suffit d'écrire « `nom_variable = valeur` »
- ▶ Règles de syntaxes incontournables :
 - ▶ Le nom de la variable ne peut être composée que de lettres, chiffres, majuscules, minuscules et du chiffre souligné
 - ▶ Le nom de la variable ne peut pas commencer par un chiffre
 - ▶ Python est sensible à la casse

VARIABLES – TYPES DE DONNEES

- ▶ int, float, str, ...
- ▶ Fonction : Elle exécute un certain nombre d'instructions déjà enregistrées. « `ma_fonction(param1, param2, ..., param3)`
- ▶ Fonction **type**
- ▶ Fonction **print**

STRUCTURES CONDITIONNELLES

- ▶ L'une des notions les plus importantes en programmation
- ▶ Syntaxe :

 If (condition):

 ...

 elif (condition):

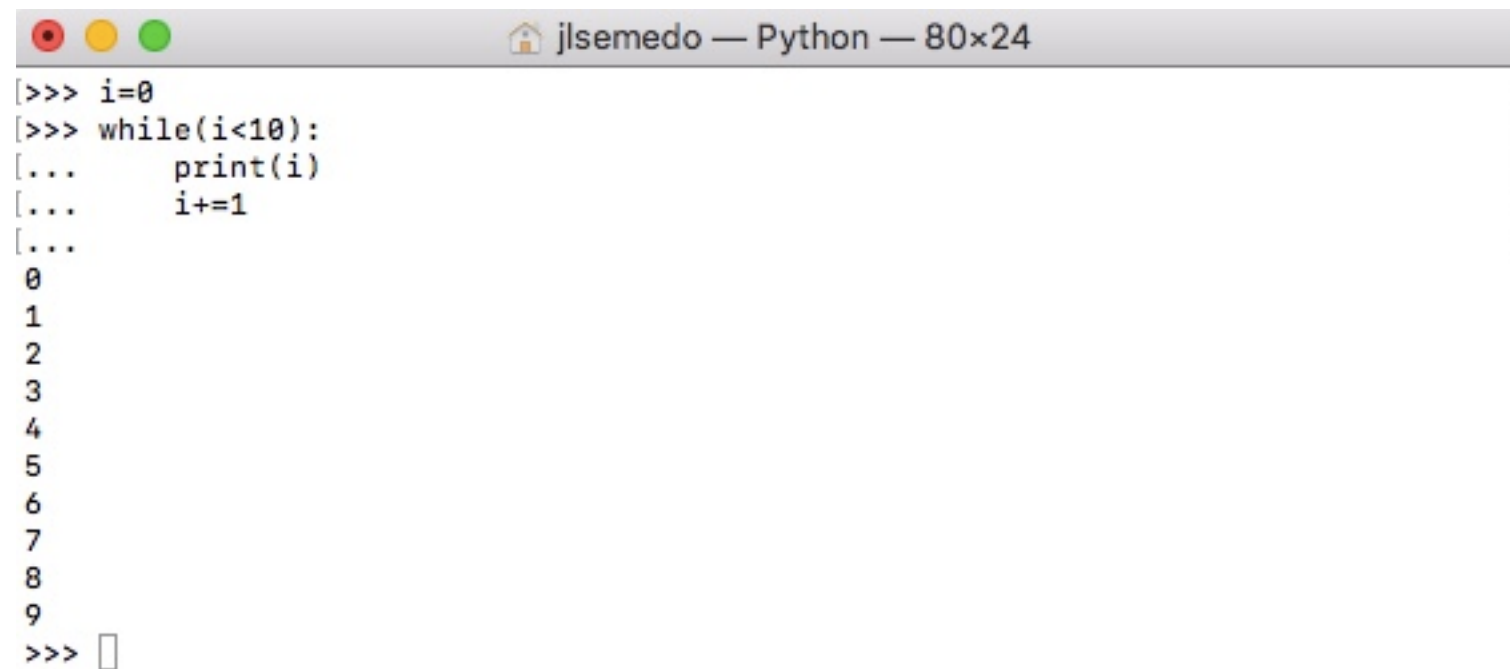
 ...

 else:

 ...

BOUCLES

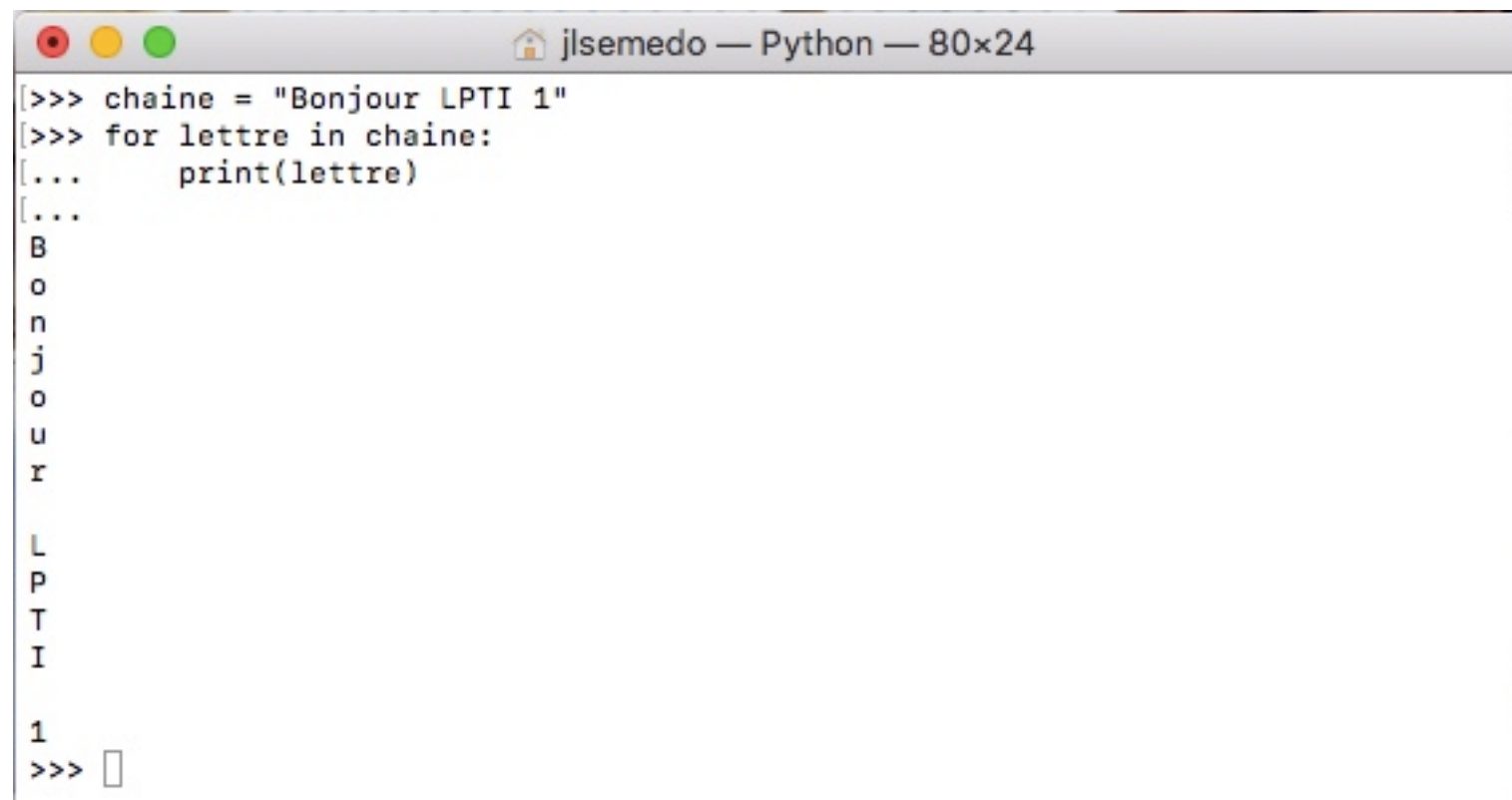
- ▶ Une boucle permet de répéter à l'infini des instructions.
- ▶ Boucle while



```
jlsemedo — Python — 80x24
[>>> i=0
[>>> while(i<10):
[...     print(i)
[...     i+=1
[...
0
1
2
3
4
5
6
7
8
9
>>> ]
```

BOUCLES

► Boucle for



```
jlsemedo — Python — 80x24
[>>> chaine = "Bonjour LPTI 1"
[>>> for lettre in chaine:
[...     print(lettre)
[...
B
o
n
j
o
u
r

L
P
T
I

1
>>> ]
```

EXCEPTIONS

- ▶ En python, on a **les erreurs de syntaxe et les exceptions**
- ▶ Quand Python rencontre une erreur dans le code, il **lève une exception**
- ▶ **Try...Except**
- ▶ **Exemples d'exceptions : NameError, TypeError, ZeroDivisionError**

LISTES, TUPLES ET DICTIONNAIRES

- ▶ Les listes (ou list / array) sont des variables pouvant contenir plusieurs variables
- ▶ Créer une liste : « **liste = []** » ou « **liste = [1, 2, « toto »]** »
- ▶ Ajouter une valeur : « **liste.append(variable)** »
- ▶ Supprimer un item : « **del(liste[0])** »
- ▶ Inverser une liste : « **liste.reverse()** »
- ▶ Nombre d'items d'une liste : « **len(liste)** »

LISTES, TUPLES ET DICTIONNAIRES

- ▶ Un tuple est une liste qui ne peut être modifiée
- ▶ Créer un tuple: « **t = ()** » ou « **t = (1, 2, « toto »)** »
- ▶ Afficher la valeur d'un tuple : « **t[position]** »
- ▶ Les tuples permettent les affectations multiples
- ▶ Il permet également à une fonction de renvoyer plusieurs valeurs

LISTES, TUPLES ET DICTIONNAIRES

- ▶ Un dictionnaire est une liste mais au lieu d'utiliser des index on utilise des clés
- ▶ Créer un dictionnaire: « **d = {}** »
- ▶ Ajouter une valeur : « **d['classe'] = 'LPTI 1'** »
- ▶ Méthode **get(key, default)** permet de récupérer une valeur et si la clé est introuvable, renvoie une valeur par défaut

FONCTIONS ET ESPACES DE NOMS

- ▶ Une fonction est un ensemble d'instructions regroupées sous un nom et s'exécutant à la demande.
- ▶ On doit définir une fonction à chaque fois qu'un bloc d'instructions se trouve à plusieurs reprises dans le code ; il s'agit d'une « factorisation de code ».

FONCTIONS ET ESPACES DE NOMS

La définition d'une fonction se compose :

- Du mot clé **def** suivi de l'identificateur de la fonction, de parenthèses entourant les paramètres de la fonction séparés par des virgules, et du caractère « deux points » qui termine toujours une instruction composée ;
- D'une chaîne de documentation (ou docstring) indentée comme le corps de la fonction ;
- Du bloc d'instructions indenté par rapport à la ligne de définition, et qui constitue le corps de la fonction.

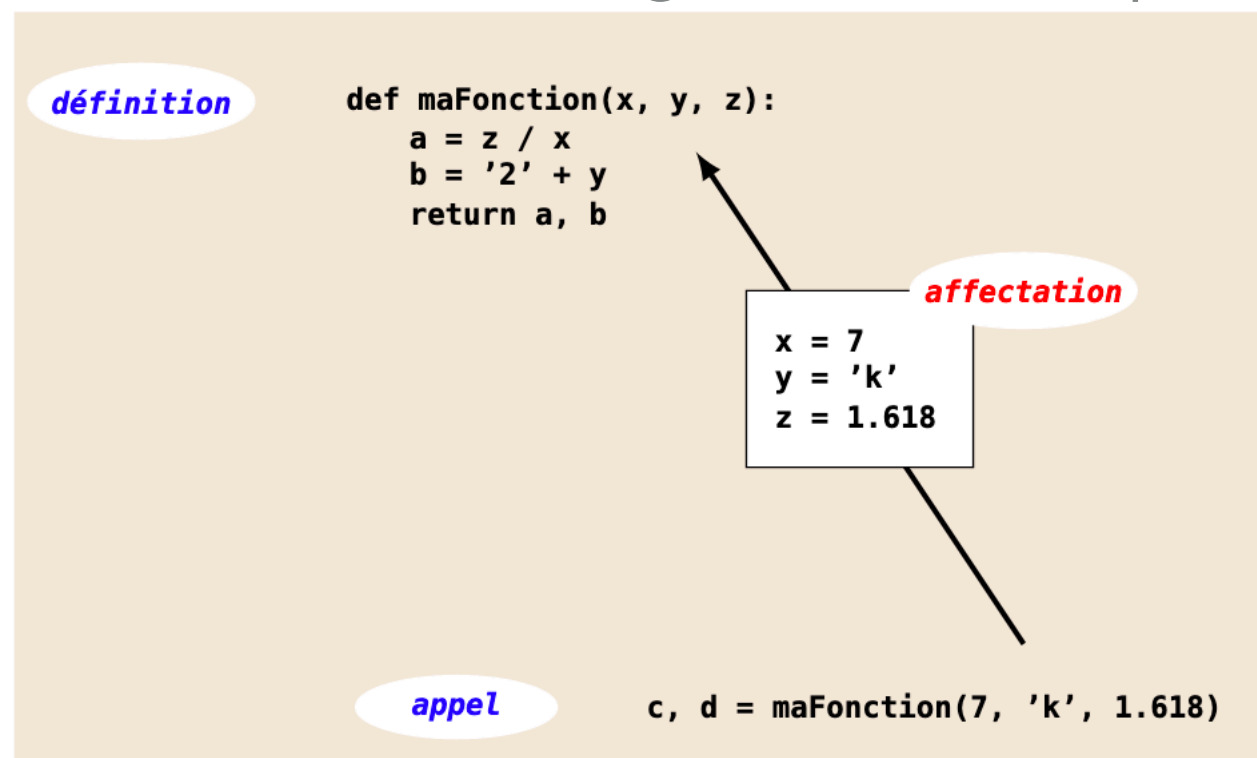
Le bloc d'instructions est obligatoire. S'il est vide, on emploie l'instruction `pass`. La documentation, bien que facultative, est fortement conseillée.


```
def afficheAddMul(a, b):  
    """  
        Calcule et affiche  
        - la somme de a et b  
        - le produit de a et b  
    """  
    somme = a+b  
    produit = a*b  
    print(a, "+", b, "=", somme, ", ", a, "*", b, "=", produit)
```

PASSAGE DES ARGUMENTS

Passage par affectation :

Chaque paramètre de la définition de la fonction correspond, dans l'ordre, à un argument de l'appel. La correspondance se fait par affectation des arguments aux paramètres.



UN OU PLUSIEURS PARAMÈTRES, PAS DE RETOUR

Exemple sans l'instruction **return**, ce qu'on appelle souvent une procédure. Dans ce cas la fonction renvoie implicitement la valeur *None*.

```
def table(base, debut, fin) :  
    """  
        Affiche la table de multiplication des <base> de <debut> à <fin>.  
    """  
    n = debut  
    while n <= fin :  
        print(n, 'x', base, '=', n * base)  
        n += 1  
  
#exemple d'appel:  
table(7, 2, 8)  
#2x7=143x7=214x7=285x7=356x7=427x7=498x7=56  
#  
# autre exemple du même appel, mais en nommant les paramètres;  
table(base=7, debut=2, fin=8)
```

UN OU PLUSIEURS PARAMÈTRES, UN OU PLUSIEURS RETOURS

```
from math import pi

def cube(x):
    """
    Retourne le cube de l'argument
    """
    return x**3

def volumeSphere(r):
    """
    Retourne le volume d'une sphère de rayon r
    """
    return 4 * pi * cube(r) / 3

# Saisie du rayon et affichage du volume
rayon = float(input("Rayon: "))

print("Volume de la sphère=", volumeSphere(rayon))
```

UN OU PLUSIEURS PARAMÈTRES, UN OU PLUSIEURS RETOURS

```
from math import pi

def surfaceVolumeSphere(r) :
    surf = 4.0 * pi * r**2
    vol = surf * r/3
    return surf, vol

rayon = float(input("Rayon:"))
s, v = surfaceVolumeSphere(rayon)
print("Sphère de surface { :g} et de volume { :g}".format(s, v))
```

PASSAGE DE FONCTION EN PARAMÈTRES

```
def f(x):  
    return 2*x+1  
  
def g(x):  
    return x//2  
  
def h(fonc, x):  
    return fonc(x)  
  
h(f, 3)  
# >>> 7  
h(g, 4)  
# >>> 2
```

PARAMÈTRES AVEC VALEURS PAR DÉFAUT

```
def accueil(nom, prenom, depart="MP", semestre="S2"):
    print(prenom, nom, "Département", depart, "Semestre", semestre)

accueil("Student", "Joe")
#Joe Student Département MP semestre S2
accueil("Student", "Eve", "Info")
#Eve Student Département Info semestre S2
accueil("Student", "Steph", semestre="S3")
#Steph Student Département MP semestre S3
```

PASSAGE D'UN TUPLE PAR VALEURS

```
def somme(*args):  
    """Retourne la somme des arguments"""  
    resultat = 0  
    for nombre in args:  
        resultat += nombre  
    return resultat
```

```
somme(23)  
# >>> 23  
somme(23, 42, 12)  
# >>> 77
```


PASSAGE D'UN DICTIONNAIRE PAR VALEURS

```
def unDict(*kwargs):  
    """Retourne la somme des arguments"""  
    return kwargs  
  
# Exemple avec des paramètres nommés  
unDict(a=23, b=14)  
# >>> {'a': 23, 'b': 14 }  
  
d = {'d': 67, 'e': 'Toto', 'f': True}  
unDict(**d)  
# >>> {'d': 67, 'e': 'Toto', 'f': True}
```

PORTEE DES OBJETS

▶ **Portée Globale**

- ▶ Celle du module ou du fichier script en cours. Un dictionnaire gère les objets globaux: l'instruction `globals()` fournit un dictionnaire contenant les couples *nom:valeur*

▶ **Portée Locale**

- ▶ Les objets internes aux fonctions sont locales. Les objets globaux ne sont pas modifiables dans les portées locales. L'instruction `locals` fournit un dictionnaire contenant les couples *nom:valeur*

PORTEE DES OBJETS

- ▶ Par défaut, tout identificateur utilisé dans le corps d'une fonction est local à celle-ci.
- ▶ Si une fonction a besoin de modifier certains identificateurs globaux, la première instruction de cette fonction doit être :
`$ global <identificateurs>`

PORTÉE DES OBJETS

```
def fonc(y) : # y et z sont affectés dans fonc => locaux
    global x # permet de modifier x
    x = x + 2
    z = x + y
    return z
```

```
x = 99
print(fonc(1)) #102
print(x) #101
```

FONCTIONS INCLUDES

```
def creer_plus(ajout):  
    def plus(increment):  
        return increment + ajout  
    return plus  
  
p = creer_plus(23)  
q = creer_plus(42)  
  
print("p(100) = ", p(100))  
print("q(100) = ", q(100))
```

DECORATEURS

- ▶ Les décorateurs permettent d'encapsuler un appel et donc d'effectuer des *pré-* ou des *post-traitements* lors de l'appel d'une fonction, d'une méthode ou d'une classe.

```
def decorateur():  
    ...  
  
@decorateur  
def fonction(*args):  
    pass
```

```
def monDeco(f):
    cptr = 0
    def _interne(*args, **kwargs):
        nonlocal cptr
        cptr = cptr + 1
        print("Fonction decoree :", f.__name__, ". Appel numero :", cptr)
        return f(*args, **kwargs)
    return _interne

@monDeco
def maFonction(a, b):
    return a + b

def autreFonction(a, b):
    return a + b

print(maFonction(1, 2))
# >>> Fonction decoree : maFonction. Appel numero : 1
# >>> 3
autreFonction = monDeco(autreFonction)
print(autreFonction(1, 2))
# >>> Fonction decoree : autreFonction. Appel numero : 1
# >>> 3

print(maFonction(3, 4))
# >>> Fonction decoree : maFonction. Appel numero : 2
# >>> 7
print(autreFonction(6, 7))
# >>> Fonction decoree : autreFonction. Appel numero : 2
# >>> 13
```

FONCTIONS LAMBDA

- ▶ Les fonctions *lambda* permettent de définir des fonctions anonymes dont le bloc d'instructions est limité à une expression dont l'évaluation fournit la valeur de retour de la fonction.
- ▶ `$ lambda [parameters] : expression`

FONCTIONS LAMBDA

- ▶ Les fonctions *lambda* permettent de définir des fonctions anonymes dont le bloc d'instructions est limité à une expression dont l'évaluation fournit la valeur de retour de la fonction.
- ▶ `$ lambda [parameters] : expression`

```
s = lambda x : "" if x == 1 else "s"

s(3)
# >>> "s"

s(1)
# >>> ""
```

FONCTIONS MAP, FILTER, REDUCE

- ▶ La fonction *map()* applique une fonction à chaque élément d'une séquence et retourne un itérateur.

```
list(map(lambda x: x*2, range(10)))
```

```
# >>> [0, 2, 4, 6, ...]
```

FONCTIONS MAP, FILTER, REDUCE

- ▶ La fonction *filter()* construit et renvoie un itérateur sur une liste qui contient tous les éléments de la séquence initiale répondant à un certain critère.

```
list(filter(lambda x: x>4, range(10)))
```

```
# >>> [5, 6, 7, 8, 9]
```

FONCTIONS MAP, FILTER, REDUCE

- ▶ La fonction *reduce()* applique de façon cumulative une fonction de deux arguments aux éléments d'une séquence, de gauche à droite, de façon à réduire cette séquence sur une seule valeur qu'elle renvoie.

```
from functools import reduce

def somme(x, y):
    return x + y

print(reduce(somme, range(5)))
```

MODULES & PACKAGES

- ▶ Module : Fichier script Python permettant de définir des éléments de programmes réutilisables. Ce mécanisme permet d'élaborer efficacement des bibliothèques de fonctions ou de classes.
- ▶ Lorsqu'on parle du module, on omet l'extension : le module *dar* est dans le fichier *dar.py*.
- ▶ Import : L'instruction *import* charge et exécute le module indiqué s'il n'est pas déjà chargé. L'ensemble des définitions contenues dans ce module deviennent alors disponibles : variables globales, fonctions, classes.

MODULES & PACKAGES

- ▶ `__main__` : Le module principal est celui qui est donné en argument sur la ligne de commande ou qui est lancé en premier lors de l'exécution du script. Son nom est contenu dans la variable globale `__name__`.

```
def cube(x):  
    return x**3  
  
if __name__ == "__main__":  
    if cube(9) == 729:  
        print("Ok!")  
    else:  
        print("Ko!")
```

GESTION DES CHAINES

- ▶ Le module *string* fournit des constantes comme *ascii_lowercase*, *digits*,... ainsi que la classe *Formatter* qui peut être spécialisée en sous-classes de formateurs de chaînes.
- ▶ Le module *textwrap* est utilisé pour formater un texte : longueur de chaque ligne, contrôle de l'indentation.
- ▶ Le module *struct* permet de convertir des nombres, booléens et des chaînes en leur représentation binaire afin de communiquer avec des bibliothèques de bas-niveau (souvent en C).

GESTION DU TEMPS ET DES DATES

- Les modules *calendar*, *time*, *datetime* fournissent les fonctions courantes de gestion du temps et des durées.

```
import calendar, time, datetime

mon_apollo11 = datetime.datetime(1969, 7, 20, 20, 17, 40)
print(mon_apollo11)
print(time.asctime(time.gmtime(0)))

mercredi_dernier = datetime.date.today()
un_jour = datetime.timedelta(days=1)

while mercredi_dernier.weekday() != calendar.FRIDAY:
    mercredi_dernier -= un_jour

print("Mercredi dernier : ", mercredi_dernier.strftime("%d-%M-%Y"))
```


P00 – CLASSES

```
Voiture
1  #!/usr/bin/env python3.4
2  # _ coding: utf-8 -*-
3
4  class Voiture:
5
6      def __init__(self):
7          self.nom = "Car rapide"
8
9      def get_nom(self):
10         return self.nom
11
12     def set_nom(self, nom):
13         self.nom = nom
```

- ▶ Classe « Voiture »
- ▶ La méthode « `__init__()` » est appelée à la création d'un objet
- ▶ « nom » est un attribut de classe
- ▶ « `get_nom` » et « `set_nom` » sont des méthodes
- ▶ « `self.nom` » est une manière de stocker une information dans la classe

P00 – OBJETS

- ▶ Pour créer un objet, on fait « `mon_objet = Voiture()` »
- ▶ Pour donner une valeur à l'attribut « `nom` », on a 2 façons de procéder dans ce cas précis :
 - ▶ `mon_objet.nom = « LPTI 1 »`
 - ▶ `mon_objet.set_nom(« LPTI 1 »)`
- ▶ Fonction **dir** donne un aperçu des méthodes d'un objet

TEXTE
