

Environnement de développement Python

Version : Python 3.x.x

Référence : La maison mère

<https://www.python.org/doc/versions/>



Le langage Python

développé en 1989 par Guido van Rossum

open-source

Portable

orienté objet

dynamique

- Comment faire fonctionner un code source Python ?



- Versions de Python :
 - Il existe 2 versions de Python : 2.x et 3.x.
 - Nous nous intéresserons uniquement à Python 3.x

Les nombres

- **Opérateurs de base dans les entiers** : $+$ $-$ $*$ $//$ $\%$ $**$
 - $a // b$ fournit le quotient de l'entier a par l'entier $b \neq 0$
 - $a \% b$ fournit le reste de la division de l'entier a par l'entier $b \neq 0$
 - $**$ permet de calculer une puissance :
- l'ordre du calcul d'une expression arithmétique tient compte de la **priorité** de chaque opérateur :

$+$ $-$	$*$ $//$ $\%$	$**$
même priorité (faible)	même priorité (haute)	(très haute)

Les variables

- Une variable devra être **connue** et contenir une **valeur**

Variable = expression

```
>>> a = 2 # a prend pour valeur 2
>>> p = 10 # p prend pour valeur 10
>>> c = a ** p # c prend pour valeur ...
>>> c
1024
```

a	→	2
p	→	10
c	→	1024

- On dit que : $a = 2 + 3$ est une **instruction d'affectation**
- Une variable peut **changer de valeur**

```
>>> a = 2 + 3
>>> a
5
```

Expression et instructions

- Ne confondez pas les **expression** et les **instructions** qui toutes deux vont contenir des variables.
- Une expression sera calculée , tandis qu'une instruction sera exécutée

Expression :

$a * x ** 2 \rightarrow 45$

```
>>> a * x ** 2  
45
```

a vaut 5
x vaut 3

Instruction:

$c = a * x ** 2 \rightarrow$ *Aucun résultat*

```
>>> c = a * x ** 2  
>>> |
```

a vaut 5
x vaut 3

Affichage de résultats avec print(...)

- La fonction **print()** permet d'afficher une suite d'expression

```
>>> k = 2 ** 10
>>> print ( 'le kilo informatique est ', k, ' et non 1000')
le kilo informatique est 1024 et non 1000
```

- Ce qui est affiché n'est pas le résultat de la fonction print, mais l'effet de cette fonction. La fonction print n'a aucun résultat ! Ou plutôt son résultat est **None**.

- Vérifions :**

```
>>> print (k) == None
1024 <
True <
>>> x = None
>>> x
>>> |
```

test d'égalité

l'effet de print (k)

le résultat du test d'égalité

None est une valeur spéciale signifiant "**rien**".

Lecture des données avec input(...)

- La fonction **input()** renvoie systématiquement une chaîne de caractères.
- Si c'est un nombre à lire et que l'on veut manipuler, il faudra utiliser la fonction **int()** s'il s'agit d'un entier ou **float()** s'il s'agit d'un réel.

Exemple :

```
chaine = input ( )  
nb = float ( chaine )
```

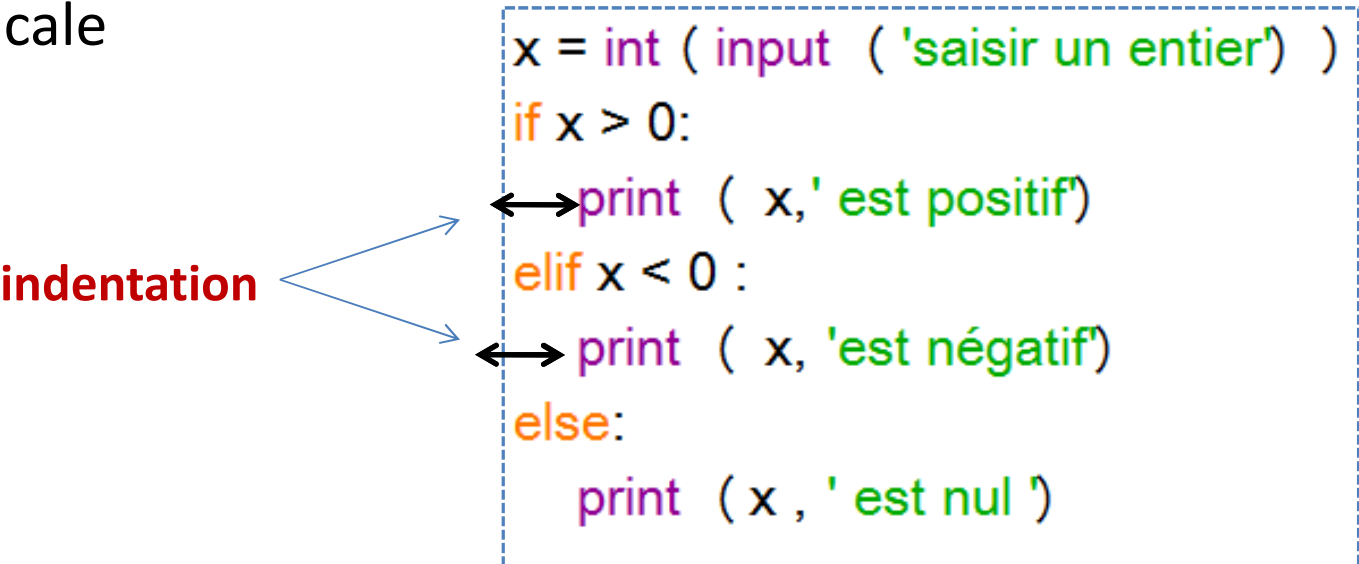
Ou bien directement :

```
nb = float ( input ( ) )
```

Indentation

- La bonne distance à la marge d'une ligne (indentation) doit être respecté. Elle permet de structurer et de comprendre un programme:
- Un bloc d'instruction est une suite d'instructions alignées à la verticale

indentation



```
x = int ( input ( 'saisir un entier' ) )
if x > 0:
    print ( x, ' est positif' )
elif x < 0 :
    print ( x, 'est négatif' )
else:
    print ( x , ' est nul ' )
```

Une bonne indentation est obligatoire en Python.

La boucle while

- Cette structure de contrôle fait répéter une séquence d'instructions, aussi longtemps qu'une condition <test> est vrai (True).
- Il est essentiel dans une boucle **while** de s'assurer que la boucle termine, donc que le <test> finit par prendre la valeur **False**. Sinon le programme entre dans une *boucle infinie* et...
On perd la main, il plante !
- Pourtant certains programmeurs optent pour un style de **boucle infinie** dans lequel la décision d'échappement est prise parmi *les* instructions du corps de la boucle avec l'instruction **break**

```
while <test> :  
    <instruction>  
    <instruction>  
    .....
```

```
while True :  
    <instruction>  
    <instruction>  
    .....
```

La boucle while

Exemple :

```
x = 1
while x <= 5 :
    print (x)
    x = x + 1
```



```
x = 1
while True :
    if x > 5 :
        break
    print (x)
    x = x + 1
```

- L'instruction **break** provoque une *sortie brutale de la boucle*, mais le *programme* continue son exécution après la boucle.

La boucle **for** sur un intervalle ou un itérable

- Un intervalle se dit *range en anglais*

```
s = 'Bonjour etudiants pc 14 '  
c = 'u'  
# calcul du nombre d'occurrence de c dans s  
occ = 0  
for i in range ( len ( s ) ) :  
    if s[i] == c :  
        occ +=1  
print ( c , ' figure ', occ, ' fois dans ', s)
```

la variable locale **i** parcourt
l'intervalle 0.. len(s) - 1.

mieux

```
s = 'Bonjour etudiants pc 14 '  
c = 'u'  
# calcul du nombre d'occurrence de c dans s  
occ = 0  
for car in s:  
    if car == c :  
        occ +=1  
print ( c , ' figure ', occ, ' fois dans ', s)
```

- Si a,b,n sont entiers :
 - range(a,b) signifie a .. b-1 avec a < b
 - range(n) dénote l'intervalle 0..n-1 avec n > 0

Les chaines de caractère

- Sous Python, une donnée de type **string** est une suite de caractères délimitée soit par :

```
>>> phrase1 = 'les œufs durs.'  
>>> phrase2 = "j'aime bien "  
>>> phrase3 = ' "N\'est-ce pas ?" répondit-elle.'  
>>> phrase4 = = """"  
... C'est plus facile sans l'antislash !  
... On peut écrire des antislashes : \  
""""  
>>> print (phrase1, phrase2, phrase3, phrase4)
```

Simple quote

Double quote

Triple quote

Les chaines de caractères sont ordonnées et non mutables

```
ch = "Mohamed Amin"  
print (ch[0], ch[5])  
ch[2]='A'
```

→ 'M', 'e'
→ erreur

Opérations sur les chaînes de caractère

Fonctions et méthodes des chaînes	sa, sb sont des objets chaînes (str)
sa=chr(n),	Renvoie un caractère dont on connaît le code ASCII n
n=ord(sa)	Renvoie le code ASCII n d'un caractère sa .
sa.count(sb,*deb,*fin)	Compte le nombre de fois où sb est dans sa entre les indices deb et fin .
sa.find(sb ,*deb,*fin)	Retourne l'indice de la première apparition de sb dans sa entre les index deb et fin.
sa.replace(st_anc ,st_nouv,*n)	Remplace les n premières apparitions de st_anc par st_nouv.
sa.split(sep, *n)	Sépare sa en n coupures maximum au niveau de chaque chaîne sep.
sa.lsplit(*char)	Supprime tous les 'blancs' ou caractère char si précisé dans sa.

Opérations sur les chaînes de caractère

Fonctions et méthodes des chaînes	sa, sb sont des objets chaînes (str)
sa.strip(*s)	Supprime tous les espaces au début et à la fin de sa. Si s est donné, tous les caractères faisant partie de s sont supprimés de sa.
sa.upper() sa.lower()	Change la casse minuscules vers majuscules ou majuscules vers minuscules de sa.
sa.capitalize()	Transforme en majuscule le premier caractère de la chaîne sa.
sa.join(Liste_mots)	Crée une chaîne à partir des mots de Liste_mots, séparés par sa.
Formatage	Exemple : <pre>for i in range(1 , 5) : print ('5 * {} = {}'.format(i,5*i))</pre>

Les listes : List

Une liste est une structure de données qui contient une série de valeurs et/ou chaînes de caractères, séparés par des virgules et limités par des crochets.

```
>>> animaux = ['girafe','tigre','singé','souris']  
>>> tailles = [5, 2.5, 1.75, 0.15]  
>>> mixte = ['girafe', 5, 'souris', 0.15]
```

Les listes sont ordonnées et mutables

liste	:	['girafe', 'tigre', 'singé', 'souris']			
indice positif :		0	1	2	3
indice négatif :		-4	-3	-2	-1

```
>>> animaux = ['girafe','tigre','singé','souris']  
>>> animaux[1] → 'tigre'  
>>> animaux[0]='lion'  
>>> animaux → ['lion','tigre','singé','souris']
```

Opérations sur les listes

Fonctions et méthodes des listes	Liste vide : $L = []$, numérotée à partir de 0 et de longueur $\text{len}(L)$.
$L.\text{append}(\text{élément})$	Ajoute élément à la liste L. Peut aussi s'écrire $L = L + [\text{élément}]$.
$L.\text{insert}(\text{index}, \text{élément})$	Insère élément à la position index.
$L.\text{extend}(K)$	Ajoute tous les éléments de la liste K à la liste L. On peut aussi utiliser les opérateurs + et *
$L.\text{remove}(\text{élément})$ $L.\text{pop}(*\text{index})$	Supprime de L ; erreur si élément n'est pas présent. Supprime l'élément n° index; si index n'est pas indiqué, le dernier élément est enlevé.
$L.\text{index}(\text{élément})$	Indique la position de élément; si pas présent, une erreur se produit.
$L.\text{sort}()$ $M = \text{sorted}(L)$	Tri de L. Retourne la liste L triée dans M.
$\text{élément in } L$ $\text{élément not in } L$	Booléen disant si élément est dans L

Manipulation de tranche (ou sous chaine)

- Un avantage des listes est la possibilité de sélectionner une partie en utilisant un indigage construit sur le modèle `[m:n+1]` pour récupérer tous les éléments, du $m^{\text{ème}}$ au $n^{\text{ème}}$ élément (de l'élément m inclus à l'élément $n+1$ exclus).

➡ On dit alors qu'on récupère une tranche de la liste

- Remarquez que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole : Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.
- On peut aussi préciser le **pas** en ajoutant un `:` supplémentaire et en indiquant le pas par un entier.

Liste [début : fin : pas]

Exemple :

```
s = 'Rayon X'
print ( s [ 1 : 4 ] ) # de l'index 1 compris à 4 non compris
print ( s [ 3 : ] )   # de l'index 3 compris à la fin
print ( s [ : : 2 ] ) # du début à la fin, de 2 en 2
```

Les tuples

- Les **tuples** correspondent aux listes à la différence qu'ils sont non modifiables.
- Pratiquement, ils utilisent les **parenthèses** au lieu des crochets.
- Ce type de donnée possède les mêmes méthodes que la liste, à l'exception de celles permettant une modification.
- **Exemple :**

```
>>> x = (1,2,3)
>>> x = 1,2,3
>>> len(x)
>>> x[0:2]
>>> x + (3,)
```

Les tuples sont ordonnées et non mutables

```
>>> x[2]      —————> 3
>>> x[2] = 15 —————> Erreur
```

Les opération sur les tuples

Fonctions et méthodes des tuples	Documentation
<code>tuple(itérable)</code>	La fonction <code>tuple()</code> fonctionne exactement comme la fonction <code>list()</code> , c-à-d qu'elle prend en argument un objet itérable et renvoie le tuple correspondant.
<code>count(obj)</code>	retourne le nombre d'occurrences de <code>obj</code> dans <code>T</code>
<code>index(obj)</code>	retourne la position de la première occurrence de <code>obj</code> dans <code>T</code>

Les ensembles : set

- Le **set** est un type Python permettant de créer des collections, formées par des éléments **non dupliqués**.
- Son grand intérêt réside dans le fait qu'on peut appliquer à ces objets des opérations propres aux ensembles (union, intersection, différence...), d'où le nom de sets (ensembles).
- On crée un set avec la fonction **set**(iterable) ou en énumérant les éléments entre **accolades** {elem1, elem2...}

```
>>> set1 = {'a', 'b', 'a', 'c'}  
>>> type(set1)
```

Les ensembles sont non ordonnées et mutables

```
>>> set1['3']           → Erreur : set not support indexing  
>>> set1.add('d')  
>>> set1               → {'b', 'c', 'd', 'a'}
```

Les opérations sur les ensembles

Fonctions et méthodes des ensembles	sortie
<code>set1 = set(itérable)</code> <code>set2 = set()</code>	La fonction <code>set</code> prend en argument un objet itérable et renvoie l'ensemble correspondant.
<code>set2 = {1, 2}</code> <code>set2.add(3)</code>	Ajout dans un ensemble <code>set2</code> .
<code>set1 set2</code>	<code>set 1 union set2</code>
<code>set1 & set2</code>	Intersection entre <code>set1</code> et <code>set2</code>
<code>set1 - set2</code>	différence
<code>set1 ^ set 2</code>	Différence symétrique
<code>< , <= , > , >=</code>	Relations d'inclusion entre ensembles. <code>{1,2} <= {1,2,3,4}</code> retourne <code>True</code> .

Les dictionnaires : dict

- Les dictionnaires se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites.
- On crée un dictionnaire vide avec les symboles {} ou avec la fonction **dict()**.
- Ensuite, on remplit le dictionnaire avec différentes clés auxquelles on affecte des valeurs (une par clé).
- Pour récupérer la valeur d'une clé donnée, il suffit d'utiliser une syntaxe du style **dictionnaire['cle']**.

```
>>> ani1 = {}  
>>> ani1['nom'] = 'girafe'  
>>> ani1['taille'] = 7.0  
>>> ani1['poids'] = 1200
```

Les dictionnaires sont non ordonnées et mutables

```
>>> ani1['taille']      → 7.0  
>>> ani1['taille'] = 11.0  
>>> ani1 → {nom:'girafe', taille:11.0, poids:1200 }
```

Les opérations sur les dictionnaires

Fonctions et méthodes des dictionnaires	sortie
d.keys()	Renvoie les clés d'un dictionnaire (sous forme de liste)
d.Values()	Renvoie les valeurs d'un dictionnaire (sous forme de liste)
d.Items()	les clés et les valeurs d'un dictionnaire
d.Clear()	Vider le dictionnaire d.
del d[clé]	Supprime l'item ayant la clé clé du dictionnaire.
d.pop(clé)	Renvoie la valeur ayant clé clé et supprime l'item (clé, vlaur) correspondant du dictionnaire.

Compréhension de liste, set et dictionnaires

- Pour construire un container de type liste, set ou dictionnaire à l'aide d'une boucle **for** assortie éventuellement d'une condition, Python offre une construction compacte appelée **compréhension expression**. Sa syntaxe est :

```
liste = [ expression for expr in iterable if cond ]  
set = { expression for expr in iterable if cond }  
dict = { expr1:expr2 for expr in iterable if cond }
```

- S'agissant d'une liste ou d'un set, l'expression évaluée à chaque itération de la boucle, constitue la valeur de l'élément inséré dans la liste ou le set.
- Concernant le dictionnaire dict, les expressions expr1 et expr2 constituent respectivement les clés et valeurs des paires insérées.
- Le test if cond est **facultatif**. Il peut aussi prendre la forme d'une expression conditionnelle :

```
[exprT if cond else exprF for expr in iterable]
```

Compréhension de liste, set et dictionnaires

Exemple:

L'expression ci-dessous fabrique la liste des carrés des nombres pairs de 1 à 10

```
carres = [ nb*nb for nb in range(1,11) if nb%2==0 ]
```

équivalent à :

```
carres = []  
for nb in range(1,11):  
    if nb%2 == 0:  
        carres.append(nb*nb)
```

L'expression ci-dessous retourne, dans l'ordre alphabétique, la liste des car. utilisés dans une chaîne

```
sorted({ car for car in 'abracadabra' })
```

L'expression ci-dessous récupère, dans le dictionnaire dic, tous les éléments dont la valeur est supérieure à 10, et les insère dans le nouveau dictionnaire sup10

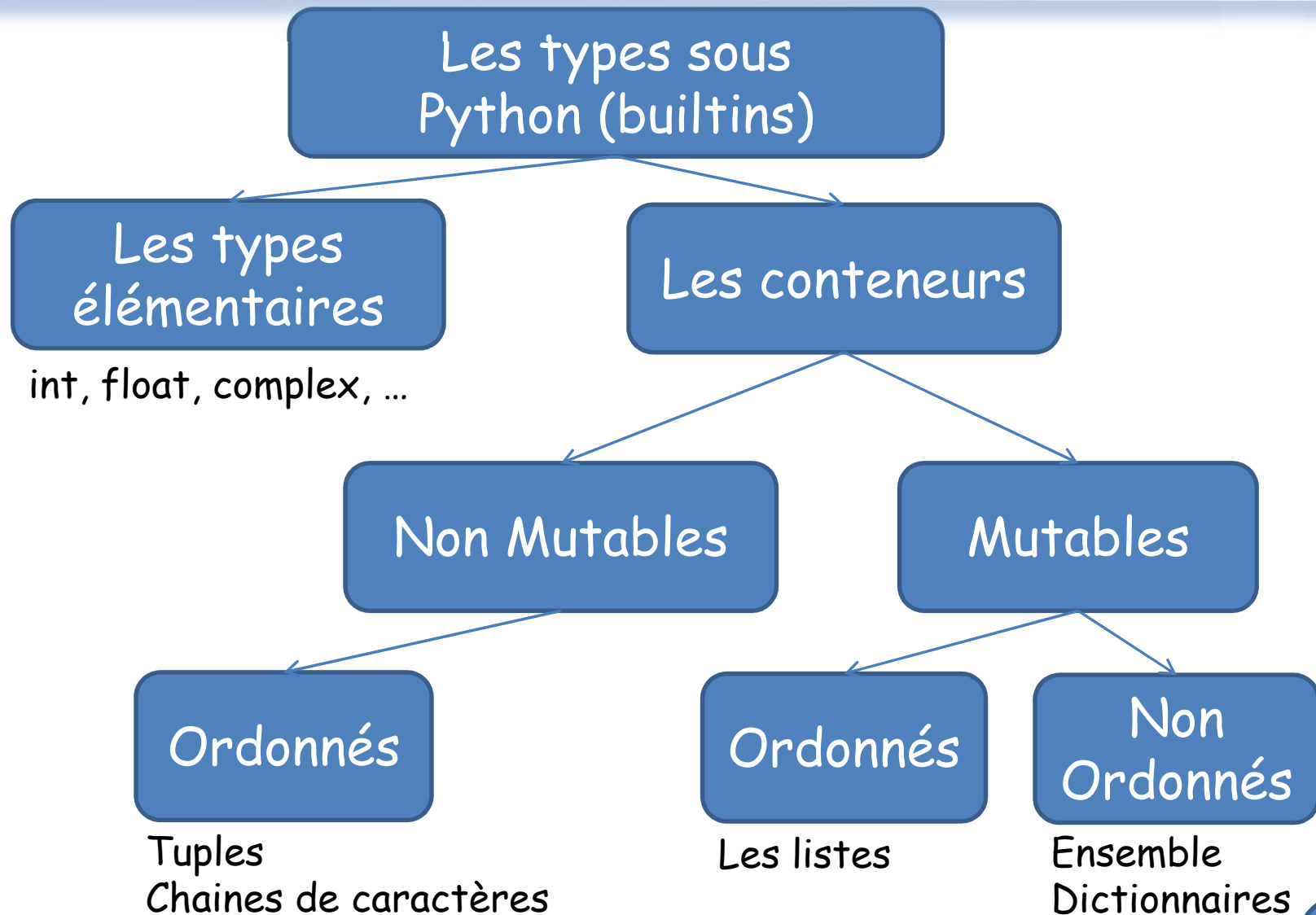
```
dic = { 'a': 12.50, 'b': 3.50, 'c': 11.00, 'd': 6.00 }
```

```
sup10 = { cle:val for cle,val in dic.items() if val>10 }
```

L'expression ci-dessous parcourt les entiers de 1 à 9 et affiche les valeurs impaires, sinon 0

```
[ x if x%2 else 0 for x in range(1,10) ]
```

Les types sous Python



Définir une nouvelle fonction

Définir une fonction

```
def nomFonction( paramètres ) :  
    # instruction 1  
    # instruction 2  
    # ...  
    # instruction n
```

- ☐ Lors de la définition d'une fonction, le code n'est pas exécuté;
- ☐ A tout moment dans le programme ou dans une fonction on peut appeler la fonction avec la commande suivante :

Appeler une fonction

```
nomFonction( paramètres )
```

Définir une nouvelle fonction

- Exemple :
 - Soit à définir la fonction $f : n \rightarrow 2n - 1$

```
def f ( n ) :  
    return 2 * n - 1  
print ( 'f ( 5 ) vaut', f ( 5 ) )
```



Run

```
>>>  
f ( 5 ) vaut 9
```

- Notez ***l'indentation***
- Le mot ***return*** signifie "le résultat est...". On dit que la fonction retourne un résultat
- On peut aussi utiliser f sur les réels approchés voire même sur les complexes :

```
>>> f ( 5.2 )  
9.4
```

```
>>> f ( 2 + 3j ) # j au lieu de i  
( 3+6j)
```

Définir une nouvelle fonction

- ❑ L'instruction **return** x interrompt l'exécution de la fonction et renvoie la valeur x;
- ❑ x peut être de n'importe quel type;
- ❑ On récupère la valeur renvoyée normalement, par exemple par une affectation :

y = maFonction(...)

- ❑ On peut aussi l'utiliser dans une expression où elle est évaluée

y = maFonction() + 3
print(maFonction())

- ❑ Par défaut, s'il n'y a pas de **return** ou si on met **return** simplement sans arguments après, la fonction renvoie **None**;

Portée des variables dans une fonction

```
def f(n):  
    n = n + 1  
    n = 3  
    f(n)  
    print('n vaut', n)
```

- ❑ Le résultat est **n vaut 3** !
- ❑ A l'appel de la fonction, la valeur du paramètre de **f** est affectée au **n** de la définition de **f**;
- ❑ Ce n'est pas le même **n**;
- ❑ Il y a le **n** principal ou global, et le **n** de **f** qu'on va noter **n_f** ;
- ❑ **n_f** prend la valeur de **n** à l'appel de **f** et est incrémenté de 1 dans la fonction.

Portée des variables dans une fonction

```
def f(x):  
    n = 1
```

```
# programme principale  
n = 3  
f(n)  
print('n vaut', n)
```

- ❑ Le résultat est toujours **n vaut 3** !
- ❑ A l'appel de la fonction, la valeur du paramètre de f est affectée au x de la définition de f;
- ❑ L'affectation dans la fonction crée une variable locale à f, notons-la **n_f**; **n_f** prend la valeur de 1 et le **n** global ne change pas.

Portée des variables dans une fonction

```
def f(n):  
    global k  
    i = n  
    k = 0  
    print( i , j , k )
```

programme principale

```
i = 2  
j = 4  
k = 6  
f(44)  
print(i, j, k)
```

- ❑ Dans le corps de la fonction n et i sont des variables **locales**;
- ❑ k est une variable **globale** modifiable;
- ❑ j est visible en tant que variable **globale** (mais non modifiable);

Portée des variables dans une fonction

- ☐ Les paramètres de la définition de la fonction sont des variables locales
- ☐ Les variables affectées dans la fonction sont des variables locales
- ☐ Ces variables locales existent pendant l'exécution de la fonction et n'existent plus après;
- ☐ Les variables affectées dans le corps du programme (hors fonctions) sont des variables globales;
- ☐ Les variables globales sont lisibles dans tout le programme;
- ☐ Les variables globales ne sont pas modifiables dans une fonction: si on affecte une variable globale dans une fonction, on crée une variable locale avec le même nom;
- ☐ Pour modifier une variable globale x dans une fonction, il faut la déclarer avec le mot clé *global*;

Définir une nouvelle fonction lambda

Fonction lambda

- Pour les fonctions dont le corps ne contiendrait qu'une seule instruction « return expression », Python offre la possibilité de définir une fonction anonyme aussi appelée **lambda** (ou lambda expression). Sa syntaxe est :

lambda arguments ... : expression

```
# La lambda fonction ci-dessous :  
somProd = lambda n1, n2 : (n1+n2, n1*n2)
```

Équivalent à

```
def somProd(n1, n2):  
    return (n1+n2, n1*n2)
```

Run

```
>>> somProd ( 3, 10)  
( 13, 30)
```

Définir une nouvelle fonction récursive

Fonction récursive

- Deux propriétés sont à respecter pour s'assurer du bon fonctionnement d'une fonction/procédure récursive :
 - elle doit contenir un cas de base (**condition d'arrêt**)
 - elle doit s'appeler elle-même, en modifiant son état, pour pouvoir se ramener au cas de base.
- Dans l'exemple de la fonction ***fact*** :
 - Un cas de base $\text{fact}(0)=1$
 - Un appel à $\text{fact}(n-1)$ qui va de proche en proche nous ramener à $\text{fact}(0)$.

```
def fact ( n) :  
    if n == 0:  
        return 1  
    else:  
        return n * fact ( n-1)
```

Définir une nouvelle fonction récursive

Fonction récursive

```
def fonction_réursive (paramètre):  
    if ... :                # close d'arrêt  
        return ...          # cette ligne détermine le retour de la  
                             # variable de sortie dans le cas de base  
    elif .... :             # cette ligne est ajoutée que lorsqu'on  
                             # a une autre condition d'arrêt  
        return ...  
    else :  
        return ...          # instructions contenant un appel à la  
                             # fonction_réursive
```

Définir une nouvelle fonction récursive

Exemple: fact(4)

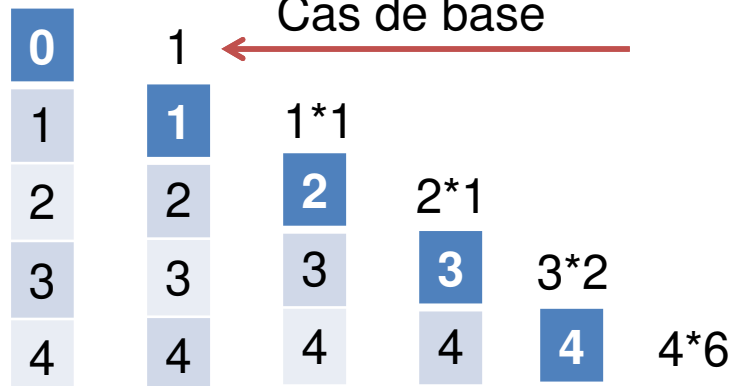
Condition d'arrêt →

Descente



Montée

← Cas de base



Appel à fact(4)

```

. 4*fact(3) = ?
. Appel à fact(3)
.   3*fact(2) = ?
.   Appel à fact(2)
.     2*fact(1) = ?
.     Appel à fact(1)
.       1*fact(0) = ?
.       Appel à fact(0)
.       Retour de la valeur 1
.       1*1
.       Retour de la valeur 1
.       2*1
.       Retour de la valeur 2
.       3*2
.       Retour de la valeur 6
.       4*6
Retour de la valeur 24
    
```

Fonctions prédéfinies de Python

- Python fournissent un large ensemble de fonctions prêtes à être utilisées :

Fonctions	Sortie
id(obj)	adresse mémoire de obj
type(obj)	donne le type de obj
float(x)	convertit int ou string en float
int(x)	convertit float ou string en int
str(obj)	convertit obj en chaîne de caractères
max(s), min(s)	valeur maximale, minimale dans la séquence s
abs(x)	valeur absolue de x
pow(x,y)	$x^{**}y$
sum(s)	somme de la séquence numérique s
len(s)	nombre d'objets dans la séquence s
range(x)	liste de x entiers allant de 0 à x-1

Fonctions prédéfinies de Python

- Certaines fonctions résident dans des **modules** spécialisés, comme fractions, math, random, os, time, calendar, ...

- **Exemples :**

Calculer le pgcd:

```
>>> import fractions
>>> fractions.gcd ( 18,12)
6
>>> from fractions import gcd
>>> gcd ( 18,12)
6
```

Générer des entiers au hasard :

```
>>> import random as rd
>>> rd.randint ( 1,10)
9
>>> rd.randint ( 1,10)
6
>>> rd.randint ( 1,10)
10
```

Les modules

- Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi bibliothèques ou libraires).
- **Importation des modules :**
 - **import <nom du module > :**

```
>>> import random
>>> random.randint(0,10)
```

L'instruction import permet d'accéder à toutes les fonctions du module random. Ensuite, nous utilisons la fonction (ou méthode) randint(a,b) du module random

```
>>> import random as rand
>>> rand.randint(1, 10)
```

il est également possible de définir un alias (un nom plus court) pour un module
 - **from <module> import <fonction> :**

```
>>> from random import randint
>>> randint(0,10)
```

*À l'aide du mot-clé **from**, on importe une fonction spécifique d'un module donné.*

```
>>> from random import *
```

On peut également importer toutes les fonctions d'un module

Les phénomènes exceptionnels

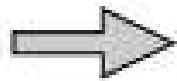
- ❑ Vous savez qu'il peut y avoir des **erreurs** dans un calcul, dus à la mauvaise gestion des cas particuliers :

```
>>> L = [6, 4, 7, 2, 1, 8]
```

```
>>> L[6]
```

```
IndexError: list index out of range
```

IndexError



```
>>> L.index(5)
```

```
ValueError: 5 is not in list
```

ValueError

```
>>> 2 / 0
```

```
ZeroDivisionError: division by zero
```

ZeroDivisionError

- ❑ En fait, ces erreurs portent le nom d'exceptions.
- ❑ Le programmeur peut gérer ces cas exceptionnels et reprendre la main pour proposer un traitement exceptionnel !

Les phénomènes exceptionnels

- ❑ Exemple : programmer la fonction `chercher ( x , L )` qui retourne la position de la première occurrence de `x` dans `L`, ou bien `-1`.

```
def chercher(x, L):  
    for i in range(len(L)):  
        if L[i] == x :  
            return i  
    return -1
```

```
>>> L = [3,2,8,6,2,4,7,6,5]  
>>> chercher(6,L)  
3
```

- ❑ Or Python fournit la méthode ***index*** qui demande à une liste `L` la position d'un élément. Hélas elle ne renvoie pas `-1` mais **lève une exception** en cas d'échec ! Comment transformer cette erreur en `-1` ?

- ❑ **Réponse** : en traitant l'exception, en essayant de la capturer au passage pour proposer un traitement de ce cas exceptionnel.

```
>>> L.index(6)  
3  
>>> L.index(9)  
ValueError
```

Les phénomènes exceptionnels

- ❑ Si j'invoque brutalement la méthode index en demandant ***L.index(x)***, je risque de provoquer une exception !

➡ Je vais simplement essayer de faire le calcul et regarder ensuite si j'ai fait une erreur...

- ❑ En Python, on utilise la construction **try: ... except: ...**

```
def cherche(x,L) :  
    try :  
        return L.index(x)  
    except :  
        return -1
```

```
try :  
    <instr1>  
    ...  
except :  
    <instr2>  
    ...
```

- ❑ Si au cours de l'exécution du bloc d'instructions <instr 1 >... Une exception se produit, l'exécution du bloc est abandonnée et le bloc <instr 2 >... est exécuté.
- ❑ Si l'exécution du bloc <instr 1 >... s'est déroulée normalement, le bloc except n'est pas utilisé.

Les phénomènes exceptionnels

❑ **Exemple** : Je souhaite convertir une chaîne comme "5634" en un entier, ou bien "45.8084" en un flottant. Facile, Python me propose les fonctions `int(...)` et `float(...)`. Elles provoquent des exceptions :

```
>>> float('56.78')
56.78
>>> float('trois')
ValueError: could not convert string to float: 'trois'
```

❑ **Conclusion** :

- les exceptions permettent d'exprimer le comportement général d'un calcul tout en sachant qu'il pourra y avoir quelques cas d'erreurs exceptionnelles qui seront traités à part.
- Il y a différentes exceptions : `ValueError`, `ZeroDivisionError`, `KeyError`, etc. Un bloc `except:` les attrape toutes.
- Il est possible de n'attraper que certaines exceptions, ou d'envisager des traitements spécialisés pour chaque exception...

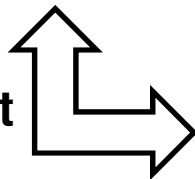
```
try :
    ....
except KeyError :
```

Les phénomènes exceptionnels

- L'instruction **raise** permet de lever volontairement une exception. On peut trouver l'instruction à tout endroit du code, pas seulement dans un bloc try
- l'instruction **assert** permet de déclencher une exception si la condition est fausse : **assert (<condition>)**

```
def KelvinToFahrenheit ( Temperature) :  
    try:  
        assert ( Temperature >= 0)  
        return ( ( Temperature-273) *1.8) +32  
    except :  
        print ( 'erreur : temperature négatif ')
```

équivalent



```
def KelvinToFahrenheit ( Temperature) :  
    try:  
        if Temperature < 0:  
            raise  
        return ( ( Temperature-273) *1.8) +32  
    except :  
        print ( 'erreur : temperature négatif ')
```

Classe de complexité

Définitions :

Afin de comparer plusieurs algorithmes résolvant un même problème, on introduit des mesures de ces algorithmes calculant leurs coûts, appelées **complexité**.

 Chercher l'algorithme dont la complexité est minimale.

❑ **Complexité temporelle** : c'est le nombre d'opérations élémentaires effectuées par une machine qui exécute l'algorithme.

❑ **Complexité spatiale** : c'est le nombre de positions mémoire utilisées par une machine qui exécute l'algorithme.

Algorithme:
Problème

Complexité?

Propriétés :

Ce que nous intéresse:

- ❑ Ressources : nombres d'instructions (complexité temporelles)
- ❑ En fonction du paramètre **n** , Complexité = **C(n)**
- ❑ Asymptotique $\uparrow$ **n** : ordre de grandeur **O(n)**

$$C(n) = n^2 + bn + c = \mathbf{O} (n^2)$$

$$C(n) = 2^n + bn^2 + c = \mathbf{O} (2^n)$$

- ❑ Pire des cas

Algorithme:
Entrée = **n**

Complexité constante :

Script Python:

Entrée = **n**

```
def f(n) :  
    k = 1  
    for i in range(10) :  
        k += 1  
        s += n  
    print(s)  
    return k
```

Exemples:

❑ Vérifier la parité d'un nombre entier n .

Complexité ne dépend pas
de l'entrée n:

$$C(n) = (2 \times 10) + 3 = 11$$

$$C(n+1) = (2 \times 10) + 3 = 11$$

➡ **Complexité est $O(1)$**

Complexité linéaire:

Script python

Entrée = **n**

```
def g (n) :  
    k = 1  
    for i in range(n) :  
        k += 1  
        s += n  
    print (s)  
    return k
```

Complexité dépend de l'entrée n:

$$C(n) = 2*n + 3$$

$$C(n+1) = 2*(n+1) + 3 = C(n) + 2$$

→ **Complexité est $O(n)$**

Exemples:

- ❑ Chercher un élément dans une liste non ordonné de taille n
- ❑ Calculer la longueur d'une chaîne de caractères

Complexité polynomiale :

Script python

Entrée = **n**

```
def G (n) :  
    k = 1  
    for i in range(n) :  
        for j in range(n) :  
            k += 1  
            s += n  
    print (s)  
    return k
```

Complexité:

$$C(n) = 2 * n * n + 3 = 2n^2 + 3 \approx n^2$$

$$\begin{aligned} C(n+1) &= 2 * (n+1) * (n+1) + 3 \\ &= 2n^2 + 4n + 5 \\ &\approx C(n) + n \end{aligned}$$

Complexité est $O(n^2)$

Exemples:

- ❑ chercher le maximum dans une matrice carrée de taille n

Complexité logarithmique:

Script python:

Entrée = **n**

```
def h(n) :  
    m = n  
    i = 1  
    while m > 0 :  
        i += 1  
        m = m // 2  
    print ( n , i )  
    return i
```

Complexité:

n	i	m
8	1, 2, 3	8, 4, 2, 0

n	i	m
16	1, 2, 3, 4	16, 8, 4, 2, 0

$$C(2n) = C(16) \approx 4$$

$$C(n) + 1 = C(8) + 1 \approx 3 + 1 \approx 4$$

$$C(2n) = C(n) + 1$$

→ Complexité est **$O(\log_2(n))$**

Exemple:

- ❑ Chercher un élément dans un tableau ordonné (dichotomique)

Complexité exponentielle:

Script Python:

Entrée = **n**

```
def e(n) :  
    m = n  
    k = 1  
    for i in range(n) :  
        j = 1  
        while j <= m :  
            k += 1  
            j += 1  
        m = m * 2  
    return k
```

Complexité:

n	i	m
3	1, 2, 3	3,6,9

n	i	m
4	1,2,3,4	4 , 8 , 16 , 32

$$C(n+1) \approx 2^* C(n)$$

→ Complexité est **$O(2^n)$**

Exemple:

□ Trouver tous les mots binaires possibles de longueur n bits