Python pour les scientifiques

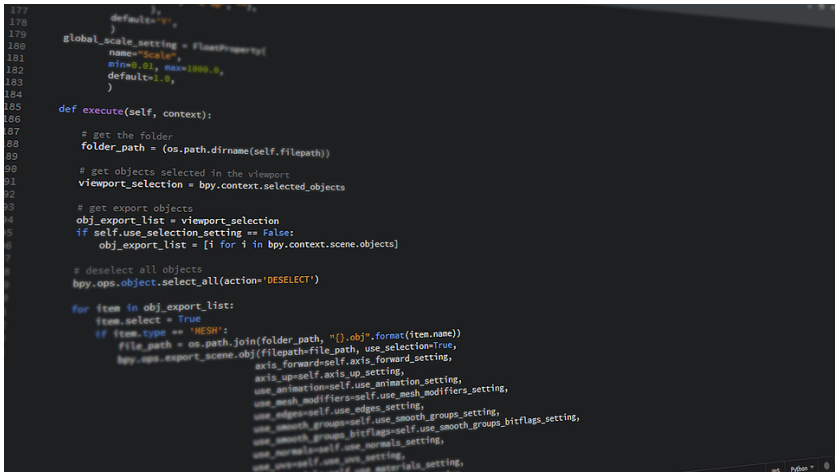
Cyril Desjouy

June, 2016
Updated : June 26, 2023

Ce cours est inspiré de documents de...

- Guido van Rossum (Python's father)
- Matt Huenerfauth (Penn State)
- Richard P. Muller (Caltech)
- SAO Telescope Data Center Team (Harvard)
- Jake VanderPlas (University of Washington)
- David Pine (New York University - Department of Physics)
- Scott Shell (UC Santa Barbara engineering)
- Gaël Varoquaux (Inria Saclay)
- Andrew M. C. Dawes (Pacific University of Oregon)
- Bruno Brouard (Le Mans Université)
- ...

La programmation



Extrait de Pixabay

Chapitre adapté du cours de programmation de Bruno Brouard

Comment résoudre un problème donné ?

Exemple : "*Faire une omelette avec 6 oeufs et la servir à 20h.*"

Prérequis :

- Trouver un exécutant : le **processeur**
- . . . Sachant exécuter certaines actions : les **instructions**

Comment résoudre un problème donné ?

Exemple : "*Faire une omelette avec 6 oeufs et la servir à 20h.*"

Prérequis :

- Trouver un exécutant : le **processeur**
- . . . Sachant exécuter certaines actions : les **instructions**

Les instructions sont énoncées en fonction du processeur :

- **Pour un enfant** (*instructions très détaillées*) :
  1. casser 6 œufs dans un bol à 19:30
  2. battre les œufs avec un fouet
  3. ajouter sel et poivre
  4. cuire à feu doux dans une poêle 15 minutes

Comment résoudre un problème donné ?

Exemple : "*Faire une omelette avec 6 oeufs et la servir à 20h.*"

Prérequis :

- Trouver un exécutant : le **processeur**
- . . . Sachant exécuter certaines actions : les **instructions**

Les instructions sont énoncées en fonction du processeur :

- **Pour un enfant** (*instructions très détaillées*) :
  1. casser 6 œufs dans un bol à 19:30
  2. battre les œufs avec un fouet
  3. ajouter sel et poivre
  4. cuire à feu doux dans une poêle 15 minutes
- **Pour un adulte** (*instructions concises*) :
  1. nombre d'œufs = 6
  2. heure du diner = 8PM

Comment résoudre un problème donné ?

Exemple : "*Faire une omelette avec 6 oeufs et la servir à 20h.*"

Prérequis :

- Trouver un exécutant : le **processeur**
- . . . Sachant exécuter certaines actions : les **instructions**

Les instructions sont énoncées en fonction du processeur :

- **Pour un enfant** (*instructions très détaillées*) :
  1. casser 6 œufs dans un bol à 19:30
  2. battre les œufs avec un fouet
  3. ajouter sel et poivre
  4. cuire à feu doux dans une poêle 15 minutes
- **Pour un adulte** (*instructions concises*) :
  1. nombre d'œufs = 6
  2. heure du diner = 8PM
- **Pour un ordinateur** (*séquence de 0 et de 1*) :
  1. 10011110100110011101001010010010100100101100111...

Comment résoudre un problème donné ?

Exemple : "*Faire une omelette avec 6 oeufs et la servir à 20h.*"

Prérequis :

- Trouver un exécutant : le **processeur**
- . . . Sachant exécuter certaines actions : les **instructions**

Les instructions sont énoncées en fonction du processeur :

- **Pour un enfant** (*instructions très détaillées*) :
  1. casser 6 œufs dans un bol à 19:30
  2. battre les œufs avec un fouet
  3. ajouter sel et poivre
  4. cuire à feu doux dans une poêle 15 minutes
- **Pour un adulte** (*instructions concises*) :
  1. nombre d'œufs = 6
  2. heure du diner = 8PM
- **Pour un ordinateur** (*séquence de 0 et de 1*) :
  1. 10011110100110011101001010010010100100101100111...

Instruction : Action encodée dans un langage connu par le processeur, contenant ou manipulant des données	**Algorithme :** Séquence d'instructions pouvant être exécutée par un processeur et permettant de résoudre un problème

Ordinateur : Rapide et puissant, mais dénoué d'intelligence et d'imagination
(pour le moment !)

Pas de solution proposée à un problème donné !

Programmeur : Lent mais doué d'intelligence et d'imagination

Explique à l'ordinateur ce qu'on veut qu'il fasse

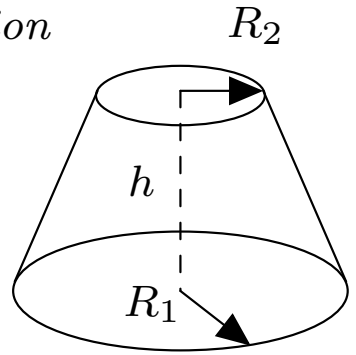
Pour arriver au programme, il existe trois étapes importantes :

1. **Réflexion**: Trouver une solution au problème
2. **Élaboration**: Expliquer comment obtenir cette solution à partir de rien, à travers un (ou des) algorithme(s)
3. **Encodage**: Choisir un langage compris par l'ordinateur, et traduire la solution algorithmique dans ce langage

Programme : Traduction d'un ou d'un ensemble d'algorithmes dans un langage compréhensible par l'ordinateur

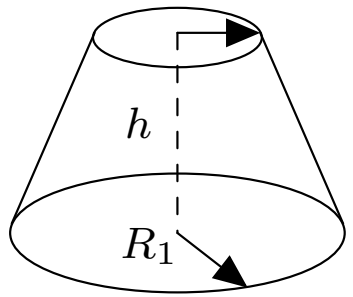
1. **Réflexion** : *trouver une solution au problème*

2. **Élaboration de l'algorithme** : *Expliquer comment obtenir la solution*



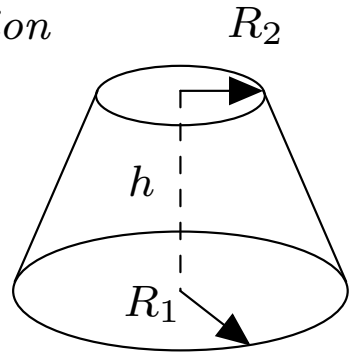
3. **Encodage** : *Choisir un langage et traduire l'algorithme*

1. **Réflexion** : *trouver une solution au problème*
   (depuis internet) $\Longrightarrow$ volume $= h\frac{\pi}{3}(R_1^2 + R_2^2 + R_1 R_2)$
2. **Élaboration de l'algorithme** : *Expliquer comment obtenir la solution*

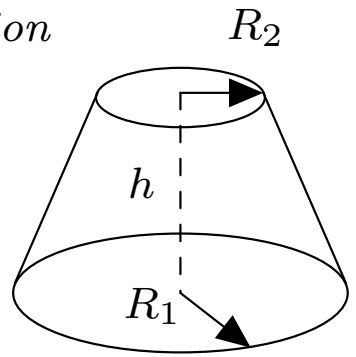


3. **Encodage** : *Choisir un langage et traduire l'algorithme*

1. **Réflexion** : *trouver une solution au problème*
   (depuis internet) $\Longrightarrow$ volume $= h\frac{\pi}{3}(R_1^2 + R_2^2 + R_1 R_2)$
2. **Élaboration de l'algorithme** : *Expliquer comment obtenir la solution*
   - 2.1 Lire le petit rayon et le sauvegarder sous le nom R2
   - 2.2 Lire le plus grand rayon et le sauvegarder sous le nom R1
   - 2.3 Lire la hauteur et la sauvegarder sous le nom H
   - 2.4 Calcul du volume et sauvegarde sous le sous VOL
   - 2.5 Écrire le résultat à l'écran
3. **Encodage** : *Choisir un langage et traduire l'algorithme*



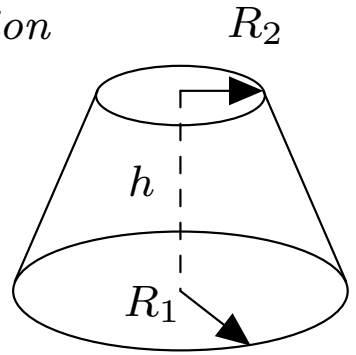
1. **Réflexion** : *trouver une solution au problème*
   (depuis internet) $\Longrightarrow$ volume $= h\frac{\pi}{3}(R_1^2 + R_2^2 + R_1 R_2)$
2. **Élaboration de l'algorithme** : *Expliquer comment obtenir la solution*
   - 2.1 Lire le petit rayon et le sauvegarder sous le nom R2
   - 2.2 Lire le plus grand rayon et le sauvegarder sous le nom R1
   - 2.3 Lire la hauteur et la sauvegarder sous le nom H
   - 2.4 Calcul du volume et sauvegarde sous le sous VOL
   - 2.5 Écrire le résultat à l'écran
3. **Encodage** : *Choisir un langage et traduire l'algorithme*



Langage : Pyhon

```
from  math import  pi
R2 = float(input("Petit rayon ?  "))
R1 = float(input("Grand rayon ?  "))
H = float(input("Hauteur ?  "))
VOL = H*pi/3*(R1**2+R2**2+R1*R2)
print("Volume du cône tronqué :  ", VOL)
```

1. **Réflexion** : *trouver une solution au problème*
   (depuis internet) $\Longrightarrow$ volume $= h\frac{\pi}{3}(R_1^2 + R_2^2 + R_1 R_2)$
2. **Élaboration de l'algorithme** : *Expliquer comment obtenir la solution*
   - 2.1 Lire le petit rayon et le sauvegarder sous le nom R2
   - 2.2 Lire le plus grand rayon et le sauvegarder sous le nom R1
   - 2.3 Lire la hauteur et la sauvegarder sous le nom H
   - 2.4 Calcul du volume et sauvegarde sous le sous VOL
   - 2.5 Écrire le résultat à l'écran
3. **Encodage** : *Choisir un langage et traduire l'algorithme*

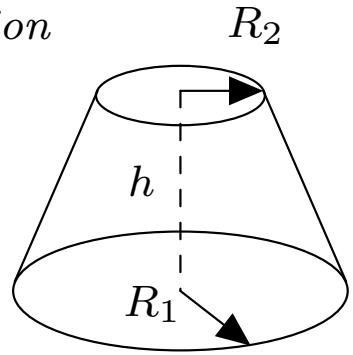


Langage : Pyhon

```
from  math import  pi
R2 = float(input("Petit rayon ?  "))
R1 = float(input("Grand rayon ?  "))
H = float(input("Hauteur ?  "))
VOL = H*pi/3*(R1**2+R2**2+R1*R2)
print("Volume du cône tronqué :  ", VOL)
```

Variables

1. **Réflexion** : *trouver une solution au problème*
   (depuis internet) $\Longrightarrow$ volume $= h\frac{\pi}{3}(R_1^2 + R_2^2 + R_1 R_2)$
2. **Élaboration de l'algorithme** : *Expliquer comment obtenir la solution*
   - 2.1 Lire le petit rayon et le sauvegarder sous le nom R2
   - 2.2 Lire le plus grand rayon et le sauvegarder sous le nom R1
   - 2.3 Lire la hauteur et la sauvegarder sous le nom H
   - 2.4 Calcul du volume et sauvegarde sous le sous VOL
   - 2.5 Écrire le résultat à l'écran
3. **Encodage** : *Choisir un langage et traduire l'algorithme*

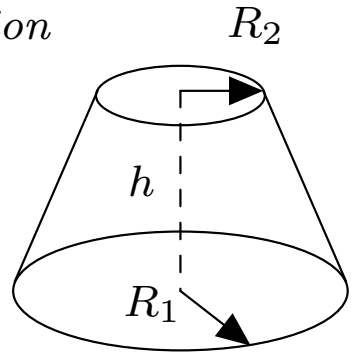


Langage : Pyhon

```
from math import pi
R2 = float(input("Petit rayon ?  "))
R1 = float(input("Grand rayon ?  "))
H = float(input("Hauteur ?  "))
VOL = H*pi/3*(R1**2+R2**2+R1*R2)
print("Volume du cône tronqué :  ", VOL)
```

Mots clés

1. **Réflexion** : *trouver une solution au problème*
   (depuis internet) $\Longrightarrow$ volume $= h\frac{\pi}{3}(R_1^2 + R_2^2 + R_1 R_2)$
2. **Élaboration de l'algorithme** : *Expliquer comment obtenir la solution*
   - 2.1 Lire le petit rayon et le sauvegarder sous le nom R2
   - 2.2 Lire le plus grand rayon et le sauvegarder sous le nom R1
   - 2.3 Lire la hauteur et la sauvegarder sous le nom H
   - 2.4 Calcul du volume et sauvegarde sous le sous VOL
   - 2.5 Écrire le résultat à l'écran
3. **Encodage** : *Choisir un langage et traduire l'algorithme*



Langage : Pyhon

```
from  math import  pi
R2 = float(input("Petit rayon ?  "))
R1 = float(input("Grand rayon ?  "))
H = float(input("Hauteur ?  "))
VOL = H*pi/3*(R1**2+R2**2+R1*R2)
print("Volume du cône tronqué :  ", VOL)
```

4. *(**Traduction** dans le langage du processeur : langage binaire)*

```
11010110101001011010101010111100111101011001010101010001110000100111
10001101010101000001010000101000101010100010010111011000110001001011O
10100100101010010010101010010101010101001001000010101010110100111O...
```

Trois différents niveaux de programmation

- **Langage machine (*processor level*) :** le plus bas niveau, tout est binaire

Trois différents niveaux de programmation

- **Langage machine (*processor level*) :** le plus bas niveau, tout est binaire
- **Langage assembleur (*intermediate level*) :** quelques commandes basiques

Trois différents niveaux de programmation

- **Langage machine (*processor level*) :** le plus bas niveau, tout est binaire
- **Langage assembleur (*intermediate level*) :** quelques commandes basiques
- **Langage haut niveau (*human level*) :** très proche de l'anglais parlé

Trois différents niveaux de programmation

- **Langage machine (*processor level*) :** le plus bas niveau, tout est binaire
- **Langage assembleur (*intermediate level*) :** quelques commandes basiques
- **Langage haut niveau (*human level*) :** très proche de l'anglais parlé

Un outil de traduction est nécessaire !

Trois différents niveaux de programmation

- **Langage machine (*processor level*) :** le plus bas niveau, tout est binaire
- **Langage assembleur (*intermediate level*) :** quelques commandes basiques
- **Langage haut niveau (*human level*) :** très proche de l'anglais parlé

Un outil de traduction est nécessaire !

Deux types de traducteurs

- **Compilateur**: le programme est traduit entièrement

1. **Analyse lexicologique :** vocabulaire et orthographe corrects ?
2. **Analyse syntaxique :** grammaire correcte ?
3. **Analyse sémantique :** signification de chaque instruction ?
4. **Création d'un objet optimal :** traduction optimisée du code en binaire

Trois différents niveaux de programmation

- **Langage machine (*processor level*) :** le plus bas niveau, tout est binaire
- **Langage assembleur (*intermediate level*) :** quelques commandes basiques
- **Langage haut niveau (*human level*) :** très proche de l'anglais parlé

Un outil de traduction est nécessaire !

Deux types de traducteurs

- **Compilateur**: le programme est traduit entièrement

1. **Analyse lexicologique :** vocabulaire et orthographe corrects ?
2. **Analyse syntaxique :** grammaire correcte ?
3. **Analyse sémantique :** signification de chaque instruction ?
4. **Création d'un objet optimal :** traduction optimisée du code en binaire

- **Interpréteur**: Instructions lues, traduites, exécutées **une à une immédiatement**

La langage Python

C'est quoi Python ?

- Langage de programmation haut niveau crée par **Guido Van Rossum** en 1991.
- Langage libre et *open source*
- Langage interprété orienté objet
- Langage extrêmement populaire depuis les années 2000 (cf. classement TIOBE) !

Programming Language	2023	2018	2013	2008	2003	1998	1993	1988
Python	1	4	8	7	12	25	19	-
C	2	2	1	2	2	1	1	1
Java	3	1	2	1	1	18	-	-
C++	4	3	4	4	3	2	2	5
C#	5	5	5	8	9	-	-	-
Visual Basic	6	15	-	-	-	-	-	-
JavaScript	7	7	11	9	8	21	-	-
SQL	8	251	-	-	7	-	-	-
Assembly language	9	13	-	-	-	-	-	-
PHP	10	8	6	5	6	-	-	-
Objective-C	19	17	3	44	53	-	-	-
Ada	25	28	18	19	15	9	6	3
Lisp	29	31	12	16	14	8	5	2
Pascal	189	147	15	18	99	12	3	14
(Visual) Basic	-	-	7	3	5	3	8	6

Pourquoi l'utiliser ? Pourquoi est il si populaire ?

- Libre et *open source* : énorme communauté
- Généraliste : grande variété d'applications
- Interprété et orienté objet : grande flexibilité
- Support de modules et packages : modularité et réutilisation de code
- Syntaxe simple : apprentissage rapide et productivité d'écriture de code
- Facile à lire : développement simplifié, coûts de maintenance réduits
- Esthétique, et souvent plus compact que les autre langages
- Standardisé sur différentes plateformes (Windows / MacOS / Linux)
- Langage de choix pour le calcul scientifique, les data sciences, le ML/IA, ...
- Extrêmement populaire dans le monde académique et dans les *tech companies*

"Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we're looking for more people with skills in this language.", `Peter Norvig, director of search quality at Google, Inc.`

"Python is fast enough for our site and allows us to produce maintainable features in record times, with a minimum of developers,", `Cuong Do, Software Architect, YouTube.com.`

"Programmers fall in love with Python because of the increased productivity it provides.", *Extract from Python executive summary*.

Hmmm... Et donc ?
Les désavantages ?

- Faible développement mobile : rares applications smartphone
- Langage interprété : codes intensifs plus lents qu'en langage compilé

... Mais,

- De nombreux packages ont été optimisés et s'exécutent à la vitesse de C
- De nombreux packages permettent de compiler du code Python *just in time* (JIT) ou de précompiler du code
- Pour les scientifiques, **NumPy/SciPy** fournissent un environnement comparable et plus compétitif que des applications commerciales coûteuses telles que MatLab™

Charm the snake

The snake charmer by Zachsmithson

Anaconda – Python distribution

- Large-scale data processing
- Predictive analytics
- Scientific computing
- Easy-to-use package manager `conda`

Spyder – **S**cientific **PY**thon **D**evelopment **E**nvi**R**onments

- Interactive environment
- Edition, testing, debugging, introspection
- Matlab-Like interface

Jupyter – Open-source notebook

- Web application
- Create and share documents
- Handle live code, text and LaTeX equations

Introduction à Jupyter Notebook

- Application web
- Facilite la création et partage de documents
- Gère du code *live*, du texte et des équations LaTeX

Important : Le lancement de Jupyter s'accompagne du lancement d'un kernel IPython localement (fenêtre de commande ms-dos sous windows). Si vous fermez cette fenêtre, Jupyter ne fonctionnera plus correctement

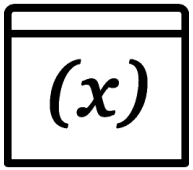
Les objets sous Python

Photography by Joe Lodge

Qu'est ce qu'un `objet` sous Python ?

- Un *objet* est un morceau de programme contenant des données
- A chaque objet peut être associé un certain nombre d'**attributs** et de fonctions spécialisées appelées **méthodes** qui agissent sur les objets
- Chaque objet est défini par son **identité**, son **type**, et sa **valeur**

Identité	• Ne change jamais une fois que l'objet a été crée • Représente "l'adresse" en mémoire • L'opérateur `is` compare l'identité de deux objets • La fonction `id`() retourne un entier représentant son identité
Type	• Ne change jamais une fois que l'objet a été crée • Détermine les opérations que l'objet supporte • Détermine les attributs et méthodes héritées par l'objet • La fonction `type`() retourne le type d'un objet
Valeur	• Peut changer, mais pas toujours ! • Les objets dont la valeur peut changer sont appelés **muable** • Les objets dont la valeur ne peut pas changer sont appelés **immuable** • La **mutabilité** d'un objet est déterminée par son type



Comment déclarer une variable sous Python ?

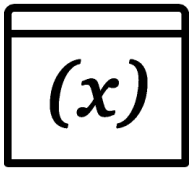
- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

Interpréteur

```
In  : a = 1 ↵
```

Espace des variables

Espace des objets



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

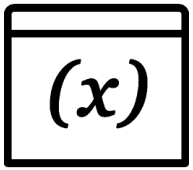
Interpréteur

```
In  : a = 1 ↵
```

Espace des variables

Espace des objets

ref	0	1
type	int	
id	0001	
meth	...	
attr	...	



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

Interpréteur

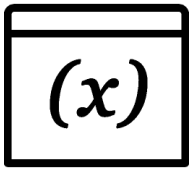
```
In  : a = 1 ↵
```

Espace des variables

a

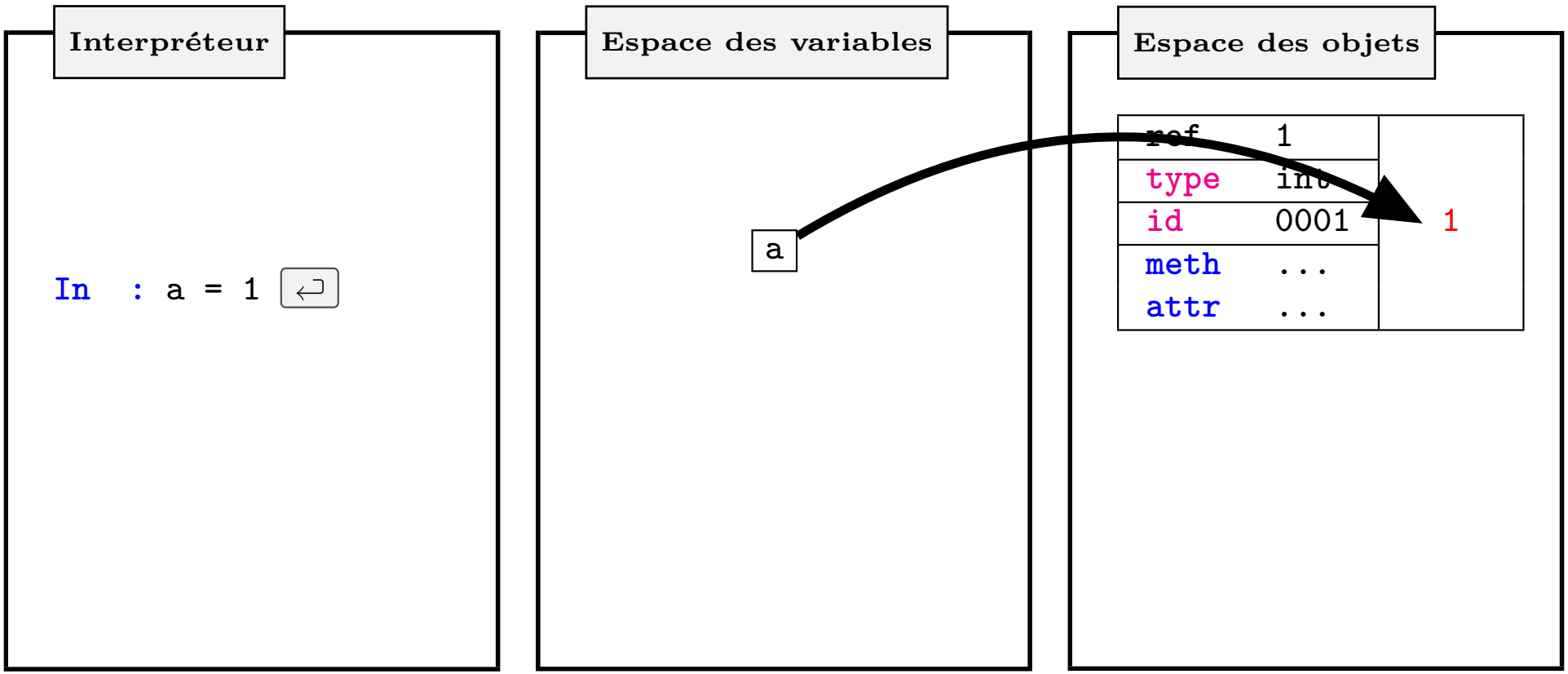
Espace des objets

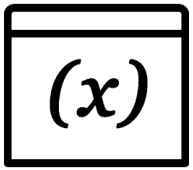
ref	0	1
type	int	
id	0001	
meth	...	
attr	...	



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement





Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
```

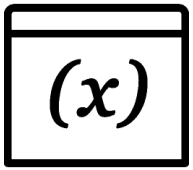
Espace des variables

a

Espace des objets

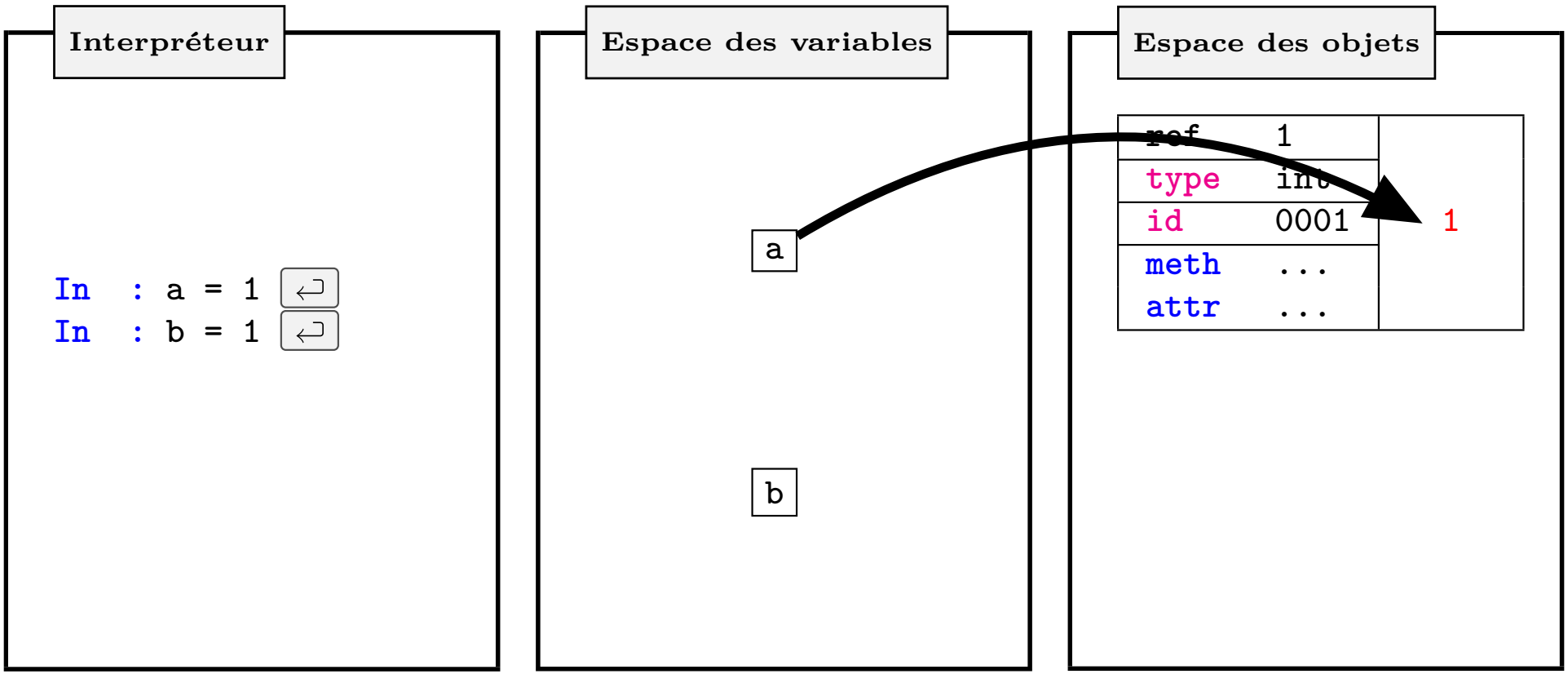
ref	1
type	int
id	0001
meth	...
attr	...

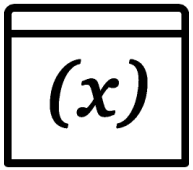
1



Comment déclarer une variable sous Python ?

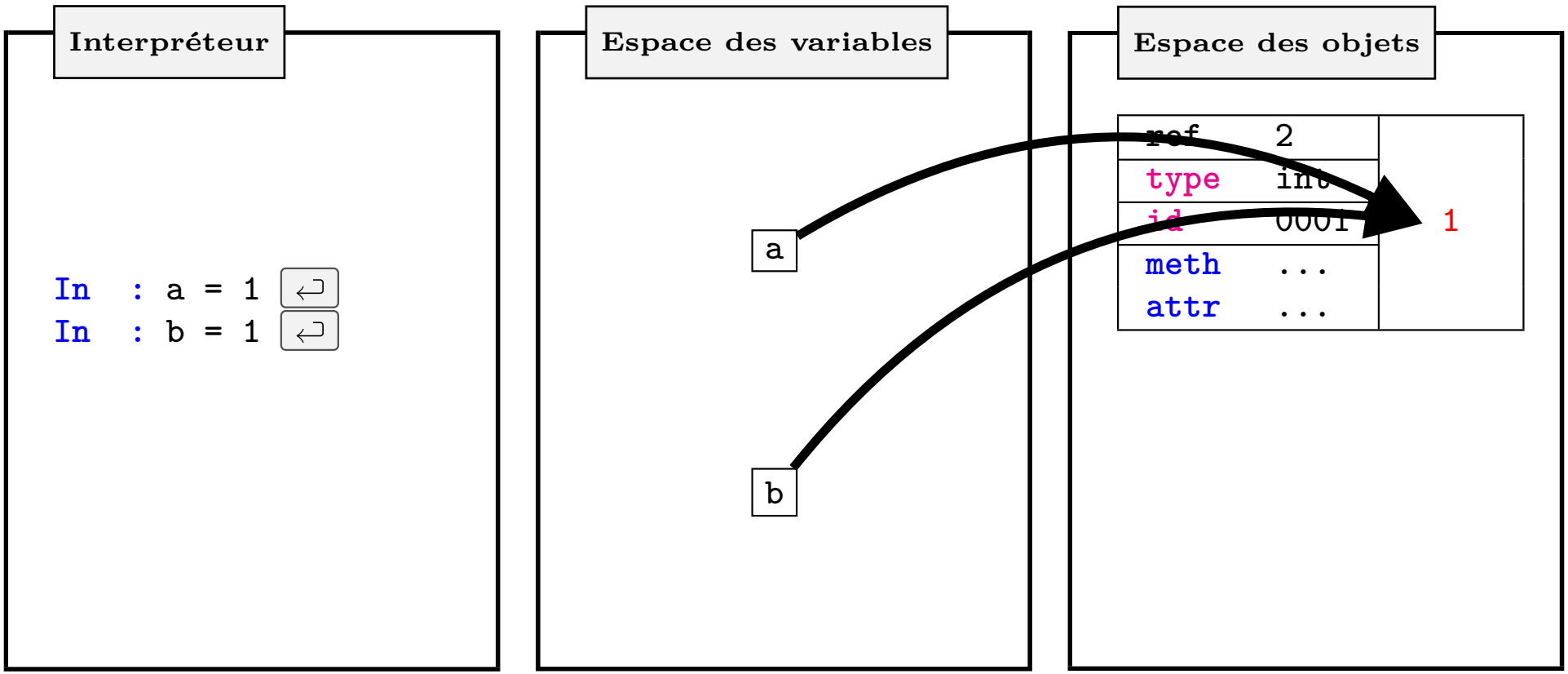
- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

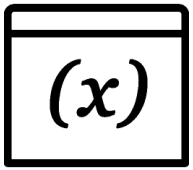




Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement





Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
```

Espace des variables

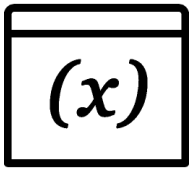
a

b

Espace des objets

ref	2
type	int
id	0001
meth	...
attr	...

1



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
```

Espace des variables

a

b

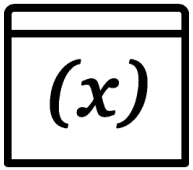
Espace des objets

ref	2
type	int
id	0001
meth	...
attr	...

1

ref	0
type	str
id	0108
meth	...
attr	...

"abc"



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
```

Espace des variables

a

b

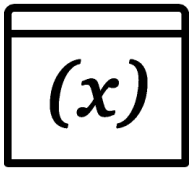
Espace des objets

ref	1
type	int
id	0001
meth	...
attr	...

1

ref	0
type	str
id	0108
meth	...
attr	...

"abc"



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
```

Espace des variables

a

b

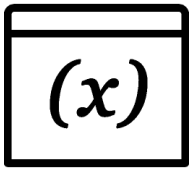
Espace des objets

ref	1
type	int
id	0001
meth	...
attr	...

1

ref	1
type	str
id	0108
meth	...
attr	...

"abc"



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
In  : a = b ↵
```

Espace des variables

a

b

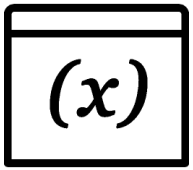
Espace des objets

ref	1
type	int
id	0001
meth	...
attr	...

1

ref	1
type	str
id	0108
meth	...
attr	...

"abc"



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
In  : a = b ↵
```

Espace des variables

a

b

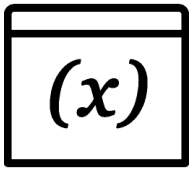
Espace des objets

ref	0
type	int
id	0001
meth	...
attr	...

1

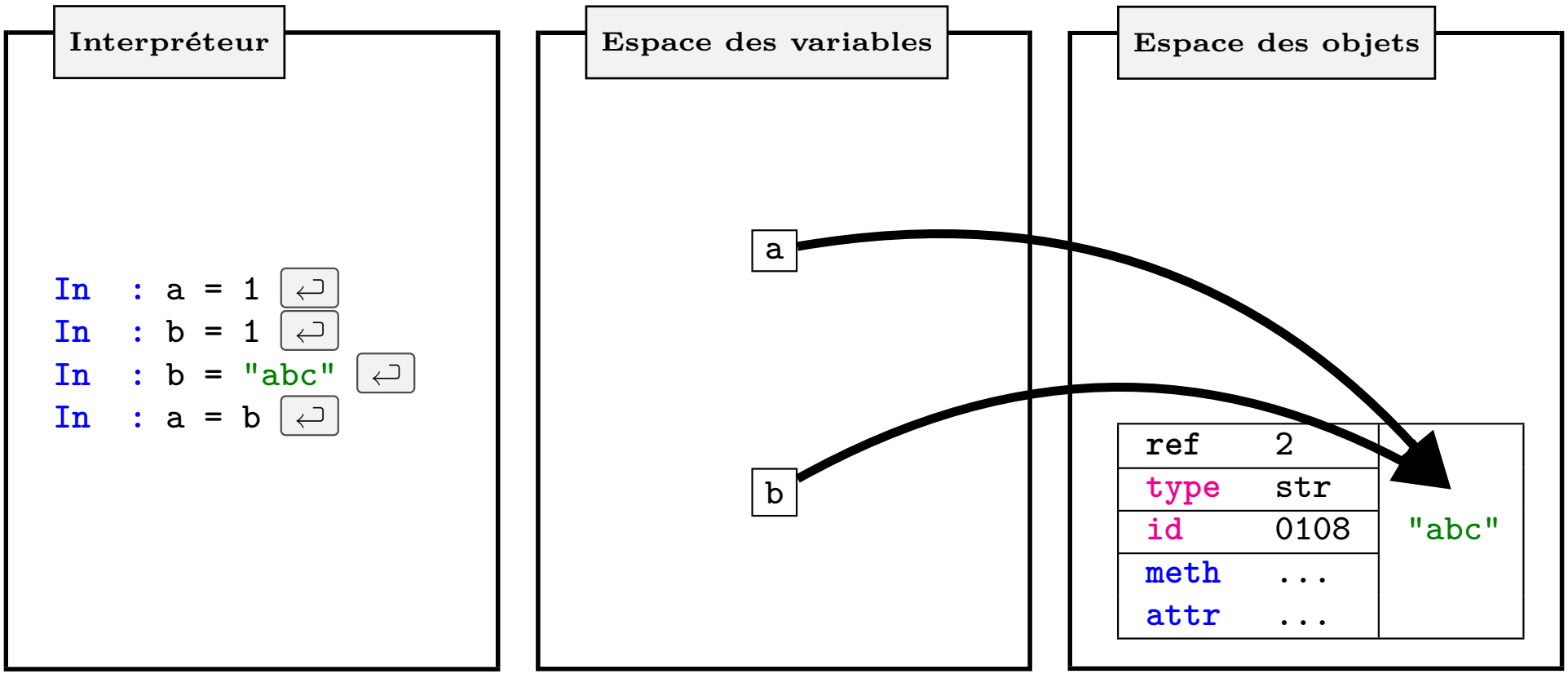
ref	1
type	str
id	0108
meth	...
attr	...

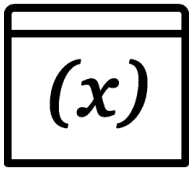
"abc"



Comment déclarer une variable sous Python ?

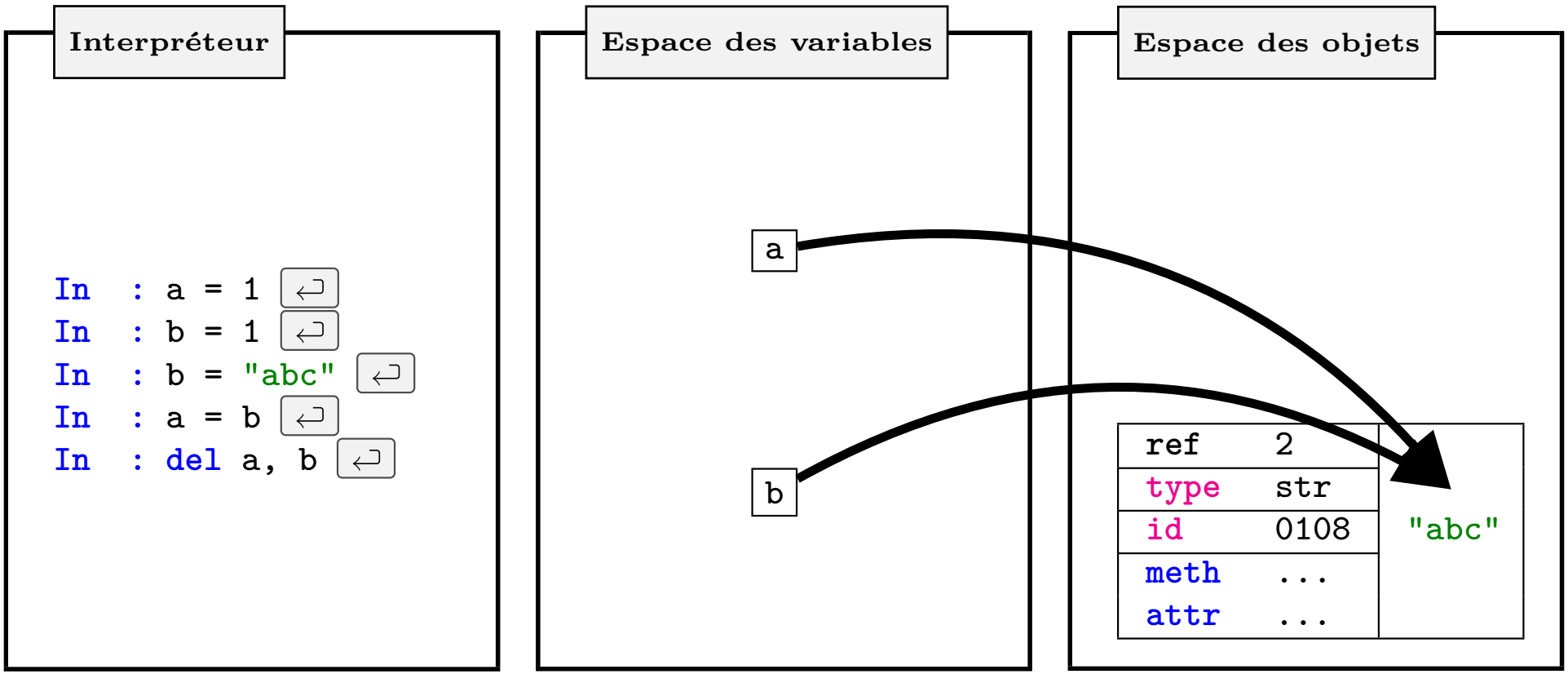
- Caractère "=" pour référencer un objet
- Type assigné dynamiquement
- Lorsqu'un objet n'est plus référencé : le "***Garbage collector***" libère la mémoire

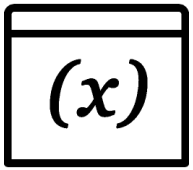




Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement
- Lorsqu'un objet n'est plus référencé : le "***Garbage collector***" libère la mémoire





Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement
- Lorsqu'un objet n'est plus référencé : le "***Garbage collector***" libère la mémoire

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
In  : a = b ↵
In  : del a, b ↵
```

Espace des variables

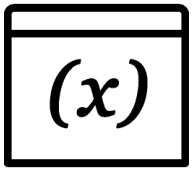
a

b

Espace des objets

ref	0
type	str
id	0108
meth	...
attr	...

"abc"



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement
- Lorsqu'un objet n'est plus référencé : le "***Garbage collector***" libère la mémoire

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
In  : a = b ↵
In  : del a, b ↵
```

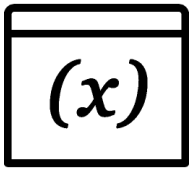
Espace des variables

a

b

Espace des objets

ref	0	"abc"
type	str	
id	0108	
meth	...	
attr	...	



Comment déclarer une variable sous Python ?

- Caractère "=" pour référencer un objet
- Type assigné dynamiquement
- Lorsqu'un objet n'est plus référencé : le "***Garbage collector***" libère la mémoire

Interpréteur

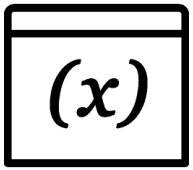
```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
In  : a = b ↵
In  : del a, b ↵
```

Espace des variables

Espace des objets

ref	0
type	str
id	0108
meth	...
attr	...

"abc"



Comment déclarer une variable sous Python ?

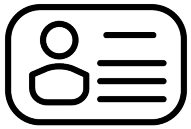
- Caractère "=" pour référencer un objet
- Type assigné dynamiquement
- Lorsqu'un objet n'est plus référencé : le "***Garbage collector***" libère la mémoire

Interpréteur

```
In  : a = 1 ↵
In  : b = 1 ↵
In  : b = "abc" ↵
In  : a = b ↵
In  : del a, b ↵
```

Espace des variables

Espace des objets



Quelles sont les règles de nommage ?

- Peut contenir les lettres majuscules et minuscules `[A-Z, a-z]` et les chiffres `[0-9]`
- Tous les autres caractères sont interdis à l'exception du caractère `"_"`
- Ne peut pas commencer par un chiffre
- Ne peut pas être l'un des mots suivants, réservés par Python :

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

Les mêmes règles s'appliquent aux noms de fichiers des scripts Python !!!

Note : *Choisissez des noms de variables explicites !*
Préférez un nom explicite comme `gamma` *plutôt que* `g`

Les types numériques sous Python

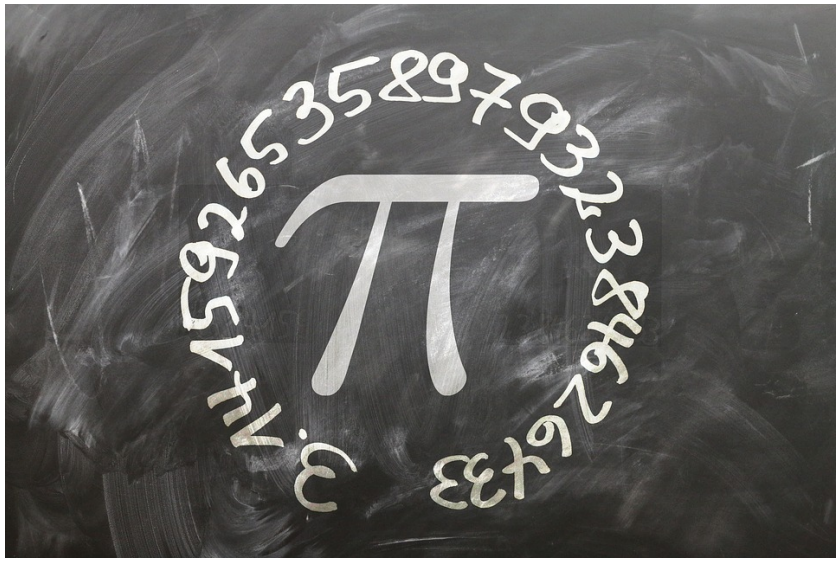


Image from Pixabay (Geralt)

Liste des types numériques

Les types numériques natifs sont :

- **Le type** `int` utilisé pour représenter les nombres entiers
  - Précision arbitraire : uniquement limitée par la mémoire disponible
  - constructeur éponyme `int`()
- **Le type** `float` utilisé pour représenter les nombres réels
  - précision dépendante du système
  - constructeur éponyme `float`()
- **Le type** `complex` utilisé pour représenter les nombres complexes
  - composé de deux `float` : partie réelle/partie imaginaire
  - constructeur éponyme `complex`() : `complex`([real[, imag]])
- **Le type** `bool` utilisé pour représenter les 2 booléens `True` et `False`
  - sous type de `int`
  - `False` correspond à l'entier 0, `True` correspond à l'entier 1
  - constructeur éponyme `bool`()

Important : Les objets de ces différents types numériques sont des objets ***immuables***
Les méthodes qui les modifient en modifient une copie à la place

Operators applied to `complex`, `float` and `integers` objects			
	Operation	**Operator**	**Description**
	Addition	`a + b`	Sum of **a** and **b**
	Subtraction	`a - b`	Subtraction of **a** and **b**
	Multiplication	`a * b`	Multiplication of **a** by **b**
	Division	`a / b`	Division of **a** by **b**
	Power	`a**b` ou `pow(a, b)`	**a** to the power of b
	Modulus	`abs(a)`	Modulus of **a**
*	Division - remainder	`a%b`	Remainder of **a / b**
*	Division - integer part	`a//b`	Integer part of **a / b**
*	Full division	`divmod(a, b)`	Return the tuple `(a//b, a%b)`
	Conjugate	`a.conjugate()`	Return the conjugate of **a**
*	Integer conversion	`int(a)`	Convert **a** to `int` (truncation)
*	Float conversion	`float(a)`	Convert **a** to `float`
	Complex conversion	`complex(a)`	Convert **a** to `complex`
* These operations are not supported by `complex` objects			

Illustration : Les types numériques

```
# Déclaration :
>> a = 2              # Crée l'entier 2, a réfère cet objet
>> b = 2.0            # Crée le réel 2.0, b réfère cet objet
>> c = 2+2j           # Crée le complexe 2+2j, c réfère cet objet

# Typage dynamique :
>> type(a + b)        # Opération impliquant plusieurs types numériques
float
>> type(a + b + c)    # donne le type le plus fort au résultat
complex

# Précision :
>> d = a**512         # Objets int de précision illimitée !
>> print(d)
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216
>> b*d                # La conversion peut conduire a une perte de précision
2.6815615859885194e+154
```

Illustration : Notion d'héritage

```
# Exemple des float
>> a = 1.5
>> a.as_integer_ratio() # La méthode as_integer_ratio()...
(3, 2)                  # ... parle d'elle même

# Exemple des complex
>> b = 1 + 2j
>> c = complex(1, 2)    # Écriture alternative de complexes
>> c.conjugate()        # Méthode conjugate() héritée de la classe complex
(1-2j)
>> c.real               # Attribut real hérité de la classe complex
1
>> c.imag               # Attribut imag hérité de la classe complex
2
```

- On utilise le point (.) sous Python pour accéder à l'héritage d'un objet
- Un **attribut** est une caractéristique de l'objet : son appel se fait **sans parenthèses**
- Une **méthode** est une fonction qui s'applique à l'objet : son appel se fait **avec des parenthèses et d'éventuels arguments**

Notebook - S01E01

Les structures de données

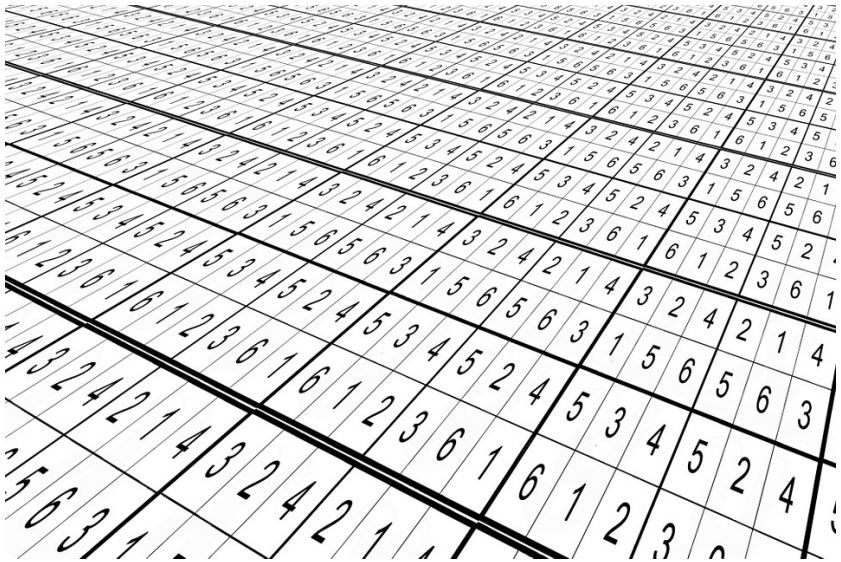


Image from Pixabay (Geralt)

Liste des types séquentiels

Les types séquentiels natifs sont :

- **Le type `string`** : séquence ordonnée *immuable* de caractères
  - type particulier de séquence appelé chaîne de caractères
  - déclaration avec `' '` ou `" "` ou `''' '''` ou `""" """`
  - conversion avec le constructeur `str()`
- **Le type `list`** : séquence ordonnée *muable* d'éléments
  - déclaration avec `[]`
  - conversion avec le constructeur `list()`
- **Le type `tuple`** : séquence ordonnée *immuable* d'éléments
  - déclaration avec `()`
  - conversion avec le constructeur `tuple()`
- **Le type `range`** : séquence ordonnée *immuable* d'entiers
  - déclaration avec le constructeur `range()`
  - couramment utilisée dans les boucles `for` en tant qu'itérateur

Le "str" : séquence ordonnée immuable de caractères

```
>> s1 = "Hello World "
>> s2 = "7"
>> type(s1), type(s2)                    # (str, str)
>> print(s1, s2)                         # "Hello World  7"
```

Objet immuable : ne peut pas changer...

```
>> s1[1] = "a"
TypeError: 'str' object does not support item assignment
```

Répétition/concaténation avec +/*

```
>> s1 + s2          # Concaténation : "Hello World 7"
>> s1 * 3           # Répétition : "Hello World Hello World Hello World "
```

Nombreuses méthodes qui agissent sur une copie...

```
>> s1.split()                          >> s1.upper()
["Hello", "World"]                     "HELLO WORLD "
>> s1.replace("World", "Bro")          >> s1.center(30)
"Hello Bro "                           "          Hello World          "
```

Le "str" : Focus sur le formatage

```
>> pi = 3.14159
```

Multiple arguments de print [à éviter...]

```
>> print("Pi equals", round(pi, 2), ", more precisely", pi)
Pi equals 3.14 , more precisely 3.14159
```

La concaténation avec + [à éviter...]

```
>> print("Pi equals " + str(round(pi, 2)) + ", more precisely " + str(pi))
Pi equals 3.14, more precisely 3.14159
```

La méthode format

```
>> print("Pi equals {:.2f}, more precisely {}".format(pi, pi))
Pi equals 3.14, more precisely 3.14159
```

Les f-strings [Python >= 3.6]

```
>> print(f"Pi equals {pi:.2f}, more precisely {pi}")
Pi equals 3.14, more precisely 3.14159
```

Digression sur la fonction input

```
>> a = input("Type a number : ")
Type a number : 5

>> type(a)                                  # .... retourne TOUJOURS un str...
str                                         # ... qui peut être converti

>> a = int(a)                               # ... en int...
>> a = float(a)                             # ... en float...
>> a = complex(a)                           # ... ou en complex

# La conversion peut se faire in place :
>> a = int(input("Type a number : "))
Type a number : 5

>> type(a)
int
```

La [list,] : séquence ordonnée muable d'éléments

```
>> beers = ["Guinness", "Chouffe", "86"]
>> type(beers)                  # list
>> print(beers)                 # ["Guinness", "Chouffe", "86"]
```

Objet muable : ajout, suppression, modification possible...

```
>> beers[2] = "DeUs"            # ["Guinness", "Chouffe", "DeUs"]
```

Chaque élément peut avoir un type différent...

```
>> mix1 = [2.1, "abc"]
>> mix2 = [[10, "cba"], 1]
```

Répétition/concaténation avec +/* comme pour les objets str...

```
>> mix1 + mix2                  # [2.1, "abc", [10, "cba"], 1]
>> mix1 * 2                     # [2.1, "abc", 2.1, "abc"]
```

Nombreuses méthodes qui agissent in-place...

```
>> beers.sort()                 # beers = ["Chouffe", "DeUs", "Guinness"]
>> beers.pop(1)                 # beers = ["Chouffe", "Guinness"]
```

Le (tuple,) : séquence ordonnée immuable d'éléments

```
>> days = ("monday", "tuesday", "wednesday")
>> type(days)                    # tuple
>> print(days)                   # ("monday", "tuesday", "wednesday")
```

Objet immuable : ajout, suppression, modification impossible...

```
>> days[1] = "sunday"            # lève une exception TypeError
```

Chaque élément peut avoir un type différent...

```
>> mix1 = (2.1, "abc")
>> mix2 = ([10, "cba"], 1)
```

Répétition/concaténation avec +/* comme pour les objets str...

```
>> mix1 + mix2                   # (2.1, "abc", [10, "cba"], 1)
>> mix1 * 2                      # (2.1, "abc", 2.1, "abc")
```

Peu de méthodes héritées...

```
>> bestbeers.index("monday")   # retourne l'indice : 0
>> bestbeers.count("monday")   # retourne le nombre d'occurences : 1
```

Le range(n, m, s) : séquence immuable d'int de n à $m-s$ par pas de s

```
>> r = range(1, 11, 2)              # séquence 1, 3, 5, 7, 9
>> type(r)                          # range
>> print(r)                         # range(1, 11, 2)
```

Objet immuable : ajout, suppression, modification impossible...

```
>> r[1] = 10                        # lève une exception TypeError
```

Quelques méthodes et attributs héritées...

```
>> r.start, r.stop, r.step          # attributs : (0, 5, 1)
```

Arguments n et s optionnels (n =0 et s =1 par défaut)

```
>> range(5)                         # séquence 0, 1, 2, 3, 4
>> range(3, 7)                      # séquence 3, 4, 5, 6
>> range(10, 4, -2)                 # séquence 10, 8, 6
```

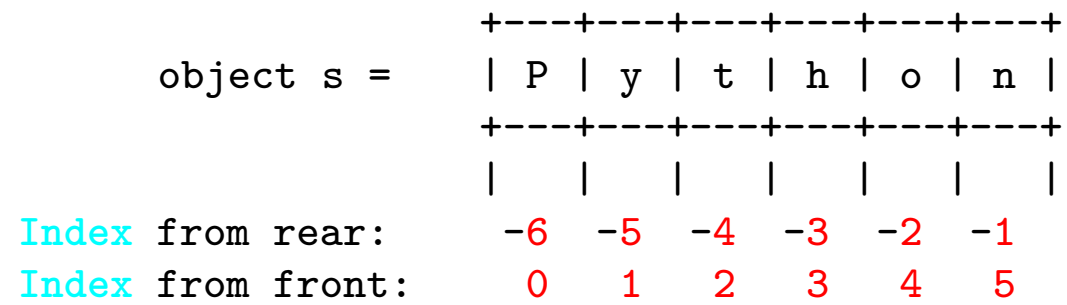
Avantage : faible empreinte mémoire (seuls n, m, et s sont stockés)

```
>> from sys import getsizeof        # détermine la taille d'un objet
>> getsizeof(range(10000))          # 48 b au lieu de 80 kb
```

Notebook - S01 complète

L'Indexation sous Python

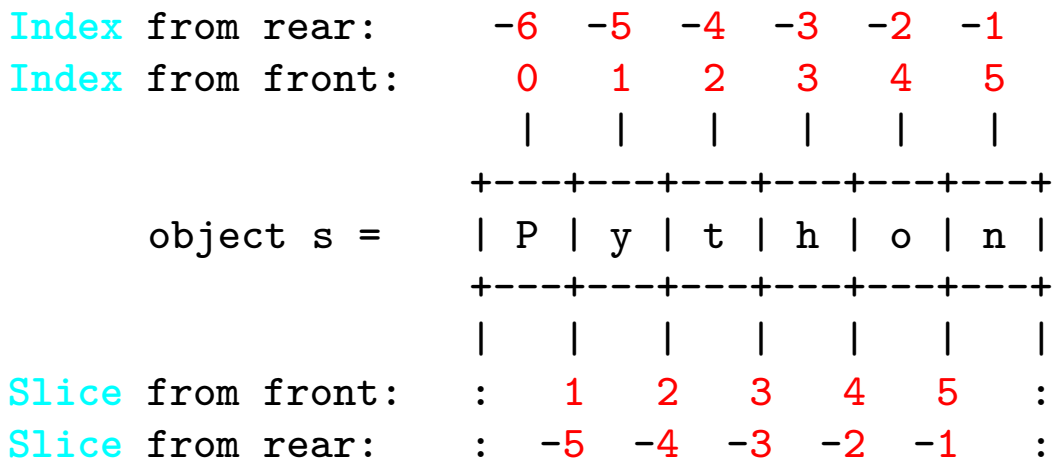
- Comme dans la plupart des langages, Python utilise `[ ]` pour l'indexation
- Comme dans la plupart des langages, Python utilise une indexation basée sur 0
- Python utilise également des indices négatifs : le dernier élément d'un conteneur itérable est l'élément d'indice -1




```
>> a = [3, 5, 7, 9]
>> a[0]
3
>> a[-1]
9
>> a[-2]
7
```

```
>> a = "Hello World!"
>> a[0]
H
>> a[-1]
!
>> a[-2]
d
```

Le slicing : un moyen rapide et méthodique pour accéder aux données



Syntaxe :

```
>> s[1:]      # ython
>> s[:5]      # Pytho
```

s[<start>:<end>:<step>]

```
>> s[-5:-2]   # Pyth
>> s[1:-1:2]  # yh
```

```
>> s[::-1]    # nohtyP
>> s[::-2]    # nhy
>> s[1::-2]   # y
>> s[2::-2]   # tP
```

Définition externe d'un slice : slice(start, stop[, step])

```
>> myslice = slice(None, None, -2)     # None pour valeur par défaut
>> s[myslice]                          # Équivalent à s[::-2]
```

Notebook - S02 complète

Les autres types natifs principaux

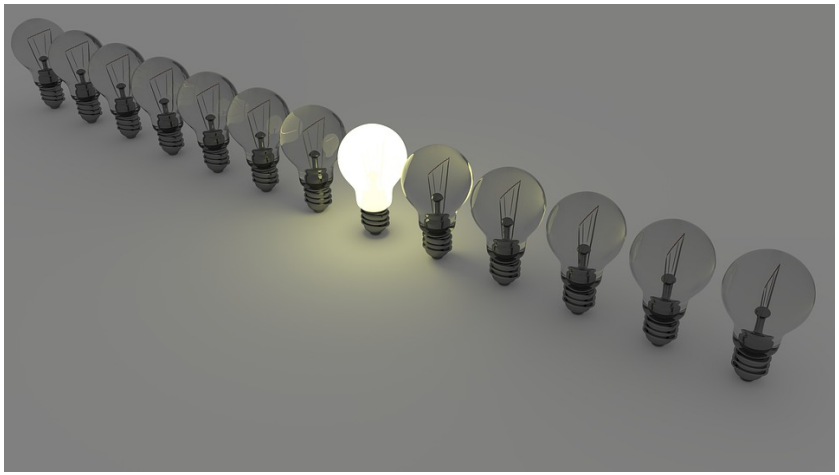
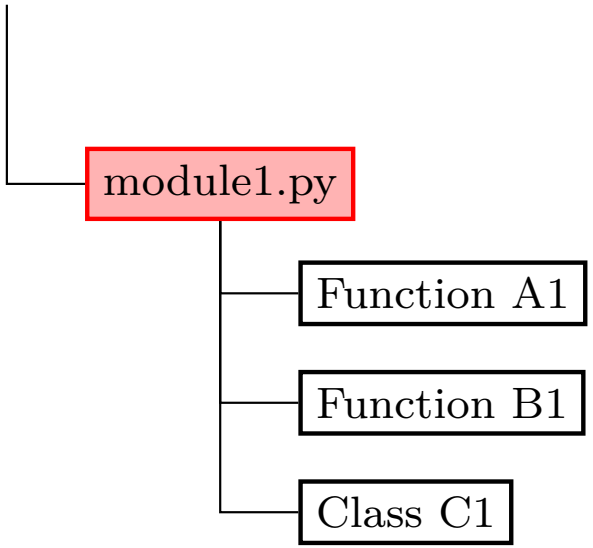


Image from Pixabay (Stevepb)

- **Les booléens** : `True` et `False`
  - utilisés dans les tests
  - sous type de `int` (0 et 1)
- **L'objet null**, unique est nommé `None`
- **Les séquences binaires** utilisées pour manipuler les données binaires
  - le type `bytes` : séquence *immuable* d'octets
  - le type `bytearray` : équivalent *muable* des objet `bytes`
  - le type `memoryview` : utilisé pour accéder aux données internes d'un objet
- **Les fonctions** (vues plus tard)
- **Les classes**, **instances de classes** et **méthodes** (vues plus tard)
- **Les modules** que nous allons traiter tout de suite

Qu'est ce qu'un `Module` ?

- Un fichier `*.py` contenant des définitions et instructions (fonctions, classes, ...)
- Le nom du fichier est celui du module avec le suffixe `.py`
- Python fournit de base une bibliothèque de modules standards

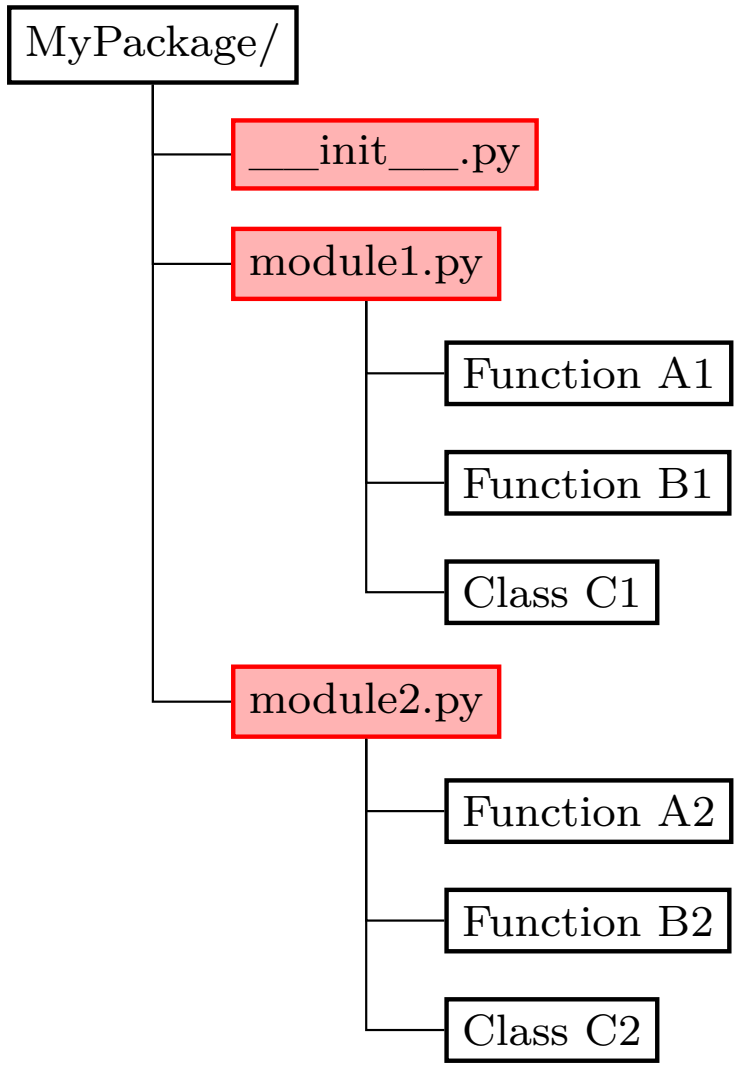


Qu'est ce qu'un `Module` ?

- Un fichier `*.py` contenant des définitions et instructions (fonctions, classes, ...)
- Le nom du fichier est celui du module avec le suffixe `.py`
- Python fournit de base une bibliothèque de modules standards

Qu'appelle-t-on un `Package` ?

- Une collection de modules organisée hiérarchiquement
- Un package contient toujours un fichier `__init__.py`
- `__init__.py` assure la distinction entre un package et un simple répertoire rempli de scripts
- Les packages peuvent être imbriqués jusqu'à n'importe quelle profondeur
- Chaque package imbriqué doit contenir son propre fichier `__init__.py`



Python vient avec de nombreux modules appelés modules standards :

- Interactions avec l'os ou le système : `sys`, `os`, `time`, ...
- Protocoles internet : `urllib`, `sockerserver`, ...
- Exécution simultanée : `threading`, `multiprocessing`, ...
- Mathématiques : `math`, `statistics`, ...
- ...

L'immense communauté développe activement de nombreux autres modules :

- Jeux vidéos : `pygame`, `pyglet`, ...
- Multimédia : `PIL`, `cv2`, ...
- General User Interface (GUI) : `wxPython`, `PyGtk`, ...
- Optimisation : `Cython`, `numba`, ...
- ...

Les scientifiques ne sont pas en reste :

- Calcul numérique : `numpy`
- Calcul scientifique : `scipy`
- Calcul symbolique : `sympy`
- Librairie graphique : `matplotlib`
- ...

Comment utiliser ces modules ?

- En les important dans l'espace des variables !
- Le mot clé `import` permet de charger un module
- Les mots clés `from`, `as`, le *splat operator* `*` permettent une importation plus pratique

Illustration

```
>> import math                  # Importe le module math
>> math.cos(math.pi)            # Retourne -1.0

>> import math as mt            # math est maintenant attaché au nom...
>> mt.cos(mt.pi)                # ...mt dans l'espace local des variables

>> from math import *           # Importe toutes les classes et fonctions
>> cos(pi)                      # cos et pi viennent du module math

>> from math import cos, pi     # Importe seulement la fonction cos...
>> cos(pi)                      # ... et la constante pi

>> import sys, math             # Importe les modules sys et math
```

Quelques mots sur les bonnes pratiques ...

- Elle dépendent généralement de ce dont vous avez besoin dans le module !
- Besoin d'une seule fonction : vous pouvez importer uniquement cette fonction
- De manière générale, gardez un lien vers les modules utilisés dans l'espace local !

Évitez les imports globaux comme : `from math import *`

- Surcharge l'espace des variables et restreint les noms de variables disponibles
- Provoque des conflits si des objets homonymes existent dans des modules différents

```
>> from numpy import *                    # Importe tout du module numpy
>> cos.__class__
<type 'numpy.ufunc'>
>> from math import *                     # Importe tout du module math
>> cos.__class__                          # la méthode cos de numpy...
<type 'builtin_function_or_method'>       # ... a été surchargée
```

Le type ndarray et le calcul scientifique

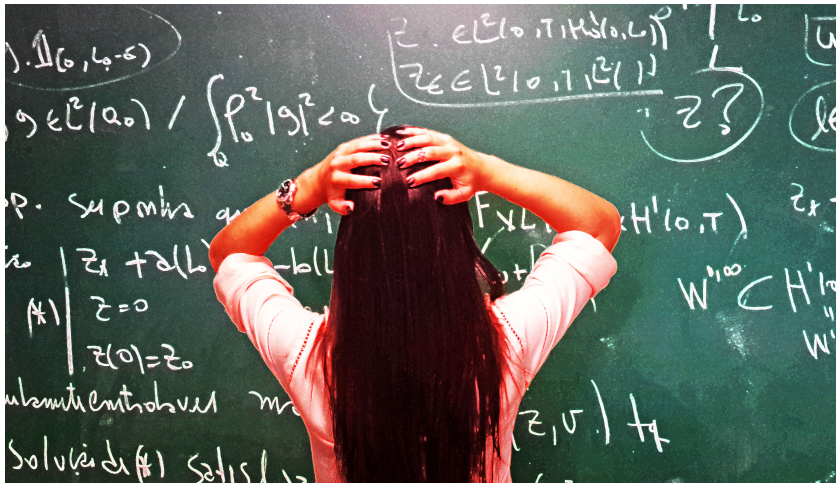


Image under Creative Common licence

Opérations vectorielles : indirectes avec les types de base (`list`, `tuple`, `dict`, `set`, ...) :

```
>> [1, 2] * 2                                  # Répétition
[1, 2, 1, 2]
>> [1, 2] + [3, 4]                             # Concaténation
[1, 2, 3, 4]
>> [1, 2] / 2
TypeError: unsupported operand type(s) for /: list and int
>> [1, 2] ** 2
TypeError: unsupported operand type(s) for **: list and int
```

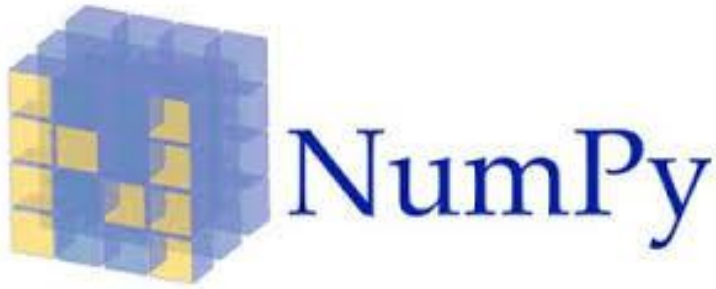
Module spécifique Numpy : dédié au calcul numérique

- Fournit le type `ndarray` (n-dimensional array)
- Fournit le support de calculs vectoriels et matriciels (*math.* ou *'element-wise'*)
- Fournit la méthode `array` classiquement utilisée pour déclarer des *matrices*

Illustration

```
>> import numpy as np
>> myarray1D = np.array([1, 3, 2, 7])
>> myarray1D**2                                # Puissance élément à élément
array([1, 9, 4, 49])
```

Introduction à Numpy



Syntaxe de la fonction array

```
>> import numpy as np
```

Les éléments constituant les ndarray sont du même type : typage dynamique

```
>> np.array([0, 1.])                          # Convertit la liste [0,1.]...
array([0., 1.])                               # ... en ndarray de float
```

Argument dtype force le type - Accepté par les autres constructeurs

```
>> np.array([0, 1.], dtype=int)               # array([0, 1])
>> np.array([0, 1.], dtype=float)             # array([0., 1.])
>> np.array([0, 1.], dtype=complex)           # array([0.+0.j, 1.+0.j])
```

Conversion de séquences imbriquées en ndarray 2D

```
>> np.array([[0, 1], [2, 3]])                 # array([[0, 1],
                                              #        [2, 3]]
>> np.array([[1, 2.], (1 + 1j, 3.), range(2)])
array([[ 1.+0.j, 2.+0.j],                     # Les dimensions des séquences
       [ 1.+1.j, 3.+0.j],                     # doivent être égales...
       [ 0.+0.j, 1.+0.j]])
```

Méthodes et attributs des ndarray: Quelques exemples

```
import numpy as np
```

Manipulation de forme...

```
>> c = np.arange(12)              # array([0, 1, 2, 3, ..., 9, 10, 11])
>> c = c.reshape(3, 4)
>> print(c)
array([[0,  1,  2,  3],
       [4,  5,  6,  7],          # Peut s'écrire en une ligne :
       [8,  9, 10, 11]])         # c = np.arange(12).reshape(3, 4)
```

Attributs importants...

```
>> c.shape                        # (3, 4)
>> c.real, c.imag                 # parties réelle et imaginaire
```

Les méthodes opèrent généralement sur toutes des dimensions par défaut...
Mais axis permet de specifier un axe particulier

```
>> c.mean()                       # 5.5
>> c.mean(axis=0)                 # array([ 4.,  5.,  6.,  7.])
>> c.mean(axis=1)                 # array([ 1.5,  5.5, 9.5])
>> c.min(), c.max(), c.sum()      # 0, 11, 66
```

Some Numpy array attributes	
ndarray.flags	Information about the memory layout of the array
ndarray.shape	Tuple of array dimensions
ndarray.strides	Tuple of bytes to step in each dimension when traversing an array
ndarray.ndim	Number of array dimensions
ndarray.size	Number of elements in the array
ndarray.itemsize	Length of one array element in bytes
ndarray.dtype	Data-type of the array's elements
ndarray.T	Transposed array
ndarray.real	Real part of the array
ndarray.imag	Imaginary part of the array

Some of the Numpy array methods	
ndarray.copy()	Return a copy of the array
ndarray.reshape()	Return array with a new shape
ndarray.transpose()	Return a view of the array with axes transposed
ndarray.repeat()	Repeat elements of an array
ndarray.sort()	Sort an array, in-place
ndarray.argsort()	Return the indices that would sort this array
ndarray.argmin()	Return indices of the min. values along given axis
ndarray.min()	Return the minimum along given axis
ndarray.conj()	Complex-conjugate all elements
ndarray.sum()	Return the sum over given axis
ndarray.mean()	Return the average along given axis
ndarray.std(])	Return the standard deviation along given axis

Les constructeurs ndarray: quelques exemples

```
>> import numpy as np
```

Les séquences numériques :

```
>> np.arange(0, 10, 2)                    # range ~ arange(start, stop, step)
array([0, 2, 4, 6, 8])
>> np.linspace(0, 1, 5)                   # linspace(start, stop, N)
array([0. , 0.25, 0.5 , 0.75, 1.])
```

Initialisation de ndarray :

```
>> np.zeros(3)                            # argument entier => 1d
array([0., 0., 0.])
>> np.zeros((2, 3))                       # argument tuple => Nd
array([[0.,  0.,  0.],
       [0.,  0.,  0.]])
>> np.zeros((2, 3), dtype=int)
array([[0,  0,  0],
       [0,  0,  0]])
```

- Toutes les routines de création de ndarray peuvent prendre dtype en argument
- Numpy propose une grande collection de routines pour la construction de ndarray

Arrays of ones and zeros	
`empty()`	New array of given shape and type without initialization
`empty_like(a)`	New empty array with same shape as the array a
`zeros()`	New array of given shape and size filled with 0
`zeros_like(a)`	New zeros array with same shape as the array a
`ones()`	New array of given shape and size filled with 1
`ones_like(a)`	New ones array with same shape as the array a
`full()`	New array of given shape and size filled val
`full_like(a)`	New full array with same shape as the array a
`eye()`	Array with ones on a diagonal and zeros elsewhere
`identity()`	Identity array (special case of `eye()`)
Arrays from existing data	
`array()`	Create a new array
`copy(a)`	Return an array copy of a
Arrays from numerical ranges	
`arange()`	Evenly spaced values over a given interval
`linspace()`	Evenly spaced values over a given interval
`logspace()`	Evenly spaced values over a given interval on log scale
`meshgrid()`	Create array from coordinate vectors
Building classical matrices	
`diag()`	Extract a diagonal or construct a diagonal array
`tri()`	Tridiagonal array
`tril()`	Lower triangular array
`triu()`	Upper triangular array
`vander()`	Vandermonde array

Opérateurs et ndarray

```
>> import numpy as np
>> a = np.array([[1, 2], [3, 4]])
>> b = np.array([[5, 6], [7, 8]])

# Les opérateurs agissent élément par élément (element-wise)
>> a*(a + b)                                    # Opérateurs arithmétiques
array([[6, 16], [30, 48]])
>> a > b                                        # Opérateurs de comparaison
array([[False, False], [False, False]])
>> np.logical_and(a>1, a<3)                     # Opérateurs logiques
array([[False, True], [False, False]])
>> np.log(a)                                    # Fonctions mathématiques
array([[0., 0.30103], [0.47712125, 0.60205999]])
```

- Opérateurs arithmétiques : `+`, `-`, `/`, `%`, `//`, `*`, `**`
- Opérateurs de comparaison : `>`, `>=`, `<`, `<=`, `==`, `!=`
- Opérateurs logiques : `logical_and`, `logical_or`, `logical_not`, `logical_xor`
- Fonctions mathématiques de `numpy` :
  - Fonctions trigonométriques : `cos`, `sin`, `tan`, `arccos`, `arcsin`, `arctan`
  - Fonctions hyperboliques : `cosh`, `sinh`, `tanh`, `arccosh`, `arcsinh`, `arctanh`
  - Exponentiels et logarithmes : `exp`, `log`, `log10`

Introduction à Matplotlib

Une figure se trace en trois étapes :

- **L'initialisation :**
  - Créé un objet *figure* avec la fonction `figure`
  - Crée éventuellement un objet *axes* avec la fonction `subplots`

Illustration

```
>> import matplotlib.pyplot as plt

>> plt.figure()                                    # Initialization
>> plt.plot(x, y, "color", label="label")          # Filling
>> plt.xlabel("x")                                 # ...
>> plt.ylabel("y")                                 # ...
>> plt.title("Title")                              # ...
>> plt.legend()                                    # ...
>> plt.show()                                      # Display
```

Une figure se trace en trois étapes :

- **L'initialisation :**
  - Créé un objet *figure* avec la fonction `figure`
  - Crée éventuellement un objet *axes* avec la fonction `subplots`
- **Le remplissage :**
  - trace des données
  - change les propriétés de la figure, du tracé, des axes...

Illustration

```
>> import matplotlib.pyplot as plt

>> plt.figure()                                    # Initialization
>> plt.plot(x, y, "color", label="label")          # Filling
>> plt.xlabel("x")                                 # ...
>> plt.ylabel("y")                                 # ...
>> plt.title("Title")                              # ...
>> plt.legend()                                    # ...
>> plt.show()                                      # Display
```

Une figure se trace en trois étapes :

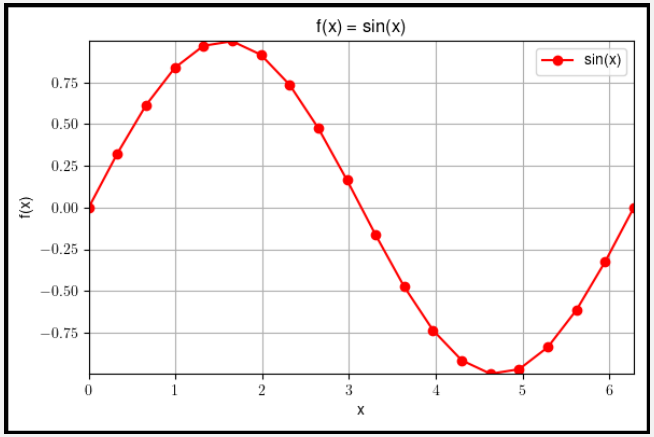
- **L'initialisation :**
  - Créé un objet *figure* avec la fonction `figure`
  - Crée éventuellement un objet *axes* avec la fonction `subplots`
- **Le remplissage :**
  - trace des données
  - change les propriétés de la figure, du tracé, des axes...
- **L'affichage :** affiche l'objet fini à l'écran

Illustration

```
>> import matplotlib.pyplot as plt

>> plt.figure()                                    # Initialization
>> plt.plot(x, y, "color", label="label")          # Filling
>> plt.xlabel("x")                                 # ...
>> plt.ylabel("y")                                 # ...
>> plt.title("Title")                              # ...
>> plt.legend()                                    # ...
>> plt.show()                                      # Display
```

Exemple de figure basique [sans options]

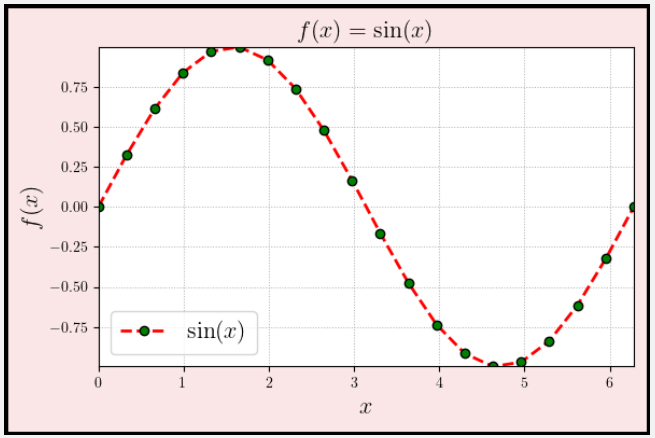
```
import numpy as np
import matplotlib.pyplot as plt

plt.close("all")

x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)

plt.figure(figsize=(6, 4))
plt.plot(x, y, "r-o",
         label="sin(x)")
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.title("f(x) = sin(x)")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.grid()
plt.show()
```



- Fonctions extensivement customisables, notamment `plot`
- Affichage du texte LaTeX avec le préfixe `"r"` (*raw string*)
- Argument `fontsize` pour gérer la taille des polices dans les fonctions de remplissage

Exemple de figure basique [avec options]

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

plt.close("all")

x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)
```




```
plt.figure(figsize=(6, 4), facecolor=(0.96, 0.98, 0.99))
plt.plot(x, y, color="r", linewidth=2, linestyle='--', label=r"$\sin(x)$",
         marker="o", markeredgecolor="k", markerfacecolor='g')
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.title(r"$f(x) = \sin(x)$", fontsize=16)
plt.xlabel(r"$x$", fontsize=16)
plt.ylabel(r"$f(x)$", fontsize=16)
plt.legend(loc=3, fontsize=16)
plt.grid(linestyle=":", linewidth=0.7)
plt.show()
```

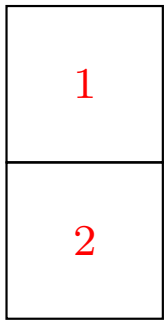
- Fonctions extensivement customisables, notamment `plot`
- Affichage du texte LaTeX avec le préfixe `"r"` (*raw string*)
- Argument `fontsize` pour gérer la taille des polices dans les fonctions de remplissage

Fonctions principales pour le tracé 2D	
`subplot()`	Crée des sous figures
`plot()`	Trace des lignes/marqueurs
`loglog()`	Trace des lignes/marqueurs en échelle logarithmique
`semilogx()/semilogy`	Trace des lignes/marqueurs en échelle log. suivant x/y
`polar()`	Trace des lignes/marqueurs en coordonnées polaires
`imshow()`	Trace une image
`scatter()`	Trace des points (marqueurs)
`quiver()`	Trace des vecteurs
`pcolor()`	Tracé en couleur
`pcolormesh()`	Tracé en couleur (plus rapide avec un maillage régulier)
`contour()`	Trace les lignes de contours
`contourf()`	Trace des contours remplis
`text()`	Écrit du texte

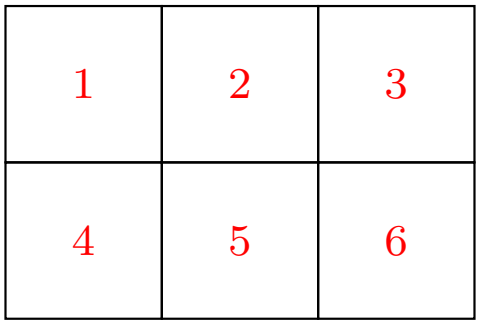
Fonctions principales pour gérer les propriétés de la figure	
`axis()`	Fixe les limites des axes des abscisses et ordonnées
`xlabel()/ylabel()`	Labels des axes des abscisses et ordonnées
`title()`	Titre de la figure
`grid()`	Affiche la grille
`legend()`	Affiche la légende (si les labels des tracés sont définis)
`colorbar()`	Affiche la colorbar (`imshow`, `pcolormesh`, `contour`, `contourf`)
`tight_layout()`	Ajuste les subplots dans la figure
`savefig()`	Sauvegarde la figure

subplot(lignes, colonnes, indice)

```
plt.figure()
plt.subplot(2, 1, 1)
plt.plot(x1, y1)
plt.xlabel("x1")
plt.ylabel("y1")
...
plt.subplot(2, 1, 2)
plt.plot(x2, y2)
plt.xlabel("x2")
plt.ylabel("y2")
...
plt.show()
```

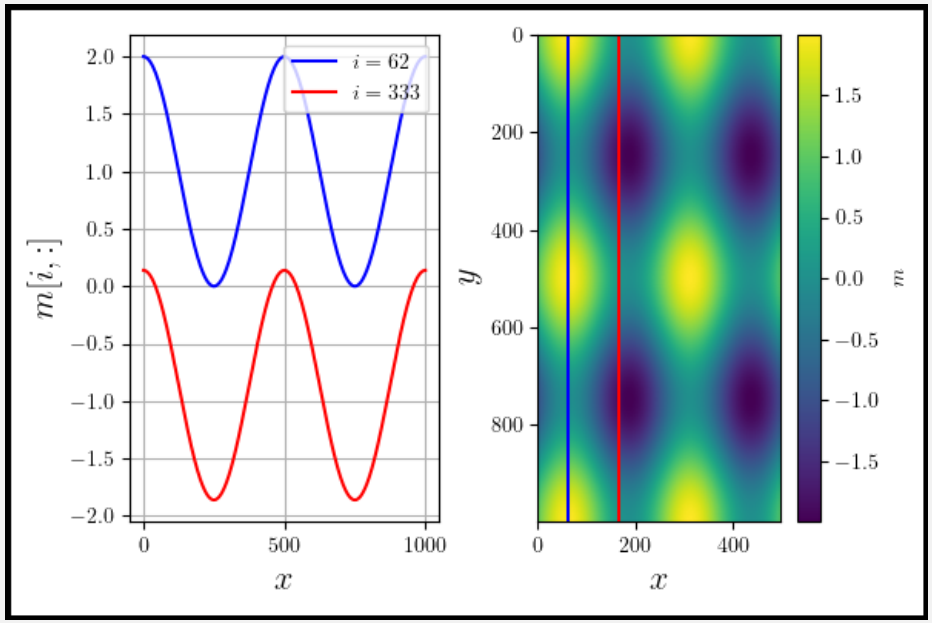



```
plt.figure()
plt.subplot(2, 3, 1)
...
plt.subplot(2, 3, 2)
...
plt.subplot(2, 3, 3)
...
plt.subplot(2, 3, 4)
...
plt.subplot(2, 3, 5)
...
plt.subplot(2, 3, 6)
...
plt.show()
```



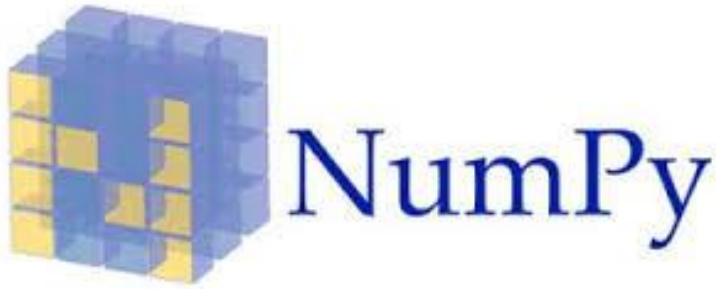
Exemple de figure avec subplot

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

Nx, Ny = 500, 1000
id1, id2 = int(Nx/8), int(Nx/3)
x = np.linspace(0, 4 * np.pi, Nx)
y = np.linspace(0, 4 * np.pi, Ny)
m = sum(np.meshgrid(np.sin(x), np.cos(y)))

plt.figure(figsize=(6, 4))

plt.subplot(1, 2, 1)
plt.plot(m[:, id1], 'b', label=rf'$i={idx1}$')
plt.plot(m[:, id2], 'r', label=rf'$i={idx2}$')
plt.grid()
plt.legend(loc=1)
plt.xlabel(r"$x$", fontsize=16)
plt.ylabel(r"$m[i, :]$", fontsize=16)

plt.subplot(1, 2, 2)
plt.imshow(m)
plt.axvline(id1, color='b')
plt.axvline(id2, color='r')
plt.xlabel(r"$x$", fontsize=16)
plt.ylabel(r"$y$", fontsize=16)
plt.colorbar(label=r'$m$')

plt.tight_layout()          # Ajuste automatiquement les subplots pour qu'ils...
plt.show()                  # ...soit correctement positionnés dans la figure.
```



Notebook - S03 complète

Introduction à Numpy



Indexation et slicing des objets ndarray

```
>> a = [[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]]          # Indexation des types built in :
>> a[0]                     # Retourne [0, 1, 2]
>> a[0][0]                  # Retourne 0
```

Indexation des objets ndarray :

```
>> import numpy as np
>> b = np.array(a)          # Conversion en ndarray
>> b[0, 0]                  # Retourne 0 !
```

Slicing puissant et élégant avec les ndarray :

```
>> a[:-1, :-1]              # Lignes de 0 à -1, colonnes de 0 à -1
array([[0, 1],
       [3, 4]])
>> a[:2, ::-1]              # Lignes de 0 à 1, colonnes inversées
array([[2, 1, 0],
       [5, 4, 3]])
>> a[[0, -1], :]            # Ligne 0 et -1, toutes les colonnes
array([[0, 1, 2],
       [6, 7, 8]]
```

Numpy functions for File I/O

Method	Description
`loadtxt()/savetxt()`	Read/Save an array to a text file
`load()/save()`	Read/Save an array to a `.npy` binary file
`load()/savez()`	Save several arrays into an `.npz` uncompressed archive
`load()/savez_compressed()`	Save several arrays into an `.npz` compressed archive

Fichier texte

```
>> v1 = np.array([1, 2, 3])
>> v2 = np.array([4, 5, 6])

>> np.savetxt("fn.txt", v1)
>> np.loadtxt("fn.txt")
array([1, 2, 3])
```

Fichier binaire

```
>> np.save('fn.npy', v1)
>> np.load('fn.npy')
array([1, 2, 3])

>> np.savez('fn.npz', v1=v1, v2=v2)
>> np.load('fn.npz')['v2']
array([4, 5, 6])
```

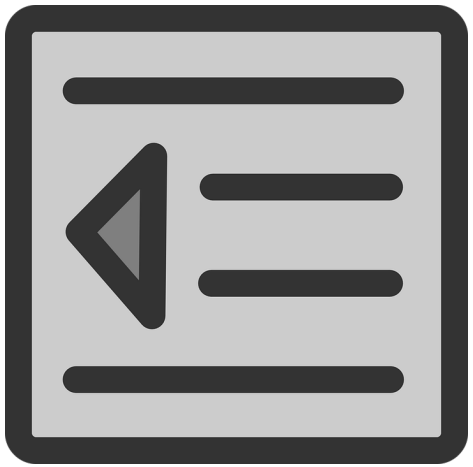
Saving/loading time for an array of 10^6 random elements

Function	Execution time	Disk usage
`loadtxt()/savetxt()`	900ms/700ms	24 Mo
`load()/save()`	2ms/25ms	7.7 Mo
`load()/savez()`	45μs/45ms	7.7 Mo
`load()/savez_compressed()`	45μs/325ms	7.2 Mo (7.4 Ko for np.zeros)

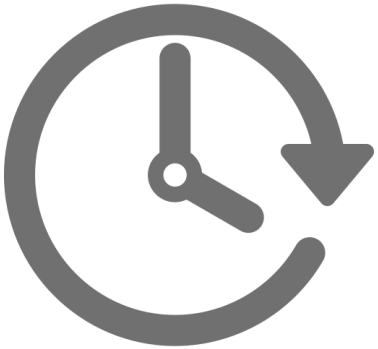
Notebook - S04 complète

Notebook - CC + S05 complète

Instructions composées



Pixabay.com



Qu'est ce qu'une instruction composée ?

- Un bloc de code contenant des instructions
- Typiquement une *instruction algorithmique* ou une *fonction/classe*

Structure des instructions composées sous Python :

- Caractère `":"` à la fin de la déclaration (1ère ligne) d'une instruction composée
- Les instructions suivantes sont indentées du **même nombre de caractères espace** `"␣"` et sont considérées comme faisant partie du **même bloc de code**
- Python utilise l'**indentation** comme méthode pour grouper les instructions

Avantages de l'indentation :

- Pas d'instruction `end` : Pas de temps perdu avec le *debug* dû aux `end` manquants !
- Force les bonnes pratiques : Lisibilité du code !

L'indentation sous Python

- Nécessaire pour les instructions composées (`if`, `for`, `while`, `with`, `try`, `def`, `class`)
- Niveau d'indentation : nombre de caractères "␣" au début d'une ligne logique
- Norme proposée par la PEP-8 : 4 caractères "␣" pour 1 niveau d'indentation
- Mélange de différents types d'indentation dans un même groupe : IndentationError

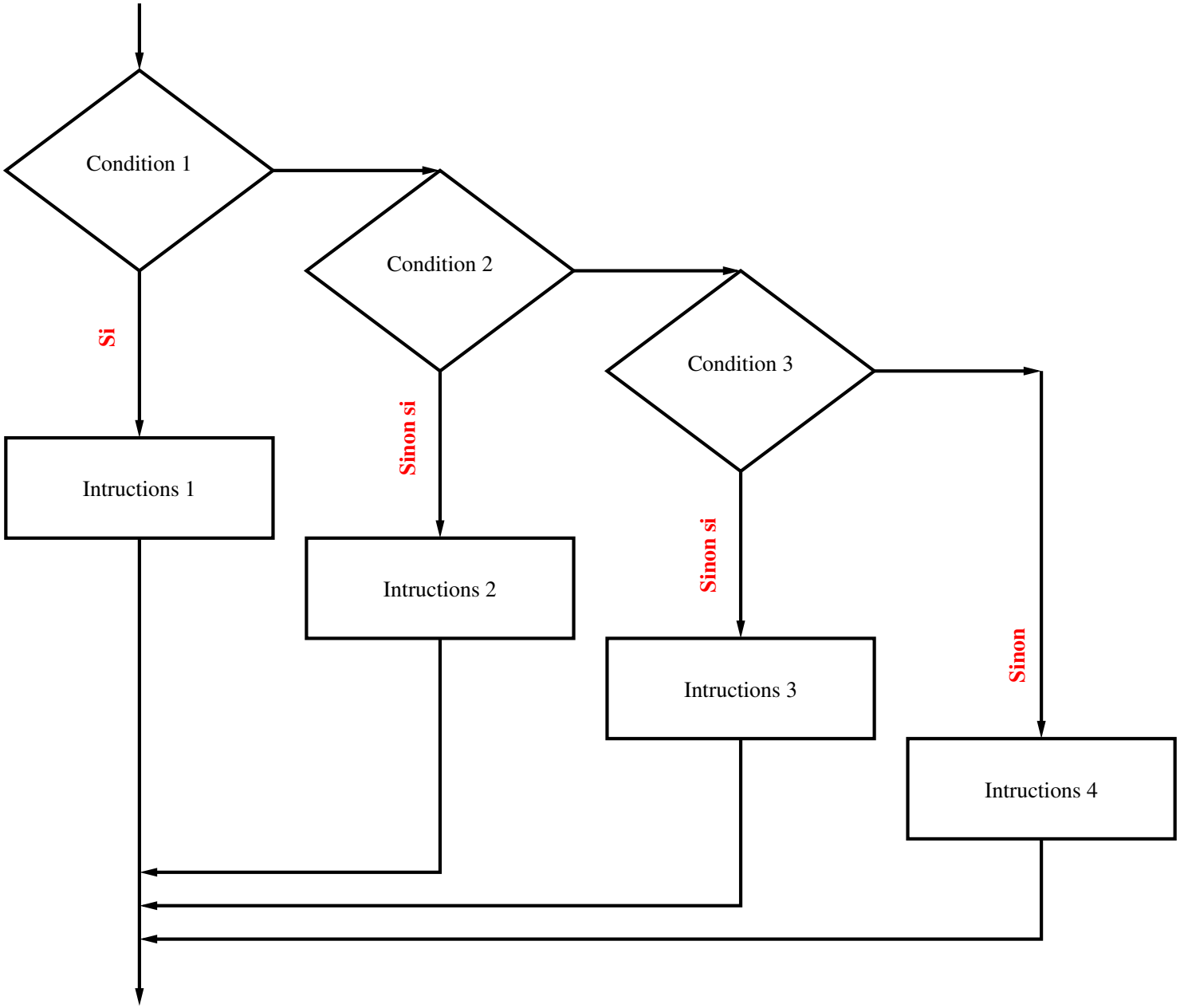
Illustration : Instructions composées imbriquées

```
def f(x):                    # Instruction composée L0
    y = x**2                 #     | Début bloc instructions indentées L1
    def g(z):                #     | Instruction composée L1
        z = z**3             #     |     | Début bloc instruction indentée L2
        return z             #     |     | Fin bloc instruction indentée L2
    return y + g(y)          #     | Fin bloc instructions indentées L1}
```

Gardez en tête : la notion d'indentation est l'une des notions les plus importantes sous Python. C'est un élément clé de sa syntaxe !

Instructions algorithmiques

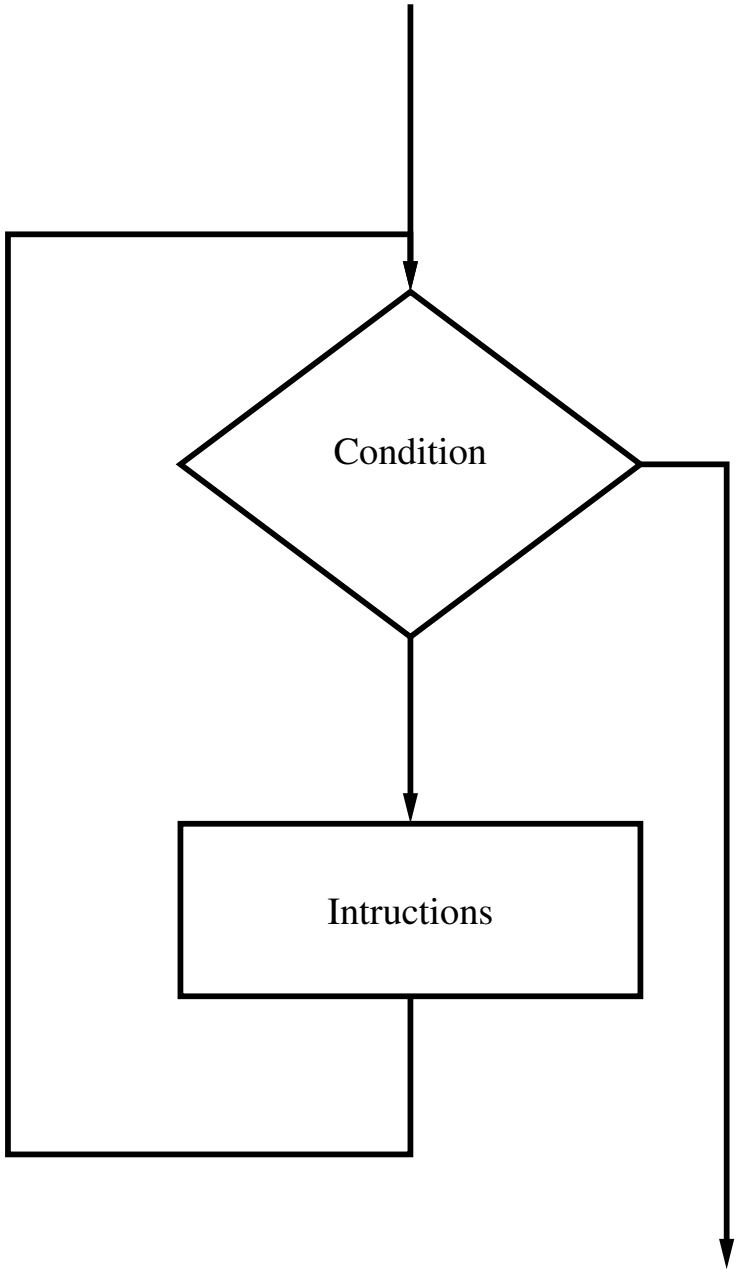
Screenshot by Gamaliel Espinoza Macedo


Condition 1
Condition 2
Condition 3
Si
Sinon si
Sinon si
Sinon
Intructions 1
Intructions 2
Intructions 3
Intructions 4

Syntaxe : if ... elif ... else ...

```
if condition is True:
    print("Verified")
elif condition is False:                # elif optionnel
    print("Not Verified")
else:                                   # else optionnel -- n'accepte pas...
    print("Other possibility ?")        # ... de condition
```

Relational operators : Compare the values on either side of them		
Equality/Difference	== et !=	–
Superiority/Inferiority	> / < *ou* >= / <=	–
Logical operators : Classical logical comparisons		
Logical .AND.	`and`	–
Logical .OR.	`or`	–
Logical .NOT.	`not`	Inverts the adjacent operator
Membership operators : Test for membership in sequence		
Included	*var* `in` *seq*	`True if` *var* `in` *seq*, `False` otherwise
Excluded	*var* `not in` *seq*	`True if` *var* `not in` *seq*, `False` otherwise
Identity operators : Compare memory location of objects		
Equality	*a* `is` *b*	`True if id`$(a) ==$ `id`(b), `False` otherwise
Difference	*a* `is not` *b*	`True if id`(a) != `id`(b), `False` otherwise

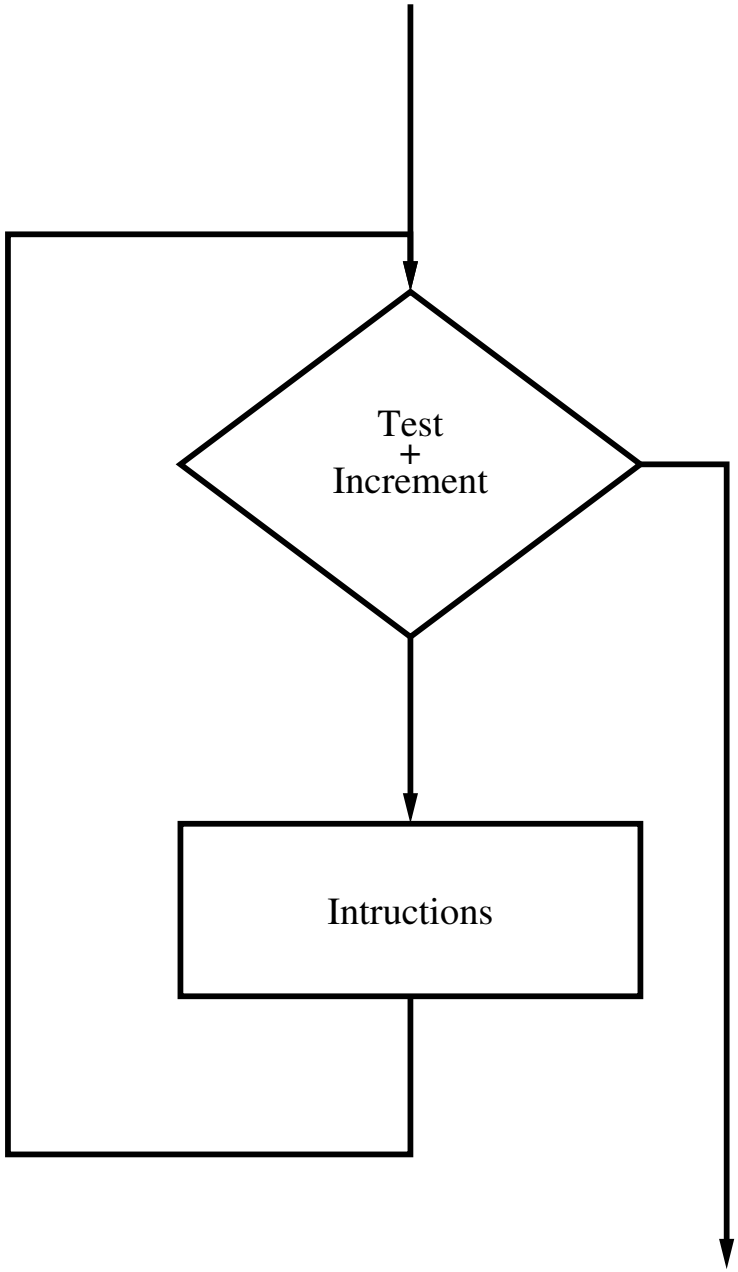

Condition
Intructions

Syntaxe : while ...

```
n = 0                # Initialisation de la variable utilisée dans la suite
while n != 10:       # Définition de l'instruction composée
    n += 1           # Bloc d'instructions répétées
    print(n)
print("Fin")         # Instruction non incluse dans la boucle
```

- Exécution théorique infinie (`while True`: ...)
- Définition d'une condition de sortie pour arrêter l'exécution
- Instructions `break` (sortie immédiate) et `continue` (itération suivante) acceptées
- Instruction `else` acceptée : contenu exécuté si la condition n'est plus vraie
- **Note :** Opérateur `"+="` : assignation et addition de manière compacte :
  - `n += 1` $\longrightarrow$ `n = n+1`
  - `n -= 1` $\longrightarrow$ `n = n-1`
  - `n *= 2` $\longrightarrow$ `n = n*2`
  - `n **= 2` $\longrightarrow$ `n = n**2`
  - `n /= 2` $\longrightarrow$ `n = n/2`
  - `n //= 2` $\longrightarrow$ `n = n//2`
  - `n %= 2` $\longrightarrow$ `n = n%2`

Notebook - S06 complète


Test
+
Increment
Intructions

Syntaxe : for ... in ...

```
for p in range(10):   # Définition de l'instruction composée
    p = p**2          # Bloc d'instructions répétées
    print(p)
print("Fin")          # Instruction non incluse dans la boucle

0                     # Exécution de l'instruction composée for
1
4
...
64
81                    # Fin de l'exécution du for
Fin                   # Exécution des instructions suivantes
```

- Exécution d'un bloc d'instructions pour un nombre fixé d'itérations
- Instructions **break** (sortie immédiate) et **continue** (itération suivante) acceptées
- Instruction **else** acceptée : contenu exécuté si l'itérable est parcouru entièrement
- Peut itérer sur différents types d'objets :
  - itérables (**list**, **tuple**, **dictionary**, **array**, **string**, **range**, fichier, ...)
  - itérateurs
  - toute fonction retournant un itérable ou un itérateur

Quelques exemples d'objets itérables

list

```
lst = [1, "a"]
for p in lst:
    print(p)        # 1, "a"
```

tuple

```
tpl = (1, "a")
for p in tpl:
    print(p)        # 1, "a"
```

dict

```
dct = {"a": 1, "b": 2}
for p in dct:
    print(p)        # "a", "b"
```

str

```
strg = "abc"
for p in strg:
    print(p)        # "a", "b", "c"
```

open file

```
file = open("myfile")
for line in file:
    print(line)     # each line
```

set

```
mset = {1, 1, 2, 4}
for p in mset:
    print(p)        # 1, 2, 4
```

De puissantes fonctions pour itérer

La fonction enumerate

```
lst = ["Python", "Is", "Amazing"]

for i in range(len(lst)):
    print(i, lst[i])

0 Python
1 Is
2 Amazing
```

enumerate retourne un tuple :

```
for index, item in enumerate(lst):
    print(index, item)

0 Python
1 Is
2 Amazing
```

La fonction zip

```
import string

ltr = string.ascii_lowercase
rtl = reversed(ltr)
idx = range(len(ltr))

for i, j, k in zip(idx, ltr, rtl):
    print(i, j, k)

0 a z
1 b y
2 c x
(...)
23 x c
24 y b
25 z a
```

La compréhension de liste

```
# Liste des carrés de 0 à 9
>> B = [x**2 for x in range(10)]      # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Filtre les valeurs paires de B :
>> [x for x in B if x % 2 == 0]       # [0, 4, 16, 36, 64]

# Un exemple avec un objet de type str :
>> ["a" * i for i in range(5)]        # ["", "a", "aa", "aaa", "aaaa"]

# Un exemple de compréhensions de listes imbriquées :
>> [[i*i for i in range(x)] for x in range(5)]
[[], [0], [0, 1], [0, 1, 4], [0, 1, 4, 9]]

# Ce mécanisme peut aider pour la définition de tableaux numpy
>> np.array([[i for i in range(j, j + 5)] for j in range(5)])
np.array([[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]])
```

Notebook - S07 complète

Programmation fonctionnelle

Photography by Vestman

Syntaxe : `def` ... `return` ...

```
def mult(x, y=2):          # x : positional argument, y : keyword argument
    return x*y
```

```
mult(3)
6
```

```
mult(3, 4)
12
```

- L'instruction `def` permet de définir une fonction
- Une fonction est un objet qui retourne un objet
- Par défaut, une fonction retourne `None` s'il n'y a pas d'instruction `return`
- Une fonction peut être utilisée comme tout autre objet. Elle peut être :
  - un argument ou un `return` d'une autre fonction
  - référencée par une variable
  - un élément d'une structure (`tuple`, `list`, `dict`, ...)
- Une définition de fonction accepte des paramètres :
  - obligatoires, appelés ***positional arguments***
  - optionnels, appelés ***keyword arguments*** spécifiant des valeurs par défaut[1]

[1] Ces valeurs sont évaluées au moment où la fonction est définie, pas lorsqu'elle est appelée

Notebook - S08 complète

Deux dernières séances: training !
Notebook - S09 complète