



جامعة السلطان مولاي
Slimane
+٢٠٦٠٣٤٣ ٩٥٨٠٦١
Université Sultan Moulay Slimane

Université Sultan Moulay Slimane

Faculté Polydisciplinaire Béni Mellal

Département de Mathématiques et Informatique

Master ISI



PYTHON 

Anouar DARIF

anouar.darif@gmail.com

Le langage Python

- **Python** est un langage de programmation **interprété**, c'est-à-dire que les instructions que vous lui envoyez sont « transcrives » en langage machine au fur et à mesure de leur lecture.
- Développé en 1989 par Guido van Rossum
- Open-source
- Fonctionne sur tous les systèmes d'exploitation : Windows, Mac Os, Linux, BeOS, ...
- Extensible
 - Il est possible de développer ses propres modules, ses propres bibliothèques.
- support pour l'intégration d'autres langages
- Typage Dynamique
 - ✓ Pas de déclaration de typage au moment de la compilation

Le langage Python

- Python intègre la notion de module
 - Python permet de décomposer un programme en modules qui peuvent être réutilisés ultérieurement.
 - Python accède à un très grand nombre de bibliothèques.
- Python est orienté objet (POO)
 - Supporte l'héritage multiple
 - Pas de mécanisme d'encapsulation
 - Pas de destructeur
 - Pas de surcharge de constructeurs ni de méthodes
 - Surcharge des opérateurs
 - ...
- Python est langage de plus en plus utilisé

Pourquoi le choix du Python ?

- Il possède une **syntaxe simple**.
- On peut donc se concentrer sur la **logique algorithmique**.
- Le mode “**console**” permet de tester des portions de codes immédiatement.
- C'est un **langage interprété**, c'est-à-dire que les instructions des programmes seront exécutées au fur et à mesure.

Pourquoi le choix du Python ?

- C'est un **langage de haut niveau**, on ne souciera pas de la gestion des ressources mémoires.
- Python possède des **structures de données évoluées**, listes, ensembles, dictionnaires, etc.
- Python est de plus en plus **populaire** et possède de nombreuses ressources bibliographiques.
- Enfin, c'est un langage **open source**.

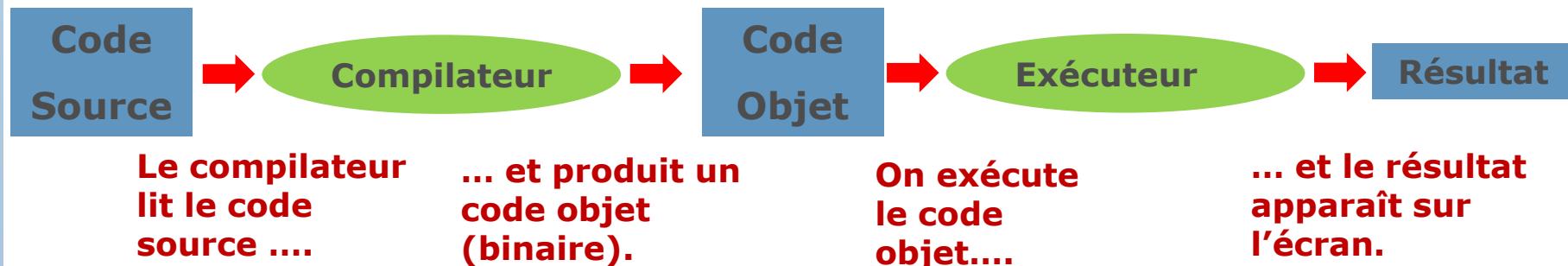
Comment faire fonctionner mon code source ?

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

- **Interprétation**

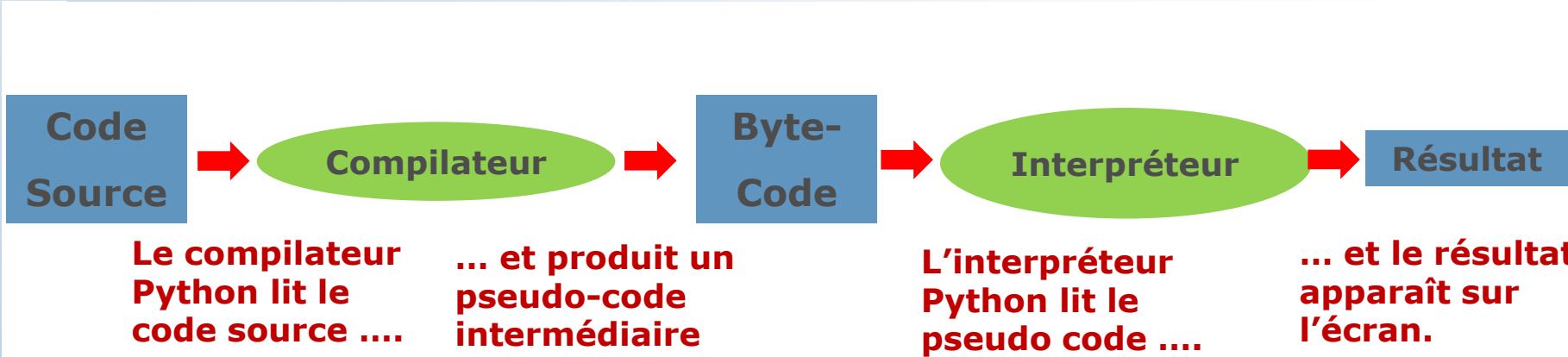


- **Compilation**



Présentation de Python

Et Python



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

- **Inconvénients :**
 - peut être lent.

Les différentes versions

□ Python ou CPython (C/C++)

- Les programmes Python sont compilés en code intermédiaire (code portable) puis interprétés sur la machine cible
 - Version 2.7.4 (6 avril 2013)
 - Version 3.2.4 (7 avril 2013)

□ Jython (2.5.3)

- Écrit en Java pour une JVM
 - Génère du byte code Java

□ IronPython (2.7.3)

- Écrit en C#
 - Code compilé puis de nouveau compilé sur la machine JIT (Just In Time)

Que peut-on faire avec Python ?

- du calcul scientifique (librairie NumPy)
- des graphiques (librairie matplotlib)
- du traitement du son, de la synthèse vocale (librairie eSpeak)
- du traitement d'image (librairie PIL)
- des applications avec interface graphique GUI
(librairies Tkinter, PyQt, wxPython, PyGTK...)
- des jeux vidéo en 2D (librairie Pygame)
- des applications Web (serveur Web Zope ; frameworks Web Flask, Django)
- interfaçer des systèmes de gestion de base de données
(librairie MySQLdb...)
- des applications réseau (framework Twisted)
- communiquer avec des ports série RS232 (librairie PySerial), en Bluetooth (librairie pybluez)...
- ...
- Des **dizaines de milliers** de librairies sont disponibles sur le dépôt officiel PyPI.

Quelle version de Python ?

- Il y a principalement deux “**branches**” différentes de Python : **2.x** et **3.x**
- Même si elles sont assez semblables dans l'esprit, on peut noter d'importantes différences de **syntaxes**, **fonctionnalités**, etc.
- Ce cours est basé sur la branche **3.x**, et plus particulièrement sur la version **3.3.5**

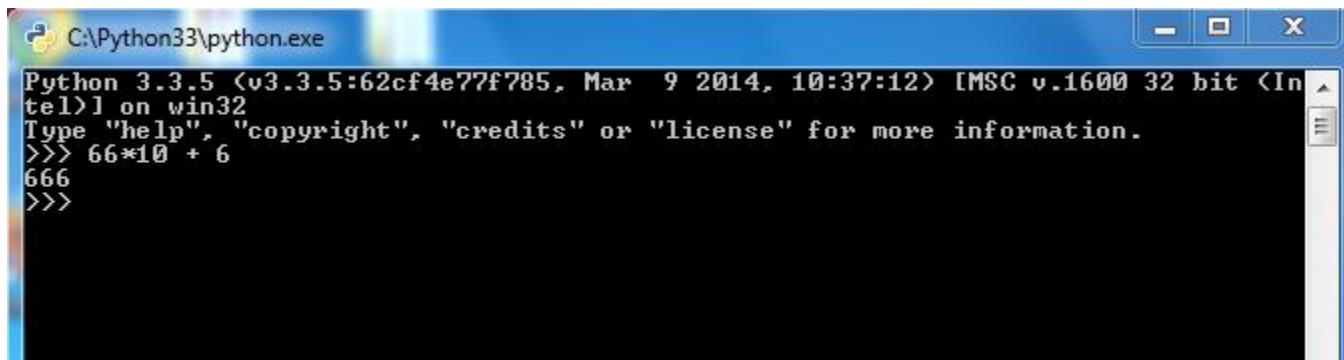
Utilisation de la console.

- Le premier mode d'utilisation de Python est un **mode interactif**, style “calculatrice”.
- Il permet d'exécuter du code “**à la volée**”.
- L'intérêt étant de pouvoir **tester** facilement des petites portions de code, de vérifier la justesse d'une syntaxe, etc.
- Ou même de réaliser des **calculs**.

Utilisation de la console.

Sous Windows : au choix

- à partir du menu “démarrer”, répertoire ‘Python 3.3’ lancer “Python (command line)”.
- À partir du répertoire “c:/Python33” lancer “Python.exe”.

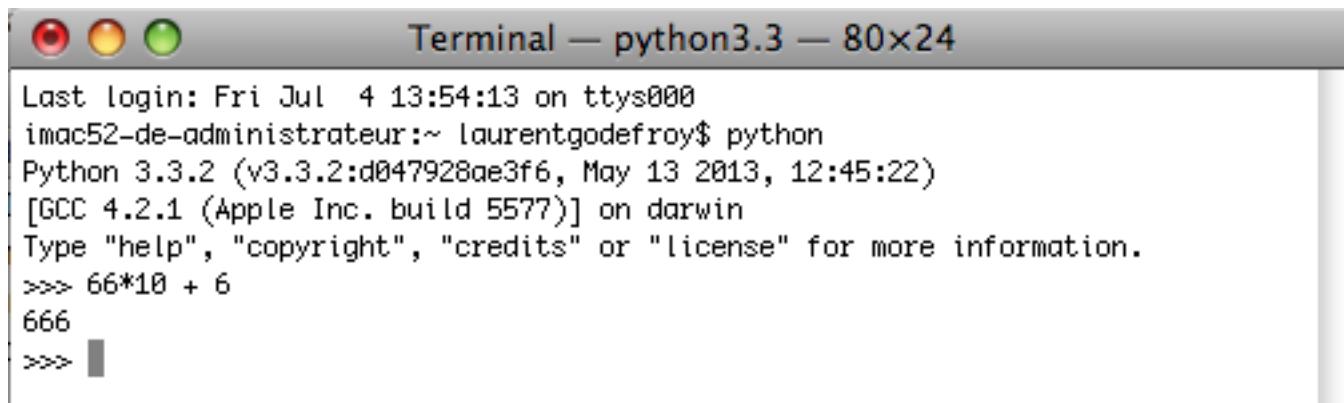


The screenshot shows a Windows command prompt window titled "C:\Python33\python.exe". The window displays the Python 3.3.5 startup message, which includes the version number, build date (Mar 9 2014), build time (10:37:12), and architecture (MSC v.1600 32 bit (Intel) on win32). It also shows the standard help text: "Type "help", "copyright", "credits" or "license" for more information." Below this, a user has entered the expression "66*10 + 6" and received the result "666". The command prompt prompt is visible at the bottom of the window.

Utilisation de la console.

Sous Mac :

- Dans “Applications/Utilitaires” ouvrir le “Terminal”.
- Taper “python”.



```
Last login: Fri Jul  4 13:54:13 on ttys000
imac52-de-administrateur:~ laurentgodefroy$ python
Python 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 12:45:22)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 66*10 + 6
666
>>>
```

Utilisation de la console.

Console d'IDLE sous Windows : au choix

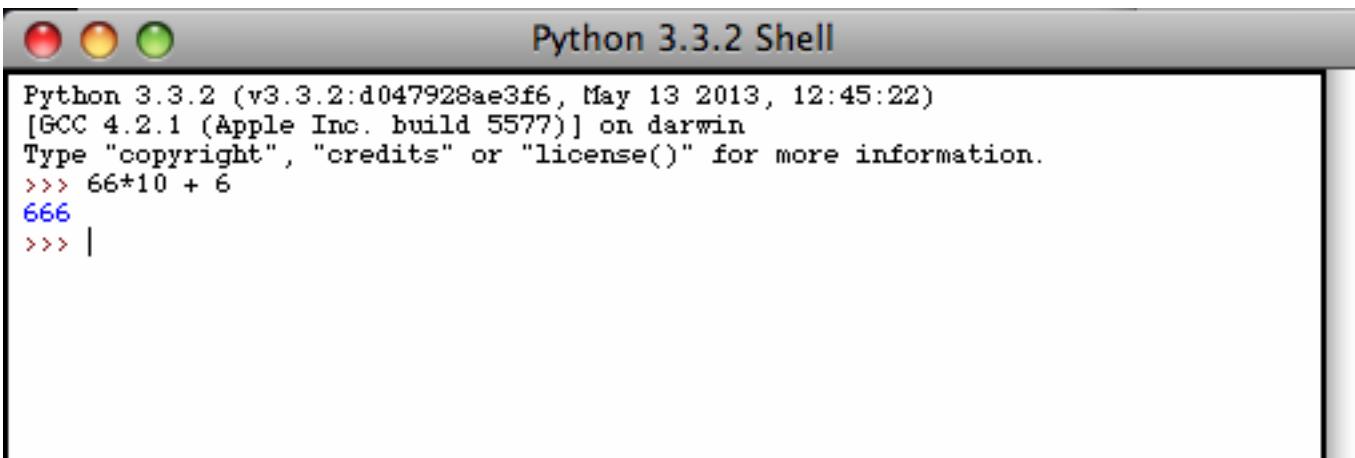
- à partir du menu “démarrer”, répertoire ‘Python 3.3” lancer “IDLE (Python GUI)”.
- À partir du répertoire “c:/Python33/lib/idlelib” lancer “idle.pyw”.

Console d'IDLE sous Mac :

- Dans “Applications/Python 3.3” lancer IDLE.

Utilisation de la console.

La console d'IDLE :

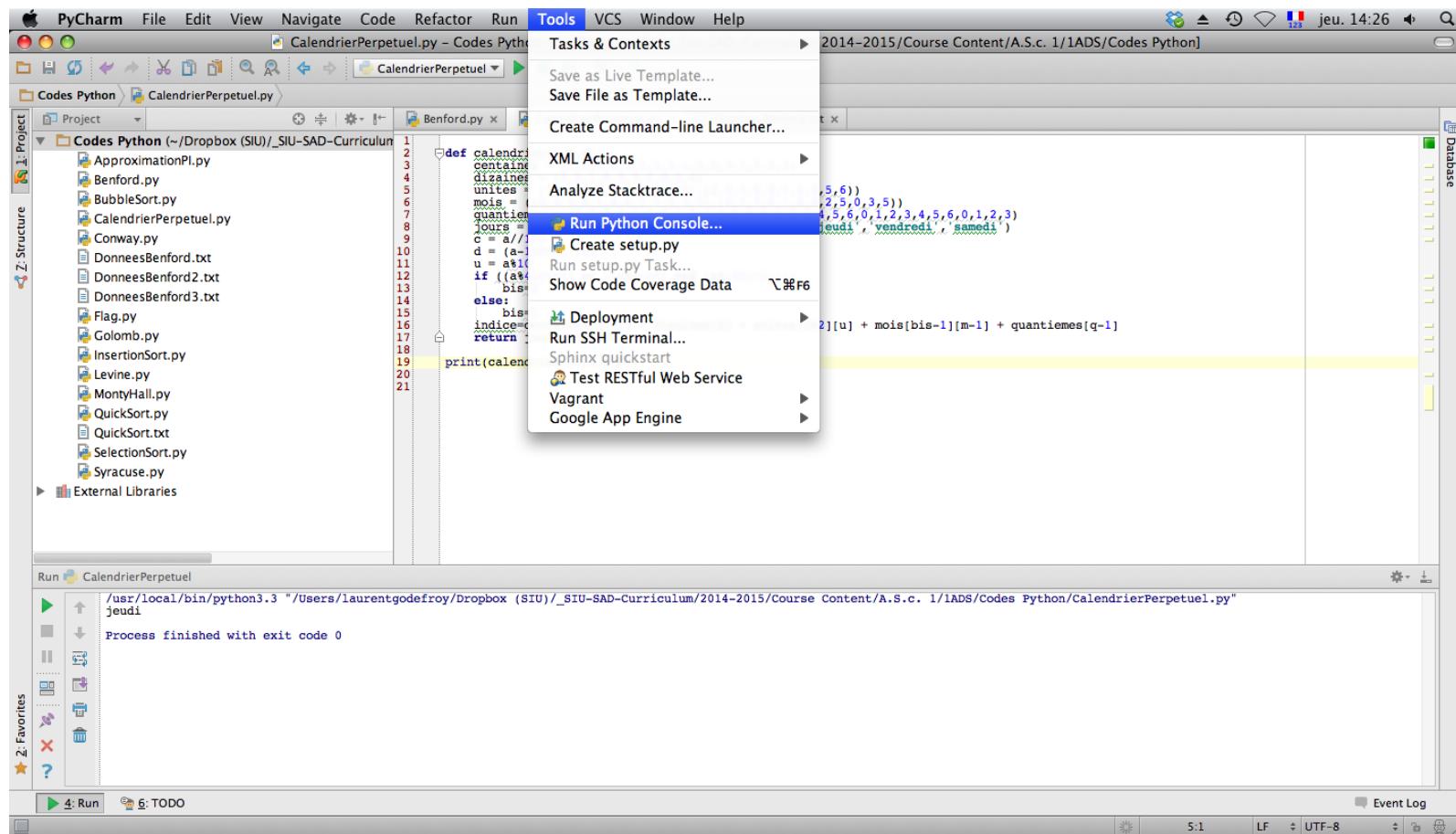


The screenshot shows a window titled "Python 3.3.2 Shell". The title bar has three colored buttons (red, yellow, green) on the left. The main area displays the following text:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 12:45:22)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> 66*10 + 6
666
>>> |
```

Utilisation de la console.

À partir de Pycharm : menu “tools”, “Run Python Console”



Utilisation de la console.

À partir de Pycharm (suite) :

The screenshot shows the PyCharm IDE interface with the following details:

- Project:** Codes Python (~Dropbox (SIU) /_SIU-SAD-Curriculum/2014-2015/Course Content/A.S.c. 1/1ADS/Codes Python)
- Code Editor:** The file `CalendrierPerpetuel.py` is open, displaying the following code:

```
1 def calendrier(a,m,q):
2     centaines = (1,0,5,3,1,0,5,3,1,0,5,3,1,0,5,3)
3     dizaines = (4,2,1,6,5,3,2,0,6,4)
4     unites = ((2,3,4,5,0,1,2,3,5,6),(2,3,5,6,0,1,3,4,5,6))
5     mois = ((0,3,5,6,1,4,6,2,5,0,3,5),(6,2,3,6,1,4,6,2,5,0,3,5))
6     quantiemes = (1,2,3,4,5,6,0,1,2,3,4,5,6,0,1,2,3,4,5,6,0,1,2,3)
7     jours = ('dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi')
8
9     a = a%100
10    c = a//100
11    d = (a-100*c)//10
12    u = a%10
13    if ((a%400==0) or ((a%4==0) and (a%100!=0))):
14        bis=2
15    else:
16        bis=1
17    indice=centaines[c-15] + dizaines[d] + unites[d%2][u] + mois[bis-1][m-1] + quantiemes[q-1]
18    return jours[indice%7]
19
20
21 print(calendrier(2014,7,8))
```

- Run Tab:** Python Console tab is selected.
- Python Console Output:**

```
/usr/local/bin/python3.3 -u /Applications/PyCharm.app/helpers/pydev/pydevconsole.py 61754 61755
PyDev console: starting.
>>> import sys; print('Python %s on %s' % (sys.version, sys.platform))
Python 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 12:45:22)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
>>> sys.path.append('/Users/laurentgodefroy/dropbox (SIU) /_SIU-SAD-Curriculum/2014-2015/Course Content/A.S.c. 1/1ADS/Codes Python')
>>> 66*10 + 6
666
>>>
```

Écriture de scripts.

- Le mode console ne permet pas de **sauvegarder** des programmes.
- Pour **écrire un script**, un simple éditeur de texte peut suffire mais on préfèrera utiliser un environnement de développement afin de bénéficier de fonctionnalités supplémentaires : **coloration syntaxique**, **auto-complétion**, etc.

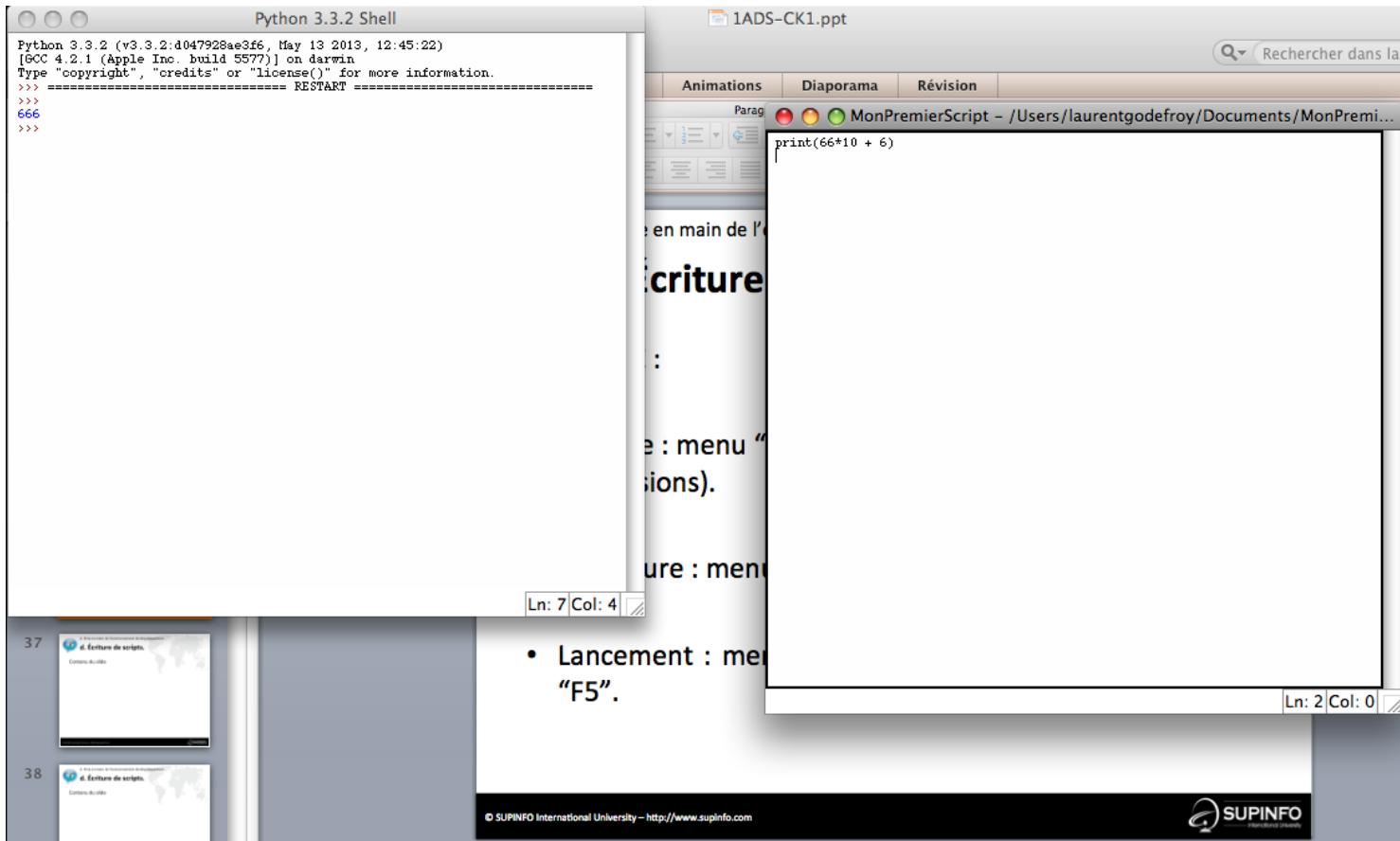
Écriture de scripts.

Avec IDLE :

- Écriture : menu “File”, “New File” (ou “New window” selon les versions).
- Ouverture : menu “File”, “Open”.
- Lancement : menu “Run”, “Run Module” ou alors touche “F5”.

Écriture de scripts.

Avec IDLE (suite) :



The image shows a presentation slide with two windows overlaid. On the left is a 'Python 3.3.2 Shell' window showing the Python interpreter prompt and some code. On the right is a Microsoft Word document titled '1ADS-CK1.ppt'. The Word document contains text about writing scripts and includes a screenshot of the Python shell and a screenshot of a presentation slide. A callout box points to the screenshot in the Word document with the text: '• Lancement : menu "F5".'

Python 3.3.2 Shell

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 12:45:22)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
666
>>>
```

1ADS-CK1.ppt

Animations Diaporama Révision

MonPremierScript – /Users/laurentgodefroy/Documents/MonPremi...

```
print(66*10 + 6)
```

Ln: 7 Col: 4

Ln: 2 Col: 0

37

38

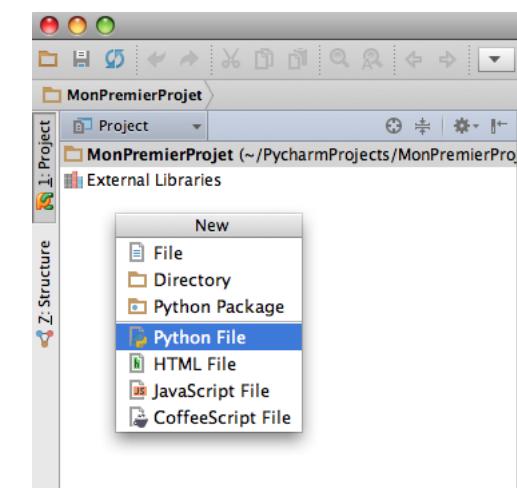
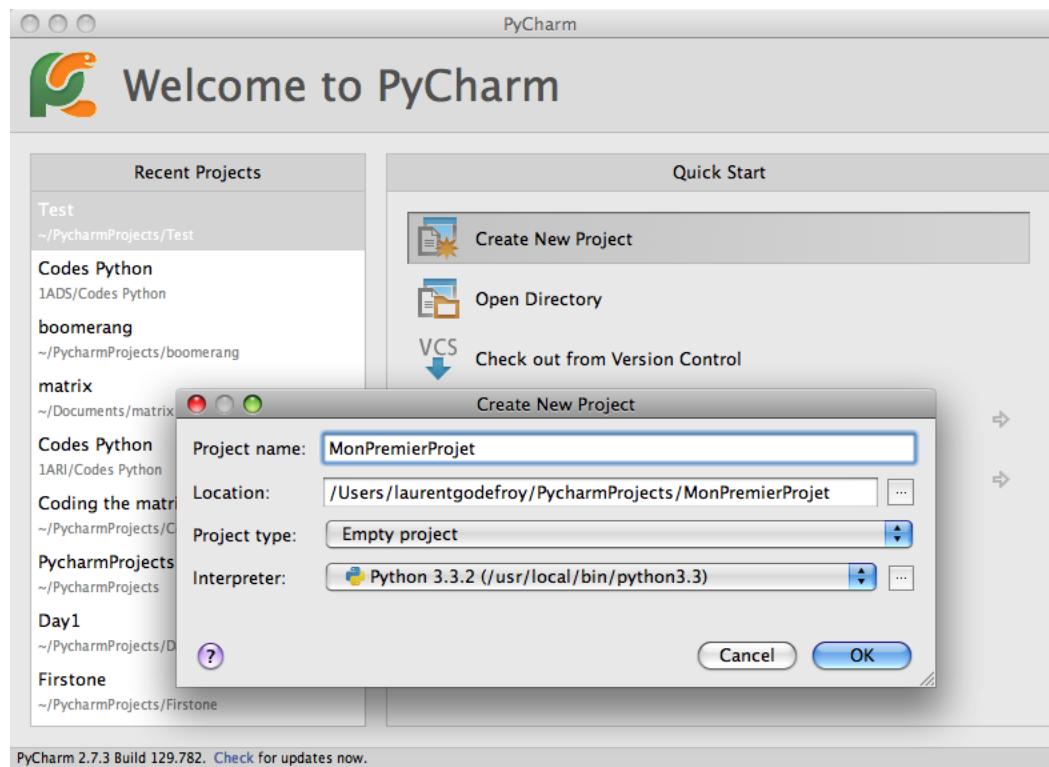
• Lancement : menu "F5".

© SUPINFO International University – <http://www.supinfo.com>

 SUPINFO

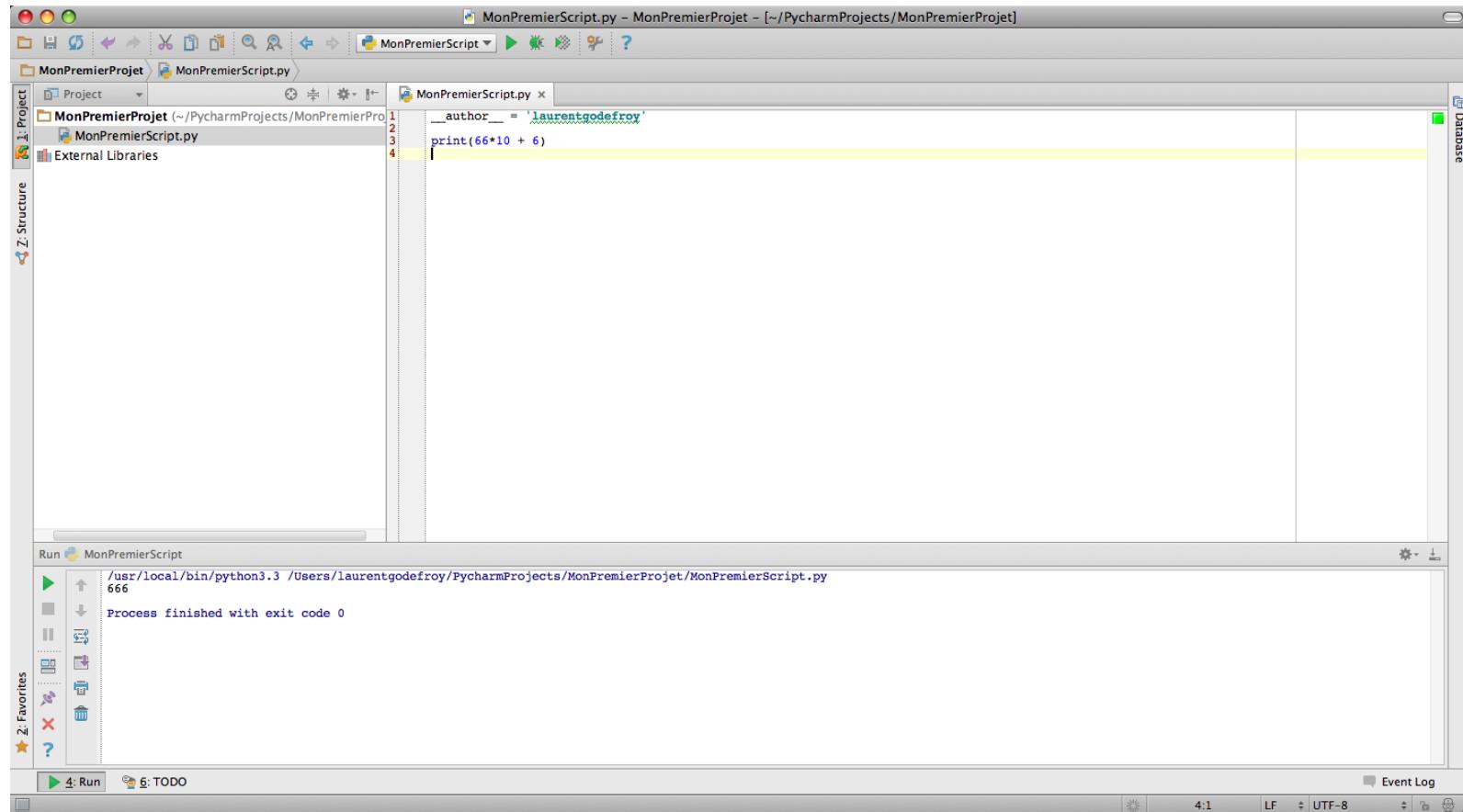
Écriture de scripts.

Avec **Pycharm** : menu “File”, “New Project”, puis menu “File”, “New”, “Python File”.



Écriture de scripts.

Avec Pycharm (suite) :



The screenshot shows the PyCharm IDE interface. The top bar displays the project name "MonPremierProjet" and the file "MonPremierScript.py". The left sidebar shows the project structure with "MonPremierProject" and "MonPremierScript.py". The main editor window contains the following Python code:

```
author__ = 'laurentgodefroy'  
print(66*10 + 6)
```

The bottom panel shows the "Run" tab with the output of running the script:

```
/usr/local/bin/python3.3 /Users/laurentgodefroy/PycharmProjects/MonPremierProjet/MonPremierScript.py  
666  
Process finished with exit code 0
```

Variables.

- Un algorithme ou un programme manipulent des **données**. Certaines sont connues dès le départ, d'autres sont calculées lors de son exécution.
- Pour pouvoir manipuler ces données il faut **garder leurs valeurs en mémoire**.
- C'est le rôle des **variables**.

Variables.

Variable :

- **Nom désignant une donnée** (nombre, texte...) susceptible de changer de valeur.

- Une variable possède un **type**, ce qui permet à l'ordinateur de savoir quelles valeurs elle peut prendre et quelles opérations on peut effectuer avec.

Variables.

Types “élémentaires” en Python :

- **int** : nombres entiers.
- **float** : nombres décimaux.
- **complex** : nombres complexes.
- **bool** : booléens (deux valeurs possibles “**True**” et “**False**”).
- **str** : chaînes de caractères.

Spécificités du Python.

Typage dynamique :

- Avant d'utiliser une variable on n'a pas besoin de déclarer explicitement son type, c'est l'interpréteur qui s'en charge.
- Cela sera fait dès que l'on attribuera une valeur à notre variable.

Spécificités du Python.

Typage fort :

- Les opérations possibles sur une variable dépendent intégralement de son type.

- Les conversions de types implicites afin de réaliser certaines opérations sont donc interdites.

Affectation de variables.

■ **Syntaxe** : affectation simple

```
maVariable = valeur
```

■ **Syntaxe** : affectation multiple

```
var1, var2, ... = val1, val2, ...
```

Affectation de variables.

Remarques :

- Comme mentionné précédemment, lorsque l'on veut utiliser une variable il n'y a donc pas besoin de déclarer son type. C'est lors de son **initialisation** que ce typage s'effectue.
- On peut toujours vérifier le type d'une variable avec l'instruction "**type**" :

```
type (maVariable)
```

Affectation de variables.

Exemple :

```
>>> i = 4
>>> type(i)
<class 'int'>
>>> x, z = -5.3, 1+1j
>>> z
(1+1j)
>>> type(z)
<class 'complex'>
>>> type(x)
<class 'float'>
```

Affectation de variables.

Exemple :

```
>>> texte = 'Street fighting man'  
>>> type(texte)  
<class 'str'>  
>>> texte  
'Street fighting man'  
>>> '666' + 111  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: Can't convert 'int' object to str  
implicitly
```

Conversions de types.

Opérations de conversions :

Opération	Contrainte / Résultat
int(x)	x est un décimal (qui sera tronqué) ou une chaîne
int(x,base)	x est une chaîne et base un entier
float(x)	x est un entier ou une chaîne
complex(x,y)	x et y sont des nombres ou x est une chaîne
bool(x)	x est un nombre ou un booléen Vaut False uniquement si x vaut 0 ou ''
str(x)	x est un nombre ou un booléen
eval(x)	x est une chaîne qui est évaluée comme une expression Python

Conversions de types.

Exemple :

```
>>> x = 666.6
>>> y = int(x)
>>> y
666
>>> eval('y+111')
777
>>> int('10101',2)
21
>>> str(y)
'666'
```

Affichage et saisie.

- En mode “**console**” on peut afficher le contenu d'une variable en tapant juste son nom (voir exemples précédents).

- Pour réaliser un affichage sur la console à partir d'un script, et/ou pour réaliser des affichages plus sophistiqués on aura besoin de l'instruction “**print**”.

Affichage et saisie.

Syntaxe de la fonction “print” :

```
print(expr_1,expr_2,...,sep=' ',end='\n')
```

- Les différentes expressions (en nombre variables) sont soit des chaînes, soit des résultats de calculs convertis en chaînes, soit des variables dont on souhaite afficher la valeur.

Affichage et saisie.

Syntaxe de la fonction “print” : (suite)

- “**sep**” indique ce qui va séparer ces expressions, par défaut il s'agit d'un espace.
- “**end**” indique ce qui sera affiché après toutes les expressions, par défaut il s'agit d'un retour à ligne.

Affichage et saisie.

Exemple :

```
>>> age = 30
>>> print("J'ai",age,"ans et dans deux ans
           j'aurais",age+2,"ans")
J'ai 30 ans et dans deux ans j'aurais 32 ans
```

Affichage et saisie.

Exemple : (suite)

```
print("des","traits","bas","pour","séparer",
      sep='_')
print("et","des","plus","à","la","suite",
      sep='_',end='+++ ')
print("et la fin")
```



```
des_traits_bas_pour_séparer
et_des_plus_à_la_suite +++ et la fin
```

Affichage et saisie.

Remarque

- Tabulation et retour à la ligne s'obtiennent avec '**\t**' et '**\n**' :

```
>>> print(« Ecole \tSupMTI Rabat»)
Ecole      SupMTI Rabat
>>> print("Ecole SupMTI \nRabat")
Ecole SupMTI
Rabat
```

Affichage et saisie.

Syntaxe de la fonction “input” :

```
var = input(expression)
```

- “expression” est facultative, elle sert juste à réaliser un affichage sur la ligne de la saisie.
- Ce que va saisir l’utilisateur sera affecté sous forme de chaîne de caractère à la variable “var”.

Affichage et saisie.

Exemple :

```
>>> a = input()
>? 12
>>> print(a,type(a))
12 <class 'str'>
>>> b = eval(input("saisir un nombre : "))
saisir un nombre : >? 12
>>> print(b,type(b))
12 <class 'int'>
```

Opérations arithmétiques.

Liste des principales opérations arithmétiques :

Opération	Résultat
$x + y$	Addition de x et y
$x - y$	Soustraction de y à x
$x * y$	Multiplication de x et y
$x ** y$	Élévation de x à la puissance y
x / y	Quotient réel de x par y
$x // y$	Quotient entier de x par y
$x \% y$	Reste du quotient entier de x par y

Opérations arithmétiques.

Précisions :

- Si les deux opérandes des opérations $+$, $-$, $*$, $**$ sont entiers (resp. réels) le résultat est entier (resp. réel).
- Si un des deux opérandes est réel, le résultat est réel.
- Le résultat de l'opération $/$ est toujours un réel.

Opérations arithmétiques.

Précisions : (suite)

- Si les deux opérandes sont des entiers, le résultat de l'opération **//** est le quotient de la division euclidienne et le résultat de l'opération **%** le reste.

- Si l'un des deux opérandes est un réel, le résultat de l'opération **//** est la partie entière du quotient de la division réelle.

Opérations arithmétiques.

Exemple :

```
>>> -3 + 5  
2  
>>> 3 * 5  
15  
>>> 5 ** 2  
25  
>>> 13 / 4  
3.25
```

Opérations arithmétiques.

Exemple : (suite)

```
>>> 13 // 4
3
>>> 13 % 4
1
>>> 13 // 4.6
2.0
>>> 13 % 4.6
3.8000000000000007
```

Opérations arithmétiques.

Quelques raccourcis :

Opération	Résultat
$x += y$	$x = x + y$
$x -= y$	$x = x - y$
$x *= y$	$x = x * y$
$x **= y$	$x = x ** y$
$x /= y$	$x = x / y$
$x //= y$	$x = x // y$
$x %= y$	$x = x \% y$

Opérations arithmétiques.

Exemple :

```
>>> x = 25
>>> y = 0.5
>>> x **= y
>>> print(x)
5.0
>>> y += 1
>>> print(y)
1.5
```

Opérations arithmétiques.

Module “math” :

- Pour effectuer des calculs mathématiques plus “élaborés”, on pourra utiliser un module de fonctions complémentaires.
- Avant de l'utiliser, il convient de l'**importer** :

```
from math import *
```

Opérations arithmétiques.

Exemple :

```
>>> from math import *
>>> exp(1)
2.718281828459045
>>> pi
3.141592653589793
>>> cos(pi/2)
6.123233995736766e-17
```

Les identifiants

- Les identifiants sont sensibles à la casse et ne doivent pas être un mot clé.
- Un identifiant commence toujours par une lettre.
- En Python 3, on peut utiliser des lettres du code Unicode mais pas en Python 2, uniquement des lettres non accentuées du code ASCII.
- Convention de nommage
 - Les constantes en majuscules ;
 - Les classes de style : **NomDeLaClasse** ;
 - Les fonctions (méthodes et interfaces graphiques) : **nomDeLaFonction** ;
 - Les autres identifiants : **nomidentifiant** ou **nom_identifiant**.

Mots clés de Python

- Certains mots-clés de Python sont **réservés**, c'est-à-dire que vous ne pouvez pas créer des variables portant ce nom :
- La version 3.3.1 de Python compte 33 mots clés :

and	del	from	None	True
as	elif	global	nonlocal	try
assert	if	else	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Les types et les opérations de base

Conversion de base

- Conversion en base 10

```
>>> int('AB',16)  
171  
>>> int('0xAA',16)  
170  
>>> int('53',8)  
43  
>>> int('10101',2)  
21
```

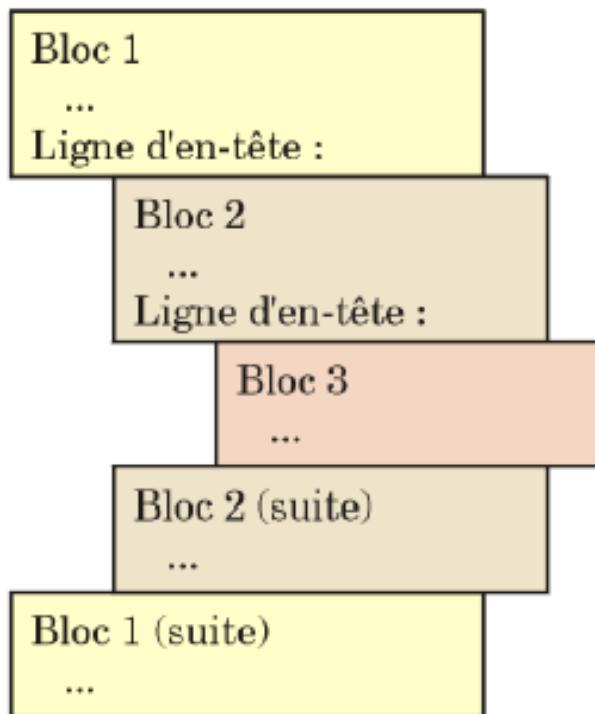
- Conversion de la base 10

```
>>> hex(22)  
'0x16'  
>>> oct(22)  
'0o26'  
>>> bin(22)  
'0b10110'
```

Les structures de contrôle

Indentation

Les blocs de code (**fonctions**, **instructions if**, **boucles for ou while** etc.) sont définis par leur indentation. L'indentation démarre le bloc et la désindentation le termine. Il n'y a pas d'accolades, de crochets ou de mots clés spécifiques. Cela signifie que les espaces blancs sont significatifs et qu'ils doivent être cohérents.



a = -150

if a<0:

print 'a est négatif'

Ligne d'en-tête:

première instruction du bloc

...

dernière instruction du bloc

Structures conditionnelles

Test simple

- L'idée de base est donc la réalisation d'une ou plusieurs actions selon la **vérification d'une condition**.
- Cela correspond au **SI ... ALORS ...** du langage courant.
- En mathématiques on utilise également le terme d'**implication**.

Syntaxe d'un test simple

```
if conditionnelle:  
    bloc d'instructions à exécuter si la  
    conditionnelle est vraie
```

Remarque plus qu'importante :

En python les **blocs d'instructions** sont délimités de deux façons :

- 1.La ligne les précédant se termine par un **double point** :
- 2.L'intégralité du bloc est **indenté** par rapport aux instructions qui le précèdent et le suivent.

Structures conditionnelles

Test simple

Exemple de test simple :

remplacer un nombre par sa valeur absolue

```
x = eval(input())
if x < 0:
    x = -x
print(x)
```



-66.6
66.6

Structures conditionnelles

Test avec alternative(s).

- Cette fois-ci, on pourra avoir **une ou plusieurs alternatives** à la conditionnelle si cette dernière n'est pas vérifiée.
- On va donc simuler des raisonnements de la forme

SI ... ALORS ...

SINON SI ... ALORS ...

SINON ...

Syntaxe d'un test avec une alternative :

```
if conditionnelle:  
    bloc d'instructions à exécuter si la  
    conditionnelle est vraie  
else:  
    bloc d'instructions à exécuter si la  
    conditionnelle est fausse
```

Structures conditionnelles

Test avec alternative(s)

Exemple de test avec une alternative :

savoir si un nombre est pair ou non

```
n = eval(input())
if n%2 == 0:
    print("le nombre saisi est pair")
else:
    print("le nombre saisi est impair")
```

666
le nombre saisi est pair

667
le nombre saisi est impair



Structures conditionnelles

Test avec alternative(s)

Syntaxe d'un test avec plusieurs alternatives :

```
if conditionnelle:  
    bloc d'instructions à exécuter si la  
    conditionnelle est vraie  
elif autre_conditionnelle:  
    bloc d'instructions à exécuter si la      seconde  
    conditionnelle est vraie  
else:  
    bloc d'instructions à exécuter si les      premières  
    conditionnelles sont fausses
```

Exemple de test avec plusieurs alternatives : signe strict d'un nombre

```
x = eval(input())  
if x < 0:  
    print("le nombre saisi est strictement négatif")  
elif x > 0:  
    print("le nombre saisi est strictement positif")  
else:  
    print("le nombre saisi est nul")
```

Structures conditionnelles

Imbrication de tests

- L'**imbrication** de structures conditionnelles est non seulement possible mais souvent fort utile.
- Cela peut éviter le recours à une succession de tests simples et donc rendre les codes plus lisibles.

Exemple : équation du premier degré $ax + b = 0$

```
a,b = eval(input()),eval(input())
if a == 0:
    if b == 0:
        print("infinité de solutions")
    else:
        print("pas de solution")
else:
    print("unique solution :" ,-b/a)
```

Structures conditionnelles

Opérateur ternaire.

- Cet opérateur permet dans certains cas une **syntaxe plus compacte** d'un test avec une alternative.
- Néanmoins son usage est beaucoup plus limité puisque les blocs à exécuter selon que la conditionnelle soit vraie ou fausse ne doivent comporter qu'une **seule instruction**.
- Plus exactement les deux alternatives doivent être le calcul d'une **expression**.

Syntaxe de l'opérateur ternaire :

```
expression_1 if conditionnelle else expression_2
```

Exemples : majeur ou mineur ?

```
statut = "mineur" if age < 18 else "majeur"
```

```
print("mineur") if age < 18 else print("majeur")
```

Structures conditionnelles

Exercices

Exercice : écrire un programme calculant les maximums et minimums de deux nombres.

Solution 1 : avec un test à une alternative

```
x,y = eval(input()),eval(input())
if x > y:
    mini,maxi = y,x
else:
    mini,maxi = x,y
print("minimum :",mini,", maximum :",maxi)
```

Solution 2 : avec un test simple

```
x, y = eval(input()),eval(input())
mini, maxi = x, y
if x > y:
    mini, maxi = y, x
print("minimum :",mini,", maximum :",maxi)
```

Structures conditionnelles

Exercices

Exercice : écrire un programme qui en fonction de la moyenne aux écrits du Bac indique si le lycéen en question est admis, recalé ou au rattrapage.

Solution :

```
note = eval(input())
if note < 8:
    print("recalé")
elif note < 10:
    print("rattrapage")
else:
    print("admis")
```

Notion d'itération

Dans certaines situations, on est amené à **exécuter plusieurs fois des actions identiques** ou du moins de même nature.

Une **Itération** est une séquence d'instructions destinée à être exécutée plusieurs fois.

Selon que le nombre de répétitions soit connu à l'écriture du programme ou pas, on utilisera une structure « **for** » ou une structure « **while** ».

Structures itératives

Structure “for”

- La structure « **for** » réalise un nombre d’itérations **fixe** et connu.
- Elle utilise une variable dont la valeur va parcourir une certaine **plage** au fil des itérations. C’est cette variable qui contrôlera le nombre d’itérations.
- Cette plage de valeurs va être construite grâce à la fonction « **range** ».

Syntaxe de la fonction “range” :

range (début, fin, pas)

- Cette fonction produit une plage de valeurs allant de ‘**début**’ (inclus) à ‘**fin**’ (non inclus) avec un certain ‘**pas**’.
- À noter que par défaut ‘début’ et ‘pas’ valent respectivement 0 et 1.

Structures itératives

Structure “for”

Exemples d'utilisation de la fonction “range” :

- **range(6)** produira la plage de valeurs **0, 1, 2, 3, 4, 5.**
- **range(3,6)** produira la plage de valeurs **3, 4, 5.**
- **range(3,10,2)** produira la plage de valeurs **3, 5, 7, 9.**
- **range(6,0,-1)** produira la plage de valeurs **6, 5, 4, 3, 2, 1.**

```
for i in range(0,n) :          # parcourt tous les entiers de 0 à n-1 inclus
for i in range(n,0,-1) :       # parcourt tous les entiers de n à 1 inclus dans le sens
                                # décroissant
for i in range(2,1000,3) :    # parcourt tous les entiers de 2 à 1000 de 3 en
                                # 3 # (2,5,8,...)
for e in li :                  # parcourt tous les éléments de la liste li
for cle,valeur in d.items () : # parcourt tous les éléments du dictionnaire d
```

Structure “for”

Syntaxe de la structure “for” :

```
for var in range(début,fin,pas) :  
    bloc d'instructions à répéter  
    pendant que var prend les valeurs de la  
    plage définie par range
```

Explication du fonctionnement :

- Au fil des itérations, la variable **var** prendra successivement toutes les valeurs de la plage créée par la fonction **range**.
- Le **nombre d'itérations** sera donc le **nombre d'éléments** de la **plage** créée par la fonction **range**.
- En pratique on se servira également très souvent des **valeurs** de la variable **var**.

Structures itératives

Structure “for”

Exemple : table de multiplication d'un entier saisi par l'utilisateur

```
n = eval(input())
for i in range(1,11):
    print(i,'*',n,'=',n*i)
```

Déroulement du programme :

- Admettons que l'utilisateur saisisse la valeur 9.
- À la première itération la variable **i** vaut 1,
il est donc affiché $1 * 9 = 9$.
- À la seconde itération la variable **i** vaut 2,
il est donc affiché $2 * 9 = 18$.
- Etc.



9	
1	* 9 = 9
2	* 9 = 18
3	* 9 = 27
4	* 9 = 36
5	* 9 = 45
6	* 9 = 54
7	* 9 = 63
8	* 9 = 72
9	* 9 = 81
10	* 9 = 90

Structures itératives

Structure “for”

Autres exemples : affichage des nombres pairs entre 20 et 0

```
for i in range(20,-1,-2):  
    print(i)
```



20
18
16
14
12
10
8
6
4
2
0

```
sum = 0  
  
for i in [1, 2, 3, 4]:  
    sum += i  
  
prod = 1  
  
for p in range(1, 10):  
    prod *= p
```

Remarques

Pour un grand nombre d'éléments, on préférera utiliser **xrange** plutôt que **range**.

Structures itératives

Structure “while”

- La structure « **while** » réalise un nombre d’itérations **non nécessairement connu** à l’écriture du programme.
- Ce nombre dépend d’une conditionnelle qui sera évaluée **avant** chaque itération.
- On la qualifie parfois d'**itération conditionnelle**.

Syntaxe de la structure “while” :

while conditionnelle:

bloc d’instructions à exécuter tant que la conditionnelle est vraie

Explication du fonctionnement :

- On exécute donc le bloc d’instructions **tant que** la conditionnelle est vraie.
- Dans ce bloc, il est donc **nécessaire de modifier la valeur de la conditionnelle** pour ne pas risquer d’avoir une infinité d’itérations.

Structures itératives

Structure “while”

Exemple : calcul de la partie entière d'un réel positif

```
x = eval(input())
n = 0
while n+1 <= x:
    n += 1
print("la partie entière de",x,"est",n)
```



2.57
la partie entière de 2.57 est 2

Déroulement du programme :

- Admettons que l'utilisateur saisisse la valeur **2.57**
- Avant les itérations, la variable **n** est initialisée à **0**.
- La conditionnelle est alors vraie donc on réalise la première itération, et **n** vaut maintenant **1**.
- La conditionnelle est encore vraie donc on réalise la seconde itération, et **n** vaut maintenant **2**.
- Cette fois-ci la conditionnelle est fausse et l'on stoppe les itérations.

Imbrication de structures itératives

- L'**imbrication** de structures itératives est non seulement possible mais souvent fort utile.
- À noter que l'on peut également imbriquer structures itératives et structures conditionnelles.

Exemple : affichage de toutes les tables de multiplications

```
for i in range(1,11):
    print("table de",i)
    for j in range(1,11):
        print("\t",j,"*",i,"=",j*i)
```

La variable “i” vaut d'abord 1, et alors “j” prend toutes les valeurs de 1 à 10, puis “i” vaut 2 et “j” prend de nouveau toutes les valeurs de 1 à 10, etc.



table de 1
1 * 1 = 1
2 * 1 = 2
3 * 1 = 3
4 * 1 = 4
5 * 1 = 5
6 * 1 = 6
7 * 1 = 7
8 * 1 = 8
9 * 1 = 9
10 * 1 = 10
table de 2
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6

Structures itératives

Sorties de boucles

Deux possibilités pour “sortir” d'une boucle :

- “**break**” : cette commande permet la **sortie** de la structure itérative “**for**” ou “**while**” qui la contient.
- “**continue**” : cette commande permet de **passer directement à l'itération suivante** dans la structure “**for**” ou “**while**” qui la contient.

Exemple de “**break**” : où l'on félicite l'utilisateur s'il saisit un ‘x’ en au plus 10 tentatives.

```
for i in range(9):
    if input() == 'x':
        print("quel talent")
        break
```



666
mick
x
quel talent

Structures itératives

Sorties de boucles

Exemple de “continue” : affichage de tous les nombres de 10 à 20 sauf le nombre 13.

```
for i in range(10,21):
    if i == 13:
        continue
    print(i)
```



10
11
12
14
15
16
17
18
19
20

Structures itératives

Clause “else” dans une boucle

- Les structures “**for**” et “**while**” peuvent contenir une clause “**else**”.
- À noter que celle-ci ne sera exécutée que si la sortie de la boucle s’effectue de façon normale, c'est-à-dire sans “**break**”.

Syntaxe de la structure “**for**” avec clause “**else**” :

```
for var in range(début,fin,pas) :  
    bloc d'instructions à répéter  
    pendant que var prend les valeurs de la  
    plage définie par range  
else :  
    bloc d'instructions à exécuter (une fois)  
    quand var a pris toutes les valeurs de la  
    plage définie par range
```

Structures itératives

Clause “else” dans une boucle

Syntaxe de la structure “while” avec clause “else” :

while conditionnelle:

bloc d'instructions à exécuter tant que la conditionnelle est vraie

else:

bloc d'instruction à exécuter (une fois) si la conditionnelle est fausse

Exemple : savoir si un nombre **n** saisi par l'utilisateur est un nombre premier ou non

```
n = eval(input())
for i in range(2,n):
    if n%i == 0:
        premier = False
        break
    else:
        premier = True
```

Exercices

Exercice : écrire un programme calculant la somme des premiers entiers naturels jusqu'à un entier positif saisi par l'utilisateur.

Solution :

```
n = -1
while n < 0:
    n = eval(input())
somme = 0
for i in range(n+1):
    somme += i
print("la somme des",n,"premiers entiers
      vaut",somme)
```

Exercices

Exercice : écrire un programme calculant la somme d'une suite d'entiers saisie par l'utilisateur se terminant par 0 (exemple 5, 4, 9, 0 renverra 18).

Solution :

```
somme,n = 0,1
while n != 0:
    n = eval(input())
    somme += n
print("la somme des entiers saisis vaut",somme)
```

Les structures de contrôle

Zip et map

- **zip** : permet de parcourir plusieurs séquences en parallèle

```
L1 = [1, 2, 3]
L2 = [4, 5, 6]
for (x, y) in zip(L1, L2):
    print (x, y, '---', x + y)
```

1	4	--	5
2	5	--	7
3	6	--	9

- **map** : applique une méthode sur une ou plusieurs séquences

```
def double(x):
    return x*2
```

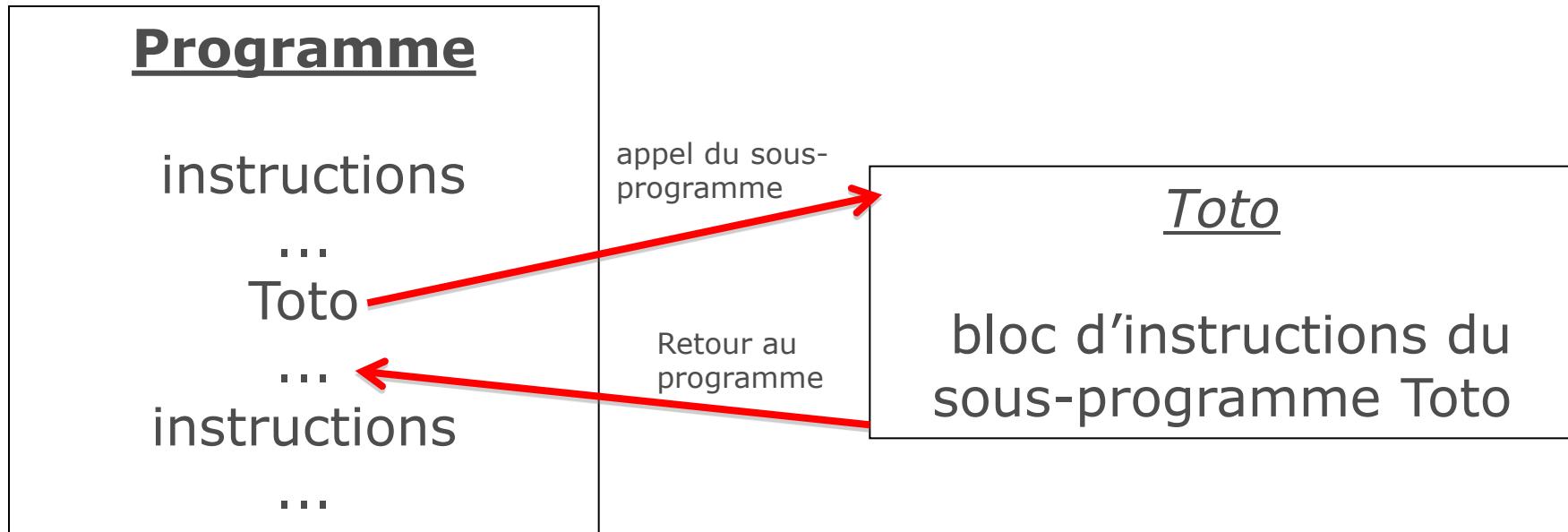
```
L=[1,2,3,4,5,6,7,8,9]
map(double,L)
[2,4,6,8,10,12,14,16,18]
```

Principe

- Un sous-programme est un **bloc d'instructions réalisant une certaine tâche.**
- Il possède un **nom** et est exécuté lorsqu'on l'appelle.
- Un script bien structuré contiendra **un programme dit “principal”**, et **plusieurs sous-programmes** dédiés à des fonctionnalités spécifiques.

Sous-programmes

Principe

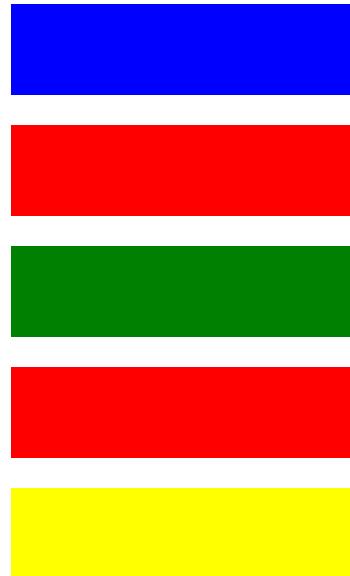


- Quand un programme dit « principal » fait appel à un sous-programme, il suspend son propre déroulement, exécute le sous-programme en question, et reprend ensuite son fonctionnement.

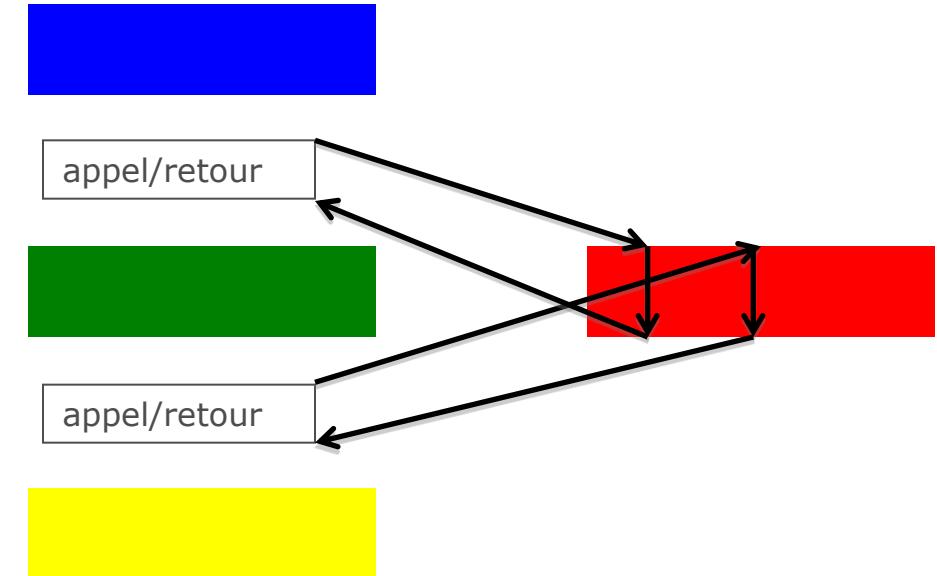
Sous-programmes

Trois grands avantages

1. Éviter la duplication de code.



Ici le bloc de code en rouge est dupliqué



Ici on crée un sous-programme correspondant à ce bloc, et on l'appelle quand on en a besoin.

Trois grands avantages.

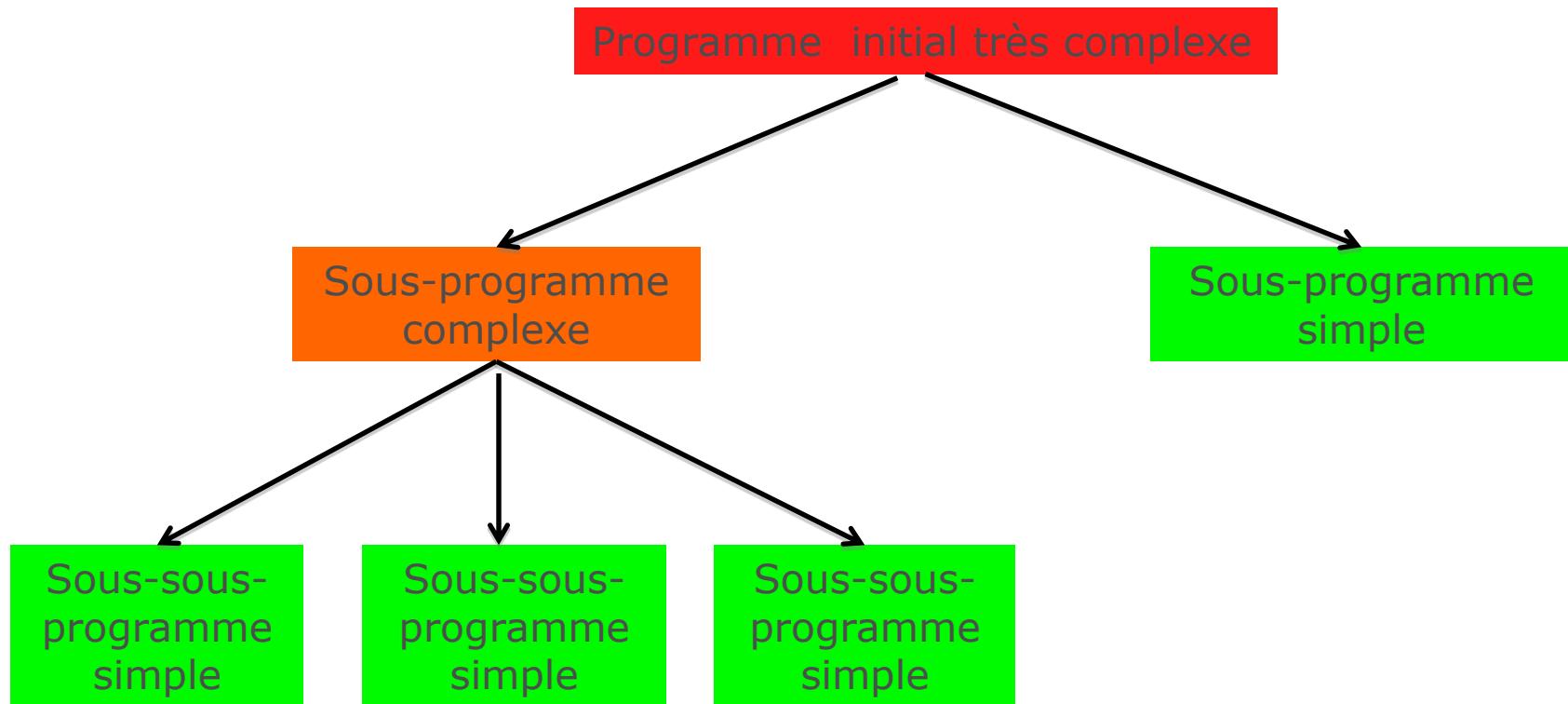
2. Favoriser la réutilisation.

- Un sous-programme écrit pour résoudre un problème donné pourra servir de nouveau dans un autre contexte.
- On pourra ainsi créer des « **librairies** » de sous-programmes.

Sous-programmes

Trois grands avantages.

3. Améliorer la conception.

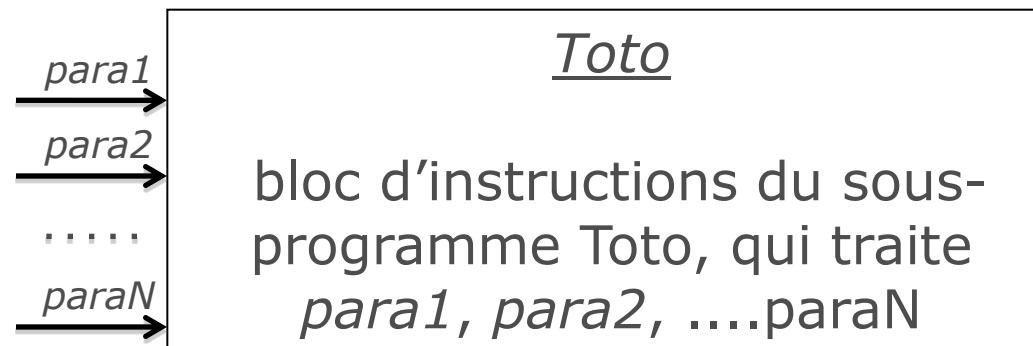


Notion de paramètre

- Un sous-programme sert donc à effectuer un **traitement générique**.
- Ce traitement porte sur des **données**, dont la valeur pourra ainsi changer d'un appel du sous-programme à un autre.
- Ce que l'on appelle "**paramètres**" ce sont justement ces données transmises au sous-programme par le programme principal.

Notion de paramètre

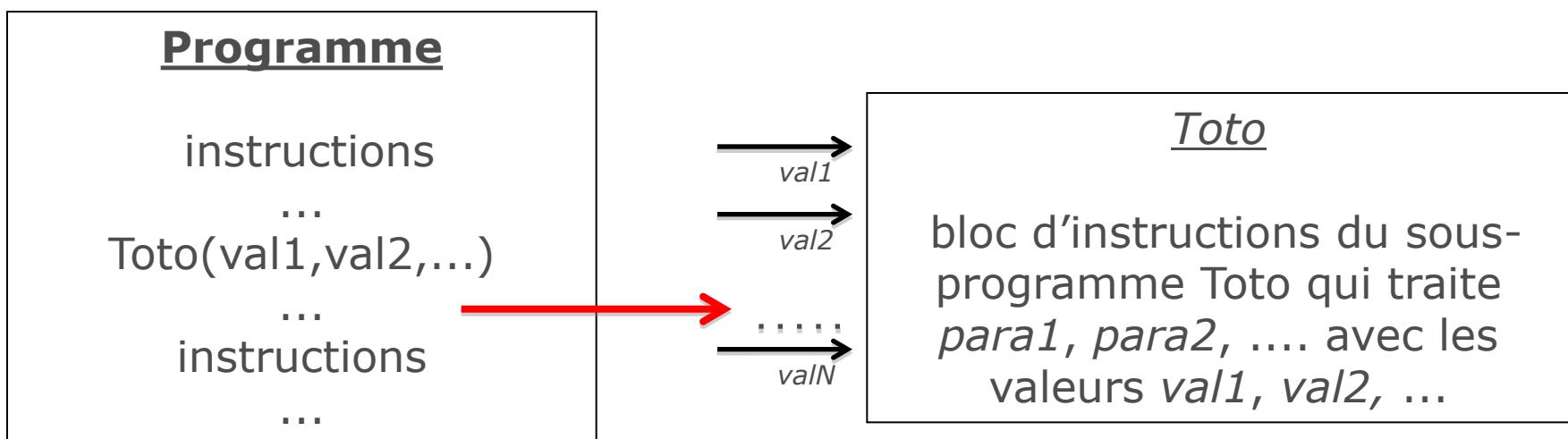
- Lors de l'**implémentation** d'un sous-programme, on va donc préciser **la liste de tous les paramètres** qu'il va utiliser.



Sous-programmes

Notion de paramètre

- Lors de l'**utilisation** d'un sous-programme, on va alors préciser la **valeur** de chacun des paramètres qu'il possède.

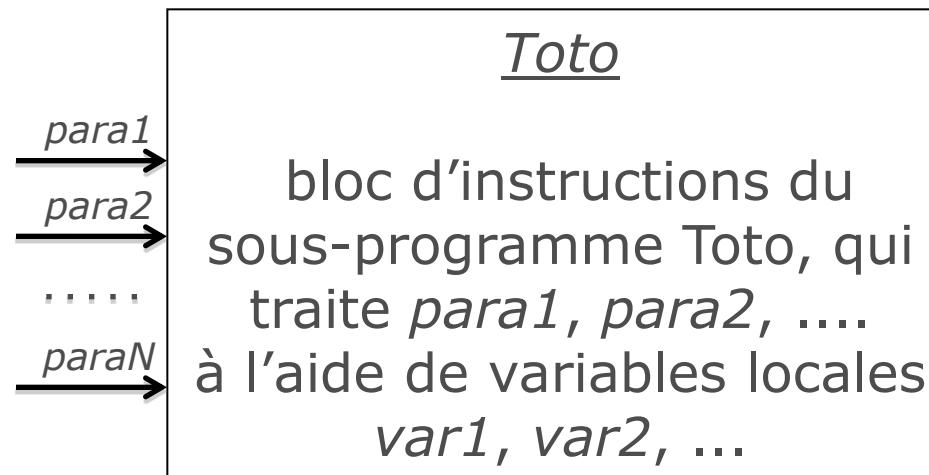


Variables locales versus globales

- Pour fonctionner un sous-programme peut également avoir besoin d'utiliser des variables qui lui sont propres. On parle alors de "**variables locales**".
- Ce sont par exemple des résultats de calculs intermédiaires, des compteurs de tours dans une structure itérative, etc.
- Ces variables ne sont accessibles qu'**au sein** du sous-programme qui les définit et utilise.

Variables locales versus globales

- Un sous programme reçoit donc des données à traiter, les **paramètres**, et pour ce faire peut avoir besoin de **variables locales** :



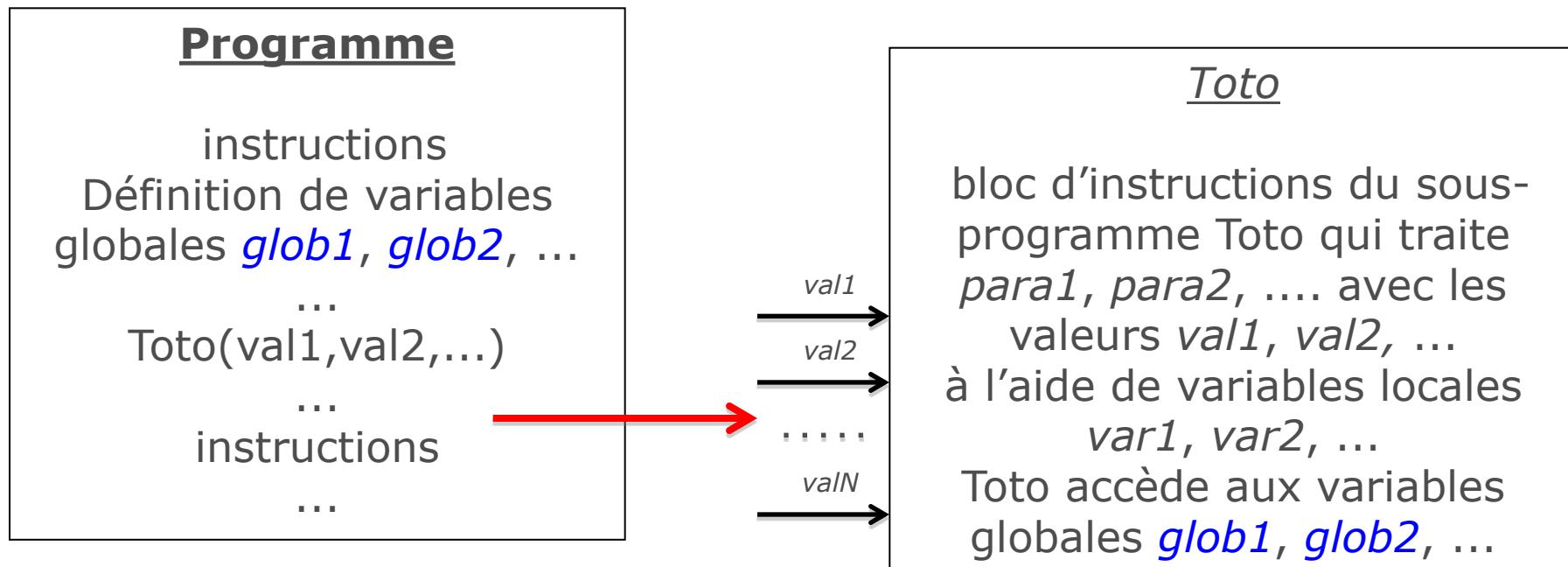
Variables locales versus globales

- On a vu qu'un sous-programme pouvait utiliser des données transmises par le programme principal sous forme de **paramètres**.
- Il peut également manipuler directement des variables définies par le programme principal. On parle alors de "**variables globales**".
- Il s'agit d'une possibilité mais également d'une **mauvaise pratique** car cela limite énormément la **réutilisabilité** des codes d'un projet à un autre.

Sous-programmes

Variables locales versus globales

- Un sous-programme peut accéder aux **variables globales** définies dans le **programme principal** :



Sous-programmes

Deux types de sous-programmes

En algorithmique on distingue **deux types de sous-programmes** :

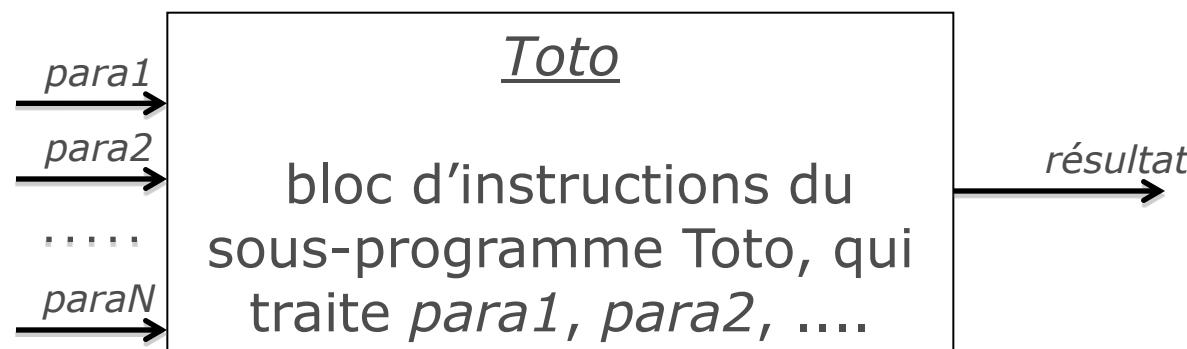
- Les **procédures**, qui modifient l'état du programme sans retourner de résultat.
- Les **fonctions**, qui elles ont pour but de retourner un résultat.

Sous-programmes

Deux types de sous-programmes

Fonctions :

- Il s'agit donc d'un sous-programme qui va **retourner un résultat** au programme principal.



Sous-programmes

Deux types de sous-programmes

Remarque :

- Tous les langages de programmation ne distinguent pas nommément ces deux types de sous-programmes.

- En Python, en C et en C++, on ne manipule ainsi a priori que des fonctions. Bien qu'en pratique la distinction se fasse.

- En Pascal par contre, les deux types sont clairement séparés.

Les sous-programmes en Python

Les Fonctions

Le langage **Python** permet de réaliser ce que l'on nomme des **fonctions**. Il s'agit d'un bloc d'instructions qu'on écrit une seule fois en leur attribuant un nom et qui dépend ou non d'un certain nombre de paramètres.

La **fonction**, peut être utilisée en tout point d'un programme en utilisant juste un appel avec le nom du sous programme (ou de la fonction) avec les paramètres voulus.

Syntaxe générale d'une fonction :

```
def maFonction(para1,para2,...,paraN):  
    bloc d'instructions de la fonction  
    return valeur
```

Exemple : calcul du cube d'un nombre

```
def cube(x):  
    return x*x*x
```

Les Fonctions

Remarques importantes :

- Une fonction peut **retourner plusieurs valeurs**, il suffit de séparer celles-ci par des virgules.
- Une fonction peut contenir plusieurs fois la commande "**return**", mais elle cesse son fonctionnement dès qu'elle en rencontre une.

Exemple : calcul du minimum et du maximum de deux nombres

```
def calculMiniMaxi(x,y):  
    if x < y:  
        return x,y  
    else:  
        return y,x
```

Les Fonctions

Utilisation d'une fonction :

- On appelle la **fonction** par son **nom**, en lui passant autant de **paramètres** qu'elle en possède.
- Pour ne pas “**perdre**” la(les) valeur(s) retournée(s), on les incorporera par exemple dans une opération d'affichage, d'affectation, etc.

Exemples d'utilisations de fonctions :

```
print(cube(5))  
a,b = 5,-2  
min,max = calculMiniMaxi(a,cube(b))  
print("Minimum : ",min," , Maximum : ",max)
```



125
Minimum : -8 , Maximum : 5

Les Fonctions

Duck Typing

Principe du “Duck Typing” :

- En Python on ne précise pas les types **attendus** des paramètres des sous-programmes.
- Cela implique que l'on peut utiliser un sous-programme avec des paramètres de **n'importe quel type**, à la condition que les **opérations** du sous-programme soient **compatibles** avec les **types** des paramètres.

Exemple de “Duck Typing” :

```
def addition(x,y):  
    return x + y  
print(addition(666,1))  
print(addition("Brown ", 'Sugar'))
```

667
Brown Sugar

Les sous-programmes en Python

Paramètres par défaut

- Les paramètres d'un sous-programme peuvent comporter des **valeurs par défaut**.
- **Deux cas possibles :**
 - Lors de l'appel on ne précise pas de valeurs pour les paramètres en question et le sous-programme utilise celles par défaut.
 - Si on précise des valeurs ce sont celles-ci qui sont utilisées.

Exemple :

```
def rectangle(x=3,y=1):  
    print("périmètre : ", 2*(x+y), "aire : ", x*y)  
rectangle()  
rectangle(2)  
rectangle(7,5)
```



```
périmètre : 8 , aire : 3  
périmètre : 6 , aire : 2  
périmètre : 24 , aire : 35
```

Les sous-programmes en Python

Paramètres par défaut

Autre exemple :

```
def rectangle(x,y=1):
    print("périmètre :" , 2*(x+y) , "aire :" , x*y)

rectangle(2)
rectangle(7,5)
```



```
périmètre : 6 , aire : 2
périmètre : 24 , aire : 35
```

Les sous-programmes en Python

Paramètres immuables

- Les paramètres de type “**int**”, “**bool**”, “**float**”, “**complex**” et “**str**” sont **immuables**.
- Cela signifie que si l'on passe une variable de l'un de ces types comme paramètre à une fonction, celle-ci ne **pourra pas en modifier sa valeur**.

Exemple : tentative pour doubler un nombre

```
def doubler(x) :  
    x *= 2  
  
    a = 3  
    print("valeur de a avant :",a)  
    doubler(a)  
    print("valeur de a après :",a)
```



```
valeur de a avant : 3  
valeur de a après : 3
```

Les sous-programmes en Python

Paramètres immuables

Exemple (suite) : contournement du problème précédent en utilisant une fonction

```
def doubler(x):  
    return 2*x  
  
a = 3  
print("valeur de a avant :", a)  
a = doubler(a)  
print("valeur de a après :", a)
```



```
valeur de a avant : 3  
valeur de a après : 6
```

Variables locales et globales

Variables locales :

- Ce sont donc des variables définies à l'**intérieur** d'un sous programme et qui ne sont accessibles qu'au sein de celui-ci.
- Elles servent essentiellement au bon fonctionnement du sous-programme.

Exemple : fonction calculant la somme des **n** premiers entiers, avec deux variables locales “utilitaires”

```
def sommeEntiers(n):  
    somme = 0  
    for i in range(n+1):  
        somme += i  
    return somme
```

Les sous-programmes en Python

Variables locales et globales

Exemple (suite) : tentative d'utilisation de la variable locale somme en dehors de la fonction

```
def sommeEntiers(n):
    somme = 0
    for i in range(n+1):
        somme += i
    return somme

print(somme)
```

```
Traceback (most recent call last):
  File "/Users/laurentgodefroy/PycharmProjects/Test/etceluila.py", line 170, in <module>
    print(somme)
NameError: name 'somme' is not defined
```

Variables locales et globales

Variables globales :

■ Ce sont des variables définies en dehors de tout sous-programme. Elles sont “globales” au sens où elles sont visibles et utilisables dans tous les sous-programmes du module courant.

Ordre de recherche des variables dans un sous-programme

La rencontre d'un nom de variable dans un sous-programme déclenche une recherche **LGI** (Locale Globale Interne) :

1. Recherche d'une variable **Locale** correspondant à ce nom.
2. Recherche d'une variable **Globale** correspondant à ce nom.
3. Recherche d'un nom **Interne** au langage.

Les sous-programmes en Python

Variables locales et globales

Exemple 1 :

```
def exemple1():
    print(i)

i = 666
exemple1()
```



666

Exemple 2 :

```
def exemple2():
    i = 111
    print(i)

i = 666
exemple2()
print(i)
```



111
666

Les sous-programmes en Python

Variables locales et globales

Exemple 3 :

```
def exemple3():
    print(i)
    i = 111
    print(i)

i = 666
exemple3()
```

```
Traceback (most recent call last):
  File "C:/Users/HP/Desktop/p3.py", line 7, in <module>
    exemple3()
  File "C:/Users/HP/Desktop/p3.py", line 2, in exemple3
    print(i)
UnboundLocalError: local variable 'i' referenced before assignment
```

Variables locales et globales

Modification de la valeur de variables globales :

- Les variables globales ne sont a priori pas modifiables par des sous-programmes.
- Pour que cela soit possible, il faut le signaler explicitement dans le sous programme à l'aide de l'instruction “**global**”.

Syntaxe :

global var

Variables locales et globales

Exemple 4 :

```
def exemple4():
    global i
    print(i)
    i = 111
    print(i)

i = 666
exemple4()
print(i)
```



666
111
111

Les sous-programmes en Python

Exercice

Ecrire une fonction (procédure) prenant en paramètre une durée en secondes et qui affiche la conversion de cette durée en heures, minutes, secondes.

Solution :

```
def conversion1(n):
    h = n // 3600
    m = (n - 3600*h) // 60
    s = n % 60
    print(h, "heures, ", m, "minutes, ", s, "secondes")
```

Les sous-programmes en Python

Récursivité

Définition : un sous-programme (fonction) est dit **récursif** s'il s'appelle lui même.

Idée : pour effectuer une tâche ou un calcul, on se ramène à la réalisation d'une tâche similaire mais de **complexité moindre**. On recommence jusqu'à obtenir une tâche élémentaire.

Exemple classique : calcul de **$n! = 1 \times 2 \times 3 \times \dots \times n$** .

On montre facilement la relation de récurrence **$n! = n \times (n-1)!$**

Si on sait calculer **$(n-1)!$** , on connaîtra donc la valeur de **$n!$**

Mais **$(n-1)! = (n-1) \times (n-2)!$**

On est donc ramené au calcul de **$(n-2)!$**

Et ainsi de suite jusqu'à **$1!$** dont on connaît la valeur : 1

Les sous-programmes en Python

Récursivité

Exemple : calcul récursif de $n!$

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

Déroulement du programme : si par exemple $n = 4$

$\text{facto}(4) = 4 \times \text{facto}(3)$

$\text{facto}(3) = 3 \times \text{facto}(2)$

$\text{facto}(2) = 2 \times \text{facto}(1)$

$\text{facto}(1) = 1$

$\text{facto}(4) = 4 \times 6 = 24$

$\text{facto}(3) = 3 \times 2 = 6$

$\text{facto}(2) = 2 \times 1 = 2$

$\text{facto}(1) = 1$

Les sous-programmes en Python

Récursivité

Remarque importante :



Il est indispensable de prévoir une condition d'arrêt à la récursions sinon le programme ne se termine jamais.

Exemple à ne pas suivre :

```
def factorielleRecursiveBadJob(n):
    return n*factorielleRecursiveBadJob(n-1)
```

Les sous-programmes en Python

Récursivité versus itération

- On peut toujours transformer un algorithme récursif en un algorithme itératif et inversement.
- L'algorithme itératif sera **plus rapide** une fois implémenté dans un langage de programmation mais souvent **plus complexe à écrire**.

Exemple : version itérative du calcul de $n!$

```
def factorielleIterative(n) :  
    resultat = 1  
    for i in range(2,n+1) :  
        resultat *= i  
    return resultat
```

Les sous-programmes en Python

Récursivité versus itération

Intérêts de la récursivité :

- Technique de programmation très **élégante** et **lisible** (elle évite souvent le recours à de nombreuses structures itératives).
- Elle est très utile pour concevoir des algorithmes sur des structures complexes.

Inconvénient majeur de la récursivité :

- Une fois implémentée dans un langage de programmation, cette technique est très « gourmande » en mémoire.
- Elle peut même provoquer des débordements de capacité.

Les sous-programmes en Python

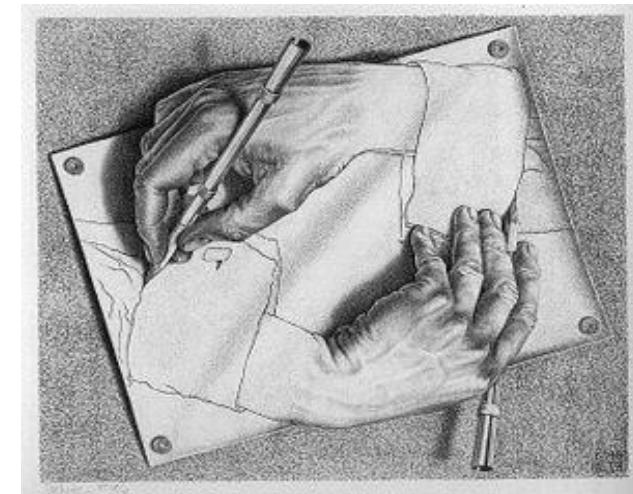
Récursivité croisée

Récursivité croisée : c'est un cas bien particulier de récursivité où une fonction appelle une autre fonction qui elle-même appelle la première.

Exemple : test de la parité d'un nombre

```
def pair(n):
    if n == 0:
        return True
    else:
        return impair(n-1)

def impair(n):
    if n == 0:
        return False
    else:
        return pair(n-1)
```



Les sous-programmes en Python

Récursivité multiple

Récursivité multiple : sous-programme récursif réalisant plusieurs appels à lui même.

Exemple : calcul des coefficients binomiaux.

$$\text{def } \binom{n}{k} = \begin{cases} 1 & \text{si } k=0 \text{ ou si } k=n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sinon} \end{cases}$$

Exemple : calcul des coefficients binomiaux

```
def coeffs(n,k):
    if k == 0 or k == n:
        return 1
    else:
        return coeffs(n-1,k-1) + coeffs(n-1,k)
```

Les sous-programmes en Python

Récursivité imbriquée

Récursivité imbriquée : sous programme récursif dont l'appel à lui même contient un autre appel à lui même.

Exemple : fonction 91 de McCarthy définie sur \mathbb{Z} par

$$f(n) = \begin{cases} n-10 & \text{si } n > 100 \\ f(f(n+11)) & \text{si } n \leq 100 \end{cases}$$

Exemple : fonction 91 de McCarthy

```
def f91(n):
    if n > 100:
        return n-10
    else:
        return f91(f91(n+11))
```

Les sous-programmes en Python

Exercice

Ecrire de deux façons (itérative et récursive) une fonction ayant pour paramètres un réel x et un entier n , et retournant la valeur de x puissance n .

Solution : version itérative

```
def puissanceIterative(x,n):
    resultat = 1
    for i in range(1,n+1):
        resultat *= x
    return resultat
```

Solution : version récursive

```
def puissanceRecurcive(x,n):
    if n == 0:
        return 1
    else:
        return x*puissanceRecurcive(x,n-1)
```

Exercice Les sous-programmes en Python

Ecrire une fonction récursive qui calcule le produit de deux entiers **A** et **B**

```
#----- Définition de la fonction-----
def prodRecu(A,B):
    if B==0:
        return 0
    return (A+prodRecu(A,B-1))

#----- appel de la fonction -----
A=int(input("entrez une valeur:"))
B=int(input("entrez une autre valeur:"))
print("le produit
de",A,"et",B,"est:",prodRecu(A,B))
```

Les sous-programmes en Python

Exercice

Ecrire une fonction récursive qui calcule le quotient de la division de deux entiers **A** et **B**

```
#----- Définition de la fonction-----
def quotRecu(A,B):
    if B>A:
        return 0
    return (1+quotRecu(A-B,B))

#----- appel de la fonction -----
A=int(input("entrez une valeur:"))
B=int(input("entrez une autre valeur:"))
print("le quotient de la division de deux
entiers",A,"et",B,"est:", quotRecu(A,B))
```

Les sous-programmes en Python

La fonction «lambda»

Définition :

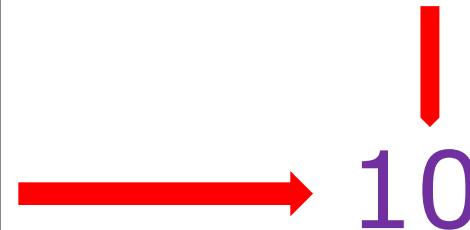
lambda argument1,... argumentN : expression utilisant les arguments

Exemples :

```
f = lambda x, i : x**i  
f(2, 4) = 32
```

```
def mx(x,y)  
if x>y:  
    return x  
else:  
    return y  
print(mx(10,7))
```

```
mx=lambda x,y:x if x>y else y  
print(mx(10,7))
```



Notion de séquence

Motivation :

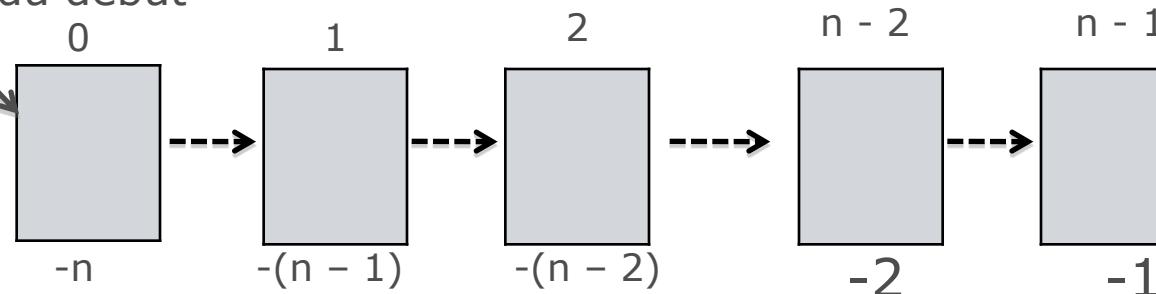
- Réunir au sein d'une même variable **plusieurs valeurs** différentes.
- L'objectif étant d'optimiser certaines opérations comme la recherche d'un élément, le tri de ces valeurs, le calcul de leur maximum, etc.

Définition :

- Suite d'éléments accessibles par leur **position**.
- Chaque élément, à part le premier, a un **prédecesseur** et, à part le dernier, a un **successeur**.

Une séquence de n éléments :

Indexation des éléments
à partir du début



Indexation des éléments à partir de la fin

Notion de séquence

Accès à un élément :

- L'accès à un élément d'une séquence se fait en utilisant la position de cet élément et des crochets [].

```
maSéquence[maPositionVoulue]
```

Les trois principaux types de séquences :

- ❖ Les **listes** dont les éléments sont quelconques et modifiables.
- ❖ Les **t-uples** dont les éléments sont quelconques et non modifiables.
- ❖ Les **chaînes de caractères** dont les éléments sont des caractères et ne sont pas modifiables.

Structures de données et leurs utilisations

Les listes

Une liste est une structure de données qui contient une série de valeurs. Python autorise la construction de liste contenant des valeurs de **type différent**.

Syntaxes de la déclaration d'une liste :

```
maListeVide = []
```

```
maListeAvecUnSeulElement = [élément]
```

```
maListe = [élément1, élément2, ..., élémentN]
```

Exemples

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> mixte = ['girafe', 5, 'souris', 0.15]
>>> animaux
['girafe', 'tigre', 'singe', 'souris']
>>> mixte
['girafe', 5, 'souris', 0.15]
```

Structures de données et leurs utilisations

Les listes

Création:

```
x = []                      # crée une liste vide
x = list()                   # crée une liste vide
x = [4,5]                    # création d'une liste composée de
                            # deux entiers
x = ["un",1,"deux",2]        # création d'une liste composée de
                            # 2 chaînes de caractères et de
                            # deux entiers, l'ordre d'écriture
                            # est important
```

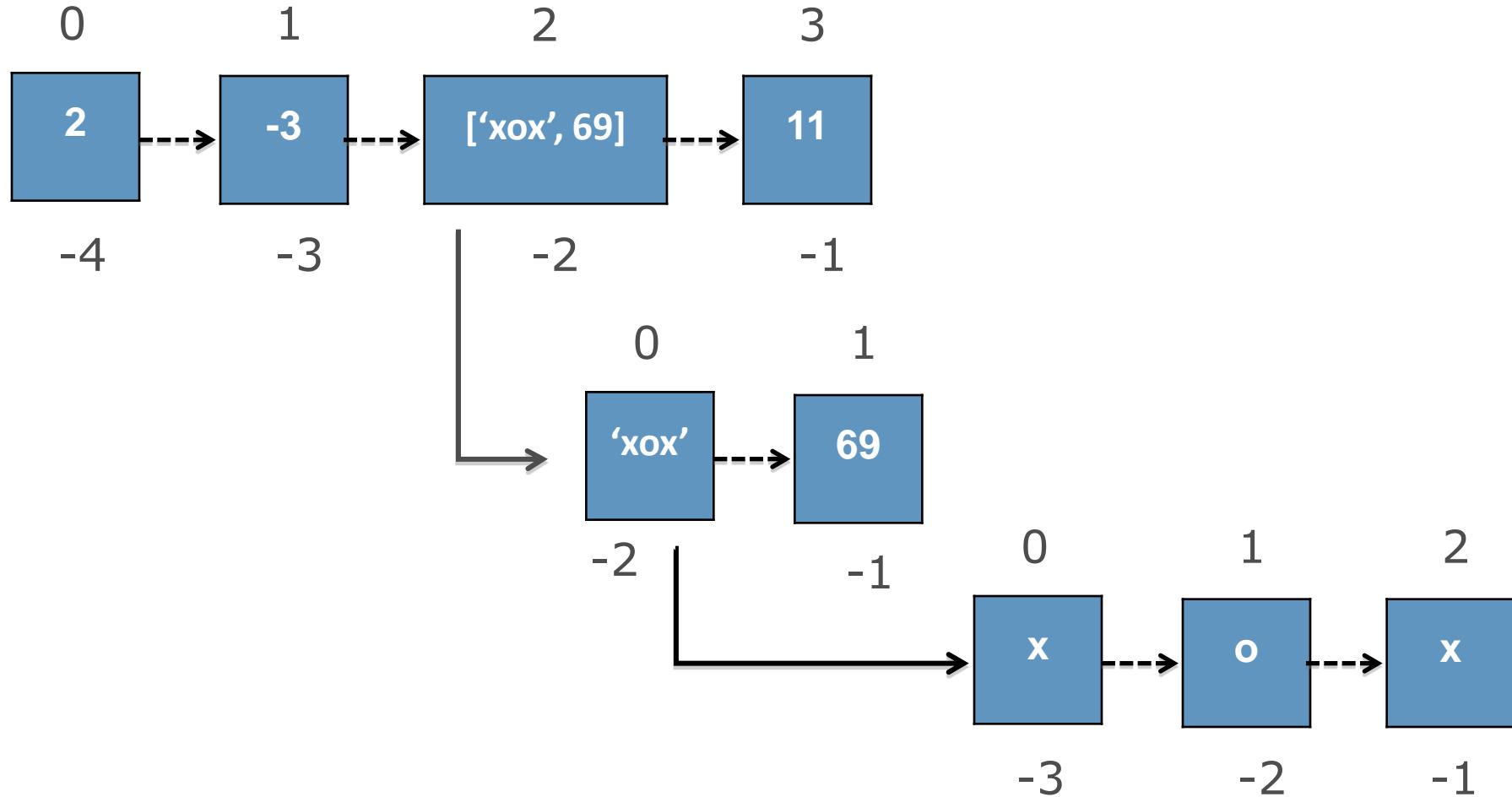
Exemple:

```
>>> maListe = [2, -3, ['xox', 69], 11]
>>> print(maListe)      → [2, -3, ['xox', 69], 11]
>>> print(maListe[-1])   →    11
>>> maListe[1] = 666
>>> print(maListe[1])      →    666
>>> print(maListe[2][0][1]) →    o
```

Structures de données et leurs utilisations

Les listes

Un petit dessin de la liste précédente pour fixer les idées :



Les listes

Avec la boucle for on peut aussi créer des listes :

```
x = [i for i in range(0,5)]  
print (x)  
[0,1,2,3,4]
```

```
y = [ i for i in x if i % 2 == 0]  
Print(y)  
[0,2,4]
```

```
z = [ i+j for i in x for j in x]  
print (z)  
[0, 1, 2, 3, 4, 1, 2, 3, 4, 5, 2, 3, 4, 5, 6, 3,  
 4, 5, 6, 7, 4, 5, 6, 7, 8]
```

Structures de données et leurs utilisations

Les listes

Les opérations sur les listes:

Opération	Résultat
x in s	Teste si x appartient à s
x not in s	Teste si x n'appartient pas à s
s + t	Concaténation de s et t
s * n ou n * s	Concaténation de n copies de s
len(s)	Nombre d'éléments de s
min(s)	Plus petit élément de s
max(s)	Plus grand élément de s
s.count(x)	Nombre d'occurrences de x dans s
s.index(x)	Indice de x dans s .
s[i]	i -ème élément de s
s[i:j]	Sous-séquence de s constituée des éléments entre le i -ème (inclus) et le j -ème (exclus)
s[i:j:k]	Sous-séquence de s constituée des éléments entre le i -ème (inclus) et le j -ème (exclus) pris avec un pas de k

Les listes

Exemple 1:

```
>>> t = [7, -3, 2, 11, 666, -1975]
>>> t[3:]
[11, 666, -1975]
>>> t[1:4]
[-3, 2, 11]
>>> t[1::2]
[-3, 11, -1975]
>>> t[23]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: tuple index out of range
```

Exemple 2:

```
>>> t = [7, -3, 2]
>>> 11 not in t
True
>>> tt = 2*t
>>> print(tt)
[7, -3, 2, 7, -3, 2]
>>> tt.count(-3)
```

Structures de données et leurs utilisations

Les listes

Méthodes permettant de modifier une liste :

Opération	Résultat
<code>s[i] = x</code>	Remplacement de l'élément <code>s[i]</code> par <code>x</code>
<code>s[i:j] = t</code>	Remplacement des éléments de <code>s[i:j]</code> par ceux de la séquence <code>t</code>
<code>del(s[i:j])</code>	Suppression des éléments de <code>s[i:j]</code>
<code>s[i:j:k] = t</code>	Remplacement des éléments de <code>s[i:j:k]</code> par ceux de la séquence <code>t</code>
<code>del(s[i:j:k])</code>	Suppression des éléments de <code>s[i:j:k]</code>

Exemple:

```
>>> maListe = [1, 2, 3, 4, 5]
>>> maListe[2:4] = (6, 'x', 7)
>>> print(maListe)
[1, 2, 6, 'x', 7, 5]
>>> maListe[1:6:2] = 'sup'
>>> print(maListe)
[1, 's', 6, 'u', 7, 'p']
```

Structures de données et leurs utilisations

Les listes

Méthodes permettant de modifier une liste :

Opération	Résultat
<code>list(s)</code>	Transforme une séquence s en une liste
<code>s.append(x)</code>	Ajoute l'élément x à la fin de s
<code>s.extend(t)</code>	Étend s avec la séquence t
<code>s.insert(i,x)</code>	Insère l'élément x à la position i
<code>s.clear()</code>	Supprime tous les éléments de s
<code>s.remove(x)</code>	Retire l'élément x de s
<code>s.pop(i)</code>	Renvoie l'élément d'indice i et le supprime
<code>s.reverse()</code>	Inverse l'ordre des éléments de s
<code>s.sort()</code>	Trie les éléments de s par ordre croissant

Structures de données et leurs utilisations

Les listes

Exemple 1:

```
>>> maListe = [1, 3, 5]
>>> maListe.append(7)
>>> print(maListe)
[1, 3, 5, 7]
>>> maListe.extend((8, 11))
>>> print(maListe)
[1, 3, 5, 7, 8, 11]
>>> maListe.remove(8)
>>> print(maListe)
[1, 3, 5, 7, 11]
>>> maListe.insert(4,9)
>>> print(maListe)
[1, 3, 5, 7, 9, 11]
```

Exemple 2:

```
>>> maListe = list(range(1,10,2))
>>> maListe
[1, 3, 5, 7, 9]
>>> taListe = list('The Strypes')
>>> taListe
['T', 'h', 'e', ' ', 'S', 't', 'r', 'y', 'p', 'e', 's']
```

Structures de données et leurs utilisations

Les listes

Méthodes permettant de modifier une liste :

Ajouter une valeur à une liste python

```
>>> liste = [1,2,3]
>>> liste.append("ok")
[1, 2, 3, 'ok']
```

Compter le nombre d'occurrences d'une valeur

```
>>> liste = ["a","a","a","b","c","c"]
>>> liste.count("a")
3
```

Transformer une string en liste

```
>>> ma_chaine = "Anouar:DARIF:SupMTI"
>>> ma_chaine.split(":")
['Anouar', 'DARIF', 'SUPMTI']
```

Transformer une liste en string

```
>>> liste = "Anouar", "DARIF", "SupMTI"
>>> ":".join(liste)
'Anouar:DARIF:SupMTI '
```

Les listes multidimensionnelles

Exercice : écrire une fonction prenant une liste de nombre entiers en paramètre, et qui retourne le nombre d'entiers pairs de cette séquence.

Solution :

```
def nbPairs(s):
    n = 0
    for x in s:
        if x%2 == 0:
            n += 1
    return n

monListe = (-4, 5, 2, 4, 1)
print(nbPairs(monListe))
```



3

Le délicat problèmes des copies

Phénomène d'alias :

- La tentative de copie d'une liste avec la syntaxe naturelle ne crée qu'un **alias**.

```
maListe2 = maListe1
```

- Les deux noms pointeront vers le **même emplacement mémoire**. Il n'y aura donc en réalité qu'une seule liste existante.

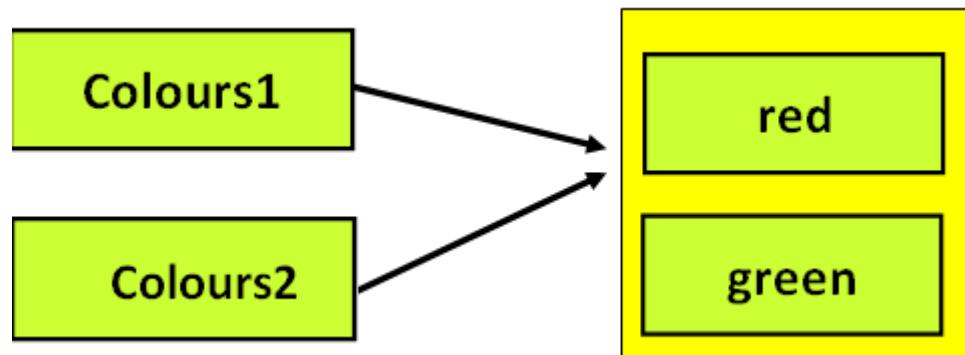
Exemple du problème soulevé :

```
>>> maListe1 = [1, 2, 3, 4]
>>> maListe2 = maListe1
>>> maListe2[3] = 666
>>> print(maListe1)
[1, 2, 3, 666]
>>> maListe1[0] = 'Python'
>>> print(maListe2)
['Python', 2, 3, 666]
```

Le délicat problèmes des copies

Autre exemple du problème soulevé :

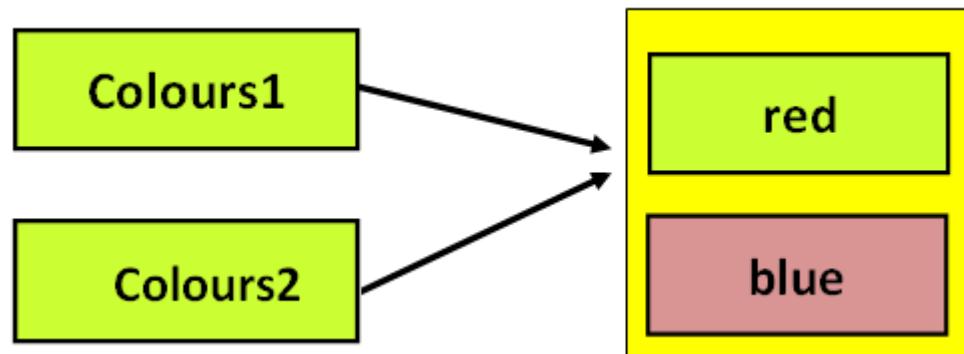
```
>>> colours1 = ["red", "green"]  
>>> colours2 = colours1  
>>> print colours2  
['red', 'green']
```



Le délicat problèmes des copies

Autre exemple du problème soulevé :

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2[1] = "blue"
>>> colours1
['red', 'blue']
```



Structures de données et leurs utilisations

Le délicat problèmes des copies

Plusieurs solutions pour réaliser une copie superficielle

Elle est utilisée lorsqu'on a besoin de manipuler le premier niveau ou que l'on veux que les niveaux inférieurs restent synchrones.

```
maListe2 = maListe1.copy()
```

```
import copy
maListe2 = copy.copy(maListe1)
```

```
maListe2 = list(maListe1)
```

```
maListe2 = []
maListe2.extend(maListe1)
```

```
maListe2 = maListe1[:]
```

Le délicat problèmes des copies

Exemple d'une copie “qui marche” :

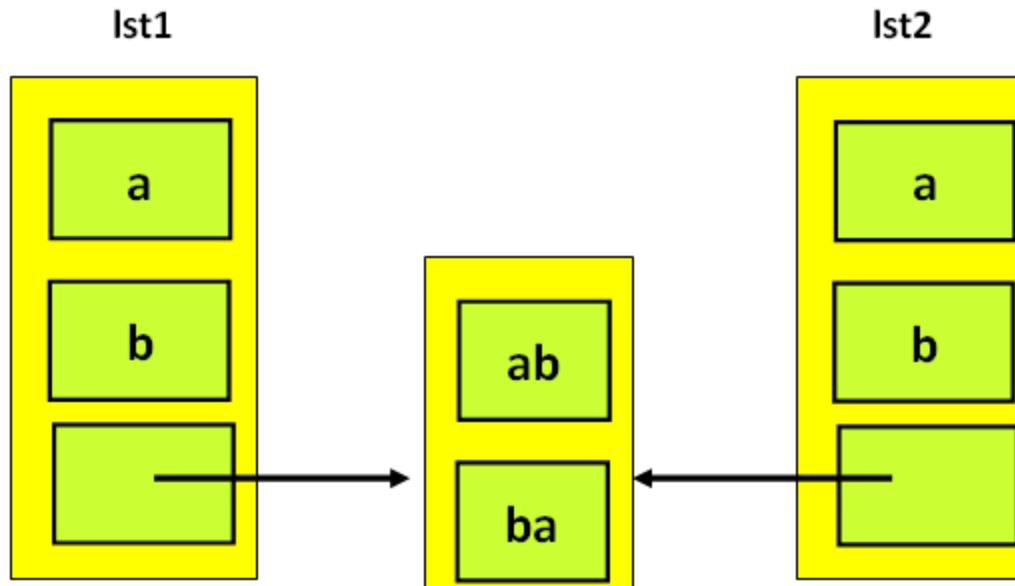
```
>>> maListe1 = [1, 2, 3, 4]
>>> maListe2 = maListe1[:]
>>> maListe2[3] = 666
>>> print(maListe1)
[1, 2, 3, 4]
>>> maListe1[0] = 'Python'
>>> print(maListe2)
[1, 2, 3, 666]
>>> print(maListe1)
['Python', 2, 3, 4]
```

Structures de données et leurs utilisations

Le délicat problèmes des copies

Exemple d'une copie “qui marche” :

```
>>> lst1 = ['a', 'b', ['ab', 'ba']]  
>>> lst2 = lst1[:]
```

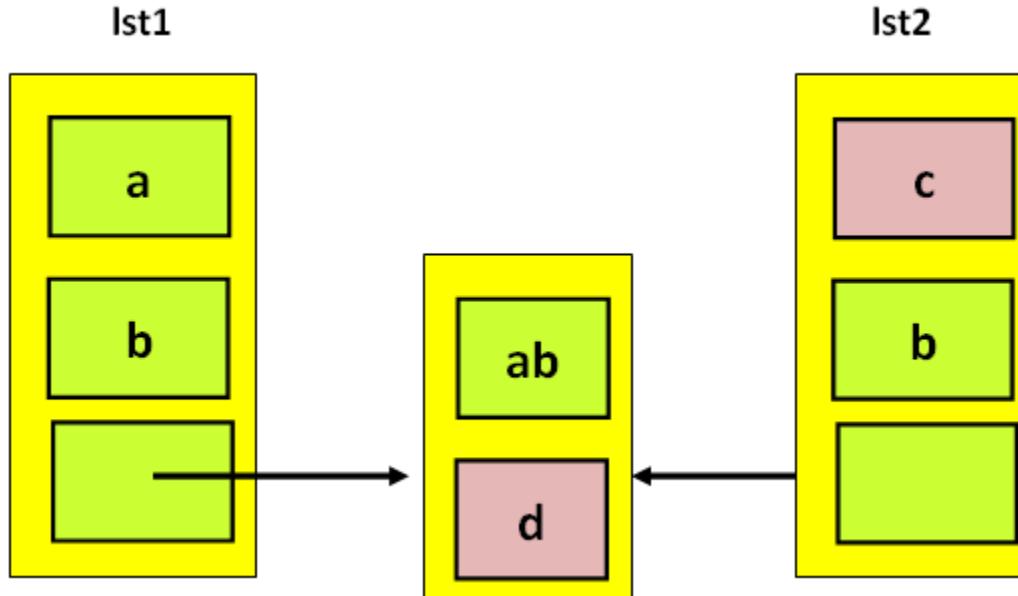


Structures de données et leurs utilisations

Le délicat problèmes des copies

Exemple d'une copie “qui marche” :

```
>>> lst1 = ['a', 'b', ['ab', 'ba']]  
>>> lst2 = lst1[:]  
>>> lst2[0] = 'c'  
>>> lst2[2][1] = 'd'  
>>> print(lst1)  
['a', 'b', ['ab', 'd']]
```



Structures de données et leurs utilisations

Le délicat problèmes des copies

Solution pour réaliser une copie profonde :

Tout est copié récursivement, il n'y a pas de référence vers des objets commun entre la source et la destination.

```
import copy  
maListe2 = copy.deepcopy(maListe1)
```

Exemple d'une copie profonde :

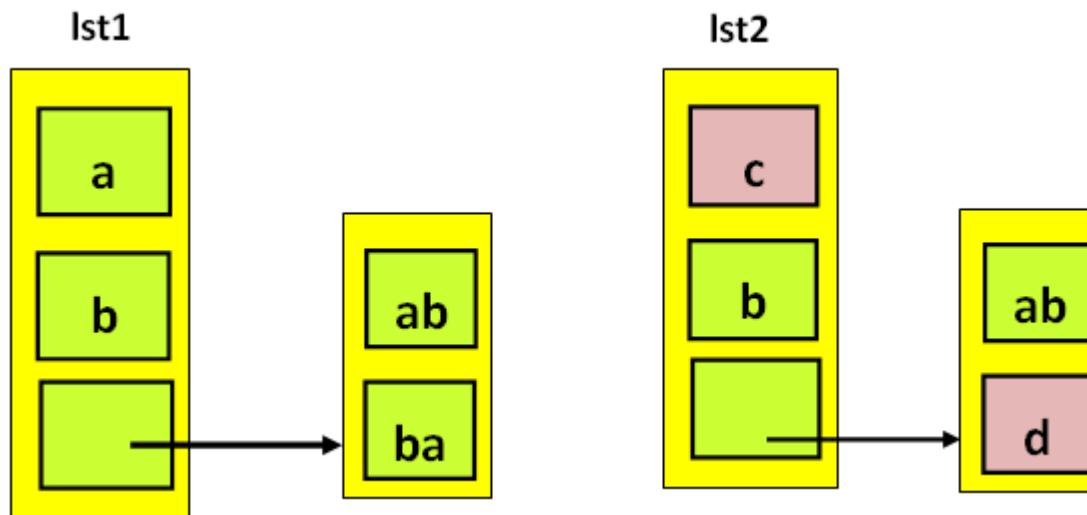
```
>>> maListe1 = [1, [11, 22]]  
>>> maListe2 = list(maListe1)  
>>> maListe2[1][0] = 33  
>>> print(maListe1)  
[1, [33, 22]]  
>>> import copy  
>>> maListe3 = copy.deepcopy(maListe1)  
>>> maListe3[1][0] = 44  
>>> print(maListe1)  
[1, [33, 22]]
```

Structures de données et leurs utilisations

Le délicat problèmes des copies

Autre exemple d'une copie profonde :

```
>>> from copy import deepcopy
>>> lst1 = ['a', 'b', ['ab', 'ba']]
>>> lst2 = deepcopy(lst1)
>>> lst2[2][1] = "d"
>>> lst2[0] = "c";
>>> print lst2
['c', 'b', ['ab', 'd']]
>>> print lst1
['a', 'b', ['ab', 'ba']]
```



Les listes multidimensionnelles

- Comme constaté sur un exemple les éléments d'une liste peuvent eux-mêmes être une liste.
- On peut ainsi créer des listes **multidimensionnelles**.
- L'accès aux élément se fait alors avec une syntaxe de la forme :

maListe[i₁] [i₂] ... [i_N]

Les listes multidimensionnelles

Remarque importante et exemple : attention à l'initialisation des listes multidimensionnelles.

```
plateau = [[0]*3]*4  
print(plateau)  
plateau[3][2] = 666  
print(plateau)
```



```
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]  
[[0, 0, 666], [0, 0, 666], [0, 0, 666], [0, 0, 666]]
```

Les listes multidimensionnelles

Remarque importante et exemple (suite) : attention à l'initialisation des listes multimensionnelles.

```
plateau = [[0]*3 for i in range(4)]  
  
print(plateau)  
  
plateau[3][2] = 666  
  
print(plateau)
```



```
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]  
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 666]]
```

Les listes multidimensionnelles

Remarque : le parcours de listes multidimensionnelles se fera en imbriquant des boucles for.

```
plateau = [[0]*3 for i in range(4)]  
plateau[3][2] = 666  
for ligne in plateau:  
    for x in ligne:  
        print(x, ' ', end=' ')  
    print('\n')
```



0	0	0
0	0	0
0	0	0
0	0	666

Diviser pour régner

Principe

Trois étapes :

- ① **Diviser** : on **divise** les données initiales en **plusieurs** sous-parties.
- ② **Régner** : on **résout récursivement** chacun des sous-problèmes associés (ou on les résout directement si leur taille est assez petite).
- ③ **Combiner** : on **combine** les différents résultats obtenus pour obtenir une solution au problème initial.

- On va se familiariser avec ce **principe général** sur plusieurs **exemples**.
- Pour chacun d'eux on présentera en détail les trois phases : **diviser, régner, combiner**.

Diviser pour régner

Ex. 1 : calcul du maximum d'une liste.

- **Problème** : calculer le maximum d'une liste de nombres
- **Résolution** :
 - ① Diviser la liste en deux sous-listes en la “coupant” par la moitié.
 - ② Rechercher le maximum de chacune de ces sous-listes.
 - ③ Comparer les résultats obtenus.

Exercice : écrire une fonction en Python implémentant l'algorithme précédent.

Diviser pour régner

Solution :

```
def maximum(l,d,f):  
    if d == f:  
        return l[d]  
    m = (d+f) // 2  
    x = maximum(l,d,m)  
    y = maximum(l,m+1,f)  
    return x if x > y else y  
  
maListe = [38,-3, 2, 1, 7]  
print(maximum(maListe,0,4))
```



38

Diviser pour régner

Ex. 2 : recherche d'un élément.

- **Problème** : rechercher la présence d'un élément dans une liste.
- **Résolution** :
 - ① Diviser la liste en deux sous-listes en la "coupant" par la moitié.
 - ② Rechercher la présence de l'élément dans chacune de ces sous-listes.
 - ③ Combiner les résultats obtenus.

Exercice : écrire une fonction en Python implémentant l'algorithme précédent.

Diviser pour régner

Solution :

```
def recherche(l,x,d,f):
    if d == f:
        return l[d] == x
    m = (d+f) // 2
    return recherche(l,x,d,m) or
recherche(l,x,m+1,f)

maListe = [38,-3, 2, 1, 7]
print(recherche(maListe,7,0,4))
```

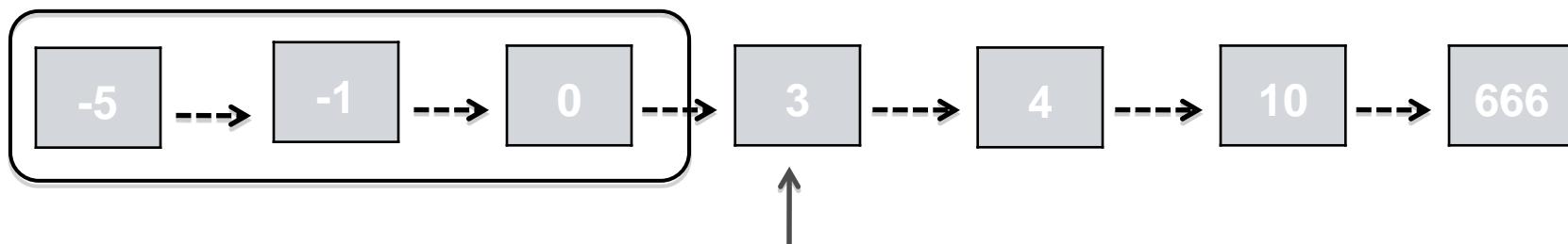


True

Diviser pour régner

Ex. 2 : recherche d'un élément.

- **Problème** : rechercher la présence d'un élément dans une liste **triée**.
- **Idée** : adopter une méthode dichotomique.
- La liste étant triée, il est facile de voir dans **quelle moitié** peut éventuellement se trouver l'élément cherché.
- **Idée** (suite) : quand on cherche un élément dans une liste triée, après comparaison avec l'élément du milieu il est facile de voir dans quelle moitié continuer la recherche.



- **Exemple** : si on recherche la valeur -4, étant plus petite que la valeur centrale 3, on va continuer la recherche uniquement dans la première partie de la liste.

Diviser pour régner

Ex. 2 : recherche d'un élément.

- **Problème** : rechercher la présence d'un élément dans une liste **triée**.
- **Résolution** :
 - ① Comparer l'élément recherché avec l'élément situé au milieu de la liste.
 - ② Diviser la liste en deux sous-listes en la "coupant" par la moitié.
 - ③ Rechercher la présence de l'élément dans la "bonne" des deux sous-listes.
 - ④ Pas de résultats à combiner puisque l'on ne "travaille" que sur une des deux sous-listes.

Exercice : écrire une fonction en Python implémentant l'algorithme précédent. On supposera donc que la liste passée en paramètre est triée.

Structures de données et leurs utilisations

Diviser pour régner

Ex. 2 : recherche d'un élément.

Solution :

```
def dicho(l,x,d,f):
    if d > f:
        return False
    else:
        m = (d+f) // 2
        if l[m] == x:
            return True
        else:
            if x < l[m]:
                return dicho(l,x,d,m-1)
            else:
                return dicho(l,x,m+1,f)
```

Diviser pour régner

Ex. 2 : recherche d'un élément.

Solution 2 : on profite ici de certaines spécificités du Python

```
def dicho2(l,x):
    if len(l) == 0:
        return False
    else:
        m = (len(l)-1) // 2
        if l[m] == x:
            return True
        else:
            if x < l[m]:
                return dicho2(l[:m],x)
            else:
                return dicho2(l[m+1:],x)
```

Structures de données et leurs utilisations

Les tableaux multi dimensionnels

Pour travailler avec les tableaux de données multidimensionnels on utilise le module ***numpy***. C'est un module utilisé dans presque tous les projets de calcul numérique sous Python. Il fournit des structures de données performantes pour la manipulation de vecteurs et matrices.

```
import numpy as np
```

On pourrait aussi faire :

```
from numpy import *
```

Dans la terminologie numpy, les vecteurs et les matrices sont appelés ***arrays***.

Avec la première importation, on peut utiliser les fonctions du module ***numpy*** de la manière suivante :

```
np.nom_fonction(...)
```

Avec la deuxième importation, on peut utiliser directement ces fonctions sans les précéder par "np.".

Les tableaux multi dimensionnels

■ Création des tableaux

- Au moyen de la fonction `numpy.array`

```
v = np.array([1,2,3,4])
```

Matrice M 4×3

```
M = np.array([[16,2,3],[0,8,3],[1,72,83],[1,3,3]])
```

Remarques

- Dans ce cas, Les tableaux **V** et **M** sont tous deux du type **ndarray** (fourni par **numpy**)
- On peut définir le type de manière explicite en utilisant le mot clé **dtype** en argument:

```
M = np.array([[1, 2], [3, 4]], dtype=complex)
```

- Autres types possibles avec **dtype** : **int**, **float**, **complex**, **bool**, etc.

Les tableaux multi dimensionnels

▪ Création des tableaux

• Au moyen de la fonction `numpy.arange`

On peut créer un vecteur à l'aide de la fonction **arange**, c'est une fonction similaire à la fonction **range** de Python :

np.arange(M,N,P) : tableau contenant les éléments de M à N avec pas P (M et P sont optionnels. Dans ce cas les valeurs par défaut sont M=0 et P=1)

```
a = np.arange(10)    ⇔ a = np.array([0, 1, 2, 3, 4, 5, 6,  
7, 8, 9])
```

• Au moyen de la fonction `numpy.empty`

```
np.empty(N,dtype='int') : tableau vide de N entiers  
np.empty(N,dtype='float') : tableau vide de N réels
```

dtype est optionnel dans toutes les créations de tableaux et fixé par défaut à **float**

Les tableaux multi dimensionnels

■ La méthode reshape

C'est une méthode qui permet de redimensionner un tableau selon les paramètres communiqués à cette méthode :

Exemple

```
A=np.empty(8,dtype='float') #A est à présent un vecteur  
#de 8 réels  
A=A.reshape(4,2) #A devient maintenant une  
#matrice 4×2 de réels  
  
for i in range(4):  
    for j in range(2):  
        A[i][j]=i*j
```

Les tableaux multi dimensionnels

■ La méthode reshape

Autres possibilités:

`np.zeros(N)` : tableau de N zéros.

`np.ones((N,M))` : tableau de dimension (N,M) de uns (matrice).

`np.zeros_like(a)` : tableau de zéros de même dimension et type que le tableau a.

`np.ones_like(a)` : tableau de uns de même dimension et type que le tableau a.

`np.empty_like(a)` : tableau vide de même dimension et type que le tableau a.

`np.linspace(a,b,N)` : tableau contenant N éléments uniformément répartis de a à b. (a et b sont compris)

`b=a.copy()` : copie d'un tableau a dans un tableau b

Rem : `b=a` ne copie pas les éléments, mais crée juste une référence, i.e. si a est modifié b l'est aussi.

Algorithmes de tri

Tri par sélection

Principe : Cette méthode peut se traduire par l'algorithme formel suivant :

- 1- **Comparer** tous les nombres afin de **sélectionner le plus petit** (ordre croissant),
- 2- **Échanger** le plus petit élément trouvé avec le premier élément,
- 3- Refaire les étapes 1 et 2 et rechercher le plus petit du tableau **sauf le premier** puis l'échanger avec le second.

```
def tri_selection(l):
    n = len(l)
    for i in range(n-1):
        k=i
        for j in range(i+1,n):
            if l[j] < l[k]:
                k = j
        if i!=k:
            tmp = L[i]
            L[i] = L[k]
            L[k] = tmp
```

```
>> L=[1,2,8,4,6,3,-5,-7,18]
>> tri_selection(L)
>> print(L)
```

[-7, -5, 1, 2, 3, 4, 6, 8, 18]

Algorithmes de tri

Tri par insertion

Principe : C'est une méthode de tri qui consiste à prendre les éléments du tableau **un par un** puis d'insérer chacun à sa bonne place façon que les éléments traités forment une liste triée.

Cette méthode peut se traduire par l'algorithme formel suivant :

- 1- On commence par le **deuxième** élément,
- 2- **Comparer** l'élément choisis avec tous ses **précédents** dans le tableau et **l'insérer** dans sa **bonne place**,
- 3- **Répéter** l'étape 2 pour l'**élément suivant** jusqu'à arriver au **dernier**.

```
def tri_Insersion(l) :  
    n = len(l)  
    for i in range(1,n) :  
        j=i  
        T=l[i]  
        while(j>0 and l[j-1]>T) :  
            l[j]=l[j-1]  
            j=j-1  
        l[j]=T
```

Algorithmes de tri

Tri bulle

Principe : Le tri bulle, consistant à comparer, en commençant par la fin de la liste, les couples d'éléments successifs : Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. On fait ainsi remonter le terme le plus petit (de même que les bulles les plus légères remontent le plus vite à la surface ...). D'où le nom donné à ce tri. Puis on itère le procédé sur le sous-tableau restant :

```
def triBulle(l) :  
    n=len(l)  
    for i in range(n-1) :  
        for j in range(n-1,i,-1) :  
            if l[j-1]>l[j] :  
                T=l[j]  
                l[j] = l[j-1]  
                l[j-1] = T
```

Les t-uples

Les t-uples correspondent aux listes à la différence qu'ils sont **non modifiables**. Pratiquement, ils utilisent les parenthèses au lieu des crochets :

- Syntaxes de la déclaration de t-uples :

```
monTupleVide = ()
```

```
monTupleAvecUnSeulElement = (élément,)
```

```
monTuple = (élément1, élément2, ..., élémentN)
```

- Exemple de création/manipulation de t-uples :

```
>>> monTuple = (2.718, 3.14, 1.414)
```

```
>>> print(monTuple)
```

```
(2.718, 3.14, 1.414)
```

```
>>> print(monTuple[2])
```

```
1.414
```

```
>>> monTuple[2] = 1.732
```

```
Traceback (most recent call last):
```

```
  File "<console>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

Structures de données et leurs utilisations

Les t-uples

L'affectation et l'indexage fonctionne comme avec les listes, mais si l'on essaie de modifier un des éléments du t-uple, Python renvoie un message d'erreur. Si nous voulons ajouter un élément (ou le modifier), nous devons créer un autre t-uple :

```
>>> x = (1,2,3)
>>>print( x + (2,) )      #concaténation
(1, 2, 3, 2)
```

Remarque : Pour utiliser un t-uple d'un seul élément, nous devons utiliser une syntaxe avec une virgule (element,), ceci pour éviter une ambiguïté avec une simple expression.

Autre particularité des t-uples, il est possible d'en créer de nouveaux sans les parenthèses.

Une fonction peut retourner plusieurs valeurs sous forme de t-uple.

```
>>> x = (1,2,3)
>>> print(x)
(1, 2, 3)
>>> x = 1,2,3
>>> print(x)
(1, 2, 3)
```

```
>>> def f(x):
        return x,x**2
>>> f(3)
(3,9)
```

Parcours d'une séquence avec “for”.

Itération sur les éléments :

```
for x in maSéquence:
    traitement de l'élément x
```

Exemple:

```
l = [-2, 3, 11]
for x in l:           →
    print(x**2)
```

4
9
121

Itération sur les couples (indice, élément) :

```
for i,x in enumerate(maSéquence):
    traitement de l'élément x et/ou de l'indice i
```

Exemple:

```
l = (-2, 3, 11)
for i, x in enumerate(l):           →
    print("élément n° ",i+1,":",x**2)
```

élément n° 1 : 4
élément n° 2 : 9
élément n° 3 : 121

Itération sur les indices :

```
for i in range(len(maSéquence)):
    traitement de l'élément maSéquence[i]
    et/ou de l'indice i
```

Exemple:

```
l = (-2, 3, 11)           →
for i in range(len(l)):
    print("élément n° ",i+1,":",l[i]**2)
```

élément n° 1 : 4
élément n° 2: 9
élément n° 3: 121

Itération sur deux séquences :

```
for x,y in zip(maSéquence1,maSéquence2):
    traitement des éléments x et y
```

Exemple: l = (-2, 3, 11)

 m = 'Keith'

```
for x,y in zip(l,m):
    print(x,y)
```

-2 k
3 e
11 i

les chaînes de caractères

- **Manipulation des chaînes de caractères**

Lecture d'une chaine de caractères :

```
Ch=input("entrer une chaine de caractère : ")
```

Affichage d'une chaine de caractères :

```
print(Ch)
```

Accès à un caractère d'une chaine

```
nomChaine[indice]
```

Remarque : En Python, les indices commencent de 0

- **Opérations sur les chaînes de caractères**

Opérateur	Signification
+	concaténation
*	répétition
=	affectation
==, <, >, >=, <=, !=	Comparaison
in	
not in	

Structures de données et leurs utilisations

les chaînes de caractères

■ Exemples

■ Concaténation

```
>>> s = 'i vaut'  
>>> i = 1  
>>> print s + i  
  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: cannot concatenate 'str' and 'int'  
objects  
>>> print s + " %d %s"%(i, "m.")  
    i vaut 1 m.  
>>> print s + ' ' + str(i)  
    i vaut 1
```

■ Répétition

```
>>> print '*'*5  
*****
```

les chaînes de caractères

- ***Les fonctions / Les méthodes de chaînes de caractères***

- ✓ On peut agir sur une chaîne (et plus généralement sur une séquence) en utilisant des fonctions (notion procédurale) ou des méthodes (notion objet).
- ✓ Pour appliquer une fonction, on utilise l'opérateur () appliqué à la chaîne.

- ***Les fonctions***

fonction	signification
Ch=input()	lecture
print(Ch)	affichage
len(Ch)	retourne la longueur
ord(caracter)	retourne le code ascii
chr(nombre)	retourne le caractère
.....

Exemple :

```
>>> ch1 = "abc"  
>>> long =  
len(ch1)  
3
```

les chaînes de caractères

•*Les méthodes*

On applique une méthode à un objet en utilisant la notation pointée entre la donnée/variable à laquelle on applique la méthode, et le nom de la méthode suivi de l'opérateur () appliqué à la méthode.

Méthode	signification
count(chars)	retourne le nombre d'occurrences de chars dans la chaîne
upper	retourne l'équivalent en majuscule de la chaîne
lower	retourne l'équivalent en minuscule de la chaîne
isupper	retourne True si la chaîne ne contient que des majuscules
islower	retourne True si la chaîne ne contient que des minuscules
isalnum	retourne True si la chaîne ne contient que des alphanumériques
isalpha	retourne True si la chaîne ne contient que des alphabétiques
isdigit	retourne True si la chaîne ne contient que des numériques
isspace	retourne True si la chaîne ne contient que des espaces
strip(chars)	supprime toutes les combinaisons de chars (ou l'espace par défaut) au début et en fin de la chaîne

Structures de données et leurs utilisations

les chaînes de caractères

Exemple

```
>>> ch2 = "abracadabra"
```

```
>>> ch3 = ch2.upper()
```

ABRACADABRA

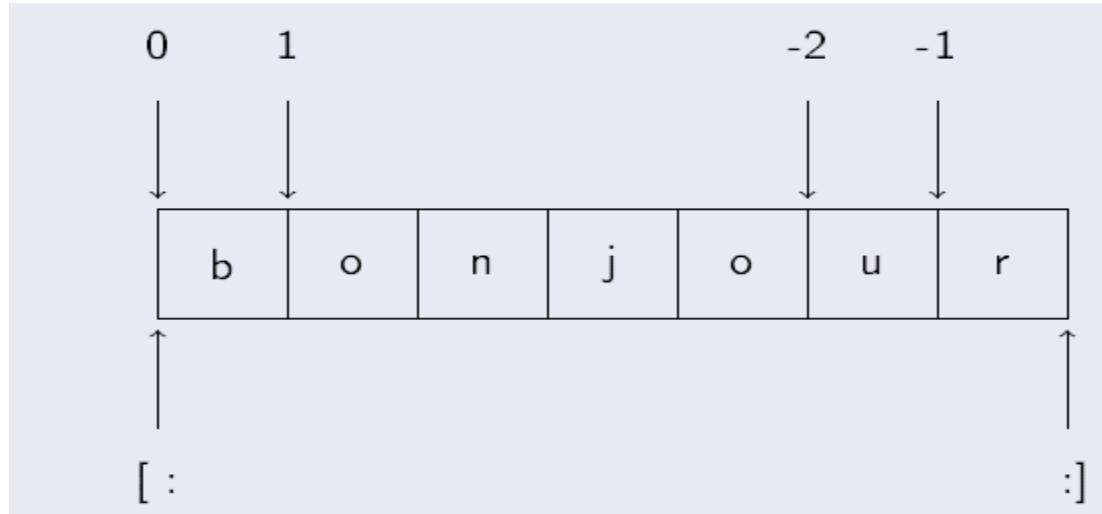
■ Accès à un caractère individuel ou à des sous-chaînes

- Python offre une méthode simple pour accéder aux caractères contenus dans une chaîne: une chaîne est manipulée comme une séquence indexée de caractères.
- chaque caractère est accessible directement par son index (le premier étant indexé 0) en utilisation des crochets.
- Il est possible d'accéder à des sous-chaînes en précisant la tranche souhaitée l'index de début (qui est inclus) étant séparé de l'index de fin (qui est exclu) par le caractère « : »
- Si l'index de début est manquant, python considère que le début est 0, et si l'index de fin est manquant il prend la longueur de la chaîne.

les chaînes de caractères

- **Accès à un caractère individuel ou à des sous-chaînes**

Accès au caractères [debut : n : pas]



Exemple 1

```
>>> x = 'hello world!'
```

```
>>> x[:5] == x[0:5]
```

True

```
>>> x[3:] == x[3:len(x)]
```

True

les chaînes de caractères

Exemple 2

```
>>> "bonjour"[3]; "bonjour"[-1]
'J'
'r'

>>> "bonjour"[2:]; "bonjour":[3]; "bonjour"[3:5]
'njour'
'bon'
'jo'
>>> 'bonjour'[-1::-1]
'ruojnob'
```

Dans le cas des sous-chaînes, la valeur fournie est une copie et non un accès à une partie de la chaîne d'origine.

<pre>>>> x = 'hello world!' >>> x[4] 'o' >>> x[2:4] 'll' >>> x[3:] 'lo world!' >>> x[:] 'hello world!'</pre>	<u>enlever le deuxième 'l'</u> <pre>>>> x= "hello!" >>> x[:3]+x[4:] 'heLo!'</pre> <u>insérer un Z entre les deux l 'l'</u> <pre>>>> x= "hello!" >>> x[:3]+"Z"+x[3:] 'heLzo!'</pre>	<u>remplacer le deuxième 'l' par P</u> <pre>>>> x= "hello!" >>> x[:3]+"P"+x[4:] 'helPo!'</pre>	<u>Enfin, un index négatif précise que le calcul s'effectue depuis la fin de la chaîne.</u> <pre>>>> x[-3:] 'lo!' >>> x[1:-1] 'ello'</pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------

Les ensembles

- Python possède des structures de données implémentant la notion d'**ensemble mathématique**.
- Par "**ensemble**" on entend ici une **collection d'éléments distincts**.
- On aura deux types pour ce faire :
 - "set" qui sera une structure **mutable**.
 - "frozenset" qui elle sera **immutable**.
- Ces structures de données ne sont pas **séquentielles**, l'accès à un élément ne se fera donc pas par sa position.
- Comme en mathématiques, un ensemble ne pourra pas contenir deux fois le même élément.
- Lors de la création d'un ensemble ou lors de sa mise à jour, les **doublons** seront donc **supprimés** automatiquement.

Les ensembles

Syntaxe de la déclaration d'un ensemble “set” :

```
monSetVide = set()  
monSet = {élément1, élément2, ..., élémentN}  
monSet = set(séquence)
```

Exemple : `>>> s = {1, 3, 5, 7, 9}`

```
>>> s  
{9, 3, 1, 5, 7}  
s = set('barbara')  
>>> s  
{'r', 'b', 'a'}  
>>> s = set(range(10,21,2))  
>>> s  
{10, 12, 14, 16, 18, 20}  
>>> s.add(30)  
>>> s  
{10, 12, 14, 16, 18, 20, 30}
```

Structures de données et leurs utilisations

Les ensembles

Syntaxe de la déclaration d'un ensemble “frozenset” :

```
monFrozenSetVide = frozenset()  
  
monFrozenSet = frozenset(séquence)
```

Exemple :

```
>>> s = frozenset('barbara')  
  
>>> s  
frozenset({'r', 'b', 'a'})  
  
>>> s = frozenset([1, 5, 666])  
  
>>> s  
frozenset({1, 666, 5})  
  
>>> s.add(3)  
  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'frozenset' object has no attribute 'add'  
  
>>> s = frozenset({5, 12, 777})  
  
>>> s  
frozenset({777, 12, 5})
```

Les ensembles

Opérations communes:

- Les opérations communes aux types “set” et “frozenset” sont celles des **ensembles en mathématiques** :
 - Test d'appartenance
 - Test d'inclusion
 - Union
 - Intersection
 - Différence
 - Différence symétrique

Structures de données et leurs utilisations

Les ensembles

Opérations d'appartenance :

Opération	Résultat
x in s	Teste si x appartient à s
x not in s	Teste si x n'appartient pas à s
len(s)	Nombre d'éléments de s
s.isdisjoint(t)	True si l'intersection de s et t est vide
s.issubset(t)	True si s est inclus dans t
s <= t	
s.issuperset(t)	True si t est inclus dans s
s >= t	

Exemple:

```
>>> s = {1, 3, 5, 7, 9}
>>> s.isdisjoint([2, 4, 6, 9]) → False
>>> s.isdisjoint({2, 4, 6, 8}) → True
>>> s.issubset(range(10)) → True
>>> s.issuperset(range(1, 6, 2)) → True
```

Structures de données et leurs utilisations

Les ensembles

Opérations classiques sur les ensembles :

Opération	Résultat
s.union(t)	Union de s et t
s t	
s.intersection(t)	Intersection de s et t
s & t	
s.difference(t)	Difference de s par t
s - t	
s.symmetric_difference(t)	Différence symétrique de s et t
s ^ t	

Exemple:

```

>>> s = {1, 3, 5, 7, 9}
>>> t = frozenset({2, 4, 6, 8})
>>> u = s | t
>>> u
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> u.symmetric_difference(range(6,16))
{1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 15}
>>> u.difference(range(3), [5, 6], 'e')
{3, 4, 7, 8, 9}

```

Les ensembles

Itération sur les éléments.

- Il est tout à fait possible d'**itérer** sur les éléments d'un "set" ou d'un "frozenset".
- Leurs éléments n'étant pas mémorisés de façon séquentielle, l'ordre de parcours sera par contre **indéfini** et **imprévisible**.

Exemple:

```
>>> s = {1, 3, 5, 7, 9}  
>>> for x in s:  
...     print(x**2)  
  
...  
81  
9  
1  
25  
49
```

Structures de données et leurs utilisations

Les ensembles

Opérations de mise à jour d'un ensemble :

Opération	Résultat
<code>s.update(t)</code>	<code>s</code> mis à jour vers l'union de <code>s</code> et <code>t</code>
<code>s = t</code>	
<code>s.intersection_update(t)</code>	<code>s</code> mis à jour vers l'intersection de <code>s</code> et <code>t</code>
<code>s &= t</code>	
<code>s.difference_update(t)</code>	<code>s</code> mis à jour vers la différence de <code>s</code> par <code>t</code>
<code>s -= t</code>	
<code>s.symmetric_difference_update(t)</code>	<code>s</code> mis à jour vers la différence symétrique de <code>s</code> et <code>t</code>
<code>s ^= t</code>	

```
>>> s = set('barbara')
>>> s
{'r', 'b', 'a'}
>>> s.update('boris')
>>> s
{'o', 'i', 'b', 'a', 'r', 's'}
>>> s.intersection_update('brio')
>>> s
{'o', 'r', 'b', 'i'}
```

```
>>> s=set({1,5,8,9})
>>> t=set({8,7,1,6,3})
>>> s.difference_update(t)
>>> s
{5, 9}
```

Structures de données et leurs utilisations

Les ensembles

Opérations d'ajouts et de suppressions d'éléments :

Opération	Résultat
<code>s.add(x)</code>	Ajoute l'élément <code>x</code> à <code>s</code>
<code>s.remove(x)</code>	Supprime l'élément <code>x</code> de <code>s</code> , retourne une erreur s'il est absent
<code>s.discard(x)</code>	Supprime l'élément <code>x</code> de <code>s</code> s'il est présent, ne retourne rien s'il est absent
<code>s.pop()</code>	Supprime et retourne un élément de <code>s</code> quelconque
<code>s.clear()</code>	Supprime tous les éléments de <code>s</code>

Exemple:

```
>>> s = set()
>>> s.add(666)
>>> s.add('VL')
>>> s.remove(777)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    KeyError: 777
>>> s.discard(777)
>>> s.pop()
666
>>> s
{'VL'}
```

Les dictionnaires

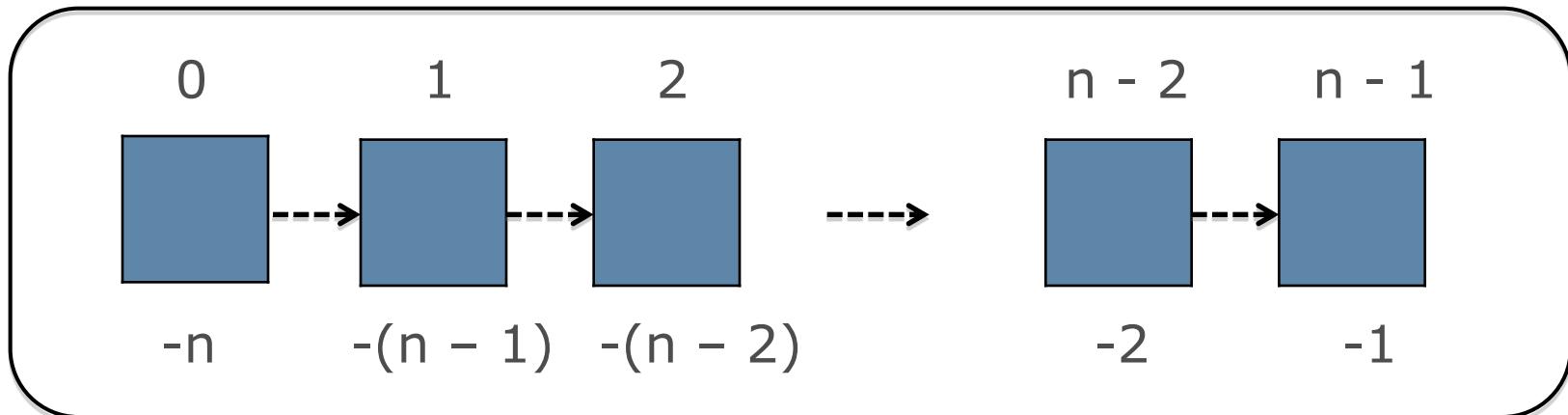
Principe

- Un dictionnaire est une structure de données **non séquentielle, mutable**, permettant de mémoriser des couples **clé : valeur**.
- L'accès à une **valeur** se fait via sa **clé**.
- Les **clés** doivent nécessairement être d'un type **immuable** (int, str, tuple,etc.) et sont toutes **différentes**.
- Les valeurs sont par contre de **n'importe quel type**.
- Contrairement aux séquences, l'accès à une valeur ne se fait donc pas par sa **position** mais par une **clé**.
- Autre différence, les données ne sont pas stockées en mémoire les unes à la suite des autres.
- Cette structure de données n'est donc pas **séquentielle**.

Structures de données et leurs utilisations

Les dictionnaires

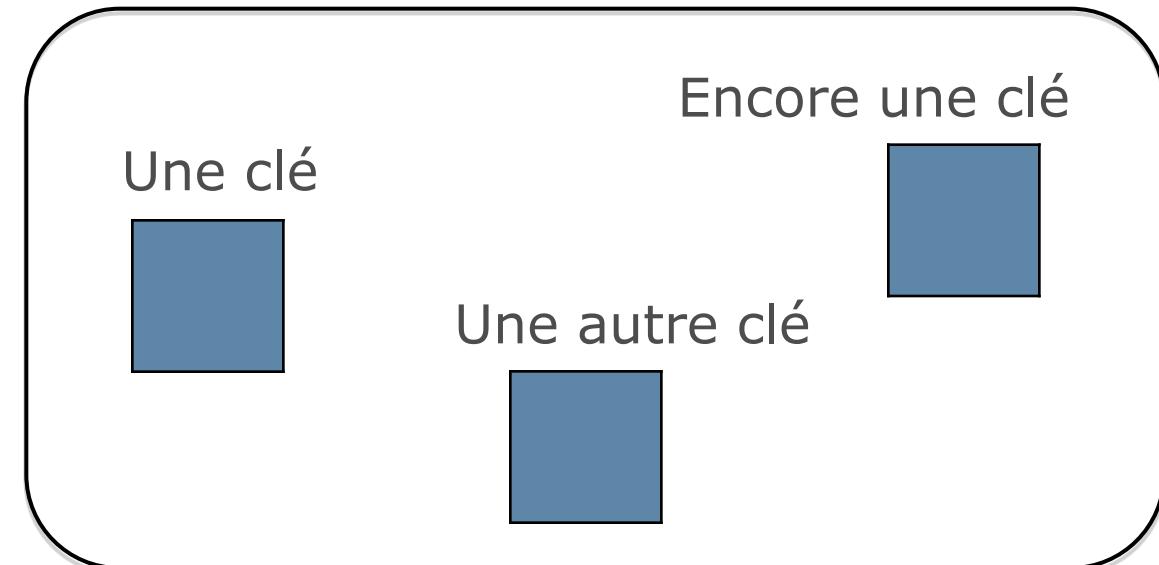
Principe



Une **séquence**

versus

un **dictionnaire**



Structures de données et leurs utilisations

Les dictionnaires

Syntaxe de la déclaration d'un dictionnaire :

```
monDicoVide = dict()  
monDicoVide = {}  
monDico = {clé1:val,clé2:val,...,cléN,val}  
monDico = dict([(clé1,val),..., (cléN,val)])  
monDico = dict((clé1,val),..., (cléN,val)))  
monDico = dict(zip([clé1,...,cléN], [val,...,val]))
```

Ajout d'un élément ou modification d'une valeur :

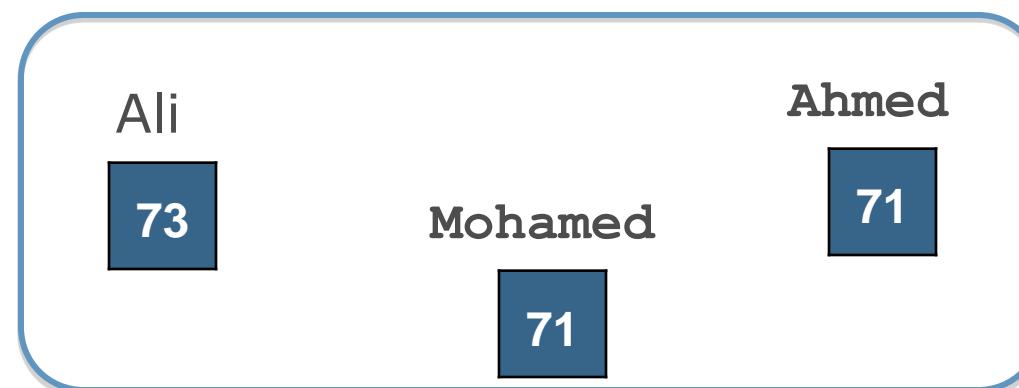
```
monDico [maClé] = maValeur
```

- Si la clé **maClé** ne fait pas partie du dictionnaire, il y a création d'un couple **maClé : maValeur**.
- Si elle en faisait partie, sa valeur associée est mise à jour vers **maValeur**.

Les dictionnaires

Exemple : à partir d'un dictionnaire vide

```
>>> rs = {}  
>>> rs['Ahmed'] = 71  
>>> rs['Mohamed'] = 71  
>>> rs['Ali'] = 83  
>>> rs  
{'Ahmed': 71, 'Ali': 83, 'Mohamed': 71}  
>>> rs['Ali'] = 73  
>>> rs  
{'Ahmed': 71, 'Ali': 73, 'Mohamed': 71}
```



Structures de données et leurs utilisations

Les dictionnaires

Exemple 2 :

```
>>> rs =  
dict([('Mick', 71), ('Keith', 71), ('Charlie', 73)])  
  
>>> rs  
  
{'Mick': 71, 'Keith': 71, 'Charlie': 73}  
  
>>> b =  
dict(zip(['John', 'Paul', 'Ringo'], [40, 72, 74]))  
  
>>> b  
  
{'Paul': 72, 'Ringo': 74, 'John': 40}
```

Les dictionnaires

Opérations sur les dictionnaires (1):

Opération	Résultat
x in d	Teste si la clé x appartient à d
x not in d	Teste si la clé x n'appartient pas à d
len(d)	Nombre d'éléments de d
d[x]	Si x est une clé, retourne d[x] sinon renvoie une erreur
del d[x]	Si x est une clé, supprime d[x] sinon renvoie une erreur
d.clear()	Supprime tous les éléments de d
d.update(e)	Met à jour d avec e

Structures de données et leurs utilisations

Les dictionnaires

Exemples :

```
>>> rs = dict([('Mick', 71), ('Keith', 71)])
>>> rs.update({'Charlie': 73, 'Ron': 67})
>>> rs
{'Ron': 67, 'Mick': 71, 'Keith': 71, 'Charlie': 73}
>>> rs['Keith']
71
>>> del rs['Ron']
>>> rs
{'Mick': 71, 'Keith': 71, 'Charlie': 73}
```

Structures de données et leurs utilisations

Les dictionnaires

Opérations sur les dictionnaires (2):

Opération	Résultat
d.get(x)	si x est une clé retourne d[x] sinon None
d.get(x,y)	si x est une clé retourne d[x] sinon y
d.pop(x)	si x est une clé retourne d[x] et supprime le couple x : d[x] , sinon renvoie une erreur
d.pop(x,y)	si x est une clé retourne d[x] et supprime le couple x : d[x] , sinon retourne y
d.popitem()	Si le dictionnaire est vide retourne une erreur, sinon retourne un couple arbitraire de d et le supprime
d.setdefault(x,y)	si x est une clé retourne d[x] sinon retourne y et insère le couple x : y

Les dictionnaires

Exemples :

```
>>> rs = {'Mick': 71, 'Keith': 71, 'Charlie': 73}  
>>> rs.get('Mick')  
71  
>>> rs.get('Ron', "Pas là")  
'Pas là'  
>>> rs.pop('Keith')  
71  
>>> rs.setdefault('Ron', 67)  
67  
>>> rs  
{'Ron': 67, 'Mick': 71, 'Charlie': 73}
```

Les dictionnaires

Itération sur les éléments.

- Il est tout à fait possible d'**itérer** sur les éléments d'un dictionnaire. Plus précisément on peut itérer sur les **clés**, sur les **valeurs** ou sur les couples **clé : valeur**.
- De même que pour les ensembles, les éléments n'étant pas mémorisés de façon séquentielle, l'ordre de parcours sera par contre **indéfini** et **imprévisible**.

Structures de données et leurs utilisations

Les dictionnaires

Exemple : itération sur les **clés**, deux possibilités

```
rs = {'mick':71,'Keith':71,'Charlie':73,'Ron':67}

for nom in rs:

    print(nom,end=' ')
```

```
rs = {'mick':71,'Keith':71,'Charlie':73,'Ron':67}

for nom in rs.keys():

    print(nom,end=' ')
```



mick Ron Charlie Keith

Structures de données et leurs utilisations

Les dictionnaires

Exemple : itération sur les **valeurs**

```
rs = {'mick':71,'Keith':71,'Charlie':  
      73,'Ron':67}  
  
total = 0  
  
for age in rs.values():  
    total += age  
  
print("âge cumulé :",total,'ans')
```



âge cumulé : 282 ans

Exemple : itération sur les couples **clé : valeur**

```
rs = {'mick':71,'Keith':71,'Charlie':73,'Ron':67}  
  
for nom,age in rs.items():  
    print(nom,':',age,'ans')
```



mick : 71 ans
Charlie : 73 ans
Keith : 71 ans
Ron : 67 ans

Les dictionnaires

Les méthodes **keys()** , **values()** et **items()**

- Les méthodes **keys()** et **values()** renvoient respectivement la liste des clés et la liste des valeurs d'un dictionnaire

```
>>> print(a.keys())
dict_keys(['nom', 'poids', 'taille'])
>>> print(a.values())
dict_values(['girafe', 1100, 5.0])
```

- On peut aussi initialiser les éléments d'un dictionnaire en une seule opération :

```
>>> b = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
```

- La méthode **items()** renvoie la liste des tuples (**clé,valeur**) d'un dictionnaire

```
>>> print(b.items())
dict_items([('nom', 'singe'), ('poids', 70),
('taille', 1.75)])
```

Structures de données et leurs utilisations

Les dictionnaires

Exemple:

```
>>> rs =  
{'mick':71,'Keith':71,'Charlie':73,'Ron':67}  
>>> c = rs.keys()  
>>> list(c)  
['mick', 'Charlie', 'Keith', 'Ron']  
>>> del rs['Ron']  
>>> list(c)  
['mick', 'Charlie', 'Keith']  
>>> tuple(rs.items())  
([('mick', 71), ('Charlie', 73), ('Keith', 71)])
```

Les Fichiers

Généralités

- **Objectif de la gestion de fichiers :**

- Pouvoir conserver des données en mémoire de **façon durable**.
- Pour l'instant nos données ne sont disponibles que **pendant** l'exécution du programme.
- Pour nous, un fichier sera donc un **support** pour conserver une masse de données.

- **Deux principaux types de fichiers :**

- **Fichier texte** : fichier organisé sous la forme d'une suite de lignes, chacune étant composée d'un certain nombres de caractères et terminée par '\n'.
- **Fichier binaire** : suite d'octets, pouvant représenter toutes sortes de donnée. Ils ne peuvent être lues qu'avec le logiciel adéquat.

Structures de données et leurs utilisations

Les Fichiers

Ouverture d'un fichier

- **Syntaxe de l'ouverture d'un fichier :**

```
monFichier = open('NomDuFichier', 'mode')
```

- Le nom du fichier contient également son **extension** (.txt, .csv, ...).
 - Le mode est la **façon** selon laquelle on va ouvrir le fichier.
-
- **Remarque :**
 - Par défaut on considère que le fichier est situé dans le **même répertoire** que notre script.
 - Dans le cas contraire, on devra préciser son **emplacement**, par exemple avec un chemin absolu (du style C:\ ... sous windows).

Structures de données et leurs utilisations

Les Fichiers

Ouverture d'un fichier

■ Trois modes d'ouverture possibles :

- ‘r’ : **lecture**. On peut lire le contenu du fichier mais pas y écrire. Le fichier doit exister auparavant.
- ‘w’ : **écriture**. Si le fichier existait, son contenu est effacé, sinon il est créé.
- ‘a’ : **ajout**. Si le fichier existait on peut écrire à la fin de celui-ci sans effacer son contenu. Sinon il est créé.

Structures de données et leurs utilisations

Les Fichiers

Fermeture d'un fichier

- **Syntaxe de la fermeture d'un fichier :**

```
monFichier.close()
```

- Cela permet de **libérer la mémoire** allouée à 'monFichier'.
- Cela permet également à d'**autres applications** d'accéder au fichier en question.



Structures de données et leurs utilisations

Les Fichiers

Lecture dans un fichier

■ Syntaxe pour lire l'intégralité d'un fichier :

```
monFichier.read()
```

- Cette instruction retourne l'intégralité du fichier sous la forme d'un 'str'.

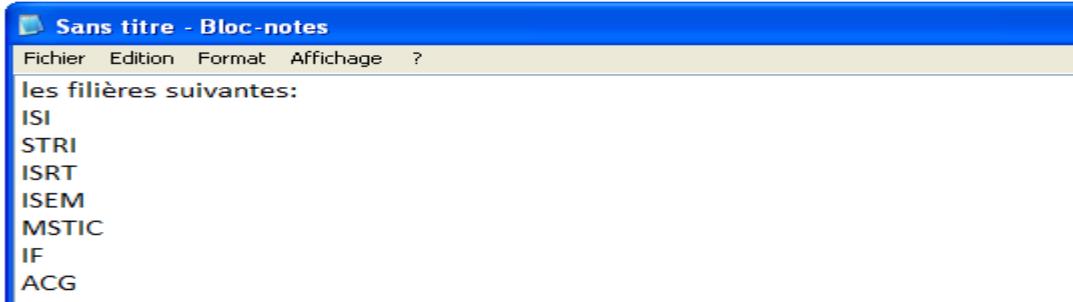
■ Remarque :

- Si l'on passe un paramètre entier à la fonction 'read', on ne lira alors que ce nombre précis de caractère.
- On sera alors positionné à cet endroit pour une lecture ultérieure.

Les Fichiers

Lecture dans un fichier

- **Exemple:** imaginons que nous ayons un fichier texte nommé `fich.txt` dans notre répertoire courant.



```
>>> monFichier = open('fich.txt', 'r')
>>> monTexte = monFichier.read()
>>> print(monTexte)
```

Les filières suivantes:

ISI
SRI
ISRT
ISEM
MSTIC
IF
ACG

Structures de données et leurs utilisations

Les Fichiers

Lecture dans un fichier

- **Remarque :**

- Si le fichier est volumineux lire l'intégralité de celui-ci d'un "seul coup" peut provoquer des **débordements de mémoire**.
- Il sera préférable alors de passer un **paramètre** à la fonction "read".

- **Syntaxe pour lire une ligne d'un fichier :**

```
monFichier.readline()
```

- **Syntaxe pour créer une liste constituée de toutes les lignes d'un fichier :**

```
monFichier.readlines()
```

Les Fichiers

Lecture dans un fichier

- Exemples:

```
>>> monFichier = open('fich.txt', 'r')  
>>> ligne1 = monFichier.readline()  
>>> print(ligne1)
```

Les filières suivantes:

```
>>> ligne2 = monFichier.readline()  
>>> print(ligne2)
```

ISI

```
>>> monFichier = open('fich.txt', 'r')  
>>> maListe = monFichier.readlines()  
>>> print(maListe)  
["Les filières suivantes:\n", "ISI\n", "ISRT\n",  
 "ISEM\n", "MSTIC\n", "IF\n", "ACG"]
```

Structures de données et leurs utilisations

Les Fichiers

Lecture dans un fichier

- **Itération sur les lignes d'un fichier :**

```
for maLigne in monFichier:  
    traitement de la ligne maLigne
```

- **Exemple :** avec le même fichier que précédemment, refermé entre temps.

```
monFichier = open('fich.txt', 'r')  
for ligne in monFichier:  
    print(ligne.upper(), end=' ')  
monFichier.close()
```

LES FILIERES SUIVANTES:

ISI
STRI
ISRT
ISEM
MSTIC
IF
ACG

Structures de données et leurs utilisations

Les Fichiers

Écriture dans un fichier.

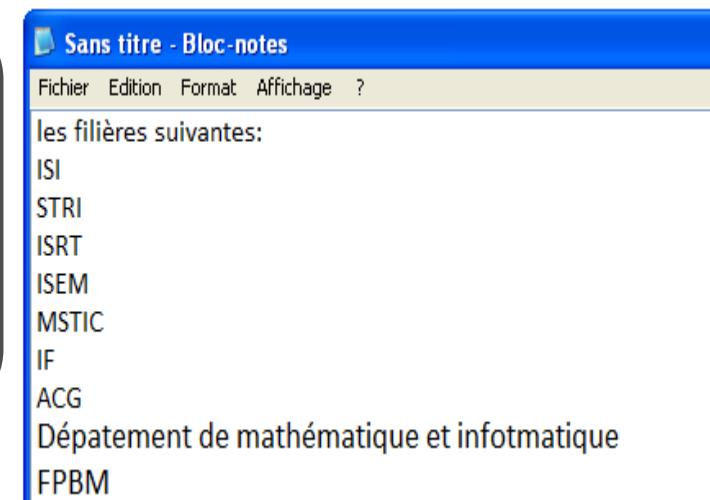
▪ **Syntaxe pour écrire dans un fichier :**

```
monFichier.write('le texte à écrire')
```

- Si on utilise plusieurs fois cette fonction, on écrira les différents textes **les uns à la suite des autres**.
 - Le paramètre est nécessairement du type ‘str’, il faut donc éventuellement **convertir** les variables numériques.

■ Exemple :

```
monFichier = open('fich.txt','a')
monFichier.write('Département de
mathématique et informatique\n')
monFichier.write('FPBM\n')
monFichier.close()
```

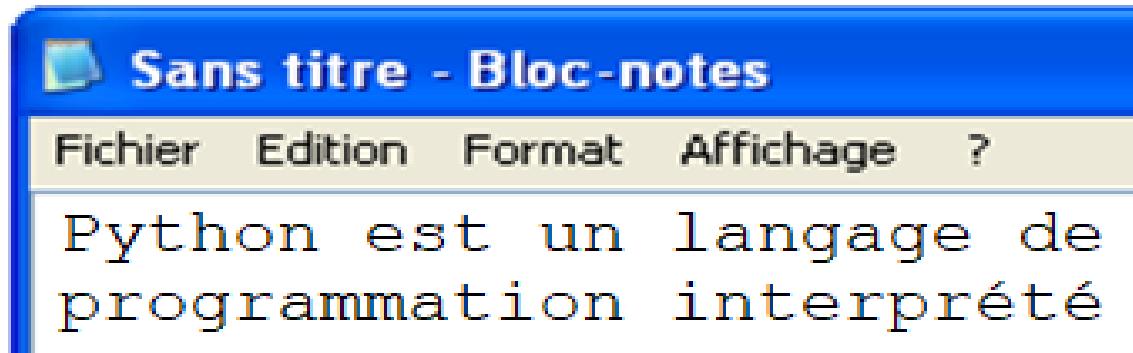


Les Fichiers

Écriture dans un fichier.

- **Exemple** : suite de l'exemple précédent, en mode ‘w’ on efface l'ancien contenu

```
monFichier = open('fich.txt','w')
monFichier.write(' Python est un langage de
programmation interprété')
monFichier.close()
```



Les exceptions

Définition

- Les erreurs détectées durant l'exécution sont appelées des **exceptions**. Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution.
- Python lève donc des **exceptions** quand il trouve une erreur, soit dans le code (une erreur de syntaxe, par exemple), soit dans l'opération que vous lui demandez de faire.

```
# Exemple classique : test d'une division par zéro
>>> variable = 1/0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

- **ZeroDivisionError**: le type de l'exception ;
- **int division or modulo by zero**: le message qu'envoie Python pour vous aider à comprendre l'erreur qui vient de se produire.

Les exceptions

Forme minimale du bloc **try**

On va parler ici de bloc **try**. Nous allons en effet mettre les instructions que nous souhaitons tester dans un premier bloc et les instructions à exécuter en cas d'erreur dans un autre bloc

try:

Bloc à essayer

except:

Bloc qui sera exécuté en cas d'erreur

Dans l'ordre, nous trouvons :

- Le mot-clé **try** suivi des deux points « : » (**try** signifie « essayer » en anglais).
 - Le bloc d'instructions à essayer.
- Le mot-clé **except** suivi, une fois encore, des deux points « : ». Il se trouve au même niveau d'indentation que le **try**.
 - Le bloc d'instructions qui sera exécuté si une erreur est trouvée dans le premier bloc.

Les exceptions

Forme minimale du bloc try

Exemple:

Dans cet exemple, on demande à l'utilisateur de saisir l'année et on essaye de convertir l'année en un entier

```
annee = input() # On demande à l'utilisateur de saisir l'année
annee = int(annee) # On essaye de convertir l'année en un entier
```

```
annee = input()
try: # On essaye de convertir l'année en entier
    annee = int(annee)
except:
    print("Erreur lors de la conversion de l'année.")
```

Vous pouvez tester ce code en précisant plusieurs valeurs différentes pour la variable annee, comme « 2010 » ou « annee2010 ».

Les exceptions

Forme plus complète

- la forme minimale est à éviter pour plusieurs raisons.
- D'abord, elle ne différencie pas les exceptions qui pourront être levées dans le bloc **try**.
- Dans l'exemple précédent, on ne pense qu'à un type d'exceptions susceptible d'être levé : le type **ValueError**.
- **Autre exemple:**

```
try:  
    resultat = numerateur / denominateur  
except:  
    print("Une erreur est survenue... laquelle ?")
```

- **NameError** : l'une des variables **numérateur** ou **dénominateur** n'a pas été définie .
- **TypeError** : l'une des variables **numérateur** ou **dénominateur** ne peut diviser ou être divisée
- **ZeroDivisionError** : encore elle ! Si **dénominateur** vaut **0**, cette exception sera levée.



Les exceptions

Forme plus complète

Remarque: Tout se joue sur la ligne du **except**. Entre ce mot-clé et les deux points, vous pouvez préciser le type de l'exception que vous souhaitez traiter.

```
try:  
    resultat = numerateur / denominateur  
except NameError:  
    print("La variable numerateur ou denominateur n'a pas été  
        définie.")
```

Ce code ne traite que le cas où une exception **NameError** est levée. On peut intercepter les autres types d'exceptions en créant d'autres blocs **except** à la suite :

```
try:  
    resultat = numerateur / denominateur  
except NameError:  
    print("La variable numerateur ou denominateur n'a pas été  
        définie.")  
except TypeError:  
    print("La variable numerateur ou denominateur possède un  
        type incompatible avec la division.")  
except ZeroDivisionError:  
    print("La variable denominateur est égale à 0.")
```



Forme plus complète

On peut capturer l'exception et afficher son message grâce au mot-clé **as** qu'on a déjà vu dans un autre contexte (si si, rappelez-vous de l'importation de modules).

```
try:  
    # Bloc de test  
except type_de_l_exception as exception_retournee:  
    print("Voici l'erreur :", exception_retournee)
```

Dans ce cas, une variable **exception_retournee** est créée par Python si une exception du type précisé est levée dans le bloc **try**.

Les exceptions

Les mots-clés **else** et **finally**

Ce sont deux mots-clés qui vont nous permettre de construire un bloc try plus complet.

■ Le mot-clé **else**

Dans un bloc **try**, **else** va permettre d'exécuter une action si aucune erreur ne survient dans le bloc (c-à-d aucune exception n'a été lancée dans le **try**).

```
try:  
    resultat = numerateur / denominateur  
except NameError:  
    print("La variable numerateur ou denominateur n'a pas été  
         définie.")  
except TypeError:  
    print("La variable numerateur ou denominateur possède un  
         type incompatible avec la division.")  
except ZeroDivisionError:  
    print("La variable denominateur est égale à 0.")  
else:  
    print("Le résultat obtenu est", resultat)
```

Les exceptions

Les mots-clés **else** et **finally**

- **Le mot-clé finally**

finally permet d'exécuter du code après un bloc **try**, quelle que soit le résultat de l'exécution dudit bloc.

```
try:  
    # Test d'instruction(s)  
except TypeDInstruction:  
    # Traitement en cas d'erreur  
finally:  
    # Instruction(s) exécutée(s) qu'il y ait eu des erreurs ou non
```

Le bloc **finally** est exécuté dans tous les cas de figures. Quand bien même Python trouverait une instruction **return** dans votre bloc **except** par exemple, il exécutera le bloc **finally**.



Les exceptions

Les mots-clés `else` et `finally`

■ Exemple:

```
def divide(x,y):  
    try:  
        result = x / y  
    except ZeroDivisionError:  
        print("division by zero!")  
    else:  
        print("result is", result)  
    finally:  
        print ("executing finally clause")
```

>>> *divide(1,3)*

result is 0.3333333333333333

executing finally clause

>>> *divide(1,0)*

division by zero!

executing finally clause

Les exceptions

Définition

- Les erreurs détectées durant l'exécution sont appelées des **exceptions**. Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution.
- Python lève donc des **exceptions** quand il trouve une erreur, soit dans le code (une erreur de syntaxe, par exemple), soit dans l'opération que vous lui demandez de faire.

```
# Exemple classique : test d'une division par zéro
>>> variable = 1/0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

- **ZeroDivisionError**: le type de l'exception ;
- **int division or modulo by zero**: le message qu'envoie Python pour vous aider à comprendre l'erreur qui vient de se produire.

Les exceptions

Forme minimale du bloc **try**

On va parler ici de bloc **try**. Nous allons en effet mettre les instructions que nous souhaitons tester dans un premier bloc et les instructions à exécuter en cas d'erreur dans un autre bloc

try:

Bloc à essayer

except:

Bloc qui sera exécuté en cas d'erreur

Dans l'ordre, nous trouvons :

- Le mot-clé **try** suivi des deux points « : » (**try** signifie « essayer » en anglais).
 - Le bloc d'instructions à essayer.
- Le mot-clé **except** suivi, une fois encore, des deux points « : ». Il se trouve au même niveau d'indentation que le **try**.
 - Le bloc d'instructions qui sera exécuté si une erreur est trouvée dans le premier bloc.

Les exceptions

Forme minimale du bloc try

Exemple:

Dans cet exemple, on demande à l'utilisateur de saisir l'année et on essaye de convertir l'année en un entier

```
année = input() # On demande à l'utilisateur de saisir l'année
année = int(année) # On essaye de convertir l'année en un entier
```

```
année = input()
try: # On essaye de convertir l'année en entier
    année = int(année)
except:
    print("Erreur lors de la conversion de l'année.")
```

Vous pouvez tester ce code en précisant plusieurs valeurs différentes pour la variable année, comme « 2010 » ou « année2010 ».

Les exceptions

Forme plus complète

- la forme minimale est à éviter pour plusieurs raisons.
- D'abord, elle ne différencie pas les exceptions qui pourront être levées dans le bloc **try**.
- Dans l'exemple précédent, on ne pense qu'à un type d'exceptions susceptible d'être levé : le type **ValueError**.
- **Autre exemple:**

```
try:  
    resultat = numerateur / denominateur  
except:  
    print("Une erreur est survenue... laquelle ?")
```

- **NameError** : l'une des variables **numérateur** ou **dénominateur** n'a pas été définie .
- **TypeError** : l'une des variables **numérateur** ou **dénominateur** ne peut diviser ou être divisée
- **ZeroDivisionError** : encore elle ! Si **dénominateur** vaut **0**, cette exception sera levée.

Les exceptions

Forme plus complète

Remarque: Tout se joue sur la ligne du **except**. Entre ce mot-clé et les deux points, vous pouvez préciser le type de l'exception que vous souhaitez traiter.

```
try:  
    resultat = numerateur / denominateur  
except NameError:  
    print("La variable numerateur ou denominateur n'a pas été  
        définie.")
```

Ce code ne traite que le cas où une exception **NameError** est levée. On peut intercepter les autres types d'exceptions en créant d'autres blocs **except** à la suite :

```
try:  
    resultat = numerateur / denominateur  
except NameError:  
    print("La variable numerateur ou denominateur n'a pas été  
        définie.")  
except TypeError:  
    print("La variable numerateur ou denominateur possède un  
        type incompatible avec la division.")  
except ZeroDivisionError:  
    print("La variable denominateur est égale à 0.")
```

Les exceptions

Forme plus complète

On peut capturer l'exception et afficher son message grâce au mot-clé **as** qu'on a déjà vu dans un autre contexte (si si, rappelez-vous de l'importation de modules).

```
try:  
    # Bloc de test  
except type_de_l_exception as exception_retournée:  
    print("Voici l'erreur :", exception_retournée)
```

Dans ce cas, une variable **exception_retournée** est créée par Python si une exception du type précisé est levée dans le bloc **try**.

Les exceptions

Les mots-clés **else** et **finally**

Ce sont deux mots-clés qui vont nous permettre de construire un bloc try plus complet.

■ Le mot-clé **else**

Dans un bloc **try**, **else** va permettre d'exécuter une action si aucune erreur ne survient dans le bloc (c-à-d aucune exception n'a été lancée dans le **try**).

```
try:  
    resultat = numerateur / denominateur  
except NameError:  
    print("La variable numerateur ou denominateur n'a pas été  
         définie.")  
except TypeError:  
    print("La variable numerateur ou denominateur possède un  
         type incompatible avec la division.")  
except ZeroDivisionError:  
    print("La variable denominateur est égale à 0.")  
else:  
    print("Le résultat obtenu est", resultat)
```

Les exceptions

Les mots-clés **else** et **finally**

- **Le mot-clé finally**

finally permet d'exécuter du code après un bloc **try**, quelle que soit le résultat de l'exécution dudit bloc.

```
try:  
    # Test d'instruction(s)  
except TypeDInstruction:  
    # Traitement en cas d'erreur  
finally:  
    # Instruction(s) exécutée(s) qu'il y ait eu des erreurs ou non
```

Le bloc **finally** est exécuté dans tous les cas de figures. Quand bien même Python trouverait une instruction **return** dans votre bloc **except** par exemple, il exécutera le bloc **finally**.

Les exceptions

Les mots-clés `else` et `finally`

- **Exemple:**

```
def divide(x,y):  
    try:  
        result = x / y  
    except ZeroDivisionError:  
        print("division by zero!")  
    else:  
        print("result is", result)  
    finally:  
        print ("executing finally clause")
```

>>> *divide(1,3)*

result is 0.3333333333333333

executing finally clause

>>> *divide(1,0)*

division by zero!

executing finally clause

Les modules

Introduction

Définition :

- Un module est un fichier contenant des sous-programmes regroupés de façon cohérente
- Le nom du fichier aura comme suffixe .py

Structure d'un projet:

Plusieurs modules de fonctions et un module particulier, dit “**main**”, où l'on définit les variables du projet et où l'on utilise les fonctions des différents modules.

Trois formats pour importer

- **from NomDuModule import ***
- **from NomDuModule import uneFonction**
- **import NomDuModule**

Les modules import ...

import nomModule

- Importe tout le contenu du fichier `nomModule.py`
- Pour faire référence à quelque chose dans le fichier il faut ajouter le nom du module avant le nom de l'entité utilisée.

Exemple :

```
import fib
fib.fib(n)
fib.fib_rec(n)
print(fib.iter(n))
```

Attention tout le contenu du module sera chargé en mémoire.

Les modules

from ... import *

from nomModule import *

- Tout le contenu du fichier nomModule.py est importé et chargé en mémoire
- Tout le module est dans le namespace courant, il est donc inutile de faire préfixé toutes les entités présentes dans le fichier par le nom du module

Exemple:

```
from fib import *
fib(n)
fib_rec(n)
```

Les modules

from ... import ...

from nomModule import item

- Seul l'élément item est importé du fichier nomModule.py.
- Comme dans le cas précédent il n'est pas nécessaire de faire préfixer le nom de cet item par le nom du module
- Attention à l'écrasement d'une entité de même nom déjà présente dans le namespace courant.

Exemple :

```
from fib import fib  
fib(n)
```

Les modules

Exemple

```
# le module exemple  
  
def doubler(x):  
    return 2*x  
  
def tripler(x):  
    return 3*x
```

```
from exemple import *  
  
print(tripler(12))
```

36

```
from exemple import doubler  
  
print(doubler(12))
```

24

```
import exemple  
  
print(exemple.tripler(12))
```

36

Les modules

Liste non exhaustive de modules Python

time	diverses fonctions relatives au temps
random	pour générer aléatoirement des nombres.
os	pour dialoguer avec le système d'exploitation.
Tkinter	pour créer les fenêtres graphiques et gérer la souris.
math	pour accéder aux fonctions mathématiques usuelles.
matplotlib	pour de multiples fonctions de tracés en deux dimensions.
numpy	bibliothèque de fonctions indispensables pour faire du calcul scientifique efficace.
scipy	bibliothèque complémentaire de numpy pour le traitement de données.
skimage	pour le traitement d'images.
io	pour lire et créer des fichiers.

Les modules

Le module numpy

Fonction	Rôle	Syntaxe	Exemple
array	Pour créer des tableaux à partir de listes	np.array (liste)	» np.array([1, 1, 1]) array([1, 1, 1])
arange	Comme range. début, fin et pas peuvent être des réels	np.arange(debut, fin, pas) N.B. la valeur fin est exclue	» np.arange(1.5,3.6,0.5) array([1.5, 2. , 2.5, 3. , 3.5])
linspace	Les n éléments du tableau sont régulièrement espacés entre début et fin	np.linspace(debut, fin, n) N.B. la valeur fin est inclue	» np.linspace(0,1,6) array([0., 0.2, 0.4, 0.6, 0.8, 1.])
empty	Pour créer des tableaux sans initialisation	np.empty(n,type) => 1 dim. np.empty((nl,nc),type)=>2 dim.	» X=np.empty(5,'float') X est un tableau non initialisé
zeros	Pour créer des tableaux remplis de 0	np. zeros (n,type) => 1 dim. np. zeros((nl,nc),type)=>2 dim.	» np.zeros(3,'float') array([[0., 0., 0.]])
ones	Pour créer des tableaux remplis de 1	np.ones(n,type) => 1 dim. np.ones((nl,nc),type)=>2 dim.	» np.ones(3,'float') array([[1., 1., 1.]])
identity ou eye	Pour créer des matrices identités	np.identity(nombre) np.eye(nombre)	» np.identity(2) array([[1., 0.], [0., 1.]])
alen	donne la première dimension d'un tableau (taille d'un vecteur, nbr de lignes d'une matrice).	I=np.alen(tab)	» T=np.array([0, 1, 2, 3, 4, 5]) » np.alen(T) =>6 » M=np.array([[1, 9, 1],[3,6,4]]) » np.alen(M) =>2
size	Pour obtenir le nombre d'éléments d'un tableau	np.size(tab) np.size(tab,0) : nbr de lignes np.size(tab,1) :nbr de colonnes	» A=np.array([[1, 9, 1],[3,6,4]]) » np.size(A) =>6 » np.size(A,0) =>2 » np.size(A,1) =>3

Les modules

Le module numpy

ndim	Pour obtenir le nombre de dimensions d'un tab	np.ndim(tab)	» A=np.array([[1, 9, 1],[3,6,4]]) » np.ndim(A) →2
shape	Pour obtenir la dimension d'un tableau	np.shape(tab) N.B. le résultat est un tuple	» T= np.array([1,9,1,3,6,4]) » np.shape(T) → (6,)
reshape	Pour redimensionner un tableau. N.B. reshape ne modifie pas le tableau. Il faut récupérer le résultat	T=np.reshape(tab,(nl,nc)) ou bien T=tab.reshape((nl,nc))	» T1=np.array([0, 1, 2, 3, 4, 5]) » T2=np.reshape(T1,(2,3)) » T2 array([[0, 1, 2], [3, 4, 5,]])
cos sin tan log exp ...	Fonctions mathématiques applicables à des vecteurs. Le résultat est un vecteur contenant les images de chaque élément du vecteur.	np.cos(tab) np.sin(tab) np.tan(tab) np.log(tab) np.exp(tab) ...	» A=np.array([1, 9, 1, 3,6,4]) » np.log(A) array([0. , 2.197224, 0. , 1.098612, 1.791759, 1.386294])
sum min max mean	la somme, le minimum, le maximum et la moyenne du tableau	np.sum(tab) np.min(tab) np.max(tab) np.mean(tab)	» A=np.array([[1, 9, 1],[3,6,4]]) » np.mean(A) 4.0
transpose	Pour calculer la transposé d'un tableau	np.transpose(tab) ou bien tab.T	» A=np.array([[1,3],[9, 6],[1, 4]]) » np.transpose(A) array([[1, 9, 1],[3,6,4]])
dot	Pour effectuer le produit matriciel de deux tableaux	np.dot(tab1,tab2)	» A=np.array([[1,9,1],[3,6,4]]) » B=np.array([[1,1],[3, 4],[3, 4]]) » P=np.dot(A,B) P=array([[31, 41],[33, 43]])
vdot	Pour effectuer le produit scalaire de deux vecteurs	np.vdot(tab1,tab2)	» A=np.array([1, 9, 1, 3, 6, 4]) » B=np.array([1, 1, 3, 4 , 3, 4]) » np.vdot(A , B) →59

Les modules

Le sous-module `numpy.linalg` pour l'algèbre linéaire

Fonction	Rôle	Syntaxe	Exemple
inv	Calcule l'inverse d'une matrice carrée	<code>inv= np.linalg.inv(mat)</code>	<pre>» A=np.array([[1, 9],[3,4]])</pre> <pre>» np.linalg.inv(A)</pre> <pre>array([-0.173913, 0.391304],</pre> <pre>[0.130434, -0.043478]))</pre>
det	Calcule le déterminant d'une matrice carrée	<code>d=np.linalg.det(mat)</code>	<pre>» A=np.array([[1, 9],[3,4]])</pre> <pre>» np.linalg.det(A)</pre> <pre>-23.0</pre>
solve	Solution du système $A^*X=b$	<code>x= np.linalg.solve (A,b)</code>	<pre>» A=[[2,1,-1],[1,-1,1],[1,1,-2]]</pre> <pre>» B=[1,2,-3]</pre> <pre>» np.linalg.solve (A,B)</pre> <pre>array([1., 2., 3.])</pre>
eig	Pour calculer les Valeurs et les vecteurs propres d'une matrice carrée	<code>x,y=np.linalg.eig(mat)</code> x : vecteur des valeurs propres y : matrice des vecteurs propres	<pre>» A=np.array([[1, 9],[3,4]])</pre> <pre>» x,y=np.linalg.eig(A)</pre> <pre>x=array([-2.90832691,</pre> <pre>7.90832691])</pre> <pre>y=array([-0.91724574,</pre> <pre>0.79325185],</pre> <pre>[0.3983218 , -0.60889368]))</pre>

Les modules

Le module `scipy.integrate` pour les techniques d'intégration

Fonction	Rôle	Syntaxe	Exemple
odeint	Effectue l'intégration numérique d'une équation différentielle définie par : $dy/dt=F(y,t)$ et $y(t_0)=y_0$	S= scipy.integrate.odeint(F,y0,t) F : est la fonction définissant l'équation différentielle, y0 : est la donnée initiale, t : un vecteur des valeurs de t	<pre>import matplotlib.pyplot as plt from scipy.integrate import odeint def F(y,t) : return t**2+sin(y)*y**2 t = linspace(0,20,1000) y0 = 3 # y(t0=0) = y0 solution = odeint(F,y0,t) plt.plot(t, solution[:,0]) plt.show()</pre>
quad	Calcule l'intégrale d'une <u>fonction</u> entre a et b et l'erreur correspondante.	x,y= <code>scipy.integrate.quad(f,a,b)</code> f : la fonction à intégrer x : c'est la valeur de l'intégrale y : c'est l'erreur de calcul	» <code>i,j=scipy.integrate.quad(sin,1,4)</code> 1.19394592673 2.0888914827e-14
romberg	Calcule l'intégrale d'une <u>fonction</u> entre a et b avec la méthode de Romberg.	x= <code>scipy.integrate.romberg(f,a,b)</code> f : la fonction à intégrer x : c'est la valeur de l'intégrale	» <code>scipy.integrate.romberg(sin,1,4)</code> 1.1939459267321892
simps trapz	Calcule l'intégrale d'un <u>tableau de valeurs</u> passé en paramètre avec les abscisses de ces valeurs. Les méthodes utilisées sont respectivement : Simpson et trapèzes.	x= <code>scipy.integrate.simps(t1,t2)</code> x= <code>scipy.integrate.trapz(t1,t2)</code> t2 : tableau des abscisses t1 : tableau de valeurs en les points de t2	def f(x): return 1/(1+x**2) x = numpy.linspace(0,20,100) v=scipy.integrate.trapz(f(x), x)

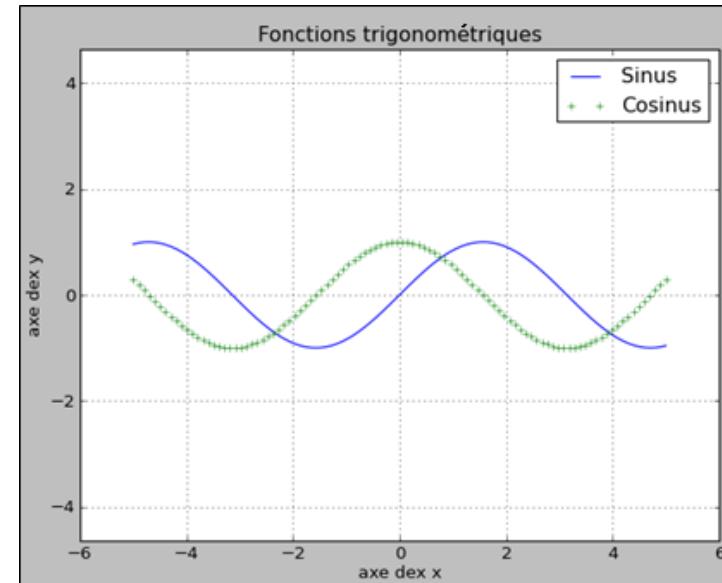
Les modules

Le module `matplotlib.pyplot` pour tracer des courbes

❖ Principales fonctions

<code>plot</code>	pour tracer une courbe à partir d'un tableau de valeurs
<code>show</code>	pour ouvrir une fenêtre et afficher l'image créée
<code>axis</code>	pour préciser la nature des axes
<code>xlabel, ylabel</code>	pour donner un nom aux axes
<code>grid</code>	pour ajouter une grille
<code>legend</code>	pour ajouter une légende
<code>savefig</code>	pour enregistrer la figure dans un fichier image

Exemple
d'utilisation de
`matplotlib.pyplot`



Les modules

Le module **matplotlib.pyplot** pour tracer des courbes

Exemple d'utilisation de **matplotlib.pyplot**

```
import matplotlib.pyplot as plt
import numpy as np

x=np.linspace(-5,5,100)
y1=np.sin(x)
y2=np.cos(x)
plt.grid(True)                      #Affichage de la grille
plt.plot(x,y1,'b',x,y2,'g+')      #tracer les courbes
plt.legend(("Sinus", "Cosinus"))    #afficher la légende
plt.xlabel('axe des x')             #étiquette sur l'axe des x
plt.ylabel('axe des y')             #étiquette sur l'axe des y
plt.axis('equal')                  #choix d'échelle
plt.title("Fonctions trigonométriques") #titre
plt.savefig('Premier exemple')     #on sauvegarde l'image
plt.show()                          #on montre le résultat
```

Les modules

Le module **matplotlib.pyplot** pour tracer des courbes

Tables des Styles de traçage

Caractère de traçage	description
'-'	solid line
'--'	dashed line
'-. '	dash-dot line
'.'	Dotted line
'.'	Point marker
', '	pixel marker
'o'	Circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
's'	Square marker
'p'	Pentagon marker

Les modules

Le module `matplotlib.pyplot` pour tracer des courbes

❖ Tables des Styles de traçage

Caractère de traçage	description
'1'	tri_down marker
'2'	tri_up marker
'3'	tri_left marker
'4'	tri_right marker
'H'	hexagon2 marker
'+'	Plus marker
'x'	X marker
'D'	diamond marker
'd'	thin_diamond marker
' '	Vline marker
'_'	Hline marker
'*''	Star marker
'h'	hexagon1 marker

❖ Table des couleurs

Caractère	Couleur
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

Bases de données SQLITE en Python

Introduction

- Dans le joyeux monde de la programmation, il est souvent nécessaire de stocker des informations.
- À petite comme à grande échelle, les **Bases de Données** (BD) s'imposent comme une forme efficace de stockage. Il est alors plutôt aisés d'interagir avec celles-ci en utilisant un **Système de Gestion de Base de Données** (SGBD).
- Parmi les **SGBD**, nous pouvons trouver **SQLite** qui utilise un sous-ensemble de SQL. Sa légèreté et le fait que les données se trouvent sur le terminal du client et non sur un serveur distant.
- **SQLite** fait partie de la famille des SGBD dits « Relationnelles », car les données sont alors placées dans des tables et traitées comme des ensembles.
- En **Python**, le module **SQLite3** permet de travailler avec ce moteur, mais ne supporte pas le multi-thread

Bases de données SQLITE en Python

Se connecter et se déconnecter

- Avant de commencer, il convient d'importer le module, comme il est coutume de faire avec Python :

```
import sqlite3
```

❖ Connexion

- Cela fait, nous pouvons nous connecter à une BDD en utilisant la méthode `connect` et en lui passant l'emplacement du fichier de stockage en paramètre. Si ce dernier n'existe pas, il est alors créé:

```
conn = sqlite3.connect("ma_base.sqlite")
```

- Comme vous pouvez le voir, nous récupérons un objet retourné par la fonction. Celui-ci est de type `Connection` et il nous permettra de travailler sur la base.
- Par ailleurs, il est aussi possible de stocker la BDD directement dans la RAM en utilisant la chaîne clef `:memory:`. Dans ce cas, il n'y aura donc pas de persistance des données après la déconnexion.

```
conn = sqlite3.connect(":memory:")
```

Bases de données SQLITE en Python

Se connecter et se déconnecter

❖ Déconnexion

- Que nous soyons connectés avec la RAM ou non, il ne faut pas oublier de nous déconnecter. Pour cela, il nous suffit de faire appel à la méthode **close** de notre objet **Connection**

```
conn.close()
```

❖ Les types de champ

- Comme nous allons bientôt voir comment exécuter des requêtes, il est important de connaître les types disponibles, avec leur correspondance en Python. Voici ci-dessous, un tableau récapitulatif :

SQLite	Python
NULL	None
INTEGER	int
REAL	float
TEXT	str par défaut
BLOB	bytes

Bases de données SQLITE en Python

Exécution des requêtes

- Pour exécuter nos requêtes, nous allons nous servir d'un objet **Cursor**, récupéré en faisant appel à la méthode **cursor** de notre objet de type **Connection**.

```
curseur = conn.cursor()          #Récupération d'un curseur
```

❖ Validation ou annulation des modifications

- Lorsque nous effectuons des modifications sur une table (insertion, modification ou encore suppression d'éléments), celles-ci ne sont pas automatiquement validées. Ainsi, sans validation, les modifications ne sont pas effectuées dans la base et ne sont donc pas visibles par les autres connexions.
- Pour résoudre cela, il nous faut donc utiliser la méthode **commit** de notre objet de type **Connection**. En outre, si nous effectuons des modifications puis nous souhaitons finalement revenir à l'état du dernier **commit**, il suffit de faire appel à la méthode **rollback**, toujours de notre objet de type **Connection**.

```
conn.commit()                  #Validation des modifications
```

```
conn.rollback()                #Retour à l'état du dernier commit
```

Bases de données SQLITE en Python

Exécution des requêtes

❖ Exécuter une requête

- Pour exécuter une requête il suffit de passer celle-ci à la méthode `execute`. Voici un exemple de création de table:

```
curseur = conn.cursor()
curseur.execute(" " "CREATE TABLE IF NOT EXISTS users(
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    name TEXT,
    age INTEGER) " " ")
conn.commit()
```

❖ Exécuter plusieurs requêtes

- Pour exécuter plusieurs requêtes, comme pour ajouter des éléments à une table par exemple, nous pouvons faire appel plusieurs fois à la méthode `execute`

```
donnees = [("Ali", 20), ("Aziz", 40), ("Khalid", 55)
for donnee in donnees:
    curseur.execute(' ' 'INSERT INTO users (name, age)
VALUES (?, ?)' ' ', donnee)
conn.commit()
```

Bases de données SQLITE en Python

Exécution des requêtes

❖ Insérer des données

- Il existe plusieurs manières d'insérer des données, la plus simple étant celle-ci:

```
curseur.execute(" " "INSERT INTO users(name, age)  
VALUES(?, ?)" " ", ("olivier", 30))
```

- Vous pouvez passer par un dictionnaire:

```
data = {"name": "Walid", "age": 30}  
curseur.execute("INSERT INTO users(name, age)  
VALUES (:name, :age)", data)
```

- Vous pouvez récupérer l'id de la ligne que vous venez d'insérer de cette manière:

```
print('dernier id:', curseur.lastrowid)
```

- Il est également possible de faire plusieurs `insert` en une seule fois avec la fonction `executemany`:

```
users = [("Ahmed", 30), ("Mohamed", 90)]  
curseur.executemany("INSERT INTO users(name, age)  
VALUES(?, ?)", users)
```

Bases de données SQLITE en Python

Exécution des requêtes

❖ Modifier des enregistrements

- Pour modifier des enregistrements:

```
curseur.execute(" " "UPDATE users SET age = ?  
                 WHERE id = 2" " ", (31,))
```

❖ Supprimer une table avec SQLite

```
curseur = conn.cursor()  
curseur.execute(" " "DROP TABLE users" " ")  
conn.commit()
```

❖ Connaître le nombre de modifications depuis le dernier commit

- Ensuite, pour être au courant du nombre de modifications (ajouts, mises à jour ou suppressions) apportées depuis notre connexion à la base, il suffit de récupérer la valeur de l'attribut `total_changes` de notre objet de type `Connection`

```
print(conn.total_changes)
```

❖ Connaitre le nombre de lignes impactées par une exécution

- ❖ Pour connaître le nombre de lignes impactées par une exécution, il suffit d'utiliser l'attribut `rowcount` de notre objet de type `Cursor`.

```
print(curseur.rowcount) #-1, aucune exécution
```

Bases de données SQLITE en Python

Exécution d'un script

- Enfin, il est aussi possible d'exécuter un script directement à l'aide de la méthode `executescript` de notre objet de type `Cursor`. Si celui-ci contient plusieurs requêtes, celles-ci doivent être séparées par des points virgules.

```
curseur.executescript("""  
    DROP TABLE IF EXISTS users;  
    CREATE TABLE IF NOT EXISTS users(  
        id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,  
        name TEXT,  
        age INTEGER);  
    INSERT INTO users(name, age) VALUES ("Ali", 20);  
    INSERT INTO users(name, age) VALUES ("Aziz", 40);  
    INSERT INTO users(name, age) VALUES ("Khalid", 55)  
    """)  
conn.commit()
```