

Introduction à l'algorithmique et à la programmation en Python

Informations générales

- **Objectifs :**
 - comprendre et utiliser des schémas algorithmique . . .
 - . . . en utilisant le langage Python (version 3)
- **Ressources :**
 - **Slides**
 - Un **TP de découverte** de Python et de l'environnement IDLE
 - **Python Tutor** : un excellent outil permettant d'exécuter un programme pas-à-pas, en visualisant le contenu de chaque variable. **A utiliser sans modération** pour bien comprendre ce qu'il se passe!
<http://www.pythontutor.com> (et choisir Python 3.6)
 - **Exercices** : quizz et exercice de programmation
 - **Projet final**

Sommaire (cliquable)

1. Variables, Affectations, Entrées/Sorties
2. Instruction If else
3. Booléens et chaînes de caractères
4. Boucles while
5. Fonctions (bases)
6. Fonctions (avancé)
7. Listes
8. Boucles For
9. Dictionnaires
10. Lecture/Écriture dans un fichier

THÈME 1: VARIABLES, AFFECTATIONS, ENTREES/SORTIES

Notions du thème:

- Introduction :
 - algorithme, programme
 - Interpréteur, compilateur
- Variables et expressions
- Entrées / Sorties

Compilation et interprétation

- L'ordinateur permet d'**automatiser** des tâches
- Mais il faut utiliser un **programme**
 - syntaxe précise dans un langage donné lisible par l'humain
 - doit être transformé en un texte lisible par la machine (suite d'octets) → compilateur ou interpréteur
- **compilateur** : traduit une fois pour toute le code source en exécutable
- **interpréteur** : traduit au fur et à mesure à chaque lancement du programme interprété
- **Python** est un langage **interprété**

Algorithme vs Programme

- Problème complexe: travail en 2 temps
 1. Résolution du problème
en s'autorisant syntaxe approximative, fautes d'orthographe, abréviations (et en s'aidant avant avec des schémas) => **algorithme**
 2. Rédaction de la solution finale
produire le **programme** qui permettra de faire faire ce qui est demandé à l'ordinateur
- Notion d'algorithme est plus générale :
cuisine, protocole expérimental, indications routières...

Exercice

- Donner l'algorithme pour faire une omelette (un seul œuf)
- Préciser
 - les ingrédients nécessaires,
 - les ustensiles nécessaires,
 - les actions à mener (dans l'ordre)



Exercice 1 : faire une omelette

Algo Omelette_Élémentaire

Début

{ingrédients}

1 œuf, sel poivre, beurre

{ustensiles}

1 saladier, une fourchette, une poêle, une spatule

{procédure}

Casser l'œuf dans un saladier

Saler et poivrer

Battre l'œuf à la fourchette

Dans une poêle, faire chauffer le beurre,

Verser l'œuf battu dans la poêle,

Cuire doucement jusqu'à l'obtention de la texture souhaitée

(baveuse à bien cuite) ←

Server

Fin

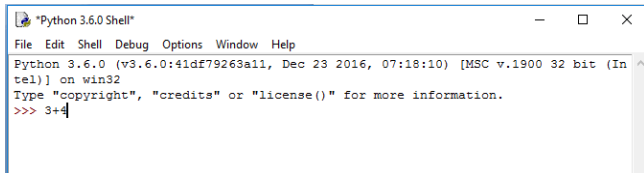
**déclarations
variables**

instructions

commentaires

Programmes en Python

- Deux modes d'exécution d'un code Python
 - utiliser l'interpréteur Python, instruction par instruction, un peu à la façon d'une calculatrice

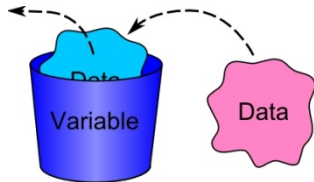


```
*Python 3.6.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 3+4
```

- écrire un ensemble d'instructions dans un fichier puis on l'exécute (via un interpréteur Python)

Variables

- Conteneur d'information
- Identifié par un nom = un **identificateur**
- Ayant un « contenu »



Identificateurs en Python

- **Suite non vide de caractères**
 - commençant par une lettre ou le caractère `_`
 - contenant seulement des lettres, des chiffres et/ou le caractère `_`
 - Ne peut pas être un mot réservé de Python
- **Exemples d'identificateurs**
 - **valides** : toto, proch_val, max1, MA_VALEUR, r2d2, bb8, _mavar
 - **non valides** : 2be, C-3PO, ma var
- Les identificateurs sont **sensibles à la casse** : `ma_var` != `Ma_Var`
- **Conventions** pour les variables en Python :
 - utiliser des minuscules
 - pas d'accents

Affectation

- Pour mémoriser une valeur dans une variable, on fait une affectation en utilisant le signe =
- **Exemples :**

```
n = 33
```

```
a = 42 + 25
```

```
ch = "bonjour"
```

```
euro = 6.55957
```

- L'identificateur (à gauche de =) reçoit la valeur (à droite du =; l'expression doit d'abord être évaluée).
- La première affectation d'une variable est aussi appelée **initialisation**

Exemple

```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:11)
Type "copyright", "credits" or "license()" for more in:
>>> a = 3 * 7
>>> a
21
>>> b = 7.3 # separateur decimal est un point
>>> b
7.3
>>> a = b + 5
>>> a
12.3
>>> a = a * 3
>>> a
36.900000000000006
>>>
```

La valeur d'une variable peut changer
au cours de l'exécution d'un programme
La valeur antérieure est perdue.

Affectation vs Condition en Python

- Le signe “=” sert seulement à faire une affectation.
- Pour savoir si 2 nombres sont égaux, on utilise “==”

Exemples :

```
>>> a = 6
```

```
>>> a
```

```
6
```

```
>>> b = 9
```

```
>>> a == b
```

```
False
```

Notion de typage

- Les valeurs des variables sont de **nature** différente
 - entier
 - réel
 - chaîne de caractères

```
n = 33
a = 42 + 25
euro = 6.55957
ch = "bonjour"
```

- En programmation, on parle de **type**
- Selon les langages de programme, le type des variables
 - est déclaré dans le programme : typage **statique**
 - est déterminé par le compilateur/interprète : type **dynamique**
- En Python, le typage est **dynamique**
 - Pour connaître le type d'une variable: `type(ma_var)`

Exemples

```
>>> a = 17
```

```
>>> type(a)
```

```
<class 'int'>
```

```
>>> a = "salut"
```

```
>>> type(a)
```

```
<class 'str'>
```

```
>>> a = 3.14
```

```
>>> type(a)
```

```
<class 'float'>
```

```
>>> type(21==7*3)
```

```
<class 'bool'>
```

Expression

- C'est une formule qui peut être évaluée

- Exemples :

42 + 2 * 5.3

3*2.0 - 5

"bonjour"

20 / 3

- Expression : des **opérandes** et des **opérateurs**.
- Les opérateurs que l'on peut utiliser **dépendent du type** des valeurs qu'on manipule
- Une expression qui ne peut prendre que les valeurs True ou False est appelée **expression booléenne**

Quelques opérateurs

- **arithmétiques** (sur des nombres) :
+, -, *, **, /, %, //
- **de comparaison** (résultat booléen) :
==, !=, <, >, <=, >=
- **logiques** (entre des booléens, résultat booléen) :
or, and, not

Exercice

Quelle est la réponse de l'interpréteur après chaque expression ?

```
>>> 2 + 3
```

```
>>> 2*3
```

```
>>> 2**3
```

```
>>> 20/3
```

```
>>> 20//3
```

```
>>> 20%3
```

```
>>> 2 > 8
```

```
>>> (2 <= 8) and (8 < 15)
```

```
>>> 2 <= 8 < 15
```

```
>>> (x % 2 == 0) or (x >= 0)
```

Entrées / Sorties

- On a généralement besoin de pouvoir **interagir** avec un programme :
- pour lui fournir les données à traiter, par exemple au clavier : **entrées**
- pour pouvoir connaître le résultat d'exécution ou pour que le programme puisse écrire ce qu'il attend de l'utilisateur, par exemple, texte écrit à l'écran : **sorties**

Les entrées : fonction **input()**

- A l'exécution, l'ordinateur :
 - interrompt l'exécution du programme
 - affiche éventuellement un message à l'écran
 - attend que l'utilisateur entre une donnée au clavier et appuie Entrée.
- C'est une saisie en **mode texte**
 - valeur saisie vue comme une **chaîne de caractères**
 - on peut ensuite changer le type

Les entrées

```
>>> texte = input()
123
>>> texte + 1 # provoque une erreur
>>> val = int(texte)
>>> val + 1 # ok
124
>>> x = float(input("Entrez un nombre :"))
Entrez un nombre :
12.3
>>> x + 2
14.3
```

Les sorties : la fonction print()

- affiche la **représentation textuelle** de n'importe quel nombre de valeurs fournies entre les parenthèses et séparées par des virgules
- à l'affichage, ces valeurs sont séparées par un **espace**
- l'ensemble se termine par un retour à la ligne
 - modifiable en utilisant sep et/ou end
- Possibilité d'insérer
 - des **sauts de ligne** en utilisant `\n` et
 - des **tabulations** avec `\t`

Exemples de sorties

```
>>> a = 20
>>> b = 13
>>> print("La somme de", a, "et", b, "vaut",
          a+b, ".")
```

La somme de 20 et 13 vaut 33.

```
>>> print(a,b,sep= ";")
20;13
```

```
>>> print("a=",a, "b=",b, sep="\n")
a=
20
b=
13
```

THÈME 2:

INSTRUCTION CONDITIONNELLE

IF ELIF ... ELSE

Instructions conditionnelles : if

- **Objectif** : effectuer des actions seulement si une certaine condition est vérifiée
- **Syntaxe en Python** :

```
if condition :  
    instructions à exécuter si vrai
```

La condition est une **expression booléenne**

- **Attention à l'indentation !**
 - Indique dans quel bloc se trouve une instruction.
 - obligatoire en Python.

Instructions conditionnelles : if . . . else

- **Objectif** : effectuer des actions **différentes** selon qu'une certaine condition est vérifiée ou pas
- **Syntaxe en Python**

if condition :

instructions à exécuter si vrai

else :

instructions à exécuter si faux

Attention : le **else** n'est pas suivi d'une condition

Exemple d'instruction conditionnelle

```
x = float(input("Entrez un nombre :"))
if x > 0 :
    print(x, "est plus grand que 0")
    print("il est strictement positif")
else :
    print(x, "est négatif ou nul")
print("Fin")
```

Instructions conditionnelles : avec elif

- **Objectif** : enchaîner plusieurs conditions
- **Exemple** : calculer le nombre de racines réelles d'un polynôme du second degré

Soit une équation au second degré : $f(x) = a x^2 + b x + c$
Les racines : valeurs de x telle que l'équation $f(x) = 0$

On calcule le discriminant : $\Delta = b^2 - 4 * a * c$

$\Delta > 0$: 2 solutions

$\Delta = 0$: 1 solution

$\Delta < 0$: 0 solution

Instructions conditionnelles : avec elif

```
a = 3.2    # coefficient du monôme de degré 2
b = 5      # coefficient du monôme de degré 1
c = -7.9   # coefficient du monôme de degré 0
d = b**2 - 4*a*c # delta
if d>0 :
    print("Deux racines reelles distinctes")
elif d==0 :
    print("Une seule racine reelle")
else :
    # ici on a forcement d < 0
    print("Aucune racine reelle")
```

On utilise autant de blocs **elif** que nécessaire.

THÈME 3: BOOLÉENS ET CHAINES DE CARACTERES

Notions du thème:

- Chaîne de caractères : format et opérateurs
- Expressions booléennes

Précisions sur les chaînes de caractères

On a déjà utilisé les chaînes de caractères, notamment dans les fonctions `print()` et `input()`.

En Python, il existe 3 syntaxes pour les chaînes de caractères :

- avec des guillemets :

```
print("toto")
```

- avec des apostrophes :

```
Print('toto')
```

- avec des guillemets triples :

```
print("""toto""")
```

Intérêt de ces syntaxes

- On peut utiliser " dans une chaîne délimitée par ' ... '
- On peut utiliser ' dans une chaîne délimitée par "..."
- On peut utiliser " et ' dans une chaîne délimitée par """"...""""
- """"..."""" permet aussi d'écrire des chaînes de caractères sur plusieurs lignes (on y reviendra plus tard)

Exemples

```
>>> print("C'est toto")
```

```
C'est toto
```

```
>>> print('C'est toto')
```

```
SyntaxError : invalid syntax
```

```
>>> print("Il a dit "hello" !")
```

```
SyntaxError : invalid syntax
```

```
>>> print('Il a dit "hello" !')
```

```
Il a dit "hello"
```

```
>>> print("""C'est toto qui a dit "hello" !""")
```

```
C'est toto qui a dit "hello" !
```

```
>>> print("""C'est toto qui a dit "hello""")
```

```
SyntaxError : ...
```

Opérations sur les chaînes de caractères

- Longueur :

```
>>> s = "abcde"
```

```
>>> len(s)
```

```
5
```

- Concaténation :

```
>>> "abc" + "def"
```

```
'abcdef'
```

- Répétition :

```
>>> "ta " * 4
```

```
'ta ta ta ta'
```

Expressions booléennes

« ou » logique : `or`

- `expr1 or expr2` vaut vrai si et seulement si au moins une des deux expressions `expr1` et `expr2` est vraie.
- En Python, le « ou » est fainéant, c'est-à-dire que si la 1^{ère} expression vaut **vrai**, la deuxième n'est pas évaluée

`(2 == 1 + 1) or (a >= 5)`

ne provoque pas d'erreur même si `a` n'existe pas, le résultat vaut vrai

`(3 == 1 + 1) or (a >= 5)`

provoque une erreur si `a` n'existe pas.

Expressions booléennes

« et » logique : `and`

- `expr1 and expr2` vaut vrai si et seulement si les deux expressions `expr1` et `expr2` sont vraies.
- En Python, le « et » est fainéant, c'est-à-dire que si la 1^{ère} expression vaut **faux**, la deuxième n'est pas évaluée

`(2 > 8) and (a > = 5)`

ne provoque pas d'erreur même si `a` n'existe pas, le résultat vaut faux

`(2 < 8) and (a > = 5)`

provoque une erreur si `a` n'existe pas.

Lois de De Morgan

not(expr1 **or** expr2) = **not**(expr 1) **and** **not**(expr2)

Exemple :

`not(a > 2 or b <= 4)` équivaut à
`(a <= 2) and (b > 4)`

not(expr1 **and** expr2) = **not**(expr 1) **or** **not**(expr2)

Exemple :

`not(a > 2 and b <= 4)` équivaut à
`(a <= 2) or (b > 4)`

THÈME 4: BOUCLES WHILE

Notion du thème:

- Boucle **while** :
 - Fonctionnement et syntaxe
 - Compteur de boucle
 - Accumulateur et drapeau
 - Boucles imbriquées

Fonctionnement et syntaxe

- But: **répéter** des instructions jusqu'à ce qu'une condition change.

- Syntaxe:

```
while condition:  
    instructions  
suite_du_programme
```

- Les `instructions` seront répétées tant que `condition` est **vraie**.
Lorsque que `condition` devient **fausse**, on exécute `suite_du_programme`.

Exemple: division de A par B

Un programme qui demande un entier A, puis un entier B jusqu'à ce que celui-ci soit non-nul, puis qui calcule le quotient de A par B.

```
a = int(input("Donnez la valeur de A : "))
b = int(input("Donnez la valeur de B : "))
while b == 0 :
    b = int(input("B est nul! Recommencez : "))
print("A / B = ", a // b)
```

*Note: si b est non-nul dès le premier essai, on n'entre **pas** dans le while.*

Attention aux boucles infinies!

Si la condition de la boucle while ne devient jamais fausse, le programme boucle **indéfiniment**:

Exemple 1:

```
n=5
while n<10 :
    print("n vaut :", n)
print("Fin")
```

Exemple 2:

```
while True :
    print("Je boucle.")
print("Fin")
```

Compteur de boucle

- Sert à compter **combien de fois l'on passe** dans la boucle (ou un nombre qui dépend de cela)

```
b = int(input("Donnez un entier b non-nul:"))
n = 0 # n est le compteur de boucle
while b==0:
    n=n+1
    b=int(input("Incorrect, recommencez: "))
print("Merci, il vous a fallu ", n , "essais supplémentaires.")
```

Important: Toujours pensez à initialiser le compteur!

Compteur de boucle: exemples

- On peut se servir du compteur dans la condition.

```
i = 0 # variable compteur
while i < 10:
    print( 2 ** i ) # 2 puissance i
    i = i + 1 # incrémenter notre compteur
print("Fin")
```

Le **pas** = augmentation du compteur à chaque étape

Ici le pas est égal à 1.

Compteur de boucle: exemples

- On peut utiliser un pas différent de 1 (même négatif)

```
i = 0 # variable compteur
while i < 100:
    print(i)
    i = i + 2
print("Fin")
```

Ici le pas = 2

Compteur de boucle: exemples

- On peut utiliser un pas différent de 1 (même négatif)

```
i = 10 # variable compteur
while i>0:
    print(i)
    i = i - 1
print("Fin")
```

Ici, pas: -1

Variable "accumulateur"

- Pour stocker des informations sur les valeurs parcourues, par exemple la somme:

```
i=1
somme = 0 # initialement, la somme est égale à 0
while i <= 10 :
    somme = somme + i
    # chaque valeur de i est rajoutée à la somme
    #(accumulation)
    i = i + 1
    # ne jamais oublier de mettre à jour le compteur
print("La somme des 10 premiers entiers est : ", somme)
```

Initialisation de « l'accumulateur »

1. Attention à ne pas l'oublier
2. Utiliser l'élément neutre de l'opération
 - Pour l'**addition** entre nombres: **0** (car $x+0=x$)
 - Pour la **multiplication** entre nombres: **1** (car $x*1=x$)
 - Pour la **concaténation** entre str: `""` (chaîne vide)
(car `""+"bonjour"="bonjour"`)

Variable « drapeau »

- Un accumulateur booléen est appelé **drapeau**.
- Exemple: lire 10 entiers et vérifier qu'ils sont tous impairs:

```
i=0
tous_impairs = True
while i < 10:
    x = int(input('Entrez un entier:'))
    tous_impairs = tous_impairs and (x % 2 != 0)
    i=i+1
if tous_impairs:
    print('Tous les nombres entrés sont impairs')
else:
    print('Au moins un nombre entré n'était pas impair')
```

Initialisation d'un « drapeau »

- Utiliser l'élément neutre des opérations booléennes
 - Pour un **ET**: `True`
car `True and b` vaut `b`
 - Pour un **OU**: `False`
car `False or b` vaut `b`

Le mot-clé **break**

- Permet de **sortir immédiatement** de la boucle while

```
i=1
while i<100:
    if i % 2 == 0 :
        print("*")
        break
    i=i+1
    print("Incrementation de i")
print("Fin")
```

Le mot-clé **continue**

- Permet de **remonter immédiatement au début** de la boucle `while` en ignorant la suite des instructions dans la boucle.

```
i=1
while i<100:
    if i % 2 == 0 :
        print("*")
        continue
    i=i+1
    print("Incrementation de i")
print("Fin")
```

Break et continue

- Inconvénients:
 - Code plus difficile à lire/analyser si plusieurs niveaux d'imbrications et/ou longues instructions dans le while
 - N'a pas toujours d'équivalent dans les autres langages de programmation
- ➔ On essaiera tant que possible de se passer de break et continue.

Au lieu de break

```
while cond:
    instructions
    if ... :
        ...
        break
    other_instruct
```

```
stop=False
while (not stop) and cond:
    instructions
    if ... :
        ...
        stop=True
    if not stop:
        other_instruct
```

Boucles imbriquées

- Une instruction d'une boucle **while** peut être une boucle **while**

Ex : résultat produit par ce programme ?

```
i = 1
while i <= 3 :
    j = 1
    while j <= 2 :
        print(i, ", ", ", j)
        j = j + 1
    i = i + 1
```

Résultat :

1, 1

1, 2

2, 1

2, 2

3, 1

3, 2

Exemple d'application

On veut écrire un programme qui affiche un « carré » de $n \times n$ fois le caractère '*'. L'utilisateur choisit le côté n du carré.

Exemple de résultat :

```
Entrez la valeur de n : 5
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

Correction

```
n = int(input("Entrez la valeur de n : "))
num_lig = 0 # compteur de ligne
while num_lig < n :

    num_col = 0 #cpt nb etoiles de la ligne

    while num_col < n :
        print("*", end="") # /\ end
        num_col = num_col + 1
    print() # saut de ligne

    num_lig = num_lig + 1 # passer ligne suivante
```

THÈME 5: FONCTIONS (BASES)

Notion du thème:

- Fonctions:
 - Principe
 - Syntaxe
 - Portée des variables

Fonctions : pourquoi ?

But: structurer son code lorsque l'on fait plusieurs fois la même chose (ou presque)

- Pour qu'il soit plus **lisible** (plusieurs morceaux)
- Pour qu'il soit plus **facilement modifiable**
- Pour qu'il soit plus **facile à tester**

Un exemple

```
import turtle

def carre(cote) :
    # trace un carre de taille egale a cote
    i = 1 # compteur du nb de cotes
    while i <= 4 :
        turtle.forward(cote)
        turtle.right(90)
        i=i+1
```

Un exemple

```
# programme principal
```

```
carre(100)
```

```
turtle.up()
```

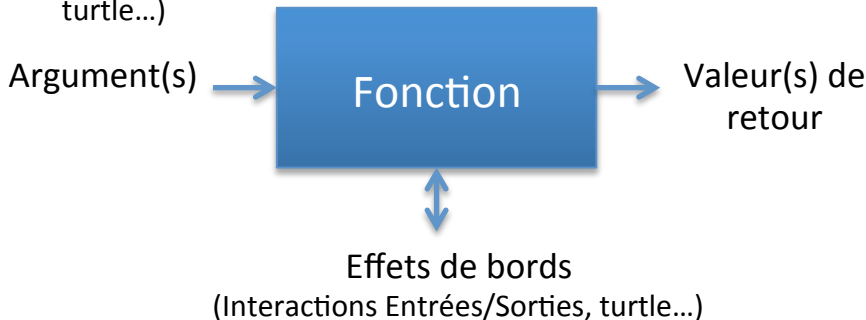
```
turtle.forward(130)
```

```
turtle.down()
```

```
carre(50)
```

Principe

- Une suite d'instructions encapsulées dans une « boîte »
- Qui prend zéro, un ou des **arguments**
- Qui retourne zéro, une ou plusieurs **valeurs de retour**
- Et qui contient éventuellement des **"effets de bord"** qui modifient l'environnement (interactions entrées/sorties, turtle...)

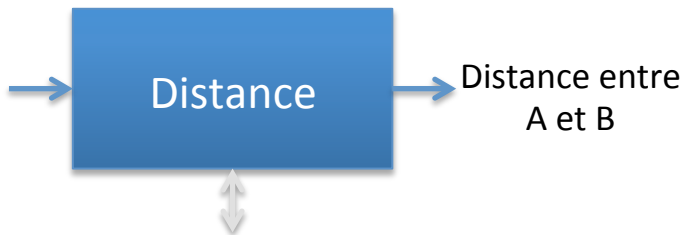


Exemple

Dans un exercice de géométrie, on doit souvent calculer la distance entre deux points.

$$\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

Abscisse A,
Ordonnée A,
Abscisse B,
Ordonnée B



Distance entre
A et B

Effets de bords (ici:aucun)
(Interactions Entrées/Sorties, turtle...)

Exemple de la distance

Fonction à définir au dessus de votre programme principal:

```
def distance(absA, ordA, absB, ordB) :  
    d=(absB-absA)**2 + (ordB-ordA)**2  
    d=d**(1/2)  
    return d
```

4 arguments: absA, ordA, absB, ordB

1 valeur de retour de type float (d)

Pas d'effets de bord

Exemple de la distance

```
if __name__=="__main__": # prog. principal

    print(distance(1, 2, 1, 5))
    xA=2
    yA=3
    z=distance(xA, yA, 0, 0)
    print("Distance de (0,0) à A :", z)
```

Ou directement dans l'interpréteur:

```
>>> distance(0, 1, 3, 5)
5.0
```

Python Tutor

Start shared session

[What are shared sessions?](#)

Python 3.6

```

1 def distance(absA, ordA, absB, ordB) :
2     d=(ordB-ordA)**2+ (absB-absA)**2
3     d=d**(1/2)
4     return d
5
6 # prog. principal
7 if __name__=="__main__":
8     xA=2
9     yA=3
10    z=distance(xA, yA, 0, 0)
11    print("Distance de (0,0) à A :", z)

```

[Edit code](#) | [Live programming](#)

→ line that has just executed

→ next line to execute

NEW! Click on a line of code to set a breakpoint. Then use the Forward and Back buttons to jump there.

<< First

< Back

Step 10 of 11

Forward >

Last >>

Print output (drag lower right corner to resize)

Frames

Objects

Global frame

distance

xA

yA

function
distance(absA, ordA, absB, ordB)

distance

absA 2

ordA 3

absB 0

ordB 0

d 3.6056

Return
value 3.6056

Syntaxe d'une nouvelle fonction

```
def nom_fonction(argument1,..., argumentN) :  
    instructions à exécuter  
    return valeur de retour
```

Note : le `return` est facultatif, ainsi que les arguments (mais pas les parenthèses!)

Programme principal

À placer en-dessous de la définition des fonctions.

```
if __name__ == '__main__': # programme principal
    instructions à exécuter
```

Note : cette ligne est facultative dans les TD/TP mais **obligatoire dans Caseine**.

Appel d'une fonction

`nom_fonction(argument1, argument2, ...)`

→ vaut la valeur de retour de la fonction (s'il y en a)

Exemple, en supposant que les fonctions `distance` et `carre` ont été définies au-dessus.

```
z=distance(2, 3, 4, 5)
print(z)
carre(50) # pas de valeur de retour,
          # mais des effets de bord (turtle)
```

Note: un appel de fonction peut se faire **dans le programme principal** mais aussi à l'intérieur **d'une autre fonction**.

Appel d'une fonction depuis une autre fonction

En supposant que la fonction `carre` a été définie au-dessus.

```
def deplace_sans_tracer(distance):  
    up()  
    forward(distance)  
    down()  
  
def ligne_carres(nb_carres, cote):  
    i=0 # compte le nb de carres déjà tracés  
    while i<nb_carres:  
        carre(cote) # appel a la fonction carre  
        deplace_sans_tracer(cote+10) # appel  
        i=i+1
```

Fonction sans argument

```
import turtle
def carre_standard():
    i = 1 # compteur du nb de cotes
    while i <= 4 :
        turtle.forward(100)
        turtle.right(90)
        i=i+1
```

```
def demander_nom():
    nom=input("Quel est ton nom?")
    return nom
```

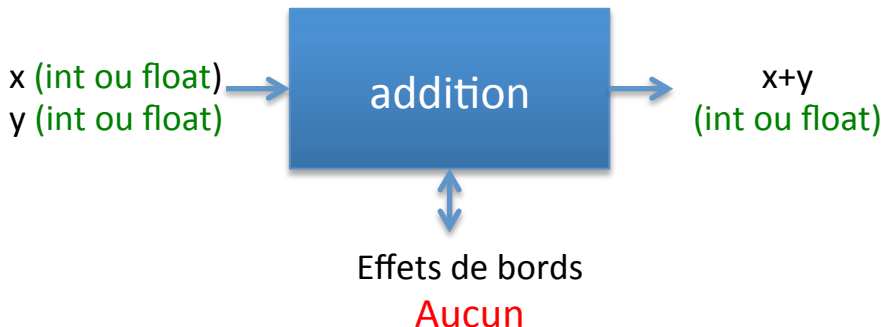
Exemple d'appels:

```
name=demander_nom() # pas d'argument
carre_standard() # pas d'argument ni valeur de retour
```

Différence

Effets de bord vs. Valeur de retour

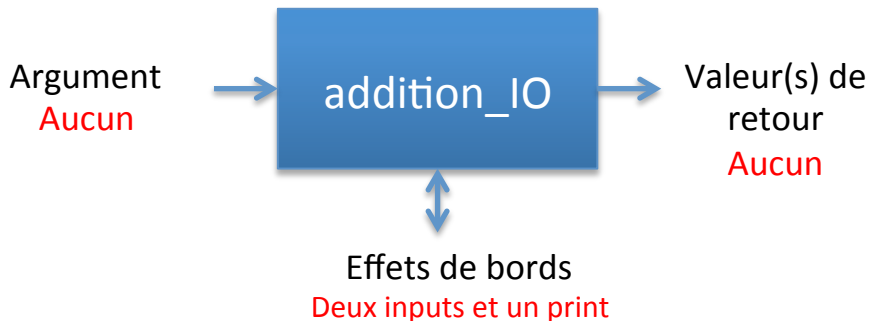
```
def addition(x, y):  
    return x+y
```



Différence

Effets de bord vs. Valeur de retour

```
def addition_IO():  
    x=float(input("x ?"))  
    y=float(input("y ?"))  
    print(x+y)
```



Mise en situation

Trois équipes:

- Une équipe fonction `moyenne`
- Une équipe fonction `ecart_plus_grand_que`
- Une équipe programme principal
- Communication entre les équipes par
 - Appel de fonction (avec les arguments)
 - Valeur de retour(modélisée par des papiers)
- Le tableau sert uniquement pour les `print` et `input`

Fonctions utilisées

```
def moyenne(x,y):  
    # renvoie la moyenne de x et y  
    resultat=(x+y)/2  
    return resultat  
  
def ecart_plus_grand_que(x,y,seuil):  
    # calcule l'ecart entre x et y  
    # et renvoie True si l'ecart est plu grand que seuil, False  
    sinon  
    if x>y:  
        ecart=x-y  
    else:  
        ecart=y-x  
    # ecart contient la valeur absolue de x-y  
    # on aurait pu faire: ecart=abs(x-y)  
    if ecart>=seuil:  
        return True  
    else:  
        return False
```


Prog. Principal utilisé

```
if __name__=="__main__": # prog. principal
    # demande deux nombres, affiche leur moyenne,
    # et recommence si les nombres étaient proches
    seuil=2
    continuer=True
    while continuer:
        a=float(input("a=?"))
        b=float(input("b=?"))
        m=moyenne(a,b) # appel de la fonction
        print("Moyenne :", m)
        if ecart_plus_grand_que(a,b,seuil): # appel de la
fonction
            continuer=False
        else:
            print("Ecart plus petit que", seuil)
    print("Fin car l'écart était plus grand que", seuil)
```

Portée des variables

Chaque fonction a son propre "lot" de variables auquel elle a le droit d'accéder (cf Python Tutor).

Une variable

- **créée ou modifiée** dans le corps d'une fonction ,
- ou qui contient un **argument** de la fonction

est dite **locale**, et ne sera pas accessible depuis le programme principal, ni depuis une autre fonction.

Variable locale: exemple

```
def moyenne(x,y):  
    # renvoie la moyenne de x et y  
    resultat=(x+y)/2  
    return resultat  
  
if __name__=="__main__": # prog. principal  
    a=5  
    b=6  
    m=moyenne(a,b)  
    print(m) # affiche 5.5  
    print(resultat) # provoque une erreur  
➔ NameError: name 'resultat' is not defined
```

Portée des variables

Une variable (de type int, float, bool ou str) définie dans le programme principal **ne peut pas être modifiée** par une instruction qui se trouve à l'intérieur d'une **fonction***.

Cela ne provoque pas d'erreur mais cela **créé une nouvelle variable locale** portant le **même identificateur (nom)** que l'autre variable.

- *1. Sauf si le mot-clé `global` est utilisé, mais nous ne le ferons pas.
- 2. Il y aura des subtilités lorsque nous verrons les listes.

Portée des variables: exemple

```
def moyenne(x,y):  
    # renvoie la moyenne de x et y  
    resultat=(x+y)/2  
    test=resultat # crée une nouvelle variable test  
    return resultat  
  
if __name__=="__main__": # prog. principal  
    a=5  
    b=6  
    test=0  
    m=moyenne(a,b)  
    print("m =", m) # affiche "m=5.5"  
    print("test =", test) # affiche "test=0"  
    # car test n'est pas modifié par l'appel de fonction
```

Portée des variables

*Les variables définies dans le programme principal sont accessibles en lecture seule depuis l'intérieur d'une fonction mais ce comportement est **dangereux** car très subtil.*

On ne l'utilisera donc pas (sauf éventuellement pour des variables "constantes", initialisées une fois au début du programme et jamais modifiées ensuite.)

On préférera passer en arguments toutes les valeurs nécessaires.

À ne pas faire: exemple

```
def decalage(s):  
    # renvoie la chaine s préfixée de n tirets  
    espaces="-" * n  
    resultat=espaces+s  
    return resultat
```

Il vaut mieux passer **n** en argument

```
if __name__=="__main__": # prog. principal  
    n=5  
    test=decalage("toto")  
    print(test)  
    n=10  
    test=decalage("maison")  
    print(test)
```

Correction de l'exemple

```
def decalage(s, n):  
    # renvoie la chaine s préfixée de n tirets  
    espaces="-" * n  
    resultat=espaces+s  
    return resultat
```

n est maintenant un argument
(donc une variable locale)

```
if __name__=="__main__": # prog. principal  
    n=5  
    test=decalage("toto", n)  
    print(test)  
    n=10  
    test=decalage("maison", n)  
    print(test)
```


Portée des variables

Résumé:

- Une variable créée ou modifiée **dans une fonction** est **locale** (= elle n'existe que dans la fonction).
- Une variable (de type int, float, str ou bool) du **programme principal** ne peut **pas être modifiée** à l'intérieur d'une fonction.
- On passera en **argument** de la fonction toutes les **valeurs nécessaires**.

THÈME 6: FONCTIONS (AVANCÉ)

Notions du thème :

- Fonctions:
 - Plusieurs valeurs de retour
 - Docstring
 - Arguments optionnels
- Le mot-clé **None**

Plusieurs valeurs de retour:

Un exemple

```
def division(a,b) :  
    # renvoie le quotient et le reste  
    # de la division de a par b  
    quotient=a//b  
    reste= a%b  
    return quotient, reste
```

```
# programme principal  
q,r = division(22,5)  
print("q=", q, "et r=", r)
```

Plusieurs valeurs de retour

Syntaxe (dans le corps de la fonction):

```
return valeur1, valeur2, ... , valeurN
```

Pour récupérer toutes les valeurs de retour lors d'un appel:

```
var1, var2, ..., varN = nom_fonction(arguments)
```

Docstring: Un exemple

```
def division(a,b) :  
    """ Renvoie le quotient et le reste  
        de la division de a par b """  
    quotient=a//b  
    reste= a%b  
    return quotient, reste
```

Dans l'interpréteur (ou dans un programme):

```
>>> help(division)
```

```
Help on function division in module __main__:
```

```
division(a, b)
```

```
    Renvoie le quotient et le reste  
    de la division de a par b
```

Docstring

Une **docstring** est une **chaîne de caractères** (encadrée par des triple guillemets) placée au tout **début d'une fonction**, qui permet de **décrire** la fonction.

```
def nom_fonction(argument1, argument2, ...):  
    """ docstring  
    """  
    instructions de la fonction
```

On peut l'afficher grâce à **help(nom_fonction)**:

```
Help on function nom_fonction in module:  
nom_fonction(argument1, argument2, ...)  
    docstring de la fonction affichée ici
```

/!\ Pas de parenthèses après `nom_fonction` dans `help(...)`

Docstring sur une fonction existante

```
>>> import random
```

```
>>> help(random.randint)
```

```
Help on method randint in module random:
```

```
randint(a, b) method of random.Random instance
```

```
    Return random integer in range [a, b],  
    including both end points.
```


Définir une fonction avec des arguments optionnels: exemple

La carte de MisterPizza comporte de multiples saveurs de pizzas, chacune pouvant être commandée en taille normale au prix de 9€, ou en taille maxi au prix de 12€. La très grande majorité des clients choisit des pizzas de taille normale.

```
def affiche_pizza(saveur, taille="normale"):  
    """ Affiche saveur, taille et prix de la pizza  
    """  
    print("Pizza", saveur, "taille:", taille)  
    if taille=="normale":  
        prix=9  
    elif taille=="maxi":  
        prix=12  
    print("Prix", prix, "euros.")
```

→ `taille` est un argument optionnel ayant comme **valeur par défaut** "normale".

Définir une fonction avec des arguments optionnels

Pour rendre un argument **optionnel** lors de la définition d'une fonction, il faut ajouter après le nom de l'argument le signe = suivi de la valeur par défaut.

```
def nom_fonc(arg1, arg_opt=valeur_par_defaut):  
    instructions
```

→ `arg_opt` **est** un argument optionnel de la fonction

Mais `arg1` **n'est pas** un argument optionnel de la fonction

Appeler une fonction avec des arguments optionnels

```
>>> affiche_pizza("4 fromages")  
Pizza 4 fromages taille: normale  
Prix 9 euros.
```

```
>>> affiche_pizza("4 fromages", "maxi")  
Pizza 4 fromages taille: maxi  
Prix 12 euros.
```

```
>>> affiche_pizza("Reine", "normale")  
Pizza Reine taille: normale  
Prix 9 euros.
```

Help et Docstring

sur une fonction avec arguments optionnels

```
>>> help(affiche_pizza)
Help on function affiche_pizza in module __main__:

affiche_pizza(saveur, taille='normale')
    Affiche saveur, taille et prix de la pizza
```

Plusieurs arguments optionnels

MisterPizza souhaite parfois afficher le prix en Francs (lorsque le client est âgé, mais il s'agit d'une situation peu fréquente).

```
def affiche_pizza(saveur, taille="normale", afficheF=False):  
    """ Affiche saveur, taille et prix de la pizza  
    """  
    print("Pizza", saveur, ", taille: ", taille)  
    if taille=="normale":  
        prix=9  
    elif taille=="maxi":  
        prix=12  
    if afficheF:  
        prixFrancs=round(prix*6.55957, 2)  
        print("Prix", prix, "euros (", prixFrancs, " F).")  
    else:  
        print("Prix", prix, "euros.")
```

Appel avec plusieurs arguments optionnels

```
>>> affiche_pizza("4 fromages")  
Pizza 4 fromages taille: normale  
Prix 9 euros.
```

```
>>> affiche_pizza("4 fromages", "maxi")  
Pizza 4 fromages taille: maxi  
Prix 12 euros.
```

```
>>> affiche_pizza("Reine", "maxi", True)  
Pizza Reine , taille: maxi  
Prix 12 euros ( 78.71 francs).
```

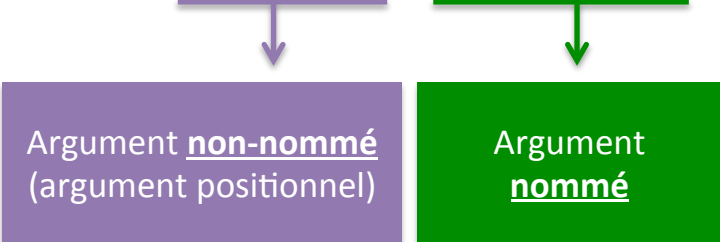
```
>>> affiche_pizza("4 saisons", True)  
Erreur car True est pris pour la taille (2eme arg.)
```

Comment faire pour préciser le 3^{ème} argument mais pas le 2^{ème}?

Arguments nommés

Lors d'un appel de fonction, on peut préciser le nom de l'argument concerné :

```
>>> affiche_pizza("4 fromages", afficheF=True)
```



Argument non-nommé
(argument positionnel)

Argument nommé

Les arguments non-nommés doivent toujours être **tous avant** les arguments nommés dans l'appel.

L'ordre des arguments nommés peut être fait selon votre préférence.

Appel de fonction

Arguments non-nommés puis Arguments nommés

```
>>> affiche_pizza("4 fromages", afficheF=True)
```

```
Pizza 4 fromages , taille: normale  
Prix 9 euros ( 59.04 francs).
```

```
>>> affiche_pizza("4 fromages", taille="maxi", afficheF=True)
```

```
Pizza 4 fromages , taille: maxi  
Prix 12 euros ( 78.71 francs).
```

```
>>> affiche_pizza("Chorizo", taille="maxi", True)
```

```
Erreur: il y a un argument non-nommé après un argument nommé
```

```
>>> affiche_pizza("Reine", afficheF=True, taille="maxi")
```

```
Pizza Reine , taille: maxi  
Prix 12 euros ( 78.71 francs).
```

```
>>> affiche_pizza("Chorizo", True, taille="maxi")
```

```
Erreur car taille est définie deux fois (True est pris pour  
taille car 2eme argument non-nommé)
```


Arguments optionnels de print

En fait, nous avons déjà rencontré une fonction avec des arguments optionnels: `print`

```
>>> print("Mon age est", 18)
>>> print("Mon age est", 18, sep="égal à")
>>> print("Mon age est", 18, end=".")
>>> print("Mon age est", 18, sep=":", end=".")
```

`sep` et `end` sont des arguments optionnels de `print`. Par défaut, `sep` vaut " " (espace) et `end` vaut "\n" (retour à la ligne).

Note: `sep` et `end` doivent toujours être nommés car la fonction `print` a un nombre variable d'arguments.

Le mot-clé **None**

Il existe une valeur constante en Python qui s'appelle **None**. Cela correspond à "rien", "aucune".

Lorsqu'une fonction n'a pas d'instruction `return`, elle renvoie la valeur `None`.

```
def dit_bonjour():  
    print("Bonjour!")  
    print("Bienvenue")  
    # pas de return  
  
# prog. Principal  
test=dit_bonjour()  
print("Test vaut", test)
```

Affichage lorsque l'on lance le module:

Bonjour!

Bienvenue.

Test vaut None

Le mot-clé `None`

Le mot-clé `None` peut aussi servir à initialiser une variable lorsque l'on ne sait pas encore précisément quelle valeur on souhaite lui attribuer.

```
reponse=None # on n'a pas encore de reponse
while reponse!="non":
    x=float(input("Veuillez entrer un nombre:"))
    print("Le carré de ce nombre est:", x*x)
    reponse=input("Voulez-vous recommencer?")
print("Terminé")
```

#!/ None n'est pas une chaîne de caractères en Python, donc il ne faut pas de guillemets.

THÈME 7: LISTES

Notions du thème:

- Listes:
 - Structure de données
 - Opérateurs sur les listes
 - Effets de bord sur les listes

Types simples et types complexes

```
>>> a = 6
>>> type(a)
<class 'int'>
>>> type(3.5)
<class 'float'>
>>> type(True)
<class 'bool'>
```

- Types simples
- Une seule valeur

- Besoin de manipuler des « structures de données » plus complexes
 - Liste
 - Ensemble
 - Tableaux multi-dimensions
 - ...

Type Python « Liste »

- Ensemble **ordonné** d'éléments (objets)
- Associée à un identificateur
- Peut grandir (ou se réduire) dynamiquement
- Les éléments peuvent être de types différents
- Exemples:

```
Weekend=["Samedi", "Dimanche"]
```

```
Multiple3 = [3, 6, 9, 12]
```

```
Romain = [[1, 'I'], [2, 'II'], [3, 'III'], [4, 'IV']]
```

```
iv = 4
```

```
FourreTout = ["Un", 2, 3.0, iv]
```

```
Vide = []
```

Type Python « Liste » (suite)

```
>>> Weekend
```

```
['Samedi', 'Dimanche']
```

```
>>> Multiple3
```

```
[3, 6, 9, 12]
```

```
>>> FourreTout
```

```
['Un', 2, 3.0, 4]
```

```
>>> Romain
```

```
[[1, 'I'], [2, 'II'], [3, 'III'], [4, 'IV']]
```


Des opérateurs sur les listes

- `Liste[index]`
obtenir l'élément à l'index i
Les éléments sont indexés à partir de 0
- `Liste.append(element)`
ajout d'un seul élément à la fin de la liste

```
>>> Multiple3 = [3, 6, 9]
```

```
>>> Multiple3[0]
```

```
3
```

```
>>> Multiple3.append(21)
```

```
>>> Multiple3
```

```
[3, 6, 9, 21]
```

Exercice : remplir les trous :

```
>> Multiple3 = [3, 6, 9, 15, 21]
```

```
>> Multiple3[2]
```

```
>> Multiple3[____]
```

```
21
```

```
>> Multiple3._____
```

```
>> Multiple3
```

```
[3, 6, 9, 15, 21, 24]
```

Des opérateurs sur les listes

- **len(Liste)**

Pour avoir la taille d'une liste

```
>>> Multiple3 = [3, 6, 9, 15, 21, 24, 27]
```

```
>>> len(Multiple3)
```

```
7
```

```
>>> Rom = [[1, "I"], [2, "II"], [3, "III"], [4, "IV"]]
```

```
>>> len(Rom)
```

```
4
```

Afficher une liste

```
>>> l1=[1,2,3] # crée une liste
>>> print(l1)
[1,2,3]
```

Tester si un élément est dans une liste: mot-clé **in**

```
def cherche(elem, l):  
    if elem in l:  
        return True  
    else:  
        return False  
  
# prog. principal  
ma_liste=[2,5,8,12,17,25]  
trouve=cherche(12,ma_liste)  
print(trouve) # affiche True  
trouve=cherche(6, ma_liste)  
print(trouve) # affiche False
```

Des opérateurs sur les listes

- `Liste.insert(index,element)`
Ajout d'un seul élément à l'index `i`
- `Liste.extend(liste2)`
Ajout d'une liste à la fin de la liste

```
>>> Multiple3 = [3, 6, 9, 21]
```

```
>>> Multiple3.insert(3,15)
```

```
>>> Multiple3
```

```
[3, 6, 9, 15, 21]
```

```
>>> Multiple3[3]
```

```
15
```

```
>>> Multiple3.extend([24,27])
```

```
>>> Multiple3
```

```
[3, 6, 9, 15, 21, 24, 27]
```

Exercice : remplir les trous :

```
>>> Multiple3 = [3, 6, 9, 15, 21, 24, 27]
```

```
>>> Multiple3.insert( ____, 12)
```

```
>>> Multiple3.insert( ____, 18)
```

```
>>> Multiple3
```

```
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

```
>>> Multiple3._____([30,33])
```

```
>>> Multiple3
```

```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33]
```

```
>>> Multiple3._____
```

```
>>> Multiple3
```

```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36]
```

**ATTENTION
ZONE
PIÉGÉE**

Des opérateurs sur les listes

- **Liste.pop(index)**
retire l'élément présent à la position **index** et le renvoie
- **Liste.remove(element)**
retire l'élément donné (le premier trouvé)

```
>>> Multiple3 = [3, 6, 9, 15, 21, 24, 27, 24, 24]
>>> a = Multiple3.pop(0)
>>> a
3
>>> Multiple3
[6, 9, 15, 21, 24, 27, 24, 24]
>>> Multiple3.remove(24)
>>> Multiple3
[6, 9, 15, 21, 27, 24, 24]
```


Exercice : remplir les trous :

```
>>> EhEh = ["tra", "la", "la", "la", "lère"])
```

```
>>> EhEh._____
```

```
>>> EhEh
```

```
["tra", "la", "la", "la"]
```

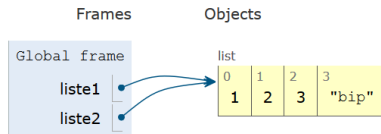
```
>>> EhEh._____
```

```
>>> EhEh
```

```
["tra", "la", "la"]
```

L'opérateur = sur les listes

```
liste1=[1,2,3]
liste2=liste1
liste2.append("bip")
```



ATTENTION
ZONE
PIÉGÉE

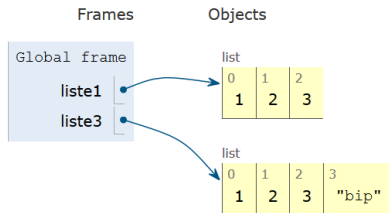
**L'égalité permet de donner 2 noms
à la même liste !**

Les modifications apportées à une des listes après la copie **s'appliquent également** à l'autre liste.

Copie de listes

- Pour copier une liste, on peut utiliser
 - la fonction `list()`
 - l'opération générique `copy.deepcopy()` (`import copy`)

```
liste1=[1,2,3]
liste3=list(liste1)
liste3.append("bip")
```

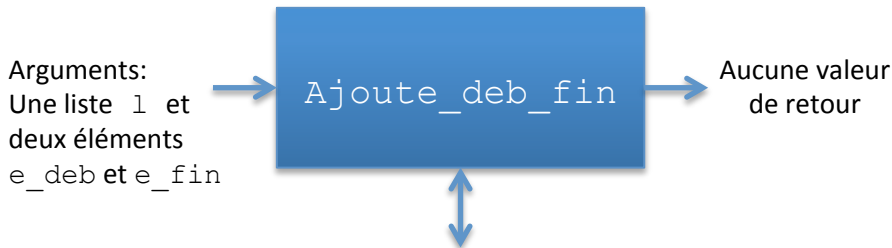


Les modifications apportées à une des listes après la copie **n'affectent pas** l'autre liste.

Effets de bord qui modifient une liste

Une fonction peut **modifier** une liste passée en argument, **indépendamment de sa valeur de retour**.

C'est une nouvelle forme **d'effet de bord** (jusqu'ici: print, input, turtle).



Effets de bords:

Insère `e_deb` au début de `l` et `e_fin` à la fin de `l`

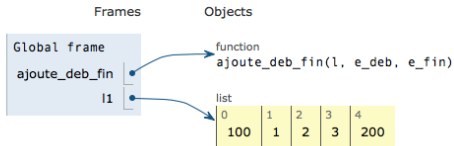
Effets de bord qui modifient une liste

Python 3.6

```
1 def ajoute_deb_fin(l, e_deb, e_fin):  
2     l.append(e_fin)  
3     l.insert(0, e_deb)  
4  
5  
6 l1=[1,2,3]  
7 ajoute_deb_fin(l1, 100, 200)  
→ 8 print(l1)
```

[Edit code](#) | [Live programming](#)

Print output (drag lower right corner to resize)



Les modifications apportées à la liste dans la fonction sont conservées après la sortie de la fonction.

Attention aux effets de bord non désirés!

Il faut bien **penser à copier** la liste passée en argument si **on ne souhaite pas qu'elle soit modifiée** par la fonction.

Exemple **avec oubli** de la copie:

```
import random

def ajoute_random(l):
    """ Renvoie une liste obtenue a partir de l en
    ajoutant un entier aleatoire entre 5 et 10"""
    x=random.randint(5,10)
    l.append(x)
    return l
```

Exemple avec oubli de la copie

La liste argument `l` est modifiée dans la fonction.

Python 3.6

```

1 import random
2
3 def ajoute_random(l):
4     x=random.randint(5,9)
5     l.append(x)
6     return l
7
8 l1=[1,2,3]
9 l2=ajoute_random(l1)
10 print(l2)

```

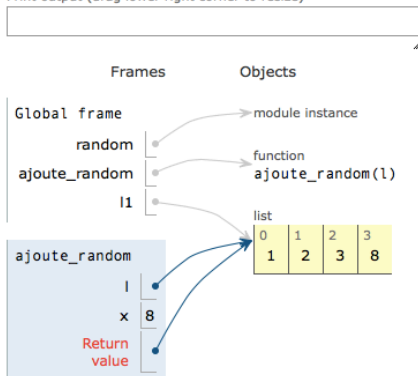
[Edit code](#) | [Live programming](#)

not executed

code

set a breakpoint; use the Back and Forward buttons to jump there.

Print output (drag lower right corner to resize)



Exemple (suite)

Python 3.6

```

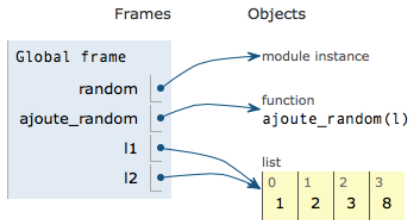
1 import random
2
3 def ajoute_random(l):
4     x=random.randint(5,9)
5     l.append(x)
6     return l
7
8 l1=[1,2,3]
9 l2=ajoute_random(l1)
10 print(l2)

```

[Edit code](#) | [Live programming](#)

Print output (drag lower right corner to resize)

[1, 2, 3, 8]



La liste `l1` a été modifiée alors que l'on ne le souhaitait pas.

Sans effets de bord non désirés

Exemple corrigé: copie de la liste avec `list(...)`

```
import random
```

```
def ajoute_random(l):
```

```
    """ Renvoie une liste obtenue a partir de l en
    ajoutant un entier aleatoire entre 5 et 10 """
```

```
    x=random.randint(5,10)
```

```
    ma_liste=list(l)
```

```
    ma_liste.append(x)
```

```
    return ma_liste
```

Sans effets de bord non désirés

Python 3.6

```

1 import random
2
3 def ajoute_random(l):
4     x=random.randint(5,10)
5     ma_liste=list(l)
6     ma_liste.append(x)
7     return ma_liste
8
9 l1=[1,2,3]
10 l2=ajoute_random(l1)
11 print(l1, l2)

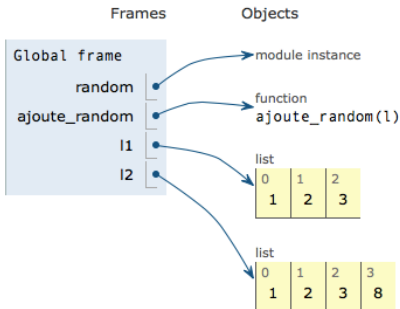
```

[Edit code](#) | [Live programming](#)

t executed
:ute

Print output (drag lower right corner to resize)

```
[1, 2, 3] [1, 2, 3, 8]
```



`l2` est créée correctement et `l1` n'est pas modifiée.

Erreur classique

Erreur à ne pas commettre:

```
import random

def ajoute_random(l):
    """ Renvoie une liste obtenue a partir de l en
    ajoutant un entier aleatoire entre 5 et 10"""
    x=random.randint(5,10)
    ma_liste=l
    ma_liste.append(x)
    return ma_liste
```

Avec effets de bord non désirés

Python 3.6

```

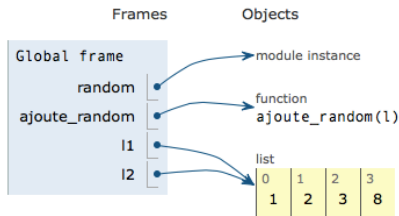
1 import random
2
3 def ajoute_random(l):
4     x=random.randint(5,10)
5     ma_liste=l
6     ma_liste.append(x)
7     return ma_liste
8
9 l1=[1,2,3]
10 l2=ajoute_random(l1)
11 print(l1, l2)

```

[Edit code](#) | [Live programming](#)

Print output (drag lower right corner to resize)

```
[1, 2, 3, 8] [1, 2, 3, 8]
```



Le symbole `=` n'a pas créé une copie de l'argument `l`
 donc la liste `l1` est **modifiée involontairement**.

THÈME 8: BOUCLES FOR

Notions du thème:

- Boucles For
 - Sur les structures de données
 - Avec range

Itération

- Boucle while : la condition détermine le nombre de fois que la boucle est exécutée
boucle conditionnelle
- Si on connaît ce nombre à l'avance, on peut utiliser le for...
boucle inconditionnelle

La boucle for

- Permet de parcourir des structures :
Listes de nombres, d'objets, lettres d'un mots

```
for e in [1, 4, 5, 0, 9, 1] :  
    print(e)
```

```
for e in ["a", "e", "i", "o", "u", "y"] :  
    print(e)
```

```
for e in "python":  
    print(e)
```

e prend successivement
les valeurs de la liste parcourue


```
for i in range(1,6) :  
    print (i,end=",")
```



```
1, 2, 3, 4, 5,
```


- **range** (*deb*, *fin*, *pas*)
 - Fonction qui prend des arguments entiers
 - génère une séquence d'entiers entre [*deb*, *fin*[avec le *pas* choisi.
- Les paramètres *deb* et *pas* sont **optionnels**
 - range** (**a**) : séquence des entiers dans [0, a[, c'est-à-dire dans [0, a-1]
 - range** (**b**, **c**) : séquence des valeurs [b, c[, c'est-à-dire dans [b, c-1]
 - range** (**e**, **f**, **g**) : séquence des valeurs [e, f[avec un pas de g
- **for var in range** (*deb*, *fin*, *pas*) :
instructions

A noter

- En cas d'**incohérence**, la boucle est **ignorée** et l'on passe aux instructions suivantes :

```
for k in range(200, 210, -2) :  
    print(k)  
for k in range(110, 100, -2) :  
    print(k)
```

} *ignorée*



- Quoi qu'il arrive** dans le corps de la boucle, la variable du compteur prend la **valeur suivante** du *range* ou de la liste à chaque nouvelle étape de la boucle

```
for i in range(1, 5) :  
    print(i)  
    i = i*2
```

1
2
3
4

Exercice

- Pour les cas suivants, indiquer les valeurs successives affichées sur la console

```
for i in range(4) :
    print(i)

for j in range(2,5):
    print(j)

for k in range(3,12,3):
    print(k)

for l in range(12, 3):
    print(l)

for m in range(12,3,-2) :
    print(m)
```

```
# la liste en arg. doit ne contenir que des nb
def somme_des_positifs(liste) :
    s=0
    for e in liste:
        if e>0:
            s=s+e
    return s

# prog. principal
ma_liste = [2,-4,6,0,-5,1]
for e in ma_liste:
    print(e+1)
t=somme_des_positifs(ma_liste)
print(t)
```

THÈME 9: DICTIONNAIRES

Dictionnaire

- ***Tout comme une liste***, un dictionnaire permet de sauvegarder en mémoire plusieurs valeurs de types quelconques.
- ***Contrairement à une liste***, les valeurs d'un dictionnaire ne sont pas stockées dans un ordre particulier.

Création et utilisation d'un dictionnaire

- Déclarer un dictionnaire D:

```
D = { cle1: valeur1, cle2: valeur2, ..., cleN: valeurN }
```

- Exemple:

```
>>> notes = { 'quentin': 15.5, 'nathan': 12.0 }
```

- La variable *notes* est un dictionnaire contenant les notes de deux étudiants.
- Les chaînes de caractères 'nathan' et 'quentin' sont les **clés** du dictionnaire. 12.0 et 15.5 sont les **valeurs** du dictionnaire.
- Les éléments du dictionnaire ne sont pas ordonnés:

```
>>> notes  
{'nathan': 12.0, 'quentin': 15.5}
```

Création et utilisation d'un dictionnaire

- L'accès à une valeur du dictionnaire se fait non pas par sa position (indice), mais grâce à sa **clé**.
- Exemple:

```
>>> notes = {'nathan': 12.0, 'quentin': 15.5}
>>> notes['quentin']
15.5
```

 - Ici, la chaîne de caractères 'quentin' est la clé, et la valeur qui y est associée dans le dictionnaire *notes* est 15.5
- Les dictionnaires sont aussi appelés « listes associatives », car ils permettent d'associer à chaque clé une valeur de type quelconque.

Ajouter une nouvelle entrée dans un dictionnaire

- Pour **rajouter** une nouvelle entrée (clé+valeur) dans un dictionnaire existant, il suffit d'utiliser l'opérateur = en spécifiant la clé comme suit:

```
>>> D = {} # crée un dictionnaire vide
```

```
>>> D
```

```
{}
```

```
>>> D['a'] = 1 # ajout de la nouvelle entrée
```

```
>>> D
```

```
{'a': 1}
```

- Si la clé existe déjà, elle prend la nouvelle valeur:

```
>>> D['a'] = 3
```

```
>>> D
```

```
{'a': 3}
```


Supprimer une entrée d'un dictionnaire

- L'opérateur **del** permet de **supprimer** une association d'un dictionnaire:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> del D['a']
>>> D
{'b': 2, 'c': 3}
```

Vérifier l'existence d'une entrée dans un dictionnaire

- Pour vérifier s'il existe une valeur associée à une clé donnée, on utilise l'opérateur **in** comme dans le cas des listes:

```
>>> prix = {'asus': 450, 'alienware': 1200, 'lenovo': 680}
```

```
>>> 'asus' in prix
```

```
True
```

```
>>> 'toshiba' in prix
```

```
False
```

ATTENTION ! L'opérateur **in** vérifie l'existence d'une clé, et non pas d'une valeur. Exemple:

```
>>> 1200 in prix
```

```
False
```

KeyError: Clé introuvable

- Si on tente d'accéder à une **entrée qui n'existe pas** dans le dictionnaire, le programme renvoie une **erreur de clé (KeyError)**, exemple:

```
>>> lettres = {'a': 103, 'b': 8, 'e': 150}
```

```
>>> lettres['k']
```

```
KeyError: 'k'
```

```
>>> lettres['u'] = lettres['u'] + 1
```

```
KeyError: 'u'
```

```
>>> del lettres['j']
```

```
KeyError: 'j'
```

- Avant d'accéder à une valeur, on prendra l'habitude de toujours vérifier l'existence de la clé:

```
if 'u' in lettres:
```

```
    lettres['u'] = lettres['u'] + 1
```

```
else:
```

```
    lettres['u']=1
```

Parcourir un dictionnaire

- La boucle for peut être utilisée pour parcourir toutes les clés d'un dictionnaire:

```
for key in D:  
    print('La clé', key, 'a pour valeur: ', D[key])
```

- Exemple:

```
dates_naissance=  
    {'ingrid':[12,6,1995], 'marc':[27,8,1996], 'brice':  
    [11,10,1995]}  
for nom in dates_naissance :  
    date = dates_naissance[nom]  
    print(nom, 'fetera son anniversaire le ',  
          date[0], '/', date[1], '/2017')
```

→ Affiche:

```
ingrid fetera son anniversaire le 12 / 6 /2017  
marc fetera son anniversaire le 27 / 8 /2017  
brice fetera son anniversaire le 11 / 10 /2017
```

Quels types pour les clés et valeurs ?

- Comme dans le cas des listes, les valeurs dans un dictionnaire peuvent être de n'importe quel type, y compris le type dictionnaire.

```
>>> mon_pc = {
    'ram': 16,
    'cpu': 3.5,
    'portable': False,
    'os': 'windows',
    'ports': ['usb3.0', 'jack', 'ethernet', 'hdmi'],
    'carte_graphique': {
        'vram': 4,
        'nom': 'gtx970',
        'bus': 256
    }
}
```

- En revanche, seuls certains types peuvent être utilisés comme **clés**. Dans ce cours on se limitera aux **entiers** et aux **chaines de caractères**.

Copie de dictionnaires

- Comme dans le cas des listes, l'affectation d'un dictionnaire vers une variable ne fait que référencer le même dictionnaire, exemple:

```
>>> D = {1: 10, 2: 20, 3: 30}
>>> E = D
>>> E[5] = 50
>>> E
{1: 10, 2: 20, 3: 30, 5: 50}
>>> D
{1: 10, 2: 20, 3: 30, 5: 50}
```

- Pour créer une copie d'un dictionnaire, on utilise `dict()`:

```
>>> F = dict(D)
>>> F[6] = 60
>>> F
{1: 10, 2: 20, 3: 30, 5: 50, 6: 60}
>>> D
{1: 10, 2: 20, 3: 30, 5: 50}
```

THÈME 10: LECTURE/ECRITURE DANS UN FICHER

Introduction aux fichiers

- Jusqu'à présent, nous avons utilisé `input()` et `print()` pour lire les entrées du programme, et afficher les résultats obtenus.
- Parfois, les données d'entrée sont stockées dans un fichier et on aimerait pouvoir y accéder directement, sans les saisir manuellement au clavier.
- Il est aussi souvent utile de sauvegarder nos résultats dans des fichiers afin de pouvoir y accéder plus tard. Exemple: Sauvegarde d'une partie dans un jeu vidéo.
- En python, il est très facile de lire et d'écrire des données dans des fichiers.

Accès à un fichier texte

- Avant de commencer la lecture d'un fichier, il faut d'abord **l'ouvrir**. Ouvrir un fichier veut simplement dire que l'on crée une variable qui permet de le manipuler.
- La fonction **open()** est utilisée pour ouvrir un fichier. Par exemple, pour ouvrir le fichier appelé « data.txt », il suffit de faire:

```
>>> f=open('data.txt')
```

- Par défaut, open() ouvre un fichier en mode « **lecture** », c'est-à-dire qu'on ne peut pas modifier son contenu.
- Si on tente d'ouvrir un fichier inexistant en mode « lecture », on reçoit une erreur:

```
>>> f = open('toto') # le fichier 'toto' n'existe pas
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'toto'
```

Lecture d'un fichier texte

- Une fois notre fichier texte ouvert, il existe plusieurs manières de lire son contenu:

- Lire en une fois tout le contenu dans une chaîne de caractères:

```
texte = f.read() # la chaîne texte contient tout le texte  
du fichier
```

- Lire en une fois toutes les lignes du fichier dans une liste de chaînes de caractères:

```
lignes = f.readlines() # Lignes est une liste qui  
contient les lignes du fichier: lignes[0] contient la  
première ligne, etc.
```

- Lire le fichier ligne par ligne dans une boucle **for**:

```
for ligne in f:  
    print(ligne) # affiche une ligne du fichier
```

Écriture dans un fichier texte

- Pour pouvoir écrire dans un fichier, il faut l'ouvrir en mode écriture:
`f = open('fichier.txt', 'w')`
- Si le fichier ouvert en mode « écriture » n'existe pas, il sera créé. S'il existe déjà, tout son contenu sera effacé.
- La fonction permettant d'écrire dans un fichier texte est **write()**.
`f.write('ce texte sera écrit dans le fichier')`
- Contrairement à `print()`, la fonction `write()` ne saute pas de ligne automatiquement. Pour sauter de ligne dans le fichier, il faut écrire un saut de ligne manuellement:
`f.write('\n')` # ceci permet de sauter la ligne
- L'argument passé à `write()` doit obligatoirement être une chaîne de caractères. Pour écrire un entier ou un autre type, il faut le convertir en chaîne de caractères en utilisant **str()**.

Fin de la manipulation

- Une fois la lecture/écriture terminée, il faut *fermer* le fichier en utilisant la fonction `close()`.

```
f = open('fichier.txt')
for ligne in f:
    print('une ligne lue :', ligne)
f.close()
```

Exemple de lecture

- Soit le fichier « nombres.txt » ci-contre qui contient des entiers (un par ligne).
- On veut calculer la somme de ces entiers:

```
fichier = open('nombres.txt')
somme = 0
for nombre in fichier:
    # nombre est une chaine, ne pas
    # oublier de la convertir en entier !
    somme = somme + int(nombre)
print(somme)
fichier.close()
```

nombres.txt
15
18
30
55
16
3
12
13

Exemple d'écriture

- On veut sauvegarder les 10 premières puissances de 2 dans un fichier « puis.txt »:

```
fichier = open('puis.txt', 'w')
for i in range(0,10):
    # ne pas oublier de convertir en chaine
    # de caractères et de sauter de ligne!
    fichier.write(str(2 ** i) + '\n')
fichier.close()
```

puis.txt
1
2
4
8
16
32
64
128
256
512

- Ces slides ont été réalisés par:
 - Amir Charif
 - Lydie Du Bousquet
 - Aurélie Lagoutte
 - Julie Peyre
- Leur contenu est placé sous les termes de la licence **Creative Commons CC BY-NC-SA**

