

Introduction à la programmation CR1. Introduction

Abdelmalek TOUMI

`toumiab@ensta-bretagne.fr`

2020/2021

ENSTA Bretagne



Sommaire

1 Introduction

Déroulement de cours
Langage Python

2 Principes généraux

Généralités : variables et données
Les types de base
Exercices

3 Les Types de Python

Rappel : types simples/composites
Transtypage et opérations

4 Instructions du langage Python - Rappel

Commentaires
Littéraux (constantes)
Branchements conditionnels et boucles



Sommaire

1 Introduction

Déroulement de cours

Langage Python

2 Principes généraux

Généralités : variables et données

Les types de base

Exercices

3 Les Types de Python

Rappel : types simples/composites

Transtypage et opérations

4 Instructions du langage Python - Rappel

Commentaires

Littéraux (constantes)

Branchements conditionnels et boucles



Déroulement

- ~~Intro linux~~ 2 séances SI pour configurer votre machine
- Les bases de Python + Introduction à la POO
- UE 2.1 : POO objet et suite de l'algorithmique
- —> UE2.4 : Projet informatique : deux parties (sans et avec une interface graphique)



Déroulement

- ~~Intro linux~~ 2 séances SI pour configurer votre machine
- Les bases de Python + Introduction à la POO
- UE 2.1 : POO objet et suite de l'algorithmique
- —→ UE2.4 : Projet informatique : deux parties (sans et avec une interface graphique)
- Programmation impérative et objet
- Notions d'architecture logicielle



Déroulement

- ~~Intro linux~~ 2 séances SI pour configurer votre machine
- Les bases de Python + Introduction à la POO
- UE 2.1 : POO objet et suite de l'algorithmique
- —> UE2.4 : Projet informatique : deux parties (sans et avec une interface graphique)
- Programmation impérative et objet
- Notions d'architecture logicielle
- Le cours : CM (55 min)
- Exercices et utilisation des cartons (vert/rouge) pour les exercices Q/R

Plateforme : <https://moodle.ensta-bretagne.fr>



Sommaire

1 Introduction

Déroulement de cours

Langage Python

2 Principes généraux

Généralités : variables et données

Les types de base

Exercices

3 Les Types de Python

Rappel : types simples/composites

Transtypage et opérations

4 Instructions du langage Python - Rappel

Commentaires

Littéraux (constantes)

Branchements conditionnels et boucles



Histoire des langages

- Début : langage machine
- Assembleur en 1950
- Fortran en 1954
- Matlab 1970
- C en 1971
- Python en 1994 (première idée en 1990)
- Java en 1995

Historique : [http ://www.levenez.com](http://www.levenez.com)



Histoire des langages

- Début : langage machine
- Assembleur en 1950
- Fortran en 1954
- Matlab 1970
- C en 1971
- Python en 1994 (première idée en 1990)
- Java en 1995

Historique : <http://www.levenez.com>

Langage choisi : Python (version 3.6.1)



Langage Python

- Python est interprété



Langage Python

- Python est interprété
- Python est orienté objet et permet une programmation impérative



Langage Python

- Python est interprété
- Python est orienté objet et permet une programmation impérative
- Python est fortement typé



Langage Python

- Python est interprété
- Python est orienté objet et permet une programmation impérative
- Python est fortement typé
- Python est typé dynamiquement



Langage Python

- Python est interprété
- Python est orienté objet et permet une programmation impérative
- Python est fortement typé
- Python est typé dynamiquement
- Python assure la gestion de la mémoire



Langage Python

- Python est interprété
- Python est orienté objet et permet une programmation impérative
- Python est fortement typé
- Python est typé dynamiquement
- Python assure la gestion de la mémoire
- Python est multitâche



Langage Python

- Python est interprété
- Python est orienté objet et permet une programmation impérative
- Python est fortement typé
- Python est typé dynamiquement
- Python assure la gestion de la mémoire
- Python est multitâche
- Python interagit avec les programmes des autres langages



Langage Python

- Python est interprété
- Python est orienté objet et permet une programmation impérative
- Python est fortement typé
- Python est typé dynamiquement
- Python assure la gestion de la mémoire
- Python est multitâche
- Python interagit avec les programmes des autres langages
- Python est indépendant de toute plate-forme

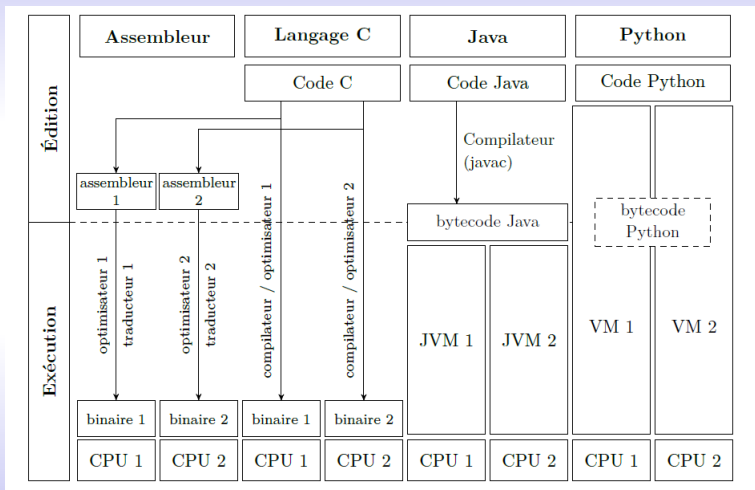


Langage Python

- Python est interprété
- Python est orienté objet et permet une programmation impérative
- Python est fortement typé
- Python est typé dynamiquement
- Python assure la gestion de la mémoire
- Python est multitâche
- Python interagit avec les programmes des autres langages
- Python est indépendant de toute plate-forme
- Python est agréable et facile à lire



Langage Python





Exécution d'un programme Python

Remarque (Un programme Python)

Un programme est une suite d'instructions Python qui peut être exécuté de différentes manières :

- **En mode interactif**
- **En script**



Exécution d'un programme Python

- En mode interactif :

- avec l'**interpréteur python** : via un terminal de commandes lancé avec la commande 'python3' ou via le **shell python**
 - Les instructions python à insérer après l'invite de commande (`>>>`), ou à l'invite de poursuit (...)
 - Les instructions sont exécutées immédiatement
 - Le résultat d'une instruction est affiché sur le terminal

```
Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
Python 3.6.1 (v3.6.1:37a07cee5969, Dec 6 2016, 01:54:25) [MSC v.1900 64 bit (AMD
64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("bonjour")
bonjour
>>>
```

GUI: OFF (TK) Lrn: 5 Col: 4



Exécution d'un programme Python

- **En mode interactif :**

- avec l'**interpréteur python** : via un terminal de commandes lancé avec la commande 'python3' ou via le **shell python**
 - Les instructions python à insérer après l'invite de commande (>>>), ou à l'invite de poursuit (...)
 - Les instructions sont exécutées immédiatement
 - Le résultat d'une instruction est affiché sur le terminal
- avec **ipython** : est une sourcouche de l'interpréteur Python offrant une plusieurs facilités. En version web : **IPython Notebook**

```
Jupyter QtConsole
File Edit View Kernel Window Help

Jupyter QtConsole 4.2.1
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 5.3.0 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

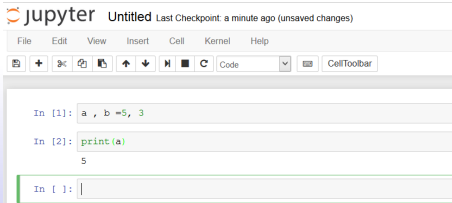
In [1]: 3+4
Out[1]: 7
```



Exécution d'un programme Python

- En mode interactif :

- avec l'**interpréteur python** : via un terminal de commandes lancé avec la commande 'python3' ou via le **shell python**
 - Les instructions python à insérer après l'invite de commande (`>>>`), ou à l'invite de poursuit (`...`)
 - Les instructions sont exécutées immédiatement
 - Le résultat d'une instruction est affiché sur le terminal
- avec **ipython** : est une sourcouche de l'interpréteur Python offrant une plusieurs facilités. En version web : **IPython Notebook**





Exécution d'un programme Python

- En mode interactif :

- avec l'**interpréteur python** : via un terminal de commandes lancé avec la commande 'python3' ou via le **shell python**
 - Les instructions python à insérer après l'invite de commande (>>>), ou à l'invite de poursuite (...)
 - Les instructions sont exécutées immédiatement
 - Le résultat d'une instruction est affiché sur le terminal
- avec **ipython** : est une sourcouche de l'interpréteur Python offrant une plusieurs facilités. En version web : **IPython Notebook**
- avec **bpython** : analogue à ipython en plus léger et plus commode, il propose une coloration syntaxique sur la ligne de commande

```
>>> from decimal import *
>>> for x in xrange(10):
    xrange: ([start, ] stop[, step])
    xrange([start,] stop[, step]) -> xrange object
    Like range(), but instead of returning a list, returns an object that
    generates the numbers in the range on demand. For looping, this is
    slightly faster than range() and more memory efficient.
```




Exécution d'un programme Python

- **En mode script** (Programming mode) :
 - Écriture du programme Python (éditeur de texte) dans un fichier portant l'extension **.py**



Exécution d'un programme Python

- **En mode script** (Programming mode) :
 - Écriture du programme Python (éditeur de texte) dans un fichier portant l'extension **.py**
 - Exécution
 - \$ `python monprogramme.py` # pour windows
 - \$ `python3.7 monprogramme.py` # pour linux



Installation d'un environnement Python

- Sous Windows, Python n'est pas installé par défaut.



Installation d'un environnement Python

- Sous Windows, Python n'est pas installé par défaut.
- Sous GNU/Linux et Mac OS X, Python et la librairie standard sont intégrés



Installation d'un environnement Python

- Sous Windows, Python n'est pas installé par défaut.
- Sous GNU/Linux et Mac OS X, Python et la librairie standard sont intégrés
 - depuis Ubuntu 20.XX on dispose en parallèle de Python v2 et v3.7
 - Mac OS X 10.8, 10.9 et 10.10, Apple propose que Python 2.7



Installation d'un environnement Python

- Sous Windows, Python n'est pas installé par défaut.
- Sous GNU/Linux et Mac OS X, Python et la librairie standard sont intégrés
 - depuis Ubuntu 20.XX on dispose en parallèle de Python v2 et v3.7
 - Mac OS X 10.8, 10.9 et 10.10, Apple propose que Python 2.7
- Installation :
 - Windows : **WinPython 3.7.1.6Qt5** distribution spécifique à Windows et très facile à mettre en œuvre. Il intègre notamment : IPython, Spyder, NumPy, SciPy, Matplotlib, Pandas, SymPy, PIP...
 - + → <http://www.lfd.uci.edu/~gohlke/pythonlibs/>



Installation d'un environnement Python

- Sous Windows, Python n'est pas installé par défaut.
- Sous GNU/Linux et Mac OS X, Python et la librairie standard sont intégrés
 - depuis Ubuntu 20.XX on dispose en parallèle de Python v2 et v3.7
 - Mac OS X 10.8, 10.9 et 10.10, Apple propose que Python 2.7
- Installation :
 - Windows : **WinPython 3.7.1.6Qt5** distribution spécifique à Windows et très facile à mettre en œuvre. Il intègre notamment : IPython, Spyder, NumPy, SciPy, Matplotlib, Pandas, SymPy, PIP...
 - + \rightarrow <http://www.lfd.uci.edu/~gohlke/pythonlibs/>
 - GNU/Linux Ubuntu ≥ 16.04 : installation standard via les dépôts officiels Ubuntu de Canonical
 - Mac OS X : **Anaconda** distribution Python multiplateforme très répandue dans les milieux scientifiques



Installation d'un environnement Python

- Sous Windows, Python n'est pas installé par défaut.
- Sous GNU/Linux et Mac OS X, Python et la librairie standard sont intégrés
 - depuis Ubuntu 20.XX on dispose en parallèle de Python v2 et v3.7
 - Mac OS X 10.8, 10.9 et 10.10, Apple propose que Python 2.7
- Installation :
 - Windows : **WinPython 3.7.1.6Qt5** distribution spécifique à Windows et très facile à mettre en œuvre. Il intègre notamment : IPython, Spyder, NumPy, SciPy, Matplotlib, Pandas, SymPy, PIP...
 - + \rightarrow <http://www.lfd.uci.edu/~gohlke/pythonlibs/>
 - GNU/Linux Ubuntu ≥ 16.04 : installation standard via les dépôts officiels Ubuntu de Canonical
 - Mac OS X : **Anaconda** distribution Python multiplateforme très répandue dans les milieux scientifiques
 - IDE : **Pycharm pro** (nécessite une inscription - licence) : votre email école



Environnement de développement

- bloc-note, notepad++, vim, emacs ;
- IDE (Integrated Development Environment) : <https://wiki.python.org/moin/IntegratedDevelopmentEnvironments>
 - Netbeans (Sun) avec Python/jPython
 - Eclipse (IBM) avec le plugin PyDev
 - **Spyder**
 - Komodo IDE
 - LiCipse avec PyDev
 - **pyCharm pro**
 -



Python et le calcul scientifique

- rapide à apprendre (mais long à maîtriser) ;
- alternative (très) sérieuse à Matlab, Scilab, Octave ;
- bibliothèques de calcul et de visualisation très complètes et performantes ;
- parallélisation possible facilement (efficacité, etc.) ;
- communauté nombreuse et active ;
- multi-plateformes (scripts python sur Abaqus sur vos machines – Windows – et sur le cluster – Linux)

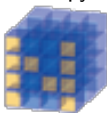


Les principales bibliothèques

1. Calcul scientifique

Pour plus d'information, consulter : <http://numpy.org/> et <http://scipy.org/>

numpy



scipy



- broadcasting
- multiplication de matrices
- traitement du signal
- traitement d'images
- optimisation
- interpolation
- intégration numérique
- algèbre linéaire

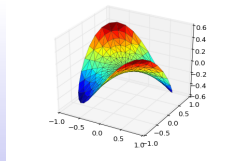
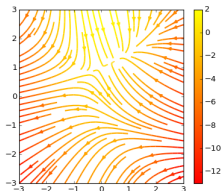
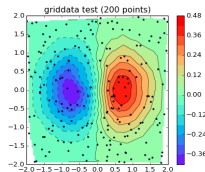
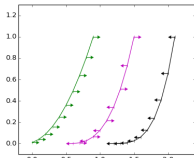


Les principales librairies

2. Visualisation de données

Plus d'informations sur les sites des modules : <http://matplotlib.org/>.

matplotlib





Quelques bibliothèques

- numexpr : <https://code.google.com/p/numexpr/>
- NLOpt : <http://ab-initio.mit.edu/wiki/index.php/NLOpt>
- scikits : <https://scikits.appspot.com/>
- **pyQt5** :
<http://www.riverbankcomputing.co.uk/software/pyqt/>
- PIL : <http://www.pythonware.com/products/pil/>
- sympy : <http://sympy.org/fr/index.html>
- guiqwt/guidata : <https://code.google.com/p/guiqwt,guidata>
- pyqtgraph : <http://www.pyqtgraph.org/>
- SfePy : <http://sfepy.org/doc-devel/index.html>
- FEniCS : <http://fenicsproject.org/>



Quelques bibliothèques

- numexpr : <https://code.google.com/p/numexpr/>
- NLOpt : <http://ab-initio.mit.edu/wiki/index.php/NLOpt>
- scikits : <https://scikits.appspot.com/>
- **pyQt5** :
<http://www.riverbankcomputing.co.uk/software/pyqt/>
- PIL : <http://www.pythonware.com/products/pil/>
- sympy : <http://sympy.org/fr/index.html>
- guiqwt/guidata : <https://code.google.com/p/guiqwt,guidata>
- pyqtgraph : <http://www.pyqtgraph.org/>
- SfePy : <http://sfepy.org/doc-devel/index.html>
- FEniCS : <http://fenicsproject.org/>

WinPython : <http://www.lfd.uci.edu/~gohlke/pythonlibs/>



Sommaire

1 Introduction

Déroulement de cours

Langage Python

2 Principes généraux

Généralités : variables et données

Les types de base

Exercices

3 Les Types de Python

Rappel : types simples/composites

Transtypage et opérations

4 Instructions du langage Python - Rappel

Commentaires

Littéraux (constantes)

Branchements conditionnels et boucles



Sommaire

- 1 Introduction
 - Déroulement de cours
 - Langage Python
- 2 **Principes généraux**
 - Généralités : variables et données
 - Les types de base
 - Exercices
- 3 **Les Types de Python**
 - Rappel : types simples/composites
 - Transtypage et opérations
- 4 **Instructions du langage Python - Rappel**
 - Commentaires
 - Littéraux (constantes)
 - Branchements conditionnels et boucles



Variables en Python

- Règles de nommage :
 - ne peut commencer par un chiffre ou un “_” (“_” est à réserver pour certaines situations)
 - peut contenir des chiffres, et “_”
 - ne peut pas contenir de séparateur
 - ne doit pas être un mot clé (33 mots clé en Python 3)
 - peut contenir des accents, mais déconseillé
- Choisir des noms parlants (rester raisonnable)



Variables en Python

- Règles de nommage :
 - ne peut commencer par un chiffre ou un “_” (“_” est à réserver pour certaines situations)
 - peut contenir des chiffres, et “_”
 - ne peut pas contenir de séparateur
 - ne doit pas être un mot clé (33 mots clé en Python 3)
 - peut contenir des accents, mais déconseillé
- Choisir des noms parlants (rester raisonnable)

`surelyReachableObjectsWhichHaveToBeMarkedAsSuch`



Variables et données

- Tous les éléments manipulés dans un programme python sont des objets (données).
- Une variable n'est que la référence à l'objet.
- Toute variable référence un objet d'un type bien connu (entier, réel, chaîne de caractères, ...)
- Un emplacement mémoire (adresse binaire) contient des données (binaire)
- Données de taille variable (selon le type défini : entier, réel, ...)
- Python gère ses ressources (mémoire, descripteurs de fichiers...) sans intervention du programmeur, par un mécanisme de comptage de références



Variables et données

- Exemple :

Espace des noms (variables)

Mémoire



```
>>> a = 2
```

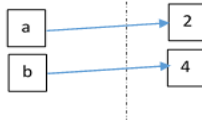


Variables et données

- Exemple :

Espace des noms (variables)

Mémoire



```
>>> a = 2  
>>> b = 4
```

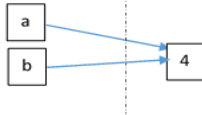


Variables et données

- Exemple :

Espace des noms (variables)

Mémoire



```
>>> a = 2
>>> b = 4
>>> a = b
```

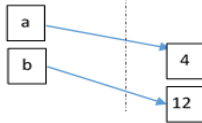


Variables et données

- Exemple :

Espace des noms (variables)

Mémoire



```
>>> a = 2
>>> b = 4
>>> a = b
>>> b = 12
```



Variables en Python

- Il n'est pas nécessaire de déclarer une variable avant de l'utiliser.
- La déclaration d'une variable et son initialisation se fait en même temps.
- Une variable peut changer de type au cours de l'exécution d'un programme.
- Noms de variables sensibles à la casse

Exemple

```
>>> a = 42
>>> b = 33.5
>>> B = "une chaîne de caractères"
>>> a = a + 1.5
>>> a
43.5
```




Fonctions utiles

- Quelques fonctions utiles intégrées à Python :
 - la fonction **help()** donne l'accès à l'aide en ligne. **help(obj)** permet de consulter l'aide sur l'objet obj.
 - la fonction **dir()** la liste des noms accessibles dans la portée actuelle. **dir(obj)** retourne la liste de tous les attributs valides de l'objet obj.
 - la fonction **type(objet)** retourne le type de l'objet obj.
 - la fonction **print(objet)** permet d'afficher le contenu de l'objet obj.

Remarque

On invoque une fonction en utilisant l'opérateur **()** sur l'objet fonction et en plaçant les paramètres entre les parenthèses.



Sommaire

1 Introduction

Déroulement de cours

Langage Python

2 Principes généraux

Généralités : variables et données

Les types de base

Exercices

3 Les Types de Python

Rappel : types simples/composites

Transtypage et opérations

4 Instructions du langage Python - Rappel

Commentaires

Littéraux (constantes)

Branchements conditionnels et boucles



Typage des variables

- **Les types intégrés de Python**

- Les types simples :
 - Entiers signés (**int**), Réels (IEEE 754) (**float**) et Complexes (**complex**), Booléens (**bool**)
- Les types composites (containers) :
 - **Les séquences** : Chaînes de caractères (**str**); Listes (**list**) et Tuples (**tuple**)
 - **Les maps** (hashs) : Dictionnaires (**dict**)
 - **Les ensembles** : le type **set** et le type **frozenset**



Types des variables

Exemple

```
>>> i = 42
>>> type(i)
<class 'int'>
>>> i = 'indice'
>>> type(i)
<class 'str'>
>>> i = 42.0
>>> type(i)
<class 'float'>
>>> print(i)
42.5
>>>
```



Sommaire

- 1 Introduction
 - Déroulement de cours
 - Langage Python
- 2 **Principes généraux**
 - Généralités : variables et données
 - Les types de base
 - Exercices
- 3 **Les Types de Python**
 - Rappel : types simples/composites
 - Transtypage et opérations
- 4 **Instructions du langage Python - Rappel**
 - Commentaires
 - Littéraux (constantes)
 - Branchements conditionnels et boucles



Exercices

Considérant le script suivant :

Exemple (cartons : vert = Vrai / rouge = Faux)

```
>>> a = 12.5
>>> b = a
>>> a = a + 0.5
>>> a
13 ?
```



Exercices

Considérant le script suivant :

Exemple (cartons : vert = Vrai / rouge = Faux)

```
>>> a = 12.5
>>> b = a
>>> a = a + 0.5
>>> a
13 ?
>>> b
12.5 ?
>>>
```



Exercices

Considérant le script suivant :

Exemple (cartons : vert = Vrai / rouge = Faux)

```
>>> a = "bonjour"
>>> b = a
>>> a += " tout le monde"
>>> print(b)
"bonjour tout le monde" ?
>>>
```




Exercices

Considérant le script suivant :

Exemple (cartons : vert = Vrai / rouge = Faux)

```
>>> a = "bonjour"  
>>> a[0:-1] # ==>"b" ?  
>>>
```



Exercices

Considérant le script suivant :

Exemple (cartons : vert = Vrai / rouge = Faux)

```
>>> a = "bonjour"  
>>> a[0] = "B"  
>>> a # ==>"Bonjour" ?  
>>>
```



Exercices

Considérant le script suivant :

Exemple (cartons : vert = Vrai / rouge = Faux)

```
>>> a, b = [1, 2], ['a']  
>>> c = a + b  
>>> b.append('b')  
>>> c # ==> [1, 2, 'a', 'b'] ?
```



Exercices

Considérant le script suivant :

Exemple (cartons : vert = Vrai / rouge = Faux)

```
>>> a, b = [[1], 2], ['a']
>>> c = a + b
>>> a[0].append('b')
>>> c # ==> [[1,'b'], 2, 'a'] ?
```



Sommaire

- 1 **Introduction**
 - Déroulement de cours
 - Langage Python
- 2 **Principes généraux**
 - Généralités : variables et données
 - Les types de base
 - Exercices
- 3 **Les Types de Python**
 - Rappel : types simples/composites
 - Transtypage et opérations
- 4 **Instructions du langage Python - Rappel**
 - Commentaires
 - Littéraux (constantes)
 - Branchements conditionnels et boucles



Sommaire

1 Introduction

Déroulement de cours
Langage Python

2 Principes généraux

Généralités : variables et données
Les types de base
Exercices

3 Les Types de Python

Rappel : types simples/composites
Transtypage et opérations

4 Instructions du langage Python - Rappel

Commentaires
Littéraux (constantes)
Branchements conditionnels et boucles



Les types numériques

- **Les entiers**
 - Leurs représentation n'est limitée que par la taille mémoire.



Les types numériques

- **Les entiers**

- Leurs représentation n'est limitée que par la taille mémoire.
⇒ pas de risque de débordement



Les types numériques

- **Les entiers**

- Leurs représentation n'est limitée que par la taille mémoire.
⇒ pas de risque de débordement

Exemple

```
>>> i = 42
>>> i
42
>>> math.factorial(i)
1405006117752879898543142606244511569936384000000000
```



Les types numériques

- **Les entiers**
- **Les réels**
 - Sont représentés en en double précision (64 bits).
⇒ le plus petit nombre qu'il est possible de distinguer de 1 est $2.22E - 16$



Les types numériques

- **Les entiers**
- **Les réels**
 - Sont représentés en en double précision (64 bits).
⇒ le plus petit nombre qu'il est possible de distinguer de 1 est $2.22E - 16$
 - Pour utiliser la simple précision, on peut faire appel à la bibliothèque `numpy`.



Les types numériques

- **Les entiers**
- **Les réels**
 - Sont représentés en en double précision (64 bits).
⇒ le plus petit nombre qu'il est possible de distinguer de 1 est $2.22E - 16$
 - Pour utiliser la simple précision, on peut faire appel à la bibliothèque `numpy`.

Exemple

```
>>> freq = 10e9
>>> freq
10000000000.0
```



Les types numériques

- **Les entiers**
- **Les réels**
- **Les complexes**
 - Sont formés d'un couple de réels qui composent la partie réelle et la partie imaginaire



Les types numériques

- **Les entiers**
- **Les réels**
- **Les complexes**
 - Sont formés d'un couple de réels qui composent la partie réelle et la partie imaginaire
 - Un suffixe **j** pour regrouper deux valeurs composant la partie réelle et la partie imaginaire



Les types numériques

- Les entiers
- Les réels
- Les complexes
 - Sont formés d'un couple de réels qui composent la partie réelle et la partie imaginaire
 - Un suffixe **j** pour regrouper deux valeurs composant la partie réelle et la partie imaginaire
 - Un nombre complexe possède deux attributs en lecture seule **.real** et **.imag** et une méthode **.conjugate()**



Les types numériques

- Les entiers
- Les réels
- Les complexes

Exemple

```
>>> nb = 10 + 5j
>>> nb.real
10.0
>>> nb.conjugate()
(10-5j)
>>> print(complex(10, 5))
(10+5j)
```




Les booléens

- Un objet booléen peut prendre 2 valeurs **True** ou **False**
- Une variable booléenne supporte toutes les opérations logiques (and, or, not)
- Tout objet peut être interprété en tant que valeur booléenne
- Tout ce qui n'est pas faux est vrai
 - l'objet None est faux
 - toutes les valeurs numériques non nulles sont vraies (0, 0.0, 0.0+0.0j sont fausses)
 - tous les agrégats (séquences, dictionnaires, ensembles) sont vrais s'ils contiennent au moins un élément, faux sinon



Les types simples

Remarques

- Les types simples (scalaires, atomiques) permettent de stocker une seule donnée par variable, contrairement aux types composites (containers)
- Les types simples sont immutables (non modifiables)



Le type None

Remarque (type `NoneType`)

- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.



Le type None

Remarque (type `NoneType`)

- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.
- C'est un objet unique utilisable à n'importe quel endroit du programme



Le type None

Remarque (type `NoneType`)

- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.
- C'est un objet unique utilisable à n'importe quel endroit du programme
- Utilisé pour indiquer qu'un identifiant (variable) n'a pas de valeur



Le type None

Remarque (type `NoneType`)

- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.
- C'est un objet unique utilisable à n'importe quel endroit du programme
- Utilisé pour indiquer qu'un identifiant (variable) n'a pas de valeur
- Une fonction qui ne renvoie pas explicitement une valeur, renvoie `None`



Le type None

Remarque (type `NoneType`)

- **None** est un objet particulier signifiant *rien* ou *nul*, dépourvu de valeur.
- C'est un objet unique utilisable à n'importe quel endroit du programme
- Utilisé pour indiquer qu'un identifiant (variable) n'a pas de valeur
- Une fonction qui ne renvoie pas explicitement une valeur, renvoie `None`
- En mode interactif, l'affichage d'une variable (sans passer par `print()`) qui vaut `None` ne donne rien.



Le type None

Exemple

```
>>> a = None
>>> a
>>>                # Rien ne s'affiche
```




Le type None

Exemple

```
>>> a = None
>>> a
>>> print(a)
None
>>> type(a)
<class 'NoneType'>
>>>
```



Les séquences

Définition (Une séquence)

est une collection **ordonnée** d'objets ou chaque objet est accessible via son index (indice)



Les séquences

Définition (Une séquence)

est une collection **ordonnée** d'objets ou chaque objet est accessible via son index (indice)

- On distingue deux familles de séquences
 - 1 Les séquences immutables (non modifiables, non-mutables) : Les chaînes de caractères (**str**) et les tuples (**tuple**)
 - 2 Les séquences mutables (modifiables) : Les listes



Les séquences

Définition (Une séquence)

est une collection **ordonnée** d'objets ou chaque objet est accessible via son index (indice)

- On distingue deux familles de séquences
 - 1 Les séquences immutables (non modifiables, non-mutables) : Les chaînes de caractères (**str**) et les tuples (**tuple**)
 - 2 Les séquences mutables (modifiables) : Les listes

Remarque

Toutes les opérations sur les séquences immutables sont disponibles sur les séquences mutables.



Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0



Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index



Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```



Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^e indice jusqu'au $(j - 1)^e$ indice

```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```




Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^e indice jusqu'au $(j - 1)^e$ indice

```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```

- `[:j]` : désigne tout ce qui précède j (j^e élément exclu)



Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^{e} indice jusqu'au $(j - 1)^{\text{e}}$ indice

```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```

- `[:j]` : désigne tout ce qui précède j (j^{e} élément exclu)
- `[i:]` : désigne tout ce qui suit i (i^{e} élément inclu)



Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^{e} indice jusqu'au $(j - 1)^{\text{e}}$ indice

```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```

- `[:j]` : désigne tout ce qui précède j (j^{e} élément exclu)
- `[i:]` : désigne tout ce qui suit i (i^{e} élément inclu)
- `[:]` : tous les éléments (utilisé pour réaliser une copie *superficielle*)



Les séquences

Indexation des séquences :

- Les séquences sont indexées en commençant par 0
- Chaque élément (objet) d'une séquence est accessible via son index
- Si l'index est négatif, on accède à la séquence à partir de la fin.

```
>>> s[-1] # renvoie le dernier élément de la séquence s
```

- La tranche `[i:j]` désigne tous les éléments commençant de i^{e} indice jusqu'au $(j - 1)^{\text{e}}$ indice

```
>>> s[0:2] # renvoie une séquence contenant s[0] et s[1]
```

- `[:j]` : désigne tout ce qui précède j (j^{e} élément exclu)
- `[i:]` : désigne tout ce qui suit i (i^{e} élément inclu)
- `[:]` : tous les éléments (utilisé pour réaliser une copie *superficielle*)
- `[i:j:pas]` : pour réaliser une sélection avec un pas donné



Chaîne de caractères

- délimitée par des apostrophes simple `'` ou triple `'''`, des guillemets simples `"` ou triple `"""`



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """

```
>>> s = 'l\'or'
```



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """

```
>>> s = 'l\'or'  
>>> s = "l'or"
```



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """

```
>>> s = 'l\'or'
>>> s = "l'or"
>>> s = """ c'est une phrase
en plusieurs ligne"""
```




Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """

	0	1	2	3	4	5	6	7	8	9	10
s	H	e	l	l	o		W	o	r	l	d
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = "Hello World"
```



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """

	0	1	2	3	4	5	6	7	8	9	10
s	H	e	l	l	o		W	o	r	l	d
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = "Hello World"
>>> print(s[0] , s[-1], s[2:5], s [:-1])
```



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """

	0	1	2	3	4	5	6	7	8	9	10
s	H	e	l	l	o		W	o	r	l	d
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = "Hello World"
>>> print(s[0] , s[-1], s[2:5], s [:-1])
H d llo Hello Worl
```



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.

	0	1	2	3	4	5	6	7	8	9	10
s	H	e	l	l	o		W	o	r	l	d
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>>s[0]='h'
```



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.

	0	1	2	3	4	5	6	7	8	9	10
s	H	e	l	l	o		W	o	r	l	d
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>>s[0]='h'
```

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

```
File "<pyshell #14>", line 1, in <module>
```

```
TypeError : 'str ' object does not support  
item ...
```



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.

	0	1	2	3	4	5	6	7	8	9	10
s	H	e	l	l	o		W	o	r	l	d
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>>s[0]='h'  
>>> ch1 = "h" + s [1:]
```



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.

	0	1	2	3	4	5	6	7	8	9	10
s	H	e	l	l	o		W	o	r	l	d
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> ch1 = "h" + s [1:]  
>>> ch1  
'hello World'
```



Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.
- Les opérateurs '*' et '+' réalisent la multiplication et la concaténation

	0	1	2	3	4	5	6	7	8	9	10
s	H	e	l	l	o		W	o	r	l	d
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> a = 'a'*2+'b'
```




Chaîne de caractères

- délimitée par des apostrophes simple ' ou triple '' , des guillemets simples " ou triple """
- Pour la modifier, il faut créer une nouvelle.
- Les opérateurs '*' et '+' réalisent la multiplication et la concaténation

	0	1	2	3	4	5	6	7	8	9	10
s	H	e	l	l	o		W	o	r	l	d
	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

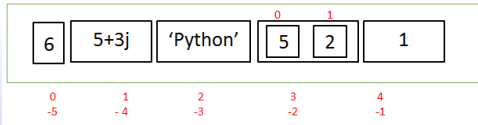
```
>>> a = 'a'*2+'b'
>>> a
'aab'
```



Les listes

- Elles sont composées d'un ensemble d'éléments placés entre crochets `[]` et séparés par des virgules,

```
Lst = [ 6 , 5+3j, 'Python', [5, 2], 1]
```

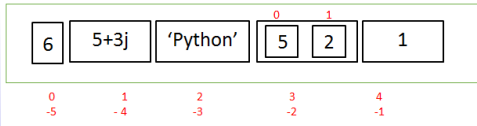




Les listes

- Elles sont composées d'un ensemble d'éléments placés entre crochets `[]` et séparés par des virgules,
- Elles sont modifiables

```
Lst = [ 6 , 5+3j, 'Python', [5, 2], 1]
```

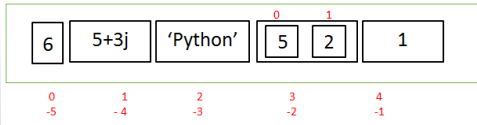




Les listes

- Elles sont composées d'un ensemble d'éléments placés entre crochets `[]` et séparés par des virgules,
- Elles sont modifiables
- Elles peuvent contenir des éléments hétérogènes (de types différents : `str`, `list`, etc)

```
Lst = [ 6 , 5+3j, 'Python', [5, 2], 1]
```





Les listes

Propriétés (Opérations)

len(s)

`ls[i] = x`

del `ls[i]`

del `ls[i:j]`

list(seq)

`ls1 += ls2`

`ls3 = ls1 + ls2`

`ls1 = ls2 * i`

taille de s

affecte x au i^e élément de ls

supprime le i^e élément de ls

supprime la tranche de i^e au (j - 1)^e élément

transforme la séquence seq en une nouvelle liste

ajoute le contenu de ls2 à la fin de ls1

renvoie une nouvelle liste : concaténation de ls1 et ls2

renvoie une nouvelle liste : duplication de i fois de ls2

ls.append(x)

ls.insert(pos, el)

ls.remove(el)

ls.pop(pos)

*****help(list)**

ajouter l'élément x à la fin de la liste ls

insérer l'élément el à l'index pos spécifié

supprime la première occurrence de la valeur el

envoie et supprime l'élément d'index pos (le dernier sinon)

pour plus de détails sur les propriétés de l'objet list



Les listes

Remarque (Manipulation de matrice)

- Utilisation des listes imbriquées \Rightarrow moins flexible et pas efficace.

```
>>> mat_2D = [ [1,2,3], [4,5,6] ] # listes imbriquées
```



Les listes

Remarque (Manipulation de matrice)

- Utilisation des listes imbriquées \Rightarrow moins flexible et pas efficace.

```
>>> mat_2D = [ [1,2,3], [4,5,6] ] # listes imbriquées
```
- Utilisation de la bibliothèque **numpy**



Les listes

Remarque (Manipulation de matrice)

- Utilisation des listes imbriquées \Rightarrow moins flexible et pas efficace.

```
>>> mat_2D = [ [1,2,3], [4,5,6] ] # listes imbriquées
```
- Utilisation de la bibliothèque **numpy**



Les listes

Remarque (Manipulation de matrice)

- Utilisation des listes imbriquées \Rightarrow moins flexible et pas efficace.

```
>>> mat_2D = [ [1,2,3], [4,5,6] ] # listes imbriquées
```
- Utilisation de la bibliothèque **numpy**

Copie d'une liste

- `lst2 = lst1` # n'effectue pas de copie des données
 \Rightarrow les 2 variables `lst1` et `lst2` référencent la même liste



Les listes

Remarque (Manipulation de matrice)

- Utilisation des listes imbriquées \Rightarrow moins flexible et pas efficace.

```
>>> mat_2D = [ [1,2,3], [4,5,6] ] # listes imbriquées
```
- Utilisation de la bibliothèque **numpy**

Copie d'une liste

- `lst2 = lst1` # n'effectue pas de copie des données
 \Rightarrow les 2 variables `lst1` et `lst2` référencent la même liste
- `lst2 = lst1[:]` # duplique que les éléments de 1^{er} niveau



Les listes

Remarque (Manipulation de matrice)

- Utilisation des listes imbriquées \Rightarrow moins flexible et pas efficace.

```
>>> mat_2D = [ [1,2,3], [4,5,6] ] # listes imbriquées
```
- Utilisation de la bibliothèque **numpy**

Copie d'une liste

- `lst2 = lst1` # n'effectue pas de copie des données
 \Rightarrow les 2 variables `lst1` et `lst2` référencent la même liste
- `lst2 = lst1[:]` # duplique que les éléments de 1^{er} niveau
- `lst2 = copy.deepcopy(lst1)` # depuis le module **copy**



Les listes

Exemple (Copie de liste)

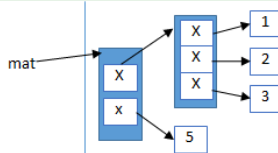
```
>>> import copy # importation du module copy
>>> mat = [[1, 2, 3], 5]
>>> m1 = mat[:]
>>> m2 = copy.deepcopy(mat)
>>> mat[1] = 100
>>> mat[0][1] = 20
>>> mat
[[1, 20, 3], 100]
>>> m1
[[1, 20, 3], 5]
>>> m2
[[1, 2, 3], 5]
>>>
```



Les listes

Exemple (Copie de liste)

```
>>> mat = [[1, 2, 3], 5]
```

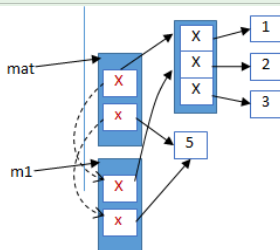




Les listes

Exemple (Copie de liste)

```
>>> mat = [[1, 2, 3], 5]  
>>> m1 = mat[:]
```

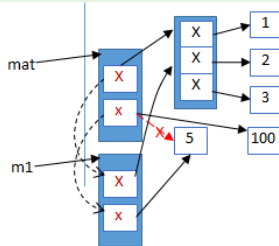




Les listes

Exemple (Copie de liste)

```
>>> mat = [[1, 2, 3], 5]  
>>> m1 = mat[:]  
>>> mat[1] = 100
```

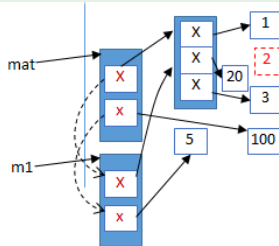




Les listes

Exemple (Copie de liste)

```
>>> mat = [[1, 2, 3], 5]  
>>> m1 = mat[:]  
>>> mat[1] = 100  
>>> mat[0][1] = 20  
>>>
```

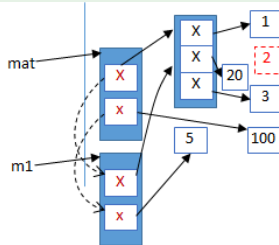




Les listes

Exemple (Copie de liste)

```
>>> mat = [[1, 2, 3], 5]  
>>> m1 = mat[:]  
>>> mat[1] = 100  
>>> mat[0][1] = 20  
>>> mat  
[[1, 20, 3], 100]  
>>> m1  
[[1, 20, 3], 5]  
>>>
```

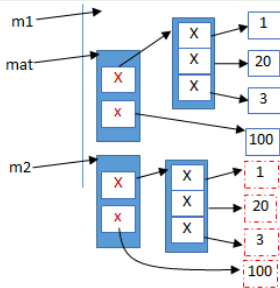




Les listes

Exemple (Copie de liste)

```
>>> mat = [[1, 2, 3], 5]
>>> m1 = mat[:]
>>> mat[1] = 100
>>> mat[0][1] = 20
>>> mat
[[1, 20, 3], 100]
>>> m1
[[1, 20, 3], 5]
>>> m2 = copy.deepcopy(mat)
```

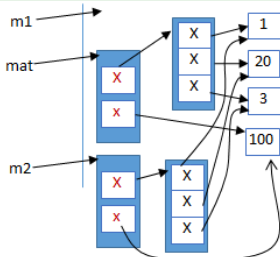




Les listes

Exemple (Copie de liste)

```
>>> mat = [[1, 2, 3], 5]  
>>> m1 = mat[:]  
>>> mat[1] = 100  
>>> mat[0][1] = 20  
>>> mat  
[[1, 20, 3], 100]  
>>> m1  
[[1, 20, 3], 5]  
>>> m2 = copy.deepcopy(mat)
```





N-Uplets (Les tuples)

Définition (Un tuple)

Une séquence (collection) ordonnée **non modifiable** d'éléments hétérogènes (*c'est une liste non modifiable*)



N-Uplets (Les tuples)

Définition (Un tuple)

Une séquence (collection) ordonnée **non modifiable** d'éléments hétérogènes (*c'est une liste non modifiable*)

- Ils sont notés entre parenthèses `()`, ou une suite d'éléments séparés par des virgules
- l'accès aux éléments du tuple est réalisé via les indices placés entre crochets `[]` (comme pour les listes).



N-Uplets (Les tuples)

Définition (Un tuple)

Une séquence (collection) ordonnée **non modifiable** d'éléments hétérogènes (*c'est une liste non modifiable*)

- Ils sont notés entre parenthèses `()`, ou une suite d'éléments séparés par des virgules
- l'accès aux éléments du tuple est réalisé via les indices placés entre crochets `[]` (comme pour les listes).

Remarques

- Si un seul élément est dans un tuple, le faire suivre d'une virgule.
- Le tuple possède les mêmes **méthodes** que la liste, à l'exception de celles permettant une modification.



N-Uplets (Les tuples)

Exemple (Tuple)

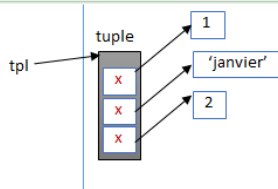
```
>>> nb = (1) ; ch = ('ab')      # => entier & str !
>>> tpl = (1,)                  # ou  tpl =1, => tuple
>>> tpl = (1, 'janvier', 2)      # on peut omettre ( )
>>> len(tpl)                     # => 3 (nombre d'éléments)
>>> tpl[0:2]                     # => le tuple (1, 'janvier')
>>> tpl[2] = 3                   # => erreur (tuple non modifiable)
>>> tpl += 5, 'mars'             # crée un nouv objet tuple
>>> 'mars' in tpl                # => True
>>> tpl2 = tuple([5,6,12])       # copie liste => nouv tuple (5,6,12)
>>> tpl3 = tuple('hello')       # copie chaine => tuple
                                #      => ('h','e','l','l','o')
>>> lst2= list(tpl2)            # copie tuple vers liste
                                #      => [5,6,12]
```



N-Uplets (Les tuples)

Exemple (Tuple)

```
>>> tpl = 1, 'janvier', 2  
>>>
```

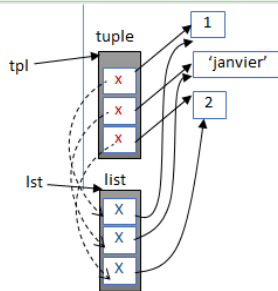




N-Uplets (Les tuples)

Exemple (Tuple)

```
>>> tpl = 1, 'janvier', 2  
>>> lst = list(tpl)  
>>>
```





Dictionnaire

Définition (Dictionnaire)

Est une collection modifiable, **non ordonnée** d'éléments hétérogènes ou chacun de ses éléments est indicé par une **clé**.



Dictionnaire

Définition (Dictionnaire)

Est une collection modifiable, **non ordonnée** d'éléments hétérogènes ou chacun de ses éléments est indicé par une **clé**.

- Il est noté entre accolades {}, ou créé à partir d'une liste de couples (clé, valeur) passée en argument à la fonction **dict()**.



Dictionnaire

Définition (Dictionnaire)

Est une collection modifiable, **non ordonnée** d'éléments hétérogènes ou chacun de ses éléments est indicé par une **clé**.

- Il est noté entre accolades {}, ou créé à partir d'une liste de couples (clé, valeur) passée en argument à la fonction **dict()**.

Exemple :

```
dic = {1: 'a', 2: 'b'}  
# ou élément après élément :  
dic={} ; dic[1] = 'a' ; dic[2] = 'b'  
# en utilisant la fonction dict()  
dic = dict([(1, 'a'), (2, 'b')])  
dic = dict(zip((1, 2), ('a', 'b')))  
len(dic) # => 2 paires clé:valeur  
1 in dic # test existence clé => True  
5 in dic # => False, car 5 n'est pas une clé
```



Dictionnaire

Propriétés

- Les **clés** doivent être de type non modifiable (exemple : les types simples) et uniques.
- Les valeurs peuvent être de n'importe quel type
- Les valeurs sont accessibles par leurs clés
- Pas de notion d'ordre



Dictionnaire

- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
 - Récupération d'une valeur

```
dic[1] # => 'a'  
dic['cx'] # retourne erreur KeyError  
dic.get(2) # => 'b'  
dic.get('cx', 'erreur blabla') # => 'erreur blabla'
```



Dictionnaire

- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
 - Récupération d'une valeur
 - Ajout, modification et suppression des éléments

```
dic['c'] = 3 # => {1: 'a', 'c': 3, 2: 'b'}  
dic[2] = 10 # => {1: 'a', 'c': 3, 2: 10}  
del dic[1] # supp. de l'élément 1: 'a'  
val = dic.pop(2) # => 'b' et supp. de l'élément 2: 'b'
```



Dictionnaire

- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
 - Récupération d'une valeur
 - Ajout, modification et suppression des éléments
 - Fusion de dictionnaires et mise à jour

```
dic.update({2: 20, 'c': 30}) # dic.update(dict)  
# => dic = {1: 'a', 2: 20, 'c': 30}
```




Dictionnaire

- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
 - Récupération d'une valeur
 - Ajout, modification et suppression des éléments
 - Fusion de dictionnaires et mise à jour
 - Parcourir un dictionnaire (via des objets itérables par une boucle `for`)

```
dic.keys() # => dict_keys([1, 2])
dic.values() # => dict_values(['a', 'b'])
dic.items() # => dict_items([(1, 'a'), (2, 'b')])

# copie sur liste ou tuple
list(dic.keys()) # => [1, 2]
tuple(dic.values()) # => ('a', 'b')
```



Dictionnaire

- Opérations (Exemple : `dic = {1: 'a', 2: 'b'}`)
 - Récupération d'une valeur
 - Ajout, modification et suppression des éléments
 - Fusion de dictionnaires et mise à jour
 - Parcourir un dictionnaire (via des objets itérables par une boucle **for**)
 - Copie d'un dictionnaire

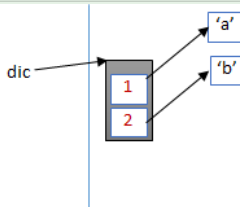
```
dict1 = dic.copy() # => copie superficielle
# import copy
dic2 = copy.deepcopy(dic) # => copie profonde
```



Dictionnaire

Exemple

```
>>> dic = {1: 'a', 2: 'b'}  
>>>
```

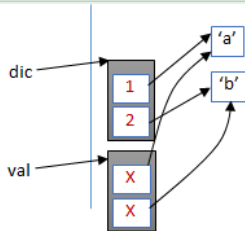




Dictionnaire

Exemple

```
>>> dic = {1: 'a', 2: 'b'}  
>>> val = list(dic.values())
```

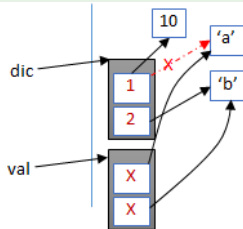




Dictionnaire

Exemple

```
>>> dic = {1: 'a', 2: 'b'}  
>>> val = list(dic.values())  
>>> dic[1] = 10  
>>> # dic ==? Val ==?
```

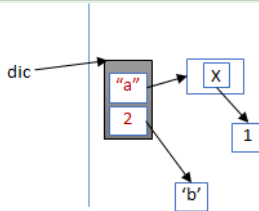




Dictionnaire

Exemple

```
>>> dic = {'a' : [1], 2 : 'b'}
```

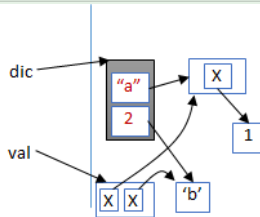




Dictionnaire

Exemple

```
>>> dic = {'a': [1], 2: 'b'}  
>>> val = list(dic.values())  
>>> # val == ?
```

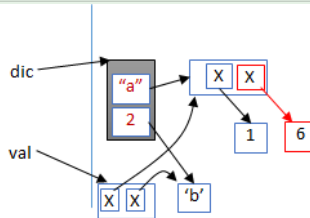




Dictionnaire

Exemple

```
>>> dic = {'a' : [1], 2 : 'b'}  
>>> val = list(dic.values())  
>>> # val == ['b', [1]]  
>>> dic['a'].append(6)  
>>> # val ==?
```

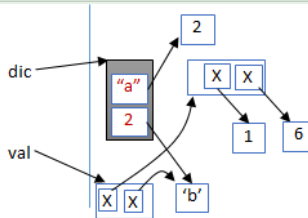




Dictionnaire

Exemple

```
>>> dic = {'a' : [1], 2 : 'b'}  
>>> val = list(dic.values())  
>>> # val == ['b', [1]]  
>>> dic['a'].append(6)  
>>> # val == ['b', [1, 6]]  
>>> dic['a'] = 2  
>>> # val == ?
```





Sommaire

1 Introduction

Déroulement de cours
Langage Python

2 Principes généraux

Généralités : variables et données
Les types de base
Exercices

3 Les Types de Python

Rappel : types simples/composites
Transtypage et opérations

4 Instructions du langage Python - Rappel

Commentaires
Littéraux (constantes)
Branchements conditionnels et boucles



Transtypage / cast

- Objectif : convertir d'un type vers un autre.
- Exemple : `b = 23; ch = str(b)`



Transtypage / cast

- Objectif : convertir d'un type vers un autre.
- Exemple : `b = 23; ch = str(b)`
- Utilisation des fonctions : `int()`, `float()`, `bool()`, `str()`, `complex()`, `list()`, `tuple()`



Transtypage / cast

Exemple (Transtypage)

```
>>> int(12.5)    # => 12
>>> int('125')  # => 125
>>> float(125)   # => 125.0
>>> int("1000110" ,2) # => 70
>>> complex("12+.5j") # => (12+0.5j)
>>> int(True)    # => 1
>>> bool(0)      # => False
>>> bool(124)    # => True
>>> str(12+3j)   # => '(12+3j)'
```



Affectation

Définition (Affectation (assignation))

Affectation : donner une ou plusieurs valeurs à une ou à plusieurs variables réalisée par l'opérateur "=". Elle peut être simple ou multiple.



Affectation

Définition (Affectation (assignation))

Affectation : donner une ou plusieurs valeurs à une ou à plusieurs variables réalisée par l'opérateur "=". Elle peut être simple ou multiple.

Exemple (Affectations)

```
# affectation simple
a = 2

# affectation multiple
a = b = 5 # => a=5 et b=5
a, b = b, a # échange le contenu de a et b
c, d, e = 1, 5, 6 # => c=1; e=5 et e=6
a, *reste = [1, 2, 3] # => a=1 et reste=[2, 3]
```



Affectation

Exemple (Forme condensée)

```
a = b = 2
```

```
a += 2
```

```
b *= 3
```




Affectation

Exemple (Forme condensée)

```
a = b = 2
```

```
a += 2
```

```
b *= 3
```

- Affectation très utile : `i += 1` (ou `i = i+1`)



Affectation

Exemple (Forme condensée)

```
a = b = 2  
a += 2  
b *= 3
```

- Affectation très utile : `i += 1` (ou `i = i+1`)
- Autres formes condensées avec : `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `>>=`, `<<=`, `&=`, `^=`, `|=`.

Attention

Python n'accepte pas les expressions : `i++`, `i--`, `++i` et `--i`



Comparaisons

Opérateur	Exemple	Signification
>	<code>a > 10</code>	strictement supérieur
<	<code>a < 10</code>	strictement inférieur
>=	<code>a >= 10</code>	supérieur ou égal
<=	<code>a <= 10</code>	inférieur ou égal
==	<code>a == 10</code>	égal à
!=	<code>a != 10</code>	différent de
is	<code>a is b</code>	a et b représentent le même objet
is not	<code>a is not b</code>	a et b ne représentent pas le même objet



Comparaisons

Opérateur	Exemple	Signification
>	<code>a > 10</code>	strictement supérieur
<	<code>a < 10</code>	strictement inférieur
>=	<code>a >= 10</code>	supérieur ou égal
<=	<code>a <= 10</code>	inférieur ou égal
==	<code>a == 10</code>	égal à
!=	<code>a != 10</code>	différent de
is	<code>a is b</code>	a et b représentent le même objet
is not	<code>a is not b</code>	a et b ne représentent pas le même objet

Remarque

- Le test d'égalité `==` concerne le contenu,
- le test `is` concerne la référence



Comparaisons

Remarque

- Le test d'égalité `==` concerne le contenu,
- le test `is` concerne la référence

Exemple (comparaison)

```
>>> 2 is 2
True
>>> 2+2 == 4
True
>>> [2+2] is [4]
False
>>> [2+2] == [4]
True
```



Opérations logiques

Opérateur	Exemple	Signification
not	not a	NON logique
and	a and b	ET logique : évaluation paresseuse
or	a or b	OU logique : évaluation paresseuse
&	a & b	ET logique
^	a ^ b	OU exclusif logique
	a b	OU logique



Opérations logiques

Opérateur	Exemple	Signification
<code>not</code>	<code>not a</code>	NON logique
<code>and</code>	<code>a and b</code>	ET logique : évaluation paresseuse
<code>or</code>	<code>a or b</code>	OU logique : évaluation paresseuse
<code>&</code>	<code>a & b</code>	ET logique
<code>^</code>	<code>a ^ b</code>	OU exclusif logique
<code> </code>	<code>a b</code>	OU logique

Remarque : évaluation paresseuse

- `a and b` évalue et retourne `a` si `a` est `False`, sinon évalue et retourne `b`.
- `a or b` évalue et retourne `a` si `a` est `True`, sinon évalue et retourne `b`



Opérations logiques

Opérateur	Exemple	Signification
not	not a	NON logique
and	a and b	ET logique : évaluation paresseuse
or	a or b	OU logique : évaluation paresseuse
&	a & b	ET logique
^	a ^ b	OU exclusif logique
	a b	OU logique

Remarque : évaluation paresseuse

- a and b évalue et retourne a si a est False, sinon évalue et retourne b.
- a or b évalue et retourne a si a est True, sinon évalue et retourne b

```
>>> 5 and 6 # => retourne 6
>>> 0 and 5 # => retourne 0
>>> 5 or 0 # => retourne 5
>>> 0 or 5 # => retourne 5
```




Sommaire

- 1 **Introduction**
 - Déroulement de cours
 - Langage Python
- 2 **Principes généraux**
 - Généralités : variables et données
 - Les types de base
 - Exercices
- 3 **Les Types de Python**
 - Rappel : types simples/composites
 - Transtypage et opérations
- 4 **Instructions du langage Python - Rappel**
 - Commentaires
 - Littéraux (constantes)
 - Branchements conditionnels et boucles



Règles de base

- Python est sensible à la casse



Règles de base

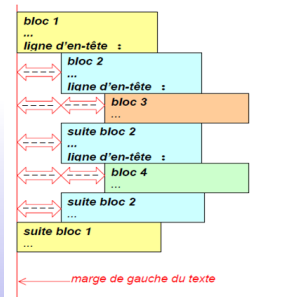
- Python est sensible à la casse
- Une instruction est forcément dans un bloc et peut tenir sur plusieurs lignes



Règles de base

- Python est sensible à la casse
- Une instruction est forcément dans un bloc et peut tenir sur plusieurs lignes
- Un bloc de code commence par " : " et est indenté plus à droite pour le bloc contenant

Exemple :





Règles de base

- Python est sensible à la casse
- Une instruction est forcément dans un bloc et peut tenir sur plusieurs lignes
- Un bloc de code commence par " : " et est indenté plus à droite pour le bloc contenant

Remarques

- Il est recommandé d'utiliser 4 <espace> pour un niveau d'indentation
- Un bloc de code doit contenir au minimum une instruction. S'il n'en a pas, on peut utiliser l'instruction **pass**



Sommaire

1 Introduction

Déroulement de cours
Langage Python

2 Principes généraux

Généralités : variables et données
Les types de base
Exercices

3 Les Types de Python

Rappel : types simples/composites
Transtypage et opérations

4 Instructions du langage Python - Rappel

Commentaires
Littéraux (constantes)
Branchements conditionnels et boucles



Commentaires

- Commentaires : remarques en langage naturel



Commentaires

- Commentaires : remarques en langage naturel
- Ignorés par l'interpréteur



Commentaires

- Commentaires : remarques en langage naturel
- Ignorés par l'interpréteur
- Deux types de commentaires :



Commentaires

- Commentaires : remarques en langage naturel
- Ignorés par l'interpréteur
- Deux types de commentaires :

Exemple (Commentaire sur une ligne)

```
# Une division réelle  
a = 12/5  
b = a // 3 # division entière
```



Commentaires

- Commentaires : remarques en langage naturel
- Ignorés par l'interpréteur
- Deux types de commentaires :

Exemple (Documentation automatique)

```
def somProd(var1, var2):  
    """Fonction calculant somme et produit  
    de var1 et var2  
    Retour :  
    un tuple (var1+var2, var1*var2)"""  
    return (var1+var2, var1*var2)
```



Sommaire

- 1 **Introduction**
 - Déroulement de cours
 - Langage Python
- 2 **Principes généraux**
 - Généralités : variables et données
 - Les types de base
 - Exercices
- 3 **Les Types de Python**
 - Rappel : types simples/composites
 - Transtypage et opérations
- 4 **Instructions du langage Python - Rappel**
 - Commentaires
 - Littéraux (constantes)
 - Branchements conditionnels et boucles



Types de littéraux

- Entiers : base 10

Exemple (Littéral)

42, -7



Types de littéraux

- Entiers : base 10 , base 16

Exemple (Littéral)

0x9AE3



Types de littéraux

- Entiers : base 10 , base 16 , base 8

Exemple (Littéral)

0o20 (= 16)



Types de littéraux

- Entiers : base 10 , base 16 , base 8, base 2

Exemple (Littéral)

0b101010 (= 42)



Types de littéraux

- Entiers : base 10 , base 16 , base 8, base 2
- Réels : double précision

Exemple (Littéral)

3.14, 6.02E23 (= $6.02 \cdot 10^{23}$)



Types de littéraux

- Entiers : base 10 , base 16 , base 8, base 2
- Réels : double précision
- Booléens

Exemple (Littéral)

True, False



Types de littéraux

- Entiers : base 10 , base 16 , base 8, base 2
- Réels : double précision
- Booléens
- Complexes

Exemple (Littéral)

```
1j, (-1)**0.5, 3**0.5/2+0.5j
```



Types de littéraux

- Entiers : base 10 , base 16 , base 8, base 2
- Réels : double précision
- Booléens
- Complexes
- Chaînes de caractères

Exemple (Littéral)

"Une chaîne de caractères", 'l\'expression "dauphine" aussi'
'Ex\ u0065llent travail' (= 'Excellent travail')



Sommaire

1 Introduction

Déroulement de cours
Langage Python

2 Principes généraux

Généralités : variables et données
Les types de base
Exercices

3 Les Types de Python

Rappel : types simples/composites
Transtypage et opérations

4 Instructions du langage Python - Rappel

Commentaires
Littéraux (constantes)
Branchements conditionnels et boucles



if, elif, else

- Teste une condition booléenne
- Si (if) **vrai**, exécute une partie du code
- Sinon (else), exécute autre partie



if, elif, else

- Teste une condition booléenne
- Si (if) **vrai**, exécute une partie du code
- Sinon (else), exécute autre partie

Exemple (if, else)

```
if expressionBooleenne1:
    action1
else:
    if expressionBooleenne2:
        action2
    else:
        action3
```



if, elif, else

- Teste une condition booléenne
- Si (if) **vrai**, exécute une partie du code
- Sinon (else), exécute autre partie

Exemple (if, elif, else)

```
if expressionBooleenne1:
    action1
elif expressionBooleenne2:
    action2
else:
    action3
```


Remarque

- `if` peut s'écrire sans `else`

Remarque

- `if` peut s'écrire sans `else`

Correct mais à proscrire

```
if condition:
    pass
else:
    action
```

Remarque

- `if` peut s'écrire sans `else`

Correct mais à proscrire

```
if condition:  
    pass  
else:  
    action
```

Forme préférée

```
if not condition :  
    action
```



Conditions multiples, et fonctions logiques

Exemple (conditions multiples)

```
if 0 <= a < 10:  
# équivalence  
if (0 <= a) and (a < 10):
```

Remarques (fonctions logiques : any, all)

- Tester des séquences et des objets itérables
 - `any(objet)` : retourne True si au moins un des éléments est True
`any(range(100)) # => True`
 - `all(objet)` : retourne True si tous les éléments sont True
`all(range(100)) # => False`



Principe

- Boucle = structure de contrôle
- But : exécuter certaines opérations plusieurs fois
- Il existe deux types de boucles en Python 3 : for, while
- Les deux boucles sont équivalentes



while

```
while condition :  
    actions
```



while

```
while condition :  
    actions
```

Exemple (boucle while)

```
i=0  
while i<10: # boucle tant que i<10  
    print(i) # affiche la valeur de i  
    i = i+1 # incrémente i  
# maintenant i vaut 10
```



while : Sortie de boucle

- Un bloc **else** : peut être ajouté à la fin de la boucle
- Le mot clé **break** permet de sortir de la boucle
- Le mot clé **continue** permet de passer immédiatement à l'itération suivante

Exemple (do ... until)

```
while True :  
    # corps de la boucle  
    if exitcondition :  
        break
```




for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable

Syntaxe :

```
for variable in collection:  
    actions # corps de la boucle
```

Exemple

```
s = ['septembre', 'octobre']  
for n in s:  
    print(n, end=' ') # => septembre octobre
```



for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable

Syntaxe :

```
for variable in collection:  
    actions # corps de la boucle
```

Exemple

```
s = "Python"  
for n in s:  
    print(n, end=' ') # => P y t h o n
```



for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable

Syntaxe :

```
for variable in collection:  
    actions # corps de la boucle
```

Exemple

```
s = 1,2,3,4  
for n in s:  
    print(n, end=' ') # => 1 2 3 4
```



for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable

Syntaxe :

```
for variable in collection:  
    actions # corps de la boucle
```

Exemple

```
s = {'a':10, 'b':'deux'}  
for n in s: # s.values(), s.keys(), s.items()  
    print(n, end=' ') # => a b
```



for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable

Syntaxe :

```
for variable in collection:  
    actions # corps de la boucle
```

Exemple

```
s = {'a':10, 'b':'deux'}  
for n, val in s.items():  
    print(n, end=' ') # => a b
```



for

Objectif : itérer les valeurs d'une collection (liste, tuple, chaîne, dictionnaire) ou de tout objet itérable

Syntaxe :

```
for variable in collection:  
    actions # corps de la boucle
```

Remarque (for : sortie de boucle)

Possibilité de modifier l'exécution d'une boucle **for** avec **break** et **continue** comme dans le cas de la boucle **while**.



Itérateur

Remarque

- L'itérateur `range(deb,fin, inc)` permet d'itérer sur une suite de nombres entiers,
- La fonction `enumerate(sequence)` retourne un objet permettant d'itérer sur l'indice et la valeur d'une séquence

Exemple

```
s = ['septembre', 'octobre']  
for i in range(2):  
    print(s[i], end=' ') # => septembre octobre
```



Itérateur

Remarque

- L'itérateur `range(deb, fin, inc)` permet d'itérer sur une suite de nombres entiers,
- La fonction `enumerate(sequence)` retourne un objet permettant d'itérer sur l'indice et la valeur d'une séquence

Exemple

```
s = ['septembre', 'octobre']  
for ind, val in enumerate(s):  
    print(ind, val, end = ' ')  
# => 0 septembre 1 octobre
```




for : forme condensée

Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.



for : forme condensée

Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.

Syntaxe :

```
liste = [expression for expr in iterable if cond]
```

```
dict = {expr1:expr2 for expr in iterable if cond}
```



for : forme condensée

Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.

Syntaxe :

```
liste = [expression for expr in iterable if cond]
```

```
dict = {expr1:expr2 for expr in iterable if cond}
```

Exemple 1 :

```
puiss2 = [nb*nb for nb in range (1,11) if nb%2==0]  
# => [4, 16, 36, 64, 100]
```



for : forme condensée

Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.

Syntaxe :

```
liste = [expression for expr in iterable if cond]
```

```
dict = {expr1:expr2 for expr in iterable if cond}
```

Exemple 1 :

```
puiss2 = [nb*nb for nb in range (1,11) if nb%2==0]  
# => [4, 16, 36, 64, 100]
```

Exemple 2 :

```
dic = {'a': 12, 'b': 1.5, 'c': 3}  
dic1 = {cle:val for cle,val in dic.items() if  
val**2>10} # => dic1 =
```



for : forme condensée

Objectif : Pour construire un container de type liste, dictionnaire ou set à l'aide d'une boucle for.

Syntaxe :

```
liste = [expression for expr in iterable if cond]
```

```
dict = {expr1:expr2 for expr in iterable if cond}
```

Exemple 1 :

```
puiss2 = [nb*nb for nb in range (1,11) if nb%2==0]  
# => [4, 16, 36, 64, 100]
```

Exemple 2 :

```
dic = {'a': 12, 'b': 1.5, 'c': 3}  
dic1 = {cle:val for cle,val in dic.items() if  
val**2>10} # => dic1 = {'a': 12}
```