

Python	LES BASES	Variables	La portées des variables et les références	Structures conditionnelles	Les chaînes de caractères	<u>Liste[]</u>	Tuples()	Sets	Utilisez des dictionnaires{}
Les opérateurs ternaires	Apprendre à faire des boucles	Useful Operators in python	Les chemins	Les fonctions	La méthode import	Unpacking	Python documentation	Les packages	Built in errors in Python
Raising errors in Python	Creating your own errors in Python	Dealing with Python errors	The on success block and re-raising exceptions	Gérez les exceptions	Compréhension	Advanced built-in functions in Python	Découvrez la récursion	Utiliser des fichiers et des dossiers	Recherche récursive d'un fichier
Déplacer des fichiers	Aller chercher de l'aide avec les fonctions dir et help	Appréhendez les classes	L'objet en python	Héritage	Polymorphisme Inheritance Composition	How to Write Cleaner Python Code using Abstract Classes	Un objet est-il callable ?	Définissez des propriétés	Appliquez des méthodes spéciales
Making Custom Containers	Appliquez 2 méthodes de tri	Découvrez la boucle for	Pratiques Python	Concurrence via concurrent.futures	Découvrez les métaclasses	Manipulez les expressions régulières	Exprimez le temps	Timing your code with Python	Faites de la programmation système
Les Modules	Utilisez des modules de mathématiques	Les Packages	Le Logging	Itertools	Web Scraping	Gérez les mots de passe	Gérez les réseaux	Créez des tests unitaires avec unittest	Bases de la programmation parallèle avec threading
Les bases de données	Projet	Typing in Python	Créez des interfaces graphiques avec Tkinter	TKinter	PySide2	Distribuer facilement nos programmes avec cx_freeze	Distribuer facilement nos programmes avec FBS	Distribuer facilement nos programmes avec PyInstaller	Les bonnes pratiques
Pour finir et bien continuer	Installer des packages supplémentaires avec PIP		Les environnements virtuels	Machine Learnia					

# Bibliographie

- <https://blog.teclado.com/30-days-of-python/>
- <https://www.udemy.com/course/complete-python-bootcamp/learn/lecture/20205526#overview>
- <https://www.udemy.com/course/formation-complete-python/learn/lecture/14497974?start=0#overview>
- <https://www.udemy.com/course/the-complete-python-course/learn/lecture/16856932#overview>
- <https://realpython.com/>

# Python

All You must know in python

# LES BASES

# Indexing start with 0

String: "Julien"

List: ['J','u','l','i','e','n']

Index	Value
0	J
1	u
2	l
3	i
4	e
5	n

# Python Math functions

## Python Math functions

Symbol	Function	Example	Result
+	addition	$5 + 3$	8
-	subtraction	$10 - 6$	4
*	multiplication	$3 * 7$	21
//	integer division	$15 // 6$	2
/	float division	$15 / 6$	2.5
**	power	$7 ** 2$	49

# Order of Operations

1. ( )
2. \*\*
3. \* / // %
4. + -
5. left to right

Example:  $x = 1 + 5 ^{\color{red}2} * (3 // 2) - 6 \% 4$

$x = 1 + 5 ^{\color{red}2} * (3 \color{gray}{//} 2) - 6 \% 4$

$x = 1 + 5 ^{\color{red}2} * (5 \color{gray}{//} 2) - 6 \% 4$

$x = 1 + 5 ^{\color{red}2} * (5 \color{gray}{//} 2) - 6 \% 2$

$x = 1 + 5 ^{\color{red}2} * 6 (3 \color{gray}{//} 2) - 6 \% 2$

$x = 1 + 5 ^{\color{red}2} * (3 \color{gray}{//} 2) - 6 \% 2$

# Variables

# Règles et conventions de nommage

Un nom de variable ne peut pas:

- commencer par un chiffre
- Contenir d'espaces

Un nom de variable ne peut:

- Contenir que des caractères alphanumériques et le \_

Certains mots sont réservés (`print`, `True`, `break`,...)

Exemples de noms de variables qui ne sont pas valides:

- 75Paris (commence par un chiffre)
- Site-Web (contient -)
- #lien vidéo (contient un caractère invalide (#) et un espace)
- True (mot clef réservé)

variables valides:

Paris75

Site\_Web

lienVideo

true

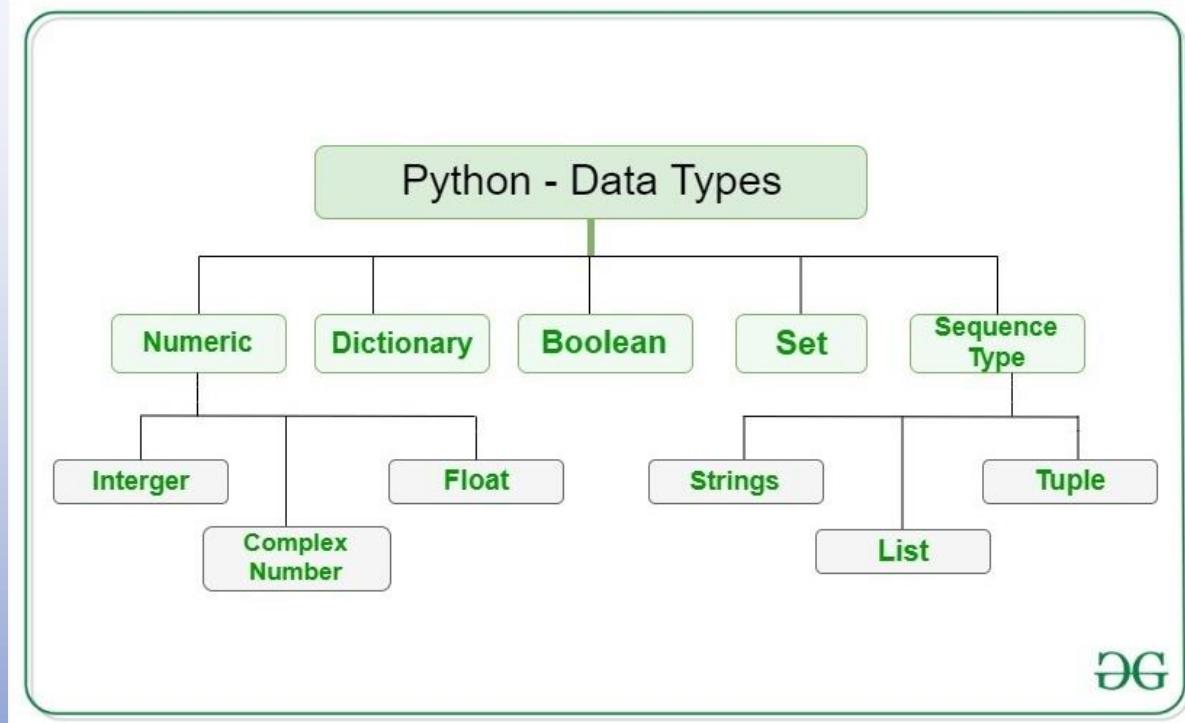
# Les types natifs

## Les types natifs de base:

Les chaînes de caractères  
Les nombres  
Les booléens

## D'autres types natifs construits à partir des types natifs de base:

Les types séquentiels (listes, tuples)  
Les types d'ensembles (set, frozenset)  
Les types de correspondances (dictionnaires)



# Les conventions de nommage

Pour nommer des variables on préfère utiliser uniquement des lettres minuscules et séparer chaque mot par \_

75Paris

Site-Web

#lien video

True

Paris75

Site\_Web

lienVideo

true

paris\_75

site\_web

lien\_video

true



# Les mots réservés

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

# Les nombres

<https://www.teclado.com/30-days-of-python/python-30-day-1-numbers-printing>



Day 1 Numbers, Arithmetic, and Printing to the Console.pdf

- Les nombres entier
  - Nombres positifs ou négatifs
  - Nombres sans virgules
  - Avec python 3 on peut écrire 1000000 comme ceci 1\_000\_000
- Les nombres décimaux (flottants)
  - Nombres qui contiennent des décimales après la virgule(en python on utilise pas la virgule mais le point (notation anglaise)) ex: -5.485 1.254 150,87

# Les booléens



Day 5 Conditionals and Booleans.pdf

<https://docs.python.org/3/library/stdtypes.html#truth-value-testing>

<https://www.teclado.com/30-days-of-python/python-30-day-5-conditionals-booleans>

- Les booléens sont des types natifs qui sont une sous-classe des nombres entiers:

1

True

0

False

- Il est donc possible d'ajouter un booléen à un entier:

True + 1 = 2

False + 5 = 5

- Tous les objets en python peuvent être vrais ou faux (truthy or falsy), pour savoir si un objet est vrais ou faux on utilise la fonction `bool()`

```
bool("bonjour") retourne True  
bool("") retourne False
```

Constants defined to be false: **None** and **False**.

Zero of any numeric type: **0**, **0.0**, **Decimal(0)**, **Fraction(0, 1)**

Empty sequences and collections: **"**, **[]**, **{}**, **Set()**, **Range(0)**

# Les booléens True ou False

## FALSE

0

0.0

""

[]

{}

None

Empty string

$3 < 2$

$-1 > 33$

1 and 0

$8 \geq 100$

0 or ""

$5 \leq 1$

5 - 5

$0 == 88$

$1.2 != 1.2$

## TRUE

any non-zero number

any non-empty string

any non-empty list

1

1 or 0

81 and -23

'pig'

'cat' == 'cat'

['dog']

'a' < 'b'

$10 > 5$

$-1 < 33$

$8 \geq 8$

$0 == 0$

$1.2 != 1.3$

$5 > 3$  and 10

$1 == 0$  or [0]

# Les fonctions\* de conversion

Même si on parle de fonctions de conversion, ce sont en réalité des classes.

str()	Chaines de caractères
int()	Nombres entiers
float()	Nombres décimaux
bool()	Booléens

On peut utiliser ces classes pour convertir des objets d'un type à un autre:

str(5) -> "5"	convertie un nombre entier en chaîne de caractère
int("2") -> 2	Convertie la chaîne de caractère "2" en nombre entier 2

En réalité python ne convertie pas l'objet il en recréait un avec le type demandé: il recréait un objet string avec le contenu 5, il recréait un objet entier avec le contenu 2.

# Immutable et Mutable variable

In Python, most variables are *immutable*, meaning they don't change in-place.

Python creates a new value in a different memory location when a variable changes.

Integers	Immutable
Floats	Immutable
Strings	Immutable
Tuples	Immutable
List[]	Mutable
Set()	Mutable
Dictionary{}	Mutable

# Comment connaitre le type d'une variable

```
type(nom_de_la_variable)
a = 3
type(a)
<class 'int'>
```

# Comment savoir si un objet est une instance d'une class 1/6

`isinstance(object, classinfo)`

The `isinstance()` function checks if the object (first argument) is an instance of subclass of `classinfo` class (second argument).

**Isinstance() Parameters:**

object – object to be checked

`Classinfo` – class, type, or tuple of classes and types

**Return Value form `isinstance()`:**

- True if the object is an instance of a class or any element of the tuple
- False otherwise

If `classinfo` is not a type or tuple of types, a `TypeError` exception is raised.

# How is isinstance() work 2/6

```
class Foo:  
    a = 5  
  
fooInstance = Foo()  
  
print(isinstance(fooInstance, Foo))  
print(isinstance(fooInstance, (list, tuple)))  
print(isinstance(fooInstance, (list, tuple, Foo)))  
  
True  
False  
True
```

# La différence entre `isinstance` et `type` 3/6

```
In[1]: nombre = 5
```

```
In[2]: type(nombre)  
Out[2]: int
```

```
In[3]: Type(nombre) == int  
Out[3]: True
```

```
In[4]: Type(nombre) == float  
Out[4]: False
```

```
In[5]: isinstance(nombre, int)  
Out[5]: True
```

```
In[6]: isinstance(nombre, float)  
Out[6]: False
```

# La difference entre isinstance et type 4/6

A savoir pour comprendre l'exemple ci-dessous: Les Booleans sont un type qui herite des nombres entiers.

L'exemple ci-dessous montre donc que isinstance() test si un objet herite ou non d'une classe. Tandis que type check uniquement la nature de l'objet teste.

```
In[7]: type(True) == bool  
Out[7]: True
```

```
In[8]: type(True) == int  
Out[8]: False
```

```
In[9]: isinstance(True, int)  
Out[9]: True
```

# Tester si ma variable appartient à un type ou à un autre 5/6

```
In[11]: isinstance(True, (int, str))  
Out[11]: True
```

```
In[11]: isinstance(True, (float, str))  
Out[11]: False
```

# Tester si ma variable appartient à un type ou à un autre 6/6

```
class Maliste(list):
    pass

l = Maliste()
print(type(l) == list)
print(isinstance(l, Maliste))
print(isinstance(l, list))

>>> False
>>> True
>>> True
```

# "is" vs "==" 1/4

```
a = 5000
b = 5000

a == b
>>> True

a is b <=> id(a) == id(b)
>>> False

id(a)
>>> 4497520016

Id(b)
>>> 4497521072
```

Quand on fait `a == b` on compare le contenu des variables `a` et `b`  
Quand on fait `a is b` on compare les adresses mémoires de `a` et de `b`  
Pour connaître l'adresse mémoire d'une variable on fait: `id(nom_de_variable)` et cela retourne l'adresse mémoire.

```
a = [1, 2, 3]
b = a

a is b
>>> True
a == b
>>> True

c = list(a)

a == c
>>> True
a is c
>>> False
```

```
# "is" expressions evaluate to True if two
# variables point to the same object

# "==" evaluates to True if the objects
# referred to by the variables are equal
```

# "is" vs "==" 2/4

Pour les nombres compris entre -5 et 256, nous avons quelque chose d'un peu surprenant du au *small integer caching*.

```
a = 5  
b = 5  
  
a is b  
True
```

a et b sont compris entre -5 et 256 ils utilisent donc une même adresse pour le stockage de leur valeur

Les nombres compris entre -5 et 256 (*small integer caching*) vont avoir une place prédéfinie en mémoire et dès que l'on va créer une variable on va pointer vers la même adresse mémoire.

Notons que si, dans Visual studio on écrit, a = 5000 et b = 5000 puis **a is b => True**  
Pourquoi ???

C'est encore due à une optimisation de python !!! Dans la slide précédente nous étions dans un interpréteur python interactif, par conséquent, les lignes sont lues une à une, il est donc pas possible de savoir ce qu'il y aura dans b au moment où on valide l'affectation de a.

Dans Visual Studio Code, le code est analysée dans son ensemble avant exécution, du coup VSCode se rend compte que l'on a deux variables pour une même valeur qui n'est pas dans l'intervalle -5, 256. Python est maintenant informé qu'il a 2 variables avec la même valeur et va donc optimiser le code en faisant pointer les variables sur la même adresse mémoire.

# "is" vs "==" 3/4

Interpréteur interactif:

```
def foo():
    a = 5000
    b = 5000
    print(a is b)

foo()
True
```

Ici la foo() est évaluée dans son ensemble car on tape sur entrée une fois que l'on a fini de saisir la fonction (entièrre)

Il y a donc une analyse du code de la fonction dans son ensemble et comme avec VSCode, Python décide d'optimiser le code en utilisant une adresse mémoire pour les deux variables.

De même avec les chaînes de caractères:

```
a = "Bonjour"
b = "Bonjour" => chaîne courte, python re-utilise l'adresse mémoire

a is b
>>> True

a = "Bonjour comment allez-vous?"
b = "Bonjour comment allez-vous?" => chaîne longue, création de deux adresses mémoire
a is b
>>> False
```

# "is" vs "==" 4/4

Le concept de singleton qui est en fait un objet unique:

*True*

*False* => ce sont des singleton, ils existent de façon unique dans la mémoire de notre script

*None*

# Affectation multiple

```
# Affectation multiple  
a = b = c = 5
```

## Affectation //

```
# Affectation parallèle  
a, b = 5, 8  
  
# L'affectation parallèle est aussi utilisée pour inverser 2 variables:  
a, b = b, a
```

# Different ways to test multiple flags at once

```
# Different ways to test multiple
# flags at once in Python
x, y, z = 0, 1, 0

if x == 1 or y == 1 or z == 1:
    print('passed')

if 1 in (x, y, z):
    print('passed')

# These only test for truthiness:
if x or y or z:
    print('passed')

if any((x, y, z)):
    print('passed')

>> passed
>> passed
>> passed
>> passed
```

# Résumé

- Les variables permettent de conserver dans le temps des données de votre programme.
- Vous pouvez vous servir de ces variables pour différentes choses : les afficher, faire des calculs avec, etc.
- Pour affecter une valeur à une variable, on utilise la syntaxe `nom_de_variable = valeur`.
- Il existe différents types de variables, en fonction de l'information que vous désirez conserver :int, float, chaîne de caractères etc.
- Pour afficher une donnée, comme la valeur d'une variable par exemple, on utilise la fonction `print`.
- Il est possible de faire diffèrent types d'affectation de variables:
  - Affectation simple `a = 5`
  - Affectation multiple `a, b = 5`
  - Affectation parallèle `a, b = 5, 7` ou `a, b = b, a` (inversion du contenu des variables a et b)

# La portées des variables et les références

# La portée des variables

En Python, comme dans la plupart des langages, on trouve des règles qui définissent la portée des variables. La portée utilisée dans ce sens c'est « quand et comment les variables sont-elles accessibles ? ». Quand vous définissez une fonction, quelles variables sont utilisables dans son corps ? Uniquement les paramètres ? Est-ce qu'on peut créer dans notre corps de la fonction des variables utilisables en dehors ? Dans nos fonctions, quelles variables sont accessibles ?

```
a = 5
def print_a():
    """Fonction chargée d'afficher la variable a.
    Cette variable a n'est pas passée en paramètre de la fonction.
    On suppose qu'elle a été créée en dehors de la fonction, on veut voir
    si elle est accessible depuis le corps de la fonction"""
    print("La variable a = {0}.".format(a))

print_a()
>>> La variable a = 5.
a = 8
print_a()
>>> La variable a = 8.
```

Surprise ! Ou peut-être pas...

La variable a n'est pas passée en paramètre de la fonction print\_a. Et pourtant, Python la trouve, tant qu'elle a été définie avant l'appel de la fonction.  
C'est là qu'interviennent les différents espaces.

# L'espace local

## L'espace local

Dans votre fonction, quand vous faites référence à une variable `a`, Python vérifie dans l'espace local de la fonction. Cet espace contient les paramètres qui sont passés à la fonction et les variables définies dans son corps. Python apprend ainsi que la variable `a` n'existe pas dans l'espace local de la fonction. Dans ce cas, il va regarder dans l'espace local dans lequel la fonction a été appelée. Et là, il trouve bien la variable `a` et peut donc l'afficher.

D'une façon générale, je vous conseille d'éviter d'appeler des variables qui ne sont pas dans l'espace local, sauf si c'est nécessaire. Ce n'est pas très clair à la lecture ; dans l'absolu, préférez travailler sur des variables globales, cela reste plus propre (nous verrons cela plus bas). Pour l'instant, on ne s'intéresse qu'aux mécanismes, on cherche juste à savoir quelles variables sont accessibles depuis le corps d'une fonction et de quelle façon.

```
a = 5
def print_a():
    """Fonction chargée d'afficher la variable a.
    Cette variable a n'est pas passée en paramètre de la fonction.
    On suppose qu'elle a été créée en dehors de la fonction, on veut voir
    si elle est accessible depuis le corps de la fonction"""

    print("La variable a = {0}.".format(a))

print_a()
La variable a = 5.
a = 8
print_a()
La variable a = 8.
```

# Qu'adviennent-il des variables définies dans un corps de fonction ? 1/2

```
def set_var(nouvelle_valeur):
    """Fonction nous permettant de tester la portée des variables
    définies dans notre corps de fonction"""

    # On essaye d'afficher la variable var, si elle existe
    try:
        print("Avant l'affectation, notre variable var vaut {0}.".format(var))
    except NameError:
        print("La variable var n'existe pas encore.")
    var = nouvelle_valeur
    print("Après l'affectation, notre variable var vaut {0}.".format(var))
```

# Qu'adviennent-il des variables définies dans un corps de fonction ? 2/2

```
set_var(5)
La variable var n'existe pas encore.
Après l'affectation, notre variable var vaut 5.
print(var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var' is not defined
```

quelques explications s'imposent :

Lors de notre appel à set\_var, notre variable var n'a pu être trouvée par Python : c'est normal, nous ne l'avons pas encore définie, ni dans notre corps de fonction, ni dans le corps de notre programme. Python affecte la valeur 5 à la variable var, l'affiche et s'arrête.

Au sortir de la fonction, on essaye d'afficher la variable var... mais Python ne la trouve pas ! En effet : elle a été définie dans le corps de la fonction (donc dans son espace local) et, à la fin de l'exécution de la fonction, l'espace est détruit... donc la variable var, définie dans le corps de la fonction, n'existe que dans ce corps et est détruite ensuite.

**Python a une règle d'accès spécifique aux variables extérieures à l'espace local : on peut les lire, mais pas les modifier.** C'est pourquoi, dans notre fonction print\_a, on arrivait à afficher une variable qui n'était pas comprise dans l'espace local de la fonction. En revanche, on ne peut modifier la valeur d'une variable extérieure à l'espace local, par affectation du moins. Si dans votre corps de fonction vous faites var = nouvelle\_valeur, vous n'allez en aucun cas modifier une variable extérieure au corps.

En fait, quand Python trouve une instruction d'affectation, comme par exemple var = nouvelle\_valeur, il va changer la valeur de la variable dans l'espace local de la fonction. Et rappelez-vous que cet espace local est détruit après l'appel à la fonction.

Pour résumer, et c'est ce qu'il faut retenir, une fonction ne peut modifier, par affectation, la valeur d'une variable extérieure à son espace local.

# Une fonction modifiant des objets

On ne peut affecter une nouvelle valeur à un paramètre dans le corps de la fonction. En revanche, on pourrait essayer d'appeler une méthode de l'objet qui le modifie... Voyons cela :

```
def ajouter(liste, valeur_a_ajouter):
...     """Cette fonction insère à la fin de la liste la valeur que l'on veut ajouter"""
...     liste.append(valeur_a_ajouter)
...
ma_liste=['a', 'e', 'i']
ajouter(ma_liste, 'o')
ma_liste
['a', 'e', 'i', 'o']
```

Cela marche ! On passe en paramètres notre objet de type list avec la valeur à ajouter. Et la fonction appelle la méthode append de l'objet. Cette fois, au sortir de la fonction, notre objet a bel et bien été modifié.

Je vois pas pourquoi. Tu as dit qu'une fonction ne pouvait pas affecter de nouvelles valeurs aux paramètres ?

Absolument. Mais c'est cela la petite subtilité dans l'histoire : on ne change pas du tout la valeur du paramètre, on appelle juste une méthode de l'objet. Et cela change tout. Si vous vous embrouillez, retenez que, dans le corps de fonction, si vous faites parametre = nouvelle\_valeur, le paramètre ne sera modifié que dans le corps de la fonction. Alors que si vous faites parametre.methode\_pour\_modifier(...), l'objet derrière le paramètre sera bel et bien modifié.

On peut aussi modifier les attributs d'un objet, par exemple changer une case de la liste ou d'un dictionnaire : ces changements aussi seront effectifs au-delà de l'appel de la fonction.

# Et les références, dans tout cela ? 1/2

Je vais schématiser volontairement : les variables que nous utilisons depuis le début de ce cours cachent en fait des références vers des objets.

Concrètement, j'ai présenté les variables comme ceci : un nom identifiant pointant vers une valeur. Par exemple, notre variable nommée `a` possède une valeur (disons 0).

En fait, une variable est un nom identifiant, pointant vers une référence d'un objet. La référence, c'est un peu sa position en mémoire. Cela reste plus haut niveau que les pointeurs en C par exemple, ce n'est pas vraiment la mémoire de votre ordinateur. Et on ne manipule pas ces références directement.

Cela signifie que deux variables peuvent pointer sur le même objet.

Bah... bien sûr, rien n'empêche de faire deux variables avec la même valeur.

Non non, je ne parle pas de valeurs ici mais d'objets. Voyons un exemple, vous allez comprendre :

```
ma_liste1 = [1, 2, 3]
ma_liste2 = ma_liste1
ma_liste2.append(4)
print(ma_liste2)
[1, 2, 3, 4]
print(ma_liste1)
[1, 2, 3, 4]
```

Nous créons une liste dans la variable `ma_liste1`. À la ligne 2, nous affectons `ma_liste1` à la variable `ma_liste2`. On pourrait croire que `ma_liste2` est une copie de `ma_liste1`. Toutefois, quand on ajoute 4 à `ma_liste2`, `ma_liste1` est aussi modifiée.

On dit que `ma_liste1` et `ma_liste2` contiennent une référence vers le même objet : si on modifie l'objet depuis une des deux variables, le changement sera visible depuis les deux variables.

Euh... j'essaye de faire la même chose avec des variables contenant des entiers et cela ne marche pas.

C'est normal. Les entiers, les flottants, les chaînes de caractères, n'ont aucune méthode travaillant sur l'objet lui-même. Les chaînes de caractères, comme nous l'avons vu, ne modifient pas l'objet appelant mais renvoient un nouvel objet modifié. Et comme nous venons de le voir, le processus d'affectation n'est pas du tout identique à un appel de méthode.

Et si je veux modifier une liste sans toucher à l'autre ?

Eh bien c'est impossible, vu comment nous avons défini nos listes. Les deux variables pointent sur le même objet par jeu de références et donc, inévitablement, si vous modifiez l'objet, vous allez voir le changement depuis les deux variables.

# Et les références, dans tout cela ? 2/2

Toutefois, il existe un moyen pour créer un nouvel objet depuis un autre :

```
ma_liste1 = [1, 2, 3]
ma_liste2 = list(ma_liste1) # Cela revient à copier le contenu de ma_liste1
ma_liste2.append(4)
print(ma_liste2)
[1, 2, 3, 4]
print(ma_liste1)
[1, 2, 3]
```

À la ligne 2, nous avons demandé à Python de créer un nouvel objet basé sur `ma_liste1`. Du coup, les deux variables ne contiennent plus la même référence : elles modifient des objets différents. Vous pouvez utiliser la plupart des constructeurs (c'est le nom qu'on donne à `list` pour créer une liste par exemple) dans ce but. Pour des dictionnaires, utilisez le constructeur `dict` en lui passant en paramètre un dictionnaire déjà construit et vous aurez en retour un dictionnaire, semblable à celui passé en paramètre, mais seulement semblable par le contenu. En fait, il s'agit d'une copie de l'objet, ni plus ni moins.

Pour approcher de plus près les références, vous avez la fonction `id` qui prend en paramètre un objet. Elle renvoie la position de l'objet dans la mémoire Python sous la forme d'un entier (plutôt grand). Je vous invite à faire quelques tests en passant divers objets en paramètre à cette fonction. Sachez au passage que `is` compare les ID des objets de part et d'autre et c'est pour cette raison que je vous ais mis en garde quant à son utilisation.

```
ma_liste1 = [1, 2]
ma_liste2 = [1, 2]
ma_liste1 == ma_liste2 # On compare le contenu des listes
>>> True
ma_liste1 is ma_liste2 # On compare leur référence
>>> False
```

# Les variables globales

Il existe un moyen de modifier, dans une fonction, des variables extérieures à celle-ci. On utilise pour cela des variables globales.

Cette distinction entre variables locales et variables globales se retrouve dans d'autres langages et on recommande souvent d'éviter de trop les utiliser. Elles peuvent avoir leur utilité, toutefois, puisque le mécanisme existe. D'un point de vue strictement personnel, tant que c'est possible, je ne travaille qu'avec des variables locales (comme nous l'avons fait depuis le début de ce cours) mais il m'arrive de faire appel à des variables globales quand c'est nécessaire ou bien plus pratique. Mais ne tombez pas dans l'extrême non plus, ni dans un sens ni dans l'autre.

## Le principe des variables globales

On ne peut faire plus simple. On déclare dans le corps de notre programme, donc en dehors de tout corps de fonction, une variable, tout ce qu'il y a de plus normal. Dans le corps d'une fonction qui doit modifier cette variable (changer sa valeur par affectation), on déclare à Python que la variable qui doit être utilisée dans ce corps est globale.

Python va regarder dans les différents espaces : celui de la fonction, celui dans lequel la fonction a été appelée... ainsi de suite jusqu'à mettre la main sur notre variable. S'il la trouve, il va nous donner le plein accès à cette variable dans le corps de la fonction.

Cela signifie que nous pouvons y accéder en lecture (comme c'est le cas sans avoir besoin de la définir comme variable globale) mais aussi en écriture. Une fonction peut donc ainsi changer la valeur d'une variable directement.

Mais assez de théorie, voyons un exemple.

## Utiliser concrètement les variables globales

Pour déclarer à Python, dans le corps d'une fonction, que la variable qui sera utilisée doit être considérée comme globale, on utilise le mot-clé **global**. On le place généralement après la définition de la fonction, juste en-dessous de la docstring, cela permet de retrouver rapidement les variables globales sans parcourir tout le code (c'est une simple convention). On précise derrière ce mot-clé le nom de la variable à considérer comme globale :

```
i = 4 # Une variable, nommée i, contenant un entier
```

```
def inc_i():
    """Fonction chargée d'incrémenter i de 1"""
    global i # Python recherche i en dehors de l'espace local de la fonction
    i += 1
```

```
print(i)
inc_i()
print(i)
>>> 4
>>> 5
```

Si vous ne précisez pas à Python que i doit être considérée comme globale, vous ne pourrez pas modifier réellement sa valeur, comme nous l'avons vu plus haut. En précisant global i, Python permet l'accès en lecture et en écriture à cette variable, ce qui signifie que vous pouvez changer sa valeur par affectation.

J'utilise ce mécanisme quand je travaille sur plusieurs classes et fonctions qui doivent s'échanger des informations d'état par exemple. Il existe d'autres moyens mais vous connaissez celui-ci et, tant que vous maîtrisez bien votre code, il n'est pas plus mauvais qu'un autre.

# Nested statements and scope

```
x = 25
def printer():
    x = 50
    return x

print(x)
25

Print(printer())
50
```

LEGB Rule:

L: Local – Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals – Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) – Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) – Names, preassigned in the built-in names module: open, range, SyntaxError, ...

# Nested statements and scope

LEGB Rule:

L: Local – Names assigned in any way within a function (def or lambda), and not declared global in that function.

```
Lambda num:num**2 # here num is local
```

E: Enclosing function locals – Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

```
name = 'THIS IS A GLOBAL STRING'  
def greet():  
    name = 'Sammy' # 2 – Python looks ENCLOSING FUNCTIONS LOCALS and it finds name defined  
    def hello():  
        print('Hello' + name) # 1 - Python looks LOCAL if name is defined and it is not so it goes to next level  
        hello()  
    greet()  
  
Hello Sammy
```

G: Global (module) – Names assigned at the top-level of a module file, or declared global in a def within the file.

```
name = 'THIS IS A GLOBAL STRING' # 2 – Python looks GLOBAL and can find the name  
def greet():  
    # name = 'Sammy'  
    def hello():  
        print('Hello' + name) # 1 - Python looks LOCAL if name is defined and it is not so it goes to next level  
        hello()  
    greet()  
  
Hello THIS IS A GLOBAL STRING
```

# Nested statements and scope

```
# 3-GLOBAL
name = "THIS IS A GLOBAL STRING" # 2 – Python looks GLOBAL and can find the name

def greet():
    # 2-ENCLOSING
    name = 'Sammy'

    def hello():
        # 1-LOCAL
        name = ' IM A LOCAL'
        print('Hello' + name) # 1 - Python looks LOCAL if name is defined and it is not so it goes to next level

    hello()

greet()
>>> Hello IM A LOCAL
```

# Nested statements and scope

```
x = 50

def func(x):
    print(f'X is {x}')
    # LOCAL REASSIGNMENT !
    x = 200
    print(f"I JUST LOCALLY CHANGED X TO {x}")

func(x) # the scope is local to the function
x is 50
I JUST LOCALLY CHANGED X TO 200

print(x)
50
```

Let's imagine that we want to grab the global x (x=50) and reassign it to be NEW VALUE

```
x = 50

def func():
    global x
    print(f'X is {x}')

    # LOCAL REASSIGNMENT ON A GLOBAL VARIABLE!
    x = 'NEW VALUE'
    print(f"I JUST LOCALLY CHANGED GLOBAL X TO {x}")

print(x)
50

func()
x is 50
I JUST LOCALLY CHANGED GLOBAL X TO NEW VALUE
```

# Nested statements and scope

It is much better to take than using global keyword.

```
x = 50

def func(x):
    print(f 'X is {x}')

# LOCAL REASSIGNMENT ON A GLOBAL VARIABLE!
x = 'NEW VALUE'
print(f"I JUST LOCALLY CHANGED GLOBAL X TO {x}")
return x

print(x)
50

x = func(x)
x is 50
I JUST LOCALLY CHANGED GLOBAL X TO NEW VALUE

x
'NEW VALUE'
```

# Résumé

## En résumé

LEGB Rule:

L: Local – Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals – Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) – Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) – Names, preassigned in the built-in names module: open, range, SyntaxError, ...

Les variables locales définies avant l'appel d'une fonction seront accessibles, depuis le corps de la fonction, en lecture seule.

Une variable locale définie dans une fonction sera supprimée après l'exécution de cette fonction.

On peut cependant appeler les attributs et méthodes d'un objet pour le modifier durablement.

Les variables globales se définissent à l'aide du mot-clé global suivi du nom de la variable préalablement créée.

Les variables globales peuvent être modifiées depuis le corps d'une fonction (à utiliser avec prudence).

# Structures conditionnelles

<https://www.teclado.com/30-days-of-python/python-30-day-5-conditionals-booleans>

# Structures conditionnelles

- Les variables permettent de conserver dans le temps des données de votre programme.
- Vous pouvez vous servir de ces variables pour différentes choses : les afficher, faire des calculs avec, etc.
- Pour affecter une valeur à une variable, on utilise la syntaxe `nom_de_variable = valeur`.
- Il existe différents types de variables, en fonction de l'information que vous désirez conserver : int, float, chaîne de caractères etc.
- Pour afficher une donnée, comme la valeur d'une variable par exemple, on utilise la fonction `print`.

# If, elif et else

```
a = 1
if a > 0: # Positif
    print("a est positif.")
elif a < 0: # Négatif
    print("a est négatif.")
else: # Nul
    print("a est nul.")
```

# De nouveaux operateurs

Operateur	Signification litterale
<	Strictement inferieur à
>	Strictement superieurs à
<=	Inferieur ou égal à
>=	Supérieur ou égal à
==	Égal à
!=	Different de

```
a = 0
a == 5
False
a > -8
True
a != 33.19
True
```

# Les mots-clés and, or et not

```
a = 5
if a >= 2 and a <= 8:
    print("a est dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")

if a < 2 or a > 8:
    print("a n'est pas dans l'intervalle.")
else:
    print("a est dans l'intervalle.")

>>> a est dans l'intervalle
>>> a est dans l'intervalle
```

```
majeur = False
if majeur is not True:
    print("Vous n'êtes pas encore majeur.")

>>> Vous n'êtes pas encore majeur.
```

# Année bissextile

```
# Programme testant si une année, saisie par l'utilisateur, est bissextile ou non

annee = input("Saisissez une année : ") # On attend que l'utilisateur saisisse l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas saisi un nombre

if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")
```

# Résumé

- Les conditions permettent d'exécuter certaines instructions dans certains cas, d'autres instructions dans un autre cas.
- Les conditions sont marquées par les mot-clés **if** (« si »), **elif** (« sinon si ») et **else** (« sinon »).
- Les mot-clés **if** et **elif** doivent être suivis d'un test (appelé aussi prédicat).
- Les booléens sont des données soit vraies (**True**) soit fausses (**False**).

# Les chaines de caractères

<https://www.teclado.com/30-days-of-python/python-30-day-2-strings-variables>



Day 2 Strings, Variables, and Getting Input from Users.pdf

<https://www.teclado.com/30-days-of-python/python-30-day-3-string-formatting>



Day 3 Formatting Strings and Processing User Input.pdf

# Strings 1/25



Complete Python 3 Bootcamp

- Strings are sequences of characters, using the syntax of either single quotes or double quotes:
  - **'hello'**
  - **"Hello"**
  - **" I don't do that "**



# Strings 2/25



Complete Python 3 Bootcamp

- Because strings are **ordered sequences** it means we can use **indexing** and **slicing** to grab sub-sections of the string.
- Indexing notation uses [ ] notation after the string (or variable assigned the string).
- Indexing allows you to grab a single character from the string...

# Strings indexing 3/25



Complete Python 3 Bootcamp

- These actions use [ ] square brackets and a number index to indicate positions of what you wish to grab.



**Character:** h e l l o

**Index:** 0 1 2 3 4

# Strings reverse indexing 4/25



Complete Python 3 Bootcamp

- These actions use [ ] square brackets and a number index to indicate positions of what you wish to grab.



**Character :** h e l l o

**Index :** 0 1 2 3 4

**Reverse Index:** 0 -4 -3 -2 -1

# Strings slicing 5/25



Complete Python 3 Bootcamp

- Slicing allows you to grab a subsection of multiple characters, a “slice” of the string.
- This has the following syntax:
  - **[start:stop:step]**
- **start** is a numerical index for the slice start
- **stop** is the index you will go up to (but not include)
- **step** is the size of the “jump” you take.

# Les chaînes de caractères 6/25



Complete Python 3 Bootcamp



- Often you will want to “inject” a variable into your string for printing. For example:
  - **my\_name = “Jose”**
  - **print(“Hello ” + my\_name)**
- There are multiple ways to format strings for printing variables in them.
- This is known as string interpolation.



# Méthodes des chaînes 7/25

Quelques méthodes pour les chaînes(str) en python:

Une méthode de chaîne travaille sur une copie, et pas sur la chaîne elle-même.

`str.upper(), str.lower(), str.capitalize(), str.title(),  
str.center(<largeur>, <caractere_remplissage>)`

`str.strip()` ex: " bonjour ".strip(" ujor") -> 'bon' (le strip examine la chaîne dans les 2 sens et s'arrête dès qu'il ne peut plus matcher, dans cette exemple de la gauche vers la droite il match les deux espaces puis plus rien car il n'y a pas de b dans " ujor", il s'arrête donc là, de la droite vers la gauche il match les espaces puis le r, le o, le j et le u, puis il arrive au n et ne peut pas le matcher il s'arrête donc là.

`str.rstrip()` ex: " bonjour ".rstrip(" ujor") -> ' bon'

`str.lstrip()` ex: " bonjour ".lstrip(" ujor") -> 'bonjour '

\* paramètres optionnels

`str.find(<chaîne>, <debut>*, <fin>*)`  
`str.index(<chaîne>, <debut>*, <fin>*)`  
`str.replace(<ancienne>, <nouvelle>, <occurrences>*)`

`str.isdigit()` ex: "100".isdigit() it will return True

`str.isalpha()`, `str.isdecimal()`, `str.isnumeric()`

`str.isalphanum`

`str.islower()`, `str.isupper()`, `str.capitalize()`, `str.title`

`str.isidentifier()`, `str.iskeyword()`

`str.endswith`, ex: "photo.jpg".endswith("jpg") it will return True

# Les chaînes de caractères 8/25

```
chaine = str() # Crée une chaîne vide
# On aurait obtenu le même résultat en tapant:
chaine = ""

while chaine.lower() != "Q":
    print("Tapez 'Q' pour quitter...")
    chaine = input()

print("Merci !")
```

# Majuscule/Minuscule 9/25

string.upper() -> convertie toute la string en majuscule  
string.lower() -> convertie toute la string en minuscule  
string.capitalize() -> La premiere lettre sera en majuscule  
string.title() -> La premiere letter de chaque mot sera en majuscule  
string.strip() -> on retire les espaces en debut et fin de chaine

```
chaine = "une chaine en minuscules"

chaine.upper() # Mettre en majuscules
'UNE CHAINE EN MINUSCULES'

chaine.lower() # Mettre en minuscule
'une chaine en minuscules'

chaine.capitalize() # La première lettre de la chaine en majuscule
'Une chaine en minuscules'

chaine.title() # La première lettre de chaque mot en majuscule
'Une Chaine En Minuscules'

espaces = " une chaine avec des espaces "
espaces.strip() # On retire les espaces au début et à la fin de la chaîne
'une chaine avec des espaces'

titre = "introduction"
titre.upper().center(20)
' INTRODUCTION '
```

# Tester les propriétés d'une chaîne 10/25

string.isupper() -> retourne un boolean  
string.islower() -> retourne un boolean  
string.isspace() -> retourne un boolean  
string.istitle() -> retourne un boolean  
String.isdigit() -> retourne un boolean

```
In [1]: "bonjour".islower()  
Out[1]: True
```

```
In [2]: "Bonjour".islower()  
Out[2]: False
```

```
In [3]: "Bonjour".istitle()  
Out[3]: True
```

```
In [4]: "Bonjour tout le monde".istitle()  
Out[4]: False
```

```
In [5]: '50'.isdigit()  
Out[5]: True
```

```
In [6]: 'a'.isdigit()  
Out[6]: False
```

# String formatting in Python

<https://blog.teclado.com/learn-python-string-formatting-in-python/>



String formatting in Python.pdf

# Formater et afficher une chaîne 11/25



Day 3 Formatting Strings and Processing User Input.pdf

<https://www.teclado.com/30-days-of-python/python-30-day-3-string-formatting>

## Première syntaxe

```
chaine = "Bonjour tout le monde !"  
print(chaine)  
  
prenom = "Paul"  
nom = "Dupont"  
age = 21  
print("Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom, nom, age))  
>> Je m'appelle Paul Dupont et j'ai 21 ans.  
  
nouvelle_chaine = "Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom, nom, age)
```

## Seconde syntaxe

```
#formatage d'une adresse  
adresse = """  
{no_rue}, {nom_rue}  
{code_postal} {nom_ville} ({pays})  
""".format(no_rue=5, nom_rue="rue des Postes", code_postal=75003, nom_ville="Paris", pays="France")  
print(adresse)
```

## Troisième syntaxe

```
no_rue=5, nom_rue="rue des Postes", code_postal=75003, nom_ville="Paris", pays="France"  
print(f"adresse : {no_rue}, {nom_rue}, {code_postal}, {nom_ville}, ({pays})")
```

## Quatrième syntaxe

```
longer_phrase="Hello, {}. Today is {}."  
formatted = longer_phrase.format("Rolf ", "Monday")  
print(formatted)  
>> Hello, Rolf. Today is Monday.
```

# Float formatting follows « value:width.precision f} » 12/25

```
result = 100/777
0,1287001287001287
print("The result was {r}".format(r=result))
The result was 0,1287001287001287

print("The result was {r:1.3f}".format(r=result))
The result was 0,129

print("The result was {r:10.3f}".format(r=result))
The result was      0,129

print("My 10 character, four decimal number is:{0:10.4f}".format(num))
print(f"The result was {result:{10}.{6}}")
My 10 character, four decimal number is: 0.12870
My 10 character, four decimal number is: 0.12870
```

Note that with f-strings, precision refers to the total number of digits, not just those following the decimal. This fits more closely with scientific notation and statistical analysis. Unfortunately, f-strings do not pad to the right of the decimal, even if precision allows it:

# Float formatting follows 13/25

```
num = 23.45
print("My 10 character, four decimal number is:{0:10.4f}".format(num))
print(f"My 10 character, four decimal number is:{num:{10}.{6}}")
My 10 character, four decimal number is: 23.4500
My 10 character, four decimal number is: 23.45
```

If this becomes important, you can always use `.format()` method syntax inside an f-string:

```
num = 23.45
print("My 10 character, four decimal number is:{0:10.4f} ".format(num))
print(f"My 10 character, four decimal number is:{num:10.4f}")
```

```
My 10 character, four decimal number is: 23.4500
My 10 character, four decimal number is: 23.4500
```

For more info on formatted string literals visit [https://docs.python.org/3/reference/lexical\\_analysis.html#f-strings](https://docs.python.org/3/reference/lexical_analysis.html#f-strings)

That is the basics of string formatting!

# Alignment, padding with .format() 14/25

Within the curly braces you can assign field lengths, left/right alignments, rounding parameters and more

```
print('{0:8} | {1:9}'.format('Fruit', Quantity))
print('{0:8} | {1:9}'.format('Apples', 3.))
print('{0:8} | {1:9}'.format('Oranges', 10))
>>> Fruit      | Quantity
>>> Apples    |      3.0
>>> Oranges   |      10
```

Within the curly braces you can assign field lengths, left/right alignments, rounding parameters and more

```
print('{0:<8} | {1:^8} | {2:>8}'.format('Left', 'Center', 'Right'))
print('{0:<8} | {1:^8} | {2:>8}'.format(11, 22, 33))
Left  | Center | Right
11   |  22  |  33
```

# Alignment, padding with .format() 15/25

Field widths and float precision are handled in a way similar to placeholders. The following two print statements are equivalent:

```
print('This is my ten-character, two-decimal number:%10.2f' %13.579)
print('This is my ten-character, two-decimal number:{0:10.2f}'.format(13.579))
>>> This is my ten-character, two-decimal number: 13.58
>>> This is my ten-character, two-decimal number: 13.58
```

Note that there are 5 spaces following the colon, and 5 characters taken up by 13.58, for a total of ten characters.

For more information on the string .format() method visit <https://docs.python.org/3/library/string.html#formatstrings>

HERE IS AN AWESOME SOURCE FOR PRINT FORMATTING:

<https://pyformat.info/>

# La concaténation des chaînes 16/25

```
prenom = "Paul"
message = "Bonjour"
chaine_complete = message + prenom # On utilise le symbole '+' pour concaténer deux chaînes
print(chaine_complete)
>>> BonjourPaul

# Pas encore parfait, il manque un espace
# Qu'à cela ne tienne !
chaine_complete = message + " " + prenom
print(chaine_complete)
>>> Bonjour Paul
```

Concaténation d'une string et d'un entier, il faut caster l'entier en string

```
age = 21
message = "J'ai " + str(age) + " ans."
print(message)
>>> J'ai 21 ans.
```

# Parcours et sélection des chaînes 17/25

Accéder aux caractères d'une chaîne:

```
chaine = "Salut les ZEROS !"  
chaine[0] # Première lettre de la chaîne  
'S'  
chaine[2] # Troisième lettre de la chaîne  
' '  
chaine[-1] # Dernière lettre de la chaîne  
'!'
```

On peut obtenir la longueur de la chaîne (le nombre de caractères qu'elle contient) grâce à la fonction len.

```
chaine = "Salut"  
len(chaine)  
5
```

# Méthode de parcours par while 18/25

```
chaine = "Salut"
i = 0 # On appelle l'indice 'i' par convention
while i < len(chaine):
    print(chaine[i]) # On affiche le caractère à chaque tour de boucle
    i += 1
```

# Slicing 19/25

```
présentation = "salut"
présentation[0:2] # On sélectionne les deux premières lettres
>>> 'sa'
présentation[2:len(présentation)] # On sélectionne la chaîne sauf les deux premières lettres
>>> 'lut'

présentation[:2] # Du début jusqu'à la troisième lettre non comprise
>>> 'sa'
présentation[2:] # De la troisième lettre (comprise) à la fin
>>> 'lut'

mot = "lac"
mot = "b" + mot[1:]
print(mot)
>>> bac
```

Pour remplacer des lettres, cela paraît un peu lourd en effet. Et d'ailleurs on s'en sert assez rarement pour cela. Pour rechercher/remplacer, nous avons à notre disposition les méthodes **count**, **find** et **replace**.

# La méthode count 20/25

```
In [1]: "Bonjour le jour".count("jour")
Out[1]: 2
```

```
In [2]: "Bonjour le jour".count(" jour")
Out[2]: 1
```

# Trouver une chaîne 21/25

```
In [1]: "Bonjour le jour".find("jour")
```

```
Out[1]: 3 # donne la position de la première occurrence de jour
```

```
In [2]: "Bonjour le jour".index("jour")
```

```
Out[2]: 3 # donne également la position de la première occurrence de jour
```

# Si la chaîne n'existe pas find et index ne se comporte alors, pas de la même façon:

```
In [3]: "Bonjour le jour".find("soir")
```

```
Out[3]: -1
```

```
In [4]: "Bonjour le jour".index("soir")
```

```
Out[4]: ValueError: substring not found
```

```
In [5]: "Bonjour le jour".rfind("jour")
```

```
Out[5]: 11 # Cette fois-ci on a 11 car la première occurrence en partant de la gauche est à 11 caractère en comptant de la droite.
```

Notons que *l*find n'existe pas!

# Chercher au début et à la fin 22/25

```
In [1]: "image.png".endswith(".png")
```

```
Out[1]: True
```

```
In [2]: "image.png".endswith(".jpg")
```

```
Out[2]: False
```

```
In [3]: "image.png".startswith(".image")
```

```
Out[3]: True
```

```
In [4]: "Bonjour le jour".index("video")
```

```
Out[4]: False
```

# Liste de toutes les méthodes de chaînes de caractères 23/25

<b>capitalize()</b> Converti le premier caractère de la chaîne en majuscule	chaîne sont des lettres de l'alphabet	argument	<b>ljust()</b> Retourne une version justifiée par la gauche de la chaîne	en argument et retourne une liste
<b>casefold()</b> Converti la phrase en minuscule	tous les caractères dans la chaîne sont de type décimal		<b>lower()</b> Converti la chaîne en minuscule	<b>rstrip()</b> Supprime tous les caractères passés individuellement en partant de la droite de la chaîne
<b>center()</b> Retourne une chaîne de caractères centrée	tous les caractères dans la chaîne sont des nombres		<b>lstrip()</b> Supprime tous les caractères passés individuellement en partant de la gauche de la chaîne	<b>split()</b> Sépare la chaîne de caractères sur les caractères passés en argument et retourne une liste
<b>count()</b> Retourne le nombre de fois que la chaîne spécifiée est trouvée	chaîne sont des nombres			<b>splitlines()</b> Sépare la chaîne de caractères sur les retours à la ligne
<b>encode()</b> Retourne une version encodée de la chaîne et tiret du bas)	<b>isidentifier()</b> Retourne True si la chaîne est un identifiant (caractères alphanumériques		<b>maketrans()</b> Retourne une table de caractères sur les retours à la ligne mapping à utiliser avec la méthode translate	
<b>endswith()</b> Retourne True si la chaîne se termine par la valeur spécifiée	<b>islower()</b> Retourne True si tous les caractères sont en minuscule		<b>partition()</b> Retourne un tuple composé de trois éléments	<b>startswith()</b> Retourne True si la chaîne commence par la valeur spécifiée
<b>expandtabs()</b> Change la taille des tabulations de la chaîne	<b>isnumeric()</b> Retourne True si tous les caractères sont numériques		<b>replace()</b> Remplace un élément de la chaîne par un autre	<b>strip()</b> Supprime les caractères spécifiés du début et de la fin de la chaîne
<b>find()</b> Cherche dans la chaîne de caractère la valeur spécifiée et retourne l'index correspondant	<b>isprintable()</b> Retourne True si tous les caractères sont imprimables		<b>rfind()</b> Cherche dans la chaîne de caractère la valeur spécifiée et retourne l'index correspondant (en partant de la droite)	<b>swapcase()</b> Change la casse (les majuscules deviennent minuscules et vice-versa)
<b>format()</b> Permet de formater une chaîne de caractères	<b>isspace()</b> Retourne True si tous les caractères sont des espaces		<b>rindex()</b> Cherche dans la chaîne de caractère la valeur spécifiée et retourne l'index correspondant (en partant de la droite)	<b>title()</b> Converti la première lettre de chaque mot en majuscule
<b>index()</b> Cherche dans la chaîne de caractère la valeur spécifiée et retourne l'index correspondant	<b>istitle()</b> Retourne True si la première lettre de chaque mot est en majuscule			<b>translate()</b> Retourne une chaîne traduite (avec une table de mapping)
<b>isalnum()</b> Retourne True si tous les caractères dans la chaîne sont de type alphanumériques	<b>isupper()</b> Retourne True si tous les caractères sont en majuscule		<b>rjust()</b> Retourne une version justifiée par la droite de la chaîne	<b>upper()</b> Converti une chaîne en majuscule
<b>isalpha()</b> Retourne True si tous les caractères dans la	<b>join()</b> Joins avec le caractère spécifié tous les éléments d'un itérable passé en		<b>rpartition()</b> Retourne un tuple composé de trois éléments	<b>zfill()</b> Rempli la chaîne spécifiée avec des 0 (de la longueur indiquée en argument)
			<b>rsplit()</b> Sépare la chaîne de caractères sur les caractères passés en argument	

# Une application pratique 24/25

+ Notions abordées: le module string, le module random.

Dans cet exercice, nous allons créer un générateur de mot de passe aléatoire.

A l'aide du module string et du module random, vous allez générer un mot de passe aléatoire de la longueur spécifiée dans la variable 'taille' (ici, 8).

Votre mot de passe doit pouvoir contenir des lettres minuscules, n'importe quel chiffre de 0 à 9 et n'importe quel caractère spécial(!#\$%& etc...)

+ Astuces: Le module string vous permet de récupérer des suites de lettres, de nombres d'ou de caractères spéciaux.

!!!! Attention cette exercice requiert des connaissances sur : compréhension de liste, les objets !!!!

*import string*

*# La première chose à faire est de récupérer toutes les lettres de l'alphabet, les nombres et tous les symboles:*

*symboles = string.ascii\_letters + string.digits + string.punctuation*

*# On utilise ensuite le module random et la fonction choice. Cette fonction va nous permettre de sélectionner une lettre aléatoire parmi la chaîne de caractère "symbole" que nous avons déclaré plus haut.*

*# Pour répéter l'opération autant de fois que nécessaire, on utilise une boucle et une compréhension de liste:*

*mot\_de\_passe = ".join(random.choice(symboles) for i in range(taille))*

*print(mot\_de\_passe)*

*# POINTS IMPORTANTS A RETENIR*

*# 1 - Pour récupérer toutes les lettres majuscules et minuscules de l'alphabet, on utilise la constante "ascii\_letters" du module "string".*

*# 2 - Pour récupérer tous les nombres, on utilise la constante digits du module string.*

*# 3 - Pour récupérer tous les symboles du clavier, on utilise la constante "punctuation" du module "string"*

*# 4 - Pour récupérer une lettre aléatoire dans une chaîne de caractère, on utilise la fonction "choice" du*

*# Fonction pour vérifier que le mot de passe est bien formé*

*def verificateur\_mdp(mdp):*  
*if len(mdp) >= 8 and any(i.isupper() for i in mdp) and len([n for n in mdp if n.isdigit()]) >= 2:*  
*return True*  
*return False*

*succes = verificateur\_mdp(mot\_de\_passe)*

*print(succes)*

# Remplir de zero 25/25

```
In [1]: '9'.zfill(4)  
Out[1]: '0009'
```

```
In [2]: '9'.zfill(3)  
Out[2]: '009'
```

```
In [3]: for i in range(100):  
    print(str(i).zfill(4))
```

# Résumé

- Indexing, reverse indexing, slicing
- Les méthodes de chaines de caractères.
- Tester les propriétés d'une chaines de caractères.
- Formater et afficher une chaine.
- Alignement and padding with .format
- Parcours et sélection de chaines
- Il est tout à fait possible de sélectionner une partie de la chaîne grâce au code suivant : chaine[indice\_debut:indice\_fin].
- Les méthodes: count, index, find, endswith, startswith
- Liste des méthodes disponibles
- Remplir de zero avec zfill()

# Liste[]

<https://www.teclado.com/30-days-of-python/python-30-day-4-lists-tuples>



Day 4 Basic Python Collections.pdf

# Création de listes[] 1/21

- Lists are ordered sequences that can hold a variety of object types.
- They use [] brackets and commas to separate objects in the list.
  - **[1,2,3,4,5]**
- Lists support indexing and slicing. Lists can be nested and also have a variety of useful methods that can be called off of them.

# Création de listes[] 2/21

```
ma_liste = list() # On crée une liste vide  
type(ma_liste)  
<class 'list'>  
ma_liste  
[]
```

Quand vous affichez la liste, vous pouvez constater qu'elle est vide. Entre les crochets (qui sont les délimiteurs des listes en Python), il n'y a rien. On peut également utiliser ces crochets pour créer une liste.

```
ma_liste = [] # On crée une liste vide
```

on peut également créer une liste non vide, en lui indiquant directement à la création les objets qu'elle doit contenir.

```
ma_liste = [1, 2, 3, 4, 5] # Une liste avec cinq objets  
print(ma_liste)  
[1, 2, 3, 4, 5]
```

# Création de listes[] 3/21

La liste que nous venons de créer compte cinq objets de type int. Ils sont classés par ordre croissant. Mais rien de tout cela n'est obligatoire.

- Vous pouvez faire des listes de toute longueur.
- Les listes peuvent contenir n'importe quel type d'objet.
- Les objets dans une liste peuvent être mis dans un ordre quelconque. Toutefois, la structure d'une liste fait que chaque objet a sa place et que l'ordre compte.

```
ma_liste = [1, 3.5, "une chaine", []]
```

on peut également créer une liste non vide, en lui indiquant directement à la création les objets qu'elle doit contenir.

```
ma_liste = ['c', 'f', 'm']
ma_liste[0] # On accède au premier élément de la liste
'c'
ma_liste[2] # Troisième élément
'm'
ma_liste[1] = 'Z' # On remplace 'f' par 'Z'
ma_liste
['c', 'Z', 'm']
```

Comme vous pouvez le voir, on accède aux éléments d'une liste de la même façon qu'on accède aux caractères d'une chaîne de caractères : on indique entre crochets l'indice de l'élément qui nous intéresse.

Contrairement à la classe **str**, la classe **list** vous permet de remplacer un élément par un autre. Les listes sont en effet des types dits **mutables**.

# Opérations sur les listes 4/21

- append
- clear
- copy
- count
- extend
- index
- insert
- pop
- remove
- reverse
- sort

# Insérer un élément dans la liste[] 5/21

Nous allons passer assez rapidement sur cette seconde méthode. On peut, très simplement, insérer un objet dans une liste, à l'endroit voulu. On utilise pour cela la méthode insert.

```
ma_liste = [1, 2, 3]
ma_liste.append(56) # On ajoute 56 à la fin de la liste
print(ma_liste)
[1, 2, 3, 56]
```

```
ma_liste = ["a", "b", "d"]
ma_liste.insert(2,"c") # On ajoute c en seconde position
print(ma_liste)
>>> [a, b, c, d]
```

Quand on demande d'insérer c à l'indice 2, la méthode va décaler les objets d'indice supérieur ou égal à 2. ca va donc s'intercaler entre b et d.

# Concaténation de listes[] 6/21

Voici les différentes façons de concaténer des listes. Vous pouvez remarquer l'opérateur + qui concatène deux listes entre elles et renvoie le résultat. On peut utiliser += assez logiquement pour étendre une liste. Cette façon de faire revient au même qu'utiliser la méthode extend

```
ma_liste1 = [3, 4, 5]
ma_liste2 = [8, 9, 10]

ma_liste1.extend(ma_liste2) # On insère ma_liste2 à la fin de ma_liste1
print(ma_liste1)
>>> [3, 4, 5, 8, 9, 10]

ma_liste1 = [3, 4, 5]
print(ma_liste1 + ma_liste2)
>>> [3, 4, 5, 8, 9, 10]

ma_liste1 += ma_liste2 # Identique à extend
print(ma_liste1)
>>> [3, 4, 5, 8, 9, 10]
```

# Suppression d' élément d'une listes[] 7/21

Nous allons voir trois méthodes pour supprimer des éléments d'une liste :

- le mot-clé del;
- la méthode remove.
- La méthode pop

```
ma_liste = [-5, -2, 1, 4, 7, 10]
del ma_liste[0] # On supprime le premier élément de la liste
print(ma_liste)
>>> [-2, 1, 4, 7, 10]
```

```
del ma_liste[0:3] # On supprime les 3 premiers éléments
print(ma_liste)
>>> [7, 10]
```

```
ma_liste = [31, 32, 33, 34, 35]
ma_liste.remove(32)
print(ma_liste)
>>> [31, 33, 34, 35]
```

Notons que la méthode remove on désigne l'élément de la liste que l'on souhaite supprimer, ici 32

```
ma_liste = [31, 32, 33, 34, 35]
print(ma_liste.pop())
print(ma_liste.pop(-1))
>>> 35
>>> 34
```

Notons que la méthode pop retourne la valeur de l'élément supprimé et permet de supprimer un élément

```
>> ma_liste = [31, 32, 33, 34, 35]
>>> ma_liste.clear()
>>> []
```

Notons que la méthode pop retourne la valeur de l'élément supprimé et permet de supprimer un élément

# Récupérer l'index d'un élément d'une liste

## 8/21

la méthode index().

```
Characters = ["alivin et les chipmunks", "Babar", "betty boop", "calimero", "casper", "le chat pote", "kirikou"]

if "Babar" in Characters:
    print(Characters.index("Babar"))
>>> 1
```

la méthode count(), retourne le nombre d'occurrence dans la liste:

```
letters = ["a", "b", "c"]
print(letters.count("d"))
print(letters.count("a"))
>>> 0
>>> 1
```

# Liste unpacking 9/21

```
numbers = [1, 2, 3]
first, second, third = numbers
print(first, second, third)
>>> 1 2 3
```

```
numbers = [1, 2, 3, 4, 4, 4, 4, 4]
first, second, third, *other = numbers
print(first, second, third)
>>> 1 2 3
print(other)
>>> [4, 4, 4, 4]
```

# Trie de liste 10/21

```
numbers = [3, 51, 2, 8, 6] # with the method .sort() the original list is modified
numbers.sort()
print(numbers)
[2, 3, 6, 8, 51]

numbers.sort(reverse=True)
print(numbers)
[51, 8, 6, 3, 2]

print(sorted(numbers)) # with sorted function the original list is not modified
print(sorted(numbers, reverse=True))
```

Notons, qu'il n'est pas possible de trier une liste de tuples avec la méthode **.sort** ou la fonction **sorted** ci-dessus cela ne fonctionne pas. Si l'on désire trier une liste de tuple il faudra faire appel à la fonction **lambda** (voir le chapitre suivant pour plus de détails).

```
items = [
    ("Product1", 10),
    ("Product2", 9),
    ("Product3", 12),
]

items.sort(key=lambda item: item[1])
print(items)
```

# Le parcours des listes[] 11/21

```
ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
i = 0 # Notre indice pour la boucle while
while i < len(ma_liste):
    print(ma_liste[i])
    i += 1 # On incrémente i, ne pas oublier !

>>> a
>>> b
>>> c
>>> d
>>> e
>>> f
>>> g
>>> h
# Cette méthode est cependant préférable
for elt in ma_liste: # elt va prendre les valeurs successives des éléments de ma_liste
    print(elt)

>>> a
>>> b
>>> c
>>> d
>>> e
>>> f
>>> g
>>> H

# on peut utiliser la propriété unpacking dans la boucle for.
letters=["a", "b", "c"]
for index, letter in enumerate(letters):
    print(index, letter)
```

# Python list slice syntax fun 12/21

```
# Python's list slice syntax can be used without indices  
# for a few fun and useful things:
```

```
liste = ["Maxime", "Martine", "Christopher", "Carlos", "Michael", "Eric"]  
  
trois_premiers = liste[:3]  
  
trois_derniers = liste[-3:]  
  
milieu = liste[1:-1]  
  
premier_dernier = liste[::-len(liste)-1]
```

```
# You can clear all elements from a list:
```

```
lst = [1, 2, 3, 4, 5]  
del lst[:]  
print(lst)  
=> []
```

```
# You can replace all elements of a list  
# without creating a new list object:
```

```
a = lst  
lst[:] = [7, 8, 9]  
print(lst)  
=> [7, 8, 9]  
print(a)  
=> [7, 8, 9]  
print(a is lst)  
=> True
```

```
# You can also create a (shallow) copy of a list:
```

```
b = lst[:]  
print(b)  
=> [7, 8, 9]  
print(b is lst)  
=> False
```

# Python split, join and slice 13/21

<https://www.teclado.com/30-days-of-python/python-30-day-7-split-join>



Day 7 split, join, and Slices.pdf

# Python's Partition Method for strings 14/22

<https://blog.teclado.com/python-partition-method/>



Python's Partition Method for Strings.pdf

# Entre chaines et listes[] 15/22

## Des chaines aux listes

Pour « convertir » une chaîne en liste, on va utiliser une méthode de chaîne nommée `split`(« éclater » en anglais). Cette méthode prend un paramètre qui est une autre chaîne, souvent d'un seul caractère, définissant comment on va découper notre chaîne initiale.

```
ma_chaine = "Bonjour à tous"  
ma_chaine.split(" ")  
['Bonjour', 'à', 'tous']
```

On passe en paramètre de la méthode `split` une chaîne contenant un unique espace. La méthode renvoie une liste contenant les trois mots de notre petite phrase. Chaque mot se trouve dans une case de la liste.

C'est assez simple en fait : quand on appelle la méthode `split`, celle-ci découpe la chaîne en fonction du paramètre donné. Ici la première case de la liste va donc du début de la chaîne au premier espace (non inclus), la deuxième case va du premier espace au second, et ainsi de suite jusqu'à la fin de la chaîne.

Sachez que `split` possède un paramètre par défaut, un code qui représente les espaces, les tabulations et les sauts de ligne. Donc vous pouvez très bien faire `ma_chaine.split()`, cela revient ici au même.

# Entre chaines et listes[] 16/22

## Des listes aux chaînes

Voyons l'inverse à présent, c'est-à-dire si on a une liste contenant plusieurs chaînes de caractères que l'on souhaite fusionner en une seule. On utilise la méthode de chaîne join(« joindre » en anglais). Sa syntaxe est un peu surprenante :

```
ma_liste = ['Bonjour', 'à', 'tous']
" ".join(ma_liste)
'Bonjour à tous'
```

En paramètre de la méthode join, on passe la liste des chaînes que l'on souhaite « ressouder ». La méthode va travailler sur l'objet qui l'appelle, ici une chaîne de caractères contenant un unique espace. Elle va insérer cette chaîne entre chaque paire de chaînes de la liste, ce qui au final nous donne la chaîne de départ, « Bonjour à tous ».

# Une application pratique 17/22

Admettons que nous ayons un nombre flottant dont nous souhaitons afficher la partie entière et les trois premières décimales uniquement de la partie flottante. Autrement dit, si on a un nombre flottant tel que « 3.9999999999998 », on souhaite obtenir comme résultat « 3.999 ». D'ailleurs, ce serait plus joli si on remplaçait le point décimal par la virgule, à laquelle nous sommes plus habitués.

```
def afficher_flottant(flottant):
    """Fonction prenant en paramètre un flottant et renvoyant une chaîne de caractères représentant la troncature de ce
    nombre. La partie flottante doit avoir une longueur maximum de 3 caractères.
    De plus, on va remplacer le point décimal par la virgule"""

    if type(flottant) is not float:
        raise TypeError("Le paramètre attendu doit être un flottant")
    flottant = str(flottant)
    partie_entiere, partie_flaotante = flottant.split(".")
    # La partie entière n'est pas à modifier
    # Seule la partie flottante doit être tronquée
    return ",".join([partie_entiere, partie_flaotante[:3]])

print(afficher_flottant(12.586))
```

En paramètre de la méthode join, on passe la liste des chaînes que l'on souhaite « ressouder ». La méthode va travailler sur l'objet qui l'appelle, ici une chaîne de caractères contenant un unique espace. Elle va insérer cette chaîne entre chaque paire de chaînes de la liste, ce qui au final nous donne la chaîne de départ, « Bonjour à tous ».

# Les listes[] et paramètres de fonctions

## 18/22

Les fonctions dont on ne connaît pas à l'avance le nombre de paramètres

Vous devriez tout de suite penser à la fonction print: on lui passe une liste de paramètres qu'elle va afficher, dans l'ordre où ils sont placés, séparés par un espace (ou tout autre délimiteur choisi).

```
def fonction(*parametres):
```

```
def fonction_inconnue(*parametres):
    """Test d'une fonction pouvant être appelée avec un nombre variable de paramètres"""
    print("J'ai reçu : {}".format(parametres))

fonction_inconnue() # On appelle la fonction sans paramètre
>>> J'ai reçu : ().

fonction_inconnue(33)
>>> J'ai reçu : (33,).

fonction_inconnue('a', 'e', 'f')
>>> J'ai reçu : ('a', 'e', 'f').

var = 3.5
fonction_inconnue(var, [4], "...")
>>> J'ai reçu : (3.5, [4], '...').
```

# Les listes[] et paramètres de fonctions

## 19/22

Vous pouvez bien entendu définir une fonction avec plusieurs paramètres qui doivent être fournis quoi qu'il arrive, suivis d'une liste de paramètres variables :

```
def fonction_inconnue(nom, prenom, *commentaires):
```

Dans cet exemple de définition de fonction, vous devez impérativement préciser un nom et un prénom, et ensuite vous mettez ce que vous voulez en commentaire, aucun paramètre, un, deux... ce que vous voulez.

Si on définit une liste variable de paramètres, elle doit se trouver après la liste des paramètres standard.

Au fond, cela est évident. Vous ne pouvez avoir une définition de fonction comme `def fonction_inconnue(*parametres, nom, prenom)`. En revanche, si vous souhaitez avoir des paramètres nommés, il faut les mettre après cette liste. Les paramètres nommés sont un peu une exception puisqu'ils ne figureront de toute façon pas dans le tuple obtenu. Voyons par exemple la définition de la fonction print:

```
def print(*values, sep=' ', end='\n', file=sys.stdout):
```

# Exercice 20/22

Petit exercice : faire une fonction afficher identique à print, c'est-à-dire prenant un nombre indéterminé de paramètres, les affichant en les séparant à l'aide du paramètre nommé sep et terminant l'affichage par la variable fin. Notre fonction afficher ne comptera pas de paramètre file. En outre, elle devra passer par print pour afficher (on ne connaît pas encore d'autres façons de faire). La seule contrainte est que l'appel à print ne doit compter qu'un seul paramètre non nommé. Autrement dit, avant l'appel à print, la chaîne devra avoir été déjà formatée, prête à l'affichage.

voici la définition de la fonction, ainsi que la doc string:

```
def afficher(*parametres, sep=' ', fin='\n'):
    """Fonction chargée de reproduire le comportement de print.

    Elle doit finir par faire appel à print pour afficher le résultat.
    Mais les paramètres devront déjà avoir été formatés.
    On doit passer à print une unique chaîne, en lui spécifiant de ne rien mettre à la fin :

    print(chaine, end=")"""

    # Les paramètres sont sous la forme d'un tuple
    # Or on a besoin de les convertir
    # Mais on ne peut pas modifier un tuple
    # On a plusieurs possibilités, ici je choisis de convertir le tuple en liste
    parametres = list(parametres)
    # On va commencer par convertir toutes les valeurs en chaîne
    # Sinon on va avoir quelques problèmes lors du join
    for i, parametre in enumerate(parametres):
        parametres[i] = str(parametre)
    # La liste des paramètres ne contient plus que des chaînes de caractères
    # À présent on va constituer la chaîne finale
    chaine = sep.join(parametres)
    # On ajoute le paramètre fin à la fin de la chaîne
    chaine += fin
    # On affiche l'ensemble
    print(chaine, end="")
```

# Transformer une liste en paramètres de fonction 21/22

C'est peut-être un peu moins fréquent mais vous devez connaître ce mécanisme puisqu'il complète parfaitement le premier. Si vous avez un tuple ou une liste contenant des paramètres qui doivent être passés à une fonction, vous pouvez très simplement les transformer en paramètres lors de l'appel. Le seul problème c'est que, côté démonstration, je me vois un peu limité.

```
liste_des_parametres = [1, 4, 9, 16, 25, 36]
print(*liste_des_parametres)
>>> 1 4 9 16 25 36
```

Oui je vous l'accorde. Ici l'intérêt ne saute pas aux yeux. Mais un peu plus tard, vous pourrez tomber sur des applications où les fonctions sont utilisées sans savoir quels paramètres elles attendent réellement. Si on ne connaît pas la fonction que l'on appelle, c'est très pratique. Là encore, vous découvrirez cela dans les chapitres suivants ou dans certains projets. Essayez de garder à l'esprit ce mécanisme de transformation.

On utilise une étoile \* dans les deux cas. Si c'est dans une définition de fonction, cela signifie que les paramètres fournis non attendus lors de l'appel seront capturés dans la variable, sous la forme d'un tuple. Si c'est dans un appel de fonction, au contraire, cela signifie que la variable sera décomposée en plusieurs paramètres envoyés à la fonction.

# ■ Pythonic ways of checking if all items in a list are equal 22/22

```
# Pythonic ways of checking if all
# items in a list are equal:

lst = ['a', 'a', 'a']

len(set(lst)) == 1
>>> True

all(x == lst[0] for x in lst)
>>> True

lst.count(lst[0]) == len(lst)
>>> True

# I ordered those from "most Pythonic" to "least Pythonic"
# and "least efficient" to "most efficient".
# The len(set()) solution is idiomatic, but constructing
# a set is less efficient memory and speed-wise.
```

# Résumé

List En résumé :

- Operateur sur les listes: **append, clear, copy, count, extend, index, insert, pop, remove, reverse, sort**
- Il est possible de concaténer des listes: **list1 + list2**
- Il est possible d' étendre une liste: **ma\_liste1.extend(ma\_liste2)** # On insère ma\_liste2 à la fin de ma\_liste1 ou bien **ma\_liste1 += ma\_liste2**
- Il est possible de faire du « **slicing** » sur les listes
- Il est possible de faire du « **unpacking** » sur les listes
- On peut découper une chaîne en fonction d'un séparateur en utilisant la méthode split de la chaîne.
- On peut joindre une liste contenant des chaînes de caractères en utilisant la méthode de chaîne join. Cette méthode doit être appelée sur le séparateur.
- On peut créer des fonctions attendant un nombre inconnu de paramètres grâce à la syntaxe **def fonction\_inconnue(\*parametres):**(les paramètres passés se retrouvent dans le tuple parametres).

# Tuples()

# Les Tuples() 1/5



Complete Python 3 Bootcamp

**Tuples** are very similar to lists. However they have one key difference - **immutability**.

Once an element is inside a tuple, it can not be reassigned.

Tuples use parenthesis: **(1,2,3)**

# Les Tuples() 2/5

Les tuples sont des listes immuables, qu'on ne peut modifier.

Un tuple se définit comme une liste, sauf qu'on utilise comme délimiteur des parenthèses au lieu des crochets.

```
tuple_vide = ()  
tuple_non_vide = (1,) # est équivalent à ci dessous  
tuple_non_vide = 1,  
tuple_avec_plusieurs_valeurs = (1, 2, 5)
```

À la différence des listes, les tuples, une fois créés, ne peuvent être modifiés : on ne peut plus y ajouter d'objet ou en retirer.

Une petite subtilité ici : si on veut créer un tuple contenant un unique élément, on doit quand même mettre une virgule après celui-ci. Sinon, Python va automatiquement supprimer les parenthèses et on se retrouvera avec une variable lambda et non une variable contenant un tuple.

Une fonction renvoyant plusieurs valeurs

Nous ne l'avons pas vu jusqu'ici mais une fonction peut renvoyer deux valeurs ou même plus :

```
def decomposer(entier, divise_par):  
    """Cette fonction retourne la partie entière et le reste de  
    entier / divise_par"""  
  
    p_e = entier // divise_par  
    reste = entier % divise_par  
    return p_e, reste
```

```
partie_entiere, reste = decomposer(20, 3)  
print(partie_entiere, reste)  
>>> 6 2
```

Là encore, on passe par des tuples sans que ce soit indiqué explicitement à Python. Si vous essayez de faire `retour = decomposer(20, 3)`, vous allez capturer un tuple contenant deux éléments : la partie entière et le reste de 20 divisé par 3.

# Les Tuples() 3/3

Les méthodes disponibles pour les tuples:

```
t = ('a', 'a', 'b')
t.count('a')
>>> 2
```

```
t.index('a')
>>> 0
```

```
t.index('b')
>>> 2
```

```
point1 = (1, 2, 3)
point2 = (1, 2) + (3, 4)
point3 = (1, 2) * 3
point4 = tuple([1, 2]) # transform a list in tuple
```

```
print(point1[0]) # we can use index
print(point1[0:2]) # we can get a range of items
x, y, z = point1 # unpacking tuples
```

```
if 10 in point1:      # le mot clef in est disponible avec les tuples comme avec les lists
    print("exists")
```

```
point1.count(1)
```

```
point1.index(1)
```

```
point1.index(3)
```

```
>>> 1
>>> (1, 2)
```

# Résumé

## **Tuple En résumé :**

- Tuples don't have a lot of methods associated with them: count, index
- Tuples are immutable

# Sets

[https://www.teclado.com/30-days-of-python/python-30-day-11-sets?\\_\\_s=wj99fjkc10g15ts6m4rh](https://www.teclado.com/30-days-of-python/python-30-day-11-sets?__s=wj99fjkc10g15ts6m4rh)



Sets.pdf

# Sets



Complete Python 3 Bootcamp

**Sets** are unordered collections of **unique** elements.

Meaning there can only be one representative of the same object.

Let's see some examples!

# Sets

```
myset = set()  
myset.add(1)  
myset  
{1}  
  
myset.add(2)  
myset  
{1, 2}  
  
myset.add(2)  
myset  
{1, 2} # set only accept unique values ! Therefore it is not possible to add the value 2 once again  
  
mylist = [1,1,1,1,1,2,2,2,2,3,3,3,3]  
set(mylist) # ici on caste la liste en set  
{1, 2, 3} # REMEMBER that set doesn't have particular order !!!  
  
mylist = [2,2,2,2,1,1,1,4,4,4,3,3,3]  
set(mylist)  
{2,1,4,3} # as mentionned above set doesn't have order !
```

# Sets

```
numbers = [1, 1, 2, 3, 4]
uniques = set(numbers)
second = {1, 4}
second.add(5)
second.remove(1)
len(second)
print(uniques)

first = set(numbers)
second = {1, 5}

print(first | second) # union of first and second, 2 syntaxes are available | or .union
print(first.union(second))

print(first & second) # intersection of first and second, 2 syntaxes are available & or .intersection
print(first.intersection(second))

print(first - second) # soustraction de deux ensembles, 2 syntaxes are available – or .difference
print(first.difference(second))

print(first ^ second) # symmetric difference (everything which is not in both set), 2 syntaxes are available ^ or .symmetric_difference
print(first.symmetric_difference(second))

If 1 in first:
    print("yes")

!!! Set doesn't have order, index !!! It is an un-ordered collection of unique items we cannot have duplicates
```

# Utilisez des dictionnaires{}

[https://www.teclado.com/30-days-of-python/python-30-day-10-dictionaries?\\_\\_s=wj99fjkc10g15ts6m4rh](https://www.teclado.com/30-days-of-python/python-30-day-10-dictionaries?__s=wj99fjkc10g15ts6m4rh)



Dictionaries.pdf

# Utilisez des dictionnaires{} 1/



Complete Python 3 Bootcamp

- Dictionaries are unordered mappings for storing objects. Previously we saw how lists store objects in an ordered sequence, dictionaries use a key-value pairing instead.
- This key-value pair allows users to quickly grab objects without needing to know an index location.

# Utilisez des dictionnaires{} 2/



Complete Python 3 Bootcamp

- Dictionaries use curly braces and colons to signify the keys and their associated values.

**{'key1':'value1','key2':'value2'}**

- So when to choose a list and when to choose a dictionary?

# Utilisez des dictionnaires{} 3/



Complete Python 3 Bootcamp

- **Dictionaries:** Objects retrieved by key name.

Unordered and can not be sorted.

- **Lists:** Objects retrieved by location.

Ordered Sequence can be indexed or sliced.

# Utilisez des dictionnaires{} - Creation 4/

Le dictionnaire est aussi un objet conteneur. Il n'a quant à lui aucune structure ordonnée, à la différence des listes. De plus, pour accéder aux objets contenus dans le dictionnaire, on n'utilise pas nécessairement des indices mais des clés qui peuvent être de bien des types distincts.

Créer un dictionnaire:

```
mon_dictionnaire = dict()  
type(mon_dictionnaire)  
>>> class 'dict'  
  
mon_dictionnaire  
{  
  
# Du coup, vous devriez trouver la deuxième manière de créer un dictionnaire vide  
mon_dictionnaire = {}  
  
mon_dictionnaire  
{}
```

Les () délimitent les tuples, les [] délimitent les listes et les accolades {} délimitent les dictionnaires ou les set.

Voyons comment ajouter des clés et valeurs dans notre dictionnaire vide :

```
point = {"x": 1, "y":2}  
point = dict(x=1, y=2)  
point["x"] = 10  
point["z"] = 20  
print(point)  
  
>> {"x" : 10, "y" : 2, "z" : 20}  
  
# Check if the key exist there are 2 methods:  
if "a" in point:  
    print(point["a"])  
  
# Or  
  
print(point.get("a", 0)) # we look at the existence of key a and if it doesn't exist the value by default (0) will be returned  
del point["x"]  
print(point)
```

# Utilisez des dictionnaires{} 5/

Nous indiquons entre crochets la clé à laquelle nous souhaitons accéder. Si la clé n'existe pas, elle est ajoutée au dictionnaire avec la valeur spécifiée après le signe=. Sinon, l'ancienne valeur à l'emplacement indiqué est remplacée par la nouvelle :

```
mon_dictionnaire = {}  
mon_dictionnaire["pseudo"] = "Prolix"  
mon_dictionnaire["mot de passe"] = "*"  
mon_dictionnaire["pseudo"] = "6pri1"  
mon_dictionnaire  
>>> {'mot de passe': '*', 'pseudo': '6pri1'}
```

La valeur 'Prolix' pointée par la clé 'pseudo' a été remplacée, à la ligne 4, par la valeur '6pri1'. Cela devrait vous rappeler la création de variables : si la variable n'existe pas, elle est créée, sinon elle est remplacée par la nouvelle valeur.

Pour accéder à la valeur d'une clé précise, c'est très simple :

```
mon_dictionnaire["mot de passe"]  
>>> '*'
```

Si la clé n'existe pas dans le dictionnaire, une exception de type `KeyError` sera levée.

Généralisons un peu tout cela : nous avons des dictionnaires, qui peuvent contenir d'autres objets. On place ces objets et on y accède grâce à des clés. Un dictionnaire ne peut naturellement pas contenir deux clés identiques (comme on l'a vu, la seconde valeur écrase la première). En revanche, rien n'empêche d'avoir deux valeurs identiques dans le dictionnaire.

# Utilisez des dictionnaires{} 6/

Nous avons utilisé ici, pour nos clés et nos valeurs, des chaînes de caractères. Ce n'est absolument pas obligatoire. Comme avec les listes, vous pouvez utiliser des entiers comme clés :

```
mon_dictionnaire = {}  
mon_dictionnaire[0] = "a"  
mon_dictionnaire[1] = "e"  
mon_dictionnaire[2] = "i"  
mon_dictionnaire[3] = "o"  
mon_dictionnaire[4] = "u"  
mon_dictionnaire[5] = "y"  
mon_dictionnaire  
{0: 'a', 1: 'e', 2: 'i', 3: 'o', 4: 'u', 5: 'y'}
```

On a l'impression de recréer le fonctionnement d'une liste mais ce n'est pas le cas : rappelez-vous qu'un dictionnaire n'a pas de structure ordonnée. Si vous supprimez par exemple l'indice 2, le dictionnaire, contrairement aux listes, ne va pas décaler toutes les clés d'indice supérieur à l'indice supprimé. Il n'a pas été conçu pour.

On peut utiliser quasiment tous les types comme clés et on peut utiliser absolument tous les types comme valeurs.

On peut aussi créer des dictionnaires déjà remplis :

```
placard = {"chemise":3, "pantalon":6, "tee-shirt":7}
```

On précise entre guillemets la clé, le signe deux points « : » et la valeur correspondante. On sépare les différents couples clé : valeur par une virgule. C'est d'ailleurs comme cela que Python vous affiche un dictionnaire quand vous le lui demandez. Certains ont peut-être essayé de créer des dictionnaires déjà remplis avant que je ne montre comment faire. Une petite précision, si vous avez tapé une instruction similaire à :

```
mon_dictionnaire = {'pseudo', 'mot de passe'}
```

# Utilisez des dictionnaires{} Suppression 7/

Supprimer des clés d'un dictionnaire

Comme pour les listes, vous avez deux possibilités mais elles reviennent sensiblement au même :

- le mot-clé **del**;
- la méthode de dictionnaire **pop**.

```
placard = {"chemise":3, "pantalon":6, "tee shirt":7}  
del placard["chemise"]
```

La méthode **pop** supprime également la clé précisée mais elle renvoie la valeur supprimée :

```
placard = {"chemise":3, "pantalon":6, "tee shirt":7}  
placard.pop("chemise")  
>>> 3
```

En plus de supprimer la clé et la valeur associée, la méthode **pop** renvoie cette valeur. Cela peut être utile parfois.

Un peu plus loin

On se sert parfois des dictionnaires pour stocker des fonctions.

Les fonctions sont manipulables comme des variables. Ce sont des objets, un peu particuliers mais des objets tout de même. Donc on peut les prendre pour valeur d'affectation ou les ranger dans des listes ou dictionnaires.

```
print_2 = print # L'objet print_2 pointera sur la fonction print  
print_2("Affichons un message")  
>>> Affichons un message
```

On copie la fonction **print** dans une autre variable **print\_2**. On peut ensuite appeler **print\_2** et la fonction va afficher le texte saisi, tout comme **print** l'aurait fait.

En pratique, on affecte rarement des fonctions de cette manière. C'est peu utile.

# Utilisez des dictionnaires{} modification 8/

Modifiez plusieurs valeurs d'un dictionnaire avec la méthode **update()**

<https://blog.teclado.com/python-updating-dictionaries/>



Updating Python Dictionaries.pdf

```
dico_en_fr={'five': 'cinq', 'four': 'quatre', 'three': 'trois'}  
dico_en_fr.update({'two': 'deux', 'one': 'un'})
```

```
print(dico_en_fr)  
>>> {'five': 'cinque', 'four': 'quatre', 'three': 'trois', "two": "deux", "one": "un"}
```

La methode **get()** is interesting as you get NONE if the key doesn't exist instead of an error when you use []

```
customer = {'name': 'John Smith', 'age': 30, 'is_verified': True}  
print(customer.get('age'))  
>>> 30
```

```
print(customer.get('birthdate'))  
>>> None
```

La methode **get()** can also provide a default value

```
customer = {"name": "John Smith", "age": 30, "is_verified": True}  
print(customer.get("birthdate", "Jan 1 1980"))  
print(customer.get("birthdate"))  
#>>> Jan 1 1980
```

# Utilisez des dictionnaires{} 9/

on met parfois des fonctions dans des dictionnaires :

```
def fete():
    print("C'est la fête.")

def oiseau():
    print("Fais comme l'oiseau...")

fonctions = {}
fonctions["fete"] = fete # on ne met pas les parenthèses
fonctions["oiseau"] = oiseau
fonctions["oiseau"]
<function oiseau at 0x00BA5198>
fonctions["oiseau"]() # on essaye de l'appeler
>>> Fais comme l'oiseau...
```

Prenons dans l'ordre si vous le voulez bien :

- On commence par définir deux fonctions, fete et oiseau.
- On crée un dictionnaire nommé fonctions.
- On met dans ce dictionnaire les fonctions fete et oiseau. La clé pointant vers la fonction est le nom de la fonction, tout bêtement, mais on aurait pu lui donner un nom plus original.
- On essaye d'accéder à la fonction oiseau en tapant fonctions["oiseau"]. Python nous renvoie un truc assez moche,<function oiseau at 0x00BA5198>, mais vous comprenez l'idée : c'est bel et bien notre fonction oiseau. Toutefois, pour l'appeler, il faut des parenthèses, comme pour toute fonction qui se respecte.
- En tapant fonctions["oiseau"](), on accède à la fonction oiseau et on l'appelle dans la foulée.

# Utilisez des dictionnaires{} 10/ parcours



[Python Dictionaries and Loops.pdf](https://blog.teclado.com/python-dictionaries-and-loops/)

Les méthodes de parcours:

## 1 - parcours des clés

```
fruits = {"pommes":21, "melons":3, "poires":31}
for cle in fruits:
...     print(cle)
...
melons
poires
pommes
```

Une méthode de la classe dict permet d'obtenir ce même résultat.

```
fruits = {"pommes":21, "melons":3, "poires":31}
for cle in fruits.keys():
    print(cle)

>>> melons
>>> poires
>>> pommes
```

## 2 - parcours des valeurs

```
fruits = {"pommes":21, "melons":3, "poires":31}
for valeur in fruits.values():
    print(valeur)

>>> 3
>>> 31
>>> 21
```

# Utilisez des dictionnaires{} parcours 11/

## parcours des valeurs simultanément

```
fruits = {"pommes":21, "melons":3, "poires":31}  
for cle, valeur in fruits.items():  
    print("La clé {} contient la valeur {}".format(cle, valeur))
```

```
>>> La clé melons contient la valeur 3.  
>>> La clé poires contient la valeur 31.  
>>> La clé pommes contient la valeur 21.
```

## Récupérer les paramètres nommés dans un dictionnaire

Il existe aussi une façon de capturer les paramètres nommés d'une fonction. Dans ce cas, toutefois, ils sont placés dans un dictionnaire. Si, par exemple, vous appelez la fonction ainsi :fonction(parametre='a'), vous aurez, dans le dictionnaire capturant les paramètres nommés, une clé 'parametre' liée à la valeur 'a'. Voyez plutôt :

```
def fonction_inconnue(**parametres_nommes):  
    """Fonction permettant de voir comment récupérer les paramètres nommés  
    dans un dictionnaire"""  
  
    print("J'ai reçu en paramètres nommés : {}".format(parametres_nommes))  
  
fonction_inconnue() # Aucun paramètre  
>>> J'ai reçu en paramètres nommés : {}  
  
fonction_inconnue(p=4, j=8)  
>>> J'ai reçu en paramètres nommés : {'p': 4, 'j': 8}
```

Pour capturer tous les paramètres nommés non précisés dans un dictionnaire, il faut mettre deux étoiles \*\* avant le nom du paramètre. Si vous passez des paramètres non nommés à cette fonction, Python lèvera une exception.

# Utilisez des dictionnaires{} 12/

Ainsi, pour avoir une fonction qui accepte n'importe quel type de paramètres, nommés ou non, dans n'importe quel ordre, dans n'importe quelle quantité, il faut la déclarer de cette manière :

```
def fonction_inconnue(*en_liste, **en_dictionnaire):
```

Tous les paramètres non nommés se retrouveront dans la variable en\_liste et les paramètres nommés dans la variable en\_dictionnaire.

Transformer un dictionnaire en paramètres nommés d'une fonction

Là encore, on peut faire exactement l'inverse : transformer un dictionnaire en paramètres nommés d'une fonction. Voyons un exemple tout simple :

```
parametres = {"sep": " // ", "end": "-\n"}  
print("Voici", "un", "exemple", "d'appel", **parametres)  
>>> Voici // un // exemple // d'appel -
```

Les paramètres nommés sont transmis à la fonction par un dictionnaire. Pour indiquer à Python que le dictionnaire doit être transmis comme des paramètres nommés, on place deux étoiles avant son nom \*\* dans l'appel de la fonction.

Comme vous pouvez le voir, c'est comme si nous avions écrit :

```
print("Voici", "un", "exemple", "d'appel", sep=" // ", end="-\n")  
>>> Voici // un // exemple // d'appel -
```

# Utilisez des dictionnaires (15) / dictionary Comprehension

## Set and Dictionary Comprehensions

Note that `set` use `{value}` as well as dictionary `{key, value}` with key value pair therefore it is easy to make comprehension with set.

Liste compréhension: [expression for item in items]

Let's adapt this expression for dictionnaries:

Dictionary Compréhension: {expression for item in items}

```
values = {x * 2 for x in range(5)} -> set of even numbers
>> Set
{0, 2, 4, 6, 8}
```

# below we have x as a key and x \* 2 as value

```
values = {x: x * 2 for x in range(5)} -> dictionary of key value pairs
>> Dictionaries
{0: 0, 1: 2, 2: 4, 3: 6, 4: 8}
```

```
users = [
    (0, 'Bob', 'password'),
    (1, 'Rolf', 'bob123'),
    (2, 'Jose', 'longp4assword'),
    (3, 'username', '1234'),
]
username_mapping = {user[1]: user for user in users}
print(username_mapping)
>> {'Bob': (0, 'Bob', 'password'), 'Rolf': (1, 'Rolf', 'bob123'), 'Jose': (2, 'Jose', 'longp4assword'), 'username': (3, 'username', '1234')}
```

# Exemple de dictionary comprehension 14/

```
users = [
    (0, 'Bob', 'password'),
    (1, 'Rolf', 'bob123'),
    (2, 'Jose', 'longp4assword'),
    (3, 'username', '1234'),
]

username_mapping = {user[1]: user for user in users}

username_input = input('Enter your username: ')
password_input = input('Enter your password: ')

_, username, password = username_mapping[username_input]

if password_input == password:
    print('Your details are correct!')
else:
    print('Your details are incorrect.')
```

# How to sort a python dict by value 15/

==get a representation by value

```
xs = {'a': 4, 'b': 3, 'c': 2, 'd': 1}
```

```
sorted(xs.items(), key=lambda x: x[1])
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

or

```
import operator
sorted(xs.items(), key=operator.itemgetter(1))
>>> [('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

Plus de détails a la slide [méthodes de trie](#)

# The get() method on Python dicts and its « default » arg 16/

```
# The get() method on dicts
# and its "default" argument

name_for_userid = {
    382: "Alice",
    590: "Bob",
    951: "Dilbert",
}

def greeting(userid):
    return "Hi %s!" % name_for_userid.get(userid, "Hi there")

greeting(382)
>>> "Hi Alice!"

greeting(333333)
>>> "Hi there!"
```

When "get()" is called it checks if the given key exists in the dict.

If it does exist, the value for that key is returned.

If it does not exist then the value of the default argument is returned instead.

# How to merge two dictionaries 17/

```
# How to merge two dictionaries
# in Python 3.5+
x = {'a': 1, 'b': 2}
y = {'b': 3, 'c': 4}

z = {**x, **y}

z
>> {'c': 4, 'a': 1, 'b': 3}

# In these examples, Python merges dictionary keys
# in the order listed in the expression, overwriting
# duplicates from left to right.
#
# See: https://www.youtube.com/watch?v=Duezw08KaC8
```

# Résume

- Un dictionnaire est un objet conteneur associant des clés à des valeurs.
- Pour créer un dictionnaire, on utilise la syntaxe `dictionnaire = {cle1:valeur1, cle2:valeur2, cleN:valeurN}`.
- On peut ajouter ou remplacer un élément dans un dictionnaire :`dictionnaire[cle] = valeur`.
- On peut supprimer une clé (et sa valeur correspondante) d'un dictionnaire en utilisant, au choix, le mot-clé `del` ou la méthode `pop`.
- On peut parcourir un dictionnaire grâce aux méthodes `keys`(parcourt les clés),`values`(parcourt les valeurs) ou `items`(parcourt les couples clé-valeur).
- On peut capturer les paramètres nommés passés à une fonction en utilisant cette syntaxe :`def fonction_inconnue(**parametres_nommes):` (les paramètres nommés se retrouvent dans le dictionnaire `parametres_nommes`).

# Les opérateurs ternaires

# Les operateurs ternaires

Voici un exemple classique de structure conditionnelle. Néanmoins dans un exemple simple comme celui-ci (if – else) nous sommes obligé d'écrire 5 lignes de codes.

```
age = 20  
  
if age >= 18:  
    majeur = True  
else:  
    majeur = False
```

Grace aux operateurs ternaires on peut grandement simplifier le code, comme suit:

```
age = 20  
  
majeur = True if age >= 18 else False
```

Attention !!!

- Cela ne marche qu'avec une structure conditionnelle de type if – else.
- Il ne faut pas oublier de mettre le else!
- Ne pas faire d'opérateur ternaire qui soit trop long rendant les choses plus compliquées qu'une structure conditionnelle classique

# Apprendre à faire des boucles

# La boucle while

[https://www.teclado.com/30-days-of-python/python-30-day-8-while-loops?\\_s=wj99fjkc10g15ts6m4rh](https://www.teclado.com/30-days-of-python/python-30-day-8-while-loops?_s=wj99fjkc10g15ts6m4rh)



Day 8 While Loops.pdf

```
while condition:  
    # instruction 1  
    # instruction 2  
    # ...  
    # instruction N
```

```
nb = 7 # On garde la variable contenant le nombre dont on veut la table de multiplication  
i = 0 # C'est notre variable compteur que nous allons incrémenter dans la boucle
```

```
while i < 10: # Tant que i est strictement inférieure à 10  
    print(i + 1, "*", nb, "=", (i + 1) * nb)  
    i += 1 # On incrémente i de 1 à chaque tour de boucle
```

# La boucle for

[https://www.teclado.com/30-days-of-python/python-30-day-6-for-loops?\\_s=wj99fjkc10g15ts6m4rh](https://www.teclado.com/30-days-of-python/python-30-day-6-for-loops?_s=wj99fjkc10g15ts6m4rh)



Day 6 For Loops.pdf

for element in sequence:

```
chaine = "Bonjour les ZEROS"  
for lettre in chaine:  
    print(lettre)
```

```
chaine = "Bonjour les ZEROS"  
for lettre in chaine:  
    if lettre in "AEIOUYaeiouy": # lettre est une voyelle  
        print(lettre)  
    else: # lettre est une consonne... ou plus exactement, lettre n'est pas une voyelle  
        print("*")
```

```
friends = ['Rolf', 'Jen', 'Anne']  
for friend in friends:  
    print(friend)
```

```
elements = [0, 1, 2, 3, 4, 5, 6, 7, 8, 0]  
for _ in elements:  
    print('Hello, world!')
```



We call the variable `_` when we don't intend to use the variable for anything. In this example we repeat 10 times the print but we don't use the content of `elements`, whereas in the example above we used the content of `friends` to print it.

# La boucle for and range

```
for index in range(10):  
    print("Hello, world!")
```

As on the previous slide, hello world! will be printed 10 times

```
for index in range(5, 10):  
    print("Hello, world!")
```

We will print 5, 6, 7, 8, 9

```
for index in range(2, 20, 3):  
    print(index)
```

We will print 2, 5, 8, 11, 14, 17

# La boucle for with else

```
cars = ["ok", "ok", "ok", "faulty", "ok", "ok"]

for status in cars:
    if status == "faulty":
        print("Stopping the production line!")
        break
    print(f"This car is {status}.")
    print("Shipping new car to customer!")
else:
    print("All cars built successfully. No faulty cars!")
```

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n // x}")
            break
        else:
            print(f"{n} is a prime number.")
```

Trouver les nombres premier entre 1 et 10

# Les mots-clés break continue

```
while 1: # 1 est toujours vrai -> boucle infinie
    lettre = input("Tapez 'Q' pour quitter : ")
    if lettre == "Q":
        print("Fin de la boucle")
        break
```

```
cars = ["ok", "ok", "ok", "faulty", "ok", "ok"]
for status in cars:
    if status == "faulty":
        print("Stopping the production line!")
        break
    print(f"This car is {status}.")
```

```
i = 1
while i < 20: # Tant que i est inférieure à 20
    if i % 3 == 0:
        i += 4 # On ajoute 4 à i
        print("On incrémente i de 4. i est maintenant égale à", i)
        continue # On retourne au while sans exécuter les autres lignes
    print("La variable i =", i)
    i += 1 # Dans le cas classique on ajoute juste 1 à i
```

# Les mots-clés break continue

```
i = 1
while i < 20: # Tant que i est inférieure à 20
    if i % 3 == 0:
        i += 4 # On ajoute 4 à i
        print("On incrémente i de 4. i est maintenant égale à", i)
        continue # On retourne au while sans exécuter les autres lignes
    print("La variable i =", i)
    i += 1 # Dans le cas classique on ajoute juste 1 à i
```

```
cars = ["ok", "ok", "ok", "faulty", "ok", "ok"]

for status in cars:
    if status == "faulty":
        print("Found faulty car, skipping...")
        continue
    print(f"This car is {status}.")
    print("Shipping new car to customer!")
```

# Les mots-clés break continue

```
# Python's `for` and `while` loops
# support an `else` clause that executes
# only if the loops terminates without
# hitting a `break` statement.

def contains(haystack, needle):
    """
    Throw a ValueError if `needle` not
    in `haystack`.
    """
    for item in haystack:
        if item == needle:
            break
    else:
        # The `else` here is a
        # "completion clause" that runs
        # only if the loop ran to completion
        # without hitting a `break` statement.
        raise ValueError('Needle not found')
```

```
>>> contains([23, 'needle', 0xbadc0ffee], 'needle')
None
```

```
>>> contains([23, 42, 0xbadc0ffee], 'needle')
ValueError: "Needle not found"
```

*# Personally, I'm not a fan of the `else`*

```
# "completion clause" in loops because
# I find it confusing. I'd rather do
# something like this:
def better_contains(haystack, needle):
    for item in haystack:
        if item == needle:
            return
    raise ValueError('Needle not found')

# Note: Typically you'd write something
# like this to do a membership test,
# which is much more Pythonic:
if needle not in haystack:
    raise ValueError('Needle not found')
```

# Assignment expressions in Python

```
user_input = input('Enter q or p: ')
while user_input != 'q':
    if user_input == 'p':
        print("Hello!")

user_input = input('Enter q or p: ')
```

The code above is fairly simple. We have this while loop which keeps asking the user to press **p** or **q**, and if the user pressed **p**, we print this cheerful little "Hello!" message. If the user presses **q**, on the other hand, the loop ends. For any other input, we do nothing.

While this code is okay, we do have this annoying bit of duplication, because we have to start up the loop by defining `user_input`. If we don't, `user_input` is undefined during the first iteration of the loop. Now here is the same code using an assignment expression:

```
while (user_input := input('Enter q or p: ')) != 'q':
    if user_input == 'p':
        print("Hello!")
```

As you can see we now have our cute little walrus next to `user_input` when we define the loop condition, but there's actually quite a lot going on here, so let's break it down.

First of all, what does the `:=` actually do? It's essentially a special type of assignment operator. As usual, we assign some value to a variable name, but the key difference is that the assignment is treated as an expression. Expressions *evaluate to something*, and in the case of assignment expressions, the value they evaluate to is the value after the assignment operator, `:=`.

The fact that the assignment is treated as an expression is why we can use `!= 'q'` directly after it: the assignment expression represents an actual value. This is different from the assignment statements we're used to in Python, which don't represent any value at all.

As you can see in the example above, we have an expression which forms the condition for our while loop which looks like this:

```
(user_input := input('Enter q or p: ')) != 'q'
```

It reads something like, "user\_input is not equal to the string 'q', where user\_input is the return value of the function call `input('Enter p or q: ')`".

In the case of our while loop, this expression, including the assignment, is evaluated for every iteration of the loop. This means that for every iteration of the loop, we're going to ask the user to enter a value, and that value will be assigned to the variable `user_input`, and compared against the string '`q`'.

Since `user_input` is just a normal loop variable, we can still use it inside the loop as well, which is why the if statement isn't complaining.

One important part of the syntax here is that the assignment is wrapped in parentheses. This is a requirement in basically all cases where an assignment expression is likely, but you can read more about it in [this section of the PEP](#).

In addition to trimming off a few lines here and there, assignment expressions can also be used to improve the efficiency of our programs by preventing the repetition of costly operations. I think this is most effective in structures like list comprehensions:

# Assignment expressions in Python

```
results = [result for x in data if (result := costly_function(x)) >= 0]
```

Here we're concerned with the values generated by this `costly_function` when we pass it values from some iterable called `data`. Instead of calculating the results twice - once when performing the comparison, and once when adding the value to the `results` list - we now just assign the result to a variable and reference the variable.

The alternative would be something like this:

```
results = []

for x in data:
    result = costly_function(x)

    if result >= 0:
        results.append(result)
```

It's quite a lot more verbose, but perhaps that verbosity is worth it for the kind of clarity we get with the traditional for loop. It's really up to you to decide whether or not the assignment expressions are readable enough to justify the reduction in code length.

I'd really recommend taking a look at some of the examples in the PEP, in addition to their advice on using the syntax well. It's really not worth putting assignment expressions all over the place, as it's only going to harm the readability of your code. It's an effective but quite niche tool that we all need to be very careful with.

# Résumé

- Une boucle sert à répéter une portion de code en fonction d'un prédictat.
- On peut créer un boucle grâce au mot-clé **while** suivi d'un prédictat.
- On peut parcourir une séquence grâce à la syntaxe **for element in sequence:**

# Useful Operators in python

# Useful Operators in python: range 1/

unpacking  
range  
enumerate

zip  
stacks  
queue / deque  
in  
min max  
random  
randin  
uniform  
randrange  
input  
counter  
Defaultdict  
Ordereddict  
Namedtuple

[https://www.teclado.com/30-days-of-python/python-30-day-9-enumerate-zip?\\_s=wj99fjkc10g15ts6m4rh](https://www.teclado.com/30-days-of-python/python-30-day-9-enumerate-zip?_s=wj99fjkc10g15ts6m4rh)

<https://www.teclado.com/30-days-of-python/python-30-day-26-standard-library>



# Useful Operators in python: range /

```
for num in range(10): # 10 is not included  
    print(num)
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

```
for num in range(0,11,2): # we put 11 to get 10
```

```
0  
2  
4  
6  
8  
10
```

```
List(range(0,11,2)  
[0,2,4,6,8,10])
```

# Useful Operators in python: unpacking /

<https://blog.teclado.com/destructuring-in-python/>



Destructuring in Python.pdf

```
title, director, year = ("Memento", "Christopher Nolan", 2000)
```

Unpacking in a loop:

```
movies = [
(
    "Eternal Sunshine of the Spotless Mind",
    "Michel Gondry",
    2004
),
(
    "Memento",
    "Christopher Nolan",
    2000
),
(
    "Requiem for a Dream",
    "Darren Aronofsky",
    2000
)
]

for title, director, year in movies:
    print(f"{title} ({year}), by {director}")
```

# Useful Operators in python: unpacking /

Let's imagine we create a list numbers = [1, 2, 3], then if we print our list we get: [1, 2, 3] but if we want only the figures ?

```
number = [1, 2, 3]
print(*numbers)
>> 1, 2, 3
```

So the unpacking operator « \* » allow to even more:

```
values = list(range(5))
print(values)
>> [0, 1, 2, 3, 4]

values = [*range(5)]
print(values)
>> [0, 1, 2, 3, 4]
```

So we can also unpack dictionaries but we need 2 \*

```
first = {"x": 1}
second = {"x": 10, "y": 2}
combined = {**first, **second, "z": 1}
print(combined)
>> {'x': 10, 'y': 2, 'z': 1}
```

So note that the value of x is 10 so if you have many times the same key the last value will be used.

# Useful Operators in python: enumerate /

<https://www.teclado.com/30-days-of-python/python-30-day-9-enumerate-zip>



Day 9 Unpacking, Enumeration, and the zip Function.pdf

Basically, enumerate() allows you to loop over a collection of items while keeping track of the current item's index in a counter variable.

Let's take a look at a quick example:

```
names = ['Bob', 'Alice', 'Guido']
for index, value in enumerate(names):
    print(f'{index}: {value}')
```

This produces the following output:

```
0: Bob
1: Alice
2: Guido
```

As you can see, this iterated over the names list and generated an index for each element by increasing a counter variable starting at zero.

Now why is keeping a running index with the enumerate function useful?

I noticed that new Python developers coming from a C or Java background sometimes use the following range(len(...)) antipattern to keep a running index while iterating over a list with a for-loop:

# La fonction enumerate /

```
# HARMFUL: Don't do this
for i in range(len(my_items)):
    print(i, my_items[i])
```

By using the enumerate function skillfully, like I showed you in the “names” example above, you can make this looping construct much more “Pythonic” and idiomatic. You see, there’s usually no need to generate element indexes manually in Python—you simply leave all of this work to the enumerate function.

And as a result your code will be easier to read and less vulnerable to typos.

Another useful feature is the ability to choose the starting index for the enumeration.

The enumerate() function accepts an optional argument which allows you to set the initial value for its counter variable:

```
names = ['Bob', 'Alice', 'Guido']
for index, value in enumerate(names, 1):
    print(f'{index}: {value}')
```

In the above example I changed the function call to enumerate(names, 1) and the extra 1 argument now starts the index at one instead of zero:

```
>>> 1: Bob
>>> 2: Alice
>>> 3: Guido
```

# La fonction enumerate /

And voilà, this is how you switch from zero-based indexing to starting with index 1 (or any other int, for that matter) using Python's `enumerate()` function.

You might be wondering how the `enumerate` function works behind the scenes—so let's talk about that for a bit:

Part of it's magic lies in the fact that `enumerate` is implemented as a Python iterator.

This means that element indexes are generated lazily (one by one, just-in-time), which keeps memory use low and keeps this construct so fast.

Let's play with some more code to demonstrate what I mean:

```
names = ['Bob', 'Alice', 'Guido']
enumerate(names)
>>> <enumerate object at 0x1057f4120>
```

In the above code snippet I set up the same enumeration you've already seen in the previous examples.

But instead of immediately looping over the result of the `enumerate` call I'm just displaying the returned object on the Python console.

As you can see, it's an “enumerate object.”

This is the actual iterator.

And like I said, it generates its output elements lazily and one by one when they're requested.

In order to retrieve those “on demand” elements so we can inspect them, I'm going to call the built-in `list()` function on the iterator:

```
list(enumerate(names))
>>> [(0, 'Bob'), (1, 'Alice'), (2, 'Guido')]
```

For each element in the input list (`names`) the iterator returned by `enumerate()` produces a tuple of the form (index, element).

In your typical `for-in` loop you'll use this to your advantage by leveraging Python's data structure unpacking feature:

```
for index, element in enumerate(iterable):
    # ...
```

And there you have it—`enumerate()` is awesome!

Let's do a quick recap:

1. "`enumerate()`" is a built-in function of Python. You use it to loop over an iterable with an automatic running index generated by a counter variable.
2. The counter starts at 0 by default, but you can set it to any integer.
3. `enumerate` was added to Python starting at version 2.3 with the implementation of PEP 279.
4. Python's `enumerate` function helps you write more Pythonic and idiomatic looping constructs that avoid the use of clunky and error-prone manual indexing.

# La fonction enumerate /

Les deux méthodes que nous venons de voir possèdent toutes deux des inconvénients :

- la méthode utilisant while est plus longue à écrire, moins intuitive et elle est perméable aux boucles infinies, si l'on oublie d'incrémenter la variable servant de compteur
- la méthode par for se contente de parcourir la liste en capturant les éléments dans une variable, sans qu'on puisse savoir où ils sont dans la liste.

```
ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
for i, elt in enumerate(ma_liste):
    print("À l'indice {} se trouve {}".format(i, elt))

>>> À l'indice 0 se trouve a.
>>> À l'indice 1 se trouve b.
>>> À l'indice 2 se trouve c.
>>> À l'indice 3 se trouve d.
>>> À l'indice 4 se trouve e.
>>> À l'indice 5 se trouve f.
>>> À l'indice 6 se trouve g.
>>> À l'indice 7 se trouve h.
```

Quand on parcourt chaque élément de l'objet renvoyé par enumerate, on voit des tuples qui contiennent deux éléments : d'abord l'indice, puis l'objet se trouvant à cet indice dans la liste passée en argument à la fonction enumerate.

# La fonction enumerate /

<https://blog.teclado.com/python-enumerate/>



Python's Enumerate Function.pdf

Quand on utilise enumerate, on capture l'indice et l'élément dans deux variables distinctes. Voyons un autre exemple pour comprendre ce mécanisme :

```
autre_liste = [
    [1, 'a'],
    [4, 'd'],
    [7, 'g'],
    [26, 'z'],
]
# J'ai étalé la liste sur plusieurs lignes

for nb, lettre in autre_liste:
    print("La lettre {} est la {}e de l'alphabet.".format(lettre, nb))

#>>> La lettre a est la 1e de l'alphabet. La lettre d est la 4e de l'alphabet.
#>>> La lettre g est la 7e de l'alphabet. La lettre z est la 26e de l'alphabet.
```

```
friends = ["Rolf", "John", "Anna"]
for counter, friend in enumerate(friends):
    # or for counter, friend in enumerate(friends, start=1):
        print(counter)
        print(friend)
print(list(enumerate(friends)))
print(dict(enumerate(friends)))
```

# La fonction zip()

[zip\(\)](#) and <https://blog.teclado.com/python-zip/>



Using zip in Python.pdf

zip is an extremely powerful and versatile function used to combine two or more iterables into a single iterable.  
For example, let's say that we have two lists like this:

```
pet_owners = ["Paul", "Andrea", "Marta"]
pets = ["Fluffy", "Bubbles", "Captain Catsworth"]
```

zip will allow us to turn this into a new iterable which contains the following:

```
("Paul", "Fluffy"), ("Andrea", "Bubbles"), ("Marta", "Captain Catsworth")
```

In essence, it takes the first item from each iterable, and puts together them in a tuple. Then it takes the second item from each iterable, and so on, until one of the iterables runs out of values. We'll come back to this point.  
To use zip, all we have to do is call the function and pass in the iterables we want to zip together.

```
pet_owners = ["Paul", "Andrea", "Marta"]
pets = ["Fluffy", "Bubbles", "Captain Catsworth"]

pets_and_owners = zip(pet_owners, pets)
```

If we want to zip three or even more iterables together, we can just keep passing more and more items to zip when we call it.

Much like range, zip is lazy, which means it only calculates the next value when we request it. We therefore can't print it directly, but we can convert it to something like a list if we want to see the output:

```
print(list(pets_and_owners))

# [('Paul', 'Fluffy'), ('Andrea', 'Bubbles'), ('Marta', 'Captain Catsworth')]
```

## Using zip in Loops

Another very common way to use zip is to iterate over two or more iterables at once in a for loop.

Let's go back to our pet owners example, but now I want to print some output which describes who owns which pet.

We can use zip and a bit of destructuring to do this in a really clear way, because we get to use nice clear variable names in the loop:

```
pet_owners = ["Paul", "Andrea", "Marta"]
pets = ["Fluffy", "Bubbles", "Captain Catsworth"]

for owner, pet in zip(pet_owners, pets):
    print(f"{owner} owns {pet}.")
```

The common alternative to using zip is the nasty range + len pattern we saw earlier with enumerate. I'd recommend avoiding that at all costs!

# A caveat for when using enumerate and zip

## A caveat for when using enumerate and zip

One thing you should be aware of when it comes to enumerate and zip is that they are consumed when we iterate over them. This generally isn't a problem when we use them directly in loops, but it can sometimes trip up newer developers when they assign a zip or enumerate object to a variable.

Here is an example where we assign the result of calling zip to a variable:

```
movie_titles = [  
    "Forrest Gump",  
    "Howl's Moving Castle",  
    "No Country for Old Men"  
]  
  
movie_directors = [  
    "Robert Zemeckis",  
    "Hayao Miyazaki",  
    "Joel and Ethan Coen"  
]  
  
movies = zip(movie_titles, movie_directors)
```

We can iterate over movies without any problems:

```
for title, director in movies:  
    print(f"{title} by {director}.")
```

However, if we now try to use movies again, we'll find that it's empty. Try running the code below to see this:

```
movie_titles = [  
    "Forrest Gump",  
    "Howl's Moving Castle",  
    "No Country for Old Men"  
]  
  
movie_directors = [  
    "Robert Zemeckis",  
    "Hayao Miyazaki",  
    "Joel and Ethan Coen"  
]  
  
movies = zip(movie_titles, movie_directors)  
  
for title, director in movies:  
    print(f"{title} by {director}.")  
  
movies_list = list(movies)  
  
print(f"There are {len(movies_list)} movies in the  
collection.")  
print(f"These are our movies: {movies_list}.")
```

If you try to iterate over movies after the initial loop, you'll also find that it contains no values.

The reason that this happens is because zip and enumerate produce something called an *iterator*. One key feature of iterators is that they're consumed when we request their values. This is actually a really useful feature, but it's also a common source of bugs if you're not familiar with this behaviour.

# A caveat for when using enumerate and zip

An easy way to bypass this limitation is to just convert the iterator to a non-iterator collection, like a list or tuple.

```
movie_titles = [  
    "Forrest Gump",  
    "Howl's Moving Castle",  
    "No Country for Old Men"  
]  
  
movie_directors = [  
    "Robert Zemeckis",  
    "Hayao Miyazaki",  
    "Joel and Ethan Coen"  
]  
  
movies = list(zip(movie_titles, movie_directors))
```

Now we can access the values in movies as many times as we like.

# Stacks /

LIFO = Last In First Out

```
browsing_session = []
browsing_session.append(1)
browsing_session.append(2)
browsing_session.append(3)
```

```
print(browsing_session)
1, 2, 3
```

```
last = browsing_session.pop()
```

```
print(browsing) # 3 has been removed
1, 2
```

# Queues or deque/double ended queue /

<https://blog.teclado.com/python-deques/>



Python Deques.pdf

FIFO: First In First Out

On large queue you might performance decreasing *as all* the Numbers *in* your *list* must be moved on the left.  
So *in* such situation you might want to use deque

```
from collections import deque
```

```
queue = deque([])
queue.append(2)
queue.append(3)
queue.appendleft(1)
print(queue)
>> deque([1, 2, 3])
queue.popleft()
print(queue)
>> deque([2, 3])
```

# In operator /

Permet de vérifier l'appartenance

```
'x' in [1,2,3]
False

'x' in ['x','y','z']
True

'a' in 'a world'
True

'mykey' in {'mykey':345}
True

d = {'mykey':345}
345 in d.keys()
False
```

# min() et max() /

```
mylist = [10, 20, 30, 40, 100]
min(mylist)
10

max(mylist)
100
```

# random() /

```
from random import shuffle
mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
shuffle(mylist) # it doesn't return anything it modifies the liste of origine. This is an inplace      #function
mylist
[3, 9, 7, 8, 10, 5, 1, 2, 6, 4]
```

# randint() /

```
from random import randint
r = randint(0,100)
print(r)
79
0 et 100 sont compris dans le range.

mynum = randint(0,10)
print(mynum)
3
0 et 10 sont compris dans le range.
```

# uniform() /

```
from random import uniform  
  
r = uniform(0, 1)  
print(r)  
  
# r retourne un réel compris entre 0 et 1 grâce à la fonction uniform
```

# randrange() /

```
from random import randrange  
  
r = randrange(999)  
print(r)  
  
# r retourne un entier compris entre 0 et 998 la valeur de fin est exclusive  
  
Il est également possible de fournir un pas comme suit:  
r = randrange(0, 101, 10)  
print(r)  
  
>> 0, 80, 50, 70, 0, 100, 100
```

# Input() /

```
Input('Enter a number here: ')  
  
Enter a number here : 50  
'50'  
  
result = input('What is your name?')  
What is your name? Julien  
  
result  
'Julien'  
  
result = input('Favourite Number: ')  
Favorite Number: 30  
  
type(result)  
str  
  
# so it is necessary to cast result depending of what you expect  
result = int(input('What is your age?'))  
type(result)  
int
```

# counter() /

```
Input('Enter a number here: ')  
  
Enter a number here : 50  
'50'  
  
result = input('What is your name?')  
What is your name? Julien  
  
result  
'Julien'  
  
result = input('Favourite Number: ')  
Favorite Number: 30  
  
type(result)  
str  
  
# so it is necessary to cast result depending of what you expect  
result = int(input('What is your age?'))  
type(result)  
int
```

# defaultdict()



<https://www.geeksforgeeks.org/defaultdict-in-python/>

Defaultdict in Python - GeeksforGeeks.pdf

```
from collections import defaultdict

coworkers = [('Rolf', 'MIT'), ('Jen', 'Oxford'), ('Rolf',
'Cambridge'), ('Charlie', 'Manchester')]

alma_maters = defaultdict(list) # it put an empty list
so that if you call a key which doesn't exist it will
return an empty dictionary instead of an error.

for coworker, place in coworkers:
    alma_maters[coworker].append(place)

print(alma_maters['Rolf'])
print(alma_maters['Anne'])
>> ['MIT', 'Cambridge']
>> []
```

```
from collections import defaultdict

my_company = 'Teclado'

coworkers = ['Jen', 'Li', 'Charlie', 'Rhys']
other_coworkers = [('Rolf', 'Apple Inc.'), ('Anna', 'Google')]

coworkers_companies = defaultdict(lambda: my_company)

for person, company in other_coworkers:
    coworkers_companies[person] = company

print(coworkers_companies[coworkers[0]])
print(coworkers_companies['Rolf'])

>> Teclado
>> Apple Inc.
```



defaultdict() takes a function as argument.

```
# Solution1:
# Function to return a default
# values for keys that is not
# present
from collections import defaultdict, OrderedDict, namedtuple, deque

def def_value():
    return "Not Present"

# Defining the dict
d = defaultdict(def_value)
d["a"] = 1
d["b"] = 2

print(d["a"])
print(d["b"])
print(d["c"])
```

```
>> 1
>> 2
>> Not Present

# Solution2:
d = defaultdict(lambda: "Not Present")

# Defining the dict
d["a"] = 1
d["b"] = 2

print(d["a"])
print(d["b"])
print(d["c"])

>> 1
>> 2
>> Not Present
```

# ordereddict() /

```
from collections import OrderedDict

o = OrderedDict()
o['Rolf'] = 6
o['Jose'] = 12
o['Jen'] = 3

print(o) # The result is in order it was inserted
o.move_to_end('Rolf')
print(o)
o.popitem()
print(o)
```

Less and less useful as in python 3.9 by default dictionary are already ordered by order it was inserted.  
It can still be useful for move\_to\_end and popitem functionalities

# namedtuple()

<https://blog.teclado.com/pythons-namedtuples/>



Python's namedtuples.pdf

```
# Why Python is Great: Namedtuples  
# Using namedtuple is way shorter than  
# defining a class manually:  
from collections import namedtuple  
Car = namedtuple('Car', 'color mileage')  
  
# Our new "Car" class works as expected:  
my_car = Car('red', 3812.4)  
my_car.color  
=>> 'red'  
my_car.mileage  
=>> 3812.4  
  
# We get a nice string repr for free:  
my_car  
=>> Car(color='red', mileage=3812.4)  
  
# Like tuples, namedtuples are immutable:  
my_car.color = 'blue'  
=>> AttributeError: "can't set attribute"
```

```
from collections import namedtuple  
  
account = ('checking', 1850.90)  
  
Account = namedtuple('Account', ['name', 'balance'])  
account = Account('checking', 1850.90)  
  
print(account.name)  
print(account.balance)  
print(account)  
  
>>> checking  
>>> 1850.9  
>>> Account(name='checking', balance=1850.9)
```

# Exercices /

- Create a `task1()` function.
- In `task1()` function, create a `defaultdict` object, and its default value would be set to the string '`unknown`'.
- Add an entry with key name '`Alan`' and its value being '`Manchester`'.
- Return the `defaultdict` object you created.

```
from collections import defaultdict, OrderedDict, namedtuple, deque
```

```
def task1() -> defaultdict:  
    d = defaultdict(lambda: "Unknown")  
    d['Alan'] = 'Manchester'  
    return d
```

```
print(task1()['Alan'])  
print(task1()['Bob'])
```

```
>>> Manchester  
>>> Unknown
```

# Exercices /

- Create a `task2()` function that takes in one argument `arg_od`, which is an `OrderedDict` object.
- Remove the first and last entry in `arg_od`.
- Move the entry with key name `Bob` to the end of `arg_od`.
- Move the entry with key name `Dan` to the start of `arg_od`.
- You may assume that `arg_od` would always contain the keys `Bob` and `Dan`, and they won't be the first or last entry initially.

An example `arg_od` would look like this:

```
OrderedDict([
    ('Alan', 'Manchester'),
    ('Bob', 'London'),
    ('Chris', 'Lisbon'),
    ('Dan', 'Paris'),
    ('Eden', 'Liverpool'),
    ('Franck', 'Newcastle')
])
```

And after calling `task2(arg_od)`, it would look like this:

```
OrderedDict([
    ('Dan', 'Paris'),
    ('Chris', 'Lisbon'),
    ('Eden', 'Liverpool'),
    ('Bob', 'London')
])
```

```
from collections import defaultdict, OrderedDict, namedtuple, deque

o = OrderedDict([
    ('Alan', 'Manchester'),
    ('Bob', 'London'),
    ('Chris', 'Lisbon'),
    ('Dan', 'Paris'),
    ('Eden', 'Liverpool'),
    ('Franck', 'Newcastle')
])

def task2(arg_od: OrderedDict):
    o.popitem(0)
    o.popitem()
    o.move_to_end('Bob')
    o.move_to_end('Dan', 0)
    print(o)

task2(o)
```

# Exercices /

In this task, you will need to:

- Create a `task3()` function that takes in two arguments, `name` and `club`.
- In `task3()` function, you will need to create a `namedtuple` with type `Player`, and it will have tow fields, `name` and `club`.
- The `Player namedtuple` instance should have the `name` and `club` set by the given arguments of `taks3()` function.
- At last, return the `Player` namedtuple instance you created

```
from collections import defaultdict, OrderedDict, namedtuple, deque
```

```
def task3(name: str, club: str) -> namedtuple:  
    Player = namedtuple('Player', ['name', 'club'])  
    player = Player(name, club)  
    print(player.name)  
    print(player.club)  
    return player
```

```
result = task3('Jose', 'Paris SG')  
print(result.name)  
print(result.club)
```

```
>> Jose  
>> ParisSG
```

# Exercices /

In this task, you will need to:

Create a `task4()` function that takes in one argument, `arg_deque` which is a `deque`.

- Manipulate the `deque` in any order you preferred to achieve the following effect:
  - Remove last element in `deque`
  - Move the first (left most) element to the end (right most)
  - Add an element '`Zack`', a string, to the start(left)

```
def task4(arg_deque: deque):
    """
    - Manipulate the `arg_deque` in any order you preferred to achieve the following effect:
        -- remove last element in `deque`
        -- move the fist (left most) element to the end (right most)
        -- add an element `Zack`, a string, to the start (left)
    """
    # your code starts here:

    arg_deque.pop()
    arg_deque.append(arg_deque.pop())
    arg_deque.appendleft('Zack', 'Milan')

task4(arg_deque)
```

# Les chemins

# Les chemins 1/2

Comment créer un chemin pour Windows ?

Le r est placé devant la chaîne pour ne pas interpréter les \ sinon les \t et \n seraient interprétés comme des tabulations ou retour chariot. Il veut dire **raw string**.

Option 1:

```
chemin = r'C:\Users\thibault\nouveau'
```

Option 2: utiliser un / à la place d'un \

```
chemin = 'C:/Users/thibault/nouveau'
```

Option 3: utiliser un \\

```
chemin = 'C:\\Users\\thibault\\nouveau'
```

Liste des caractères interprétés par Python lorsqu'ils sont précédés d'un \

- \a caractère d'appel (BEL)
- \b caractère de retour arrière
- \f saut de la page
- \n retour à la ligne
- \r retour chariot
- \t tabulation horizontal
- \v tabulation verticale

# Les chemins 2/2

Transformer un chemin relatif en absolu

```
import os  
chemin = "/Users/MOTTIER Lucie/Documents/GitHub/Udemy/Tests/../PyQtTests"  
print(os.path.normpath(chemin))  
>> /Users/MOTTIER Lucie/Documents/GitHub/Udemy/PyQtTests
```

[Plus sur les paths, fichiers et répertoires](#)

# Les fonctions

# Les fonctions

```
def nom_de_la_fonction(parametre1, parametre2, parametreN):
    """
        Docstring explains function
    """
    # bloc d'instruction
```

- **def**, mot-clé qui est l'abréviation de « define » (définir, en anglais) et qui constitue le prélude à toute construction de fonction.
- Le nom de la fonction, qui se nomme exactement comme une variable (nous verrons par la suite que ce n'est pas par hasard). N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
- La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses ouvrante et fermante (là encore, les espaces sont optionnels mais améliorent la lisibilité).
- Les deux points, encore et toujours, qui clôturent la ligne.

**Les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.**

# Creation de fonctions

```
def table_par_7():
    """DOCSTRING: Information about the function
    INPUT: nb, max # by default nb = 7
    OUTPUT: 7 or other value defined by the function caller, table from 0 to value provided by the user in the function argument
    """
    nb = 7
    i = 0 # Notre compteur ! L'auriez-vous oublié ?
    while i < 10: # Tant que i est strictement inférieure à 10,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1 # On incrémente i de 1 à chaque tour de boucle.

def table(nb: int, max: int = 10):
    """Fonction affichant la table de multiplication par nb
    de 1*nb à max*nb
    (max >= 0)
    """
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

# Creation de fonctions

```
# Imagine you've got some code that calculates the fuel efficiency of a car:
```

```
# But this is not a very reusable function since it only calculates the mpg of a single car.
```

```
# What if we made it calculate the mpg of "any" arbitrary car?
```

```
car = {"make": "Ford", "model": "Fiesta", "mileage": 23000, "fuel_consumed": 460}
```

```
def calculate_mpg(car_to_calculate): # This can be renamed to `car`  
    mpg = car_to_calculate["mileage"] / car_to_calculate["fuel_consumed"]  
    name = f"{car_to_calculate['make']} {car_to_calculate['model']}"  
    print(f"{name} does {mpg} miles per gallon.")
```

```
# This means that given a list of cars with the correct data format, we can run the function for all  
of them!
```

```
cars = [  
    {"make": "Ford", "model": "Fiesta", "mileage": 23000, "fuel_consumed": 460},  
    {"make": "Ford", "model": "Focus", "mileage": 17000, "fuel_consumed": 350},  
    {"make": "Mazda", "model": "MX-5", "mileage": 49000, "fuel_consumed": 900},  
    {"make": "Mini", "model": "Cooper", "mileage": 31000, "fuel_consumed": 235},  
]  
  
for car in cars:  
    calculate_mpg(car)
```

# Valeurs par défaut des paramètres 1/

```
def func(a: int = 1, b: int = 2, c: int = 3, d: int = 4, e: int = 5):
    print("a =", a, "b =", b, "c =", c, "d =", d, "e =", e)
```

Instruction	Résultat
func()	a=1 b=2 c=3 d=4 e=5
func(4)	a=4 b=2 c=3 d=4 e=5
func(b=8, d=5)	A=1 b=8 c=3 d=5 e=5
func(b=35, c=48, a=4, e=9)	A=4 b=35 c=48 d=4 e=9

When you call a function and you use a named argument all subsequent arguments must have a name i.e:

```
def add(x: int, y: int = 3):
    total = x + y
    return total

print(add(x=5, y=2)) # works
print(add(5, y=2)) # works print(add(x=5, 2) won't work and returns an error
```

The same apply to the definition of a function, after a named argument all subsequent arguments must have a name i.e:

```
def add(x: int, y: int=3): # works
    total = x + y
    return total
print(add(5, y=2))

def add(x: int=5, y):
    # won't work and returns an error
    total = x + y
    return total

print(add(5, y=2))
```

# Valeurs par défaut des paramètres 2/

```
accounts = {  
    'checking': 1958.00,  
    'savings': 3695.50  
}  
  
def add_balance(amount: float, name: str = 'checking') -> float:  
    """Function to update the balance of an account and return the new balance"""  
    accounts[name] += amount  
    return accounts[name]  
  
add_balance(500.00)  
  
print(accounts['checking'])
```

You can use an existing named arguments such as **sep**

```
print(1, 2, 3, 4, 5, sep="-")  
>>> 1 - 2 - 3 - 4 - 5
```

# Mutable default argument (bad idea) 3/

```
def create_account(name: str, holder: str, account_holders: list = []):
    print(id(account_holders))
    account_holders.append(holder)
    # The default parameter for the create_account function gets evaluated when the function is defined, not
    # when the function is called
    return {
        'name': name,
        'main_account_holder': holder,
        'account_holders': account_holders
    }

a1 = create_account('checking', 'Rolf')
a2 = create_account('savings', 'Jen')

print(a2)
```

The default parameter for the create\_account function gets evaluated when the function is defined, not when the function is called. We use the same list each time we call the function (list is mutable) we can see it while we print id(account\_holders) the id remains the same.

There are two ways to solve this problem:

1. Not have a list as default argument

```
def create_account(name: str, holder: str, account_holders):
    print(id(account_holders))
    account_holders.append(holder)
    return {
        'name': name,
        'main_account_holder': holder,
        'account_holders': account_holders
    }

a1 = create_account('checking', 'Rolf', [])
a2 = create_account('savings', 'Jen', [])

print(a2)
```

# Mutable default argument (bad idea) 4/

```
def create_account(name: str, holder: str, account_holders: list = []):
    account_holders.append(holder)

    return {
        'name': name,
        'main_account_holder': holder,
        'account_holders': account_holders
    }

a1 = create_account('checking', 'Rolf')
a2 = create_account('savings', 'Jen')
print(a2)

>>> {'name': 'savings', 'main_account_holder': 'Jen', 'account_holders': ['Rolf', 'Jen']}
```



The problem, here is that we get two people as account\_holders, so how this happened? We never provided Rolf as argument with a2. The default parameter for the create\_account function gets evaluated when the function is defined not when the function is called. So we always used the same list the one created when the function get evaluated instead of 2 different list.

There are 2 ways to solve this problem (which is a common pb btw), first one is to not have a default argument:

```
def create_account(name: str, holder: str, account_holders: list):
    account_holders.append(holder)

    return {
        'name': name,
        'main_account_holder': holder,
        'account_holders': account_holders
    }

a1 = create_account('checking', 'Rolf', [])
a2 = create_account('savings', 'Jen', [])
print(a2)

>>> {'name': 'savings', 'main_account_holder': 'Jen', 'account_holders': ['Jen']}
```

The second option is to not make account\_holder a list but make it equal to None

```
def create_account(name: str, holder: str, account_holders = None):
    if not account_holders:
        account_holders = []

    account_holders.append(holder)

    return {
        'name': name,
        'main_account_holder': holder,
        'account_holders': account_holders
    }

a1 = create_account('checking', 'Rolf')
a2 = create_account('savings', 'Jen')
print(a2)

>>> {'name': 'savings', 'main_account_holder': 'Jen', 'account_holders': ['Jen']}
```

# Positional and keyword arguments

```
def greet_user(first_name, last_name):
    print(f'Hi {first_name} {last_name}!')
    print('Welcome aboard')

print('Positional arguments')
greet_user('John', 'Smith') # positional argument have to be provided in order

print('Keyword arguments')
greet_user(first_name='John', last_name='Smith') # keywords arguments
```

When it is not obvious what the parameters are, ex calc\_cost(50, 5, 0.1) it is better to use **keyword arguments** like this: calc\_cost(total=50, shipping=5, discount=0.1) rather than **positional arguments**

If you wish to use positional and keyword argument together you have to use positional argument **first** and **then** keyword argument

```
greet_user("John", last_name="Smith")
```

For the most part use positional arguments, if you are dealing with functions that take multiple numerical values and it's not quite clear whose values represent, use keywords arguments to improve the readability of your code, and finally if your passing both positional and keyword arguments, use the keyword arguments after the positional arguments.

# In Python 3 you can use a bare "\*" asterisk in function parameter lists to force the caller to use keyword arguments for certain parameters:

```
def f(a, b, *, c='x', d='y', e='z'):
    return 'Hello'

# To pass the value for c, d, and e you
# will need to explicitly pass it as
# "key=value" named arguments:
f(1, 2, 'p', 'q', 'v')
TypeError:
"f() takes 2 positional arguments but 5 were given"

f(1, 2, c='p', d='q',e='v')
'Hello'
```

# Signature d'une fonction

```
def greet_user(first_name, last_name):
    print(f'Hi {first_name} {last_name}!')
    print('Welcome aboard')

print('Positional arguments')
greet_user('John', 'Smith') # positional argument have to be provided in order

print('Keyword arguments')
greet_user(first_name='John', last_name='Smith') # keywords arguments
```

En Python comme vous avez pu le voir, on ne précise pas les types des paramètres. Dans ce langage, la signature d'une fonction est tout simplement son nom. Cela signifie que vous ne pouvez définir deux fonctions du même nom (si vous le faites, l'ancienne définition est écrasée par la nouvelle).

```
def exemple():
    print("Un exemple d'une fonction sans paramètre")
```

```
exemple()
>> Un exemple d'une fonction sans paramètre
```

```
def exemple(): # On redéfinit la fonction exemple
    print("Un autre exemple de fonction sans paramètre")
```

```
exemple()
>> Un autre exemple de fonction sans paramètre
```

A la ligne 1 on définit la fonction exemple. On l'appelle une première fois à la ligne 4. On redéfinit à la ligne 6 la Fonction exemple. L'ancienne définition est écrasée et l'ancienne fonction ne pourra plus être appelée.

# L'instruction return

Ce que nous avons fait était intéressant, mais nous n'avons pas encore fait le tour des possibilités de la fonction. Et d'ailleurs, même à la fin de ce chapitre, il nous restera quelques petites fonctionnalités à voir. Si vous vous souvenez bien, il existe des fonctions comme print qui ne renvoient rien (attention, « renvoyer » et « afficher » sont deux choses différentes) et des fonctions telles que input ou type qui renvoient une valeur. Vous pouvez capturer cette valeur en plaçant une variable devant (exemple variable2 = type(variable1)). En effet, les fonctions travaillent en général sur des données et renvoient le résultat obtenu, suite à un calcul par exemple. Prenons un exemple simple : une fonction chargée de mettre au carré une valeur passée en argument. Je vous signale au passage que Python en est parfaitement capable sans avoir à coder une nouvelle fonction, mais c'est pour l'exemple.

```
def carre(valeur):  
    return valeur * valeur
```

Sachez que l'on peut renvoyer plusieurs valeurs que l'on sépare par des virgules, et que l'on peut les capturer dans des variables également séparées par des virgules

```
def rectangle(high, width):  
    return high, width  
  
a, b = rectangle(4,5)
```

```
def dog_check(mystring):  
    return 'dog' in mystring.lower()  
  
print(dog_check('my dog run away'))  
>>> True
```

# L'instruction return



PDF

<https://www.teclado.com/30-days-of-python/python-30-day-13-return-statements>

Day 13 Scope and Returning Values from Functions.pdf

Example 1:

```
def calculate_mpg(car):
    mpg = car["mileage"] / car["fuel_consumed"]
    return mpg # Ends the function, gives back the value
```

```
def car_name(car):
    return f"{car['make']} {car['model']}
```

```
def print_car_info(car):
    name = car_name(car)
    mpg = calculate_mpg(car)
```

```
print(f"{name} does {mpg} miles per gallon.")
# Returns None by default, as all functions do
```

```
cars = [
    {"make": "Ford", "model": "Fiesta", "mileage": 23000, "fuel_consumed": 460},
    {"make": "Ford", "model": "Focus", "mileage": 17000, "fuel_consumed": 350},
    {"make": "Mazda", "model": "MX-5", "mileage": 49000, "fuel_consumed": 900},
    {"make": "Mini", "model": "Cooper", "mileage": 31000, "fuel_consumed": 235},
]
```

```
for car in cars:
    print_car_info(car)
    # try print(print_car_info(car)), you'll see None
```

```
# -- Multiple returns --
```

Example 2:

```
def divide(x, y):
    if y == 0:
        return "You tried to divide by zero!"
    else:
        return x / y
```

```
print(divide(10, 2)) # 5
print(divide(6, 0)) # You tried to divide by zero!
```

# First-class functions 1/4

<https://www.udemy.com/course/the-complete-python-course/learn/lecture/18471560#overview>

We can assign functions to variables, and we can pass them in as arguments to other functions so first let's see about assigning functions to variables and using them as value and list something like that and why it can be useful.

```
def greet():
    print("hello")

hello = greet

hello()
>> hello
```

Function is a first-class citizen; you can assign it to variables, and you can put it inside list, dictionaries or anything else.

```
def average(seq):
    return sum(seq) / len(seq)

# let's make it shorter with lambda function
avg = lambda seq: sum(seq) / len(seq)
total = lambda seq: sum(seq)
top = lambda seq: max(seq)

operations = {
    "average": avg,
    "total": total,
    "top": top
}

students = [
    {"name": "Rolf", "grades": (67, 90, 95, 100)},
    {"name": "Bob", "grades": (56, 78, 80, 90)},
    {"name": "Jen", "grades": (98, 90, 95, 99)},
    {"name": "Anne", "grades": (100, 100, 95, 100)}
]

for student in students:
    name = student["name"]
    grades = student["grades"]

    print(f"Student: {name}")
    operation = input("Enter 'average', 'total', or 'top': ")

    operation_function = operations[operation]
    print(operation_function(grades))
```

# First-class functions 2/4

We can make this code even shorter by in lining the lambda functions into this dictionary

```
def average(seq):
    return sum(seq) / len(seq)

# let's make it even shorter:
operations = {
    "average": lambda seq: sum(seq) / len(seq),
    "total": sum,
    "top": max
}

students = [
    {"name": "Rolf", "grades": (67, 90, 95, 100)},
    {"name": "Bob", "grades": (56, 78, 80, 90)},
    {"name": "Jen", "grades": (98, 90, 95, 99)},
    {"name": "Anne", "grades": (100, 100, 95, 100)},
]

for student in students:
    name = student["name"]
    grades = student["grades"]

    print(f"Student: {name}")
    operation = input("Enter 'average', 'total', or 'top': ")

    operation_function = operations[operation]
    print(operation_function(grades))
```

# First-class functions 3/

```
def search(sequence, expected, finder):
    for elem in sequence:
        if finder(elem) == expected:
            return elem
    raise RuntimeError(f"Could not find an element with {expected}.")  
  
friends = [
    {"name": "Rolf Smith", "age": 24},
    {"name": "Adam Wool", "age": 34},
    {"name": "Anne Pun", "age": 27},
]  
  
def get_friend_name(friend):
    return friend["name"]  
  
print(search(friends, "Rolf Smith", get_friend_name))  
  
>> {'name': 'Rolf Smith', 'age': 24}
```

# First-class functions 4/

First class functions just means that functions are just variables, so you can pass them in arguments to functions and use them in the same way you would use any other variable.

```
def divide(dividend, divisor):
    if divisor == 0
        raise ZeroDivisionError("Divisor cannot be 0.")

    return dividend / divisor

def calculate(*values, operator):
    return operator(*values)

result = calculate(20, 4, operator=divide)
print(result)

>>> 5.0
```

```
# Because Python has first-class functions
# they can
# be used to emulate switch/case
# statements

def dispatch_if(operator, x, y):
    if operator == 'add':
        return x + y
    elif operator == 'sub':
        return x - y
    elif operator == 'mul':
        return x * y
    elif operator == 'div':
        return x / y
    else:
        return None
```

```
def dispatch_dict(operator, x, y):
    return {
```

```
'add': lambda: x + y,
'sub': lambda: x - y,
'mul': lambda: x * y,
'div': lambda: x / y,
}.get(operator, lambda: None)()
```

```
dispatch_if('mul', 2, 8)
>>> 16
```

```
dispatch_dict('mul', 2, 8)
>>> 16
```

```
dispatch_if('unknown', 2, 8)
>>> None
```

```
dispatch_dict('unknown', 2, 8)
>>> None
```

```
def search(sequence, expected, finder):
    for elem in sequence:
        if finder(elem) == expected:
            return elem
    raise RuntimeError(f"Could not find an element with {expected}.")
```

```
friends = [
    {"name": "Rolf Smith", "age": 24},
    {"name": "Adam Wool", "age": 34},
    {"name": "Anne Pun", "age": 27},
]
```

```
def get_friend_name(friend):
    return friend["name"]

print(search(friends, "Rolf Smith", get_friend_name))

>>> {'name': 'Rolf Smith', 'age': 24}
```

# \*args and \*\*kwargs in python

<https://www.teclado.com/30-days-of-python/python-30-day-17-args-kwargs>



Day 17 Flexible Functions with args and kwargs.pdf

Nombre indefini de parameter pour une fonction:

```
def myfunc(*args):  
    return sum(args)* 005
```

```
myfunc(40,60)
```

```
>> 5
```

```
myfunc(40,60,100)
```

```
>> 10
```

```
myfunc(40,60,100,1,34)
```

```
>> 11,75
```

```
def myfunc2(*args)  
print(args)  
>> (40, 60, 100, 1, 34) # this is a tuple !!!
```

```
# Python 3.5+ allows passing multiple sets  
# of keyword arguments ("kwargs") to a  
# function within a single call, using  
# the "***" syntax:
```

```
def process_data(a, b, c, d):  
    print(a, b, c, d)
```

```
x = {"a": 1, "b": 2}  
y = {"c": 3, "d": 4}
```

```
process_data(**x, **y)  
process_data(**x, c=23, d=42)  
>> 1 2 3 4  
>> 1 2 23 42
```

Nombre indefini de key word argument pour une fonction:

```
def myfunc(**kwargs):  
    if 'fruit' is in kwargs:  
        print('My fruit of choice is {}'.format(kwargs['fruit']))  
    else:  
        print('I did not find any fruit here')  
  
myfunc(fruit='apple', veggie='lettuce')  
  
>> My fruit of choice is apple
```

```
# Why Python Is Great:  
# Function argument unpacking
```

```
def myfunc(x, y, z):  
    print(x, y, z)  
  
tuple_vec = (1, 0, 1)  
dict_vec = {'x': 1, 'y': 0, 'z': 1}  
  
myfunc(*tuple_vec)  
>>> 1, 0, 1  
  
myfunc(**dict_vec)  
>>> 1, 0, 1
```

# Les fonctions lambda 1/3

<https://www.teclado.com/30-days-of-python/python-30-day-16-lambda-expressions>



Day 16 First Class Functions and Lambda Expressions.pdf

Nous venons de voir comment créer une fonction grâce au mot-clé **def**. Python nous propose un autre moyen de créer des fonctions, des fonctions extrêmement courtes car limitées à une seule instruction.

Pourquoi une autre façon de créer des fonctions ? La première suffit, non ?

Disons que ce n'est pas tout à fait la même chose, comme vous allez le voir. Les fonctions lambda sont en général utilisées dans un certain contexte, pour lequel définir une fonction à l'aide de `def` serait plus long et moins pratique.

## Syntaxe

Avant tout, voyons la syntaxe d'une définition de fonction lambda. Nous allons utiliser le mot-clé `lambda` comme ceci :`lambda arg1, arg2,... : instruction de retour.`

Un exemple semblera plus clair. On veut créer une fonction qui prend un paramètre et renvoie ce paramètre au carré.

```
lambda x: x * x  
<function <lambda> at 0x00BA1B70>
```

exemple : si vous voulez créer une fonction **lambda** prenant deux paramètres et renvoyant la somme de ces deux paramètres, la syntaxe sera la suivante :

```
lambda x, y: x + y
```

# Les fonctions lambda 2/3

```
##Exemple 1:  
def square(num):  
    return num ** 2  
  
print(square(3))  
>> 9  
  
# la même chose en lambda expression  
square = lambda num: num ** 2  
print(square(5))  
>> 25  
Ou  
(lambda num: num ** 2)(5)  
>> 25  
  
# lambda sont utilisés quand on a besoin de l'expression qu'une fois, dans ce cas écrire une fonction prend beaucoup de place pour rien. On utilise lambda expression  
mynums = [1, 2, 3, 4, 5, 6]  
print(list(map(lambda num: num ** 2, mynums)))  
>> [1, 4, 9, 16, 25, 36]  
  
## Exemple 2:  
def chec_even(num):  
    return num % 2 == 0  
# transformons cette fonction en lambda expression  
lambda num: num % 2 == 0, mynums  
# utilisons filter() avec cette nouvelle lambda expression  
list(filter(lambda num: num % 2 == 0, mynums))  
[2, 4, 6]  
  
# Exemple 3:  
names = ['Andy', 'Eve', 'Sally']  
list(map(lambda x: x[0], names))  
['A', 'E', 'S']
```

# Les fonctions lambda 3/3

```
# Exemple 4:  
# Si je veux inverser les lettres  
  
list(map(lambda x:x[::-1],names))
```

```
# Exemple 5:  
# The lambda keyword in Python provides a  
# shortcut for declaring small and  
# anonymous functions:  
  
add = lambda x, y: x + y  
add(5, 3)  
>>> 8  
  
# You could declare the same add()  
# function with the def keyword:  
  
def add(x, y):  
    return x + y  
add(5, 3)  
>>> 8  
  
# So what's the big fuss about?  
# Lambdas are *function expressions*:  
(lambda x, y: x + y)(5, 3)  
>>> 8  
  
# • Lambda functions are single-expression  
# functions that are not necessarily bound  
# to a name (they can be anonymous).  
  
# • Lambda functions can't use regular  
# Python statements and always include an  
# implicit `return` statement.
```

# La fonction map() 1/3

```
friends = ['Rolf', 'Jose', 'Randy', 'Anna', 'Mary']
start_with_r = filter(lambda friend: friend.startswith('R'), friends)
another_starts_with_r = (f for f in friends if f.startswith('R'))

friends_lower = map(lambda x: x.lower(), friends) # when you code with others who don't know well list and generator comprehension
friends_lower = [friend.lower() for friend in friends] # List comprehension
friends_lower = (friend.lower() for friend in friends) # Generator comprehension prefered unless you need them all to be in a list

print(next(friends_lower))
```

# La fonction map() 2/3

```
def square(num):
    return num**2

my_nums = [1,2,3,4,5]

# plutôt que d'utiliser une loop for pour appliquer la série de
# chiffre dans my_nums à square on peut utiliser map

for item in map(square, my_nums)
    print(item)

# pour récupérer la liste
print(list(map(square, my_nums)))

####

# voyons un exemple plus compliqué
def splicer(mystring):
    if len(mystring) % 2 == 0:
        return 'EVEN'
    else:
        return mystring[0]

names = ['Andy', 'Eve', 'Sally']
print(list(map(splicer, names))) # note que la fonction splicer ne prend pas de () dans le map !!!
```

# La fonction map() 3/3

```
items = [  
    ("Product1", 10),  
    ("Product2", 9),  
    ("Product3", 12)  
]
```

Disons que nous sommes simplement intéressé par les prix, il va falloir transformer cette liste en une liste de nombre comme ceci:

```
prices = list (map(lambda item: item[1], items))  
Print(prices)
```

# La fonction filter()

```
def check_even(num):  
    return num%2 == 0  
  
mynums = [1,2,3,4,5,6]  
  
list(filter(check_even, mynums))  
  
for n in filter(check_even, mynums):  
    print(n)
```

2  
4  
6

```
def check_even(num):  
    return num%2 == 0  
  
mynums = [1,2,3,4,5,6]  
  
list(filter(check_even, mynums))  
  
for n in filter(check_even, mynums):  
    print(n)
```

2  
4  
6

```
items = [  
    ("Product1", 10),  
    ("Product2", 9),  
    ("Product3", 12)  
]  
  
filtered = list(filter(lambda item: item[1] >= 10, items))  
print(filtered)
```

```
def start_with_r(friend):  
    return friend.startswith('R')  
  
friends = ['Rolf', 'Jose', 'Randy', 'Anna', 'Mary']  
start_with_r = filter(start_with_r, friends) # arg 1: function that returns True/False  
  
# you can get the list  
print(list(start_with_r))  
# if you try to print again like this  
print(list(start_with_r))  
# you'll get nothing, the list object here doesn't raise a stop iteration but it sees that the stop iteration  
# is raised and it gives an empty list, because the generator has already been used and it reaches the end  
# so it can't give anything else.  
  
# # filter returns a generator so you can also get the values like this:  
# print(next(start_with_r))  
# print(next(start_with_r))  
# print(next(start_with_r))  
  
# Other solution consist in using a lambda function:  
friends = ['Rolf', 'Jose', 'Randy', 'Anna', 'Mary']  
start_with_r = filter(lambda friend: friend.startswith('R'), friends) # lambda function  
another_starts_with_r = (f for f in friends if f.startswith('R')) # Generator comprehension  
# Those 2 lines above are the same! but the generator comprehension is faster/perform better. The reason for that  
# is because you have to create the lambda function in the filter and you don't have to create the lambda function  
# in the generator. If you already have the function define in the program the filter function can perform better.
```

# La fonction any() and all()

```
friends = [
    {
        'name': 'Rolf',
        'location': 'Washington, D.C.'
    },
    {
        'name': 'Anna',
        'location': 'San Francisco'
    },
    {
        'name': 'Charlie',
        'location': 'San Francisco'
    },
    {
        'name': 'Jose',
        'location': 'San Francisco'
    },
]

your_location = input('Where are you right now? ')
friends_nearby = [friend for friend in friends if friend['location'] == your_location]
# True if there's at least one; or False if empty

if any(friends_nearby):
    print('You are not alone!')

print(all([1, 2, 3, 4, 5]))      -> return True
print(all([0, 1, 2, 3, 4, 5]))  -> return False because 0 is interpreted as False value
```

Rappel:

Interpreted as False:

0, 0.0

None

[], (), {}

False

# La fonction any()

```
any([False, False, True, False]) # cette fonction retourne True car au moins un élément est à True
```

```
all([False, False, True, False]) # cette fonction retourne False pour qu'elle retourne True tous les éléments doivent être à True.
```

Exemple concret d'utilisation de ces fonctions:

```
all([f.endswith(“.jpg”) for f in files]) # on check si tous les fichiers ont l’extension .jpeg
```

```
Notes= [12, 14, 20, 10, 8]
```

```
any([x > 18 for x in notes]) # permet de vérifier si au moins 1 personne a eu plus de 18 (retourne True)
```

# La methodé import

# La méthode import 1/2

Lorsque vous ouvrez l'interpréteur Python, les fonctionnalités du module math ne sont pas incluses. Il s'agit en effet d'un module, il vous appartient de l'importer si vous vous dites « tiens, mon programme risque d'avoir besoin de fonctions mathématiques ». Nous allons voir une première syntaxe d'importation.

```
import math
```

La syntaxe est facile à retenir : le mot-clé **import**, qui signifie « importer » en anglais, suivi du nom du module, ici `math`.

Après l'exécution de cette instruction, rien ne se passe... en apparence. En réalité, Python vient d'importer le module `math`.

Toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles. Pour appeler une fonction du module, il faut taper le nom du module suivi d'un point « `.` » puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

```
print(math.sqrt(16))  
>>> 4
```

# La méthode import 2/2

## Une autre méthode d'importation :from ... import ...

Il existe une autre méthode d'importation qui ne fonctionne pas tout à fait de la même façon. En fonction du résultat attendu, j'utilise indifféremment l'une ou l'autre de ces méthodes. Reprenons notre exemple du module math. Admettons que nous ayons uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

```
from math import fabs  
fabs(-5)  
>>> 5  
fabs(2)  
>>> 2
```

Pour ceux qui n'ont pas encore étudié les valeurs absolues, il s'agit tout simplement de l'opposé de la variable si elle est négative, et de la variable elle-même si elle est positive. Une valeur absolue est ainsi toujours positive.

Vous aurez remarqué qu'on ne met plus le préfixe math. devant le nom de la fonction. En effet, nous l'avons importée avec la méthode from: celle-ci charge la fonction depuis le module indiqué et la place dans l'interpréteur au même plan que les fonctions existantes, comme print par exemple. Si vous avez compris les explications sur les espaces de noms, vous voyez que print et fabs sont dans le même espace de noms (principal).

Vous pouvez appeler toutes les variables et fonctions d'un module en tapant « \* » à la place du nom de la fonction à importer.

```
from math import *  
sqrt(4)  
>> 2  
fabs(5)  
>> 5
```

À la ligne 1 de notre programme, l'interpréteur a parcouru toutes les fonctions et variables du module math et les a importées directement dans l'espace de noms principal sans les emprisonner dans l'espace de noms math.

# Unpacking

# Unpacking function arguments

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

nums = [3, 5]
print(add(*nums)) # here you destructure nums and you pass one value for each parameter
>> 8

nums = {"x": 15, "y": 25}
print(add(**nums))
>> 40
```

```
def multiply(*args):
    total = 1
    for arg in args:
        total = total * arg

    return total

def apply(*args, operator):
    if operator == "*":
        return multiply(
            *args) # here the * is to unpack into the 4 values provided in arguments, and pass them as individual value to function
    multiply
    elif operator == "+":
        return sum(args)
    else:
        return "No valid operator provided to apply()."

print(apply(1, 3, 6, 7, operator="+"))
```

```
# Why Python Is Great:
# Function argument unpacking
```

```
def myfunc(x, y, z):
    print(x, y, z)

tuple_vec = (1, 0, 1)
dict_vec = {'x': 1, 'y': 0, 'z': 1}

myfunc(*tuple_vec)
>>> 1, 0, 1

myfunc(**dict_vec)
>>> 1, 0, 1
```

# Argument unpacking in Python

```
accounts = {  
    'checking': 1958.00,  
    'savings': 3965.50  
}
```

```
def add_balance(amount: float, name: str) -> float:  
    """Function to update the balance of an account and return the new balance.""""  
    accounts[name] += amount  
    return accounts[name]
```

```
transactions = [  
    (-180.67, 'checking'),  
    (-220.00, 'checking'),  
    (220.00, 'savings'),  
    (-15.70, 'checking'),  
    (-23.90, 'checking'),  
    (-13.00, 'checking'),  
    (1579.50, 'checking'),  
    (-600.50, 'checking'),  
    (600.50, 'savings'),  
]
```

```
for t in transactions:  
    add_balance(*t)
```

Other possibility:  
*for t in transactions:*  
 add\_balance(amount=t[0], name=t[1])

# Argument unpacking in Python

```
accounts = {
    'checking': 1958.00,
    'savings': 3965.50
}

def add_balance(amount: float, name: str) -> float:
    """Function to update the balance of an account and return the new balance."""
    accounts[name] += amount
    return accounts[name]

transactions = [
    (-180.67, 'checking'),
    (-220.00, 'checking'),
    (220.00, 'savings'),
    (-15.70, 'checking'),
    (-23.90, 'checking'),
    (-13.00, 'checking'),
    (1579.50, 'checking'),
    (-600.50, 'checking'),
    (600.50, 'savings'),
]
for t in transactions:
    add_balance(amount=t[0], name=t[1])

class User:
    def __init__(self, username, password):
        self.username = username
        self.password = password

# imagine these users are coming from a database...
users = [
    {'username': 'rolf', 'password': '123'},
    {'username': 'tecladoisawesome', 'password': 'youaretoo'}
]

# 1st option
user_objects = [User(username=data['username'], password=data['password']) for data in users]

# 2nd option
user_objects = [User(**data) for data in users] # this unpack the dictionary
```

# Unpacking keyword arguments

1st way to proceed:

```
def named(**kwargs):  
    print(kwargs)
```

```
named(name="Bob", age=25)  
>> {'name': 'Bob', 'age': 25}
```

2nd way to proceed:

```
def named(name, age):  
    print(name, age)
```

```
details = {"name": "Bob", "age": 25}
```

named(\*\*details) *with* this syntax name *and* age are treated *as* keys *for* the arguments.

```
>> Bob 25
```

3rd way to proceed:

```
def named(**kwargs):  
    print(kwargs)
```

```
details = {"name": "Bob", "age": 25}
```

```
named(**details)
```

```
>> Bob 25
```

4th way to proceed:

```
def named(**kwargs):  
    print(kwargs)
```

```
def print_nicely(**kwargs):  
    named(**kwargs)  
    for arg, value in kwargs.items():  
        print(f"{arg}: {value}")
```

```
>> {"name": "Bob", "age": 25}  
>> name: Bob  
>> age: 25
```

```
print(apply(1, 3, 6, 7, operator="+"))
```

# Higher-order functions in Python

We pass greet function as a parameter to the function before\_and\_after:

```
def greet():
    print("Hello")

def before_and_after(func):
    print("Before...")
    func()
    print("After...")

before_and_after(greet)
```

```
movies = [
    {"name": "The Matrix", "director": "Wachowski"},
    {"name": "A Beautiful Day in the Neighborhood", "director": "Heller"},
    {"name": "The Irishman", "director": "Scorsese"},
    {"name": "Klaus", "director": "Pablos"},
    {"name": "1917", "director": "Mendes"},
]
```

```
def find_movie(expected, finder):
    found = []
    for movie in movies:
        if finder(movie) == expected:
            found.append(movie)

    return found
```

```
find_by = input("What property are we searching by? ")
looking_for = input("What are you looking for? ")
movies = find_movie(looking_for, lambda movie: movie[find_by])
print(movies or 'No movies found. ')
```

*# Functions are first-class citizens in Python:*

*# They can be passed as arguments to other functions,  
# returned as values from other functions, and  
# assigned to variables and stored in data structures.*

```
def myfunc(a, b):
    return a + b

funcs = [myfunc]
funcs[0]
>>> <function myfunc at 0x107012230>
funcs[0](2, 3)
>>> 5
```

# Résumé

## A completer

- Une fonction est une portion de code contenant des instructions, que l'on va pouvoir réutiliser facilement.
- Découper son programme en fonctions permet une meilleure organisation.
- Les fonctions peuvent recevoir des informations en entrée et renvoyer une information grâce au mot-clé return.
- Les fonctions se définissent de la façon suivante :`def nom_fonction(parametre1, parametre2, parametreN):`
- Quand une fonction est appelée qu'une seule fois on peut utiliser les lambda expression.
- On peut également utiliser les fonctions map() et filter() pour garder le code plus concis

# Python documentation

# Methods and Function



Deep Learning

- Built-in objects in Python have a variety of methods you can use!
- Let's explore in a bit more detail how to find methods and how to get information about them.

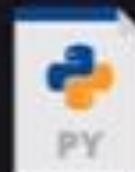
# Python documentation

- <https://docs.python.org/3/library/index.html>

# Les packages

Module  
Packages  
Librairie  
Framework

# Module



# Package



# Librairie



# Framework



# Importer des packages

Si vous voulez utiliser, dans votre programme, la bibliothèque fictive que nous venons de voir, vous avez plusieurs moyens qui tournent tous autour des mots clés **from** et **import**:

```
import nom_bibliotheque
```

Cette ligne importe le package contenant la bibliothèque. Pour accéder aux sous-packages, vous utiliserez un point « . » afin de modéliser le chemin menant au module ou à la fonction que vous voulez utiliser :

```
nom_bibliotheque.evenements # Pointe vers le sous-package evenements  
nom_bibliotheque.evenements.clavier # Pointe vers le module clavier
```

Si vous ne voulez importer qu'un seul module (ou qu'une seule fonction) d'un package, vous utiliserez une syntaxe similaire, assez intuitive :

```
from nom_bibliotheque.objets import bouton
```

# Créer ses propres packages

Si vous voulez créer vos propres packages, commencez par créer, dans le même dossier que votre programme Python, un répertoire portant le nom du package.

Dans ce répertoire, vous pouvez soit :

- mettre vos modules, vos fichiers à l'extension.py
- créer des sous-packages de la même façon, en créant un répertoire dans votre package.

Ne mettez pas d'espaces dans vos noms de packages et évitez aussi les caractères spéciaux. Quand vous les utilisez dans vos programmes, ces noms sont traités comme des noms de variables et ils doivent donc obéir aux mêmes règles de nommage.

## Le fichier d'initialisation

En Python, vous trouverez souvent le fichier d'initialisation de package `__init__.py` dans un répertoire destiné à devenir un package. Ce fichier est optionnel depuis la version 3.3 de Python. Vous n'êtes pas obligé de le créer mais vous pouvez y mettre du code d'initialisation pour votre package.

# PEP8 l'appel des modules/packages

1. Le premier bloc est réservé aux modules de la librairie standard
2. Le second bloc est réservé aux modules qui sont ni dans la librairie standard ni un module créé par le développeur
3. Le troisième bloc est réservé aux modules créés par le développeur

```
import json  
import logging # librairie standard  
import os  
  
import requests # module n'appartenant pas à la librairie standard et pas créé par le développeur  
  
import constants import DATA_DIR # module créé par le développeur
```

# Résumé

- On peut écrire les programmes Python dans des fichiers portant l'extension .py.
- On peut créer des fichiers contenant des modules pour séparer le code.
- On peut créer des répertoires contenant des packages pour hiérarchiser un programme.

# Built in errors in Python

# Built in errors in Python

- IndexError
- KeyError
- NameError
- AttributeError
- NotImplementedError
- RuntimeError
- SyntaxError
- IndentationError
- TabError
- TypeError
- ValueError
- ImportError
- DeprecationWarning

# Built in errors in Python

## IndexError

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> friends = ['Rolf', 'Anne']
>>> friends[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

# Built in errors in Python

## KeyError

```
def show_movie_details(movie):
    print(f"Name: {movie['name']}")
    print(f"Director: {movie['director']}")
    print(f"Release year: {movie['release']}")
```

It should be 'year'

```
Traceback (most recent call last):
  File "/Users/jslvtr/Desktop/milestone_1/app.py", line 88, in <module>
    menu()
  File "/Users/jslvtr/Desktop/milestone_1/app.py", line 37, in menu
    show_movies(movies)
  File "/Users/jslvtr/Desktop/milestone_1/app.py", line 60, in show_movies
    show_movie_details(movie)
  File "/Users/jslvtr/Desktop/milestone_1/app.py", line 66, in show_movie_details
    print(f"Release year: {movie['release']}")
KeyError: 'release'
```

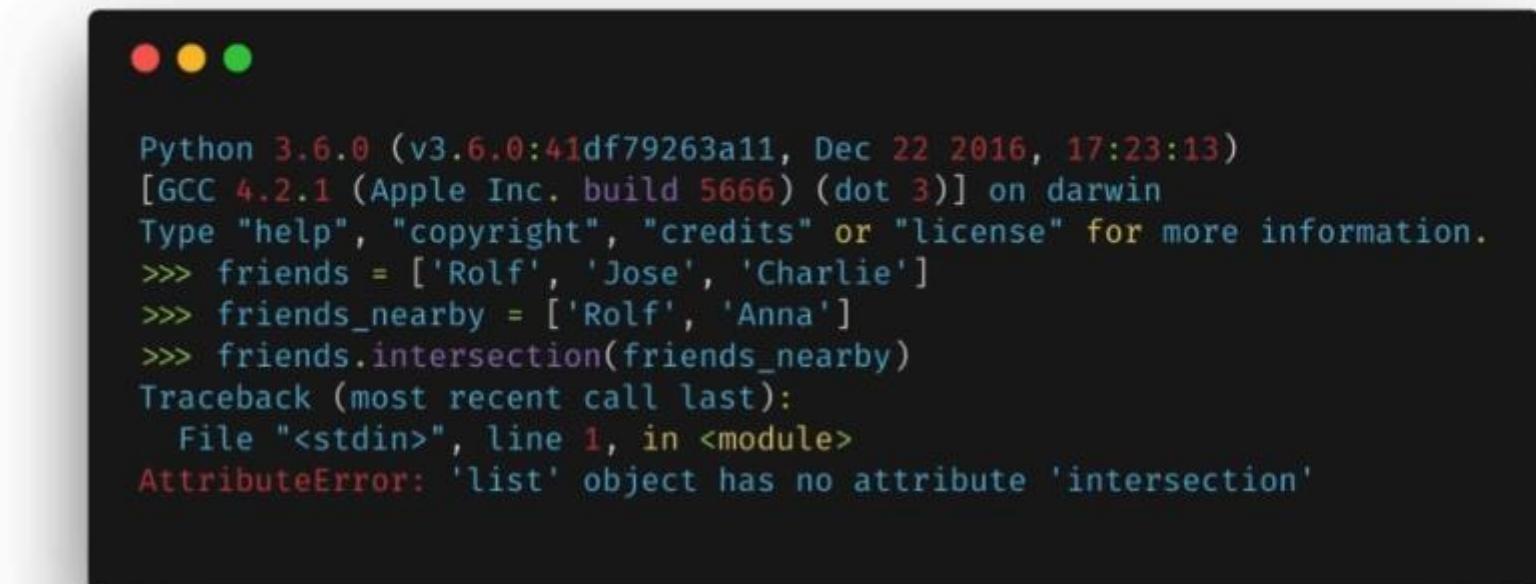
# Built in errors in Python

## NameError

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print(hello)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hello' is not defined
```

# Built in errors in Python

## AttributeError



A screenshot of a macOS terminal window. The window has the standard OS X title bar with red, yellow, and green buttons. The main area of the terminal shows the following Python session:

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> friends = ['Rolf', 'Jose', 'Charlie']
>>> friends_nearby = ['Rolf', 'Anna']
>>> friends.intersection(friends_nearby)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'intersection'
```

# Built in errors in Python

## NotImplementedError

```
● ● ●

class User:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def login(self):
        raise NotImplementedError('This feature has not been implemented yet.')
```

# Built in errors in Python

**RuntimError**

?

# Built in errors in Python

## SyntaxError

```
● ● ●  
class User  
    def __init__(self, username, password):  
        self.username = username  
        self.password = password
```

# Built in errors in Python

## IndentationError

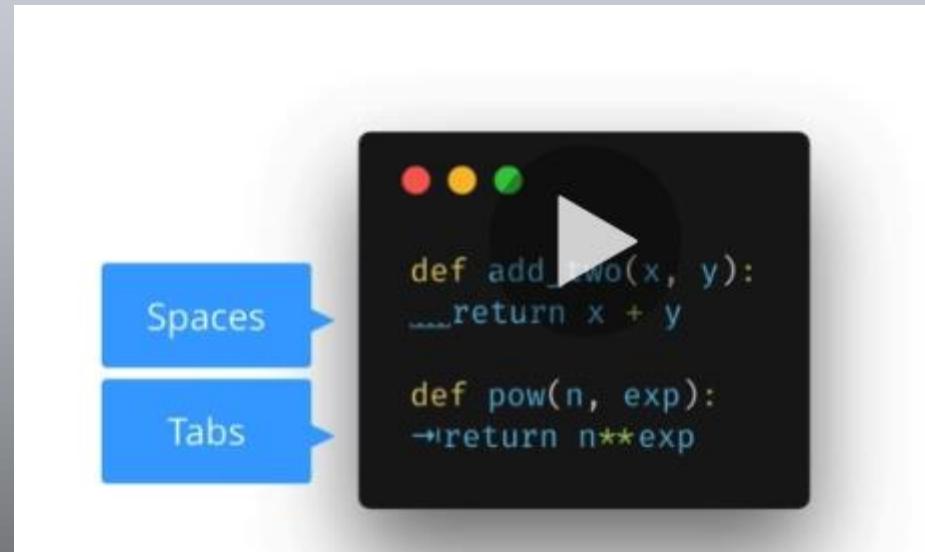
```
def add_two(x, y):  
    return x + y
```

```
def add_two(x, y):  
    pass  
  
    return x + y
```

This would also be an  
error...

NameError 

# Built in errors in Python



# Built in errors in Python

## TypeError

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 5 + 5
10
>>> 'hi' + 'ha'
'hiha'
>>> 5 + 'hi'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# Built in errors in Python

## ValueError

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> int('20.5')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '20.5'
```

# Built in errors in Python

## ImportError

```
● ● ●  
# app.py  
  
import blog  
  
def menu():  
    pass
```



```
● ● ●  
# blog.py  
  
from app import menu  
  
def do_something():  
    pass
```

# Built in errors in Python

## DeprecationWarning

```
from database import Database

class User:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def register(self):
        Database.write(self.username, self.password)
        raise DeprecationWarning('User#register still works, but is deprecated.')

    @classmethod
    def register_user(cls, username, password):
        Database.write(username, password)
        return cls(username, password)
```

# Raising errors in Python

# Raising errors in Python

....

Let's say you have the following code:

....

```
class Garage:  
    def __init__(self):  
        self.cars = []  
  
    def __len__(self):  
        return len(self.cars)  
  
    def add_car(self, car):  
        print('This method is a work in progress.')  
  
....
```

We're working on a class and we've not yet got around to implementing the `add\_car` method. Instead of printing something out, we can raise a `NotImplementedError`.

....

```
class Garage:  
    def __init__(self):  
        self.cars = []  
  
    def __len__(self):  
        return len(self.cars)  
  
    def add_car(self, car):  
        raise NotImplementedError("We can't add cars to the garage yet.")  
  
Garage().add_car('Fiesta') # raises error, comment this line out to run the rest of the file.  
  
....
```

That way we can't call the method and assume it works—it will now fail and crash our program. We'll know that we're doing something that won't work (because it's not implemented yet).

That's how you `raise` an error: use the keyword and create a new error object from the class you want. All built-in

errors (what we looked at in the last video) are available everywhere for you to use.

Let's say we're implementing the method and we want to only allow cars of type `Car`:

....

```
class Car:  
    def __init__(self, make, model):  
        self.make = make  
        self.model = model  
  
    def __repr__(self):  
        return f'<Car {self.make} {self.model}>'  
  
....
```

```
class Garage:  
    def __init__(self):  
        self.cars = []  
  
    def __len__(self):  
        return len(self.cars)
```

```
def add_car(self, car):  
    if not isinstance(car, Car):  
        raise TypeError(f'Tried to add a `{car.__class__.__name__}` to the garage, but you can only add `Car` objects.')  
    self.cars.append(car)
```

```
ford_garage = Garage()  
fiesta = Car('Ford', 'Fiesta')
```

```
ford_garage.add_car(fiesta) # All good  
ford_garage.add_car('Fiesta') # raises error
```

# Creating your own errors in Python

# Creating your own errors in python

```
"""
Sometimes it can be useful to create and raise errors with names we define, as opposed to only using the built-in
errors.
```

```
If we want to create a custom error, we can do so very easily by subclassing the `Exception` class:
```

```
"""
```

```
class MyCustomError(Exception):
    pass
```

```
"""
The `pass` keyword just means "nothing here". It is required because Python expects there to be an indented block
after a colon, so we must at least have _something_ so Python can see the indentation.
```

```
This `MyCustomError` class just inherits everything from `Exception`, which means it behaves just like any other
error.
```

```
"""
```

```
raise MyCustomError('A message describing the error')
```

```
"""


```

```
You can of course also create custom errors that have more than just the base `Exception` functionality. For
example if you wanted to include an error code in your errors, you could do this:
```

```
"""
```

```
class MyErrorWithCode(Exception):
    def __init__(self, message, code):
        super().__init__(message)
        self.code = code
```

```
## Docstrings
```

```
"""


```

```
Docstrings in Python are just strings that are commonly used to describe what a class or function does or when it
should be used.
```

A docstring has this format:

```
"""
```

```
"""
Your docstring goes here.
"""

"""


```

It so happens that in Python the triple-quotation mark is a \*multi-line string\*. You can use it instead of a normal
string anywhere, if you want multiple lines.

But back to docstrings! We can add a docstring to our exception to explain when it should be used:

```
"""
```

```
class MyErrorWithCode(Exception):
    """
    Exception raised when a specific error code is needed.
    """
    def __init__(self, message, code):
        super().__init__(message)
        self.code = code
```

```
"""


```

Notice how here the multi-line string is in one line; and that's OK. We could put it into multiple lines if we want to.

```
"""


```

# Dealing with Python errors

# Dealing with python errors

```
"""
One of Python's core tenets is: "ask for forgiveness, not for
permission".
```

Now, I know how well this works with friends and family (hint: not so well), but it works fantastically in Python. Remember this piece of code?

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def __repr__(self):
        return f'<Car {self.make} {self.model}>'
```

```
class Garage:
    def __init__(self):
        self.cars = []
```

```
def __len__(self):
    return len(self.cars)
```

```
def add_car(self, car):
    self.cars.append(car)
```

We would use these classes in this way:

```
ford_garage = Garage()
fiesta = Car('Ford', 'Fiesta')
```

```
ford_garage.add_car(fiesta)
```

If we wanted to make sure that we're only adding 'Car' objects to the 'Garage', we could do this:

```
car = Car('Ford', 'Focus')
if isinstance(car, Car):
```

```
    ford_garage.add_car(car)
else:
    print("Your car was not a Car!")
```

"""
This is a typical structure of calling a function (in this case, the 'add\_car()' method):

```
if can_call_function():
    call_function()
else:
    say_error_happened()
```

What we do there is ask for permission (the 'can\_call\_function()' statement).

Python suggests that, in many cases, our code can be made more readable by asking for forgiveness instead. Doing this (not real Python code):

```
try to call_function()
if failed:
    say_error_happened
```

Circling back to raising exceptions, we could modify the 'add\_car()' method to do this:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
```

```
    def __repr__(self):
        return f'<Car {self.make} {self.model}>'
```

```
class Garage:
    def __init__(self):
        self.cars = []

    def __len__(self):
        return len(self.cars)
```

```
def add_car(self, car):
    if not isinstance(car, Car):
        raise TypeError(f'Tried to add a `{car.__class__.__name__}` to the garage, but you can only add `Car` objects.')
    self.cars.append(car)
```

"""
And then we could call it like so:

```
car = Car('Ford', 'Focus')
try:
    ford_garage.add_car(car)
except TypeError:
    print("Your car was not a Car!")
```

"""
There are two benefits:

1. Our code reacts more nicely: we try to do something that we expect to be able to do, and if we cannot then we say an error happened;
2. Our check for whether it is something we can do is now encapsulated inside the 'add\_car()' method; we don't need to have an if statement every time we want to add a car.

The syntax is the 'try-catch' syntax.

We try to do whatever is inside the 'try' block, and then if an error happens we jump to the 'except' block. We only do so for errors that match the one in the block (in this case, 'TypeError' would be caught, other errors would not be caught).

We can catch multiple errors (even though our method won't raise them, just showing you the syntax here):

```
car = Car('Ford', 'Focus')
try:
    ford_garage.add_car(car)
except TypeError:
    print("Your car was not a Car!")
except ValueError:
    print("Something was wrong with your Car...")
```

"""
Over the next few sections we'll be making use of this, which is why it's really useful to know how to use 'try' and 'catch'.

'try' and 'catch' also have a final counterpart: 'finally'.

We can use 'finally' to run a block of code no matter what happens: whether or not an exception is raised. For example:

```
car = Car('Ford', 'Focus')
try:
    ford_garage.add_car(car)
except TypeError:
    print("Your car was not a Car!")
finally:
    print(f'Your garage has {len(ford_garage)} cars.')
```

# The on success block and re-raising exceptions

# The on success block and re-raising exceptions

```
class User:  
    def __init__(self, name, engagement):  
        self.name = name  
        self.engagement_metrics = engagement  
        self.score = 0  
  
    def __repr__():  
        return f'<User {self.name}>'  
  
  
def email_engaged_user(user):  
    try:  
        user.score = perform_calculation(user.engagement_metrics)  
    except KeyError:  
        print('Incorrect values provided to our calculation function.')  
        raise # this allow the program to throw error  
    else:  
        if user.score > 500:  
            send_engagement_notification(user)  
  
  
def perform_calculation(metrics):  
    return metrics['clicks'] * 5 + metrics['hits'] * 2  
  
  
def send_engagement_notification(user):  
    print(f'Notification sent to {user}.')  
  
my_user = User('Rolf', {'clicks': 61, 'hits': 100})  
email_engaged_user(my_user)
```

# Exercice

In the last lecture we learned how to handle errors in Python with the help of the `try`, `except`, `else` and `finally` keywords. Now let's put them into practice!

We challenge you to build a simple Artificial Intelligence (AI). Our AI is disguised as a function

Called `interact()` which works as follow:

- It first asks the user for an integer input.
- Then it tells the user whether that integer is even or odd.
- Then it asks the user whether she wants to play this game again.
- If the user inputs `y`, then it prints out `Goodbye`. And quits.

(this AI looks a lot like a user menu for a text application!)

Here's our AI version 1.0:

```
def interact():
    while True: # Keep looping until user reach break statement
        user_input = int(input('Please input an integer:')) # turn the user input into an integer

        print('{} is {}'.format(user_input, 'even' if user_input % 2 == 0 else 'odd')) # point out the message '{user_input} is {even/odd}.'

        user_input = input('Do you want to play again? (y/N): ')

        if user_input != 'y': # quit if the user didn't input 'y'
            print('Goodbye.')
            break # break the while loop to quit
```

However, our AI 1.0 is not very smart, since users can input things other than an integer, such as a string. Our AI will break with a `ValueError` if that happens. The `ValueError` would be raised in the first line: `user_input = int(input('Please input an integer:'))`, when the user inputs something other than an integer.

Now that we've located the failure point, we ask you to build our AI 2.0 which can recover from the invalid user input. If the user enters an invalid input at first, it will proceed to ask the user if she wants to play again.

Here are some sample in inputs and expectations:

Valid inputs:

```
AI: Please input an integer:
User: 17
AI : 17 is odd.
AI: Do you want to play again? (y/N):
User: y
AI: Please input an integer:
User: 11
AI: 11 is odd.
AI: Do you want to play again? (y/N)
User: n
AI: Goodbye.
```

Hint: insert your `try`, `except`, `else` and `finally` key words in the appropriate places and change/add some code if you find necessary.

# Solution

```
#Welcome back to another coding exercise solution!
#This is my solution to the coding exercise. Again, it may be slightly different from yours!
def interact():
    while True:
        try:
            user_input = int(input('Please input an integer:')) # try to turn user input into an integer
        except ValueError:
            print('Please input integers only.') # print a message if user didn't input an integer
        else:
            print('{} is {}'.format(user_input, 'even' if user_input % 2 == 0 else 'odd')) # print even/odd if the user input an integer
        finally: # regardless of the previous input being valid or not
            user_input = input('Do you want to play again? (y/N):') # ask if the user wants to play again
            if user_input != 'y': # quit if the user didn't input 'y'
                print('Goodbye.')
                break # break the while loop to quit
```

# Gérez les exceptions

Look Before You Leap – LBYL

It's Easier to Ask for Forgiveness than  
Permission - EAFTP

# Gérez les exceptions 1/

```
# Look Before You Leap LBYL:  
if 'cle' in dict:  
    print(dict['cle'])
```

```
#It's Easier to Ask for Forgiveness than Permission EAFP:  
try:  
    print(dict['cle'])  
except:  
    pass
```

# Gérez les exceptions 2/

```
# Look Before You Leap LBYL:
```

```
liste = [2, 7, "texte", 4]
for i in liste:
    if not str(i).isdigit():
        liste.remove(i)
totale = sum(liste)
```

```
# It's Easier to Ask for Forgiveness than Permission EAFP:
```

```
liste = [2, 7, "texte", 4]
try:
    total = sum(liste)
except:
    total = 0
```

# Gérez les exceptions 2/

```
try:  
    # Bloc à essayer  
except:  
    # Bloc qui sera exécuté en cas d'erreur
```

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

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

# Forme plus complète 3/

On peut capturer l'exception et afficher son message grâce au mot-clé as.

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

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

Je vous conseille de *toujours* préciser un type d'exceptions après **except**(sans nécessairement capturer l'exception dans une variable, bien entendu). D'abord, vous ne devez pas utiliser **try** comme une méthode miracle pour tester n'importe quel bout de code. Il est important que vous gardiez le maximum de contrôle sur votre code. Cela signifie que, si une erreur se produit, vous devez être capable de l'anticiper. En pratique, vous n'irez pas jusqu'à tester si une variable quelconque existe bel et bien, il faut faire un minimum confiance à son code. Mais si vous êtes en face d'une division et que le dénominateur pourrait avoir une valeur de 0, placez la division dans un bloc **try** et précisez, après le **except**, le type de l'exception qui risque de se produire (ZeroDivisionError dans cet exemple).

Si vous adoptez la forme minimale (à savoir **except** sans préciser un type d'exception qui pourrait se produire sur le bloc **try**), toutes les exceptions seront traitées de la même façon. Et même si exception = erreur la plupart du temps, ce n'est pas toujours le cas. Par exemple, Python lève une exception quand vous voulez fermer votre programme avec le raccourci CTRL + C. Ici vous ne voyez peut-être pas le problème mais si votre bloc **try** est dans une boucle, vous ne pourrez pas arrêter votre programme avec CTRL + C, puisque l'exception sera traitée par votre **except**. Je vous conseille donc de toujours préciser un type d'exception possible après votre **except**. Vous pouvez bien entendu faire des tests dans l'interpréteur de commandes Python pour reproduire l'exception que vous voulez traiter et ainsi connaître son type.

# Forme plus complète

## Les mots-clés else et finally 4/

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

Le mot-clé **else**

Vous avez déjà vu ce mot-clé et j'espère que vous vous en rappelez. Dans un bloc **try**, **else** va permettre d'exécuter une action si aucune erreur ne survient dans le bloc.

```
try:  
    # Test d'instruction(s)  
except type_de_l_exception:  
    # Traitement en cas d'erreur  
else:  
    # Instruction(s) exécutée(s) si il n'y a pas d'erreur
```

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

go to google and look for python3 built-in exception

# Forme plus complète exemple 4/

```
a = 5
b = 3

try:
    resultat = a / b
except ZeroDivisionError:
    print("Division par zero impossible.")
except TypeError:
    print("La variable b n'est pas du bon type.")
except NameError as e: # permet d'afficher le type d'erreur
    print("Erreur:", e)
else:
    print(resultat)

try:
    resultat = a / b
except ZeroDivisionError:
    print("Division par zero impossible.")
except TypeError:
    print("La variable b n'est pas du bon type.")
except NameError as e: # permet d'afficher le type d'erreur
    print("Erreur:", e)
finally:
    print("Fin du bloc.")
```

# Forme plus complète exemple 5/

```
try:  
    with open("DataStructures.py") as file:  
        print("File opened.")  
  
        age = int(input("Age: "))  
        xfactor = 10 / age  
    except (ValueError, ZeroDivisionError):  
        print("You didn't enter a valid age.")  
    else:  
        print("No exceptions were thrown.")  
    finally:  
        file.close()  
    print("Execution continues")  
  
    print("\n The with statement")  
  
    print("\n Raising Exceptions")  
  
  
def calculate_xfactor(age):  
    if age <= 0:  
        raise ValueError # go to google and look for python3 built-in exception  
    return 10 / age
```

# Un petit bonus : le mot-clé pass 6/

Il peut arriver, dans certains cas, que l'on souhaite tester un bloc d'instructions... mais ne rien faire en cas d'erreur. Toutefois, un bloc try ne peut être seul.

```
try:  
    1/0
```

File "<stdin>", line 3

  ^

SyntaxError: invalid syntax

Il existe un mot-clé que l'on peut utiliser dans ce cas. Son nom est **pass** et sa syntaxe est très simple d'utilisation :

```
try:  
    # Test d'instruction(s)  
except type_de_l_exception: # Rien ne doit se passer en cas d'erreur  
    pass
```

# Custom error classes

```
class TooManyPagesReadError(ValueError):
    pass

class Book:
    def __init__(self, name: str, page_count: int):
        self.name = name
        self.page_count = page_count
        self.pages_read = 0

    def __repr__(self):
        return(
            f"<Book {self.name}, read {self.pages_read} pages out of {self.page_count}>"
        )

    def read(self, pages: int):
        if self.pages_read + pages > self.page_count:
            raise TooManyPagesReadError(
                f"You tried to read {self.pages_read + pages} pages, but this book only has {self.page_count} pages."
            )
        self.pages_read += pages
        print(f"You have now read {self.pages_read} pages out of {self.page_count}.")

>>> python101 = Book("Python 101", 50)
try:
    >>> python101.read(35)
    >>> python101.read(50)
except TooManyReadError as e:
    print(e)
```

# Les assertions 1/3

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée. En général, on les utilise dans des blocs try ... except.

Voyons comment cela fonctionne : nous allons pour l'occasion découvrir un nouveau mot-clé (encore un), assert. Sa syntaxe est la suivante :

```
try:  
    1/0  
  
File "<stdin>", line 3  
  
    ^  
SyntaxError: invalid syntax
```

Il existe un mot-clé que l'on peut utiliser dans ce cas. Son nom est pass et sa syntaxe est très simple d'utilisation :

```
try:  
    # Test d'instruction(s)  
except type_de_l_exception: # Rien ne doit se passer en cas d'erreur  
    pass
```

# Les assertions 2/3

Si le test renvoie True, l'exécution se poursuit normalement. Sinon, une exception AssertionError est levée.

```
var = 5
assert var == 5
assert var == 8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Comme vous le voyez, la ligne 2 s'exécute sans problème et ne lève aucune exception. On teste en effet si var == 5. C'est le cas, le test est donc vrai, aucune exception n'est levée.

À la ligne suivante, cependant, le test est var == 8. Cette fois, le test est faux et une exception du type AssertionError est levée.

# Les assertions 3/3

À quoi cela sert-il, concrètement ?

Dans le programme testant si une année est bissextile, on pourrait vouloir s'assurer que l'utilisateur ne saisit pas une année inférieure ou égale à 0 par exemple. Avec les assertions, c'est très facile à faire :

```
annee = input("Saisissez une année supérieure à 0 :")
try:
    annee = int(annee) # Conversion de l'année
    assert annee > 0
except ValueError:
    print("Vous n'avez pas saisi un nombre.")
except AssertionError:
    print("L'année saisie est inférieure ou égale à 0.")
```

# Lever une exception

On utilise un nouveau mot-clé pour lever une exception... le mot-clé raise.

```
raise TypeDeLException("message à afficher")
```

Prenons un petit exemple, toujours autour de notre programme bissextile. Nous allons lever une exception de type ValueError si l'utilisateur saisit une année négative ou nulle.

```
annee = input() # L'utilisateur saisit l'année
try:
    annee = int(annee) # On tente de convertir l'année
    if annee<=0:
        raise ValueError("l'année saisie est négative ou nulle")
except ValueError:
    print("La valeur saisie est invalide (l'année est peut-être négative).")
```

# Résumé

- La syntaxe d'une assertion est assert test.
- Les assertions lèvent une exception AssertionError si le test échoue.
- On peut lever une exception grâce au mot-clé raise suivi du type de l'exception.
- On peut intercepter les erreurs (ou exceptions) levées par notre code grâce aux blocs try except.

# Compréhension

[https://www.teclado.com/30-days-of-python/python-30-day-15-comprehensions?\\_\\_s=wj99fjk10g15ts6m4rh](https://www.teclado.com/30-days-of-python/python-30-day-15-comprehensions?__s=wj99fjk10g15ts6m4rh)



Day 15 Comprehensions.pdf

# List comprehension 1/9



Complete Python Bootcamp

List Comprehensions are a unique way of quickly creating a list with Python.

If you find yourself using a for loop along with `.append()` to create a list, List Comprehensions are a good alternative!

To do this, let's go to a Jupyter Notebook!

# List comprehension 2/9

```
# Example 1:  
  
vals = [expression for value in collection if condition]  
  
numbers = [0, 1, 2, 3, 4]  
doubled_numbers = []  
  
for number in range(5):  
    doubled_numbers.append(number*2)  
  
print(double_numbers)  
  
# Or with a list comprehension  
doubled_numbers = [number * 2 for number in range(5)]
```

```
# Example 2:  
  
vals = []  
for value in collection:  
    if condition:  
        vals.append(expression)  
  
# Or with a list comprehension  
even_squares = [x * x for x in range(10) if not x % 2]  
>>> even_squares  
[0, 4, 16, 36, 64]
```

# List comprehension 3/9

# Example 3:

```
friend_ages = [22, 31, 35, 37]
age_strings = [f"My friend is {age} years old." for age in friend_ages]

print(age_strings)
```

# Example 4:

```
vals = []
for value in collection:
    if condition:
        vals.append(expression)
```

# Or with a list comprehension

```
even_squares = [x * x for x in range(10) if not x % 2]
>>> even_squares
[0, 4, 16, 36, 64]
```

# List comprehension 4/9

# Example 5:

```
names = ["Rolf", "Bob", "Jen"]
lower = [name.lower() for name in names]
print(lower)
```

Example 6:

```
friend = input("Enter your friend name: ")
friends = ["Rolf", "Bob", "Jen", "Charlie", "Anne"]
friends_lower = [name.lower() for name in friends]

if friend.lower() in friends_lower:
    print(f"{friend.title()} is one of your friends.")
```

# List compréhension 5/9

## Parcours simple

Les compréhensions de liste permettent de parcourir une liste en renvoyant une seconde, modifiée ou filtrée. Pour l'instant, nous allons voir une simple modification.

```
liste_origine = [0, 1, 2, 3, 4, 5]
[nb * nb for nb in liste_origine]
[0, 1, 4, 9, 16, 25]
```

Étudions un peu la ligne 2 de ce code. Comme vous avez pu le deviner, elle signifie en langage plus conventionnel « Mettre au carré tous les nombres contenus dans la liste d'origine ». Nous trouvons dans l'ordre, entre les crochets qui sont les délimiteurs d'une instruction de compréhension de liste :

nb \* nb: la valeur de retour. Pour l'instant, on ne sait pas ce qu'est la variable nb, on sait juste qu'il faut la mettre au carré. Notez qu'on aurait pu écrire nb\*\*2, cela revient au même.

for nb in liste\_origine: voilà d'où vient notre variable nb. On reconnaît la syntaxe d'une boucle for, sauf qu'on n'est pas habitué à la voir sous cette forme.

Quand Python interprète cette ligne, il va parcourir la liste d'origine et mettre chaque élément de la liste au carré. Il renvoie ensuite le résultat obtenu, sous la forme d'une liste qui est de la même longueur que celle d'origine. On peut naturellement capturer cette nouvelle liste dans une variable.

# List compréhension 6/9

## exemple1

Générons un aléatoirement un octet:

```
import random

liste_random = [str(random.choice(range(2))) for _ in range(8)]
print("".join(liste_random))
```



Vous noterez qu'on utilise un nom de variable quelque peu particulier pour la boucle for, dans cette list compréhension. En effet, cette boucle sert uniquement à répéter l'opération 8 fois, mais on ne fait aucune utilisation de la variable générée à chaque itération de la boucle for. Par convention, quand on déclare une variable qui n'est pas utilisée, on utilise un Under score.

# List compréhension 7/9

## exemple 2

Filtrage avec un branchement conditionnel

On peut aussi filtrer une liste de cette façon :

```
liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[nb for nb in liste_origine if nb % 2 == 0]
>>> [2, 4, 6, 8, 10]
```

On rajoute à la fin de l'instruction une condition qui va déterminer quelles valeurs seront transférées dans la nouvelle liste. Ici, on ne transfère que les valeurs paires. Au final, on se retrouve donc avec une liste deux fois plus petite que celle d'origine.

Mélangeons un peu tout cela

Il est possible de filtrer et modifier une liste assez simplement. Par exemple, on a une liste contenant les quantités de fruits stockées pour un magasin (je ne suis pas sectaire, vous pouvez prendre des hamburgers si vous préférez). Chaque semaine, le magasin va prendre dans le stock une certaine quantité de chaque fruit, pour la mettre en vente. À ce moment, le stock de chaque fruit diminue naturellement. Inutile, en conséquence, de garder les fruits qu'on n'a plus en stock.

Je vais un peu reformuler. On va avoir une liste simple, qui contiendra des entiers, précisant la quantité de chaque fruit (c'est abstrait, les fruits ne sont pas précisés). On va faire une compréhension de liste pour diminuer d'une quantité donnée toutes les valeurs de cette liste, et on en profite pour retirer celles qui sont inférieures ou égales à 0.

```
qtt_a_retirer = 7 # On retire chaque semaine 7 fruits de chaque sorte
fruits_stockes = [15, 3, 18, 21] # Par exemple 15 pommes, 3 melons...
[nb_fruits-qtt_a_retirer for nb_fruits in fruits_stockes if nb_fruits>qtt_a_retirer]
>>> [8, 11, 14]
```

# List compréhension 8/9

## exemple 2

Filtrage avec un branchement conditionnel

On peut aussi filtrer une liste de cette façon :

```
liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[nb for nb in liste_origine if nb % 2 == 0]
>>> [2, 4, 6, 8, 10]
```

On rajoute à la fin de l'instruction une condition qui va déterminer quelles valeurs seront transférées dans la nouvelle liste. Ici, on ne transfère que les valeurs paires. Au final, on se retrouve donc avec une liste deux fois plus petite que celle d'origine.

Mélangeons un peu tout cela

Il est possible de filtrer et modifier une liste assez simplement. Par exemple, on a une liste contenant les quantités de fruits stockées pour un magasin (je ne suis pas sectaire, vous pouvez prendre des hamburgers si vous préférez). Chaque semaine, le magasin va prendre dans le stock une certaine quantité de chaque fruit, pour la mettre en vente. À ce moment, le stock de chaque fruit diminue naturellement. Inutile, en conséquence, de garder les fruits qu'on n'a plus en stock.

Je vais un peu reformuler. On va avoir une liste simple, qui contiendra des entiers, précisant la quantité de chaque fruit (c'est abstrait, les fruits ne sont pas précisés). On va faire une compréhension de liste pour diminuer d'une quantité donnée toutes les valeurs de cette liste, et on en profite pour retirer celles qui sont inférieures ou égales à 0.

```
qtt_a_retirer = 7 # On retire chaque semaine 7 fruits de chaque sorte
fruits_stockes = [15, 3, 18, 21] # Par exemple 15 pommes, 3 melons...
[nb_fruits-qtt_a_retirer for nb_fruits in fruits_stockes if nb_fruits>qtt_a_retirer]
>>> [8, 11, 14]
```

# List compréhension 9/9

## exemple 3

Nouvelle application concrète

Nous allons en gros reprendre l'exemple précédent, en le modifiant un peu pour qu'il soit plus cohérent. Nous travaillons toujours avec des fruits sauf que, cette fois, nous allons associer un nom de fruit à la quantité restant en magasin. Nous verrons au prochain chapitre comment le faire avec des dictionnaires ; pour l'instant on va se contenter de listes :

```
inventaire = [  
    ("pommes", 22),  
    ("melons", 4),  
    ("poires", 18),  
    ("fraises", 76),  
    ("prunes", 51),  
]
```

```
# On change le sens de l'inventaire, la quantité avant le nom  
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit, qtt in inventaire]  
  
# On n'a plus qu'à trier dans l'ordre décroissant l'inventaire inversé  
# On reconstitue l'inventaire trié  
inventaire = [(nom_fruit, qtt) for qtt, nom_fruit in sorted(inventaire_inverse, \  
    reverse=True)]
```

Vous pouvez trier l'inventaire inversé avant la reconstitution, si vous trouvez cela plus compréhensible. Il faut privilégier la lisibilité du code.

```
# On change le sens de l'inventaire, la quantité avant le nom  
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit, qtt in inventaire]  
# On trie l'inventaire inversé dans l'ordre décroissant  
inventaire_inverse.sort(reverse=True)  
# Et on reconstitue l'inventaire  
inventaire = [(nom_fruit, qtt) for qtt, nom_fruit in inventaire_inverse]
```

# List compréhension 10/9

## exemple 4

```
# On change le sens de l'inventaire, la quantité avant le nom
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit, qtt in inventaire]

# On n'a plus qu'à trier dans l'ordre décroissant l'inventaire inversé
# On reconstitue l'inventaire trié
inventaire = [(nom_fruit, qtt) for qtt, nom_fruit in sorted(inventaire_inverse, \
    reverse=True)]
```

Vous pouvez trier l'inventaire inversé avant la reconstitution, si vous trouvez cela plus compréhensible. Il faut privilégier la lisibilité du code.

```
# On change le sens de l'inventaire, la quantité avant le nom
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit, qtt in inventaire]
# On trie l'inventaire inversé dans l'ordre décroissant
inventaire_inverse.sort(reverse=True)
# Et on reconstitue l'inventaire
inventaire = [(nom_fruit, qtt) for qtt, nom_fruit in inventaire_inverse]
```

Un autre exemple:

```
# Exemple1:
celsius = [0, 10, 20, 34.5]
fahrenheit = [((9 / 5) * temp + 32) for temp in celsius]
print(fahrenheit)
>>> [32.0, 50.0, 68.0, 94.1]
```

```
# Exemple2:
mylist = [x ** 2 for x in range(0, 11) if x % 2 == 0]
print(mylist)
>>> [32.0, 50.0, 68.0, 94.1]
>>> [0, 4, 16, 36, 64, 100]
```

# List compréhension 11/9

## exemple 5

```
results = [x if x%2==0 else 'ODD' for x in range(0,11)]  
print(results)  
>>> [0, 'ODD', 2, 'ODD', 4, 'ODD', 6, 'ODD', 8, 'ODD', 10]
```

Notons que l'ordre change par rapport à un if sans else, mylist = [x\*\*2 for x in range(0,11) if x%2==0] vs results = [x if x%2==0 else 'ODD' for x in range(0,11)]

It is also possible to run nested loop but you'll sacrifice readability

```
mylist = [x*y for x in [2,4,6] for y in [1,10,100]]  
mylist  
>>> [2, 20, 200, 4, 40, 400, 6, 60, 600]
```

# List Comprehension 12

<https://teclado.com/30-days-of-python/python-30-day-15-comprehensions>



Day 15 Comprehensions.pdf

```
numbers = [0, 1, 2, 3, 4]
doubled_numbers = []
for num in numbers:
    doubled_numbers.append(num * 2)

print(doubled_numbers)

# -- List comprehension --
numbers = [0, 1, 2, 3, 4] # list(range(5)) is better
doubled_numbers = [num * 2 for num in numbers]
# [num * 2 for num in range(5)] would be even better.

print(doubled_numbers)

# -- You can add anything to the new list --
friend_ages = [22, 31, 35, 37]
age_strings = [f"My friend is {age} years old." for age in friend_ages]

print(age_strings)

# -- This includes things like --
names = ["Rolf", "Bob", "Jen"]
lower = [name.lower() for name in names]

# That is particularly useful for working with user input.
# By turning everything to lowercase, it's less likely we'll miss a match.

friend = input("Enter your friend name: ")
friends = ["Rolf", "Bob", "Jen", "Charlie", "Anne"]
friends_lower = [name.lower() for name in friends]

if friend.lower() in friends_lower:
    print(f"I know {friend}!")
```

# Comprehension with conditional 13

<https://teclado.com/30-days-of-python/python-30-day-20-map-filter>



Day 20 map, filter, and Conditional Comprehensions.pdf

```
ages = [22, 35, 27, 21, 20]
odds = [n for n in ages if n % 2 == 1]

# -- with strings --

friends = ["Rolf", "ruth", "charlie", "Jen"]
guests = ["jose", "Bob", "Rolf", "Charlie", "michael"]

friends_lower = [f.lower() for f in friends]

present_friends = [name.capitalize() for name in guests if name.lower() in friends_lower]

# -- nested list comprehensions --# Don't do this, because it's almost completely unreadable.
# Splitting things out into variables is better.

friends = ["Rolf", "ruth", "charlie", "Jen"]
guests = ["jose", "Bob", "Rolf", "Charlie", "michael"]

present_friends = [
    name.capitalize() for name in guests if name.lower() in [f.lower() for f in friends]
]
```

# Set a dictionary comprehension 14

Exemple 1:

```
friends = ["Rolf", "ruth", "charlie", "Jen"]
guests = ["jose", "Bob", "Rolf", "Charlie", "michael"]
```

```
friends_lower = {n.lower() for n in friends}
guests_lower = {n.lower() for n in guests}
```

```
present_friends = friends_lower.intersection(guests_lower)
present_friends = {name.capitalize() for name in friends_lower & guests_lower}
```

```
print(present_friends)
```

Example 2:

```
# Transforming data for easier consumption and processing is a very common task.
# Working with homogeneous data is really nice, but often you can't (e.g. when working with user input!).
# -- Dictionary comprehension –
# Works just like set comprehension, but you need to do key-value pairs.
```

```
friends = ["Rolf", "Bob", "Jen", "Anne"]
time_since_seen = [3, 7, 15, 11]
```

```
long_timers = {
    friends[i]: time_since_seen[i]
    for i in range(len(friends))
    if time_since_seen[i] > 5
}
```

```
print(long_timers)
```

# Résumé

Les compréhensions de listes permettent de parcourir et filtrer une séquence en renvoyant une nouvelle.

La syntaxe pour effectuer un filtrage est la suivante :`nouvelle_squence = [element for element in ancienne_squence if condition]`

Comme pour les liste compréhension on peut écrire un set compréhension ou un dictionary compréhensions [voir](#)

# Advance built-in functions in Python

# Generator

<https://www.teclado.com/30-days-of-python/python-30-day-22-iterators>  
<https://www.teclado.com/30-days-of-python/python-30-day-23-generators-yield>



Day 22 Iterators.pdf



Day 23 Generators and Generator Expressions.pdf

A generator in python is a special function that remembers the state it's in, in between executions so you can run the function multiple times and it would remember what it did the last time you ran it. Let's have a look to an Example:  
Imagine you wish to build a list of 100 numbers:

```
def hundred_numbers():
    nums = []
    i = 0
    while i < 100:
        nums.append(i)
        i += 1
    return nums

print(hundred_numbers())
```

The problem with the code above is that you'll store your 100 numbers in memory. So, even if 100 number is not a lot it will become a problem if you start working on million or billion numbers. But more importantly you don't need the 100 numbers at once, you only need them 1 by 1. So the code above is very inefficient. So let's rewrite this code in more efficient way:

```
def hundred_numbers():
    i = 0
    while i < 100:
        yield i
        i += 1

g = hundred_numbers()
print(next(g))
print(next(g))
```

# Generator

It is really, important to remember in generators, they remember where they were, when you create one that's it from the moment you start using it and the moment your start calling next on it, it remembers where it is, and you can't go backwards.

# Generator exercise

Recall the code in an earlier lecture **Two loop key words + finding prime numbers**, where we learned how to find prime numbers:

```
for n in range(2, 20):
    for x in range(2, n):
        if n % 2 == 0:
            print('{} equals {} * {}'.format(n, x, n//x))
            break
        else:
            print('{} is a prime number.'.format(n))
```

We used some nested for loops as well as **for – else** statement to determine primality of a number in a certain range (from 2 to 20 in the above code).

In this exercise, we ask you to define a generator function which take bound as input and generates prime numbers up to, but not including, the bound parameter.

```
def prime_generator(bound):
    for n in range(2, bound): # n starts from 2 to bound
        for x in range(2, n): # check if there is a number x (1<x<n) that can divide n
            if n % x == 0: # as long as we can find any such x, then n is not prime
                break
        else: # if no such x is found after exhausting all 1<x<n
            yield n # generate this prime
```

# Generator

```
from sys import getsizeof

values = (x * 2 for x in range(100000)) -> this is a generator expression
print("gen:", getsizeof(values))
>> 120

values = [x * 2 for x in range(100000)] -> list comprehension
print("List:", getsizeof(values))
>> 824464
```

A list is taking over 824464 bytes of memory whereas our generator object only take 120 bytes. So when dealing with large data set or potentially an infinite stream of data set it is necessary to use generator expression rather than list which consume too much memory.

However because the generator object generate values rather than storing them in memory, you can't get the lenght of a generator object.

```
from sys import getsizeof

values = (x * 2 for x in range(100000)) -> this is a generator expression
print(len(values))
>> TypeError: object of type 'generator' has no len()
```

# Generator classe and iterator

```
class FirstHundredGenerator:  
    def __init__(self):  
        self.number = 0  
  
    def __next__(self):  
        if self.number < 100:  
            current = self.number  
            self.number += 1  
            return current  
        else:  
            raise StopIteration() # it says python we have reached the end of generator (100)
```

```
my_gen = FirstHundredGenerator()  
print(next(my_gen))  
print(next(my_gen))
```

# Generator classe and iterator

In the previous lecture, we learned a new way of defining a generator, which we refer to as generator classes, or class-based generators.

Recall that in last exercises, we built a function-based generator which generates prime numbers like this:

```
def prime_generator(bound):
    for n in range(2, bound):
        for x in range(2, n)
            if n % x == 0:
                break
            else:
                yield n
```

In this exercise, refactor the above generator, a function-based generator into a generator class.

Hint: Think about how we can turn a class into a generator, what method(s) do we need to define?

```
class PrimeGenerator:
    def __init__(self, stop):
        self.stop = stop
        self.start = 2

    def __next__(self):
        for n in range(self.start, self.stop): # always search from current start (inclusive) to stop (exclusive)
            for x in range(2, n):
                if n % x == 0: # not prime
                    break
                else: # n is prime, because we've gone through the entire loop without having a non-prime situation
                    self.start = n + 1 # next time we need to start from n + 1, otherwise we will be trapped on n
                    return n # return n for this round
        raise StopIteration() # this is what tells Python we've reached the end of the generator
```

```
prime = PrimeGenerator(10)
print(next(prime))
print(next(prime))
print(next(prime))
```

# Generator iterator vs iterable

The **iterator** is used to get the next value from sequence or generated file. So the previous slide is an iterator not an iterable.

**Iterator:** used to get the next value.

You can iterate over an **iterable**.

**Iterable:** used to go over all the values of the iterator

All **Generator** are **iterator**.

# Iterables in Python

"""  
So what in the heck is an iterable?

Funny you'd ask! An iterable is an object that has an `\_\_iter\_\_` method defined. The `\_\_iter\_\_` method \*must return an iterator\*.

Here's an example of using our generator to make an iterable.

```
class FirstHundredGenerator:  
    def __init__(self):  
        self.number = 0  
  
    def __next__(self):  
        if self.number < 100:  
            current = self.number  
            self.number += 1  
            return current  
        else:  
            raise StopIteration()
```

```
class FirstHundredIterable:  
    def __iter__(self):  
        return FirstHundredGenerator()
```

Now we have an iterable which uses the iterator to get the next value of the séquencée it generates. We can do this:

```
print(sum(FirstHundredIterable())) # gives 4950  
  
for i in FirstHundredIterable():  
    print(i)
```

Wait... I remember something about for loops. We needed an object with `\_\_len\_\_` and `\_\_getitem\_\_` defined!

How come we can use a for loop with this object that doesn't have either of those?

Here's something new! You can perform iteration over an iterable. An iterable either has:

- \* `\_\_len\_\_` and `\_\_getitem\_\_` defined; or
- \* An `\_\_iter\_\_` method that returns an iterator.

If you have either of those two, you have yourself an iterable.

---

So the `FirstHundredIterable` is returning an object of type `FirstHundredGenerator`.

Inside `FirstHundredGenerator`, what is `self`?

(Hint: it's an object, what is its type?)

(Hint hint: it's of type `FirstHundredGenerator`).

Knowing that, we can change the generator to this:

----

```
class FirstHundredGenerator:
```

```
    def __init__(self):  
        self.number = 0
```

```
    def __next__(self):  
        if self.number < 100:  
            current = self.number  
            self.number += 1  
            return current
```

else:  
 raise StopIteration()

```
def __iter__(self):  
    return self
```

"""  
And then we don't need a separate iterable at all—the generator itself is now both an iterator and an iterable.  
"""

# Destructuring Syntax

# Destructuring syntax

```
friends = [("Rolf", 25), ("Anne", 37), ("Charlie", 31), ("Bob", 22)]  
  
for name, age in friends:  
    print(f'{name} is {age} years old.')
```

# Utiliser des fichiers et des dossiers

[https://www.teclado.com/30-days-of-python/python-30-day-14-files?\\_\\_s=wj99fjkc10g15ts6m4rh](https://www.teclado.com/30-days-of-python/python-30-day-14-files?__s=wj99fjkc10g15ts6m4rh)



Day 14 Working with Files.pdf

# Utiliser des fichiers 1/

```
my_file = open('data.txt', 'r')
file_content = my_file.read()

my_file.close()

print(file_content)
>> Rolph

user_name = input('Enter your name: ')

my_file_writing = open('data.txt', 'w')
my_file_writing.write(user_name)

my_file_writing.close()
```

data.txt

Rolph

# Utiliser des fichiers /

Concaténation d'un chemin et un directory (sans se soucier des / ou \)

```
import os  
chemin = "c:\Users\Palleau Julien\Documents"  
dossier = os.path.join(chemin, "GitHub")  
print(dossier)
```

Changer de répertoire

```
import os  
os.chdir("C:\tests python")
```

Ouverture d'un fichier

```
with open(mon_fichier, mode_ouverture) as variable:  
    # Opérations sur le fichier
```

```
with open('fichier.txt', 'r') as mon_fichier:  
    texte = mon_fichier.read()
```

Le mot-clé **with** permet de créer un "context manager" (gestionnaire de contexte) qui vérifie que le fichier est ouvert et fermé, même si des erreurs se produisent pendant le bloc. Vous verrez plus loin d'autres objets utilisant le même mécanisme. Qu'est-ce que le "context manager":



Context Managers and Python's with Statement – Real Python.pdf

```
# In Python 3.4+ you can use  
# contextlib.suppress() to selectively  
# ignore specific exceptions:  
  
import contextlib  
  
with contextlib.suppress(FileNotFoundError):  
    os.remove('somefile.tmp')  
  
# This is equivalent to:  
  
try:  
    os.remove('somefile.tmp')  
except FileNotFoundError:  
    pass
```

```
os.remove('somefile.tmp')  
except FileNotFoundError:  
    pass  
  
# contextlib.suppress docstring:  
#  
# "Return a context manager that suppresses any  
# of the specified exceptions if they occur in the body  
# of a with statement and then resumes execution with  
# the first statement following the end of  
# the with statement."
```

# Utiliser des fichiers /

Tester l'existence du fichier

```
import os.path

chemin = "/tmp/dir/dir2/monFichier.txt"
print(os.path.exists(chemin))
>>> True

print(os.path.isfile(chemin))
>>> True

print(os.path.isdir(chemin))
>>> False

print(os.path.isdir(os.path.dirname(chemin)))
>>> True
```

# Utiliser des fichiers /

```
import os
Import json

# Ecrire du texte à l'intérieur d'un fichier:
with open('fichier.txt', 'a', encoding='utf-8') as f: # encoding est nécessaire pour les accents.
    f.write("Bonjour tout le monde ! \n")

# Lire toutes les lignes d'un fichier:
# Malheureusement les retour à la ligne \n seront affichés
with open('fichier.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        print(line)

# Pour éviter d'afficher les retour à la ligne:
with open('fichier.txt', 'r')as f:
    lines = f.read().splitlines()
    for line in lines: # affichage en colonne
        print(line)

# Lire un fichier json
with open(file) as f:
    line = json.load(f)
```

# Utiliser des fichiers /

```
Chemin = r"C:\Users\MOTTIER LUCIE\Documents\GitHub\Udemy\LaFormationCompletePython"
f = open(chemin, "r")
print(f.read())
f.seek(0) # Permet de remettre le curseur sur la première ligne avant le premier caractère.
contenu = f.read()
print(contenu)
f.close()

# lecture jusqu'au 10eme caractère.
Chemin = r"C:\Users\MOTTIER LUCIE\Documents\GitHub\Udemy\LaFormationCompletePython"
f = open(chemin, "r")
print(f.read(10))
f.seek(0) # Permet de remettre le curseur sur la première ligne avant le premier caractère.
contenu = f.read(10)
print(contenu)
f.close()
```

# Utiliser des fichiers /

```
# Si je veux lire mon fichier plusieurs fois !!!
import os
import json

# Current directory under which a .py is executed
cur_dir = os.path.dirname(__file__) # __file__ permet de recuperer le chemin complet vers le script execute
                                    # dirname permet de recuperer le dossier parent
# autre possibilite pour recuperer cur_dir
cur_dir = os.getcwd() # recupere le dossier parent ou le script python est execute
fichier = os.path.join(cur_dir, "fichier.txt") # off fait une concaténation du directory path avec le file name

# Ecrire du texte à l'intérieur d'un fichier
with open(fichier, 'a') as f:
    f.write("Bonjour tout le monde ! \n")

# Ecrire dans un fichier json
with open(fichier, 'w') as f:
    json.dump(list(range(10)), f, indent = 4)

# Lire toutes les lignes d'un fichier
with open(fichier, 'r') as f:
    lines = f.readlines()
    for line in lines:
        print(line.rstrip()) # rstrip enlève le retour à la ligne, sinon on a un mot une ligne blanche un mot etc...
```

# Utiliser des fichiers /

Enregistrer des objets dans des fichiers

```
import pickle
```

Ouverture d'un fichier

```
with open('donnees', 'wb') as f:  
    mon_pickler = pickle.Pickler(f)  
    # enregistrement ...
```

Notez le mode d'ouverture : on ouvre le fichier données en mode d'écriture binaire. Il suffit de rajouter, derrière la lettre symbolisant le mode, la lettre b pour indiquer un mode binaire.

# Utiliser des fichiers /

## Enregistrer un objet dans un fichier

On utilise la méthode **dump** du pickler pour enregistrer l'objet. Son emploi est des plus simples:

```
score = {  
    "joueur 1": 5,  
    "joueur 2": 35,  
    "joueur 3": 20,  
    "joueur 4": 2,  
}  
with open('donnees', 'wb') as f:  
    mon_pickler = pickle.Pickler(f)  
    mon_pickler.dump(score)
```

# Utiliser des fichiers /

## Récupérer nos objets enregistrés

Nous allons utiliser une autre classe définie dans notre module pickle. Cette fois, assez logiquement, c'est la classe Unpickler.

Commençons par créer notre objet. À sa création, on lui passe le fichier dans lequel on va lire les objets. Puisqu'on va lire, on change de mode, on repasse en mode r, et même rb puisque le fichier est binaire.

```
with open('donnees', 'rb') as f:  
    mon_depickler = pickle.Unpickler(f)  
    # Lecture des objets contenus dans le fichier...
```

Pour lire l'objet dans notre fichier, il faut appeler la méthode load de notre depickler. Elle renvoie le premier objet qui a été lu (s'il y en a plusieurs, il faut l'appeler plusieurs fois).

```
with open('donnees', 'rb') as f:  
    mon_depickler = pickle.Unpickler(f)  
    score_recupere = mon_depickler.load()
```

# CheetSheet/

<https://www.pythonforbeginners.com/cheatsheet/python-file-handling>

## File Handling

File handling in Python requires no importing of modules.

## File Object

Instead we can use the built-in object “file”. That object provides basic functions and methods necessary to manipulate files by default. Before you can read, append or write to a file, you will first have to use it using Python’s built-in open() function. In this post I will describe how to use the different methods of the file object.

### Open()

The open() function is used to open files in our system, the filename is the name of the file to be opened. The mode indicates, how the file is going to be opened “r” for reading, “w” for writing and “a” for appending. The open function takes two arguments, the name of the file and the mode for which we would like to open the file. By default, when only the filename is passed, the open function opens the file in read mode.

### Example

This small script, will open the (hello.txt) and print the content. This will store the file information in the file object “filename”.

```
filename = "hello.txt"
file = open(filename, "r")
for line in file:
    print line,
```

### Read ()

The read functions contains different methods, read(), readline() and readlines()

```
read()  #return one big string
readline #return one line at a time
read-lines #returns a list of lines
```

### Write ()

This method writes a sequence of strings to the file.

# Lister fichiers et dossiers /

```
from pathlib import Path

# Absolute path
path = Path()
for file in path.glob("*.*"): # liste tous les fichiers
    print(file)

# Absolute path
path = Path()
for file in path.glob("*.py"): # liste tous les fichiers python
    print(file)

# Absolute path
path = Path()
for file in path.glob("*"): # liste tous les repertoires
    print(file)
```

# Lister fichiers et dossiers /

La fonction `listdir(repertoire)` du module `os.path` retourne le contenu du répertoire passé en argument sans distinction entre les fichiers et les répertoires.

```
import os.path
print(os.listdir("/tmp/dir"))
['villes.json', 'quantiteFournitures.json', 'dir2']
```

Exemple: Créons un programme de surveillance de la présence ou non d'un dossier

```
import os
import time

dossier = r"C:\Users\MOTTIER LUCIE\Documents"
dossier_a_chercher = "Python"

nbsec = input("Entrez un temps de rafraîchissement:")

while dossier_a_chercher not in os.listdir(dossier):
    print("Dossier introuvable...")
    time.sleep(int(nbsec))
print("Trouve")
```

# Lister fichiers et dossiers /

```
import os

chemin = "c:\MOTTIER Lucie\Documents\GitHub\Udemy\ LaFormationCompletePython"
dossier = os.path.join(chemin, "dossier", "test")

if not os.path.exists(dossier):
    os.makedirs(dossier)

# Autre possibilite
os.makedirs(dossier, exist_ok=True)
os.makedirs("/ dossier1 / dossier2 / dossier3") # cree les trois dossiers
```

# Supprimer un fichier ou un dossier /

```
import os

chemin = "c:\MOTTIER Lucie\Documents\GitHub\Udemy\ LaFormationCompletePython"
dossier = os.path.join(chemin, "dossier", "test")

if os.path.exists(dossier):
    os.removedirs(dossier)

# malheureusement il n'y a pas d'équivalent à makedirs(dossier, exist_ok=True) donc nous sommes obligé d'utiliser une structure conditionnelle.
```

La méthode `remove(fichier)` du module `os` et la fonction `rmmtree(dossier)` du module `shutil` permettent respectivement de supprimer un fichier et un répertoire.

```
import os
import shutil

os.remove("/tmp/dir/exemple.txt")
shutil.rmtree("/tmp/dir/perso")
```

# Exercice

Ask the user for a list of 3 friends

For each friend, we'll tell the user whether they are nearby

For each nearby friend, we'll save their name to 'nearby\_friends.txt'

```
friends = input('Enter three friend names, separated by commas (no spaces, please): ').split(',')  
  
people = open('people.txt', 'r')  
people_nearby = [line.strip() for line in people.readlines()]  
  
people.close()  
  
friends_set = set(friends)  
people_nearby_set = set(people_nearby)  
  
friends_nearby_set = friends_set.intersection(people_nearby_set)  
  
nearby_friends_file = open('nearby_friends.txt', 'w')  
  
for counter, friend in enumerate(friends_nearby_set):  
    if counter == len(friends_nearby_set) - 1: # this test is to avoid an empty line at the end of nearby_friends.txt  
        print(f'{friend} is nearby! Meet up with them.')  
        nearby_friends_file.write(f'{friend}')  
    else:  
        print(f'{friend} is nearby! Meet up with them.')  
        nearby_friends_file.write(f'{friend}\n')  
  
nearby_friends_file.close()
```

sample `questions.txt` file:

1+1=2

2+2=4

8-4=4

task description:

- read from `questions.txt`

- for each question, print out the question and wait for the user's answer

for example, for the first question, print out: `1+1=

- after the user answers all the questions, calculate her score and write it to the `result.txt` file

the result should be in such format: `Your final score is n/m.

where n and m are the number of correct answers and the maximum score respectively

```
# read from questions.txt and append each line into a list
questions = open("questions.txt", "r") # read from questions.txt

# read all lines and get rid of line break for each line, then append each stripped line to a list
question_list = [line.strip() for line in questions]
questions.close()

score = 0 # initialize score
total = len(question_list) # set total score

for line in question_list:
    # split equation with '=' into question and answer
    q, a = line.split("=")

    # print question and wait for user to input their answer
    ans = input(f"{q}=")

    if a == ans: # if user input matches answer
        score += 1 # increase score

result = open("result.txt", "w") # open result.txt
# write final score to result.txt
result.write(f"Your final score is {score}/{total}.")
result.close()
```

# CSV files /

```
# https://www.youtube.com/watch?v=W7QByFjVom8

file = open('csv_data.txt', 'r')
lines = file.readlines()
file.close()

lines = [line.strip() for line in lines[1:]]

for line in lines:
    person_data = line.split(',')
    name = person_data[0].title()
    age = person_data[1]
    university = person_data[2].title()
    degree = person_data[3].capitalize()

    print(f'{name} is {age}, studying {degree} at {university}.')

# how to create a csv line in order to store it.
sample_csv_value = ','.join(['Rolf', '25', 'MIT', 'Computer Science'])
print(sample_csv_value)
```

# CSV files with csv module /

```
# Without csv module

movies = [
    {"name": "The Matrix", "director": "Wachowski"}, 
    {"name": "Green Book", "director": "Farrelly"}, 
    {"name": "Amadeus", "director": "Forman"}]

def write_to_file(output):
    with open("file.csv", "w") as f:
        f.write("name,director\n")
        for line in output:
            f.write(f"{line['name']},{line['director']}\n")

def read_from_file():
    with open("file.csv", "r") as f:
        content = f.readlines()
        for line in content[1:]:
            columns = line.strip().split(",")
            print(f"Name: {columns[0]}\tDirector: {columns[1]}")
```

```
# With csv module
import csv

movies = [
    {"name": "The Matrix", "director": "Wachowski"}, 
    {"name": "Green Book", "director": "Farrelly"}, 
    {"name": "Amadeus", "director": "Forman"}]

def write_to_file(output):
    with open("file.csv", "w") as f:
        writer = csv.DictWriter(f, fieldnames=["name", "director"])
        writer.writeheader()
        writer.writerows(output)

def read_from_file():
    with open("file.csv", "r") as f:
        reader = csv.DictReader(f)
        for line in reader:
            print(f"Name: {line['name']}\tDirector: {line['director']}")
```

## Practical Python: the csv module

- ✓ Instead of manually writing and reading, use `csv.writer` and `csv.reader`
- ✓ `csv.DictWriter` and `csv.DictReader` help you with named data
- ✓ The `csv` module also takes care of formatting edge cases, like commas in your strings

# Exercice JASON to CSV file

json\_import.py

```
import json

file = open('friends_json.txt', 'r')
file_contents = json.load(file) # reads file and turns it to dictionary
file.close()

print(file_contents['friends'][0])

cars = [
    {'make': 'Ford', 'model': 'Fiesta'},
    {'make': 'Ford', 'model': 'Focus'}
]

file = open('cars_json.txt', 'w')
json.dump(cars, file)
file.close()

my_json_string = '[{"name": "Alfa Romeo", "released": 1950}]'
incorrect_car = json.loads(my_json_string) # here we use loads and not load for load string
print(incorrect_car[0]['name'])
```

friends\_json.txt

```
{
    "friends": [
        {
            "name": "Jose",
            "degree": "Applied Computing"
        },
        {
            "name": "Rolf",
            "degree": "Computer Science"
        },
        {
            "name": "Anna",
            "degree": "Physics"
        }
    ]
}
```

cars\_json.txt

```
[{"make": "Ford", "model": "Fiesta"}, {"make": "Ford", "model": "Focus"}]
```

In this exercise, create a CSV to JSON converter that can be handy for others. Your converter should achieve the following tasks:

*Manchester United, Manchester, UK*

*Real Madrid, Madrid, Spain*

*Juventus, Turin, Italy*

Read and process the file and store its content in JSON format into [json\\_file.txt](#). The according keys to each field in the CSV file are club, city and country. Thus the output should be, according to the given sample CSV file, like this: [{"club": "Manchester United", "country": "UK", "city": "Manchester"}, {"club": "Real Madrid", "country": "Spain", "city": "Madrid"}, {"club": "Juventus", "country": "Italy", "city": "Turin"}]

```
import json

json_list = []

with open('csv_file.txt', 'r') as f:
    content = f.readlines()
    i = 0
    for line in content:
        club, city, country = line.strip().split(',')
        data = {
            'club': club,
            'city': city,
            'country': country,
        }
        json_list.append(data)

json_file = open('json_file.txt', 'w')
json.dump(json_list, json_file)
json_file.close()
```

# Exercice Importing module

```
from addition import Addition

# You don't need to change the import statement
# now you can use Addition.add() function from the addition module like
this:
# res = Addition.add(100, 150)
# the Addition.add() function takes in two parameters `num1` and `num2`
and return the sum of `num1` and `num2`

# Please create and implement a Calculator class, which makes use of the
`addition` module.
# Your Calculator should achieve these goals:
# - It should implement `Addition.add()`, `subtract()`, `multiply()` and
`divide()` methods.
# - It cannot use addition, subtraction, multiplication and division
operators (`+`, `-`, `*` and `/`) directly.
# Instead, it should be only based on the `Addition.add()` function from
the `addition` module.
# To simplify the problem, you may expect input for the multiply() and
divide() methods are all non-integers,
# and will always be valid, i.e. all non-negative integers and no 0 as
divisor.

# the class definition and a sample class method `Addition.add()` is
provided below
class Calculator:

    # a sample add() method in our calculator is shown below
    # you may learn from it and implement the other methods
    @classmethod
    def add(cls, num1, num2):
        return Addition.add(num1, num2) # make use of add() from addition
module

    # implement a class method `subtract()` that takes in num1 and num2
    # your `subtract()` method cannot use the + - * / calculation operators,
but can use - as a negative sign operator
    @classmethod
    def subtract(cls, num1, num2):
        return Addition.add(num1, -num2)

    # implement a class method `multiply()` that takes in num1 and num2
    # your `multiply()` method cannot use the + - * / calculation operators,
but can use - as a negative sign operator
    # you may assume num1 and num2 are always non-negative integers
    @classmethod
    def multiply(cls, num1, num2):
        result = 0
        for i in range(num2):
            result = Addition.add(result, num1)
        return result

    # implement a class method `divide()` that takes in num1 and num2
    # your `divide()` method cannot use the + - * / calculation operators,
but can use - as a negative sign operator
    # you may assume num1 is always a non-negative integer, and num2 is
always a positive integer
    @classmethod
    def divide(cls, num1, num2):
        result = 0
        while num1 >= num2:
            num1 = Addition.add(num1, -num2)
            result = Addition.add(result, 1)
        return result

calc = Calculator()
# print(calc.subtract(7, 2))
# print(calc.multiply(10, 6))
print(calc.divide(50, 10))
```

# Résumé

## En résumé

On peut ouvrir un fichier en utilisant la fonction open prenant en paramètre le chemin vers le fichier et le mode d'ouverture.

On peut lire dans un fichier en utilisant la méthode read.

On peut écrire dans un fichier en utilisant la méthode write.

Un fichier doit être refermé après usage en utilisant la méthode close.

Le module pickle est utilisé pour enregistrer des objets Python dans des fichiers et les recharger ensuite.

Concaténation de chemin avec **os.path.join()**

Création d'un dossier avec **os.makedirs(dossier, exist\_ok=True)**

Vérification de l'existence d'un dossier avec **os.path.exists(dossier)**

Effacer un dossier avec **os.removedirs(dossier)**

# • Recherche récursive d'un fichier

# Recherche récursive

```
dossier = r"C:\Users\MOTTIER LUCIE\Documents\testglob"  
  
for repertoire in glob(dossier + '**/*', recursive=True):  
    print(repertoire)
```

Indique que l'on veut faire une recherche récursive.

Indique que l'on sélectionne tous les répertoires

# Recherche récursive de fichiers

```
import os
from glob import glob

dossier = "C:\Users\MOTTIER LUCIE\Documents"
fichier_a_trouver = "fichier_a_trouver.txt"

fichiers = glob(dossier + "**", recursive=True) # retourne une liste de fichiers
fichiers_trouves = [f for f in fichiers if os.path.split(f)[1] == fichier_a_trouver]
print(fichiers_trouves)
```

/\*\* mach tous les fichiers, et répertoires et sous répertoires.

os.path.split divise le path en 2 parties:  
os.path.split(f)[0] = retourne le path jusqu'au fichier  
os.path.split(f)[1] = retourne le nom du fichier

# Recherche recursive de fichiers

```
for root, directories, files in os.walk("/Downloads"):
    for file in files:
        print(file)
```

Recherche récursive depuis le répertoire /Downloads, et on retourne tous les fichiers à chaque niveau de répertoire exploré.

# Compter récursivement le nombre de fichier et répertoire dans un dossier

```
from glob import glob  
  
dossier = "C:\\Users\\MOTTIER LUCIE\\Documents"  
  
fichiers = glob(f "{dossier}/**", recursive=True) # retourne une liste de fichiers  
print(len(fichiers))
```

/\*\* mach tous les fichiers, et répertoires et sous répertoires.

# Déplacer des fichiers

# Déplacer des fichiers

```
import os
import shutil

# Déplacer des fichiers
-----
# Déplacer des fichiers avec os.rename()
# Avec cette façon il faut spécifier un fichier de départ le path jusqu'au nouveau répertoire et le nom du fichier d'arrive
os.rename('fichier1.txt', 'test\\fichier1.txt')

# Déplacer des fichiers avec shutil():
# Avec shutil on a juste à spécifier le fichier de départ et l'emplacement où l'on veut copier le fichier.
shutil.move("fichier1.txt", "test\\")
```

Aller chercher de  
l'aide avec les  
fonctions `dir` et `help`

Aller chercher de l'aide avec les fonctions dir et help

```
import random
from pprint import pprint

pprint(dir(random))
help(random.randint)
```

# Appréhendez les classes

# Appréhendez le vocabulaire 1/

```
class Voiture:  
    attribut de classe  
    pneus = 4  
    méthode  
    def __init__(self, marque):  
        attribut d'instance  
        self.marque = marque  
instance  
lamborghini = Voiture("Lamborghini")
```

# Appréhendez les classes 2/

Dans ce chapitre, sans plus attendre, nous allons créer nos premières classes, nos premiers attributs et nos premières méthodes. Nous allons aussi essayer de comprendre les mécanismes de la programmation orientée objet en Python.

Au-delà du mécanisme, l'orienté objet est une véritable philosophie.

## Les classes, tout un monde

Dans la partie précédente, j'avais brièvement décrit les objets comme des variables pouvant contenir elles-mêmes des fonctions et variables. Nous sommes allés plus loin tout au long de la seconde partie, pour découvrir que nos « fonctions contenues dans nos objets » sont appelées des méthodes. En vérité, je me suis cantonné à une définition « pratique » des objets, alors que derrière la POO (Programmation Orientée Objet) se cache une véritable philosophie.

## Pourquoi utiliser des objets ?

Les premiers langages de programmation n'incluaient pas l'orienté objet. Le langage C, pour ne citer que lui, n'utilise pas ce concept et il aura fallu attendre le C++ pour utiliser la puissance de l'orienté objet dans une syntaxe proche de celle du C.

Java, un langage apparu à peu près en même temps que Python, définit une philosophie assez différente de celle du C++ : contrairement à ce dernier, le Java exige que tout soit rangé dans des classes. Même l'application standard Hello World est contenue dans une classe.

En Python, la liberté est plus grande. Après tout, vous avez pu passer une partie de ce tutoriel sans connaître la façade objet de Python. Et pourtant, le langage Python est totalement orienté objet : en Python, tout est objet, vous n'avez pas oublié ? Quand vous croyez utiliser une simple variable, un module, une fonction..., ce sont des objets qui se cachent derrière.

Loin de moi l'idée de faire un comparatif entre différents langages. Ce sur quoi je souhaite attirer votre attention, c'est que plusieurs langages intègrent l'orienté objet, chacun avec une philosophie distincte. Autrement dit, si vous avez appris l'orienté objet dans un autre langage, tel que le C++ ou le Java, ne tenez pas pour acquis que vous allez retrouver les mêmes mécanismes et surtout, la même philosophie. Gardez autant que possible l'esprit dégagé de tout préjugé sur la philosophie objet de Python.

Pour l'instant, nous n'avons donc vu qu'un aspect technique de l'objet. J'irais jusqu'à dire que ce qu'on a vu jusqu'ici, ce n'était qu'une façon « un peu plus esthétique » de coder : il est plus simple et plus compréhensible d'écrire `ma_liste.append(5)` que `append_to_list(ma_liste, 5)`. Mais derrière la POO, il n'y a pas qu'un souci esthétique, loin de là.

# Appréhendez les classes 3/

## Choix du modèle

Bon, comme vous vous en souvenez sûrement (du moins, je l'espère), une classe est un peu un modèle suivant lequel on va créer des objets. C'est dans la classe que nous allons définir nos méthodes et attributs, les attributs étant des variables contenues dans notre objet.

Mais qu'allons-nous modéliser ? L'orienté objet est plus qu'utile dès lors que l'on s'en sert pour modéliser, représenter des données un peu plus complexes qu'un simple nombre, ou qu'une chaîne de caractères. Bien sûr, il existe des classes que Python définit pour nous : les nombres, les chaînes et les listes en font partie. Mais on serait bien limité si on ne pouvait faire ses propres classes.

Pour l'instant, nous allons modéliser... une personne. C'est le premier exemple qui me soit venu à l'esprit, nous verrons bien d'autres exemples avant la fin de la partie.

## Convention de nommage

Si on se réfère à la PEP 8 de Python, il est préférable d'utiliser pour des noms de classes la convention dite Camel Case.

Les PEP sont les « Python Enhancement Proposals », c'est à dire les propositions d'amélioration de Python.

Cette convention n'utilise pas le signe souligné \_ pour séparer les mots. **Le principe consiste à mettre en majuscule chaque lettre débutant un mot**, par exemple :**MaClasse**.

C'est donc cette convention que je vais utiliser pour les noms de classes. Libre à vous d'en changer, encore une fois rien n'est imposé.

Pour définir une nouvelle classe, on utilise le mot-clé class.

Sa syntaxe est assez intuitive :**class NomDeLaClasse:**

N'exécutez pas encore ce code, nous ne savons pas comment définir nos attributs et nos méthodes.

Petit exercice de modélisation : que va-t-on trouver dans les caractéristiques d'une personne ? Beaucoup de choses, vous en conviendrez. On ne va en retenir que quelques-unes : le nom, le prénom, l'âge, le lieu de résidence... allez, cela suffira.

Cela nous fait donc quatre attributs. Ce sont les variables internes à notre objet, qui vont le caractériser. Une personne telle que nous la modélisons sera caractérisée par son nom, son prénom, son âge et son lieu de résidence.

Pour définir les attributs de notre objet, il faut définir un constructeur dans notre classe. Voyons cela de plus près.

# Nos premiers attributs 4/

Nous avons défini les attributs qui allaient caractériser notre objet de classePersonne. Maintenant, il faut définir dans notre classe une méthode spéciale, appelée un constructeur, qui est appelée invariablement quand on souhaite créer un objet depuis notre classe.

Concrètement, un constructeur est une méthode de notre objet se chargeant de créer nos attributs. En vérité, c'est même la méthode qui sera appelée quand on voudra créer notre objet.

Voyons le code, ce sera plus parlant :

```
class Personne: # Définition de notre classe Personne
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son âge
    - son lieu de résidence"""

    def __init__(self): # Notre méthode constructeur
        """Pour l'instant, on ne va définir qu'un seul attribut"""
        self.nom = "Dupont"
```

Voyons en détail :

D'abord, la définition de la classe. Elle est constituée du mot-clé class, du nom de la classe et des deux points rituels « : ».

Une docstring commentant la classe. Encore une fois, c'est une excellente habitude à prendre et je vous encourage à le faire systématiquement. Ce pourra être plus qu'utile quand vous lancerez dans de grands projets, notamment à plusieurs.

La définition de notre constructeur. Comme vous le voyez, il s'agit d'une définition presque « classique » d'une fonction. Elle a pour nom `__init__`, c'est invariable : en Python, tous les constructeurs s'appellent ainsi. Nous verrons plus tard que les noms de méthodes entourés de part et d'autre de deux signes soulignés(`__nommethode__`)sont des méthodes spéciales. Notez que, dans notre définition de méthode, nous passons un premier paramètre nommé `self`.

Une nouvelle docstring. Je ne complique pas inutilement, je précise donc qu'on va simplement définir un seul attribut pour l'instant dans notre constructeur.

Dans notre constructeur, nous trouvons linstanciation de notre attribut nom. On crée une variable `self.nom` et on lui donne comme valeur Dupont. Je vais détailler un peu plus bas ce qui se passe ici.

# Nos premiers attributs 5/

Avant tout, pour voir le résultat en action, essayons de créer un objet issu de notre classe :

```
bernard = Personne()
bernard
<__main__.Personne object at 0x00B42570>
bernard.nom
'Dupont'
```

Quand on demande à l'interpréteur d'afficher directement notre objet bernard, il nous sort quelque chose d'un peu imbuvable... Bon, l'essentiel est la mention précisant la classe dont l'objet est issu. On peut donc vérifier que c'est bien notre classe Personne dont est issu notre objet. On essaye ensuite d'afficher l'attribut nom de notre objet bernard et on obtient 'Dupont' (la valeur définie dans notre constructeur). Notez qu'on utilise le point (.), encore et toujours utilisé pour une relation d'appartenance (nom est un attribut de l'objet bernard). Encore un peu d'explications :

## Quand on crée notre objet...

Quand on tape Personne(), on appelle le constructeur de notre classe Personne, d'une façon quelque peu indirecte que je ne détaillerai pas ici. Celui-ci prend en paramètre une variable un peu mystérieuse :self. En fait, il s'agit tout bêtement de notre objet en train de se créer. On écrit dans cet objet l'attribut nom le plus simplement du monde :self.nom = "Dupont". À la fin de l'appel au constructeur, Python renvoie notre objet self modifié, avec notre attribut. On va réceptionner le tout dans notre variable bernard.

Si ce n'est pas très clair, pas de panique ! Vous pouvez vous contenter de vous familiariser avec la syntaxe du constructeur Python, qui sera souvent la même, et laisser l'aspect un peu théorique de côté, pour plus tard. Nous aurons l'occasion d'y revenir avant la fin du chapitre.

## Étoffons un peu notre constructeur

Bon, on avait dit quatre attributs, on n'en a fait qu'un. Et puis notre constructeur pourrait éviter de donner les mêmes valeurs par défaut à chaque fois, tout de même !

C'est juste. Dans un premier temps, on va se contenter de définir les autres attributs, le prénom, l'âge, le lieu de résidence. Essayez de le faire, normalement vous ne devriez éprouver aucune difficulté.

Voici le code, au cas où :

# Nos premiers attributs 6/

```
class Personne:  
    """Classe définissant une personne caractérisée par :  
        - son nom  
        - son prénom  
        - son âge  
        - son lieu de résidence"""  
  
    def __init__(self): # Notre méthode constructeur  
        """Constructeur de notre classe. Chaque attribut va être instancié  
        avec une valeur par défaut... original"""  
  
        self.nom = "Dupont"  
        self.prenom = "Jean"  
        self.age = 33  
        self.lieu_residence = "Paris"
```

```
jean = Personne()  
print(jean.nom)  
print(jean.prenom)  
print(jean.age)  
print(jean.lieu_residence)  
>>> Dupont  
>>> Jean  
>>> 33  
>>> Paris
```

# Nos premiers attributs 7/

Une toute petite explication en ce qui concerne la ligne 11 : dans beaucoup de tutoriels, on déconseille de modifier un attribut d'instance (un attribut d'un objet) comme on vient de le faire, en faisant simplement `objet.attribut = valeur`. Si vous venez d'un autre langage, vous pourrez avoir entendu parler des accesseurs et mutateurs. Ces concepts sont repris dans certains tutoriels Python, mais ils n'ont pas précisément lieu d'être dans ce langage. Tout cela, je le détaillerai dans le prochain chapitre. Pour l'instant, il vous suffit de savoir que, quand vous voulez modifier un attribut d'un objet, vous écrivez `objet.attribut = nouvelle_valeur`. Nous verrons les cas particuliers plus loin.

Bon. Il nous reste encore à faire un constructeur un peu plus intelligent. Pour l'instant, quel que soit l'objet créé, il possède les mêmes nom, prénom, âge et lieu de résidence. On peut les modifier par la suite, bien entendu, mais on peut aussi faire en sorte que le constructeur prenne plusieurs paramètres, disons... le nom et le prénom, pour commencer.

```
class Personne:  
    """Classe définissant une personne caractérisée par :  
        - son nom  
        - son prénom  
        - son âge  
        - son lieu de résidence"""  
  
    def __init__(self, nom, prenom):  
        """Constructeur de notre classe"""""  
        self.nom = nom  
        self.prenom = prenom  
        self.age = 33  
        self.lieu_residence = "Paris"
```

```
bernard = Personne("Micado", "Bernard")  
bernard.nom  
>>> 'Micado'  
bernard.prenom  
>>> 'Bernard'  
bernard.age  
>>> 33
```

# Attributs de classe 8/

Dans les exemples que nous avons vus jusqu'à présent, nos attributs sont contenus dans notre objet. Ils sont propres à l'objet : si vous créez plusieurs objets, les attributs nom, prenom,... de chacun ne seront pas forcément identiques d'un objet à l'autre. Mais on peut aussi définir des attributs dans notre classe. Voyons un exemple :

```
class Compteur:  
    """Cette classe possède un attribut de classe qui s'incrémente à chaque  
    fois que l'on crée un objet de ce type"""  
  
    objets_crees = 0 # Le compteur vaut 0 au départ  
  
    def __init__(self):  
        """À chaque fois qu'on crée un objet, on incrémente le compteur"""""  
        Compteur.objets_crees += 1
```

On définit notre attribut de classe directement dans le corps de la classe, sous la définition et la docstring, avant la définition du constructeur. Quand on veut l'appeler dans le constructeur, on préfixe le nom de l'attribut de classe par le nom de la classe. Et on y accède de cette façon également, en dehors de la classe. Voyez plutôt :

```
print(Compteur.objets_crees)  
  
a = Compteur() # On crée un premier objet  
print(Compteur.objets_crees)  
  
b = Compteur()  
print(Compteur.objets_crees)  
>>> 0  
>>> 1  
>>> 2
```

À chaque fois qu'on crée un objet de type Compteur, l'attribut de classe objets\_crees s'incrémente de 1. Cela peut être utile d'avoir des attributs de classe, quand tous nos objets doivent avoir certaines données identiques. Nous aurons l'occasion d'en reparler par la suite.

# Les méthodes, la recette 9/

Les attributs sont des variables propres à notre objet, qui servent à le caractériser. Les méthodes sont plutôt des actions, comme nous l'avons vu dans la partie précédente, agissant sur l'objet. Par exemple, la méthode append de la classe list permet d'ajouter un élément dans l'objet list manipulé.

Pour créer nos premières méthodes, nous allons modéliser... un tableau. Un tableau noir, oui c'est très bien.

Notre tableau va posséder une surface (un attribut) sur laquelle on pourra écrire, que l'on pourra lire et effacer. Pour créer notre classe TableauNoiret notre attribut surface, vous ne devriez pas avoir de problème :

```
class TableauNoir:  
    """Classe définissant une surface sur laquelle on peut écrire,  
    que l'on peut lire et effacer, par jeu de méthodes. L'attribut modifié  
    est 'surface'''  
  
    def __init__(self):  
        """Par défaut, notre surface est vide'''  
        self.surface = ""
```

Nous avons déjà créé une méthode, aussi vous ne devriez pas être trop surpris par la syntaxe que nous allons voir. Notre constructeur est en effet une méthode, elle en garde la syntaxe. Nous allons donc écrire notre méthode écrire pour commencer.

# Les méthodes, la recette 10/

```
class TableauNoir:  
    """Classe définissant une surface sur laquelle on peut écrire,  
    que l'on peut lire et effacer, par jeu de méthodes. L'attribut modifié  
    est 'surface'''  
  
    def __init__(self):  
        """Par défaut, notre surface est vide'''  
        self.surface = ''  
  
    def ecrire(self, message_a_ecrire):  
        """Méthode permettant d'écrire sur la surface du tableau.  
        Si la surface n'est pas vide, on saute une ligne avant de rajouter  
        le message à écrire'''  
  
        if self.surface != "":  
            self.surface += "\n"  
        self.surface += message_a_ecrire
```

```
tab = TableauNoir()  
print(tab.surface)  
  
tab.ecrire("Cooooool ! Ce sont les vacances !")  
print(tab.surface)  
  
tab.ecrire("Joyeux Noël !")  
print(tab.surface)  
>>>  
>>> Cooooool ! Ce sont les vacances !  
>>> Cooooool ! Ce sont les vacances !  
>>> Joyeux Noël !
```

# Les méthodes, la recette 11/

Notre méthode ecrire se charge d'écrire sur notre surface, en rajoutant un saut de ligne pour séparer chaque message.

On retrouve ici notre paramètre self. Il est temps de voir un peu plus en détail à quoi il sert.

## Le paramètre self

Dans nos méthodes d'instance, qu'on appelle également des méthodes d'objet, on trouve dans la définition ce paramètre self. L'heure est venue de comprendre ce qu'il signifie.

Une chose qui a son importance : quand vous créez un nouvel objet, ici un tableau noir, les attributs de l'objet sont propres à l'objet créé. C'est logique : si vous créez plusieurs tableaux noirs, ils ne vont pas tous avoir la même surface. Donc les attributs sont contenus dans l'objet.

En revanche, les méthodes sont contenues dans la classe qui définit notre objet. C'est très important. Quand vous tapez tab.ecrire(...), Python va chercher la méthode ecrire non pas dans l'objet tab, mais dans la classe TableauNoir.

```
tab.ecrire
>>> <bound method TableauNoir.ecrire of <__main__.TableauNoir object at 0x00B3F3F0>>
TableauNoir.ecrire
>>> <function ecrire at 0x00BA5810>
help(TableauNoir.ecrire)
>>> Help on function ecrire in module __main__:
ecrire(self, message_a_ecrire)

    Méthode permettant d'écrire sur la surface du tableau.
    Si la surface n'est pas vide, on saute une ligne avant de rajouter
    le message à écrire.
TableauNoir.ecrire(tab, "essai")
tab.surface
>>> 'essai'
```

# Les méthodes, la recette 12/

Comme vous le voyez, quand vous tapez `tab.ecrire(...)`, cela revient au même que si vous écrivez `TableauNoir.ecrire(tab, ...)`. Votre paramètre `self`, c'est l'objet qui appelle la méthode. C'est pour cette raison que vous modifiez la surface de l'objet en appelant `self.surface`.

Pour résumer, quand vous devez travailler dans une méthode de l'objet sur l'objet lui-même, vous allez passer par `self`.

Le nom `self` est une très forte convention de nommage. Je vous déconseille de changer ce nom. Certains programmeurs, qui trouvent qu'écrire `self` à chaque fois est excessivement long, l'abrègent en une unique lettres. Évitez ce raccourci. De manière générale, évitez de changer le nom. Une méthode d'instance travaille avec le paramètre `self`.

N'est-ce pas effectivement plutôt long de devoir toujours travailler avec `self` à chaque fois qu'on souhaite faire appel à l'objet ?

Cela peut le sembler, oui. C'est d'ailleurs l'un des reproches qu'on fait au langage Python. Certains langages travaillent implicitement sur les attributs et méthodes d'un objet sans avoir besoin de les appeler spécifiquement. Mais c'est moins clair et cela peut susciter la confusion. En Python, dès qu'on voit `self`, on sait que c'est un attribut ou une méthode interne à l'objet qui va être appelé.

Bon, voyons nos autres méthodes. Nous devons encore coder `lire` qui va se charger d'afficher notre surface et `effacer` qui va effacer le contenu de notre surface. Si vous avez compris ce que je viens d'expliquer, vous devriez écrire ces méthodes sans aucun problème, elles sont très simples. Sinon, n'hésitez pas à relire, jusqu'à ce que le déclic se fasse.

# Les méthodes, la recette 13/

```
class TableauNoir:  
    """Classe définissant une surface sur laquelle on peut écrire,  
    que l'on peut lire et effacer, par jeu de méthodes. L'attribut modifié  
    est 'surface'''  
  
    def __init__(self):  
        """Par défaut, notre surface est vide'''  
        self.surface = ""  
  
    def écrire(self, message_a_ecrire):  
        """Méthode permettant d'écrire sur la surface du tableau. Si la surface n'est pas vide, on saute une ligne avant de rajouter  
        le message à écrire'''  
  
        if self.surface != "":  
            self.surface += "\n"  
        self.surface += message_a_ecrire  
  
    def lire(self):  
        """Cette méthode se charge d'afficher, grâce à print,  
        la surface du tableau'''  
        print(self.surface)  
  
    def effacer(self):  
        """Cette méthode permet d'effacer la surface du tableau'''  
        self.surface = ""
```

# Les méthodes, recette 14/

```
tab = TableauNoir()
tab.lire()
tab.ecrire("Salut tout le monde.")
tab.ecrire("La forme ?")
tab.lire()
>>> Salut tout le monde.
>>> La forme ?
tab.effacer()
tab.lire()
```

## Méthodes de classe et méthodes statiques

Comme on trouve des attributs propres à la classe, on trouve aussi des méthodes de classe, qui ne travaillent pas sur l'instance `self` mais sur la classe même. C'est un peu plus rare mais cela peut être utile parfois. Notre méthode de classe se définit exactement comme une méthode d'instance, à la différence qu'elle ne prend pas en premier paramètre `self`(l'instance de l'objet) mais `cls` (la classe de l'objet). En outre, on utilise ensuite une fonction built-in de Python pour lui faire comprendre qu'il s'agit d'une méthode de classe, pas d'une méthode d'instance.

```
class Compteur:
    """Cette classe possède un attribut de classe qui s'incrémente à chaque
    fois que l'on crée un objet de ce type"""

    objets_crees = 0 # Le compteur vaut 0 au départ

    def __init__(self):
        """À chaque fois qu'on crée un objet, on incrémente le compteur"""
        Compteur.objets_crees += 1

    def combien(cls):
        """Méthode de classe affichant combien d'objets ont été créés"""
        print("Jusqu'à présent, {} objets ont été créés.".format(
            cls.objets_crees))

    combien = classmethod(combien)
```

# Les méthodes, la recette 15/

Voyons d'abord le résultat :

```
Compteur.combien()
>>> Jusqu'à présent, 0 objets ont été créés.
a = Compteur()
Compteur.combien()
>>> Jusqu'à présent, 1 objets ont été créés.
b = Compteur()
Compteur.combien()
>>> Jusqu'à présent, 2 objets ont été créés.
```

Une méthode de classe prend en premier paramètre non pas self mais cls. Ce paramètre contient la classe (ici Compteur).

Notez que vous pouvez appeler la méthode de classe depuis un objet instancié sur la classe. Vous auriez par exemple pu écrire a.combien().

Enfin, pour que Python reconnaissse une méthode de classe, il faut appeler la fonction class method qui prend en paramètre la méthode que l'on veut convertir et renvoie la méthode convertie.

Si vous êtes un peu perdus, retenez la syntaxe de l'exemple. La plupart du temps, vous définirez des méthodes d'instance comme nous l'avons vu plutôt que des méthodes de classe.

On peut également définir des méthodes statiques. Elles sont assez proches des méthodes de classe sauf qu'elles ne prennent aucun premier paramètre, ni self ni cls. Elles travaillent donc indépendamment de toute donnée, aussi bien contenue dans l'instance de l'objet que dans la classe.

Voici la syntaxe permettant de créer une méthode statique. Je ne veux pas vous surcharger d'informations et je vous laisse faire vos propres tests si cela vous intéresse :

```
class Test:
    """Une classe de test tout simplement"""
    def afficher():
        """Fonction chargée d'afficher quelque chose"""
        print("On affiche la même chose.")
        print("peu importe les données de l'objet ou de la classe.")
    afficher = staticmethod(afficher)
```

# Les méthodes, la recette 16/

Si vous vous emmêlez un peu avec les attributs et méthodes de classe, ce n'est pas bien grave. Retenez surtout les attributs et méthodes d'instance, c'est essentiellement sur ceux-ci que je me suis attardé et c'est ceux que vous retrouverez la plupart du temps.

Rappel : les noms de méthodes encadrés par deux soulignés de part et d'autre sont des méthodes spéciales. Ne nommez pas vos méthodes ainsi. Nous découvrirons plus tard ces méthodes particulières. Exemple de nom de méthode à éviter : \_mamethode.

# La méthode setattr() 17/

```
class Agent:  
  
    def __init__(self, agent_attributes):  
        self.agreeableness = agent_attributes['agreeableness']  
  
    agent_attributes = {"neuroticism": -0.0739192627121713, "language": "Shona", "latitude": -19.922097800281783,  
                        "country_tld": "zw", "age": 12, "income": 333, "longitude": 29.798455535838603, "sex": "Male",  
                        "religion": "syncretic", "extraversion": 1.051833688742943, "date_of_birth": "2005-01-10",  
                        "agreeableness": 0.1441229877537559, "id_str": "LB3-3CI", "conscientiousness": 0.2419104411765549,  
                        "internet": "false", "country_name": "Zimbabwe", "openness": -0.024607605122172617,  
                        "id": 6636726630}  
  
    first_agent = Agent(agent_attributes)  
    print(first_agent.agreeableness)  
  
    def __init__(self, agent_attributes):  
        for attr_name, attr_value in agent_attributes.items():  
            setattr(self, attr_name, attr_value)
```

La méthode setattr() est équivalente au code suivant:

```
my_object.attribute = value
```

Il s'agit simplement d'une autre notation.

Dans notre cas, utiliser cette méthode est bien plus utile car nous nous situons à l'intérieur d'une boucle (nous souhaitons utiliser les variables attr\_name et attr\_value).

# Un peu d'introspection 18/

Encore de la philosophie ?

Eh bien... le terme d'introspection, je le reconnaiss, fait penser à quelque chose de plutôt abstrait. Pourtant, vous allez très vite comprendre l'idée qui se cache derrière : Python propose plusieurs techniques pour explorer un objet, connaître ses méthodes ou attributs.

Quel est l'intérêt ? Quand on développe une classe, on sait généralement ce qu'il y a dedans, non ?

En effet. L'utilité, à notre niveau, ne saute pas encore aux yeux. Et c'est pour cela que je ne vais pas trop m'attarder dessus. Si vous ne voyez pas l'intérêt, contentez-vous de garder dans un coin de votre tête les deux techniques que nous allons voir. Arrivera un jour où vous en aurez besoin ! Pour l'heure donc, voyons plutôt l'effet :

## La fonction dir

La première technique d'introspection que nous allons voir est la fonction dir. Elle prend en paramètre un objet et renvoie la liste de ses attributs et méthodes.

```
maclass Test:  
    """Une classe de test tout simplement"""  
    def afficher():  
        """Fonction chargée d'afficher quelque chose"""  
        print("On affiche la même chose.")  
        print("peu importe les données de l'objet ou de la classe.")  
    afficher = staticmethod(afficher)
```

```
# Créons un objet de la classe Test  
un_test = Test()  
un_test.afficher_attribut()  
>>> Mon attribut est ok.  
dir(un_test)  
>>> ['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__',  
     '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',  
     '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',  
     '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'afficher_attribut', 'mon_attribut']
```

# Un peu d'introspection 19/

La fonction `dir` renvoie une liste comprenant le nom des attributs et méthodes de l'objet qu'on lui passe en paramètre. Vous pouvez remarquer que tout est mélangé, c'est normal : pour Python, les méthodes, les fonctions, les classes, les modules sont des objets. Ce qui différencie en premier lieu une variable d'une fonction, c'est qu'une fonction est exécutable (callable). La fonction `dir` se contente de renvoyer tout ce qu'il y a dans l'objet, sans distinction.

Euh, c'est quoi tout cela ? On n'a jamais défini toutes ces méthodes ou attributs !

Non, en effet. Nous verrons plus loin qu'il s'agit de méthodes spéciales utiles à Python.

L'attribut spécial `__dict__`

Par défaut, quand vous développez une classe, tous les objets construits depuis cette classe posséderont un attribut spécial `__dict__`. Cet attribut est un dictionnaire qui contient en guise de clés les noms des attributs et, en tant que valeurs, les valeurs des attributs.

```
class Test:  
    """Une classe de test tout simplement"""  
    def afficher():  
        """Fonction chargée d'afficher quelque chose"""  
        print("On affiche la même chose.")  
        print("peu importe les données de l'objet ou de la classe.")  
    afficher = staticmethod(afficher)
```

```
un_test = Test()  
un_test.afficher_attribut()  
>>> Mon attribut est ok.  
dir(un_test)  
>>>['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__g  
e__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',  
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__  
setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'affich  
er_attribut', 'mon_attribut']
```

# Un peu d'introspection 20/

Voyez plutôt :

```
# Créons un objet de la classe Test  
un_test = Test()  
un_test.__dict__  
>>> {'mon_attribut': 'ok'}
```

Pourquoi « attribut spécial » ?

C'est un attribut un peu particulier car ce n'est pas vous qui le créez, c'est Python. Il est entouré de deux signes soulignés\_\_ de part et d'autre, ce qui traduit qu'il a une signification pour Python et n'est pas un attribut « standard ». Vous verrez plus loin dans ce cours des méthodes spéciales qui reprennent la même syntaxe.

Peut-on modifier ce dictionnaire ?

Vous le pouvez. Sachez qu'en modifiant la valeur de l'attribut, vous modifiez aussi l'attribut dans l'objet.

```
un_test.__dict__["mon_attribut"] = "plus ok"  
un_test.afficher_attribut()  
>>> Mon attribut est plus ok.
```

De manière générale, ne faites appel à l'introspection que si vous avez une bonne raison de le faire et évitez ce genre de syntaxe. Il est quand même plus propre d'écrire `objet.attribut = valeur` que `objet.__dict__[nom_attribut] = valeur`.

Nous n'irons pas plus loin dans ce chapitre. Je pense que vous découvrirez dans la suite de ce livre l'utilité des deux méthodes que je vous ai montrées.

# Classmethod et Staticmethod

```
Class ClassTest:  
    def instance_method(self):  
        print(f"Called instance_method of {self}")  
  
    @classmethod  
    def class_method(cls):  
        print(f"Called class_method of {cls}") # take the class as parameter  
  
    @staticmethod  
    def static_method():  
        print("Called static_method.") # This method doesn't have a parameter cls or self and have @staticmethod on top. This methods don't get anything when you call them  
  
ClassTest.class_method() # python will pass the class (ClassTest) as the argument to the parameter cls. So you don't need an instance or object you can directly call the class.  
ClassTest.static_method() # Here python will just called that method  
  
>>> Called class_method of <class '__main__.ClassTest'>  
>>> Called static_method.
```

**Instance methods** are used for most things when you want to produce an action that uses the data inside the object that you created earlier on for example that is when instance methods will get used. Also if you want to call a method to modify some sort of data inside self or the object you will also use instance method.

**Class methods** are used often as factories

**Static methods** are used to just place a method inside a class

# Classmethod and Staticmethod

There are 3 types of methods:

1. One which take an instance as first argument
2. A second one which take a class as argument
3. A third one which take nothing as argument

Class method are used when you want something to have access to the class

"""  
We've looked at how we can define classes and methods, including some special methods like `\_\_init\_\_` and `\_\_len\_\_`.

All these methods had something in common: the `self` parameter at the start. As a reminder, here's some code:  
"""

```
class Student:  
    def __init__(self, name, school):  
        self.name = name  
        self.school = school  
        self.marks = []  
  
    def average(self):  
        return sum(self.marks) / len(self.marks)  
"""
```

When we create a new object from the `Student` class and we call a method, we are automatically passing in the `self` parameter:  
"""

```
rolf = Student('Rolf', 'MIT')
```

```
rolf.marks.append(78)  
rolf.marks.append(99)
```

```
print(rolf.average())
```

"""  
This is identical to that last line:  
"""

```
print(Student.average(rolf))
```

"""  
When we do `object.method()`, Python is in the background calling

`Class.method(object)` , so that `self` is always the object that called the method.

Indeed, if we were to have two objects:  
"""

```
rolf = Student('Rolf', 'MIT')  
anne = Student('Anne', 'Cambridge')
```

```
rolf.marks.append(78)  
rolf.marks.append(99)
```

```
anne.marks.append(34)  
anne.marks.append(56)
```

```
print(rolf.average())  
print(anne.average())
```

"""  
In the first case, `self` would be the `rolf` object, and in the second case `self` would be the `anne` object.

Notice that this knowledge now lets us do some very weird stuff (not recommended, as it'll likely break things):  
"""

```
Student.average('hello') # self is now 'hello', comment this out to run the rest of the file.
```

"""  
Just remember `self` is a parameter like any other; and you can give it any value you want. However, because the method is then accessing `hello.marks` , you'll get an error for the string doesn't have that property.

Anyway, so why is this important?

The first type of method we've looked at is called "instance method": one that takes the caller object as the first argument (that's `self` ).

There are two more types of method:

- \* One that takes the caller's class as the first argument; and
- \* One that takes nothing as the first argument.

## @classmethod  
"""

Let's look at the one that takes the caller's class as the first argument.  
"""

```
class Foo:  
    @classmethod  
    def hi(cls):  
        print(cls.__name__)
```

```
my_object = Foo()  
my_object.hi() # prints Foo
```

## @staticmethod  
"""

Now one that takes nothing as the first argument.  
"""

```
class Foo:  
    @staticmethod  
    def hi():  
        print("I don't take parameters.!")
```

```
my_object = Foo()  
my_object.hi()
```

"""  
Those are some terrible examples! Let's look at some more in the next section.  
"""

# Classmethod et Staticmethod examples:

```
class Book:  
    TYPES = ("hardcover", "paperback")  
  
    def __init__(self, name, book_type, weight):  
        self.name = name  
        self.book_type = book_type  
        self.weight = weight  
  
    def __repr__():  
        return f"<Book {self.name}, {self.book_type}, weighing {self.weight}g>"  
  
@classmethod  
def hardcover(cls, name, page_weight):  
    return Book(name, cls.TYPES[0], page_weight + 100) # rappel cls means class and here it can be replaced by Book.  
  
@classmethod  
def paperback(cls, name, page_weight):  
    return cls(name, cls.TYPES[1], page_weight)  
  
book = Book.hardcover("Harry Potter", 1500)  
light = Book.paperback("Python 101", 600)  
  
print(book)  
print(light)  
->>> <Book Harry Potter, hardcover, weighing 1500g>  
->>> <Book Python 101, paperback, weighing 600g>
```

# Classmethod et Staticmethod example:

"""

Those were some terrible examples in the last section, but I just wanted to show you the syntax for these two types of method.

The `@...` is called a decorator. Those are important in Python, and we'll be looking at creating our own later on. They are used to modify the function directly below them.

```
class FixedFloat:  
    def __init__(self, amount):  
        self.amount = amount  
  
    def __repr__(self):  
        return f'<FixedFloat {self.amount:.2f}>'  
  
number = FixedFloat(18.5746)  
print(number) # 18.57
```

"""  
We have this `FixedFloat` class that is really basic—doesn't really do anything other than print the number out to two decimal places with the class name.

Imagine we wanted to get a new `FixedFloat` object which is a result of summing two numbers together:

```
class FixedFloat:  
    def __init__(self, amount):  
        self.amount = amount  
  
    def __repr__(self):  
        return f'<FixedFloat {self.amount:.2f}>'  
  
    def from_sum(self, value1, value2):  
        return FixedFloat(value1 + value2)  
  
number = FixedFloat(18.5746)  
new_number = number.from_sum(19.575, 0.789)  
print(new_number)
```

"""  
This doesn't make any sense, because we created a `FixedFloat` object ('number'), and then proceeded to call an instance method to create a new object. But that instance method didn't use 'self' at all—so really the fact that it's a method inside a class is not very useful.

Instead, we could make it a `@staticmethod`. That way, we're not getting 'self' but we can still put the method in the class, since it is \_related\_ to the class:

```
class FixedFloat:  
    def __init__(self, amount):  
        self.amount = amount  
  
    def __repr__(self):  
        return f'<FixedFloat {self.amount:.2f}>'  
  
    @staticmethod  
    def from_sum(value1, value2):  
        return FixedFloat(value1 + value2)  
  
static_number = FixedFloat.from_sum(19.575, 0.789)  
print(static_number)
```

"""  
That looks a bit better! Now we don't have the useless parameter AND we don't need to create an object before we can call the method. Win-win!

However, let's now include some inheritance. We'll create a `Currency` class that extends this `Float` class.

```
class Euro(FixedFloat):  
    def __init__(self, amount):  
        super().__init__(amount)  
        self.symbol = '€'  
  
    def __repr__(self):  
        return f'<Euro {self.symbol}{self.amount:.2f}>'  
  
    # Skip defining from_sum as that's inherited
```

"""  
We've defined this new class that extends the `FixedFloat` class. It's got an `\_\_init\_\_` method that calls the parent's `\_\_init\_\_`, and a `\_\_repr\_\_` method that overrides the parents'. It doesn't have a `from\_sum` method as that's inherited and we'll just use the one the parent defined.

```
euros = Euro(18.5963)  
print(euros) # <Euro €18.59>  
  
result = Euro.from_sum(15.76, 19.905)  
print(result) # <FixedFloat 35.66>
```

"""  
Oops! When we called the `Euro` constructor directly, we got a `Euro` object with the symbol. But when we call `from\_sum`, we got a `FixedFloat` object. Not what we wanted!

In order to fix this, we must make the `from\_sum` method return an object of the class that called it—so that:

- \* `FixedFloat.from\_sum()` returns a `FixedFloat` object; and
- \* `Euro.from\_sum()` returns an `Euro` object.

`@classmethod` to the rescue! If we modify the `FixedFloat` class:

```
class FixedFloat:  
    def __init__(self, amount):  
        self.amount = amount  
  
    def __repr__(self):  
        return f'<FixedFloat {self.amount:.2f}>'  
  
    @classmethod  
    def from_sum(cls, value1, value2):  
        return cls(value1 + value2)
```

```
class Euro(FixedFloat):  
    def __init__(self, amount):  
        super().__init__(amount)  
        self.symbol = '€'
```

```
def __repr__(self):  
    return f'<Euro {self.symbol}{self.amount:.2f}>'
```

"""  
When we now call:

- \* `Euro.from\_sum()`, `cls` is the `Euro` class.
- \* `FixedFloat.from\_sum()`, `cls` is the `FixedFloat` class.

```
print(Euro.from_sum(16.7565, 90)) # <Euro €106.75>
```

# Classmethod vs Staticmethod

```
# @classmethod vs @staticmethod vs "plain" methods
# What's the difference?
```

```
class MyClass:
    def method(self):
        """
        Instance methods need a class instance and
        can access the instance through `self`.
        """
        return 'instance method called', self
```

```
@classmethod
def classmethod(cls):
    """
    Class methods don't need a class instance.
    They can't access the instance (self) but
    they have access to the class itself via `cls`.
    """
    return 'class method called', cls
```

```
@staticmethod
def staticmethod():
    """
    Static methods don't have access to `cls` or `self`.
    They work like regular functions but belong to
    the class's namespace.
    """
    return 'static method called'
```

```
# All methods types can be
# called on a class instance:
obj = MyClass()
obj.method()
>>> ('instance method called', <MyClass instance at 0x1019381b8>)
obj.classmethod()
```

```
>>> ('class method called', <class MyClass at 0x101a2f4c8>)
obj.staticmethod()
>>> 'static method called'

# Calling instance methods fails
# if we only have the class object:
MyClass.classmethod()
>>> ('class method called', <class MyClass at 0x101a2f4c8>)
MyClass.staticmethod()
>>> 'static method called'
MyClass.method()
TypeError:
    "unbound method method() must be called with MyClass "
    "instance as first argument (got nothing instead)"
```

# Résumé

## En résumé

On définit une classe en suivant la syntaxe CamelCase (première lettre de chaque mot en majuscule) class NomClasse:.

Les méthodes se définissent comme des fonctions, sauf qu'elles se trouvent dans le corps de la classe.

Les méthodes d'instance prennent en premier paramètre self, l'instance de l'objet manipulé.

On construit une instance de classe en appelant son constructeur, une méthode d'instance appelée \_\_init\_\_.

On définit les attributs d'une instance dans le constructeur de sa classe, en suivant cette syntaxe : self.nom\_attribut = valeur.

# L'objet en python

# Vocabulaire

Classe

: le moule de l'objet à créer (ex : animal)

Objet

: instance de classe (ex : chat)

Attribut

: variable de classe (ex : prénom, animal, couleur du poile)

Propriété

: manier de manipuler les attributs (lecture seule, accès non autorisé en dehors de la classe, etc.)

Méthode d'instance

: fonction d'une classe (ex : manger, marcher, chasser, dormir, mourir)

Méthode de classe

: fonction d'une classe (explication à venir...)

Method statique

: fonction d'une classe, mais indépendante de celle-ci.

Héritage

: classe chat qui hérite de la classe animal (Chat est une sorte d'animal)

# OOP object oriented programming 1/10

- Object Oriented Programming (OOP) allows programmers to create their own objects that have methods and attributes.
- Recall that after defining a string, list, dictionary, or other objects, you were able to call methods off of them with the `.method_name()` syntax.

# OOP object oriented programming 2/10

These methods act as functions that use information about the object, as well as the object itself to return results, or change the current object.

For example this includes appending to a list, or counting the occurrences of an element in a tuple.

# OOP object oriented programming 3/10

OOP allows users to create their own objects.

The general format is often confusing when first encountered, and its usefulness may not be completely clear at first.

In general, OOP allows us to create code that is repeatable and organized.

# OOP object oriented programming 4/10

For much larger scripts of Python code, functions by themselves aren't enough for organization and repeatability.

Commonly repeated tasks and objects can be defined with OOP to create code that is more usable.

Let's check out the syntax.

# OOP object oriented programming 5/10

```
class NameOfClass():

    def __init__(self,param1,param2):
        self.param1 = param1
        self.param2 = param2

    def some_method(self):
        # perform some action
        print(self.param1)
```

# OOP object oriented programming 6/10

Complete Python Bootcamp

```
class NameOfClass():
```

```
    def __init__(self,param1,param2):  
        self.param1 = param1  
        self.param2 = param2
```

```
    def some_method(self):  
        # perform some action  
        print(self.param1)
```

# OOP object oriented programming 7/10

Complete Python Bootcamp

```
class NameOfClass():

    def __init__(self,param1,param2):
        self.param1 = param1
        self.param2 = param2
```

```
    def some_method(self):
        # perform some action
        print(self.param1)
```

# OOP object oriented programming 8/10

```
class NameOfClass():

    def __init__(self,param1,param2):
        self.param1 = param1
        self.param2 = param2

    def some_method(self):
        # perform some action
        print(self.param1)

DATA
```

# OOP object oriented programming 9/10

Complete Python Bootcamp

```
class NameOfClass():

    def __init__(self,param1,param2):
        self.param1 = param1
        self.param2 = param2

    def some_method(self):
        # perform some action
        print(self.param1)
```

# OOP object oriented programming

## 10/10

Complete Python Bootcamp

```
class NameOfClass():

    def __init__(self,param1,param2):
        self.param1 = param1
        self.param2 = param2
```

```
def some_method(self):
```

```
    # perform some action
    print(self.param1)
```

# Le paramètre self

Ces 2 lignes sont identiques

```
class Personnage:  
    def dit(self, message):  
        print(message)  
  
patrick = Personnage()  
patrick.dit('Bonjour')  
Personnage.dit(patrick, 'Bonjour')
```

Quand on utilise la méthode dit sur l'instance patrick, ce que l'on ne voit pas c'est que en arrière plan python utilise la classe personnage (comme sur la dernière ligne ci-dessus)

# Magic Methods 1/2

"""\nIn a class, not all methods are the same. Python sometimes makes a distinction depending on the method name. Here's one of these special methods:\n"""

```
class Student:\n    def __init__(self, name):\n        self.name = name
```

"""\nThis method is different from other methods because it gets called automatically for you when you create a new object. For example:\n"""

```
my_student = Student('Jose')
```

"""\nWhat happens here is that a new object is created, and then the `\_\_init\_\_` method is called with the new object as `self` and the string you passed as `name`.\n"""

```
## Other interesting special methods\n### 'len()'
```

"""\nGiven an \*iterable\* (generally a list, tuple, set, or dictionary; something you can iterate over), `len()` gives you the number of elements. For example:\n"""

```
movies = ['Matrix', 'Finding Nemo']
```

```
print(movies.__class__) # what's this?
```

```
count = len(movies)\nprint(count) # 2
```

"""\nWe can make `len()` work on our classes too, by adding the `\_\_len\_\_` method:\n"""

```
class Garage:\n    def __init__(self):\n        self.cars = []\n\ndef __len__(self):\n    return len(self.cars)
```

```
ford_garage = Garage()\nford_garage.cars.append('Fiesta')\nford_garage.cars.append('Focus')\n\nprint(len(ford_garage))\n\n### Getting a specific item (square bracket notation)
```

"""\nWe can also use square bracket notation in our 'Garage':\n"""

```
class Garage:\n    def __init__(self):\n        self.cars = []\n\ndef __len__(self):\n    return len(self.cars)
```

```
def __getitem__(self, i):\n    return self.cars[i]
```

```
ford_garage = Garage()\nford_garage.cars.append("Fiesta")\nford_garage.cars.append("Focus")
```

```
print(ford_garage[1]) # Focus
```

"""\nA great thing about this is now you can iterate over the garage using a for loop. To do this you need both `\_\_len\_\_` and `\_\_getitem\_\_`:

```
for car in ford_garage:\n    print(car)
```

### String representation

"""\nIf you want to print your objects out (and sometimes during development it can be handy, as we'll see), we can use `\_\_repr\_\_` and `\_\_str\_\_`:

\* `\_\_repr\_\_` should be used to print out a string representing the object such that with that string you can re-create the object fully.  
\* `\_\_str\_\_` should be used when printing the object out to a user, for example—can be more descriptive or even miss out some details.

```
class Garage:\n    def __init__(self):\n        self.cars = []
```

```
def __repr__(self):\n    return f'Garage {self.cars}'
```

```
def __str__(self):\n    return f'Garage with {len(self.cars)} cars'
```

"""\nYou should implement at least `\_\_repr\_\_`.

In order to call these methods, you can:

```
garage = Garage()\ngarage.cars.append('Fiesta')\ngarage.cars.append('Focus')
```

```
print(garage)\nprint(str(garage))\nprint(repr(garage))
```

## More

"""\nThere are many magic "dunder" methods you can implement, including some to overload what mathematical operators do, what boolean operators do, make your objects callable, adding context managers, and more.

"""\nWe'll be learning about all this throughout the course!

# Magic Methods 2/2

<https://blog.teclado.com/creating-a-new-sequence-type-in-python-part-1/>



Creating a New Sequence Type in Python - Part 1.pdf

<https://blog.teclado.com/creating-a-new-sequence-type-in-python-part-2/>



Creating a New Sequence Type in Python - Part 2.pdf

# We have a class called Club, and it is initialized like this (no need to change):

```
class Club:  
    def __init__(self, name):  
        self.name = name  
        self.players = []  
  
    # optional  
    def __len__(self):  
        return len(self.players)  
  
    # define a method that allows us to access the i-th player in the club directly via indexing.  
    # for example, if some_club is an object of Club class,  
    # we can access the i-th player in some_club like this (you may assume i is always valid):  
    # some_club[i]  
    def __getitem__(self, i):  
        return self.players[i]  
  
    # define a method that returns a string representation of this object,  
    # which can be used to recreate this object.  
    # The return value should be in such format (beware of the spacing):  
    # Club {club_name}: {list_of_players}  
    # the club_name and list_of_players should be replaced by the according value of current object  
    def __repr__(self):  
        return f"Club {self.name}: {self.players}"  
  
    # define a method that returns a readable string representation of this object for the user.  
    # The return value should be in such format (beware of the spacing):  
    # Club {club_name} with {count_of_players} players  
    # the club_name and count_of_players should be replaced by the according value of current object  
    def __str__(self):  
        return f"Club {self.name} with {len(self)} players"
```

# You only need to finish the method, we will take care of the object creation and call those methods for you!

```
my_club = Club('Arsenal')  
my_club.players.append('Rolf')  
my_club.players.append('Anne')  
print(my_club[1])  
print(my_club.__repr__())  
print(my_club.__str__())
```

# The @property decorator

```
class Student:  
    def __init__(self, name, school):  
        self.name = name  
        self.school = school  
        self.marks = []  
  
    def average(self):  
        return sum(self.marks) / len(self.marks)
```

```
anna = Student("Anna", "Oxford")
```

"""\nImagine you've got a class like the above, and you want to create a similar\nclass with some extra functionality. For example, a student that not only has\nmarks but also a salary—a `WorkingStudent`:\n"""

```
class WorkingStudent:  
    def __init__(self, name, school, salary):  
        self.name = name  
        self.school = school  
        self.marks = []  
        self.salary = salary  
  
    def average(self):  
        return sum(self.marks) / len(self.marks)
```

```
rolf = WorkingStudent("Rolf", "MIT", 15.50)
```

"""\nHowever you can see there's a lot of duplication between our `Student` and\n`WorkingStudent` classes. Instead, we may choose to make our

'WorkingStudent' extend the 'Student'. It keeps all the same functionality,\nbut we can add more.

"""

```
class WorkingStudent(Student):  
    def __init__(self, name, school, salary):  
        super().__init__(name, school)  
        self.salary = salary
```

```
rolf = WorkingStudent("Rolf", "MIT", 15.50)  
rolf.marks.append(57)  
rolf.marks.append(99)  
print(rolf.average())
```

"""

By the way, notice how the `average()` function doesn't take any inputs\nother than `self`. There's nothing in the brackets.

In those cases, and if you think it makes sense, we can make it into a\nproperty, just like `marks` and `salary`.

All we have to do is:

"""

```
class Student:  
    def __init__(self, name, school):  
        self.name = name  
        self.school = school  
        self.marks = []
```

```
@property  
def average(self):  
    return sum(self.marks) / len(self.marks)
```

"""

Now the `average()` function can be used as if it were a property instead of\na method; like so:

"""

```
jose = Student("Jose", "Stanford")  
jose.marks.append(80)  
jose.marks.append(90)  
print(jose.average)
```

"""

You can do that with any method that doesn't take any arguments. But\nremember, this method only returns a value calculated from the object's\nproperties. If you have a method that does things (e.g. save to a database or\ninteract with other things), it can be better to stay with the brackets.

Normally:

- \* Brackets: this method does things, performs actions.
- \* No brackets: this is a value (or a value calculated from existing values, in\nthe case of `@property`).

"""

# OOP object oriented programming – Attributes and Class Keyword 1/2

```
# class SampleWord(): we use camel case every first letter is capitalized
# let's see an exemple

class Dog:
    # init is the constructor for a class
    def __init__(self, mybreed):
        # self connect this method to the instance of the class. Self represents the instance of the object itself
        # Attributes
        # We take in the argument
        # Assign it using self.attribute_name
        self.mybreed = mybreed

my_dog = Dog(mybreed='Lab')
print(type(my_dog))
>>> __main__.Dog

print(my_dog.my_attribute)
>>> 'Lab'
```

However by convention what is wrote above must be written as below

```
class Dog():
    # init is the constructor for a class
    def __init__(self, breed):
        # self connect this methode to the instance of the class. Self represents the instance of the object itself
        # Attributes
        # We take in the argument
        # Assign it using self.attribute_name
        self.breed = breed

my_dog = Dog( Lab)
```

# OOP object oriented programming – Attributes and Class Keyword 2/2

```
class Dog:  
    # init is the constructor for a class  
    def __init__(self, breed, name, spots): # self connect this methode to the instance of the class. Self represents the instance of the object itself  
        # Attributes  
        # We take in the argument  
        # Assign it using self.attribute_name  
        self.breed = breed  
        self.name = name  
        # expect boolean True/False  
        self.spots = spots  
  
my_dog = Dog(breed='Lab', name='Sammy', spots=False)  
print(type(my_dog))  
print(my_dog.breed)  
print(my_dog.name)  
print(my_dog.spots)  
  
>>> <class '__main__.Dog'>  
>>> Lab  
>>> Sammy  
>>> False
```

# OOP object oriented programming – Object Attributes and Methods 1/2

```
class Dog():
    # CLASS OBJECT ATTRIBUTE
    # SAME FOR ANY INSTANCE OF A CLASS
    species = 'mammal'

    # init is the constructor for a class
    def __init__(self, breed, name,
                 spots): # self connect this methode to the instance of the class. Self represents the instance of the object itself
        # Attributes
        # We take in the argument
        # Assign it using self.attribute_name
        self.breed = breed
        self.name = name
        self.spots = spots # expect boolean True/False

    # OPERATIONS/Actions --> Methods
    def bark(self, number):
        print("WOOF! My name is {} and the number is {}".format(self.name,
                                                               number)) # ce n'est pas self.number car number est fournie en paramètre de la methode par l'utilisateur

my_dog = Dog(breed='Lab', name='Frankie', spots=False)
print(type(my_dog))
print(my_dog.species)
print(my_dog.name)
my_dog.bark(10)
>>> <class '__main__.Dog'>
>>> mammal
>>> Frankie
>>> WOOF! My name is Frankie and the number is 10
```

# OOP object oriented programming – Object Attributes and Methods 2/2

```
class Circle():
    # CLASS OBJECT ATTRIBUTE
    pi = 3.14

    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * self.pi # with an object attribute like pi you can reference it either by self.pi or by Circle.pi

    # Method
    def get_circumference(self):
        return self.radius * Circle.pi * 2

my_circle = Circle()

print(my_circle.pi)
print(my_circle.radius)

my_circle = Circle(30)
print(my_circle.radius)
print(my_circle.area)

print(my_circle.get_circumference())

>>> 3.14
>>> 1
>>> 30
>>> 2826.0
>>> 188.4
```

# Héritage

# L'héritage simple 1/11

## Pour bien commencer

L'héritage est une fonctionnalité objet qui permet de déclarer que telle classe sera elle-même modelée sur une autre classe, qu'on appelle la classe parente, ou la **classe mère**. Concrètement, si une classe B **hérite** de la classe A, les objets créés sur le modèle de la classe B auront accès aux méthodes et attributs de la classe A.

Et c'est tout ? Cela ne sert à rien !

La première chose, c'est que la classe B dans notre exemple ne se contente pas de reprendre les méthodes et attributs de la classe A : elle va pouvoir en définir d'autres. D'autres méthodes et d'autres attributs qui lui seront propres, en plus des méthodes et attributs de la classe A. Et elle va pouvoir également redéfinir les méthodes de la classe mère.

Prenons un exemple simple : on a une classe Animal permettant de définir des animaux. Les animaux tels que nous les modélisons ont certains attributs (le régime : carnivore ou herbivore) et certaines méthodes (manger, boire, crier...).

On peut maintenant définir une classe Chien qui hérite de Animal, c'est-à-dire qu'elle reprend ses méthodes. Nous allons voir plus bas ce que cela implique exactement.

Si vous ne voyez pas très bien dans quel cas on fait hériter une classe d'une autre, faites le test :

- on fait hériter la classe Chien de Animal parce qu'un chien est un animal ;
- on ne fait pas hériter Animal de Chien parce qu'Animal n'est pas un Chien.

# L'héritage simple 2/11

Sur ce modèle, vous pouvez vous rendre compte qu'une voiture est un véhicule. La classe Voiture pourrait donc hériter de Véhicule.

Intéressons-nous à présent au code.

On oppose l'héritage simple, dont nous venons de voir les aspects théoriques dans la section précédente, à l'héritage multiple que nous verrons dans la prochaine section.

Il est temps d'aborder la syntaxe de l'héritage. Nous allons définir une première classe A et une seconde classe B qui hérite de A.

**class A:**

```
"""Classe A, pour illustrer notre exemple d'héritage"""
pass # On laisse la définition vide, ce n'est qu'un exemple
```

**class B(A):**

```
"""Classe B, qui hérite de A.
Elle reprend les mêmes méthodes et attributs (dans cet exemple, la classe
A ne possède de toute façon ni méthode ni attribut)"""
pass
```

Vous pourrez expérimenter par la suite sur des exemples plus constructifs. Pour l'instant, l'important est de bien noter la syntaxe qui, comme vous le voyez, est des plus simples : **class**

**MaClasse(MaClasseMere):** Dans la définition de la classe, entre le nom et les deux points, vous précisez entre parenthèses la classe dont elle doit hériter. Comme je l'ai dit, dans un premier temps, toutes les méthodes de la classe A se retrouveront dans la classe B.

J'ai essayé de mettre des constructeurs dans les deux classes mais, dans la classe fille, je ne retrouve pas les attributs déclarés dans ma classe mère, c'est normal ?

Tout à fait. Vous vous souvenez quand je vous ai dit que les méthodes étaient définies dans la classe, alors que les attributs étaient directement déclarés dans l'instance d'objet ? Vous le voyez bien de toute façon : c'est dans le constructeur qu'on déclare les attributs et on les écrit tous dans l'instance self. Quand une classe B hérite d'une classe A, les objets de type B reprennent bel et bien les méthodes de la classe A en même temps que celles de la classe B. Mais, assez logiquement, ce sont celles de la classe B qui sont appelées d'abord.

Si vous faites `objet_de_type_b.ma_methode()`, Python va d'abord chercher la méthode `ma_methode` dans la classe B dont l'objet est directement issu. S'il ne trouve pas, il va chercher récursivement dans les classes dont hérite B, c'est-à-dire A dans notre exemple. Ce mécanisme est très important : il induit que si aucune méthode n'a été redéfinie dans la classe, on cherche dans la classe mère. On peut ainsi redéfinir une certaine méthode dans une classe et laisser d'autres directement hériter de la classe mère.

# L'héritage simple 3/11

Petit code d'exemple :

```
class Personne:  
    """Classe représentant une personne"""  
  
    def __init__(self, nom):  
        """Constructeur de notre classe"""""  
        self.nom = nom  
        self.prenom = "Martin"  
  
    def __str__(self):  
        """Méthode appelée lors d'une conversion de l'objet en chaîne"""""  
        return "{0} {1}".format(self.prenom, self.nom)  
  
class AgentSpecial(Personne):  
    """Classe définissant un agent spécial.  
    Elle hérite de la classe Personne"""""  
  
    def __init__(self, nom, matricule):  
        """Un agent se définit par son nom et son matricule"""""  
        self.nom = nom  
        self.matricule = matricule  
  
    def __str__(self):  
        """Méthode appelée lors d'une conversion de l'objet en chaîne"""""  
        return "Agent {0}, matricule {1}".format(self.nom, self.matricule)
```

# L'héritage simple 4/11

Vous voyez ici un exemple d'héritage simple. Seulement, si vous essayez de créer des agents spéciaux, vous risquez d'avoir de drôles de surprises :

```
agent = AgentSpecial("Fisher", "18327-121")
agent.nom
print(agent)
print(agent.prenom)
>>> Agent Fisher, matricule 18327-121
>>> Traceback (most recent call last):
      File "C:\Users\MOTTIER LUCIE\Documents\GitHub\TheCompletePythonCourse\test.py", line 77, in <module>
        print(agent.prenom)
    AttributeError: 'AgentSpecial' object has no attribute 'prenom'
```

Argh... mais tu n'avais pas dit qu'une classe reprenait les méthodes et attributs de sa classe mère ?

Si. Mais en suivant bien l'exécution, vous allez comprendre : tout commence à la création de l'objet. Quel constructeur appeler ? S'il n'y avait pas de constructeur défini dans notre classe AgentSpecial, Python appelleraient celui de Personne. Mais il en existe bel et bien un dans la classe AgentSpecial et c'est donc celui-ci qui est appelé. Dans ce constructeur, on définit deux attributs, nom et matricule. Mais c'est tout : le constructeur de la classe Personne n'est pas appelé, sauf si vous lappelez explicitement dans le constructeur d'AgentSpecial.

Dans le premier chapitre, je vous ai expliqué que `mon_objet.ma_methode()` revenait au même que `MaClasse.ma_methode(mon_objet)`. Dans notre méthode `ma_methode`, le premier paramètre `self` sera `mon_objet`. Nous allons nous servir de cette équivalence. La plupart du temps, écrire `mon_objet.ma_methode()` suffit. Mais dans une relation d'héritage, il peut y avoir, comme nous l'avons vu, plusieurs méthodes du même nom définies dans différentes classes. Laquelle appeler ? Python choisit, s'il la trouve, celle définie directement dans la classe dont est issu l'objet, et sinon parcourt la hiérarchie de l'héritage jusqu'à tomber sur la méthode. Mais on peut aussi se servir de la notation `MaClasse.ma_methode(mon_objet)` pour appeler une méthode précise d'une classe précise. Et cela est utile dans notre cas :

# L'héritage simple 5/11

```
class Personne:  
    """Classe représentant une personne"""  
  
    def __init__(self, nom):  
        """Constructeur de notre classe"""  
        self.nom = nom  
        self.prenom = "Martin"  
  
    def __str__(self):  
        """Méthode appelée lors d'une conversion de l'objet en chaîne"""  
        return "{0} {1}".format(self.prenom, self.nom)  
  
  
class AgentSpecial(Personne):  
    """Classe définissant un agent spécial.  
    Elle hérite de la classe Personne"""  
  
    def __init__(self, nom, matricule):  
        """Un agent se définit par son nom et son matricule"""  
        # On appelle explicitement le constructeur de Personne :  
        Personne.__init__(self, nom)  
        self.matricule = matricule  
  
    def __str__(self):  
        """Méthode appelée lors d'une conversion de l'objet en chaîne"""  
        return "Agent {0}, matricule {1}".format(self.nom, self.matricule)
```

Si cela vous paraît encore un peu vague, expérimentez : c'est toujours le meilleur moyen. Entraînez-vous, contrôlez l'écriture des attributs, ou revenez au premier chapitre de cette partie pour vous rafraîchir la mémoire au sujet du paramètre `self`, bien qu'à force de manipulations vous avez dû comprendre l'idée.

# L'héritage simple 6/11

Reprenez notre code de tout à l'heure qui, cette fois, passe sans problème :

```
agent = AgentSpecial("Fisher", "18327-121")
print(agent.nom)
print(agent)
print(agent.prenom)

>>> Fisher
>>> Agent Fisher, matricule 18327-121
>>> Martin
```

Cette fois, notre attribut prenom se trouve bien dans notre agent spécial car le constructeur de la classe AgentSpecial appelle explicitement celui de Personne.

Vous pouvez noter également que, dans le constructeur d'AgentSpecial, on n'instancie pas l'attribut nom. Celui-ci est en effet écrit par le constructeur de la classe Personne que nous appelons en lui passant en paramètre le nom de notre agent.

Notez que l'on pourrait très bien faire hériter une nouvelle classe de notre classe Personne, la classe mère est souvent un modèle pour plusieurs classes filles.

## Petite précision

Dans le chapitre précédent, je suis passé très rapidement sur l'héritage, ne voulant pas trop m'y attarder et brouiller les cartes inutilement. Mais j'ai expliqué brièvement que toutes les classes que vous créez héritent de la classe object. C'est elle, notamment, qui définit toutes les méthodes spéciales que nous avons vues au chapitre précédent et qui connaît, bien mieux que nous, le mécanisme interne de l'objet. Vous devriez un peu mieux, à présent, comprendre le code du chapitre précédent. Le voici, en substance :

```
def __setattr__(self, nom_attribut, valeur_attribut):
    """Méthode appelée quand on fait objet.attribut = valeur"""
    print("Attention, on modifie l'attribut {0} de l'objet !".format(nom_attribut))
    object.__setattr__(self, nom_attribut, valeur_attribut)
```

# L'héritage simple 7/11

En redéfinissant la méthode `__setattr__`, on ne peut, dans le corps de cette méthode, modifier les valeurs de nos attributs comme on le fait habituellement (`self.attribut = valeur`) car alors, la méthode s'appellerait elle-même. On fait donc appel à la méthode `__setattr__` de la classe `object`, cette classe dont héritent implicitement toutes nos classes. On est sûr que la méthode de cette classe sait écrire une valeur dans un attribut, alors que nous ignorons le mécanisme et que nous n'avons pas besoin de le connaître : c'est la magie du procédé, une fois qu'on a bien compris le principe !  
Deux fonctions très pratiques

Python définit deux fonctions qui peuvent se révéler utiles dans bien des cas : `issubclass` et `isinstance`.

`issubclass`

Comme son nom l'indique, elle vérifie si une classe est une sous-classe d'une autre classe. Elle renvoie `True` si c'est le cas, `False` sinon :

```
print(issubclass(AgentSpecial, Personne)) # AgentSpecial hérite de Personne
print(issubclass(AgentSpecial, object))
print(issubclass(Personne, object))
print(issubclass(Personne, AgentSpecial)) # Personne n'hérite pas d'AgentSpecial
>>> True
>>> True
>>> True
>>> False
```

`isinstance`

`isinstance` permet de savoir si un objet est issu d'une classe ou de ses classes filles :

```
>>> agent = AgentSpecial("Fisher", "18327-121")
>>> isinstance(agent, AgentSpecial) # Agent est une instance d'AgentSpecial
True
>>> isinstance(agent, Personne) # Agent est une instance héritée de Personne
True
```

Ces quelques exemples suffisent, je pense. Peut-être devrez-vous attendre un peu avant de trouver une utilité à ces deux fonctions mais ce moment viendra.

# L'héritage 8/11

Class mère

```
2 class Utilisateur:
3     def __init__(self, nom, prenom):
4         self.nom = nom
5         self.prenom = prenom
6
7     def __str__(self):
8         return f"Utilisateur {self.nom} {self.prenom}"
9
10    def afficher_projets(self):
11        for projet in projets:
12            print(projet)
13
14 class Junior(Utilisateur):
15     def __init__(self, nom, prenom):
16         Utilisateur.__init__(self, nom, prenom)
17
18 paul = Junior("Paul", "Durand")
19 paul.afficher_projets()
```

Class fille

La classe fille peut appeler les méthodes de la classe mère. Ci-dessus: Paul est une instance de la classe Junior (classe fille de la classe Utilisateur) et appelle la méthode afficher\_projets() définie dans la classe mère

# L'héritage 9/11

## la méthode super()

```
2 class Utilisateur:
3     def __init__(self, nom, prenom):
4         self.nom = nom
5         self.prenom = prenom
6
7     def __str__(self):
8         return f"Utilisateur {self.nom} {self.prenom}"
9
10    def afficher_projets(self):
11        for projet in projets:
12            print(projet)
13
14 class Junior(Utilisateur):
15     def __init__(self, nom, prenom):
16         super().__init__(nom, prenom)
17
18 paul = Junior("Paul", "Durand")
19 paul.afficher_projets()
20
```

Super() permet d'appeler les méthodes de la classe parente sans avoir besoin d'indiquer son nom. Comparez le code de la slide précédente et de celle-ci.

!!! IMPORTANT !!!  
La fonction super ne prend pas **self** en paramètre, on ne passe donc que les paramètres noms et prénoms dans cette exemple.

# L'héritage 10/11

## La surcharge

```
projets = ["pr_GameOfThrones", "HarryPotter", "pr_Avengers"]
class Utilisateur:
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def __str__(self):
        return f"Utilisateur {self.nom} {self.prenom}"

    def afficher_projets(self):
        for projet in projets:
            print(projet)

class Junior(Utilisateur):
    def __init__(self, nom, prenom):
        super().__init__(nom, prenom)
```

```
14 class Junior(Utilisateur):
15     def __init__(self, nom, prenom):
16         super().__init__(nom, prenom)
17
18     def afficher_projets(self):
19         for projet in projets:
20             if not projet.startswith("pr_"):
21                 print(projet)
22
23 paul = Junior("Paul", "Durand")
24 paul.afficher_projets()
```

Ici la **surcharge** est faite dans la classe enfant (Junior) où l'on a redéfini la méthode `afficher_projets()` de façon à ne pas afficher les projets qui commencent par `pr_`. Python exécute la méthode la plus proche donc si on écrit `paul = Junior("Paul", "Durand")` on appellera `afficher_projets` de la **class Junior**. Mais on peut toujours écrire `super().afficher_projets`, qui appellera `afficher_projets` de la **class utilisateur**.

# L'héritage 11/11

## La surcharge

The diagram illustrates a class hierarchy in Python. On the left, three categories are listed: 'Classe mère' (Mother Class) pointing to the `Student` class, 'Classe fille' (Daughter Class) pointing to the `WorkingStudent` class, and 'Surcharge' (Overriding) pointing to the overridden methods in the `WorkingStudent` class.

```
class Student:
    def __init__(self, name, school):
        self.name = name
        self.school = school
        self.marks = []

    def average(self):
        return sum(self.marks) / len(self.marks)

class WorkingStudent(Student):
    def __init__(self, name, school, salary):
        super().__init__(name, school)
        self.salary = salary

    def weekly_salary(self):
        return self.salary * 37.5

rolf = WorkingStudent('Rolf', 'MIT', 15.50)
print(rolf.salary)
rolf.marks.append(57)
rolf.marks.append(99)
print(rolf.average())
print(rolf.weekly_salary())
```

# L'héritage multiple 1/4

Python inclut un mécanisme permettant l'héritage multiple. L'idée est en substance très simple : au lieu d'hériter d'une seule classe, on peut hériter de plusieurs.

Ce n'est pas ce qui se passe quand on hérrite d'une classe qui hérrite elle-même d'une autre classe ?

Pas tout à fait. La hiérarchie de l'héritage simple permet d'étendre des méthodes et attributs d'une classe à plusieurs autres, mais la structure reste fermée. Pour mieux comprendre, considérez l'exemple qui suit.

On peut s'asseoir dans un fauteuil. On peut dormir dans un lit. Mais on peut s'asseoir et dormir dans certains canapés (la plupart en fait, avec un peu de bonne volonté). Notre classe Fauteuil pourra hériter de la classe ObjetPourASseoir et notre classe Lit, de notre classe ObjetPourDormir. Mais notre classe Canape alors ? Elle devra logiquement hériter de nos deux classes ObjetPourASseoir et ObjetPourDormir. C'est un cas où l'héritage multiple pourrait se révéler utile.

Assez souvent, on utilisera l'héritage multiple pour des classes qui ont besoin de certaines fonctionnalités définies dans une classe mère. Par exemple, une classe peut produire des objets destinés à être enregistrés dans des fichiers. On peut faire hériter de cette classe toutes celles qui produiront des objets à enregistrer dans des fichiers. Mais ces mêmes classes pourront hériter d'autres classes incluant, pourquoi pas, d'autres fonctionnalités.

C'est une des utilisations de l'héritage multiple et il en existe d'autres. Bien souvent, l'utilisation de cette fonctionnalité ne vous semblera évidente qu'en vous penchant sur la hiérarchie d'héritage de votre programme. Pour l'instant, je vais me contenter de vous donner la syntaxe et un peu de théorie supplémentaire, en vous encourageant à essayer par vous-mêmes :

```
class MaClasseHeritee(MaClasseMere1, MaClasseMere2):
```

Vous pouvez faire hériter votre classe de plus de deux autres classes. Au lieu de préciser, comme dans les cas d'héritage simple, une seule classe mère entre parenthèses, vous en indiquez plusieurs, séparées par des virgules.

Recherche des méthodes

La recherche des méthodes se fait dans l'ordre de la définition de la classe. Dans l'exemple ci-dessus, si on appelle une méthode d'un objet issu de MaClasseHeritee, on va d'abord chercher dans la classe MaClasseHeritee. Si la méthode n'est pas trouvée, on la cherche d'abord dans MaClasseMere1. Encore une fois, si la méthode n'est pas trouvée, on cherche dans toutes les classes mères de la classe MaClasseMere1, si elle en a, et selon le même système. Si, encore et toujours, on ne trouve pas la méthode, on la recherche dans MaClasseMere2 et ses classes mères successives.

C'est donc l'ordre de définition des classes mères qui importe. On va chercher la méthode dans les classes mères de gauche à droite. Si on ne trouve pas la méthode dans une classe mère donnée, on remonte dans ses classes mères, et ainsi de suite.

# L'héritage multiple 2/4

## Retour sur les exceptions

Depuis la première partie, nous ne sommes pas revenus sur les exceptions. Toutefois, ce chapitre me donne une opportunité d'aller un peu plus loin.

Les exceptions sont non seulement des classes, mais des classes hiérarchisées selon une relation d'héritage précise.

Cette relation d'héritage devient importante quand vous utilisez le mot-clé `except`. En effet, le type de l'exception que vous précisez après est intercepté... ainsi que toutes les classes qui héritent de ce type.

Mais comment fait-on pour savoir qu'une exception hérite d'autres exceptions ?

Il y a plusieurs possibilités. Si vous vous intéressez à une exception en particulier, consultez l'aide qui lui est liée.

```
Help on class AttributeError in module builtins:
```

```
class AttributeError(Exception)
| Attribute not found.
|
| Method resolution order:
|     AttributeError
|     Exception
|     BaseException
|     object
```

Vous apprenez ici que l'exception `AttributeError` hérite de `Exception`, qui hérite elle-même de `BaseException`.

Vous pouvez également retrouver la hiérarchie des exceptions built-in sur le site de Python.

Ne sont répertoriées ici que les exceptions dites built-in. D'autres peuvent être définies dans des modules que vous utiliserez et vous pouvez même en créer vous-mêmes (nous allons voir cela un peu plus bas).

Pour l'instant, souvenez-vous que, quand vous écrivez `except TypeException`, vous pourrez intercepter toutes les exceptions du type `TypeException` mais aussi celles des classes héritées de `TypeException`.

La plupart des exceptions sont levées pour signaler une erreur... mais pas toutes. L'exception `KeyboardInterrupt` est levée quand vous interrompez votre programme, par exemple avec `CTRL + C`. Si bien que, quand on souhaite intercepter toutes les erreurs potentielles, on évitera d'écrire un simple `except:` et on le remplacera par `except Exception:` toutes les exceptions « d'erreurs » étant dérivées de `Exception`.

# L'héritage multiple 3/4

## Création d'exceptions personnalisées

Il peut vous être utile de créer vos propres exceptions. Puisque les exceptions sont des classes, comme nous venons de le voir, rien ne vous empêche de créer les vôtres. Vous pourrez les lever avec `raise`, les intercepeter avec `except`.

Se positionner dans la hiérarchie

Vos exceptions doivent hériter d'une exception built-in proposée par Python. Commencez par parcourir la hiérarchie des exceptions built-in pour voir si votre exception peut être dérivée d'une exception qui lui serait proche. La plupart du temps, vous devrez choisir entre ces deux exceptions :

`BaseException` : la classe mère de toutes les exceptions. La plupart du temps, si vous faites hériter votre classe de `BaseException`, ce sera pour modéliser une exception qui ne sera pas fondamentalement une erreur, par exemple une interruption dans le traitement de votre programme.

`Exception` : c'est de cette classe que vos exceptions hériteront la plupart du temps. C'est la classe mère de toutes les exceptions « d'erreurs ».

Si vous pouvez trouver, dans le contexte, une exception qui se trouve plus bas dans la hiérarchie, c'est toujours mieux.

Que doit contenir notre classe exception ?

Deux choses : un constructeur et une méthode `__str__` car, au moment où l'exception est levée, elle doit être affichée. Souvent, votre constructeur ne prend en paramètre que le message d'erreur et la méthode `__str__` renvoie ce message :

```
class MonException(Exception):
    """Exception levée dans un certain contexte... qui reste à définir"""

    def __init__(self, message):
        """On se contente de stocker le message d'erreur"""
        self.message = message

    def __str__(self):
        """On renvoie le message"""
        return self.message
```

Cette exception s'utilise le plus simplement du monde :

```
raise MonException("OUPS... j'ai tout cassé")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MonException: OUPS... j'ai tout cassé
```

# L'héritage multiple 4/4

Mais vos exceptions peuvent aussi prendre plusieurs paramètres à l'instanciation :

```
class ErreurAnalyseFichier(Exception):
    """Cette exception est levée quand un fichier (de configuration)
    n'a pas pu être analysé.

    Attributs :
        fichier -- le nom du fichier posant problème
        ligne -- le numéro de la ligne posant problème
        message -- le problème proprement dit"""

    def __init__(self, fichier, ligne, message):
        """Constructeur de notre exception"""
        self.fichier = fichier
        self.ligne = ligne
        self.message = message

    def __str__(self):
        """Affichage de l'exception"""
        return "[{}:{}]: {}".format(self.fichier, self.ligne, self.message)
```

Et pour lever cette exception :

```
raise ErreurAnalyseFichier("plop.conf", 34,
    "Il manque une parenthèse à la fin de l'expression")
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.ErreurAnalyseFichier: [plop.conf:34]: il manque une parenthèse à la fin de l'expression
```

Voilà, ce petit retour sur les exceptions est achevé. Si vous voulez en savoir plus, n'hésitez pas à consulter la documentation Python concernant [les exceptions](#) ainsi que celle sur [les exceptions personnalisées](#).

# En Résumé

## En résumé

L'héritage permet à une classe d'hériter du comportement d'une autre en reprenant ses méthodes.

La syntaxe de l'héritage est `class NouvelleClasse(ClasseMere):`.

On peut accéder aux méthodes de la classe mère directement via la syntaxe : `ClasseMere.methode(self)`.

L'héritage multiple permet à une classe d'hériter de plusieurs classes mères.

La syntaxe de l'héritage multiple s'écrit donc de la manière suivante : `class NouvelleClasse(ClasseMere1, ClasseMere2, ClasseMereN):`.

Les exceptions définies par Python sont ordonnées selon une hiérarchie d'héritage.

# Polymorphisme Inheritance Composition

# Polymorphisme 1/5

On a une classe mère Véhicule

On a deux classes filles Voiture et Avion.

Chaque classe possède une méthode avance.

Dans les classes filles lorsque l'on écrit super().avance, on effectue le print(" Le véhicule démarre") de la classe mère, puis le print de la classe fille. On a donc enrichie la classe mère avec la classe fille.

```
1 class Vehicule:
2     def avance(self):
3         print("Le véhicule démarre")
4
5 class Voiture(Vehicule):
6     def avance(self):
7         super().avance()
8         print("La voiture roule")
9
10 class Avion(Vehicule):
11     def avance(self):
12         super().avance()
13         print("L'avion vol")
14
15 v = Voiture()
16 a = Avion()
17 v.avance()
18 a.avance()
```

# OOP object oriented programming – inheritance and Polymorphism 2/5

```
class Animal():
    def __init__(self):
        print("Animal CREATED")

    def who_am_i(self):
        print("I am an animal")

    def eat(self):
        print("I am eating")

class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog Created")

    # overwritting a method from parent class
    def who_am_i(self):
        print("I am a dog!")

    def eat(self):
        print("I am a dog and eating")

    # add methods on parent class
    def bark(self):
        print("WOOF !")

mydog = Dog()
print(mydog.eat())
print(mydog.who_am_i())
print(mydog.bark)
```

>>> Animal CREATED  
>>> Dog Created  
>>> I am eating  
>>> I am a dog!  
>>> WOOF!

# OOP object oriented programming – inheritance and Polymorphisme 3/5

Polymorphism refers the way in which different object classes can share the same method name

```
class Dog():
    def __init__(self, name):
        self.name = name

    # overwriting a method from parent class
    def speak(self):
        return self.name + " Says woof!"

class Cat():
    def __init__(self, name):
        self.name = name

    # overwriting a method from parent class
    def speak(self):
        return self.name + " Says meow!"

niko = Dog("nico")
felix = Cat("felix")

print(niko.speak())
print(felix.speak())

for pet in [niko, felix]:
    print(type(pet))
    print(type(pet.speak()))
```

```
>>> nico Says woof!
>>> felix Says meow!
>>> <class '__main__.Dog'>
>>> <class 'str'>
>>> <class '__main__.Cat'>
>>> <class 'str'>
```

This is an example of polymorphism both niko and felix share the same method name called speak. However they are different types here, we have main.dog and main.cat

# OOP object oriented programming – inheritance and Polymorphism 4/5

```
class Animal():
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclass must implement this abstract method")

myanimal = Animal('Fred')
myanimal.speak()

in speak raise NotImplementedError(NotImplementedError: Subclass must implement this abstract method)
```

THE WAY IT IS DESIGNED TO WORK IS AS FOLLOW:



```
class Animal():
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclass must implement this abstract method")

class Dog(Animal):
    def speak(self):
        return self.name + " says woof!"

class Cat(Animal):
    def speak(self):
        return self.name + " says meow!"

fido = Dog("Fido")
isis = Cat("Isis")
print(fido.speak())
print(isis.speak())

>>> Fido says woof!
>>> Isis says meow!
```

# OOP object oriented programming – inheritance and Polymorphism 5/5

```
class Device:  
    def __init__(self, name, connected_by):  
        self.name = name  
        self.connected_by = connected_by  
        self.connected = True  
  
    def __str__(self):  
        return f"Device {self.name}!r {self.connected_by}"  
  
    def disconnect(self):  
        self.connected = False  
        print("Disconnected.")  
  
class Printer(Device):  
    def __init__(self, name, connected_by, capacity):  
        super().__init__(name, connected_by)  
        self.capacity = capacity  
        self.remaining_pages = capacity  
  
    def __str__(self):  
        return f"{super().__str__()} {self.remaining_pages} pages remaining"  
  
    def print(self, pages):  
        if not self.connected:  
            print("Your printer is not connected!")  
            return  
        print(f"Printing {pages} pages.")  
        self.remaining_pages -= pages  
  
printer = Printer("Printer", "USB", 500)  
printer.print(20)  
print(printer)  
printer.disconnect() # the method disconnect is not in the class printer, so Python will go in the parent class to see if there is  
# a disconnect method.  
printer.print(30)
```

>>> Printing 20 pages.  
>>> Device 'Printer' (USB) (480 pages remaining)  
>>> Disconnected.  
>>> Your printer is not connected!

# OOP object oriented programming – class composition 5/5

```
class BookShelf:  
    def __init__(self, *book):  
        self.books = book  
  
    def __str__(self):  
        return f"BookShel with {len(self.books)} books."  
  
class Book:  
    def __init__(self, name):  
        self.name = name  
  
    def __str__(self):  
        return f"Book{self.name}"  
  
book = Book("Harry Potter")  
book2 = Book("Python 101")  
shelf = BookShelf(book, book2)  
  
print(shelf)  
  
>>> BookShel with 2 books.
```

# How to Write Cleaner Python Code using Abstract Classes

# How to Write Cleaner Python Code using Abstract Classes

<https://blog.teclado.com/python-abc-abstract-base-classes/>



How to Write Cleaner Python Code Using Abstract Classes.pdf

Un objet est-il  
callable ?

# Un objet est-il callable ?

```
import os
from pprint import pprint # ici c'est la function pprint qui se trouve dans le module pprint

print(callable(os.name))
>> False # donc os.name n'est pas callable, on ne pas ecrire os.name()

# A l'inverse
print(callable(pprint))
>> True car la function pprint est callable

# Attention: la function pprint est callable mais pas le module:
Import pprint #ici c'est le module pprint

print(callable(pprint))
>> False # En effet on ne peut pas appeler le module
```

# Définissez des propriétés

# Définissez des propriétés

## Définissez des propriétés

Au chapitre précédent, nous avons appris à créer nos premiers attributs et méthodes. Mais nous avons encore assez peu parlé de la philosophie objet. Il existe quelques confusions que je vais tâcher de lever.

Nous allons découvrir dans ce chapitre les propriétés, un concept propre à Python et à quelques autres langages, comme le Ruby. C'est une fonctionnalité qui, à elle seule, change l'approche objet et le principe d'encapsulation.

Qu'est-ce que l'encapsulation ?

L'encapsulation est un principe qui consiste à cacher ou protéger certaines données de notre objet. Dans la plupart des langages orientés objet, tels que le C++, le Java ou le PHP, on va considérer que nos attributs d'objets ne doivent pas être accessibles depuis l'extérieur de la classe. Autrement dit, vous n'avez pas le droit de faire, depuis l'extérieur de la classe, `mon_objet.mon_attribut`.

Mais c'est stupide ! Comment fait-on pour accéder aux attributs ?

On va définir des méthodes un peu particulières, appelées des accesseurs et mutateurs. Les accesseurs donnent accès à l'attribut. Les mutateurs permettent de le modifier. Concrètement, au lieu d'écrire `mon_objet.mon_attribut`, vous allez écrire `mon_objet.get_mon_attribut()`. De la même manière, pour modifier l'attribut Écrivez `mon_objet.set_mon_attribut(valeur)` et non pas `mon_objet.mon_attribut = valeur`.

`get` signifie « récupérer », c'est le préfixe généralement utilisé pour un accesseur.

`set` signifie, dans ce contexte, « modifier » ; c'est le préfixe usuel pour un mutateur.

# Qu'est-ce que l'encapsulation

Ah mais d'abord, je n'ai pas dit que vous ne pouviez pas. Vous pouvez très bien accéder aux attributs d'un objet directement, comme on l'a fait au chapitre précédent. Je ne fais ici que résumer le principe d'encapsulation tel qu'on peut le trouver dans d'autres langages. En Python, c'est un peu plus subtil.

Mais pour répondre à la question, il peut être très pratique de sécuriser certaines données de notre objet, par exemple faire en sorte qu'un attribut de notre objet ne soit pas modifiable, ou alors mettre à jour un attribut dès qu'un autre attribut est modifié. Les cas sont multiples et c'est très utile de pouvoir contrôler l'accès en lecture ou en écriture sur certains attributs de notre objet.

L'inconvénient de devoir écrire des accesseurs et mutateurs, comme vous l'aurez sans doute compris, c'est qu'il faut créer deux méthodes pour chaque attribut de notre classe. D'abord, c'est assez lourd. Ensuite, nos méthodes se ressemblent plutôt. Certains environnements de développement proposent, il est vrai, de créer ces accesseurs et mutateurs pour nous, automatiquement. Mais cela ne résout pas vraiment le problème, vous en conviendrez.

**Python a une philosophie un peu différente : pour tous les objets dont on n'attend pas une action particulière, on va y accéder directement, comme nous l'avons fait au chapitre précédent. On peut y accéder et les modifier en écrivant simplement `mon_objet.mon_attribut`. Et pour certains, on va créer des propriétés.**

# Les propriétés à la casserole

Pour commencer, une petite précision : en C++ ou en Java par exemple, dans la définition de classe, on met en place des principes d'accès qui indiquent si l'attribut (ou le groupe d'attributs) est privé ou public. Pour schématiser, si l'attribut est public, on peut y accéder depuis l'extérieur de la classe et le modifier. S'il est privé, on ne peut pas. On doit passer par des accesseurs ou mutateurs.

En Python, il n'y a pas d'attribut privé. Tout est public. Cela signifie que si vous voulez modifier un attribut depuis l'extérieur de la classe, vous le pouvez. Pour faire respecter l'encapsulation propre au langage, on la fonde sur des conventions que nous allons découvrir un peu plus bas mais surtout sur le bon sens de l'utilisateur de notre classe (à savoir, si j'ai écrit que cet attribut est inaccessible depuis l'extérieur de la classe, je ne vais pas chercher à y accéder depuis l'extérieur de la classe).

Les propriétés sont un moyen transparent de manipuler des attributs d'objet. Elles permettent de dire à Python : « Quand un utilisateur souhaite modifier cet attribut, fais cela ». De cette façon, on peut rendre certains attributs tout à fait inaccessibles depuis l'extérieur de la classe, ou dire qu'un attribut ne sera visible qu'en lecture et non modifiable. Ou encore, on peut faire en sorte que, si on modifie un attribut, Python recalcule la valeur d'un autre attribut de l'objet.

Pour l'utilisateur, c'est absolument transparent : il croit avoir, dans tous les cas, un accès direct à l'attribut. C'est dans la définition de la classe que vous allez préciser que tel ou tel attribut doit être accessible ou modifiable grâce à certaines propriétés.

Mais ces propriétés, c'est quoi ?

Hum... eh bien je pense que pour le comprendre, il vaut mieux les voir en action. Les propriétés sont des objets un peu particuliers de Python. Elles prennent la place d'un attribut et agissent différemment en fonction du contexte dans lequel elles sont appelées. Si on les appelle pour modifier l'attribut, par exemple, elles vont rediriger vers une méthode que nous avons créée, qui gère le cas où « on souhaite modifier l'attribut ». Mais trêve de théorie.

# Les propriétés en action

Une propriété ne se crée pas dans le constructeur mais dans le corps de la classe. J'ai dit qu'il s'agissait d'une classe, son nom est `property`. Elle attend quatre paramètres, tous optionnels :

- la méthode donnant accès à l'attribut ;
- la méthode modifiant l'attribut ;
- la méthode appelée quand on souhaite supprimer l'attribut ;
- la méthode appelée quand on demande de l'aide sur l'attribut.

En pratique, on utilise surtout les deux premiers paramètres : ceux définissant les méthodes d'accès et de modification, autrement dit nos accesseur et mutateur d'objet.

# Les propriétés en action

<http://sammy76.free.fr/conseils/informatique/python.html>

```
class Personne:  
    """Classe dénissant une personne caractérisée par : son nom ; son prénom ; son âge ; son lieu de résidence"""  
  
    def __init__(self, nom, prenom):  
        """Constructeur de notre classe"""  
        self.nom = nom  
        self.prenom = prenom  
        self.age = 33  
        self._lieu_residence = "Paris" # Notez le souligné _ devant le nom  
  
    def __repr__(self):  
        """Quand on entre notre objet dans l'interpréteur"""  
        return "{} {}, âgé de {} ans".format(self.prenom, self.nom, self.age)  
  
    def __del__(self):  
        """Méthode appelée quand l'objet est supprimé"""  
        print("C'est la fin ! On me supprime !")  
  
    def _get_lieu_residence(self):  
        """Méthode qui sera appellée quand on souhaitera accéder en lecture à l'attribut 'lieu_residence'"""  
        print("On accéde à l'attribut lieu_residence !")  
        return self._lieu_residence  
  
    def _set_lieu_residence(self, nouvelle_residence):  
        """Méthode appellée quand on souhaite modifier le lieu de résidence"""  
        print("Attention, il semble que {} déménage à {}".format(self.prenom, nouvelle_residence))  
        self._lieu_residence = nouvelle_residence  
  
# On va dire à Python que notre attribut lieu_residence pointe vers une propriété  
lieu_residence = property(_get_lieu_residence, _set_lieu_residence)
```

# Les propriétés en action

Vous devriez (j'espère) reconnaître la syntaxe générale de la classe. En revanche, au niveau du lieu de résidence, les choses changent un peu :

Tout d'abord, dans le constructeur, on ne crée pas un attribut `self.lieu_residence` mais `self._lieu_residence`. Il n'y a qu'un petit caractère de différence, le signe souligné `_` placé en tête du nom de l'attribut. Et pourtant, ce signe change beaucoup de choses. La convention veut qu'on n'accède pas, depuis l'extérieur de la classe, à un attribut commençant par un souligné `_`. C'est une convention, rien ne vous l'interdit... sauf, encore une fois, le bon sens.

On définit une première méthode, commençant elle aussi par un souligné `_`, nommée `_get_lieu_residence`. C'est la même règle que pour les attributs : on n'accède pas, depuis l'extérieur de la classe, à une méthode commençant par un souligné `_`. Si vous avez compris ma petite explication sur les accesseurs et mutateurs, vous devriez comprendre rapidement à quoi sert cette méthode : elle se contente de renvoyer le lieu de résidence. Là encore, l'attribut manipulé n'est pas `lieu_residence` mais `_lieu_residence`. Comme on est dans la classe, on a le droit de le manipuler.

La seconde méthode a la forme d'un mutateur. Elle se nomme `_set_lieu_residence` et doit donc aussi être inaccessible depuis l'extérieur de la classe. À la différence de l'accesseur, elle prend un paramètre : le nouveau lieu de résidence. En effet, c'est une méthode qui doit être appelée quand on cherche à modifier le lieu de résidence, il lui faut donc le nouveau lieu de résidence qu'on souhaite voir affecté à l'objet.

Enfin, la dernière ligne de la classe est très intéressante. Il s'agit de la définition d'une propriété. On lui dit que l'attribut `lieu_residence` (cette fois, sans signe souligné `_`) doit être une propriété. On définit dans notre propriété, dans l'ordre, la méthode d'accès (l'accesseur) et celle de modification (le mutateur).

Quand on veut accéder à `objet.lieu_residence`, Python tombe sur une propriété redirigeant vers la méthode `_get_lieu_residence`. Quand on souhaite modifier la valeur de l'attribut, en écrivant `objet.lieu_residence = valeur`, Python appelle la méthode `_set_lieu_residence` en lui passant en paramètre la nouvelle valeur.

# Les propriétés en action

Ce n'est pas clair ? Voyez cet exemple :

```
jean = Personne("Micado", "Jean")
print(jean.nom)
print(jean.prenom)
print(jean.age)
print(jean.lieu_residence)
jean.lieu_residence = "Berlin"
print(jean.lieu_residence)

>>> Micado
>>> Jean
>>> 33
>>> On accéde à l'attribut lieu_residence !
>>> Paris
>>> Attention, il semble que Jean déménage à Berlin.
>>> On accéde à l'attribut lieu_residence !
>>> Berlin
>>> C'est la fin ! On me supprime !
```

Notre accesseur et notre mutateur se contentent d'afficher un message, pour bien qu'on se rende compte que ce sont eux qui sont appelés quand on souhaite manipuler l'attribut lieu\_residence. Vous pouvez aussi ne définir qu'un accesseur, dans ce cas l'attribut ne pourra pas être modifié.

Il est aussi possible de définir, en troisième position du constructeur property, une méthode qui sera appelée quand on fera del objet.lieu\_residence et, en quatrième position, une méthode qui sera appelée quand on fera help(objet.lieu\_residence). Ces deux dernières fonctionnalités sont un peu moins utilisées mais elles existent.

Voilà, vous connaissez à présent la syntaxe pour créer des propriétés. Entraînez-vous, ce n'est pas toujours évident au début. C'est un concept très puissant, il serait dommage de passer à côté.

# Les propriétés en action

Autre exemple

```
class Product:  
    def __init__(self, price):  
        self.price = price  
  
    @property  
    def price(self):  
        return self.__price  
  
    @price.setter  
    def price(self, value):  
        if value < 0:  
            raise ValueError("- Price cannot be negative.")  
        else:  
            self.__price = value  
  
  
product = Product(10)  
product.price = -1  
print(product.price)  
# A properti is an object that seats in front of an attribute and allow us to set or get the value of an attribut  
  
Traceback (most recent call last):  
File "test.py", line 64, in <module>  
    product.price = -1  
File "test.py", line 57, in price  
    raise ValueError("- Price cannot be negative.")  
ValueError: - Price cannot be negative.
```

# En Résumé

- Les propriétés permettent de contrôler l'accès à certains attributs d'une instance.
- Elles se définissent dans le corps de la classe en suivant cette syntaxe :`nom_propriete = property(methode_accesseur, methode_mutateur, methode_suppression, methode_aide).`
- On y fait appel ensuite en écrivant `objet.nom_propriete` comme pour n'importe quel attribut.
- Si l'on souhaite juste lire l'attribut, c'est la méthode définie comme accesseur qui est appelée.
- Si l'on souhaite modifier l'attribut, c'est la méthode mutateur, si elle est définie, qui est appelée.
- Chacun des paramètres à passer à `property` est optionnel.

# Appliquez des méthodes spéciales

# Appliquez des méthodes spéciales

## Appliquez des méthodes spéciales

Les méthodes spéciales sont des méthodes d'instance que Python reconnaît et sait utiliser, dans certains contextes. Elles peuvent servir à indiquer à Python ce qu'il doit faire quand il se retrouve devant une expression comme `mon_objet1 + mon_objet2`, voire `mon_objet[indice]`. Et, encore plus fort, elles contrôlent la façon dont un objet se crée, ainsi que l'accès à ses attributs.

Bref, encore une fonctionnalité puissante et utile du langage, que je vous invite à découvrir. Prenez note du fait que je ne peux pas expliquer dans ce chapitre la totalité des méthodes spéciales. Il y en a qui ne sont pas de notre niveau, il y en a sur lesquelles je passerai plus vite que d'autres. En cas de doute, ou si vous êtes curieux, je vous encourage d'autant plus à aller faire un tour sur le site officiel de Python.

# Édition de l'objet et accès aux attributs

Vous avez déjà vu, dès le début de cette troisième partie, un exemple de **méthode spéciale**. Pour ceux qui ont la mémoire courte, il s'agit de notre constructeur. Une méthode spéciale, en Python, voit son nom entouré de part et d'autre par deux signes « souligné » `_`. Le nom d'une méthode spéciale prend donc la forme : `__methodespeciale__`.

Pour commencer, nous allons voir les méthodes qui travaillent directement sur l'objet. Nous verrons ensuite, plus spécifiquement, les méthodes qui permettent d'accéder aux attributs.

## Édition de l'objet

Les méthodes que nous allons voir permettent de travailler sur l'objet. Elles interviennent au moment de le créer et au moment de le supprimer. La première, vous devriez la reconnaître : c'est notre constructeur. Elle s'appelle `__init__`, prend un nombre variable d'arguments et permet de contrôler la création de nos attributs.

```
class Exemple:  
    """Un petit exemple de classe"""\n    def __init__(self, nom):  
        """Exemple de constructeur"""\n        self.nom = nom  
        self.autre_attribut = "une valeur"
```

Pour créer notre objet, nous utilisons le nom de la classe et nous passons, entre parenthèses, les informations qu'attend notre constructeur :

```
mon_objet = Exemple("un premier exemple")
```

J'ai un peu simplifié ce qui se passe mais, pour l'instant, c'est tout ce qu'il vous faut retenir. Comme vous pouvez le voir, à partir du moment où l'objet est créé, on peut accéder à ses attributs grâce à `mon_objet.nom_attribut` et exécuter ses méthodes grâce à `mon_objet.nom_methode(...)`.

Il existe également une autre méthode, `__del__`, qui va être appelée au moment de la destruction de l'objet.

La destruction ? Quand un objet se détruit-il ?

Bonne question. Il y a plusieurs cas : d'abord, quand vous voulez le supprimer explicitement, grâce au mot-clé `del(mon_objet)`. Ensuite, si l'espace de noms contenant l'objet est détruit, l'objet l'est également. Par exemple, si vous instanciez l'objet dans le corps d'une fonction : à la fin de l'appel à la fonction, la méthode `__del__` de l'objet sera appelée. Enfin, si votre objet résiste envers et contre tout pendant l'exécution du programme, il sera supprimé à la fin de l'exécution.

# Édition de l'objet et accès aux attributs

```
def __del__(self):
    """Méthode appelée quand l'objet est supprimé"""
    print("C'est la fin ! On me supprime !")
```

À quoi cela peut-il bien servir, de contrôler la destruction d'un objet ?

Souvent, à rien. Python s'en sort comme un grand garçon, il n'a pas besoin d'aide. Parfois, on peut vouloir récupérer des informations d'état sur l'objet au moment de sa suppression. Mais ce n'est qu'un exemple : les méthodes spéciales sont un moyen d'exécuter des actions personnalisées sur certains objets, dans un cas précis. Si l'utilité ne saute pas aux yeux, vous pourrez en trouver une un beau jour, en codant votre projet.

Souvenez-vous que si vous ne définissez pas de méthode spéciale pour telle ou telle action, Python aura un comportement par défaut dans le contexte où cette méthode est appelée. Écrire une méthode spéciale permet de modifier ce comportement par défaut. Dans l'absolu, vous n'êtes même pas obligés d'écrire un constructeur.

## Représentation de l'objet

Nous allons voir deux méthodes spéciales qui permettent de contrôler comment l'objet est représenté et affiché. Vous avez sûrement déjà pu constater que, quand on instancie des objets issus de nos propres classes, si on essaye de les afficher directement dans l'interpréteur ou grâce à `print`, on obtient quelque chose d'assez laid :

```
<__main__.XXX object at 0x00B46A70>
```

On a certes les informations utiles, mais pas forcément celles qu'on veut, et l'ensemble n'est pas magnifique, il faut bien le reconnaître.

La première méthode permettant de remédier à cet état de fait est `__repr__`. Elle affecte la façon dont est affiché l'objet quand on tape directement son nom. On la redéfinit quand on souhaite faciliter le `debug` sur certains objets :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
    def __repr__(self):
        """Quand on entre notre objet dans l'interpréteur"""
        return "Personne: nom({}), prénom({}), âge({})".format(
            self.nom, self.prenom, self.age)
```

# Édition de l'objet et accès aux attributs

Et le résultat en images :

```
p1 = Personne("Micado", "Jean")
```

```
Print(p1)
```

```
>>> Personne: nom(Micado), prénom(Jean), âge(33)
```

Comme vous le voyez, la méthode `__repr__` ne prend aucun paramètre (sauf, bien entendu, `self`) et renvoie une chaîne de caractères : la chaîne à afficher quand on entre l'objet directement dans l'interpréteur.

On peut également obtenir cette chaîne grâce à la fonction `repr`, qui se contente d'appeler la méthode spéciale `__repr__` de l'objet passé en paramètre :

```
p1 = Personne("Micado", "Jean")
```

```
repr(p1)
```

```
>>> 'Personne: nom(Micado), prénom(Jean), âge(33)'
```

Il existe une seconde méthode spéciale, `__str__`, spécialement utilisée pour afficher l'objet avec `print`. Par défaut, si aucune méthode `__str__` n'est définie, Python appelle la méthode `__repr__` de l'objet. La méthode `__str__` est également appelée si vous désirez convertir votre objet en chaîne avec le constructeur `str`.

```
class Personne:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def __str__(self):  
        return f'Person {self.name}, {self.age} years old.'  
  
    def __repr__(self):  
        return f"<Person('{self.name}', {self.age})>"  
  
bob = Person("Bob", 35)  
print(bob)  
  
>>> Person Bob, 35 years old. # affiche par __str__  
>>> <Person('Bob', 35)> # affiche par __repr__
```

La méthode `__str__` permet d'indiquer la représentation en chaîne de caractères d'un objets.

La méthode `__repr__` permet d'indiquer une chaîne de caractères qui sert de représentation à une classe

# Édition de l'objet et accès aux attributs

Et en pratique :

```
p1 = Personne("Micado", "Jean")
print(p1)
>>> Jean Micado, âgé de 33 ans

chaine = str(p1)
print(chaine)
>>> 'Jean Micado, âgé de 33 ans'
```

Accès aux attributs de notre objet

Nous allons découvrir trois méthodes permettant de définir comment accéder à nos attributs et les modifier.

La méthode `__getattribute__`

La méthode spéciale `__getattribute__` permet de définir une méthode d'accès à nos attributs plus large que celle que Python propose par défaut. En fait, cette méthode est appelée quand vous tapez `objet.attribut` (non pas pour modifier l'attribut mais simplement pour y accéder). Python recherche l'attribut et, s'il ne le trouve pas dans l'objet et si une méthode `__getattribute__` existe, il va l'appeler en lui passant en paramètre le nom de l'attribut recherché, sous la forme d'une chaîne de caractères.

Un petit exemple ?

# Édition de l'objet et accès aux attributs

```
class Protege:  
    """Classe possédant une méthode particulière d'accès à ses  
attributs :  
    Si l'attribut n'est pas trouvé, on affiche une alerte et  
renvoie None"""  
    def __init__(self):  
        """On crée quelques attributs par défaut"""  
        self.a = 1  
        self.b = 2  
        self.c = 3  
  
    def __getattr__(self, nom):  
        """Si Python ne trouve pas l'attribut nommé nom, il  
appelle  
    cette méthode. On affiche une alerte"""  
        print("Alerte ! Il n'y a pas d'attribut {} ici !".format(nom))  
  
pro = Protege()  
pro.a  
=>> 1  
pro.c  
=>> 3  
pro.e  
=>> Alerte ! Il n'y a pas d'attribut e ici !
```

# Édition de l'objet et accès aux attributs

Vous comprenez le principe ? Si l'attribut auquel on souhaite accéder existe, notre méthode n'est pas appelée. En revanche, si l'attribut n'existe pas, notre méthode `__getattr__` est appelée. On lui passe en paramètre le nom de l'attribut auquel Python essaye d'accéder. Ici, on se contente d'afficher une alerte. Mais on pourrait tout aussi bien rediriger vers un autre attribut. Par exemple, si on essaye d'accéder à un attribut qui n'existe pas, on redirige vers `self.c`. Je vous laisse faire l'essai, cela n'a rien de difficile.

## La méthode `__setattr__`

Cette méthode définit l'accès à un attribut destiné à être modifié. Si vous écrivez `objet.nom_attribut = nouvelle_valeur`, la méthode spéciale `__setattr__` sera appelée ainsi : `objet.__setattr__("nom_attribut", nouvelle_valeur)`. Là encore, le nom de l'attribut recherché est passé sous la forme d'une chaîne de caractères. Cette méthode permet de déclencher une action dès qu'un attribut est modifié, par exemple enregistrer l'objet :

```
def __setattr__(self, nom_attr, val_attr):
    """Méthode appelée quand on fait objet.nom_attr = val_attr.
    On se charge d'enregistrer l'objet"""

    object.__setattr__(self, nom_attr, val_attr)
    self.enregistrer()
```

Une explication s'impose concernant la ligne 6, je pense. Je vais faire de mon mieux, sachant que j'expliquerai bien plus en détail, au prochain chapitre, le concept d'héritage. Pour l'instant, il vous suffit de savoir que toutes les classes que nous créons sont héritées de la classe `object`. Cela veut dire essentiellement qu'elles reprennent les mêmes méthodes. La classe `object` est définie par Python. Je disais plus haut que, si vous ne définissiez pas une certaine méthode spéciale, Python avait un comportement par défaut : ce comportement est défini par la classe `object`.

La plupart des méthodes spéciales sont déclarées dans `object`. Si vous faites par exemple `objet.attribut = valeur` sans avoir défini de méthode `__setattr__` dans votre classe, c'est la méthode `__setattr__` de la classe `object` qui sera appelée.

Mais si vous redéfinissez la méthode `__setattr__` dans votre classe, la méthode appelée sera alors celle que vous définissez, et non celle de `object`. Oui mais... vous ne savez pas comment Python fait, réellement, pour modifier la valeur d'un attribut. Le mécanisme derrière la méthode vous est inconnu.

# Édition de l'objet et accès aux attributs

Tout cela pour dire que, dans votre méthode `__setattr__`, vous ne pouvez pas modifier d'attribut de la façon que vous connaissez. Si vous le faites, `__setattr__` appellera `__setattr__` qui appellera `__setattr__`... à l'infini. Donc si on souhaite modifier un attribut, on va se référer à la méthode `__setattr__` définie dans la classe `object`, la classe mère dont toutes nos classes héritent.

Si toutes ces explications vous ont paru plutôt dures, ne vous en faites pas trop : je détaillerai au prochain chapitre ce qu'est l'héritage, vous comprendrez sûrement mieux à ce moment.

## La méthode `__delattr__`

Cette méthode spéciale est appelée quand on souhaite supprimer un attribut de l'objet, en faisant `del objet.attribut` par exemple. Elle prend en paramètre, outre `self`, le nom de l'attribut que l'on souhaite supprimer. Voici un exemple d'une classe dont on ne peut supprimer aucun attribut :

```
def __delattr__(self, nom_attr):
    """On ne peut supprimer d'attribut, on lève l'exception
    AttributeError"""
    raise AttributeError("Vous ne pouvez supprimer aucun attribut de cette classe")
```

Là encore, si vous voulez supprimer un attribut, n'utilisez pas dans votre méthodedel `self.attribut`. Sinon, vous risquez de mettre Python très en colère ! Passez par `object.__delattr__` qui sait mieux que nous comment tout cela fonctionne.

## Un petit bonus

Voici quelques fonctions qui font à peu près ce que nous avons fait mais en utilisant des chaînes de caractères pour les noms d'attributs. Vous pourrez en avoir l'usage :

```
objet = MaClasse() # On crée une instance de notre classe
getattr(objet, "nom") # Semblable à objet.nom
setattr(objet, "nom", val) # = objet.nom = val ou objet.__setattr__("nom", val)
delattr(objet, "nom") # = del objet.nom ou objet.__delattr__("nom")
hasattr(objet, "nom") # Renvoie True si l'attribut "nom" existe, False sinon
```

# Édition de l'objet et accès aux attributs

Tout cela pour dire que, dans votre méthode `__setattr__`, vous ne pouvez pas modifier d'attribut de la façon que vous connaissez. Si vous le faites, `__setattr__` appellera `__setattr__` qui appellera `__setattr__`... à l'infini. Donc si on souhaite modifier un attribut, on va se référer à la méthode `__setattr__` définie dans la classe `object`, la classe mère dont toutes nos classes héritent.

Si toutes ces explications vous ont paru plutôt dures, ne vous en faites pas trop : je détaillerai au prochain chapitre ce qu'est l'héritage, vous comprendrez sûrement mieux à ce moment.

## La méthode `__delattr__`

Cette méthode spéciale est appelée quand on souhaite supprimer un attribut de l'objet, en faisant `del objet.attribut` par exemple. Elle prend en paramètre, outre `self`, le nom de l'attribut que l'on souhaite supprimer. Voici un exemple d'une classe dont on ne peut supprimer aucun attribut :

```
def __delattr__(self, nom_attr):
    """On ne peut supprimer d'attribut, on lève l'exception
    AttributeError"""
    raise AttributeError("Vous ne pouvez supprimer aucun attribut de cette classe")
```

Là encore, si vous voulez supprimer un attribut, n'utilisez pas dans votre méthodedel `self.attribut`. Sinon, vous risquez de mettre Python très en colère ! Passez par `object.__delattr__` qui sait mieux que nous comment tout cela fonctionne.

## Un petit bonus

Voici quelques fonctions qui font à peu près ce que nous avons fait mais en utilisant des chaînes de caractères pour les noms d'attributs. Vous pourrez en avoir l'usage :

```
objet = MaClasse() # On crée une instance de notre classe
getattr(objet, "nom") # Semblable à objet.nom
setattr(objet, "nom", val) # = objet.nom = val ou objet.__setattr__("nom", val)
delattr(objet, "nom") # = del objet.nom ou objet.__delattr__("nom")
hasattr(objet, "nom") # Renvoie True si l'attribut "nom" existe, False sinon
```

# Édition de l'objet et accès aux attributs

Peut-être ne voyez-vous pas trop l'intérêt de ces fonctions qui prennent toutes, en premier paramètre, l'objet sur lequel travailler et en second le nom de l'attribut (sous la forme d'une chaîne). Toutefois, cela peut être très pratique parfois de travailler avec des chaînes de caractères plutôt qu'avec des noms d'attributs. D'ailleurs, c'est un peu ce que nous venons de faire, dans nos redéfinitions de méthodes accédant aux attributs.

Là encore, si l'intérêt ne saute pas aux yeux, laissez ces fonctions de côté. Vous pourrez les retrouver par la suite.

# Les méthodes de conteneur

Nous allons commencer à travailler sur ce que l'on appelle la surcharge d'opérateurs. Il s'agit assez simplement d'expliquer à Python quoi faire quand on utilise tel ou tel opérateur. Nous allons ici voir quatre méthodes spéciales qui interviennent quand on travaille sur des objets conteneurs.

## Accès aux éléments d'un conteneur

Les objets conteneurs, j'espère que vous vous en souvenez, ce sont les chaînes de caractères, les listes et les dictionnaires, entre autres. Tous ont un point commun : ils contiennent d'autres objets, auxquels on peut accéder grâce à l'opérateur`[]`.

Les trois premières méthodes que nous allons voir sont`__getitem__`,`__setitem__` et`__delitem__`. Elles servent respectivement à définir quoi faire quand on écrit :

```
objet[index];  
objet[index] = valeur;  
del objet[index];
```

Pour cet exemple, nous allons voir une classe enveloppe de dictionnaire. Les classes enveloppes sont des classes qui ressemblent à d'autres classes mais n'en sont pas réellement. Cela vous avance ?

Nous allons créer une classe que nous allons appeler`ZDict`. Elle va posséder un attribut auquel on ne devra pas accéder de l'extérieur de la classe, un dictionnaire que nous appellerons`_dictionnaire`. Quand on créera un objet de type`ZDict` et qu'on voudra faire`objet[index]`, à l'intérieur de la classe on fera`self._dictionnaire[index]`. En réalité, notre classe fera semblant d'être un dictionnaire, elle réagira de la même manière, mais elle n'en sera pas réellement un.

# Les méthodes de conteneur

```
class ZDict:  
    """Classe enveloppe d'un dictionnaire"""  
  
    def __init__(self):  
        """Notre classe n'accepte aucun paramètre"""  
        self._dictionnaire = {}  
  
    def __getitem__(self, index):  
        """Cette méthode spéciale est appelée quand on fait objet[index]  
        Elle redirige vers self._dictionnaire[index]"""  
  
        return self._dictionnaire[index]  
  
    def __setitem__(self, index, valeur):  
        """Cette méthode est appelée quand on écrit objet[index] = valeur  
        On redirige vers self._dictionnaire[index] = valeur"""  
  
        self._dictionnaire[index] = valeur
```

Vous avez un exemple d'utilisation des deux méthodes `__getitem__` et `__setitem__` qui, je pense, est assez clair. Pour `__delitem__`, je crois que c'est assez évident, elle ne prend qu'un seul paramètre qui est l'index que l'on souhaite supprimer. Vous pouvez étendre cet exemple avec d'autres méthodes que nous avons vues plus haut, notamment `__repr__` et `__str__`. N'hésitez pas, entraînez-vous, tout cela peut vous servir.

## La méthode spéciale derrière le mot-clé `in`

Il existe une quatrième méthode, appelée `__contains__`, qui est utilisée quand on souhaite savoir si un objet se trouve dans un conteneur.

Exemple classique :

```
ma_liste = [1, 2, 3, 4, 5]  
8 in ma_liste # Revient au même que ...  
ma_liste.__contains__(8)
```

# Les méthodes de conteneur

Ainsi, si vous voulez que votre classe enveloppe puisse utiliser le mot-clé `in` comme une liste ou un dictionnaire, vous devez redéfinir cette méthode `__contains__` qui prend en paramètre, outre `self`, l'objet qui nous intéresse. Si l'objet est dans le conteneur, on doit renvoyer `True`; sinon `False`.

Je vous laisse redéfinir cette méthode, vous avez toutes les indications nécessaires.

## Connaître la taille d'un conteneur

Il existe enfin une méthode spéciale `__len__()`, appelée quand on souhaite connaître la taille d'un objet conteneur, grâce à la fonction `len`.

`len(objet)` équivaut à `objet.__len__()`. Cette méthode spéciale ne prend aucun paramètre et renvoie une taille sous la forme d'un entier. Là encore, je vous laisse faire l'essai.

# Les méthodes mathématiques

Pour cette section, nous allons continuer à voir les méthodes spéciales permettant la surcharge d'opérateurs mathématiques, comme +, -, \* et j'en passe.  
Ce qu'il faut savoir

Pour cette section, nous allons utiliser un nouvel exemple, une classe capable de contenir des durées. Ces durées seront contenues sous la forme d'un nombre de minutes et un nombre de secondes.

Voici le corps de la classe, gardez-le sous la main :

```
class Duree:  
    """Classe contenant des durées sous la forme d'un nombre de minutes  
    et de secondes"""  
  
    def __init__(self, min=0, sec=0):  
        """Constructeur de la classe"""  
        self.min = min # Nombre de minutes  
        self.sec = sec # Nombre de secondes  
  
    def __str__():  
        """Affichage un peu plus joli de nos objets"""  
        return "{0:02}:{1:02}".format(self.min, self.sec)
```

On définit simplement deux attributs contenant notre nombre de minutes et notre nombre de secondes, ainsi qu'une méthode pour afficher tout cela un peu mieux. Si vous vous interrogez sur l'utilisation de la méthode `format` dans la méthode `__str__`, sachez simplement que le but est de voir la durée sous la forme MM:SS; pour plus d'informations sur le formatage des chaînes, vous pouvez consulter [la documentation de Python](#).

Créons un premier objet `Duree` que nous appelons `d1`.

```
d1 = Duree(3, 5)  
print(d1)  
>>> 03:05
```

# Les méthodes mathématiques

Si vous essayez de faire `d1 + 4`, par exemple, vous allez obtenir une erreur. Python ne sait pas comment additionner un type `Duree` et un `int`. Il ne sait même pas comment ajouter deux durées ! Nous allons donc lui expliquer.

La méthode spéciale à redéfinir est `__add__`. Elle prend en paramètre l'objet que l'on souhaite ajouter. Voici deux lignes de code qui reviennent au même :

```
d1 + 4  
d1.__add__(4)
```

Comme vous le voyez, quand vous utilisez le symbole `+` ainsi, c'est en fait la méthode `__add__` de l'objet `Duree` qui est appelée. Elle prend en paramètre l'objet que l'on souhaite ajouter, peu importe le type de l'objet en question. Et elle doit renvoyer un objet exploitable, ici il serait plus logique que ce soit une nouvelle durée.

Si vous devez faire différentes actions en fonction du type de l'objet à ajouter, testez le résultat de `type(objet_a_ajouter)`.

```
def __add__(self, objet_a_ajouter):  
    """L'objet à ajouter est un entier, le nombre de secondes"""  
    nouvelle_duree = Duree()  
    # On va copier self dans l'objet créé pour avoir la même durée  
    nouvelle_duree.min = self.min  
    nouvelle_duree.sec = self.sec  
    # On ajoute la durée  
    nouvelle_duree.sec += objet_a_ajouter  
    # Si le nombre de secondes >= 60  
    if nouvelle_duree.sec >= 60:  
        nouvelle_duree.min += nouvelle_duree.sec // 60  
        nouvelle_duree.sec = nouvelle_duree.sec % 60  
    # On renvoie la nouvelle durée  
    return nouvelle_duree
```

# Les méthodes mathématiques

Prenez le temps de comprendre le mécanisme et le petit calcul pour vous assurer d'avoir une durée cohérente. D'abord, on crée une nouvelle durée qui est l'équivalent de la durée contenue dans self. On l'augmente du nombre de secondes à ajouter et on s'assure que le temps est cohérent (le nombre de secondes n'atteint pas 60). Si le temps n'est pas cohérent, on le corrige. On renvoie enfin notre nouvel objet modifié. Voici un petit code qui montre comment utiliser notre méthode :

```
d1 = Duree(12, 8)
print(d1)
>>> 12:08
d2 = d1 + 54 # d1 + 54 secondes
print(d2)
>>> 13:02
```

Pour mieux comprendre, remplacez `d2 = d1 + 54` par `d2 = d1.__add__(54)`: cela revient au même. Ce remplacement ne sert qu'à bien comprendre le mécanisme. Il va de soi que ces méthodes spéciales ne sont pas à appeler directement depuis l'extérieur de la classe, les opérateurs n'ont pas été inventés pour rien.

Sachez que sur le même modèle, il existe les méthodes :

- `__sub__`: surcharge de l'opérateur -;
- `__mul__`: surcharge de l'opérateur \*;
- `__truediv__`: surcharge de l'opérateur /;
- `__floordiv__`: surcharge de l'opérateur // (division entière) ;
- `__mod__`: surcharge de l'opérateur % (modulo) ;
- `__pow__`: surcharge de l'opérateur \*\* (puissance) ;
- ...

Il y en a d'autres que vous pouvez consulter sur le site web de Python.

# Les méthodes mathématiques

Tout dépend du sens

Vous l'avez peut-être remarqué, et c'est assez logique si vous avez suivi mes explications, mais écrire `objet1 + objet2` ne revient pas au même qu'écrire `objet2 + objet1` si les deux objets ont des types différents.

En effet, suivant le cas, c'est la méthode `__add__` de l'un ou l'autre des objets qui est appelée.

Cela signifie que, lorsqu'on utilise la classe `Duree`, si on écrit `d1 + 4` cela fonctionne, alors que `4 + d1` ne marche pas. En effet, la class `int` ne sait pas quoi faire de votre objet `Duree`.

Il existe cependant une panoplie de méthodes spéciales pour faire le travail de `__add__` si vous écrivez l'opération dans l'autre sens. Il suffit de préfixer le nom des méthodes spéciales par un `r`.

```
def __add__(self, objet_a_ajouter):
    """Cette méthode est appelée si on écrit 4 + objet et que
    le premier objet (4 dans cet exemple) ne sait pas comment ajouter
    le second. On se contente de rediriger sur __add__ puisque,
    ici, cela revient au même : l'opération doit avoir le même résultat,
    posée dans un sens ou dans l'autre"""
    return self + objet_a_ajouter
```

À présent, on peut écrire `4 + d1`, cela revient au même que `d1 + 4`.

N'hésitez pas à relire ces exemples s'ils vous paraissent peu clairs.

## D'autres opérateurs

Il est également possible de surcharger les opérateurs `+=`, `-=`, etc. On préfixe cette fois-ci les noms de méthode que nous avons vus par un `i`.

Exemple de méthode `__iadd__` pour notre classe `Duree`:

# Les méthodes mathématiques

```
def __iadd__(self, objet_a_ajouter):
    """L'objet à ajouter est un entier, le nombre de secondes"""
    # On travaille directement sur self cette fois
    # On ajoute la durée
    self.sec += objet_a_ajouter
    # Si le nombre de secondes >= 60
    if self.sec >= 60:
        self.min += self.sec // 60
        self.sec = self.sec % 60
    # On renvoie self
    return self
```

Et en images :

```
d1 = Duree(8, 5)
d1 += 128
print(d1)
>>> 10:13
```

Je ne peux que vous encourager à faire des tests, pour être bien sûrs de comprendre le mécanisme. Je vous ai donné ici une façon de faire en la commentant mais, si vous ne pratiquez pas ou n'essayez pas par vous-mêmes, vous n'allez pas la retenir et vous n'allez pas forcément comprendre la logique.

# Les méthodes de comparaison

Pour finir, nous allons voir la surcharge des opérateurs de comparaison que vous connaissez depuis quelque temps maintenant : `==, !=, <, >, <=, >=`.

Ces méthodes sont donc appelées si vous tentez de comparer deux objets entre eux. Comment Python sait-il que 3 est inférieur à 18 ? Une méthode spéciale de la classe `int` le permet, en simplifiant. Donc si vous voulez comparer des durées, par exemple, vous allez devoir redéfinir certaines méthodes que je vais présenter plus bas. Elles devront prendre en paramètre l'objet à comparer à `self`, et doivent renvoyer un booléen (`True` ou `False`).

Je vais me contenter de vous faire un petit tableau récapitulatif des méthodes à redéfinir pour comparer deux objets entre eux :

```
d1 = Duree(8, 5)
d1 += 128
print(d1)
>>> 10:13
```

Je ne peux que vous encourager à faire des tests, pour être bien sûrs de comprendre le mécanisme. Je vous ai donné ici une façon de faire en la commentant mais, si vous ne pratiquez pas ou n'essayez pas par vous-mêmes, vous n'allez pas la retenir et vous n'allez pas forcément comprendre la logique.

# Les méthodes de comparaison 1/2

Sachez que ce sont ces méthodes spéciales qui sont appelées si, par exemple, vous voulez trier une liste contenant vos objets.

Sachez également que, si Python n'arrive pas à faire `objet1 < objet2`, il essaiera l'opération inverse, soit `objet2 >= objet1`. Cela vaut aussi pour les autres opérateurs de comparaison que nous venons de voir.

Allez, je vais vous mettre deux exemples malgré tout, il ne tient qu'à vous de redéfinir les autres méthodes présentées plus haut :

Opérateur	Méthode spéciale	Résumé
<code>==</code>	<code>def __eq__(self, objet_a_comparer):</code>	Opérateur d'égalité ( <i>equal</i> ). Renvoie <code>True</code> si <code>self</code> et <code>objet_a_comparer</code> sont égaux, <code>False</code> sinon.
<code>!=</code>	<code>def __ne__(self, objet_a_comparer):</code>	Différent de ( <i>non equal</i> ). Renvoie <code>True</code> si <code>self</code> et <code>objet_a_comparer</code> sont différents, <code>False</code> sinon.
<code>&gt;</code>	<code>def __gt__(self, objet_a_comparer):</code>	Teste si <code>self</code> est strictement supérieur ( <i>greater than</i> ) à <code>objet_a_comparer</code> .
<code>&gt;=</code>	<code>def __ge__(self, objet_a_comparer):</code>	Teste si <code>self</code> est supérieur ou égal ( <i>greater or equal</i> ) à <code>objet_a_comparer</code> .
<code>&lt;</code>	<code>def __lt__(self, objet_a_comparer):</code>	Teste si <code>self</code> est strictement inférieur ( <i>lower than</i> ) à <code>objet_a_comparer</code> .
<code>&lt;=</code>	<code>def __le__(self, objet_a_comparer):</code>	Teste si <code>self</code> est inférieur ou égal ( <i>lower or equal</i> ) à <code>objet_a_comparer</code> .

# Les méthodes de comparaison 2/2

```
class Point6:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __eq__(self, other):  
        return self.x == other.x and self.y == other.y  
  
    def __gt__(self, other):  
        return self.x > other.x and self.y > other.y  
  
point = Point6(10, 20)  
other = Point6(1, 2)  
  
print(point == other)  
print(point > other)  
print(point < other)  
  
>>> False  
>>> True  
>>> False
```

# Des méthodes spéciales utiles à pickle

Vous vous souvenez de `pickle`, j'espère. Pour conclure ce chapitre sur les méthodes spéciales, nous allons en voir deux qui sont utilisées par ce module pour influencer la façon dont nos objets sont enregistrés dans des fichiers.

Prenons un cas concret, d'une utilité pratique discutable.

On crée une classe qui va contenir plusieurs attributs. Un de ces attributs possède une valeur temporaire, qui n'est utile que pendant l'exécution du programme. Si on arrête ce programme et qu'on le relance, on doit récupérer le même objet mais la valeur temporaire doit être remise à 0, par exemple.

Il y a d'autres moyens d'y parvenir, je le reconnais. Mais les autres applications que j'ai en tête sont plus dures à développer et à expliquer rapidement, donc gardons cet exemple.

## La méthode spéciale `__getstate__`

La méthode `__getstate__` est appelée au moment de sérialiser l'objet. Quand vous voulez enregistrer l'objet à l'aide du module `pickle`, `__getstate__` va être appelée juste avant l'enregistrement.

Si aucune méthode `__getstate__` n'est définie, `pickle` en registre le dictionnaire des attributs de l'objet à enregistrer. Vous vous rappelez ? Il est contenu dans `objet.__dict__`.

Sinon, `pickle` enregistre dans le fichier la valeur renvoyée par `__getstate__` (généralement, un dictionnaire d'attributs modifié).

Voyons un peu comment coder notre exemple grâce à `__getstate__`:

```
class Temp:  
    """Classe contenant plusieurs attributs, dont un temporaire"""
```

```
    def __init__(self):  
        """Constructeur de notre objet"""""  
        self.attribut_1 = "une valeur"  
        self.attribut_2 = "une autre valeur"  
        self.attribut_temporaire = 5
```

```
    def __getstate__(self):  
        """Renvoie le dictionnaire d'attributs à sérialiser"""""  
        dict_attr = dict(self.__dict__)  
        dict_attr["attribut_temporaire"] = 0  
        return dict_attr
```

# Des méthodes spéciales utiles à pickle

Avant de revenir sur le code, vous pouvez en voir les effets. Si vous tentez d'enregistrer cet objet grâce à `pickle` et que vous le récupérez ensuite depuis le fichier, vous constatez que l'attribut `attribut_temporaire` est à 0, peu importe sa valeur d'origine.

Voyons le code de `__getstate__`. La méthode ne prend aucun argument (excepté `self` puisque c'est une méthode d'instance).

Elle enregistre le dictionnaire des attributs dans une variable locale `dict_attr`. Ce dictionnaire a le même contenu que `self.__dict__` (le dictionnaire des attributs de l'objet). En revanche, il a une référence différente. Sans cela, à la ligne suivante, au moment de modifier `attribut_temporaire`, le changement aurait été également appliqué à l'objet, ce que l'on veut éviter.

À la ligne suivante, donc, on change la valeur de l'attribut `attribut_temporaire`. Étant donné que `dict_attr` et `self.__dict__` n'ont pas la même référence, l'attribut n'est changé que dans `dict_attr` et le dictionnaire de `self` n'est pas modifié.

Enfin, on renvoie `dict_attr`. Au lieu d'enregistrer dans notre fichier `self.__dict__`, `pickle` enregistre notre dictionnaire modifié, `dict_attr`.

Si ce n'est pas assez clair, je vous encourage à tester par vous-mêmes, essayez de modifier la méthode `__getstate__` et manipulez `self.__dict__` pour bien comprendre le code.

## La méthode `__setstate__`

À la différence de `__getstate__`, la méthode `__setstate__` est appelée au moment de déserialiser l'objet. Concrètement, si vous récupérez un objet à partir d'un fichier sérialisé, `__setstate__` sera appelée après la récupération du dictionnaire des attributs.

Pour schématiser, voici l'exécution que l'on va observer derrière `unpickler.load()`:

1. L'objet **Unpickler** lit le fichier.
2. Il récupère le dictionnaire des attributs. Je vous rappelle que si aucune méthode `__getstate__` n'est définie dans notre classe, ce dictionnaire est celui contenu dans l'attribut spécial `__dict__` de l'objet au moment de sa sérialisation.
3. Ce dictionnaire récupéré est envoyé à la méthode `__setstate__` si elle existe. Si elle n'existe pas, Python considère que c'est le dictionnaire des attributs de l'objet à récupérer et écrit donc l'attribut `__dict__` de l'objet en y plaçant ce dictionnaire récupéré.

Le même exemple mais, cette fois, par la méthode `__setstate__`:

# Des méthodes spéciales utiles à pickle

```
def __setstate__(self, dict_attr):
    """Méthode appelée lors de la désérialisation de l'objet"""
    dict_attr["attribut_temporaire"] = 0
    self.__dict__ = dict_attr
```

Quelle est la différence entre les deux méthodes que nous avons vues ?

L'objectif que nous nous étions fixé peut être atteint par ces deux méthodes. Soit notre classe met en œuvre une méthode `__getstate__`, soit elle met en œuvre une méthode `__setstate__`.

Dans le premier cas, on modifie le dictionnaire des attributs avant la sérialisation. Le dictionnaire des attributs enregistré est celui que nous avons modifié avec la valeur de notre attribut temporaire à `0`.

Dans le second cas, on modifie le dictionnaire d'attributs après la désérialisation. Le dictionnaire que l'on récupère contient un attribut `attribut_temporaire` avec une valeur quelconque (on ne sait pas laquelle) mais après avoir récupéré l'objet qui est déjà instancié (et avant le retour de la désérialisation !), on met cette valeur à `0`.

Ce sont deux moyens différents, qui ici reviennent au même. À vous de choisir la meilleure méthode en fonction de vos besoins (les deux peuvent être présentes dans la même classe si nécessaire).

Là encore, je vous encourage à faire des essais si ce n'est pas très clair.

## On peut enregistrer dans un fichier autre chose que des dictionnaires

Votre méthode `__getstate__` n'est pas obligée de renvoyer un dictionnaire d'attributs. Elle peut renvoyer un autre objet, un entier, un flottant, mais dans ce cas une méthode `__setstate__` devra exister pour savoir « quoi faire » avec l'objet enregistré. Si ce n'est pas un dictionnaire d'attributs, Python ne peut pas le deviner !

Là encore, je vous laisse tester si cela vous intéresse.

## Je veux encore plus puissant !

`__getstate__` et `__setstate__` sont les deux méthodes les plus connues pour agir sur la sérialisation d'objets. Mais il en existe d'autres, plus complexes.

Si vous êtes intéressés, jetez un œil du côté de la [PEP 307](#).

# Résumé

En résumé

Les méthodes spéciales permettent d'influencer la manière dont Python accède aux attributs d'une instance et réagit à certains opérateurs ou conversions.

Les méthodes spéciales sont toutes entourées de deux signes « souligné » (\_).

Les méthodes `__getattr__`, `__setattr__` et `__delattr__` contrôlent l'accès aux attributs de l'instance.

Les méthodes `__getitem__`, `__setitem__` et `__delitem__` surchargent l'indexation ([]).

Les méthodes `__add__`, `__sub__`, `__mul__` ... surchargent les opérateurs mathématiques.

Les méthodes `__eq__`, `__ne__`, `__gt__` ... surchargent les opérateurs de comparaison.

# Making Custom Containers

# Making Custom Containers

```
class TagCloud:  
    def __init__(self):  
        self.__tags = {} # when you prefix an attribute with __ like __tags it is considered as private attribute  
  
    def add(self, tag):  
        self.__tags[tag.lower()] = self.__tags.get(tag.lower(), 0) + 1  
  
    def __getitem__(self, tag):  
        return self.__tags.get(tag.lower(), 0)  
  
    def __setitem__(self, tag, count):  
        self.__tags[tag.lower()] = count  
  
    def __len__(self):  
        return len(self.__tags)  
  
    def __iter__(self):  
        return iter(self.__tags)  
  
# why are we not using a classical dictionary? Because a dictionary can't deal with Python and python,  
# they will make distinct count when we want one. So we can make our custom containers smarter than a  
# simple dictionary  
  
cloud = TagCloud()  
# cloud(cloud.__tags)  
cloud.add("Python")  
cloud.add("python")  
cloud.add("python")  
print(cloud["Python"])  
# print(cloud.__tags["PYTHON"]])
```

# Appliquez 2 méthodes de tri

# Appliquez 2 méthodes de tri

- Trier une liste d'informations quelconque peut s'avérer très utile... et souvent difficile. Python nous offre plusieurs techniques pour trier, que ce soit de simples listes de nombres, de chaînes de caractères ou de données plus complexes (comme des objets dont nous avons créé nous-mêmes les classes).
- Ce chapitre est une parenthèse : vous pouvez aller tout de suite au chapitre suivant sans problème, et revenir à celui-ci plus tard.

# Première approche du tri

La première question que vous devriez vous poser : on a une liste, on veut la trier, mais que veut-on dire par « trier » ?

Trier, c'est ordonner la liste d'une façon cohérente. Par exemple, on pourrait vouloir trier une liste de noms par ordre alphabétique. Ou on pourrait vouloir trier une liste de nombres du plus petit au plus grand.

Dans tous les cas, trier une liste c'est la réordonner (changer son ordre, si nécessaire) selon certains critères. Il est important que vous gardiez en tête cette notion de « critères » par la suite, car nous allons en reparler.

Deux méthodes

Pour trier une séquence de données, Python nous propose deux méthodes :

- La première est une méthode de liste. Elle s'appelle tout simplement `sort` (trier en anglais). Elle travaille sur la liste-même et change donc son ordre, si c'est nécessaire.
- La seconde est la fonction `sorted`. Il s'agit d'une fonction builtin, c'est-à-dire qu'elle est disponible d'office dans Python sans avoir besoin d'importer quoique ce soit. Contrairement à la méthode `sort` de la class `list`, `sorted` travaille sur n'importe quel type de séquence (tuple, liste ou même dictionnaire). Une importante différence avec la méthode `list.sort` est qu'elle ne modifie pas l'objet d'origine, mais en retourne un nouveau.

Voyons quelques exemples :

```
prenoms = ["Jacques", "Laure", "André", "Victoire", "Albert", "Sophie"]
print(f"1 - {prenoms.sort()}")
print(f"2 - {prenoms}")
# Et avec la fonction 'sorted'
prenoms = ["Jacques", "Laure", "André", "Victoire", "Albert", "Sophie"]
print(f"3 - {sorted(prenoms)}")
print(f"4 - {prenoms}")
```

```
1 - None
2 - ['Albert', 'André', 'Jacques', 'Laure', 'Sophie', 'Victoire']
3 - ['Albert', 'André', 'Jacques', 'Laure', 'Sophie', 'Victoire']
4 - ['Jacques', 'Laure', 'André', 'Victoire', 'Albert', 'Sophie']
```

Vous devriez remarquer deux choses ici :

1. D'abord, Python a trié notre liste par ordre alphabétique. Nous verrons plus tard pourquoi.
2. Le second moyen (avec la fonction `sorted`) n'a pas modifié la liste, elle a juste retournée une nouvelle liste triée. La méthode de liste `sort`, elle, a travaillé sur notre liste et l'a modifiée.

# Aperçu des critères de tri

Python a trié la liste par ordre alphabétique... mais nous ne lui avons rien demandé à cet égard. En un sens, tant mieux, si c'est ce que vous vouliez faire, mais il est préférable de comprendre pourquoi. Je vous met ici un petit code qui devrait vous aider à comprendre sur quelle information Python se fonde pour déterminer la meilleure méthode de tri :

```
print(sorted([1, 8, -2, 15, 9]))  
print(sorted(["1", "8", "-2", "15", "9"]))  
>>> [-2, 1, 8, 9, 15]  
>>> ['-2', '1', '15', '8', '9']
```

La réponse se trouve dans la différence entre la ligne 1 et la ligne 3. Vous avez trouvé ?

Pour Python, la méthode de tri dépend du type des éléments que la séquence contient. On lui a demandé de trier une liste de nombres (type int) et Python trie du plus petit au plus grand. Sans surprise.

À la ligne 3 cependant, on lui demande de trier la même liste, sauf que nos nombres sont devenus des chaînes de caractères (type str). Python choisit donc de trier la liste par ordre alphabétique.

Et si on a une liste contenant plusieurs types ?

Dans ce cas, Python va vous dire, à sa façon, qu'il ne sait pas quelle méthode de tri choisir.

```
sorted([1, "8", "-2", "15", 9])  
Traceback (most recent call last):  
  File "C:\Users\MOTTIER LUCIE\Documents\GitHub\TheCompletePythonCourse\test.py", line 46, in <module>  
    sorted([1, "8", "-2", "15", 9])  
TypeError: '<' not supported between instances of 'str' and 'int'
```

Notre liste contient des nombres (type int) et des chaînes de caractères (type str). Le message d'erreur n'est peut-être pas très explicite tant qu'on ne connaît pas la façon dont Python trie une séquence, nous verrons ça un peu plus loin dans le chapitre. En attendant, intéressons-nous à des types plus particuliers !

# Trier avec des clés précises

Les deux moyens que nous venons de voir sont pratiques, mais limités. Si nous voulons trier une liste contenant des données de types différents, selon des critères un peu plus particuliers, on va avoir quelques problèmes.

Considérez cet exemple : on veut conserver, dans une liste simple, les étudiants, leur âge et leur note moyenne (entre 0 et 20). On va commencer par créer une liste assez simple, contenant des tuples. Pour chaque tuple, on indiquera le nom de l'étudiant, son âge et sa moyenne. Voyons le code :

```
etudiants = [  
    ("Clément", 14, 16),  
    ("Charles", 12, 15),  
    ("Oriane", 14, 18),  
    ("Thomas", 11, 12),  
    ("Damien", 12, 15),  
]
```

Souvenez-vous : première colonne, prénom, deuxième colonne, âge et troisième colonne, moyenne entre 0 et 20.  
Maintenant, si vous essayez de trier cette liste sans préciser de méthode :

```
print(sorted(etudiants))  
[('Charles', 12, 15), ('Clément', 14, 16), ('Damien', 12, 15), ('Oriane', 14, 18), ('Thomas', 11, 12)]
```

Le plus important pour nous, c'est que le tri semble s'effectuer sur la première colonne : sur les prénoms. L'ordre retourné est celui des étudiants par ordre alphabétique.

Maintenant, supposons que nous voulions trier par note.

Il suffit de changer les colonnes de notre liste, non ?

Oui, c'est une solution et il s'agit probablement de la solution à laquelle on pense le plus vite : changer les colonnes de notre liste, pour mettre les notes au début de notre tuple, et après trier la liste.

Mais il y a plus simple !

# L'argument key 1/2

La méthode list.sort ou la fonction sorted ont tous deux un paramètre optionnel, appelé key.

Cet argument attend... une fonction. Attendez ! Je m'explique.

La fonction à passer en paramètre prend un élément de la liste et retourne ce sur quoi doit s'effectuer le tri.

Donc la première chose est de créer une fonction ?

Oui, mais de façon assez simple : nous allons utiliser nos fonctions lambdas. Vous vous en souvenez ? Je vous donne un petit exemple de code si besoin :

```
doubler = lambda x: x * 2
print(doubler)
print(doubler(8))
>> <function <lambda> at 0x0000019A2443CF70>
>> 16
```

Les fonctions lambdas sont des fonctions particulières que l'on peut créer grâce au mot clé lambda.

Sa syntaxe est la suivante :

1. D'abord, après le mot clé lambda, les arguments de la fonction à créer, séparés par une virgule si il y en a plusieurs ;
2. Ensuite, les deux points (:) ;
3. Et ensuite le retour de la fonction. Ici, on retourne le paramètre fois 2, tout simplement.

Pourquoi ce rappel sur les lambdas ?

Parce que, pour trier, nous allons nous en servir. Pour préciser la méthode de tri, il nous faut une fonction qui prenne en paramètre un élément de la liste à trier et retourne l'élément qui doit être utilisé pour trier.

- L'élément de notre liste étudiants, c'est un tuple contenant le prénom, l'âge et la moyenne de l'étudiant ;
- On veut trier le tableau des étudiants en fonction des notes (la troisième colonne du tuple).

Est-ce que ces informations vous aident pour créer notre fonction lambda ?

La voici :

```
lambda colonnes: colonnes[2]
```

# L'argument key 2/2

colonnes contiendra un élément de la liste des étudiants (c'est-à-dire un tuple). Si on retourne colonnes[2], cela signifie qu'on veut récupérer la moyenne de l'étudiant (troisième colonne). Souvenez-vous, pour un tuple, la première colonne est toujours 0.

Essayons à présent de trier notre liste d'étudiants en fonction de leur moyenne :

```
sorted(etudiants, key=lambda colonnes: colonnes[2])
>>>[('Thomas', 11, 12), ('Charles', 12, 15), ('Damien', 12, 15), ('Clément', 14, 16), ('Oriane', 14, 18)]
```

Si le code ne vous paraît pas clair, prenez le temps de relire les explications. Il faut un peu de temps pour s'adapter aux fonctions lambdas, mais vous verrez qu'elles sont parfois très utiles.

# Trier une liste d'objets 1/3

Jusqu'ici, nous avons trié des listes contenant des nombres ou chaînes de caractères. Ce sont des objets, bien entendu, mais maintenant je voudrais vous montrer comment trier des objets issus de classes que nous avons créées.

Je vais reprendre le même exemple de notre tableau d'étudiants. Simplement, au lieu de conserver des tuples, nous allons conserver des objets. Plus intuitif et plus lisible, je trouve :

```
class Etudiant:  
    """Classe représentant un étudiant.  
  
    On représente un étudiant par son prénom (attribut prenom), son âge  
(attribut age) et sa note moyenne (attribut moyenne, entre 0 et 20).  
  
    Paramètres du constructeur :  
        prenom -- le prénom de l'étudiant  
        age -- l'âge de l'étudiant  
        moyenne -- la moyenne de l'étudiant  
  
    """  
  
    def __init__(self, prenom, age, moyenne):  
        self.prenom = prenom  
        self.age = age  
        self.moyenne = moyenne  
  
    def __repr__(self):  
        return "<Étudiant {} (âge={}, moyenne={})>".format(  
            self.prenom, self.age, self.moyenne)
```

# Trier une liste d'objets 2/3

Maintenant, recréons notre liste :

```
etudiants = [  
    Etudiant("Clément", 14, 16),  
    Etudiant("Charles", 12, 15),  
    Etudiant("Oriane", 14, 18),  
    Etudiant("Thomas", 11, 12),  
    Etudiant("Damien", 12, 15),  
]
```

Si vous essayez de trier notre liste telle quelle, vous allez avoir une erreur qui devrait vous sembler familière :

```
etudiants  
[  
    <Étudiant Clément (âge=14, moyenne=16)>,  
    <Étudiant Charles (âge=12, moyenne=15)>,  
    <Étudiant Oriane (âge=14, moyenne=18)>,  
    <Étudiant Thomas (âge=11, moyenne=12)>,  
    <Étudiant Damien (âge=12, moyenne=15)>  
]  
sorted(etudiants)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unorderable types: Etudiant() < Etudiant()
```

Python ne sait pas comment trier nos étudiants. Il y a deux façons de le lui expliquer :

1. L'une est de définir la méthode spéciale `__lt__` de notre classe. C'est en effet cette méthode (utilisée pour la comparaison) qui est utilisée par Python pour trier une liste, en comparant chacun de ses éléments. La méthode `__lt__` (lower than) correspond à l'opérateur `<`;
2. On peut aussi utiliser l'argument `key`, comme nous l'avons fait précédemment.

# Trier une liste d'objets 3/3

Ici notre seconde possibilité est plus pertinente. Redéfinir la méthode `__lt__` est une bonne idée si notre objet est un nombre (par exemple une durée ou bien une heure). Dans ce cas précis, il est préférable d'utiliser l'argument `key` de la fonction `sorted` (ou de la méthode `list.sort`).

Saurez-vous trier cette liste d'étudiants en fonction de leur moyenne ?

Voici le code :

```
sorted(etudiants, key=lambda etudiant: etudiant.moyenne)
>>>
[
    <Étudiant Thomas (âge=11, moyenne=12)>,
    <Étudiant Charles (âge=12, moyenne=15)>,
    <Étudiant Damien (âge=12, moyenne=15)>,
    <Étudiant Clément (âge=14, moyenne=16)>,
    <Étudiant Oriane (âge=14, moyenne=18)>
]
```

On obtient la même chose que dans notre exercice précédent, quand nous utilisions des tuples. Je trouve personnellement cette méthode plus lisible.

# Trier dans l'ordre inverse

Il arrive souvent que l'on veuille trier dans l'ordre inverse. Par exemple, que l'on veuille trier nos étudiants par ordre inverse d'âge (du plus grand au plus petit).

Une solution est de trier et ensuite d'inverser la liste, mais là encore, il existe plus rapide : l'argument reverse.

C'est un argument booléen que l'on peut passer à la méthode de liste sort ou à la fonction sorted.

Essayons par exemple de trier nos étudiants par ordre inverse d'âge :

```
sorted(etudiants, key=lambda etudiant: etudiant.age, reverse=True)
>>>
[
    <Étudiant Clément (âge=14, moyenne=16)>,
    <Étudiant Oriane (âge=14, moyenne=18)>,
    <Étudiant Charles (âge=12, moyenne=15)>,
    <Étudiant Damien (âge=12, moyenne=15)>,
    <Étudiant Thomas (âge=11, moyenne=12)>
]
```

Plutôt simple, n'est-ce pas ?

# Plus rapide et plus efficace 1/2

Les méthodes de tri que nous avons vues jusqu'ici sont très pratiques. Leur plus grand inconvénient est de reposer sur des fonctions **lambdas**. Il est vrai que définir une **lambda** est rapide (et, une fois qu'on s'est habitué à la syntaxe, assez lisible). Par contre les fonctions **lambdas** ne sont pas le meilleur choix au niveau rapidité, si vous voulez trier une liste contenant beaucoup d'objets.

Mais tu as dit que le paramètre key attendait une fonction, ne peut-on définir une fonction « ordinaire » ?

Si. C'est tout à fait possible. Mais la plupart du temps, une des fonctions du module operator que nous allons voir fait très bien le travail.

## Les fonctions du module operator

Le module operator propose plusieurs fonctions qui vont s'avérer utiles pour nous, dans ce cas précis. Nous allons nous intéresser tout particulièrement aux fonctions **itemgetter** et **attrgetter**, mais sachez qu'il en existe d'autres et que le module operator n'est pas uniquement utile pour le tri, loin s'en faut.

## Trier une liste de tuples

D'abord, voyons notre exemple avec les tuples :

```
etudiants = [
    ("Clément", 14, 16),
    ("Charles", 12, 15),
    ("Oriane", 14, 18),
    ("Thomas", 11, 12),
    ("Damien", 12, 15),
]
```

Si on veut trier par moyenne ascendante, nous avons vu qu'il suffisait de faire :

```
sorted(etudiants, key=lambda etudiant: etudiant[2])
```

# Plus rapide et plus efficace 2/2

Pour faire la même chose sans fonction lambda, avec la fonction **itemgetter** du module operator :

```
from operator import itemgetter  
sorted(etudiants, key=itemgetter(2))
```

On appelle la fonction **itemgetter** avec le paramètre 2. Un objet operator.**itemgetter** est créé et passé au paramètre key de la fonction sorted. Ensuite, pour chaque étudiant contenu dans notre liste, l'objet operator.**itemgetter** est appelé et retourne la note moyenne de l'étudiant.

Au final, on obtient le même résultat qu'avec notre fonction lambda, mais cette méthode est plus rapide sur un grand nombre de données et, une fois qu'on s'est habitué à son aspect, plus facile à lire.

# Trier une liste d'objets 1/2

On peut faire la même chose si on parcourt une liste d'objets, mais cette fois, on utilise la fonction **attrgetter**. Je vous remet le code pour être sûr que vous avez le même que moi :

```
class Etudiant:
```

```
    """Classe représentant un étudiant.
```

```
    On représente un étudiant par son prénom (attribut prenom), son âge  
(attribut age) et sa note moyenne (attribut moyenne, entre 0 et 20).
```

```
    Paramètres du constructeur :
```

```
        prenom -- le prénom de l'étudiant
```

```
        age -- l'âge de l'étudiant
```

```
        moyenne -- la moyenne de l'étudiant
```

```
    """
```

```
def __init__(self, prenom, age, moyenne):
```

```
    self.prenom = prenom
```

```
    self.age = age
```

```
    self.moyenne = moyenne
```

```
def __repr__(self):
```

```
    return "<Étudiant {} (âge={}, moyenne={})>".format(
```

```
        self.prenom, self.age, self.moyenne)
```

```
etudiants = [
```

```
    Etudiant("Clément", 14, 16),
```

```
    Etudiant("Charles", 12, 15),
```

```
    Etudiant("Oriane", 14, 18),
```

```
    Etudiant("Thomas", 11, 12),
```

```
    Etudiant("Damien", 12, 15),
```

```
]
```

# Trier une liste d'objets 1/2

Et maintenant pour trier notre liste d'étudiants par note moyenne ascendante :

```
from operator import attrgetter  
sorted(etudiants, key=attrgetter("moyenne"))
```

Le système est le même, sauf que l'on travaille ici sur une liste d'objets et que le calcul est fait sur un attribut de l'objet (ici "moyenne") au lieu d'un tuple.

# Trier selon plusieurs critères

Trier selon un critère, c'est déjà très bien, mais trier selon plusieurs critères, ce peut être encore mieux. Si nous voulons, disons, trier nos étudiants par âge et note moyenne. C'est-à-dire que le tri se fera par âge, mais si deux étudiants ont le même âge, le tri se fera sur leur moyenne.  
La bonne nouvelle ? Rien de nouveau : passez juste un nouveau paramètre à la fonction attrgetter :

```
sorted(etudiants, key=attrgetter("age", "moyenne"))
>>>
[
    <Étudiant Thomas (âge=11, moyenne=12)>,
    <Étudiant Charles (âge=12, moyenne=15)>,
    <Étudiant Damien (âge=12, moyenne=15)>,
    <Étudiant Clément (âge=14, moyenne=16)>,
    <Étudiant Oriane (âge=14, moyenne=18)>
]
```

Vous avez peut-être remarqué que l'ordre de Charles et Damien dans la liste est identique à avant, même si d'autres étudiants ont changé de place : en effet, Charles et Damien ont le même âge et la même moyenne et leur ordre n'est pas modifié par Python.

Cette propriété est appelée « stabilité ». Si deux éléments de la séquence à comparer sont identiques, leur ordre est conservé.  
Cette propriété du tri en Python permet de chaîner nos tris.

# Chaînage de tris 1/3

Pour vous montrer un exemple concret, nous allons changer d'objets : nous allons travailler sur un inventaire de produits avec leur prix et quantité vendues.

```
class LigneInventaire:
```

"""Classe représentant une ligne d'un inventaire de vente.

Attributs attendus par le constructeur :

produit -- le nom du produit

prix -- le prix unitaire du produit

quantite -- la quantité vendue du produit.

"""

```
def __init__(self, produit, prix, quantite):
```

self.produit = produit

self.prix = prix

self.quantite = quantite

```
def __repr__(self):
```

return "<Ligne d'inventaire {} ({})X{}>".format(

self.produit, self.prix, self.quantite)

# Création de l'inventaire

```
inventaire = [
```

LigneInventaire("pomme rouge", 1.2, 19),

LigneInventaire("orange", 1.4, 24),

LigneInventaire("banane", 0.9, 21),

LigneInventaire("poire", 1.2, 24),

```
]
```

# Chaînage de tris 2/3

On veut trier cette liste par prix et par quantité. Facile, c'est ce qu'on a fait un peu plus haut :

```
from operator import attrgetter
print(sorted(inventaire, key=attrgetter("prix", "quantite")))
```

Ce qui vous renvoie :

```
>>>
[
    <Ligne d'inventaire banane (0.9X21)>,
    <Ligne d'inventaire pomme rouge (1.2X19)>,
    <Ligne d'inventaire poire (1.2X24)>,
    <Ligne d'inventaire orange (1.4X24)>
]
```

Mais si vous voulez trier par prix croissant et par quantité décroissante ? C'est-à-dire qu'on veut trier par prix croissant, mais que si deux lignes d'inventaires ont le même prix, alors on trie dans l'ordre décroissant de quantité ?

Le plus simple ici est de faire deux tris en utilisant la propriété de stabilité. La subtilité, c'est que l'on va trier d'abord par notre second critère et ensuite par notre premier. Ici, nous allons donc trier d'abord par ordre décroissant de quantité, puis ensuite par ordre croissant de prix.

Si vous vous demandez pourquoi, faites plusieurs essais (dans l'ordre que j'ai indiqué et dans l'ordre inverse). Si cela vous aide, essayez d'écrire l'inventaire sur une feuille et de trier dans un ordre et dans l'autre.

Voici le code pour notre tri. D'abord par quantité, ensuite par prix :

```
from operator import attrgetter
inventaire.sort(key=attrgetter("quantite"), reverse=True)
sorted(inventaire, key=attrgetter("prix"))
```

# Chaînage de tris 3/3

Et vous devriez obtenir :

```
from operator import attrgetter
sorted(inventaire, key=attrgetter("prix", "quantite"))
```

Ce qui vous renvoie :

```
>>>
[
    <Ligne d'inventaire banane (0.9X21)>,
    <Ligne d'inventaire pomme rouge (1.2X19)>,
    <Ligne d'inventaire poire (1.2X24)>,
    <Ligne d'inventaire orange (1.4X24)>
]
```

On utilise ici la méthode de liste sort comme on aurait pu utiliser la fonction sorted.

Regardez surtout l'ordre dans lequel la poire et la pomme rouge apparaissent : les deux lignes d'inventaire ont le même prix, mais puisque la poire a été vendue en plus grande quantité, elle apparaît en premier. Ceci n'aurait pas été possible sans la stabilité dans le tri.

Sans cette propriété, le second tri (par prix) aurait complètement modifié l'ordre de notre liste, rendant inutile notre premier tri (par quantité inverse).

Voilà pour ce tour d'horizon des méthodes de tri proposées par Python. Sachez que vous pourrez retrouver les fonctions clés (souvent en paramètre key d'une fonction) pour d'autres usages que le tri.

# Résumé

- Le tri en Python se fait grâce à la méthode de liste sort, qui modifie la liste d'origine, et la fonction sorted, qui ne modifie pas la liste (ou la séquence) passée en paramètre ;
- On peut spécifier des fonctions clés grâce à l'argument key. Ces fonctions sont appelées pour chaque élément de la séquence à trier, et retournent le critère du tri ;
- Le module operator propose les fonctions itemgetter et attrgetter qui peuvent être très utiles en tant que fonction clés, si on veut trier une liste de tuples ou une liste d'objets selon un attribut ;
- Le tri en Python est « stable », c'est-à-dire que l'ordre de deux éléments dans la liste n'est pas modifié s'ils sont égaux. Cette propriété permet le chaînage de tri.

# Découvrez la boucle for

# Découvrez la boucle for 1/12

## Découvrez la boucle for

Voilà pas mal de chapitres, nous avons étudié les boucles. Ne vous alarmez pas, ce que nous avons vu est toujours d'actualité ... mais nous allons un peu approfondir le sujet, maintenant que nous explorons le monde de l'objet.

Nous allons ici parler d'itérateurs et de générateurs. Nous allons découvrir ces concepts du plus simple au plus complexe et de telle sorte que chacun des concepts abordés reprenne les précédents. N'hésitez pas, par la suite, à revenir sur ce chapitre et à le relire, partiellement ou intégralement si nécessaire.

# Les itérateurs 2/12

Nous utilisons des itérateurs sans le savoir depuis le moment où nous avons abordé les boucles et surtout, depuis que nous utilisons le mot-clé `for` pour parcourir des objets conteneurs.

```
ma_liste = [1, 2, 3]
for element in ma_liste:
```

## Utiliser les itérateurs

C'est sur la seconde ligne que nous allons nous attarder : à force d'utiliser ce type de syntaxe, vous avez dû vous y habituer et ce type de parcours doit vous être familier. Mais il se cache bel et bien un mécanisme derrière cette instruction.

Quand Python tombe sur une ligne du type `for element in ma_liste:`, il va appeler l'itérateur de `ma_liste`. L'itérateur, c'est un objet qui va être chargé de parcourir l'objet conteneur, ici une liste.

L'itérateur est créé dans la méthode spéciale `__iter__` de l'objet. Ici, c'est donc la méthode `__iter__` de la classe `list` qui est appelée et qui renvoie un itérateur permettant de parcourir la liste.

À chaque tour de boucle, Python appelle la méthode spéciale `__next__` de l'itérateur, qui doit renvoyer l'élément suivant du parcours ou lever l'exception `StopIteration` si le parcours touche à sa fin.

Ce n'est peut-être pas très clair... alors voyons un exemple.

Avant de plonger dans le code, sachez que Python utilise deux fonctions pour appeler et manipuler les itérateurs : `iter` permet d'appeler la méthode spéciale `__iter__` de l'objet passé en paramètre et `next` appelle la méthode spéciale `__next__` de l'itérateur passé en paramètre.

# Les itérateurs 3/12

## Utiliser les itérateurs

C'est sur la seconde ligne que nous allons nous attarder : à force d'utiliser ce type de syntaxe, vous avez dû vous y habituer et ce type de parcours doit vous être familier. Mais il se cache bel et bien un mécanisme derrière cette instruction.

Quand Python tombe sur une ligne du type `for element in ma_liste:`, il va appeler l'itérateur de `ma_liste`. L'itérateur, c'est un objet qui va être chargé de parcourir l'objet conteneur, ici une liste.

L'itérateur est créé dans la méthode spéciale `__iter__` de l'objet. Ici, c'est donc la méthode `__iter__` de la classe `list` qui est appelée et qui renvoie un itérateur permettant de parcourir la liste.

À chaque tour de boucle, Python appelle la méthode spéciale `__next__` de l'itérateur, qui doit renvoyer l'élément suivant du parcours ou lever l'exception `StopIteration` si le parcours touche à sa fin.

Ce n'est peut-être pas très clair... alors voyons un exemple.

Avant de plonger dans le code, sachez que Python utilise deux fonctions pour appeler et manipuler les itérateurs : `iter` permet d'appeler la méthode spéciale `__iter__` de l'objet passé en paramètre et `next` appelle la méthode spéciale `__next__` de l'itérateur passé en paramètre.

```
ma_chaine = "test"
ite = iter(ma_chaine)
print(ite)
print(next(ite))
print(next(ite))
print(next(ite))
print(next(ite))
print(next(ite))
<str_iterator object at 0x0000021D4BC58AC0>
t
e
s
t
Traceback (most recent call last):
File "C:\Users\PALLEAU JULIEN\Documents\GitHub\TheCompletePythonCourse\test.py", line 53, in <module>
    print(next(ite))
StopIteration
```

# Les itérateurs 4/12

On commence par créer une chaîne de caractères (jusque là, rien de compliqué).

On appelle ensuite la fonction `iter` en lui passant en paramètre la chaîne. Cette fonction appelle la méthode spéciale `__iter__` de la chaîne, qui renvoie l'itérateur permettant de parcourir `ma_chaine`.

On va ensuite appeler plusieurs fois la fonction `next` en lui passant en paramètre l'itérateur. Cette fonction appelle la méthode spéciale `__next__` de l'itérateur. Elle renvoie successivement chaque lettre contenue dans notre chaîne et lève une exception `StopIteration` quand la chaîne a été parcourue entièrement.

Quand on parcourt une chaîne grâce à une boucle `for` (`for lettre in chaine:`), c'est ce mécanisme d'itérateur qui est appelé. Chaque lettre renvoyée par notre itérateur se retrouve dans la variable `lettre` et la boucle s'arrête quand l'exception `StopIteration` est levée.

Vous pouvez reprendre ce code avec d'autres objets conteneurs, des listes par exemple.

# Les itérateurs 5/12

```
class RevStr(str):  
    """Classe reprenant les méthodes et attributs des chaînes construites  
depuis 'str'. On se contente de définir une méthode de parcours  
différente : au lieu de parcourir la chaîne de la première à la dernière  
lettre, on la parcourt de la dernière à la première.
```

*Les autres méthodes, y compris le constructeur, n'ont pas besoin  
d'être redéfinies""""*

```
def __iter__(self):  
    """Cette méthode renvoie un itérateur parcourant la chaîne  
dans le sens inverse de celui de 'str'"""""  
  
    return ItRevStr(self) # On renvoie l'itérateur créé pour l'occasion
```

```
class ItRevStr:  
    """Un itérateur permettant de parcourir une chaîne de la dernière lettre  
à la première. On stocke dans des attributs la position courante et la  
chaîne à parcourir"""""
```

```
def __init__(self, chaine_a_parcourir):  
    """On se positionne à la fin de la chaîne"""""  
    self.chaine_a_parcourir = chaine_a_parcourir  
    self.position = len(chaine_a_parcourir)
```

```
def __next__(self):  
    """Cette méthode doit renvoyer l'élément suivant dans le parcours,  
ou lever l'exception 'StopIteration' si le parcours est fini""""`
```

```
if self.position == 0: # Fin du parcours  
    raise StopIteration  
self.position -= 1 # On décrémente la position  
return self.chaine_a_parcourir[self.position]
```

# Les itérateurs 6/12

À présent, vous pouvez créer des chaînes devant se parcourir du dernier caractère vers le premier.

```
ma_chaine = RevStr("Bonjour")
ma_chaine
>>> 'Bonjour'
for lettre in ma_chaine:
    print(lettre)

>>> r
>>> u
>>> o
>>> j
>>> n
>>> o
>>> B
```

Sachez qu'il est aussi possible de mettre en œuvre directement la méthode `__next__` dans notre objet conteneur. Dans ce cas, la méthode `__iter__` pourra renvoyer `self`. Vous pouvez voir un exemple, dont le code ci-dessus est inspiré, sur la documentation de Python.

Cela reste quand même plutôt lourd non, de devoir faire des itérateurs à chaque fois ? Surtout si nos objets conteneurs doivent se parcourir de plusieurs façons, comme les dictionnaires par exemple.

Oui, il subsiste quand même beaucoup de répétitions dans le code que nous devons produire, surtout si nous devons créer plusieurs itérateurs pour un même objet. Souvent, on utilisera des itérateurs existants, par exemple celui des listes. Mais il existe aussi un autre mécanisme, plus simple et plus intuitif : la raison pour laquelle je ne vous montre pas en premier cette autre façon de faire, c'est que cette autre façon passe quand même par des itérateurs, même si c'est implicite, et qu'il n'est pas mauvais de savoir comment cela marche en coulisse.

Il est temps à présent de jeter un coup d'œil du côté des générateurs.

# Les générateurs 7/12

Les générateurs sont avant tout un moyen plus pratique de créer et manipuler des itérateurs. Vous verrez un peu plus loin dans ce chapitre qu'ils permettent des choses assez complexes, mais leur puissance tient surtout à leur simplicité et leur petite taille.

## Les générateurs simples

Pour créer des générateurs, nous allons découvrir un nouveau mot-clé : **yield**. Ce mot-clé ne peut s'utiliser que dans le corps d'une fonction et il est suivi d'une valeur à renvoyer.

Attends un peu... une valeur ? À renvoyer ?

Oui. Le principe des générateurs étant un peu particulier, il nécessite un mot-clé pour lui tout seul. L'idée consiste à définir une fonction pour un type de parcours. Quand on demande le premier élément du parcours (grâce à `next`), la fonction commence son exécution. Dès qu'elle rencontre une instruction **yield**, elle renvoie la valeur qui suit et se met en pause. Quand on demande l'élément suivant de l'objet (grâce, une nouvelle fois, à `next`), l'exécution reprend à l'endroit où elle s'était arrêtée et s'interrompt au **yield** suivant... et ainsi de suite. À la fin de l'exécution de la fonction, l'exception `StopIteration` est automatiquement levée par Python.

Nous allons prendre un exemple très simple pour commencer :

```
def mon_générateur():
    """Notre premier générateur. Il va simplement renvoyer 1, 2 et 3"""
    yield 1
    yield 2
    yield 3

print(mon_générateur)

print(mon_générateur())

mon_iterateur = iter(mon_générateur())
print(next(mon_iterateur))
print(next(mon_iterateur))
print(next(mon_iterateur))
print(next(mon_iterateur))

>>> <function mon_générateur at 0x00000295CF9DCF70>
>>> <generator object mon_générateur at 0x00000295CF9DB660>
>>> 1
>>> 2
>>> 3
```

```
>>> Traceback (most recent call last):
      File "C:\Users\MOTTIER LUCIE\Documents\GitHub\TheCompletePythonCourse\test.py", line 60, in <module>
        print(next(mon_iterateur))
      StopIteration
```

# Les générateurs 8/12

Je pense que cela vous rappelle quelque chose ! Cette fonction, à part l'utilisation de **yield**, est plutôt classique. Quand on l'exécute, on se retrouve avec un générateur. Ce générateur est un objet créé par Python qui définit sa propre méthode spéciale **\_\_iter\_\_** et donc son propre itérateur. Nous aurions tout aussi bien pu faire :

```
for nombre in mon_generateur(): # Attention on exécute la fonction
    print(nombre)
```

Cela rend quand même le code bien plus simple à comprendre.

Notez qu'on doit exécuter la fonction `mon_generateur` pour obtenir un générateur. Si vous essayez de parcourir notre fonction (`for nombre in mon_generateur`), cela ne marchera pas.

Bien entendu, la plupart du temps, on ne se contentera pas d'appeler **yield** comme cela. Le générateur de notre exemple n'a pas beaucoup d'intérêt, il faut bien le reconnaître.

Essayons de faire une chose un peu plus utile : un générateur prenant en paramètres deux entiers, une borne inférieure et une borne supérieure, et renvoyant chaque entier compris entre ces bornes. Si on écrit par exemple `interval(5, 10)`, on pourra parcourir les entiers de 6 à 9.

Le résultat attendu est donc :

```
>>> for nombre in interval(5, 10):
...     print(nombre)
...
6
7
8
9
```

Vous pouvez essayer de faire l'exercice, c'est un bon entraînement et pas très compliqué de surcroît.

Au cas où, voici la correction :

```
def intervalle(borne_inf, borne_sup):
    """Générateur parcourant la série des entiers entre borne_inf et borne_sup.
```

Note: `borne_inf` doit être inférieure à `borne_sup`""""

```
borne_inf += 1
while borne_inf < borne_sup:
    yield borne_inf
    borne_inf += 1
```

# Les générateurs 9/12

Là encore, vous pouvez améliorer cette fonction. Pourquoi ne pas faire en sorte que, si la borne inférieure est supérieure à la borne supérieure, le parcours se fasse dans l'autre sens ?

L'important est que vous compreniez bien l'intérêt et le mécanisme derrière. Je vous encourage, là encore, à tester, à disséquer cette fonctionnalité, à essayer de reprendre les exemples d'itérateurs et à les convertir en générateurs.

Si, dans une classe quelconque, la méthode spéciale `__iter__` contient un appel à `yield`, alors ce sera ce générateur qui sera appelé quand on voudra parcourir la boucle. Même quand Python passe par des générateurs, comme vous l'avez vu, il utilise (implicitement) des itérateurs. C'est juste plus confortable pour le codeur, on n'a pas besoin de créer une classe par itérateur ni de coder une méthode `__next__`, ni même de lever l'exception `StopIteration` : Python fait tout cela pour nous. Pratique non ?

## Les générateurs comme co-routines

Jusqu'ici, que ce soit avec les itérateurs ou avec les générateurs, nous créons un moyen de parcourir notre objet au début de la boucle `for`, en sachant que nous ne pourrons pas modifier le comportement du parcours par la suite. Mais les générateurs possèdent un certain nombre de méthodes permettant, justement, d'interagir avec eux pendant le parcours.

Malheureusement, à notre niveau, les idées d'applications utiles me manquent et je vais me contenter de vous présenter la syntaxe et un petit exemple. Peut-être trouverez-vous par la suite une application utile des co-routines quand vous vous lancerez dans des programmes conséquents, ou que vous aurez été plus loin dans l'apprentissage du Python.

Les co-routines sont un moyen d'altérer le parcours... pendant le parcours. Par exemple, dans notre générateur `intervalle`, on pourrait vouloir passer directement de 5 à 10.

Le système des co-routines en Python est contenu dans le mot-clé `yield` que nous avons vu plus haut et l'utilisation de certaines méthodes de notre générateur.

### Interrompre la boucle

La première méthode que nous allons voir est `close`. Elle permet d'interrompre prématurément la boucle, comme le mot-clé `break` en somme.

# Les générateurs comme co-routines

## 10/12

### Les générateurs comme co-routines

Jusqu'ici, que ce soit avec les itérateurs ou avec les générateurs, nous créons un moyen de parcourir notre objet au début de la boucle for, en sachant que nous ne pourrons pas modifier le comportement du parcours par la suite. Mais les générateurs possèdent un certain nombre de méthodes permettant, justement, d'interagir avec eux pendant le parcours.

Malheureusement, à notre niveau, les idées d'applications utiles me manquent et je vais me contenter de vous présenter la syntaxe et un petit exemple. Peut-être trouverez-vous par la suite une application utile des co-routines quand vous vous lancerez dans des programmes conséquents, ou que vous aurez été plus loin dans l'apprentissage du Python.

Les co-routines sont un moyen d'altérer le parcours... pendant le parcours. Par exemple, dans notre générateur intervalle, on pourrait vouloir passer directement de 5 à 10.

Le système des co-routines en Python est contenu dans le mot-clé yield que nous avons vu plus haut et l'utilisation de certaines méthodes de notre générateur.

#### Interrompre la boucle

La première méthode que nous allons voir est close. Elle permet d'interrompre prématurément la boucle, comme le mot-clé break en somme.

```
generateur = intervalle(5, 20)
for nombre in generateur:
    if nombre > 17:
        generateur.close() # Interruption de la boucle
```

Comme vous le voyez, pour appeler les méthodes du générateur, on doit le stocker dans une variable avant la boucle. Si vous aviez écrit directement for nombre in intervalle(5, 20), vous n'auriez pas pu appeler la méthode close du générateur.

# Les générateurs comme co-routines

## 11/12

### Envoyer des données à notre générateur

Pour cet exemple, nous allons étendre notre générateur pour qu'il accepte de recevoir des données pendant son exécution.

Le point d'échange de données se fait au mot-clé `yield`. `yield` valeur « renvoie » valeur qui deviendra donc la valeur courante du parcours. La fonction se met ensuite en pause. On peut, à cet instant, envoyer une valeur à notre générateur. Cela permet d'altérer le fonctionnement de notre générateur pendant le parcours.

Reprendons notre exemple en intégrant cette fonctionnalité :

```
def intervalle(borne_inf, borne_sup):
    """Générateur parcourant la série des entiers entre borne_inf et borne_sup.
    Notre générateur doit pouvoir "sauter" une certaine plage de nombres
    en fonction d'une valeur qu'on lui donne pendant le parcours. La
    valeur qu'on lui passe est la nouvelle valeur de borne_inf.

    Note: borne_inf doit être inférieure à borne_sup"""
    borne_inf += 1
    while borne_inf < borne_sup:
        valeur_recue = (yield borne_inf)
        if valeur_recue is not None: # Notre générateur a reçu quelque chose
            borne_inf = valeur_recue
        borne_inf += 1
```

# Les générateurs comme co-routines

## 12/12

Nous configurons notre générateur pour qu'il accepte une valeur éventuelle au cours du parcours. S'il reçoit une valeur, il va l'attribuer au point du parcours.

Autrement dit, au cours de la boucle, vous pouvez demander au générateur de sauter tout de suite à 20 si le nombre est 15.

Tout se passe à partir de la ligne du **yield**. Au lieu de simplement renvoyer une valeur à notre boucle, on capture une éventuelle valeur dans `valeur_recue`. La syntaxe est simple : `variable = (yield valeur_a_renvoyer)`. N'oubliez pas les parenthèses autour de **yield** valeur.

Si aucune valeur n'a été passée à notre générateur, notre `valeur_recue` vaudra `None`. On vérifie donc si elle ne vaut pas `None` et, dans ce cas, on affecte la nouvelle valeur à `borne_inf`.

Voici le code permettant d'interagir avec notre générateur. On utilise la méthode `send` pour envoyer une valeur à notre générateur :

```
generateur = intervalle(10, 25)
for nombre in generateur:
    if nombre == 15: # On saute à 20
        generateur.send(20)
    print(nombre, end=" ")
```

Il existe d'autres méthodes permettant d'interagir avec notre générateur. Vous pouvez les retrouver, ainsi que des explications supplémentaires, sur la documentation officielle traitant du mot-clé `yield`.

# Résumé

- Quand on utilise la boucle `for element in sequence:`, un itérateur de cette séquence permet de la parcourir.

On peut récupérer l'itérateur d'une séquence grâce à la fonction `iter`.

- Une séquence renvoie l'itérateur permettant de la parcourir grâce à la méthode spéciale `__iter__`.
- Un itérateur possède une méthode spéciale, `__next__`, qui renvoie le prochain élément à parcourir ou lève l'exception `StopIteration` qui arrête la boucle.
- Les générateurs permettent de créer plus simplement des itérateurs.
- Ce sont des fonctions utilisant le mot-clé `yield` suivi de la valeur à transmettre à la boucle.



# 5 astuces python 1/5

Vérification du code de réponse du requête web (on imagine que ces 4 lignes de codes sont dans une fonction):

```
if r.status_code == 200:  
    return True  
else:  
    return False
```

Ceci peut être optimisé comme suit:

```
return r.status_code == 200(car l'évaluation de l'expression est déjà un booléen !)
```

# 5 astuces python 2/5

```
if user_solution.is_correct:  
    progress += 1
```

On va tirer partie du fait que True et False ont une valeur:

True -> 1  
False -> 0

```
progress += user_solution.is_correct
```

# 5 astuces python 3/5

```
default_value = 5
user_input = input("Entrer un nombre: ")
if user_input:
    value = user_input
else:
    value = default_value
```

On peut utiliser un opérateur ternaire

```
value = user_input if user_input else default_value
```

On peut faire encore plus simple, à la place d'utiliser un opérateur ternaire on va utiliser un opérateur or.

```
value = user_input or default_value
```

# 5 astuces python 4/5

```
if country == "fr" or country == "en" or country == "it":  
    return True  
else:  
    return False
```

On peut utiliser une liste

```
if country in ["fr", "en", "it"]  
    return True
```

On peut faire encore plus simple, comme pour la première astuce:

```
return country in ['fr', 'en', 'it']
```

# 5 astuces python 5/5

```
if event == "payment":  
    process_payment()  
elif event == "subscribe":  
    subscribe_user()  
elif event == "delete":  
    delete_user()  
elif event == "invoice":  
    send_invoice()
```

On peut utiliser un dictionnaire

```
d = {"payment": process_payment,  
      "subscribe": subscribe_user,  
      "delete": delete_user,  
      "invoice": send_invoice  
      }  
  
func = d.get(event, default)  
func()
```

# Apprehendez les decorateurs

<https://blog.teclado.com/decorators-in-python/>



How to write decorators in Python.pdf

# Appréhendez les décorateurs

Nous allons ici nous intéresser à un concept fascinant de Python, un concept de programmation assez avancé. Vous n'êtes pas obligés de lire ce chapitre pour la suite de ce livre, ni même connaître cette fonctionnalité pour coder en Python. Il s'agit d'un plus que j'ai voulu détailler mais qui n'est certainement pas indispensable.

Les décorateurs sont un moyen simple de modifier le comportement « par défaut » de fonctions. C'est un exemple assez flagrant de ce qu'on appelle la métaprogrammation, que je vais décrire assez brièvement comme l'écriture de programmes manipulant... d'autres programmes. Cela donne faim, non ?

Qu'est-ce que c'est ?

Les décorateurs sont des fonctions de Python dont le rôle est de modifier le comportement par défaut d'autres fonctions ou classes. Pour schématiser, une fonction modifiée par un décorateur ne s'exécutera pas elle-même mais appellera le décorateur. C'est au décorateur de décider s'il veut exécuter la fonction et dans quelles conditions.

Mais quel est l'intérêt ? Si on veut juste qu'une fonction fasse quelque chose de différent, il suffit de la modifier, non ? Pourquoi s'encombrer la tête avec une nouvelle fonctionnalité plus complexe ?

Il peut y avoir de nombreux cas dans lesquels les décorateurs sont un choix intéressant. Pour comprendre l'idée, je vais prendre un unique exemple.

On souhaite tester les performances de certaines de nos fonctions, en l'occurrence, calculer combien de temps elles mettent pour s'exécuter.

Une possibilité, effectivement, consiste à modifier chacune des fonctions devant intégrer ce test. Mais ce n'est pas très élégant, ni très pratique, ni très sûr... bref ce n'est pas la meilleure solution.

Une autre possibilité consiste à utiliser un décorateur. Ce décorateur se chargera d'exécuter notre fonction en calculant le temps qu'elle met et pourra, par exemple, afficher une alerte si cette durée est trop élevée.

Pour indiquer qu'une fonction doit intégrer ce test, il suffira d'ajouter une simple ligne avant sa définition. C'est bien plus simple, clair et adapté à la situation.

Et ce n'est qu'un exemple d'application.

Les décorateurs sont des fonctions standard de Python mais leur construction est parfois complexe. Quand il s'agit de décorateurs prenant des arguments en paramètres ou devant tenir compte des paramètres de la fonction, le code est plus complexe, moins intuitif.

Je vais faire mon possible pour que vous compreniez bien le principe. N'hésitez pas à y revenir à tête reposée, une, deux, trois fois pour que cela soit bien clair.

# En Théorie

Une fois n'est pas coutume, je vais vous montrer les différentes constructions possibles en théorie avec quelques exemples, mais je vais aussi consacrer une section entière à des exemples d'utilisations pour expliciter cette partie théorique indispensable.

Format le plus simple

Comme je l'ai dit, les décorateurs sont des fonctions « classiques » de Python, dans leur définition. Ils ont une petite subtilité en ce qu'ils prennent en paramètre une fonction et renvoient une fonction.

On déclare qu'une fonction doit être modifiée par un (ou plusieurs) décorateurs grâce à une (ou plusieurs) lignes au-dessus de la définition de fonction, comme ceci :

```
@nom_du_decorateur  
def ma_fonction(...)
```

Le décorateur s'exécute au moment de la définition de fonction et non lors de l'appel. Ceci est important. Il prend en paramètre, comme je l'ai dit, une fonction (celle qu'il modifie) et renvoie une fonction (qui peut être la même).

Voyez plutôt :

```
def mon_decorateur(fonction):  
    """Premier exemple de décorateur"""\n    print("Notre décorateur est appelé avec en paramètre la fonction {0}".format(fonction))\n    return fonction
```

```
@mon_decorateur  
def salut():  
    """Fonction modifiée par notre décorateur"""\n    print("Salut !")
```

```
>>> Notre décorateur est appelé avec en paramètre la fonction salut at 0x00BA5198>
```

Euh... qu'est-ce qu'on a fait là ?

# En Théorie

D'abord, on crée le décorateur. Il prend en paramètre, comme je vous l'ai dit, la fonction qu'il modifie. Dans notre exemple, il se contente d'afficher cette fonction puis de la renvoyer.

On crée ensuite la fonction `salut`. Comme vous le voyez, on indique avant la définition la ligne `@mon_décorateur`, qui précise à Python que cette fonction doit être modifiée par notre décorateur. Notre fonction est très utile : elle affiche « Salut ! » et c'est tout.

À la fin de la définition de notre fonction, on peut voir que le décorateur est appelé. Si vous regardez plus attentivement la ligne affichée, vous vous rendez compte qu'il est appelé avec, en paramètre, la fonction `salut` que nous venons de définir.

Intéressons-nous un peu plus à la structure de notre décorateur. Il prend en paramètre la fonction à modifier (celle que l'on définit sous la ligne `du @`), je pense que vous avez pu le constater. Mais il renvoie également cette fonction et cela, c'est un peu moins évident !

En fait, la fonction renvoyée remplace la fonction définie. Ici, on renvoie la fonction définie, c'est donc la même. Mais on peut demander à Python d'exécuter une autre fonction à la place, pour modifier son comportement. Nous allons voir cela un peu plus loin.

Pour l'heure, souvenez-vous que les deux codes ci-dessous sont identiques :

```
# Exemple avec décorateur
@decorateur
def fonction(...):
    ...
```

```
# Exemple équivalent, sans décorateur
def fonction(...):
    ...
fonction = decorateur(fonction)
```

# En Théorie

Relisez bien ces deux codes, ils font la même chose. Le second est là pour que vous compreniez ce que fait Python quand il manipule des fonctions modifiées par un (ou plusieurs) décorateur(s).

Quand vous exécutez salut, vous ne voyez aucun changement. Et c'est normal puisque nous renvoyons la même fonction. Le seul moment où notre décorateur est appelé, c'est lors de la définition de notre fonction. Notre fonction salut n'a pas été modifiée par notre décorateur, on s'est contenté de la renvoyer telle quelle.

## Modifier le comportement de notre fonction

Vous l'aurez deviné, un décorateur comme nous l'avons créé plus haut n'est pas bien utile. Les décorateurs servent surtout à modifier le comportement d'une fonction. Je vous montre cependant pas à pas comment cela fonctionne, sinon vous risquez de vite vous perdre.

Comment faire pour modifier le comportement de notre fonction ?

En fait, vous avez un élément de réponse un peu plus haut. J'ai dit que notre décorateur prenait en paramètre la fonction définie et renvoyait une fonction (peut-être la même, peut-être une autre). C'est cette fonction renvoyée qui sera directement affectée à notre fonction définie. Si vous aviez renvoyé une autre fonction que salut, dans notre exemple ci-dessus, la fonction salut aurait redirigé vers cette fonction renvoyée.

Mais alors... il faut définir encore une fonction ?

Eh oui ! Je vous avais prévenus (et ce n'est que le début), notre construction se complexifie au fur et à mesure : on va devoir créer une nouvelle fonction qui sera chargée de modifier le comportement de la fonction définie. Et, parce que notre décorateur sera le seul à utiliser cette fonction, on va la définir directement dans le corps de notre décorateur.

Je suis perdu. Comment cela marche-t-il, concrètement ?

Je vais vous mettre le code, cela vaudra mieux que des tonnes d'explications. Je le commente un peu plus bas, ne vous inquiétez pas :

# En Théorie

```
def mon_decorateur(fonction):
    """Notre décorateur : il va afficher un message avant l'appel de la
fonction définie"""

def fonction_modifiee():
    """Fonction que l'on va renvoyer. Il s'agit en fait d'une version
un peu modifiée de notre fonction originellement définie. On se
contente d'afficher un avertissement avant d'exécuter notre fonction
originellement définie"""

    print("Attention ! On appelle {0}".format(fonction))
    return fonction()

return fonction_modifiee

@mon_decorateur
def salut():
    print("Salut !")
```

Voyons l'effet, avant les explications. Aucun message ne s'affiche en exécutant ce code. Par contre, si vous exécutez votre fonction salut:

```
salut()
>>> Attention ! On appelle <function salut at 0x00BA54F8>
Salut !
```

Et si vous affichez la fonction salut dans l'interpréteur, vous obtenez quelque chose de surprenant :

```
salut
<function fonction_modifiee at 0x00BA54B0>
```

# En Théorie

Pour comprendre, revenons sur le code de notre décorateur :

Comme toujours, il prend en paramètre une fonction. Cette fonction, quand on place l'appel au décorateur au-dessus de def salut, c'est salut (la fonction définie à l'origine).

Dans le corps même de notre décorateur, vous pouvez voir qu'on a défini une nouvelle fonction, fonction\_modifiee. Elle ne prend aucun paramètre, elle n'en a pas besoin. Dans son corps, on affiche une ligne avertissant qu'on va exécuter la fonction fonction(là encore, il s'agit de salut). À la ligne suivante, on l'exécute effectivement et on renvoie le résultat de son exécution (dans le cas de salut, il n'y en a pas mais d'autres fonctions pourraient renvoyer des informations).

De retour dans notre décorateur, on indique qu'il faut renvoyer fonction\_modifiee.

Lors de la définition de notre fonction salut, on appelle notre décorateur. Python lui passe en paramètre la fonction salut. Cette fois, notre décorateur ne renvoie pas salut mais fonction\_modifiee. Et notre fonction salut, que nous venons de définir, sera donc remplacée par notre fonction fonction\_modifiee, définie dans notre décorateur.

Vous le voyez bien, d'ailleurs : quand on cherche à afficher salut dans l'interpréteur, on obtient fonction\_modifiee.

Souvenez-vous bien que le code :

```
@mon_decorateur  
def salut():  
    ...
```

revient au même, pour Python, que le code :

```
def salut():  
    ...  
  
salut = mon_decorateur(salut)
```

# En Théorie

Ce n'est peut-être pas plus clair. Prenez le temps de lire et de bien comprendre l'exemple. Ce n'est pas simple, la logique est bel et bien là mais il faut passer un certain temps à tester avant de bien intégrer cette notion.

Pour résumer, notre décorateur renvoie une fonction de substitution. Quand on appelle salut, on appelle en fait notre fonction modifiée qui appelle également salut après avoir affiché un petit message d'avertissement.

Autre exemple : un décorateur chargé tout simplement d'empêcher l'exécution de la fonction. Au lieu d'exécuter la fonction d'origine, on lève une exception pour avertir l'utilisateur qu'il utilise une fonctionnalité obsolète.

```
def obsolete(fonction_origine):
    """Décorateur levant une exception pour noter que la fonction_origine
    est obsolète"""

    def fonction_modifiee():
        raise RuntimeError("la fonction {0} est obsolète !".format(fonction_origine))

    return fonction_modifiee
```

Là encore, faites quelques essais : tout deviendra limpide après quelques manipulations.

## Un décorateur avec des paramètres

Toujours plus dur ! On voudrait maintenant passer des paramètres à notre décorateur. Nous allons essayer de coder un décorateur chargé d'exécuter une fonction en contrôlant le temps qu'elle met à s'exécuter. Si elle met un temps supérieur à la durée passée en paramètre du décorateur, on affiche une alerte.

La ligne appelant notre décorateur, au-dessus de la définition de notre fonction, sera donc sous la forme :

```
@controler_temps(2.5) # 2,5 secondes maximum pour la fonction ci-dessous
```

# En Théorie 1/6

Jusqu'ici, nos décorateurs ne comportaient aucune parenthèse après leur appel. Ces deux parenthèses sont très importantes : notre fonction de décorateur prendra en paramètres non pas une fonction, mais les paramètres du décorateur (ici, le temps maximum autorisé pour la fonction). Elle ne renverra pas une fonction de substitution, mais un décorateur.

Encore et toujours perdu. Pourquoi est-ce si compliqué de passer des paramètres à notre décorateur ?

En fait... ce n'est pas si compliqué que cela mais c'est dur à saisir au début. Pour mieux comprendre, essayez encore une fois de vous souvenir que ces deux codes reviennent au même :

```
@decorateur  
def fonction(...):  
    ...
```

```
def fonction(...):  
    ...  
  
fonction = decorateur(fonction)
```

Là encore, faites quelques essais : tout deviendra limpide après quelques manipulations.

C'est la dernière ligne du second exemple que vous devez retenir et essayer de comprendre :fonction = decorateur(fonction).

On remplace la fonction que nous avons définie au-dessus par la fonction que renvoie notre décorateur.

C'est le mécanisme qui se cache derrière notre@decorateur.

Maintenant, si notre décorateur attend des paramètres, on se retrouve avec une ligne comme celle-ci :

```
@decorateur(parametre)  
def fonction(...):  
    ...
```

# En Théorie 2/6

Et si vous avez compris l'exemple ci-dessus, ce code revient au même que :

```
@decorateur  
def fonction(...):  
    ...
```

Et si vous avez compris l'exemple ci-dessus, ce code revient au même que :

```
def fonction(...):  
    ...  
  
fonction = decorateur(fonction)
```

Je vous avais prévenus, ce n'est pas très intuitif ! Mais relisez bien ces exemples, le déclic devrait se faire tôt ou tard.

Comme vous le voyez, on doit définir comme décorateur une fonction qui prend en arguments les paramètres du décorateur (ici, le temps attendu) et qui renvoie un décorateur. Autrement dit, on se retrouve encore une fois avec un niveau supplémentaire dans notre fonction.

Je vous donne le code sans trop insister. Si vous arrivez à comprendre la logique qui se trouve derrière, c'est tant mieux, sinon n'hésitez pas à y revenir plus tard :

```
"""Pour gérer le temps, on importe le module time  
On va utiliser surtout la fonction time() de ce module qui renvoie le nombre  
de secondes écoulées depuis le premier janvier 1970 (habituellement).  
On va s'en servir pour calculer le temps mis par notre fonction pour  
s'exécuter"""
```

```
import time  
  
def controler_temps(nb_secs):  
    """Contrôle le temps mis par une fonction pour s'exécuter.  
    Si le temps d'exécution est supérieur à nb_secs, on affiche une alerte"""  
  
    def decorateur(fonction_a_executer):  
        """Notre décorateur. C'est lui qui est appelé directement LORS  
        DE LA DEFINITION de notre fonction (fonction_a_executer)"""
```

# En Théorie 3/6

```
def fonction_modifiee():
    """Fonction renvoyée par notre décorateur. Elle se charge
    de calculer le temps mis par la fonction à s'exécuter"""

    tps_avant = time.time() # Avant d'exécuter la fonction
    valeur_renvoyee = fonction_a_executer() # On exécute la fonction
    tps_apres = time.time()
    tps_execution = tps_apres - tps_avant
    if tps_execution >= nb_secs:
        print("La fonction {0} a mis {1} pour s'exécuter".format( \
            fonction_a_executer, tps_execution))
    return valeur_renvoyee
return fonction_modifiee
return decorateur
```

# En Théorie 4/6

Ouf ! Trois niveaux dans notre fonction ! D'abord controler\_temps, qui définit dans son corps notre décorateur decorateur, qui définit lui-même dans son corps notre fonction modifiée fonction\_modifiee.

J'espère que vous n'êtes pas trop embrouillés. Je le répète, il s'agit d'une fonctionnalité très puissante mais qui n'est pas très intuitive quand on n'y est pas habitué. Jetez un coup d'œil du côté des exemples au-dessus si vous êtes un peu perdus.

Nous pouvons maintenant utiliser notre décorateur. J'ai fait une petite fonction pour tester qu'un message s'affiche bien si notre fonction met du temps à s'exécuter. Voyez plutôt :

```
@controler_temps(4)
def attendre():
    input("Appuyez sur Entrée...")

attendre() # Je vais appuyer sur Entrée presque tout de suite
>>> Appuyez sur Entrée...
attendre() # Cette fois, j'attends plus longtemps
>>> Appuyez sur Entrée...
>>> La fonction <function attendre at 0x00BA5810> a mis 4.14100003242 pour s'exécuter
```

Ça marche ! Et même si vous devez passer un peu de temps sur votre décorateur, vu ses différents niveaux, vous êtes obligés de reconnaître qu'il s'utilise assez simplement.

Il est quand même plus intuitif d'écrire :

```
@controler_temps(4)
def attendre(...)
```

Que :

```
def attendre(...):
    ...
attendre = controler_temps(4)(attendre)
```

# En Théorie 5/6

## Tenir compte des paramètres de notre fonction

Jusqu'ici, nous n'avons travaillé qu'avec des fonctions ne prenant aucun paramètre. C'est pourquoi notre fonction `fonction_modifiee` n'en prenait pas non plus.

Oui mais... tenir compte des paramètres, cela peut être utile. Sans quoi on ne pourrait construire que des décorateurs s'appliquant à des fonctions sans paramètre.

Il faut, pour tenir compte des paramètres de la fonction, modifier ceux de notre fonction `fonction_modifiee`. Là encore, je vous invite à regarder les exemples ci-dessus, explicitant ce que Python fait réellement lorsqu'on définit un décorateur avant une fonction. Vous pourrez vous rendre compte que `fonction_modifiee` remplace notre fonction et que, par conséquent, elle doit prendre des paramètres si notre fonction définie prend également des paramètres.

C'est dans ce cas en particulier que nous allons pouvoir réutiliser la notation spéciale pour nos fonctions attendant un nombre variable d'arguments. En effet, le décorateur que nous avons créé un peu plus haut devrait pouvoir s'appliquer à des fonctions ne prenant aucun paramètre, ou en prenant un, ou plusieurs... au fond, notre décorateur ne doit ni savoir combien de paramètres sont fournis à notre fonction, ni même s'en soucier.

Là encore, je vous donne le code adapté de notre fonction modifiée. Souvenez-vous qu'elle est définie dans notre décorateur, lui-même défini dans `controler_temps`(je ne vous remets que le code de `fonction_modifiee`).

```
def fonction_modifiee(*parametres_non_nommes, **parametres_nommes):
    """Fonction renvoyée par notre décorateur. Elle se charge
    de calculer le temps mis par la fonction à s'exécuter"""

    tps_avant = time.time() # avant d'exécuter la fonction
    ret = fonction_a_executer(*parametres_non_nommes, **parametres_nommes)
    tps_apres = time.time()
    tps_execution = tps_apres - tps_avant
    if tps_execution >= nb_secs:
        print("La fonction {0} a mis {1} pour s'exécuter".format( \
            fonction_a_executer, tps_execution))
    return ret
```

À présent, vous pouvez appliquer ce décorateur à des fonctions ne prenant aucun paramètre, ou en prenant un certain nombre, nommés ou non. Pratique, non ?

# En Théorie 6/6

## Des décorateurs s'appliquant aux définitions de classes

Vous pouvez également appliquer des décorateurs aux définitions de classes. Nous verrons un exemple d'application dans la section suivante. Au lieu de recevoir en paramètre la fonction, vous allez recevoir la classe.

```
def decorateur(classe):
    print("Définition de la classe {0}".format(classe))
    return classe

@decorateur
class Test:
    pass

>>> Définition de la classe <class '__main__.Test'>
```

Voilà. Vous verrez dans la section suivante quel peut être l'intérêt de manipuler nos définitions de classes à travers des décorateurs. Il existe d'autres exemples que celui que je vais vous montrer, bien entendu.

## Chaîner nos décorateurs

Vous pouvez modifier une fonction ou une définition de classe par le biais de plusieurs décorateurs, sous la forme :

```
@decorateur1
@decorateur2
def fonction():
```

Ce n'est pas plus compliqué que ce que vous venez de faire. Je vous le montre pour qu'il ne subsiste aucun doute dans votre esprit, vous pouvez tester à loisir cette possibilité, par vous-mêmes.

Je vais à présent vous présenter quelques applications possibles des décorateurs, inspirées en grande partie de la PEP 318.

# Decorating functions with parameters

```
import functools

user = {"username": "jose", "access_level": "guest"}

def make_secure(func):
    @functools.wraps(func) # it keeps the name and the documentation of get_admin_password() if there are any
    def secure_function(*args, **kwargs):
        if user["access_level"] == "admin":
            return func(*args, **kwargs)
        else:
            return f"No admin permissions for {user['username']}."

    return secure_function

@make_secure
def get_password(panel):
    if panel == "admin":
        return "1234"
    elif panel == "billing":
        return "super_secure_password"

print(get_password("billing"))
```

# Decorators with parameters

[https://www.youtube.com/watch?v=iF\\_vhsrGbbM](https://www.youtube.com/watch?v=iF_vhsrGbbM)

```
import functools

user = {"username": "jose", "access_level": "guest"}

def make_secure(access_level):
    def decorator(func):
        @functools.wraps(func)
        def secure_function(*args, **kwargs):
            if user["access_level"] == access_level:
                return func(*args, **kwargs)
            else:
                return f"No {access_level} permissions for {user['username']}."

        return secure_function
    return decorator

@make_secure("admin")
def get_admin_password():
    return "admin: 1234"

@make_secure("guest")
def get_dashboard_password():
    return "user: user_password"

print(get_admin_password())
print(get_dashboard_password())

user = {"username": "anna", "access_level": "admin"}

print(get_admin_password())
print(get_dashboard_password())

>>> No admin permissions for jose.
>>> user: user_password
>>> admin: 1234
>>> No guest permissions for anna.
```

# Exemples d'applications

Nous allons voir deux exemples d'applications des décorateurs dans cette section. Vous en avez également vu quelques-uns dans la section précédente mais, maintenant que vous maîtrisez la syntaxe, nous allons nous pencher sur des exemples plus parlants !

## Les classes singleton

Certains reconnaîtront sûrement cette appellation. Pour les autres, sachez qu'une classe dite singleton est une classe qui ne peut être instanciée qu'une fois.

Autrement dit, on ne peut créer qu'un seul objet de cette classe.

Cela peut-être utile quand vous voulez être absolument certains qu'une classe ne produira qu'un seul objet, qu'il est inutile (voire dangereux) d'avoir plusieurs objets de cette classe. La première fois que vousappelez le constructeur de ce type de classe, vous obtenez le premier et l'unique objet nouvellement instancié. Tout appel ultérieur à ce constructeur renvoie le même objet (le premier créé).

Ceci est très facile à modéliser grâce à des décorateurs.

## Code de l'exemple

```
def singleton(classe_definie):
    instances = {} # Dictionnaire de nos instances singletons
    def get_instance():
        if classe_definie not in instances:
            # On crée notre premier objet de classe_definie
            instances[classe_definie] = classe_definie()
        return instances[classe_definie]
    return get_instance
```

## Explications

D'abord, pour utiliser notre décorateur, c'est très simple : il suffit de mettre l'appel à notre décorateur avant la définition des classes que nous souhaitons utiliser en tant que singleton:

# Exemples d'applications

```
@singleton  
class Test:  
    pass
```

```
a = Test()  
b = Test()  
a is b  
True
```

Quand on crée notre premier objet celui se trouvant dans, notre constructeur est bien appelé. Quand on souhaite créer un second objet, c'est celui contenu dans a qui est renvoyé. Ainsi, a et b pointent vers le même objet.

Intéressons-nous maintenant à notre décorateur. Il définit dans son corps un dictionnaire. Ce dictionnaire contient en guise de clé la classe singleton et en tant que valeur l'objet créé correspondant. Il renvoie notre fonction interne get\_instance qui va remplacer notre classe. Ainsi, quand on voudra créer un nouvel objet, ce sera get\_instance qui sera appelée. Cette fonction vérifie si notre classe se trouve dans le dictionnaire. Si ce n'est pas le cas, on crée notre premier objet correspondant et on l'insère dans le dictionnaire. Dans tous les cas, on renvoie l'objet correspondant dans le dictionnaire (soit il vient d'être créé, soit c'est notre objet créé au premier appel du constructeur).

Grâce à ce système, on peut avoir plusieurs classes déclarées comme des singleton et on est sûr que, pour chacune de ces classes, un seul objet sera créé.

## Contrôler les types passés à notre fonction

Vous l'avez déjà observé dans Python : aucun contrôle n'est fait sur le type des données passées en paramètres de nos fonctions. Certaines, comme print, acceptent n'importe quel type. D'autres lèvent des exceptions quand un paramètre d'un type incorrect leur est fourni.

Il pourrait être utile de coder un décorateur qui vérifie les types passés en paramètres à notre fonction et qui lève une exception si les types attendus ne correspondent pas à ceux reçus lors de l'appel à la fonction.

# Exemples d'applications

Notre décorateur `controler_types` doit s'assurer qu'à chaque fois qu'on appelle la fonction `intervalle`, ce sont des entiers qui sont passés en paramètres en tant que `base_infetbase_sup`.

Ce décorateur est plus complexe, bien que j'aie simplifié au maximum l'exemple de la PEP 318.

Encore une fois, s'il est un peu long à écrire, il est d'une simplicité enfantine à utiliser.

Code de l'exemple:

```
def controler_types(*a_args, **a_kwargs):
    """On attend en paramètres du décorateur les types souhaités. On accepte
    une liste de paramètres indéterminés, étant donné que notre fonction
    définie pourra être appelée avec un nombre variable de paramètres et que
    chacun doit être contrôlé"""
    def decorateur(fonction_a_executer):
        """Notre décorateur. Il doit renvoyer fonction_modifiee"""
        def fonction_modifiee(*args, **kwargs):
            """Notre fonction modifiée. Elle se charge de contrôler
            les types qu'on lui passe en paramètres"""

            # La liste des paramètres attendus (a_args) doit être de même
            # Longueur que celle reçue (args)
            if len(a_args) != len(args):
                raise TypeError("le nombre d'arguments attendu n'est pas égal " \
                               "au nombre reçu")
            # On parcourt la liste des arguments reçus et non nommés
            for i, arg in enumerate(args):
                if a_args[i] is not type(arg):
                    raise TypeError("l'argument {0} n'est pas du type " \
                                   "{1}".format(i, a_args[i]))
            # On parcourt à présent la liste des paramètres reçus et nommés
            for cle in kwargs:
                if cle not in a_kwargs:
                    raise TypeError("l'argument {0} n'a aucun type " \
                                   "précisé".format(repr(cle)))
                if a_kwargs[cle] is not type(kwargs[cle]):
                    raise TypeError("l'argument {0} n'est pas de type" \
                                   "{1}".format(repr(cle), a_kwargs[cle]))
            return fonction_a_executer(*args, **kwargs)

            return fonction_modifiee

        return decorateur
    return fonction_modifiee
```

# Exemples d'applications

## Explications

C'est un décorateur assez complexe (et pourtant, croyez-moi, je l'ai simplifié autant que possible). Nous allons d'abord voir comment l'utiliser :

```
@controler_types(int, int)
def intervalle(base_inf, base_sup):
    print("Intervalle de {0} à {1}".format(base_inf, base_sup))

intervalle(1, 8)
# Intervalle de 1 à 8
intervalle(5, "oups!")
>>> Intervalle de 1 à 8
Traceback (most recent call last):
File "C:\Users\MOTTIER LUCIE\Documents\GitHub\TheCompletePythonCourse\test.py", line 90, in <module>
    intervalle(5, "oups!")
File "C:\Users\MOTTIER LUCIE\Documents\GitHub\TheCompletePythonCourse\test.py", line 67, in fonction_modifiee
    raise TypeError("l'argument {0} n'est pas du type "
TypeError: l'argument 1 n'est pas du type <class 'int'>
```

Là encore, l'utilisation est des plus simples. Intéressons-nous au décorateur proprement dit, c'est déjà un peu plus complexe.

Notre décorateur doit prendre des paramètres (une liste de paramètres indéterminés d'ailleurs, car notre fonction doit elle aussi prendre une liste de paramètres indéterminés et l'on doit contrôler chacun d'eux). On définit donc un paramètre `a_args` qui contient la liste des types des paramètres non nommés attendus, et un second paramètre `a_kwargs` qui contient le dictionnaire des types des paramètres nommés attendus.

Vous suivez toujours ?

Vous devriez comprendre la construction d'ensemble, nous l'avons vue un peu plus haut. Elle comprend trois niveaux, puisque nous devons influer sur le comportement de la fonction et que notre décorateur prend des paramètres. Notre code de contrôle se trouve, comme il se doit, dans notre fonction `fonction_modifiee`(qui va prendre la place de notre fonction `a_executer`).

On commence par vérifier que la liste des paramètres non nommés attendus est bien égale en taille à la liste des paramètres non nommés reçus. On vérifie ensuite individuellement chaque paramètre reçu, en contrôlant son type. Si le type reçu est égal au type attendu, tout va bien. Sinon, on lève une exception. On répète l'opération sur les paramètres nommés (avec une petite différence, puisqu'il s'agit de paramètres nommés : ils sont contenus dans un dictionnaire, pas une liste).

Si tout va bien (aucune exception n'a été levée), on exécute notre fonction en renvoyant son résultat.

Voilà nos exemples d'applications. Il y en a bien d'autres, vous pouvez en retrouver plusieurs sur la PEP 318 consacrée aux décorateurs, ainsi que des informations supplémentaires : n'hésitez pas à y faire un petit tour.

# Résumé

## En résumé

- Les décorateurs permettent de modifier le comportement d'une fonction.
- Ce sont eux-mêmes des fonctions, prenant en paramètre une fonction et renvoyant une fonction (qui peut être la même).
- On peut déclarer une fonction comme décorée en plaçant, au-dessus de la ligne de sa définition, la ligne @nom\_decorateur.
- Au moment de la définition de la fonction, le décorateur est appelé et la fonction qu'il renvoie sera celle utilisée.
- Les décorateurs peuvent également prendre des paramètres pour influer sur le comportement de la fonction décorée.

# Découvrez les metaclasses

# Découvrez les métaclasses

## Découvrez les métaclasses

Toujours plus loin vers la métaprogrammation ! Nous allons ici nous intéresser au concept des métaclasses, ou comment générer des classes à partir... d'autres classes ! Je ne vous cache pas qu'il s'agit d'un concept assez avancé de la programmation Python, prenez donc le temps nécessaire pour comprendre ce nouveau concept.

# Retour sur le processus d'instanciation

Depuis la troisième partie de ce cours, nous avons créé bon nombre d'objets. Nous avons découvert au début de cette partie le constructeur, cette méthode appelée quand on souhaite créer un objet.

Je vous ai dit alors que les choses étaient un peu plus complexes que ce qu'il semblait. Nous allons maintenant voir en quoi !

Admettons que vous ayez défini une classe :

```
class Personne:  
    """Classe définissant une personne.  
    """
```

*Elle possède comme attributs :*

*nom -- le nom de la personne*

*prenom -- son prénom*

*age -- son âge*

*lieu\_residence -- son lieu de résidence*

*Le nom et le prénom doivent être passés au constructeur.*""""

```
def __init__(self, nom, prenom):  
    """Constructeur de notre personne.  
    """  
    self.nom = nom  
    self.prenom = prenom  
    self.age = 23  
    self.lieu_residence = "Lyon"
```

Cette syntaxe n'a rien de nouveau pour nous.

Maintenant, que se passe-t-il quand on souhaite créer une personne ? Facile, on rédige le code suivant :

```
personne = Personne("Doe", "John")
```

# Retour sur le processus d'instanciation

Lorsque l'on exécute cela, Python appelle notre constructeur `__init__` en lui transmettant les arguments fournis à la construction de l'objet. Il y a cependant une étape intermédiaire.

Si vous examinez la définition de notre constructeur :

```
def __init__(self, nom, prenom):
```

Vous ne remarquez rien d'étrange ? Peut-être pas, car vous avez été habitués à cette syntaxe depuis le début de cette partie : la méthode prend en premier paramètre `self`. Or, `self`, vous vous en souvenez, c'est l'objet que nous manipulons. Sauf que, quand on crée un objet... on souhaite récupérer un nouvel objet mais on n'en passe aucun à la classe. D'une façon ou d'une autre, notre classe crée un nouvel objet et le passe à notre constructeur. La méthode `__init__` se charge d'écrire dans notre objet ses attributs, mais elle n'est pas responsable de la création de notre objet. Nous allons à présent voir qui s'en charge.

La méthode `__new__`

La méthode `__init__`, comme nous l'avons vu, est là pour initialiser notre objet (en écrivant des attributs dedans, par exemple) mais elle n'est pas là pour le créer. La méthode qui s'en charge, c'est `__new__`.

C'est aussi une méthode spéciale, vous en reconnaîtrez la particularité. C'est également une méthode définie par object, que l'on peut redéfinir en cas de besoin.

Avant de voir ce qu'elle prend en paramètres, voyons plus précisément ce qui se passe quand on tente de construire un objet :

- On demande à créer un objet, en écrivant par exemple `Personne("Doe", "John")`.
- La méthode `__new__` de notre classe (ici `Personne`) est appelée et se charge de construire un nouvel objet.
- Si `__new__` renvoie une instance de la classe, on appelle le constructeur `__init__` en lui passant en paramètres cette nouvelle instance ainsi que les arguments passés lors de la création de l'objet.

Maintenant, intéressons-nous à la structure de notre méthode `__new__`.

C'est une méthode statique, ce qui signifie qu'elle ne prend pas `self` en paramètre. C'est logique, d'ailleurs : son but est de créer une nouvelle instance de classe, l'instance n'existe pas encore. Elle ne prend donc pas `self` en premier paramètre (l'instance d'objet). Cependant, elle prend la classe manipulée `cls`.

Autrement dit, quand on souhaite créer un objet de la classe `Personne`, la méthode `__new__` de la classe `Personne` est appelée et prend comme premier paramètre la classe `Personne` elle-même.

Les autres paramètres passés à la méthode `__new__` seront transmis au constructeur.

# Retour sur le processus d'instanciation

Voyons un peu cela, exprimé sous forme de code :

```
class Personne:  
    """Classe définissant une personne.  
  
    Elle possède comme attributs :  
    nom -- le nom de la personne  
    prenom -- son prénom  
    age -- son âge  
    lieu_residence -- son lieu de résidence  
  
    Le nom et le prénom doivent être passés au constructeur.""""  
  
    def __new__(cls, nom, prenom):  
        print("Appel de la méthode __new__ de la classe {}".format(cls))  
        # On laisse le travail à object  
        return object.__new__(cls, nom, prenom)  
  
    def __init__(self, nom, prenom):  
        """Constructeur de notre personne.""""  
        print("Appel de la méthode __init__")  
        self.nom = nom  
        self.prenom = prenom  
        self.age = 23  
        self.lieu_residence = "Lyon"
```

Essayons de créer une personne :

```
personne = Personne("Doe", "John")  
>>> Appel de la méthode __new__ de la classe <class '__main__.Personne'>  
>>> Appel de la méthode __init__
```

# Retour sur le processus d'instanciation

Redéfinir `__new__` peut permettre, par exemple, de créer une instance d'une autre classe. Elle est principalement utilisée par Python pour produire des types immuables (en anglais, immutable), que l'on ne peut modifier, comme le sont les chaînes de caractères, les tuples, les entiers, les flottants...

La méthode `__new__` est parfois redéfinie dans le corps d'une métaclasse. Nous allons à présent voir ce dont il s'agit.

# Créer une classe dynamique

Je le répète une nouvelle fois, en Python, tout est objet. Cela veut dire que les entiers, les flottants, les listes sont des objets, que les modules sont des objets, que les packages sont des objets... mais cela veut aussi dire que les classes sont des objets !

## La méthode que nous connaissons

Pour créer une classe, nous n'avons vu qu'une méthode, la plus utilisée, faisant appel au mot-clé `class`.

```
class MaClasse:
```

Vous pouvez ensuite créer des instances sur le modèle de cette classe, je ne vous apprends rien.

Mais là où cela se complique, c'est que les classes sont également des objets.

Si les classes sont des objets... cela veut dire que les classes sont elles-mêmes modelées sur des classes ?

Eh oui. Les classes, comme tout objet, sont modelées sur une classe. Cela paraît assez difficile à comprendre au début. Peut-être cet extrait de code vous aidera-t-il à comprendre l'idée.

```
print(type(5))
print(type("une chaîne"))
print(type([1, 2, 3]))
print(type(int))
print(type(str))
print(type(list))
```

```
<class 'int'>
<class 'str'>
<class 'list'>
<class 'type'>
<class 'type'>
<class 'type'>
```

# Créer une classe dynamique

On demande le type d'un entier et Python nous répond `class int`. Sans surprise. Mais si on lui demande la classe de `int`, Python nous répond `class type`.

En fait, par défaut, toutes nos classes sont modelées sur la classe `type`. Cela signifie que :

quand on crée une nouvelle classe (class `Personne`: par exemple), Python appelle la méthode `__new__` de la classe `type`;

une fois la classe créée, on appelle le constructeur `__init__` de la classe `type`.

Cela semble sans doute encore obscur. Ne désespérez pas, vous comprendrez peut-être un peu mieux ce dont je parle en lisant la suite. Sinon, n'hésitez pas à relire ce passage et à faire des tests par vous-mêmes.

Créer une classe dynamiquement

Résumons :

nous savons que les objets sont modelés sur des classes ;

nous savons que nos classes, étant elles-mêmes des objets, sont modelées sur une classe ;

la classe sur laquelle toutes les autres sont modelées par défaut s'appelle `type`.

Je vous propose d'essayer de créer une classe dynamiquement, sans passer par le mot-clé `class` mais par la classe `type` directement.

La classe `type` prend trois arguments pour se construire :

le nom de la classe à créer ;

un tuple contenant les classes dont notre nouvelle classe va hériter ;

un dictionnaire contenant les attributs et méthodes de notre classe.

# Créer une classe dynamique

```
Personne = type("Personne", (), {})
print(Personne)

john = Personne()
print(dir(john))

>>> <class '__main__.Personne'>
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

J'ai simplifié le code au maximum. Nous créons bel et bien une nouvelle classe que nous stockons dans notre variable Personne, mais elle est vide. Elle n'hérite d'aucune classe et elle ne définit aucun attribut ni méthode de classe.

Nous allons essayer de créer deux méthodes pour notre classe :

un constructeur `__init__`;

une méthode présenter affichant le prénom et le nom de la personne.

Je vous donne ici le code auquel on peut arriver :

# Créer une classe dynamique

```
def creer_personne(personne, nom, prenom):
    """La fonction qui jouera le rôle de constructeur pour notre classe Personne.

    Elle prend en paramètre, outre la personne :
    nom -- son nom
    prenom -- son prenom"""

    personne.nom = nom
    personne.prenom = prenom
    personne.age = 21
    personne.lieu_residence = "Lyon"

def presenter_personne(personne):
    """Fonction présentant la personne.

    Elle affiche son prénom et son nom"""

    print("{} {}".format(personne.prenom, personne.nom))

# Dictionnaire des méthodes
methodes = {
    "__init__": creer_personne,
    "presenter": presenter_personne,
}

# Création dynamique de la classe
Personne = type("Personne", (), methodes)

# Création dynamique de la classe
Personne = type("Personne", (), methodes)
john = Personne("Doe", "John")
john.nom
john.prenom
john.age
john.presenter()
```

```
>>> Doe
>>> John
>>> 21
>>> John Doe
```

# Créer une classe dynamique

Je ne vous le cache pas, c'est une fonctionnalité que vous utiliserez sans doute assez rarement. Mais cette explication était à propos quand on s'intéresse aux métaclasses.

Pour l'heure, décomposons notre code :

1. On commence par créer deux fonctions, `creer_personne` et `presente_personne`. Elles sont amenées à devenir les méthodes `__init__` et présenter de notre future classe. Étant de futures méthodes d'instance, elles doivent prendre en premier paramètre l'objet manipulé.
2. On place ces deux fonctions dans un dictionnaire. En clé se trouve le nom de la future méthode et en valeur, la fonction correspondante.
3. Enfin, on fait appel à `type` en lui passant, en troisième paramètre, le dictionnaire que l'on vient de constituer.

Si vous essayez de mettre des attributs dans ce dictionnaire passé à `type`, vous devez être conscients du fait qu'il s'agira d'attributs de classe, pas d'attributs d'instance.

## Définition d'une métaclassse

Nous avons vu que `type` est la métaclassse de toutes les classes par défaut. Cependant, une classe peut posséder une autre métaclassse que `type`.

Construire une métaclassse se fait de la même façon que construire une classe. Les métaclasses héritent de `type`. Nous allons retrouver la structure de base des classes que nous avons vues auparavant.

Nous allons notamment nous intéresser à deux méthodes que nous avons utilisées dans nos définitions de classes :

- la méthode `__new__`, appelée pour créer une classe ;
- la méthode `__init__`, appelée pour construire la classe.

# Créer une classe dynamique

## La méthode `__new__`

Elle prend quatre paramètres :

- la métaclassé servant de base à la création de notre nouvelle classe ;
- le nom de notre nouvelle classe ;
- un **tuple** contenant les classes dont héritent notre classe à créer ;
- le dictionnaire des attributs et méthodes de la classe à créer.

Les trois derniers paramètres, vous devriez les reconnaître : ce sont les mêmes que ceux passés à `type`.

Voici une méthode `__new__` minimalisté.

```
class MaMetaClasse(type):
    """Exemple d'une métaclassé."""

    def __new__(metacls, nom, bases, dict):
        """Création de notre classe."""
        print("On crée la classe {}".format(nom))
        return type.__new__(metacls, nom, bases, dict)
```

# Créer une classe dynamique

Pour dire qu'une classe prend comme métaclass autre chose que type, c'est dans la ligne de la définition de la classe que cela se passe :

```
class MaClasse(metaclass=MaMetaClasse):  
    pass
```

En exécutant ce code, vous pouvez voir :

```
>>> On crée la classe MaClasse
```

La méthode `__init__`

Le constructeur d'une métaclass prend les mêmes paramètres que `__new__`, sauf le premier, qui n'est plus la métaclass servant de modèle mais la classe que l'on vient de créer.

Les trois paramètres suivants restent les mêmes : le nom, le tuple des classes-mères et le dictionnaire des attributs et méthodes de classe.

Il n'y a rien de très compliqué dans le procédé, l'exemple ci-dessus peut être repris en le modifiant quelque peu pour qu'il s'adapte à la méthode `__init__`.

Maintenant, voyons concrètement à quoi cela peut servir.

Les métaclasses en action

Comme vous pouvez vous en douter, les métaclasses sont généralement utilisées pour des besoins assez complexes. L'exemple le plus répandu est une métaclass chargée de tracer l'appel de ses méthodes. Autrement dit, dès qu'on appelle une méthode d'un objet, une ligne s'affiche pour le signaler. Mais cet exemple est assez difficile à comprendre car il fait appel à la fois au concept des métaclasses et à celui des décorateurs, pour décorer les méthodes tracées.

Je vous propose quelque chose de plus simple. Il va de soi qu'il existe bien d'autres usages, dont certains complexes, des métaclasses.

Nous allons essayer de garder nos classes créées dans un dictionnaire prenant comme clé le nom de la classe et comme valeur la classe elle-même.

# Créer une classe dynamique

Par exemple, dans une bibliothèque destinée à construire des interfaces graphiques, on trouve plusieurs widgets (ce sont des objets graphiques) comme des boutons, des cases à cocher, des menus, des cadres... Généralement, ces objets sont des classes héritant d'une classe mère commune. En outre, l'utilisateur peut, en cas de besoin, créer ses propres classes héritant des classes de la bibliothèque.

Par exemple, la classe mère de tous nos widgets s'appellera Widget. De cette classe hériteront les classes Bouton, CaseACocher, Menu, Cadre, etc. L'utilisateur de la bibliothèque pourra par ailleurs en dériver ses propres classes.

Le dictionnaire que l'on aimerait créer se présente comme suit :

```
{  
    "Widget": Widget,  
    "Bouton": Bouton,  
    "CaseACocher": CaseACocher,  
    "Menu": Menu,  
    "Cadre": Cadre,  
}
```

Ce dictionnaire pourrait être rempli manuellement à chaque fois qu'on crée une classe héritant de Widget mais avouez que ce ne serait pas très pratique.

Dans ce contexte, les métaclasses peuvent nous faciliter la vie. Vous pouvez essayer de faire l'exercice, le code n'est pas trop complexe. Cela dit, étant donné qu'on a vu beaucoup de choses dans ce chapitre et que les métaclasses sont un concept plutôt avancé, je vous donne directement le code qui vous aidera peut-être à comprendre le mécanisme :

```
trace_classes = {} # Notre dictionnaire vide  
  
class MetaWidget(type):  
    """Notre métaclass pour nos Widgets.  
  
    Elle hérite de type, puisque c'est une métaclass.  
    Elle va écrire dans le dictionnaire trace_classes à chaque fois  
    qu'une classe sera créée, utilisant cette métaclass naturellement.""""  
  
    def __init__(cls, nom, bases, dict):  
        """Constructeur de notre métaclass, appelé quand on crée une classe.""""  
        type.__init__(cls, nom, bases, dict)  
        trace_classes[nom] = cls
```

# Créer une classe dynamique

Pas trop compliqué pour l'heure. Créons notre classe Widget:

```
class Widget(metaclass=MetaWidget):
    """Classe mère de tous nos widgets."""
    pass
```

Après avoir exécuté ce code, vous pouvez voir que notre classe Widget a bien été ajoutée dans notre dictionnaire :

```
trace_classes
>>> {'Widget': <class '__main__.Widget'>}
```

Maintenant, construisons une nouvelle classe héritant de Widget.

```
class bouton(Widget):
    """Une classe définissant le widget bouton."""
    pass
```

Si vous affichez de nouveau le contenu du dictionnaire, vous vous rendrez compte que la classe Bouton a bien été ajoutée. Héritant de Widget, elle reprend la même métaclasses (sauf mention contraire explicite) et elle est donc ajoutée au dictionnaire.

Vous pouvez étoffer cet exemple, faire en sorte que l'aide de la classe soit également conservée, ou qu'une exception soit levée si une classe du même nom existe déjà dans le dictionnaire.

## Pour conclure

Les métaclasses sont un concept de programmation assez avancé, puissant mais délicat à comprendre de prime abord. Je vous invite, en cas de doute, à tester par vous-mêmes ou à rechercher d'autres exemples, ils sont nombreux.

# Résumé

## En résumé

Le processus d'instanciation d'un objet est assuré par deux méthodes, `__new__` et `__init__`.

`__new__` est chargée de la création de l'objet et prend en premier paramètre sa classe.

`__init__` est chargée de l'initialisation des attributs de l'objet et prend en premier paramètre l'objet précédemment créé par `__new__`.

Les classes étant des objets, elles sont toutes modelées sur une classe appelée **métaclasse**.

À moins d'être explicitement modifiée, la métaclass de toutes les classes est *type*.

On peut utiliser *type* pour créer des classes dynamiquement.

On peut faire hériter une classe de *type* pour créer une nouvelle métaclass.

Dans le corps d'une classe, pour spécifier sa métaclass, on exploite la syntaxe suivante :`class MaClasse(metaclass=NomDeLaMetaClasse):`

Manipulez les  
expressions régulières

# Que sont les expressions régulières ?

Pour s'entraîner aux expressions régulières: <https://regexr.com/>

Dans ce chapitre, je vais m'attarder sur les expressions régulières et sur le module `re` qui permet de les manipuler. En quelques mots, sachez que les expressions régulières permettent de réaliser très rapidement et facilement des recherches sur des chaînes de caractères.

Il existe, naturellement, bien d'autres modules permettant de manipuler du texte. C'est toutefois sur celui-ci que je vais m'attarder aujourd'hui, tout en vous donnant les moyens d'aller plus loin si vous le désirez.

## Que sont les expressions régulières ?

Les **expressions régulières** sont un puissant moyen de rechercher et d'isoler des expressions d'une chaîne de caractères.

Pour simplifier, imaginez que vous faites un programme qui demande un certain nombre d'informations à l'utilisateur afin de les stocker dans un fichier. Lui demander son nom, son prénom et quelques autres informations, ce n'est pas bien difficile : on va utiliser la fonction `input` et récupérer le résultat. Jusqu'ici, rien de nouveau.

Mais si on demande à l'utilisateur de fournir un numéro de téléphone ? Qu'est-ce qui l'empêche de taper n'importe quoi ? Si on lui demande de fournir une adresse e-mail et qu'il tape quelque chose d'invalidé, par exemple « `je_te_donnerai_pas_mon_email` », que va-t-il se passer si l'on souhaite envoyer automatiquement un email à cette personne ?

Si ce cas n'est pas géré, vous risquez d'avoir un problème. Les expressions régulières sont un moyen de rechercher, d'isoler ou de remplacer des expressions dans une chaîne. Ici, elles nous permettraient de vérifier que le numéro de téléphone saisi compte bien dix chiffres, qu'il commence par un 0 et qu'il compte éventuellement des séparateurs tous les deux chiffres. Si ce n'est pas le cas, on demande à l'utilisateur de le saisir à nouveau.

## Quelques éléments de syntaxe pour les expressions régulières

Si vous connaissez déjà les expressions régulières et leur syntaxe, vous pouvez passer directement à la section consacrée au module `re`. Sinon, sachez que je ne pourrai vous présenter que brièvement les expressions régulières. C'est un sujet très vaste, qui mérite un livre à lui tout seul. Ne paniquez pas, toutefois, je vais vous donner quelques exemples concrets et vous pourrez toujours trouver des explications plus approfondies sur le Web.

## Concrètement, comment cela se présente-t-il ?

Le module `re`, que nous allons découvrir un peu plus loin, nous permet de faire des recherches très précises dans des chaînes de caractères et de remplacer des éléments de nos chaînes, le tout en fonction de critères particuliers. Ces critères, ce sont nos expressions régulières. Pour nous, elles se présentent sous la forme de chaînes de caractères. Les expressions régulières deviennent assez rapidement difficiles à lire mais ne vous en faites pas : nous allons y aller petit à petit.

# Que sont les expressions régulières ?

## Des caractères ordinaires

Quand on forme une expression régulière, on peut utiliser des caractères spéciaux et d'autres qui ne le sont pas. Par exemple, si nous recherchons le mot chat dans notre chaîne, nous pouvons écrire comme expression régulière la chaîne « chat ». Jusque là, rien de très compliqué.

Mais vous vous doutez bien que les expressions régulières ne se limitent pas à ce type de recherche extrêmement simple, sans quoi les méthodes `find` et `replace` de la classe `str` auraient suffi.

## Rechercher au début ou à la fin de la chaîne

Vous pouvez rechercher au début de la chaîne en plaçant en tête de votre regex (abréviation de Regular Expression) le signe d'accent circonflexe `^`. Si, par exemple, vous voulez rechercher la syllabe `cha` en début de votre chaîne, vous écrirez donc l'expression `^cha`. Cette expression sera trouvée dans la chaîne '`chaton`' mais pas dans la chaîne '`achat`'.

Pour matérialiser la fin de la chaîne, vous utiliserez le signe `$`. Ainsi, l'expression `q$` sera trouvée uniquement si votre chaîne se termine par la lettre `q` minuscule.

## Contrôler le nombre d'occurrences

Les caractères spéciaux que nous allons découvrir permettent de contrôler le nombre de fois où notre expression apparaît dans notre chaîne.

Regardez l'exemple ci-dessous :

`chat*`

Nous avons rajouté un astérisque (\*) après le caractère `t` de `chat`. Cela signifie que notre lettres pourra se retrouver 0, 1, 2, ... fois dans notre chaîne. Autrement dit, notre expression `chat*` sera trouvée dans les chaînes suivantes :`'chat'`,`'chaton'`,`'chateau'`,`'herbe à chat'`,`'chapeau'`,`'chatterton'`,`'chattttttt'`...

Regardez un à un les exemples ci-dessus pour vérifier que vous les comprenez bien. On trouvera dans chacune de ces chaînes l'expression régulière `chat*`. Traduite en français, cette expression signifie : « on recherche une lettre `c` suivie d'une lettre `h` suivie d'une lettre `a` suivie, éventuellement, d'une lettre `t` qu'on peut trouver zéro, une ou plusieurs fois ». Peu importe que ces lettres soient trouvées au début, à la fin ou au milieu de la chaîne.

Un autre exemple ? Considérez l'expression régulière ci-dessous et essayez de la comprendre :

`bat*e`

# Que sont les expressions régulières ?

Cette expression est trouvée dans les chaînes suivantes : 'bateau', 'batteur' et 'joan baez'.

Dans nos exemples, le signe \* n'agit que sur la lettre qui le précède directement, pas sur les autres lettres qui figurent avant ou après.

Il existe d'autres signes permettant de contrôler le nombre d'occurrences d'une lettre. Je vous ai fait un petit récapitulatif dans le tableau suivant, en prenant des exemples d'expressions avec les lettres a, b et c :

Signe	Explication	Expression	Chaînes contenant l'expression
*	0, 1 ou plus	abc*	'ab', 'abc', 'abcc', 'abcccccc'
+	1 ou plus	abc+	'abc', 'abcc', 'abccc'
?	0 ou 1	abc?	'ab', 'abc'

Vous pouvez également contrôler précisément le nombre d'occurrences grâce aux accolades :

E{4}: signifie 4 fois la lettre E majuscule ;

E{2,4}: signifie de 2 à 4 fois la lettre E majuscule ;

E{,5}: signifie de 0 à 5 fois la lettre E majuscule ;

E{8,}: signifie 8 fois minimum la lettre E majuscule.

Character classes	
.	any character except newline
\w \d \s	word, digit, whitespace
\W \D \S	not word, digit, whitespace
[abc]	any of a, b, or c
[^abc]	not a, b, or c
[a-g]	character between a & g
Anchors	
^abc\$	start / end of the string
\b \B	word, not-word boundary
Escaped characters	
\. \* \\	escaped special characters
\t \n \r	tab, linefeed, carriage return
Groups & Lookaround	
(abc)	capture group
\1	backreference to group #1
(?:abc)	non-capturing group
(?=abc)	positive lookahead
(?!abc)	negative lookahead
Quantifiers & Alternation	
a* a+ a?	0 or more, 1 or more, 0 or 1
a{5} a{2,}	exactly five, two or more
a{1,3}	between one & three
a+? a{2,}?	match as few as possible
ab   cd	match ab or cd

# Le module re

Le module **re** a été spécialement conçu pour travailler avec les expressions régulières (Regular Expressions). Il définit plusieurs fonctions utiles, que nous allons découvrir, ainsi que des objets propres pour modéliser des expressions.

```
import re
email = 'jpalleau@gmail.com'
expression = '[a-z]+'

matches = re.findall(expression, email)
print(matches)

name = matches[0]
domain = f'{matches[1]}.{matches[2]}'
print(name)
print(domain)

>>> ['jpalleau', 'gmail', 'com']
>>> jpalleau
>>> gmail.com
```

Note that in this case we don't use any regular expression at all, we can do the same with the `split()` method

```
email = 'jpalleau@gmail.com'
parts = email.split('@')

name = parts[0]
domain = parts[1]

print(name)
print(domain)

>>> jpalleau
>>> gmail.com
```

A better way to do the same as above will be:

```
import re
email = 'jpalleau@gmail.com'
expression = '[a-z\.]+' # compare to above here we add \.

matches = re.findall(expression, email)
print(matches)

name = matches[0]
domain = matches[1]
print(name)
print(domain)

>>> ['jpalleau', 'gmail', 'com']
>>> jpalleau
>>> gmail.com
```

```
import re

price = "Price: $189.50"
expression = "Price: \$([0-9]*\.[0-9]*)"

matches = re.search(expression, price)
print(matches.group(0)) # entire match
print(matches.group(1)) # First thing in brackets

price_number = float(matches.group(1))
print(price_number)

>>> 189.50
>>> 189.5
```

# Exercice

## Le module re

Build a secure filename checker using regex. It can be useful if you're dealing with user uploads and you want to limit the filename characters as well as extensions.

Our definition of a secure filename is:

- The filename must start with an English letters or a number (a-zA-Z0-9).
- The filename can only contains English letters, numbers and symbols among these four: `-_()`.
- The filename must end with a proper file extension among `.jpg`, `.jpeg`, `.png` and `.gif`

Hint: The filename should not contains any dots `.`. Except for the one in the file extension.

We will build a function `is_filename_safe()` that returns a Boolean value to indicate if the filename is secure according to our definition.

A sample implementation is shown below:

```
import re

"""

Our definition of a secure filename is:
- The filename must start with an English letters or a number (a-zA-Z0-9).
- The filename can **only** contain English letters, numbers and symbols among these four: `-_()``.
- The filename must end with a proper file extension among `jpg`, `jpeg`, `png` and `gif`
"""

def is_filename_safe(filename: str) -> bool:
    # you only need to change the regular expression (regex) below
    regex = '^[a-zA-Z0-9][a-zA-Z0-9-_()]*\.(jpg|jpeg|png|gif)$'
    return re.match(regex, filename) is not None
```

# Résumé

## En résumé

Les expressions régulières permettent de chercher et remplacer certaines expressions dans des chaînes de caractères.

Le module `re` de Python permet de manipuler des expressions régulières en Python.

La fonction `search` du module `re` permet de chercher une expression dans une chaîne.

Pour remplacer une certaine expression dans une chaîne, on utilise la fonction `sub` du module `re`.

On peut également compiler les expressions régulières grâce à la fonction `compile` du module `re`.

# Exprimez le temps

# Exprimez le temps

Exprimer un temps en informatique, cela soulève quelques questions. Disposer d'une mesure du temps dans un programme peut avoir des applications variées : connaître la date et l'heure actuelles et faire remonter une erreur, calculer depuis combien de temps le programme a été lancé, gérer des alarmes programmées, faire des tests de performance... et j'en passe !

Il existe plusieurs façons de représenter des temps, que nous allons découvrir maintenant.

Pour bien suivre ce chapitre, vous aurez besoin de maîtriser l'objet : savoir ce qu'est un objet et comment en créer un.

# Timezones

## Dates and times in Python

A date & time object in Python that does not know about timezones is called “naïve”

One that does is called “aware”

```
from datetime import datetime  
  
print(datetime.now())
```

```
from datetime import datetime, timezone  
  
print(datetime.now(timezone.utc))
```

# Le module time

Le module time est sans doute le premier à être utilisé quand on souhaite manipuler des temps de façon simple.

Notez que, dans la documentation de la bibliothèque standard, ce module est classé dans la rubrique Generic Operating System Services (c'est-à-dire les services communs aux différents systèmes d'exploitation). Ce n'est pas un hasard : time est un module très proche du système. Cela signifie que certaines fonctions de ce module pourront avoir des résultats différents sur des systèmes différents. Pour ma part, je vais surtout m'attarder sur les fonctionnalités les plus génériques possibles afin de ne perdre personne.

Je vous invite à consulter la documentation de Python sur la bibliothèque standard et sur le module time, pour plus d'informations.

Représenter une date et une heure dans un nombre unique

Comment représenter un temps ? Il existe, naturellement, plusieurs réponses à cette question. Celle que nous allons voir ici est sans doute la moins compréhensible pour un humain, mais la plus adaptée à un ordinateur : on stocke la date et l'heure dans un seul entier.

Comment représenter une date et une heure dans un unique entier ?

L'idée retenue a été de représenter une date et une heure en fonction du nombre de secondes écoulées depuis une date précise. La plupart du temps, cette date est l'Epoch Unix, le 1er janvier 1970 à 00:00:00.

Pourquoi cette date plutôt qu'une autre ?

Il fallait bien choisir une date de début. L'année 1970 a été considérée comme un bon départ, compte tenu de l'essor qu'a pris l'informatique à partir de cette époque. D'autre part, un ordinateur est inévitablement limité quand il traite des entiers ; dans les langages de l'époque, il fallait tenir compte de ce fait tout simple : on ne pouvait pas compter un nombre de secondes trop important. La date de l'Epoch ne pouvait donc pas être trop reculée dans le temps.

Nous allons voir dans un premier temps comment afficher ce fameux nombre de secondes écoulées depuis le 1er janvier 1970 à 00:00:00. On utilise la fonction time du module time.

```
import time  
time.time()  
>>> 1297642146.562
```

# Le module time

Cela fait beaucoup ! D'un autre côté, songez quand même que cela représente le nombre de secondes écoulées depuis plus de quarante ans à présent.

Maintenant, je vous l'accorde, ce nombre n'est pas très compréhensible pour un humain. Par contre, pour un ordinateur, c'est l'idéal : les durées calculées en nombre de secondes sont faciles à additionner, soustraire, multiplier... bref, l'ordinateur se débrouille bien mieux avec ce nombre de secondes, ce timestamp comme on l'appelle généralement.

Faites un petit test : stockez la valeur renvoyée `partime.time()` dans une première variable, puis quelques secondes plus tard stockez la nouvelle valeur renvoyée `partime.time()` dans une autre variable. Comparez-les, soustrayez-les, vous verrez que cela se fait tout seul :

```
import time
debut = time.time()
# On attend quelques secondes avant de taper la commande suivante
fin = time.time()
print(debut, fin)
print(debut < fin)
print(fin - debut) # Combien de secondes entre debut et fin ?

>>> 1623762849.1780844 1623762849.1780844
>>> False
>>> 0.0
```

Vous pouvez remarquer que la valeur renvoyée `partime.time()` n'est pas un entier mais bien un flottant. Le temps ainsi donné est plus précis qu'à une seconde près. Pour des calculs de performance, ce n'est en général pas cette fonction que l'on utilise. Mais c'est bien suffisant la plupart du temps.

## La date et l'heure de façon plus présentable

Vous allez me dire que c'est bien joli d'avoir tous nos temps réduits à des nombres mais que ce n'est pas très lisible pour nous. Nous allons découvrir tout au long de ce chapitre des moyens d'afficher nos temps de façon plus élégante et d'obtenir les diverses informations relatives à une date et une heure. Je vous propose ici un premier moyen : une sortie sous la forme d'un objet contenant déjà beaucoup d'informations.

Nous allons utiliser la fonction `localtime` du module `time`.

# Le module time

time.localtime()

Elle renvoie un objet contenant, dans l'ordre :

tm\_year: l'année sous la forme d'un entier ;

tm\_mon: le numéro du mois (entre 1 et 12) ;

tm\_mday: le numéro du jour du mois (entre 1 et 31, variant d'un mois et d'une année à l'autre) ;

tm\_hour: l'heure du jour (entre 0 et 23) ;

tm\_min: le nombre de minutes (entre 0 et 59) ;

tm\_sec: le nombre de secondes (entre 0 et 61, même si on n'utilisera ici que les valeurs de 0 à 59, c'est bien suffisant) ;

tm\_wday: un entier représentant le jour de la semaine (entre 0 et 6, 0 correspond par défaut au lundi) ;

tm\_yday: le jour de l'année, entre 1 et 366 ;

tm\_isdst: un entier représentant le changement d'heure local.

Comme toujours, si vous voulez en apprendre plus, je vous renvoie à la documentation officielle du module time.

Comme je l'ai dit plus haut, nous allons utiliser la fonction localtime. Elle prend un paramètre optionnel : le timestamp tel que nous l'avons découvert plus haut. Si ce paramètre n'est pas précisé, localtime utilisera automatiquement time.time() et renverra donc la date et l'heure actuelles.

# Le module time

```
import time
time.localtime()
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=3, tm_min=22, tm_sec=7,
tm_wday=0, tm_yday=45, tm_isdst=0)
time.localtime(début)
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=1, tm_min=9, tm_sec=55,
tm_wday=0, tm_yday=45, tm_isdst=0)
time.localtime(fin)
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=1, tm_min=10, tm_sec=2,
tm_wday=0, tm_yday=45, tm_isdst=0)
```

Pour savoir à quoi correspond chaque attribut de l'objet, je vous renvoie un peu plus haut. Pour l'essentiel, c'est assez clair je pense. Malgré tout, la date et l'heure renvoyées ne sont pas des plus lisibles. L'avantage de les avoir sous cette forme, c'est qu'on peut facilement extraire une information si on a juste besoin, par exemple, de l'année et du numéro du jour.

## Récupérer un timestamp depuis une date

Je vais passer plus vite sur cette fonction car, selon toute vraisemblance, vous l'utiliserez moins souvent. L'idée est, à partir d'une structure représentant les date et heure telles que renvoyées par *localtime*, de récupérer le timestamp correspondant. On utilise pour ce faire la fonction *mkttime*.

## Mettre en pause l'exécution du programme pendant un temps déterminé

C'est également une fonctionnalité intéressante, même si vous n'en voyez sans doute pas l'utilité de prime abord. La fonction qui nous intéresse est *sleep* et elle prend en paramètre un nombre de secondes qui peut être sous la forme d'un entier ou d'un flottant. Pour vous rendre compte de l'effet, je vous encourage à tester par vous-mêmes :

```
time.sleep(3.5) # Faire une pause pendant 3,5 secondes
```

Comme vous pouvez le voir, Python se met en pause et vous devez attendre 3,5 secondes avant que les trois chevrons s'affichent à nouveau.

# Le module time

## Formater un temps

Intéressons nous maintenant à la fonction *strftime*. Elle permet de formater une date et heure en la représentant dans une chaîne de caractères.

Elle prend deux paramètres :

- La chaîne de formatage (nous verrons plus bas comment la former).
- Un temps optionnel tel que le renvoie *localtime*. Si le temps n'est pas précisé, c'est la date et l'heure courantes qui sont utilisées par défaut.

Pour construire notre chaîne de formatage, nous allons utiliser plusieurs caractères spéciaux. Python va remplacer ces caractères par leur valeur (la valeur du temps passé en second paramètre ou du temps actuel sinon).

Exemple :

```
time.strftime('%Y') # strftime: string format time
```

Voici un tableau récapitulatif des quelques symboles que vous pouvez utiliser dans cette chaîne :

Symbole	Signification
%A	Nom du jour de la semaine
%B	Nom du mois
%d	Jour du mois (de 01 à 31)
%H	Heure (de 00 à 23)
%M	Minute (entre 00 et 59)
%S	Seconde (de 00 à 59)
%Y	Année

# Le module time

Donc pour afficher la date telle qu'on y est habitué en France :

```
time.strftime("%A %d %B %Y %H:%M:%S")
```

Mais... c'est en anglais !

Eh oui. Mais avec ce que vous savez déjà et ce que vous allez voir par la suite, vous n'aurez pas de difficulté à personnaliser tout cela !

Bien d'autres fonctions

Le module time propose bien d'autres fonctions. Je ne vous ai montré que celles que j'utilise le plus souvent tout en vous présentant quelques concepts du temps utilisé en informatique. Si vous voulez aller plus loin, vous savez quoi faire... non ? Allez, je vous y encourage fortement donc je vous remets le lien vers la documentation du module time.

Le module datetime

Le module datetime propose plusieurs classes pour représenter des dates et heures. Vous n'allez rien découvrir d'absolument spectaculaire dans cette section mais nous nous avançons petit à petit vers une façon de gérer les dates et heures qui est davantage orientée objet.

Encore et toujours, je ne prétends pas remplacer la documentation. Je me contente d'extraire de celle-ci les informations qui me semblent les plus importantes. Je vous encourage, là encore, à jeter un coup d'œil du côté de la documentation du module datetime.

Représenter une date

# Le module datetime

Vous le reconnaîtrez probablement avec moi, c'est bien d'avoir accès au temps actuel avec une précision d'une seconde sinon plus... mais parfois, cette précision est inutile. Dans certains cas, on a juste besoin d'une date, c'est-à-dire un jour, un mois et une année.

Il est naturellement possible d'extraire cette information de notre timestamp. Le module `datetime` propose une classe `date`, représentant une date, rien qu'une date.

L'objet possède trois attributs :

`year`: l'année ;

`month`: le mois ;

`day`: le jour du mois.

Comment fait-on pour construire notre objet `date` ?

Il y a plusieurs façons de procéder. Le constructeur de cette classe prend trois arguments qui sont, dans l'ordre, l'année, le mois et le jour du mois.

# Le module `datetime`

Il y a plusieurs façons de procéder. Le constructeur de cette classe prend trois arguments qui sont, dans l'ordre, l'année, le mois et le jour du mois.

```
import datetime
date = datetime.date(2010, 12, 25)
print(date)
>>> 2010-12-25
```

Il existe deux méthodes de classe qui peuvent vous intéresser :

- `date.today()`: renvoie la date d'aujourd'hui ;
- `date.fromtimestamp(timestamp)`: renvoie la date correspondant au timestamp passé en argument.

Voyons en pratique :

```
import time
import datetime
aujourdhui = datetime.date.today()
print(aujourdhui)
>> 2021-06-11
print(datetime.date(2011, 2, 14))
>> 2011-02-14
print(datetime.date.fromtimestamp(time.time())) # Équivalent à date.today
>> 2021-06-11
print(datetime.date(2011, 2, 14))
2011-02-14
```

Et bien entendu, vous pouvez manipuler ces dates simplement et les comparer grâce aux opérateurs usuels, je vous laisse essayer !

Représenter une heure

C'est moins courant mais on peut également être amené à manipuler une heure, indépendamment de toute date. La classe `time` du module `datetime` est là pour cela.

On construit une heure avec non pas trois mais cinq paramètres, tous optionnels :

- Hour (0 par défaut) : les heures, valeur comprise entre 0 et 23 ;
- Minute (0 par défaut) : les minutes, valeur comprise entre 0 et 59 ;
- Second (0 par défaut) : les secondes, valeur comprise entre 0 et 59 ;
- Microsecond (0 par défaut) : la précision de l'heure en micro-secondes, entre 0 et 1.000.000 ;
- `tzinfo` (None par défaut) : l'information de fuseau horaire (je ne détaillerai pas cette information ici).

Cette classe est moins utilisée que `datetime.date` mais elle peut se révéler utile dans certains cas. Je vous laisse faire quelques tests, n'oubliez pas de vous reporter à la documentation du module `datetime` pour plus d'informations.

# Le module datetime

## UTC

```
from datetime import datetime, timezone  
  
print(datetime.now(timezone.utc))  
>> 2021-06-11 11:34:27.710027+00:00
```



The programs, should always work in UTC time, when you ask for a date to a user it is IMPORTANT to convert it in UTC time and work with UTC.

# Le module datetime



Hey there! Today we bring you frequently used datetime syntax, as well as best practices and examples!

## Frequently used syntax

This syntax below can be used in the `strftime` and `strptime` methods, like so:

```
import datetime
now = datetime.datetime.now()
print(now.strftime('%d-%m-%Y')) # 13-03-2018
print(now.strftime('%A, %B %d, %Y')) # Tuesday, March 13, 2018

user_date = input('Enter the current date as %Y-%m-%d: ')
print(datetime.datetime.strptime(user_date, "%Y-%m-%d"))
```

# Le module datetime

Code	Meaning	Example
%A	Day of the week as text	Monday
%d	Day of the month from 01 to 31	17
%B	Month name as text	October
%m	Month of the year from 01 to 12	7
%Y	Year with century	2016
%H	Hour from 00 to 23	15
%I	Hour from 00 to 12	7
%p	Equivalent of AM or PM in the current language	AM
%M	Month from 01 to 12	10
%S	Seconds from 00 to 59	54
%X	Date representation in the current language	06/10/15
%x	Time representation in the current language	19:54:22

## Examples

Here's a few code examples from things I've personally written in the past. Use them as a reference or just to have a read over!

```
import datetime

user_date = input('Enter the current date as %Y-%m-%d: ')
print(datetime.datetime.strptime(user_date, "%Y-%m-%d"))
```

Here's an example of both using the input function in a dictionary, to get the values for each field, and also getting the current time.

There can be issues with debugging if you do shorthands like these, so it can sometimes be easier to just create a variable for each value (as in the example directly below this next one).

# Le module datetime

```
import datetime

users = []

new_user = {
    'name': input('Enter your name: ')
    'location': input('Enter your location: ')
    'registered': datetime.datetime.now(datetime.timezone.utc)
}

users.append(new_user)
```

Here's an example with arguably better (and more readable) code. Creating a variable for each means that debugging can be a bit easier—you can just set a breakpoint on the line that is going wrong to easily identify what's happening.

```
import datetime

users = []

name = input('Enter your name: ')
location = input('Enter your location: ')
registered = datetime.datetime.now(datetime.timezone.utc)

new_user = {
    'name': name
    'location': location
    'registered': registered
}

users.append(new_user)
```

Frequently when creating a new object you may want to store when it was created. This is common with users, for example. Below is a way to do this. Notice that some database engines let you populate a field in the database automatically with the current date when a new row is created, so sometimes you won't need to do this in Python.

```
import datetime

class User:
    def __init__(self, username, password):
        self.username = username
        self.password = password
        self.registered = datetime.datetime.now(datetime.timezone.utc)
```

# Le module datetime

Below we mention a best practice which is using timestamps. Here's how you can turn a datetime object into a timestamp and back again:

```
import datetime

now = datetime.datetime.now(datetime.timezone.utc)
current_timestamp = now.timestamp()

now_from_timestamp = datetime.utcfromtimestamp(current_timestamp)
```

## Best practices

The single most important best practice is, as mentioned in the course, to store all your dates and times in a database as UTC. That means that each datetime will have some timezone information associated with it. That timezone must be UTC (which has an "offset" of 0 hours).

The offset is always relative to UTC, so naturally a UTC datetime has an offset of 0. A CET timezone has a +01:00 offset, which means it is an hour ahead of UTC.

If you store all your timezones as UTC, then you can display them to your user as their local timezone (this is quite simple, just ask them where they live or get that information from their IP address).

Something else we do quite frequently is store dates and times as timestamps. A timestamp is the number of seconds since 1st January 1970 at midnight in UTC. Thus normally you'll calculate a timestamp and store that in your database, then turn it back to a human-readable format using the last example above.

# Le module `datetime`

## Représenter des dates et heures

Et nous y voilà ! Vous n'allez pas être bien surpris par ce que nous allons aborder. Nous avons vu une manière de représenter une date, une manière de représenter une heure, mais on peut naturellement représenter une date et une heure dans le même objet, ce sera probablement la classe que vous utiliserez le plus souvent. Celle qui nous intéresse s'appelle `datetime`, comme son module.

Elle prend d'abord les paramètres de `datetime.date(année, mois, jour)` et ensuite les paramètres `datetime.time(heures, minutes, secondes, micro-secondes et fuseau horaire)`.

Voyons dès à présent les deux méthodes de classe que vous utiliserez le plus souvent :

`datetime.now()`: renvoie l'objet `datetime` avec la date et l'heure actuelles ;

`datetime.fromtimestamp(timestamp)`: renvoie la date et l'heure d'un timestamp précis.

```
import datetime
print(datetime.datetime.now())
print(datetime.datetime(2011, 2, 14, 5, 8, 22, 359000))

>> 2021-06-11 15:37:36.645296
>> 2011-02-14 05:08:22.359000
```

Il y a bien d'autres choses à voir dans ce module `datetime`. Si vous êtes curieux ou que vous avez des besoins plus spécifiques, que je n'aborde pas ici, référez-vous à la documentation officielle du module.

# Calcul du nombres de jours entre 2 dates

```
from datetime import date  
  
f_date = date(2014, 7, 2)  
l_date = date(2018, 7, 11)  
  
delta = l_date - f_date  
  
print(delta.days)
```

A la première ligne, on importe la **classe date** depuis **le module datetime**.

On crée deux instances à partir de la classe date avec deux dates différentes (2014/07/02 et 2018/07/11).

On peut ensuite récupérer ce qu'on appelle un **delta** tout simplement en soustrayant les 2 dates:

delta = l\_date – f\_date

L'objet delta récupéré contient une propriété « days » qui permet de récupérer le **nombre de jours entre les deux dates** que l'on a soustrait

- **Points importants à retenir**

Pour faire des calculs avec des dates, on utilise le module datetime et la classe date.

# Résumé

## En résumé

Le module *time* permet, entre autres, d'obtenir la date et l'heure de votre système.

La fonction *time* du module *time* renvoie le timestamp actuel.

La méthode *localtime* du module *time* renvoie un objet isolant les informations d'un timestamp (la date et l'heure).

Le module *datetime* permet de représenter des dates et heures.

Les classes *date*, *time* et *datetime* permettent respectivement de représenter des dates, des heures, ainsi que des ensembles « date et heure ».

# Timing your code with Python

# Timing your code with Python

First way to time a function:

```
import time

def powers(limit):
    return [x**2 for x in range(limit)]

start = time.time() # module time function time in that module
powers(5000000)
end = time.time()

print(end - start)

>> 1.389604091644287
```

Second way to time a function:

```
def measure_runtime(func):
    start = time.time()
    func()
    end = time.time()
    print(end - start)

def powers(limit):
    return [x ** 2 for x in range(limit)]

measure_runtime(lambda: powers(5000000))

>> 1.3915793895721436
```

# Timing your code with Python

First way to time a function:

```
import time

def powers(limit):
    return [x**2 for x in range(limit)]

start = time.time() # module time function time in that module
powers(5000000)
end = time.time()

print(end - start)

>> 1.389604091644287
```

Second way to time a function:

```
def measure_runtime(func):
    start = time.time()
    func()
    end = time.time()
    print(end - start)

def powers(limit):
    return [x ** 2 for x in range(limit)]

measure_runtime(lambda: powers(5000000))

>> 1.3915793895721436
```

# Timing your code with Python

```
import timeit

print(timeit.timeit("[x**2 for x in range(10)]"))
print(timeit.timeit("list(map(lambda x: x**2, range(10))))"))

>>> 2.5521236
>>> 2.9925559
```

So should we always use list comprehension which is faster ? The answer is: it depends what you want:

1. If you want a list of things use list comprehension, it is faster and more readable and shorter.
2. If you want to use the item of this list, one by one then use map, use the generator it gives you.

```
# The "timeit" module lets you measure the execution
# time of small bits of Python code

import timeit
timeit.timeit("-".join(str(n) for n in range(100)), number=10000)
>>> 0.3412662749997253

timeit.timeit("-".join([str(n) for n in range(100)]), number=10000)
>>> 0.2996307989997149

timeit.timeit("-".join(map(str, range(100))), number=10000)
>>> 0.24581470699922647
```

Faites de la  
programmation système

# Faites de la programmation système

Dans ce chapitre, nous allons découvrir plusieurs modules et fonctionnalités utiles pour interagir avec le système. Python peut servir à créer bien des choses, des jeux, des interfaces, mais il peut aussi faire des scripts systèmes et, dans ce chapitre, nous allons voir comment.

Les concepts que je vais présenter ici risquent d'être plus familiers aux utilisateurs de Linux. Toutefois, pas de panique si vous êtes sur Windows : je vais prendre le temps de vous expliquer à chaque fois tout le nécessaire.

# Les flux standard

Pour commencer, nous allons voir comment accéder aux flux standard (entrée standard et sortie standard) et de quelle façon nous devons les manipuler.

À quoi cela ressemble-t-il ?

Vous vous êtes sûrement habitués, quand vous utilisez la fonction print, à ce qu'un message s'affiche sur votre écran. Je pense que cela vous paraît même assez logique à présent.

Sauf que, comme pour la plupart de nos manipulations en informatique, le mécanisme qui se cache derrière nos fonctions est plus complexe et puissant qu'il y paraît. Sachez que vous pourriez très bien faire en sorte qu'en utilisant print, le texte s'écrive dans un fichier plutôt qu'à l'écran.

Quel intérêt ? print est fait pour afficher à l'écran non ?

Pas seulement, non. Mais nous verrons cela un peu plus loin. Pour l'instant, voilà ce que l'on peut dire : quand vous appelez la fonction print, si le message s'affiche à l'écran, c'est parce que la sortie standard de votre programme est redirigée vers votre écran.

On distingue trois flux standard :

- **L'entrée standard** : elle est appelée quand vous utilisez input. C'est elle qui est utilisée pour demander des informations à l'utilisateur. Par défaut, l'entrée standard est votre clavier.
- **La sortie standard** : comme on l'a vu, c'est elle qui est utilisée pour afficher des messages. Par défaut, elle redirige vers l'écran.
- **L'erreur standard** : elle est notamment utilisée quand Python vous affiche le traceback d'une exception. Par défaut, elle redirige également vers votre écran.

## Accéder aux flux standard

On peut accéder aux objets représentant ces flux standard grâce au module sys qui propose plusieurs fonctions et variables permettant d'interagir avec le système. Nous en reparlerons un peu plus loin dans ce chapitre, d'ailleurs.

# Les flux standard

## Accéder aux flux standard

On peut accéder aux objets représentant ces flux standard grâce au module sys qui propose plusieurs fonctions et variables permettant d'interagir avec le système. Nous en reparlerons un peu plus loin dans ce chapitre, d'ailleurs.

```
import sys
print(sys.stdin) # L'entrée standard (standard input)
print(sys.stdout) # La sortie standard (standard output)
print(sys.stderr) # L'erreur standard (standard error)

>>> <_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'
>>> <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'
>>> <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'
```

Ces objets ne vous rappellent rien ? Vraiment ?

Ils sont de la même classe que les fichiers ouverts grâce à la fonction open. Et il n'y a aucun hasard derrière cela.

En effet, pour lire ou écrire dans les flux standard, on utilise les méthodes read et write.

Naturellement, l'entrée standard stdin peut lire (méthode read) et les deux sorties stdout et stderr peuvent écrire (méthode write).

Essayons quelque chose :

```
sys.stdout.write("un test")
>>> un test7
```

Pas trop de surprise, sauf que ce serait mieux avec un saut de ligne à la fin. Là, ce que renvoie la méthode (le nombre de caractères écrits) est affiché juste après notre message.

```
sys.stdout.write("Un test\n")
>>> Un test
>>> 8
```

# Les flux standard

## Modifier les flux standard

Vous pouvez modifier sys.stdin, sys.stdout et sys.stderr. Faisons un premier test :

```
>>> fichier = open('sortie.txt', 'w')
>>> sys.stdout = fichier
>>> print("Quelque chose...")
```

Ici, rien ne s'affiche à l'écran. En revanche, si vous ouvrez le fichier sortie.txt, vous verrez le message que vous avez passé à print.

Je ne trouve pas le fichier sortie.txt, où est-il ?

Il doit se trouver dans le répertoire courant de Python. Pour connaître l'emplacement de ce répertoire, utilisez le module *os* et la fonction *getcwd* (Get Current Working Directory).

Une petite subtilité : si vous essayez de faire appel à *getcwd* directement, le résultat ne va pas s'afficher à l'écran... il va être écrit dans le fichier. Pour rétablir l'ancienne sortie standard, tapez la ligne :

```
sys.stdout = sys.__stdout__
```

Vous pouvez ensuite faire appel à la fonction *getcwd* :

```
import os
os.getcwd()
```

Dans ce répertoire, vous devriez trouver votre fichier sortie.txt.

Si vous avez modifié les flux standard et que vous cherchez les objets d'origine, ceux redirigeant vers le clavier (pour l'entrée) et vers l'écran (pour les sorties), vous pouvez les trouver dans sys.\_\_stdin\_\_, sys.\_\_stdout\_\_ et sys.\_\_stderr\_\_.

La documentation de Python nous conseille malgré tout de garder de préférence les objets d'origine sous la main plutôt que d'aller les chercher dans sys.\_\_stdin\_\_, sys.\_\_stdout\_\_ et sys.\_\_stderr\_\_.

Voilà qui conclut notre bref aperçu des flux standard. Là encore, si vous ne voyez pas d'application pratique à ce que je viens de vous montrer, cela viendra certainement par la suite.

# Les signaux

Les signaux sont un des moyens dont dispose le système pour communiquer avec votre programme. Typiquement, si le système doit arrêter votre programme, il va lui envoyer un signal.

Les signaux peuvent être interceptés dans votre programme. Cela vous permet de déclencher une certaine action si le programme doit se fermer (enregistrer des objets dans des fichiers, fermer les connexions réseau établies avec des clients éventuels, ...).

Les signaux sont également utilisés pour faire communiquer des programmes entre eux. Si votre programme est décomposé en plusieurs programmes s'exécutant indépendamment les uns des autres, cela permet de les synchroniser à certains moments clés. Nous ne verrons pas cette dernière fonctionnalité ici, elle mériterait un cours à elle seule tant il y aurait de choses à dire !

## Les différents signaux

Le système dispose de plusieurs signaux génériques qu'il peut envoyer aux programmes quand cela est nécessaire. Si vous demandez l'arrêt du programme, un signal particulier lui sera envoyé.

Tous les signaux ne se retrouvent pas sur tous les systèmes d'exploitation, c'est pourquoi je vais surtout m'attacher à un signal : le signal **SIGINT** envoyé à l'arrêt du programme.

Pour plus d'informations, un petit détour par la documentation s'impose, notamment du côté du [module signal](#).

## Intercepter un signal

Commencez par importer le module signal.

Le signal qui nous intéresse, comme je l'ai dit, se nomme **SIGINT**.

# Les signaux

```
import signal
```

Le signal qui nous intéresse, comme je l'ai dit, se nomme **SIGINT**.

```
signal.SIGINT  
>>> 2
```

Pour intercepter ce signal, il va falloir créer une fonction qui sera appelée si le signal est envoyé. Cette fonction prend deux paramètres :

- le signal (plusieurs signaux peuvent être envoyés à la même fonction) ;
- le **frame** qui ne nous intéresse pas ici.

Cette fonction, c'est à vous de la créer. Ensuite, il faudra la connecter avec le signal **SIGINT**.

# Les signaux

D'abord, créons notre fonction :

```
import sys

def fermer_programme(signal, frame):
    """Fonction appelée quand vient l'heure de fermer notre programme"""
    print("C'est l'heure de la fermeture !")
    sys.exit(0)
```

C'est quoi, la dernière ligne ?

On demande simplement à notre programme Python de se fermer. C'est le comportement standard quand on réceptionne un tel signal et notre programme doit bien s'arrêter à un moment ou à un autre.

Pour ce faire, on utilise la fonction `exit` (sortir, en anglais) du module `sys`. Elle prend en paramètre le code de retour du programme.

Pour simplifier, la plupart du temps, si votre programme renvoie **0**, le système comprendra que tout s'est bien passé. Si c'est un entier autre que **0**, le système interprétera cela comme une erreur ayant eu lieu pendant l'exécution de votre programme.

Ici, notre programme s'arrête normalement, on passe donc à **exit 0**.

Connectons à présent notre fonction au signal **SIGINT**, sans quoi notre fonction ne serait jamais appelée.

On utilise pour cela la fonction **signal**. Elle prend en paramètre :

- le signal à intercepter ;
- la fonction que l'on doit connecter à ce signal.

```
signal.signal(signal.SIGINT, fermer_programme)
```

Ne mettez pas les parenthèses à la fin du nom de la fonction. On envoie la référence vers la fonction, on ne l'exécute pas.

Cette ligne va connecter le signal **SIGINT** à la fonction **fermer\_programme** que vous avez définie plus haut. Dès que le système enverra ce signal pour fermer le programme, la fonction **fermer\_programme** sera appelée.

# Les signaux

Pour vérifier que tout fonctionne bien, lancez une boucle infinie dans votre programme :

```
print("Le programme va boucler...")  
while True: # Boucle infinie, True est toujours vrai  
    continue
```

Je vous remets le code en entier, si cela vous rend les choses plus claires :

```
import signal  
import sys  
  
def fermer_programme(signal, frame):  
    """Fonction appelée quand vient l'heure de fermer notre programme"""  
    print("C'est l'heure de la fermeture !")  
    sys.exit(0)  
  
# Connexion du signal à notre fonction  
signal.signal(signal.SIGINT, fermer_programme)  
  
# Notre programme...  
print("Le programme va boucler...")  
while True:  
    continue
```

Quand vous lancez ce programme, vous voyez un message vous informant que le programme va boucler... et le programme continue de tourner. Il ne s'arrête pas. Il ne fait rien, il boucle simplement mais il va continuer de boucler tant que son exécution n'est pas interrompue.

Dans la fenêtre du programme, tapez CTRL + C sur Windows ou Linux, Cmd + C sur Mac OS X.

Cette combinaison de touches va demander au programme de s'arrêter. Après l'avoir saisie, vous pouvez constater qu'effectivement, votre fonction fermer\_programme est bien appelée et s'occupe de fermer le programme correctement.

Voilà pour les signaux. Si vous voulez aller plus loin, rendez-vous sur [la documentation du module signal](#).

# Interpréter les arguments de la ligne de commande

Python nous offre plusieurs moyens, en fonction de nos besoins, pour interpréter les arguments de la ligne de commande. Pour faire court, ces arguments peuvent être des paramètres que vous passez au lancement de votre programme et qui influeront sur son exécution.

Ceux qui travaillent sur Linux n'auront, je pense, aucun mal à me suivre. Mais je vais faire une petite présentation pour ceux qui viennent de Windows, afin qu'ils puissent suivre sans difficulté.

Si vous êtes allergiques à la console, passez à la suite.

## Accéder à la console de Windows

Il existe plusieurs moyens d'accéder à la console de Windows. Celui que j'utilise et que je vais vous montrer passe par le Menu Démarrer.

Ouvrez le Menu Démarrer et cliquez sur exécuter.... Dans la fenêtre qui s'ouvre, tapez cmd puis appuyez sur Entrée.

Vous devriez vous retrouver dans une fenêtre en console, vous donnant plusieurs informations propres au système.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\utilisateur>
```

Ce qui nous intéresse, c'est la dernière ligne. C'est un chemin qui vous indique à quel endroit de l'arborescence vous vous trouvez. Il y a toutes les chances que ce chemin soit le répertoire utilisateur de votre compte.

```
C:\Documents and Settings\utilisateur>
```

# Interpréter les arguments de la ligne de commande

Nous allons commencer par nous déplacer dans le répertoire contenant l'interpréteur Python. Là encore, si vous n'avez rien changé lors de l'installation de Python, le chemin correspondant est C:\pythonXY, XY représentant les deux premiers chiffres de votre version de Python. Avec Python 3.4, ce sera donc probablement C:\python34.

Déplacez-vous dans ce répertoire grâce à la commande cd.

```
C:\Documents and Settings\utilisateur>cd C:\python34  
C:\Python34>
```

Si tout se passe bien, la dernière ligne vous indique que vous êtes bien dans le répertoire Python.

En vérité, vous pouvez appeler Python de n'importe où dans l'arborescence mais ce sera plus simple si nous sommes dans le répertoire de Python pour commencer.

## Accéder aux arguments de la ligne de commande

Nous allons une fois encore faire appel à notre module sys. Cette fois, nous allons nous intéresser à sa variable argv.  
Créez un nouveau fichier Python. Sur Windows, prenez bien soin de l'enregistrer dans le répertoire de Python (C:\python34 sous Python 3.4).

Placez-y le code suivant :

# Interpréter les arguments de la ligne de commande

```
import sys  
print(sys.argv)
```

Sur Windows :

```
C:\Python34>python test_console.py  
['test_console.py']  
C:\Python34>python test_console.py arguments  
['test_console.py', 'arguments']  
C:\Python34>python test_console.py argument1 argument2 argument3  
['test_console.py', 'argument1', 'argument2', 'argument3']  
C:\Python34>
```

# Interpréter les arguments de la ligne de commande

Comme vous le voyez, le premier élément de sys.argv contient le nom du programme, de la façon dont vous l'avez appelé. Le reste de la liste contient vos arguments (s'il y en a).

Note : vous pouvez très bien avoir des arguments contenant des espaces. Dans ce cas, vous devez alors encadrer l'argument de guillemets :

```
C:\Python34>python test_console.py "un argument avec des espaces"  
['test_console.py', 'un argument avec des espaces']  
C:\Python34>
```

## Interpréter les arguments de la ligne de commande

Accéder aux arguments, c'est bien, mais les interpréter peut être utile aussi.

### Des actions simples

Parfois, votre programme devra déclencher plusieurs actions en fonction du premier paramètre fourni. Par exemple, en premier argument, vous pourriez préciser l'une des valeurs suivantes : start pour démarrer une opération, stop pour l'arrêter, restart pour la redémarrer, status pour connaître son état... bref, les utilisateurs de Linux ont sûrement bien plus d'exemples à l'esprit.

Dans ce cas de figure, il n'est pas vraiment nécessaire d'interpréter les arguments de la ligne de commande, comme on va le voir. Notre programme Python ressemblerait simplement à cela :

# Interpréter les arguments de la ligne de commande

```
import sys

if len(sys.argv) < 2:
    print("Précisez une action en paramètre")
    sys.exit(1)

action = sys.argv[1]

if action == "start":
    print("On démarre l'opération")

elif action == "stop":
    print("On arrête l'opération")
elif action == "restart":
    print("On redémarre l'opération")
elif action == "status":
    print("On affiche l'état (démarré ou arrêté ?) de l'opération")
else:
    print("Je ne connais pas cette action")
```

# Interpréter les arguments de la ligne de commande

## Des options plus complexes

Mais la ligne de commande permet également de transmettre des arguments plus complexes comme des options. La plupart du temps, nos options sont sous la forme : -option\_courte (une seule lettre), --option\_longue, suivie d'un argument ou non.

Souvent, une option courte est accessible aussi depuis une option longue.

Ici, mon exemple va être tiré de Linux, mais vous n'avez pas vraiment besoin d'être sur Linux pour le comprendre, rassurez-vous.

La commande ls permet d'afficher le contenu d'un répertoire. On peut lui passer en paramètres plusieurs options qui influent sur ce que la commande va afficher au final.

Par exemple, pour afficher tous les fichiers (cachés ou non) du répertoire, on utilise l'option courte a.

```
$ ls -a  
. .. fichier1.txt .fichier_cache.txt image.png
```

Cette option courte est accessible depuis une option longue, all. Vous arrivez donc au même résultat en tapant :

```
$ ls --all  
. .. fichier1.txt .fichier_cache.txt image.png
```

# Interpréter les arguments de la ligne de commande

Pour récapituler, nos options courtes sont précédées d'un seul tiret et composées d'une seule lettre. Les options longues sont précédées de deux tirets et composées de plusieurs lettres.

Certaines options attendent un argument, à préciser juste après l'option.

Par exemple (toujours sur Linux), pour afficher les premières lignes d'un fichier, vous pouvez utiliser la commande head. Si vous voulez afficher les X premières lignes d'un fichier, vous utiliserez la commande head -n X.

```
$ head -n 5 fichier.txt
ligne 1
ligne 2
ligne 3
ligne 4
ligne 5
```

Dans ce cas, l'option **-n** attend un argument qui est le nombre de lignes à afficher.

## Interpréter ces options grâce à Python

Cette petite présentation faite, revenons à Python.

Nous allons nous intéresser au module **argparse** qui est utile, justement, pour interpréter les arguments de la ligne de commande selon un certain schéma. La base du code est la suivante :

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

# Interpréter les arguments de la ligne de commande

1. D'abord, on importe le module argparse ;
2. on crée ensuite un argparse.ArgumentParser qui va être utile pour configurer nos options à interpréter ;
3. enfin, on appelle la méthode parse\_args() sur notre parser. Cette méthode retourne les arguments interprétés. Nous allons voir comment préciser des options dans notre parser, pour rendre les choses plus intéressantes. Notez que, par défaut, l'interprétation des arguments se fait depuis sys.argv[1:] (c'est-à-dire la liste des arguments sans le nom du script).

En fait, notre parser n'est pas tout à fait vide. Si vous exécutez le script ci-dessus avec l'option --help :

```
>python code.py --help
usage: code.py [-h]

optional arguments:
  -h, --help  show this help message and exit
```

# Interpréter les arguments de la ligne de commande

Ce qui vous donne un petit aperçu de comment utiliser notre programme. L'aide (option -h ou --help) est générée par défaut. Et si vous n'utilisez pas le script convenablement :

```
>python code.py --inexistante  
usage: code.py [-h]  
code.py: error: unrecognized arguments: --inexistante
```

Les messages d'erreurs sont en anglais, mais vous devriez pouvoir comprendre l'erreur. Ici nous avons simplement spécifié une option qui n'a pas été définie. Essayons d'en définir une :

```
import argparse  
parser = argparse.ArgumentParser()  
parser.add_argument("x", help="le nombre à mettre au carré")  
parser.parse_args()
```

# Interpréter les arguments de la ligne de commande

Nous avons ajouté une option grâce à la méthode `add_argument()`. Elle prend plusieurs paramètres (de nombreux paramètres optionnels, en fait) mais nous n'en avons précisé que deux ici : l'option et le message d'aide lié.

Si vous demandez l'aide du script :

```
>python code.py --help
usage: code.py [-h] x

positional arguments:
  x      le nombre à mettre au carré

optional arguments:
  -h, --help  show this help message and exit
```

Nous devons maintenant préciser un nombre `x` en paramètre. Essayons de récupérer sa valeur :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", help="le nombre à mettre au carré")
args = parser.parse_args()
print("Vous avez précisé X =", args.x)
```

Pour récupérer les options (ce que nous voudrons faire la plupart du temps ;), on récupère le retour de la méthode `parse_args()`. Elle retourne un objet `namespace` avec nos options en attribut. Accéder à `args.x` retourne donc le nombre précisé par l'utilisateur :

```
python code.py 5
Vous avez précisé X = 5
```

# Interpréter les arguments de la ligne de commande

Dans ce contexte, on veut un nombre... mais l'utilisateur peut entrer n'importe quoi. Ce n'est pas une bonne chose,修改ons notre méthode add\_argument pour que l'utilisateur ne puisse entrer que des nombres :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="le nombre à mettre au carré")
args = parser.parse_args()
x = args.x
retour = x ** 2
print(retour)
```

Comme vous le voyez, la méthode add\_argument est précisée ici avec un nouvel argument : type. On lui précise int, ce qui veut dire que l'on attend un nombre (l'entrée de l'utilisateur sera automatiquement convertie).

Vous pouvez voir aussi que notre programme fait maintenant quelque chose de concret :

```
>python code.py 5
25

>python code.py -8
64

>python code.py test
usage: code.py [-h] x
code.py: error: argument x: invalid int value: 'test'
```

Comme vous le voyez, la conversion marche bien, jusqu'au message d'erreur affiché si l'utilisateur n'entre pas un nombre.

Jusqu'ici nous avons créé des "positional arguments", qui doivent être précisés sans option. Voyons comment ajouter des options facultatives :

# Interpréter les arguments de la ligne de commande

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="le nombre à mettre au carré")
parser.add_argument("-v", "--verbose", action="store_true",
    help="augmente la verbosité")
args = parser.parse_args()

x = args.x
retour = x ** 2
if args.verbose:
    print("{} ^ 2 = {}".format(x, retour))
else:
    print(retour)
```

Nous avons ajouté une nouvelle option : -v ou --verbose. Le nom commençant par un tiret, argparse suppose qu'il s'agit d'une option facultative, même si cela peut être modifié.

Notez que l'on appelle la méthode add\_argument avec l'argument action. L'action précisée, "store\_true", permet de convertir l'option précisée en booléen :

- Si l'option est précisée, alors args.verbose vaudra True ;
- si l'option n'est pas précisée, alors args.verbose vaudra False.

Le résultat affiché est différent en fonction de l'option, si elle est précisée, le message de retour est un peu plus détaillé :

# Interpréter les arguments de la ligne de commande

```
>python code.py -h  
usage: code.py [-h] [-v] xpositional arguments:  
x le nombre à mettre au carréoptional arguments:  
-h, --help show this help message and exit  
-v, --verbose augmente la verbosité>python code.py 5  
25>python code.py 5 –verbose
```

```
5 ^ 2 = 25>python code.py -v 5  
5 ^ 2 = 25>
```

Vous voyez que le retour est différent en fonction du niveau de verbosité. Notez aussi que le message d'aide intègre bien notre nouvelle option. C'est l'une des raisons (il y en a beaucoup) qui rendent l'utilisation de argparse si pratique.

Nous n'avons vu que le tout début des fonctionnalités de ce module. Si vous voulez en apprendre plus, les ressources suivantes vont bien plus loin :

- Le [tutoriel consacré à argparse](#), qui présente les fonctionnalités les plus couramment utilisées du module ;
- La [documentation officielle du module argparse](#), qui liste les fonctionnalités de manière plus complète. Je ne vous conseille pas de lire cette documentation sans lire le tutoriel avant.

# Exécuter une commande système depuis Python

Nous allons ici nous intéresser à la façon d'exécuter des commandes depuis Python. Nous allons voir deux moyens, il en existe cependant d'autres.

Ceux que je vais présenter ont l'avantage de fonctionner sur Windows.

## La fonction system

Vous vous souvenez peut-être de cette fonction du module **os**. Elle prend en paramètre une commande à exécuter, affiche le résultat de la commande et renvoie son code de retour.

```
os.system("ls") # Sur Linux  
os.system("dir") # Sur Windows
```

Vous pouvez capturer le code de retour de la commande mais vous ne pouvez pas capturer le retour affiché par la commande.

En outre, la fonction **system** exécute un environnement particulier rien que pour votre commande. Cela veut dire, entre autres, que **system** retournera tout de suite même si la commande tourne toujours.

En gros, si vous faites **os.system("sleep 5")**, le programme ne s'arrêtera pas pendant cinq secondes.

## La fonction popen

Cette fonction se trouve également dans le module **os**. Elle prend également en paramètre une commande.

Toutefois, au lieu de renvoyer le code de retour de la commande, elle renvoie un objet, un pipe (mot anglais pour un « tuyau ») qui vous permet de lire le retour de la commande.

Un exemple sur Linux :

# Exécuter une commande système depuis Python

```
import os  
cmd = os.popen("ls")  
cmd  
<os._wrap_close object at 0x7f81d16554d0>  
cmd.read()  
>>> 'fichier1.txt\nimage.png\n'
```

Le fait de lire le pipe bloque le programme jusqu'à ce que la commande ait fini de s'exécuter.

Je vous ai dit qu'il existait d'autres moyens. Et au-delà de cela, vous avez beaucoup d'autres choses intéressantes dans le module `os` vous permettant d'interagir avec le système... et pour cause !

En résumé

# Résumé

## En résumé

- Le module sys propose trois objets permettant d'accéder aux flux standard : stdin, stdout et stderr.
- Le module signal permet d'intercepter les signaux envoyés à notre programme.
- Le module argparse permet d'interpréter les arguments passés en console à notre programme.
- Enfin, le module os possède, entre autres, plusieurs fonctions pour envoyer des commandes au système.

# Les Modules

# Les Modules /

Les syntaxes et leur sens:

Depuis le module **nom\_module import nom\_fonction**

**from random import uniform**

Depuis le module random import la fonction uniform().

Dans le code, il est alors possible d'appeler directement la fonction uniform()

**Néanmoins:** il est préférable d'utiliser la syntaxe **nom\_module.fonction** soit **random.uniform()**

« Explicit is better than implicit »

➤ **random.uniform(2, 5)**

➤ **uniform(2, 5)**

Pourquoi ?

Lorsque l'on écrit: from random import uniform, on importe le nom de la fonction uniform dans l'espace globale, et si jamais on définit une fonction uniform on écrasera la fonction uniform().

!!!!!!!!!!!!!!

Pire encore !

**from random import \*** cette syntaxe est à proscrire absolument

Avec cette syntaxe nous importons toutes les fonctions contenues dans le module random sans même les connaître. Il devient alors, encore, plus facile d'écraser une fonction du module random sans même le savoir.

!!!!!!!!!!!!!!

# Les Modules

## Créer notre propre module Python

Création d'un module au même niveau que le script python, pour cela nous créons un fichier mon\_module.py dans le même répertoire que le script script.py.

script.py:

```
import mon_module  
  
print(mon_module.a)
```

n\_module.py:

```
mon_module.py:
```

```
a = 5
```

Exécution du script.py:

```
>>> 5
```

!!!!!!!!!!!!!!  
! Faire attention au nom des modules, python à une liste de noms réservés ex: random !  
!!!!!!!!!!!!!!

# Les Modules

## La Variable \_\_name\_\_

Cette variable va permettre d'exécuter le code à l'intérieur d'un module uniquement si on exécute directement ce module.

script.py:

```
import utils
```

utils.py:

```
def addition(a, b):
    return a + b

print(addition(4, 5))
```

Exécution du utils.py:

```
>>> 9
```

Exécution du script.py:

```
>>> 9
```

On peut voir que lorsqu'on exécute script.py on obtient 9, c'est normal car python exécute toujours ce qu'il importe. Par conséquent si il y a des print dans le fichier importe, les print seront affichés au moment de l'import

```
# You can get the name of
# an object's class as a
# string:
```

```
class MyClass: pass

obj = MyClass()
obj.__class__.__name__
>>> 'MyClass'
```

```
# Functions have a
# similar feature:
```

```
def myfunc(): pass

myfunc.__name__
>>> 'myfunc'
```

# Les Modules

## La Variable `__name__`

script.py:

```
import utils
```

utils.py:

```
def addition(a, b):
    return a + b

if __name__ == "__main__": # dunder name === dunder main
    print(addition(4, 5))
```

Si on exécute utils.py on obtient 9

Si on exécute script.py on a rien d'affiche car on fait un import et du coup la variable `__name__` n'est pas égale à `__main__` mais au nom du module, dans notre cas **utils**

script.py:

```
import sys
from pprint import pprint
pprint(sys.path)

# affiche les variables relative à Python, chemin jusqu'au Lib, Packages etc...
>>> ['C:\\\\Users\\\\PALLEAU JULIEN\\\\Documents\\\\GitHub\\\\TheCompletePythonCourse',
    >>> 'C:\\\\Users\\\\ PALLEAU JULIEN \\\\Documents\\\\GitHub\\\\TheCompletePythonCourse',
    >>> 'C:\\\\Users\\\\ PALLEAU JULIEN \\\\Documents\\\\mes_modules',
    >>> 'C:\\\\Users\\\\ PALLEAU JULIEN \\\\VPython\\\\VPython395\\\\python39.zip',
    >>> 'C:\\\\Users\\\\ PALLEAU JULIEN \\\\VPython\\\\VPython395\\\\DLLs',
    >>> 'C:\\\\Users\\\\ PALLEAU JULIEN \\\\VPython\\\\VPython395\\\\lib',
    >>> 'C:\\\\Users\\\\ PALLEAU JULIEN \\\\VPython\\\\VPython395',
    >>> 'C:\\\\Users\\\\ PALLEAU JULIEN \\\\AppData\\\\Roaming\\\\Python\\\\Python39\\\\site-packages',
    >>> 'C:\\\\Users\\\\ PALLEAU JULIEN \\\\VPython\\\\VPython395\\\\lib\\\\site-packages']
```

# Les Modules

## Le Python Path

Dans cmdér

```
cd c:\Julien PALLEAU\Documents  
mkdir mes_modules  
cd mes_modules
```

script.py:

```
import sys  
sys.path.append(r"C:\Users\Julien PALLEAU\Documents\mes_modules")  
  
import module_test
```

Editer module\_test.py:

```
c:\Julien PALLEAU\Documents\mes_modules  
code module_test.py # code est un alias sur visual studio sous cmdér  
# Ajouter la ligne  
print("Bonjour")
```

```
>>> Bonjour
```

Le problème de cette méthode c'est qu'il faut le faire dans tous les scripts où l'on veut utiliser le module\_test.  
Cette solution fonctionne mais ce n'est pas la plus pratique.

# Les Modules

## actualiser le module python

Actualiser un module Python

```
import importlib  
importlib.reload(utils) # cela reload le module utils et met a jour les changement qu'il y a eu dans ce module
```

Utilisez des modules de  
mathématiques

# Utilisez des modules de mathématiques

Dans ce chapitre, nous allons découvrir trois modules. Je vous ai déjà fait utiliser certains de ces modules, ce sera ici l'occasion de revenir dessus plus en détail.

- Le module `math` qui propose un bon nombre de fonctions mathématiques.
- Le module `fractions`, dont nous allons surtout voir la classe `Fraction`, permettant... vous l'avez deviné ? De modéliser des fractions.
- Et enfin le module `random` que vous connaissez de par nos TP et que nous allons découvrir plus en détail ici.

# Pour commencer, le module math

Le module math, vous le connaissez déjà : nous l'avons utilisé comme premier exemple de module créé par Python. Vous avez peut-être eu la curiosité de regarder l'aide du module pour voir quelles fonctions y étaient définies. Dans tous les cas, je fais un petit point sur certaines de ces fonctions.

Je ne vais pas m'attarder très longtemps sur ce module en particulier car il est plus vraisemblable que vous cherchiez une fonction précise et que la documentation sera, dans ce cas, plus accessible et explicite.

## Fonctions usuelles

Vous vous souvenez des opérateurs +, -, \*, / et % j'imagine, je ne vais peut-être pas y revenir.

Trois fonctions pour commencer notre petit tour d'horizon :

```
import math
print(math.pow(5, 2)) # 5 au carré
print(5 ** 2) # Pratiquement identique à pow(5, 2)
print(math.sqrt(25)) # Racine carrée de 25 (square root)
print(math.exp(5)) # Exponentielle
print(math.fabs(-3)) # Valeur absolue

>>> 25.0
>>> 25
>>> 5.0
>>> 148.4131591025766
>>> 3.0
```

Il y a bel et bien une différence entre l'opérateur `**` et la fonction `math.pow`. La fonction renvoie toujours un flottant alors que l'opérateur renvoie un entier quand cela est possible.

# Pour commencer, le module math

## Un peu de trigonométrie

Avant de voir les fonctions usuelles en trigonométrie, j'attire votre attention sur le fait que les angles, en Python, sont donnés et renvoyés en radians (rad).

Pour rappel :

Citation

1 rad = 57,29 degrés

Cela étant dit, il existe déjà dans le module math les fonctions qui vont nous permettre de convertir simplement nos angles.

```
import math
print(math.pow(5, 2)) # 5 au carré
print(5 ** 2) # Pratiquement identique à pow(5, 2)
print(math.sqrt(25)) # Racine carrée de 25 (square root)
print(math.exp(5)) # Exponentielle
print(math.fabs(-3)) # Valeur absolue
print("")

angle_en_radians = 2
angle_en_degres = 90
print(f"2 radians en degrés : {math.degrees(angle_en_radians)}") # Convertit en degrés
print(f"90 degrés en radian : {math.radians(angle_en_degres)}") # Convertit en radians

>>> 25.0
>>> 25
>>> 5.0
>>> 148.4131591025766
>>> 3.0

>>> 2 radians en degrés : 114.59155902616465
>>> 90 degrés en radian : 1.5707963267948966
```

Voyons maintenant quelques fonctions. Elles se nomment, sans surprise :

**cos** : cosinus ;  
**sin** : sinus ;  
**tan** : tangente ;  
**acos** : arc cosinus ;  
**asin** : arc sinus ;  
**atan** : arc tangente.

# Pour commencer, le module math

## Arrondir un nombre

Le module math nous propose plusieurs fonctions pour arrondir un nombre selon différents critères :

```
import math
print(math.ceil(2.3)) # Renvoie le plus petit entier >= 2.3
print(math.floor(5.8)) # Renvoie le plus grand entier <= 5.8
print(math.trunc(9.5)) # Tronque 9.5
>>> 3
>>> 5
>>> 9
```

Quant aux constantes du module, elles ne sont pas nombreuses : **math.pi** naturellement, ainsi que **math.e**.

Voilà, ce fut rapide mais suffisant, sauf si vous cherchez quelque chose de précis. En ce cas, un petit tour du côté de [la documentation officielle du module](#) **math** s'impose.

# Pour commencer, le module math

## Des fractions avec le module fractions

Ce module propose, entre autres, de manipuler des objets modélisant des **fractions**. C'est la classe **Fraction** du module qui nous intéresse :

```
from fractions import Fraction
```

### Créer une fraction

Le constructeur de la classe **Fraction** accepte plusieurs types de paramètres :

Deux entiers, le numérateur et le dénominateur (par défaut le numérateur vaut **0** et le dénominateur **1**). Si le dénominateur est **0**, une exception **ZeroDivisionError** est levée.

Une autre fraction.

Une chaîne sous la forme '**numérateur / dénominateur**'.

```
from fractions import Fraction

un_demi = Fraction(1, 2)
print(un_demi)
un_quart = Fraction(1/4)
print(un_quart)
autre_fraction = Fraction(-5, 30)
print(autre_fraction)
un_sixieme = Fraction(-1, 6)
print(un_sixieme)
```

```
>>> 1/2
>>> 1/4
>>> -1/6
>>> -1/6
```

# Pour commencer, le module math

Ne peut-on pas créer des fractions depuis un flottant ?

Si, mais pas dans le constructeur. Pour créer une fraction depuis un flottant, on utilise la méthode de classe `from_float` :

```
print(Fraction.from_float(0.5))
print(Fraction(1, 2))
>>> 1/2
>>> 1/2
```

Et pour retomber sur un flottant, rien de plus simple :

```
un_quart = Fraction(1/4)
print(float(un_quart))
>>> 0.25
```

## Manipuler les fractions

Maintenant, quel intérêt d'avoir nos nombres sous cette forme ? Surtout pour la précision des calculs. Les fractions que nous venons de voir acceptent naturellement les opérateurs usuels :

```
un_dixieme = Fraction(1, 10)
print(un_dixieme + un_dixieme + un_dixieme)
>>> 3/10
```

Alors que :

```
0.1 + 0.1 + 0.1
>>> 0.30000000000000004
```

Bien sûr, la différence n'est pas énorme mais elle est là. Tout dépend de vos besoins en termes de précision.

# Pour commencer, le module math

D'autres calculs ?

```
from fractions import Fraction

un_demi = Fraction(1/2)
un_quart = Fraction(1/4)
un_dixieme = Fraction(1, 10)
print(un_dixieme * un_quart)
print(un_dixieme + 5)
print(un_demi / un_quart)
print(un_quart / un_demi)

>>> 1/40
>>> 51/10
>>> 2
>>> 1/2
```

Voilà. Cette petite démonstration vous suffira si ce module vous intéresse. Et si elle ne suffit pas, rendez-vous sur [la documentation officielle du module fractions](#).

# Du pseudo-aléatoire avec random

## Du pseudo-aléatoire

L'ordinateur est une machine puissante, capable de faire beaucoup de choses. Mais lancer les dés n'est pas son fort. Une calculatrice standard n'a aucune difficulté à additionner, soustraire, multiplier ou diviser des nombres. Elle peut même faire des choses bien plus complexes. Mais, pour un ordinateur, choisir un nombre au hasard est bien plus compliqué qu'il n'y paraît.

Ce qu'il faut bien comprendre, c'est que derrière notre appel à `random.randrange` par exemple, Python va faire un véritable calcul pour trouver un nombre aléatoire. De ce fait, le nombre généré n'est pas réellement aléatoire puisqu'un calcul identique, effectué dans les mêmes conditions, donnera le même nombre. Cependant, les algorithmes mis en place pour générer de l'aléatoire sont maintenant suffisamment complexes pour que les nombres générés ressemblent bien à une série aléatoire. Souvenez-vous toutefois que, pour un ordinateur, le véritable hasard ne peut pas exister.

### La fonction `random`

Cette fonction, on ne l'utilisera peut-être pas souvent de manière directe mais elle est implicitement utilisée par le module quand on fait appel à `randrange` ou `choice` que nous verrons plus bas.

Elle génère un nombre pseudo-aléatoire compris entre 0 et 1. Ce sera donc naturellement un flottant :

```
import random
print(random.random())
>>> 0.9565461152605507
```

### `randrange` et `randint`

La fonction `randrange` prend trois paramètres :

- la marge inférieure de l'intervalle ;
- la marge supérieure de l'intervalle ;
- l'écart entre chaque valeur de l'intervalle (1 par défaut).

Que représente le dernier paramètre ?

# Du pseudo-aléatoire avec random

Prenons un exemple, ce sera plus simple :

```
print(random.randrange(5, 10, 2))
>>> 5
```

Cette instruction va chercher à générer un nombre aléatoire entre **5** inclus et **10** non inclus, avec un écart de **2** entre chaque valeur. Elle va donc chercher dans la liste des valeurs **[5, 7, 9]**.

Si vous ne précisez pas de troisième paramètre, il vaudra **1** par défaut (c'est le comportement attendu la plupart du temps).

La fonction **randint** prend deux paramètres :

là encore, la marge inférieure de l'intervalle ;

la marge supérieure de l'intervalle, cette fois incluse.

Pour tirer au hasard un nombre entre **1** et **6**, il est donc plus intuitif de faire :

```
print(random.randint(1, 6))
>>> 4
```

# Du pseudo-aléatoire avec random

Opérations sur des séquences

Nous allons voir deux fonctions : la première, choice, renvoie au hasard un élément d'une séquence passée en paramètre :

```
>>> random.choice(['a', 'b', 'k', 'p', 'i', 'w', 'z'])  
'k'
```

La seconde s'appelle shuffle. Elle prend en paramètre une séquence et la mélange ; elle modifie donc la séquence qu'on lui passe et ne renvoie rien :

```
>>> liste = ['a', 'b', 'k', 'p', 'i', 'w', 'z']  
>>> random.shuffle(liste)  
>>> liste  
['p', 'k', 'w', 'z', 'i', 'b', 'a']
```

Voilà. Là encore, ce fut rapide mais si vous voulez aller plus loin, vous savez où aller... droit vers [la documentation officielle de Python sur random](#).

# Résumé

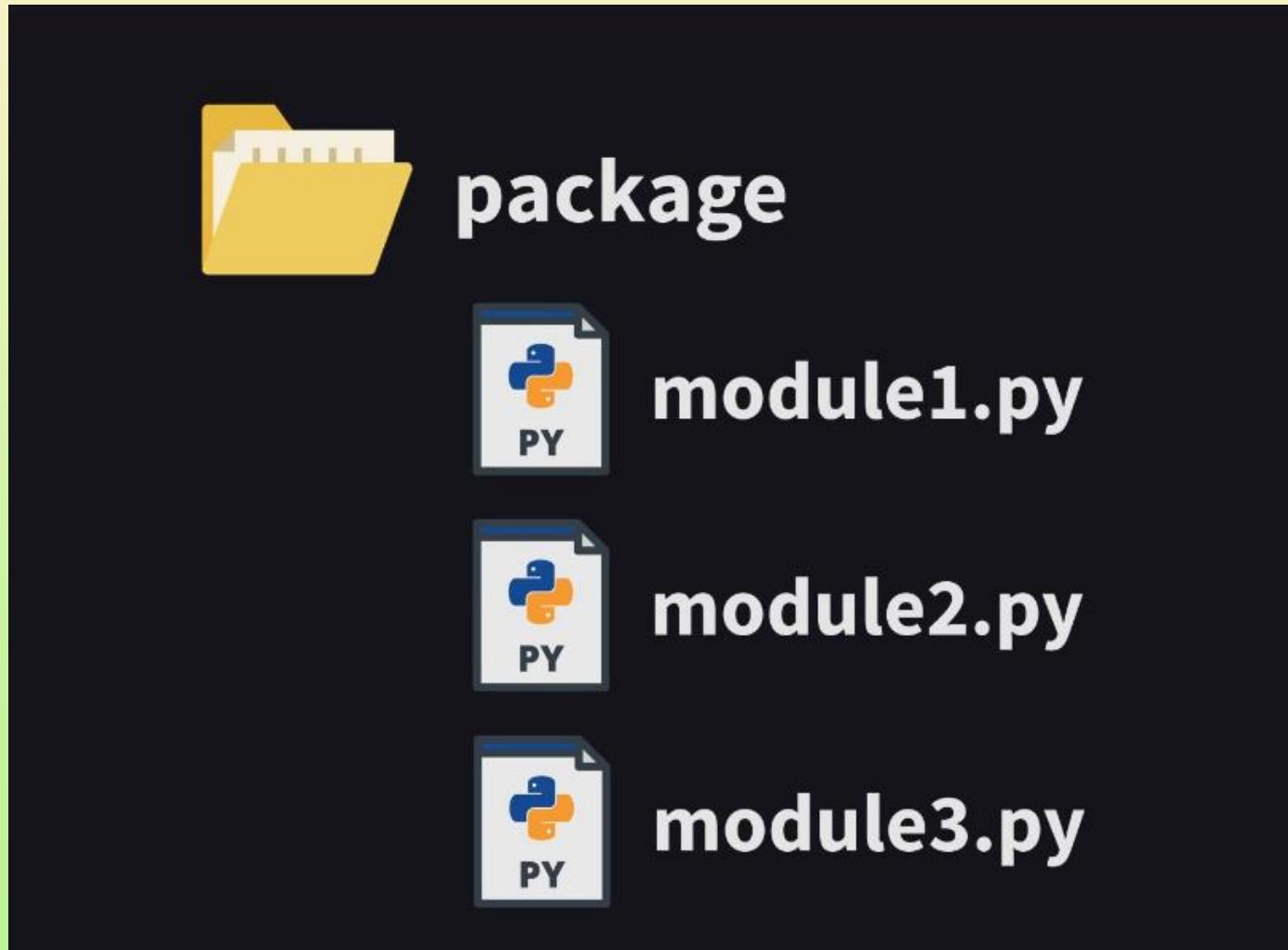
## En résumé

- Le module math possède plusieurs fonctions et constantes mathématiques usuelles.
- Le module fractions possède le nécessaire pour manipuler des fractions, parfois utiles pour la précision des calculs.
- Le module random permet de générer des nombres pseudo-aléatoires.

# Les Packages

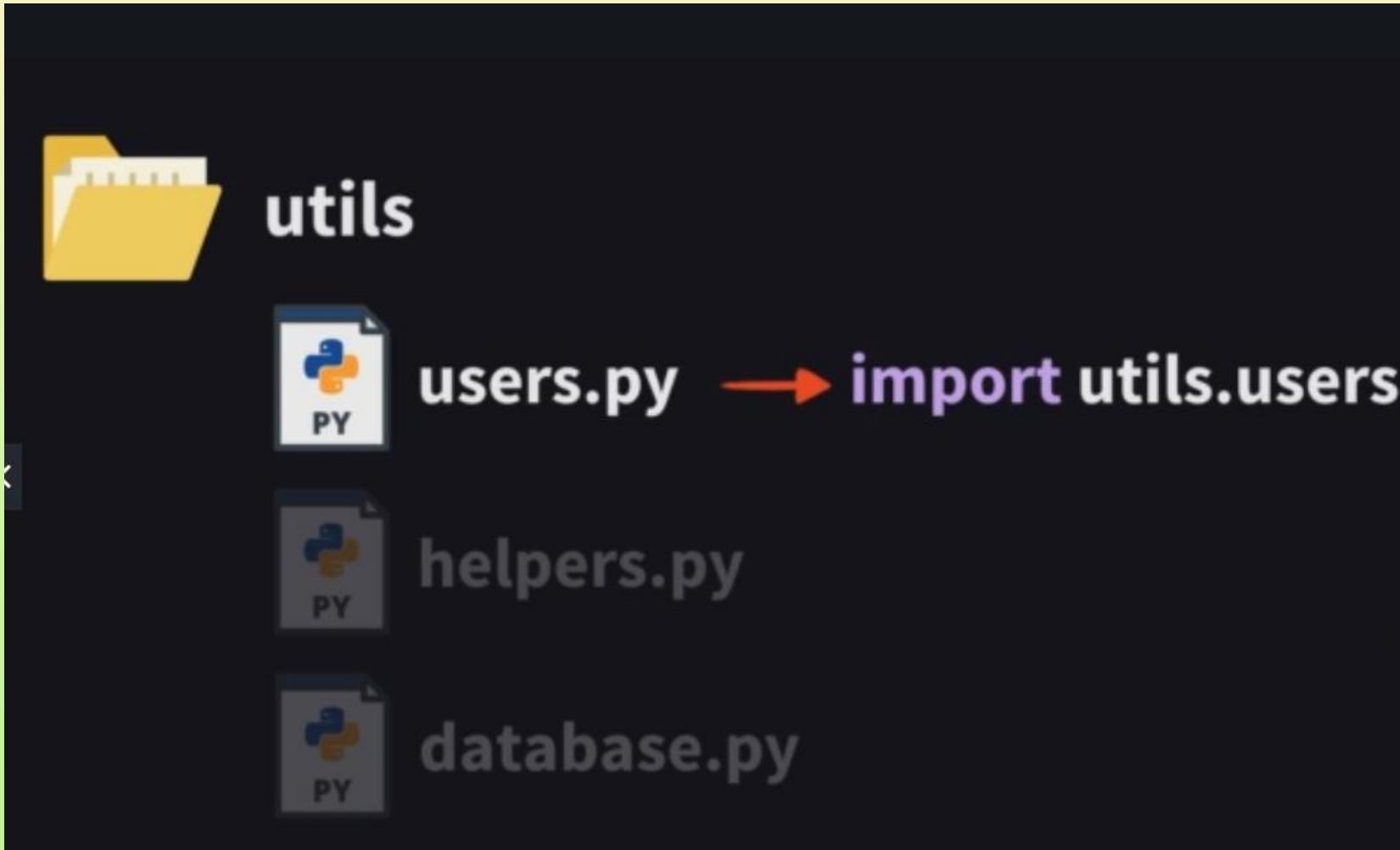
# Package

## Qu'est-ce qu'un package



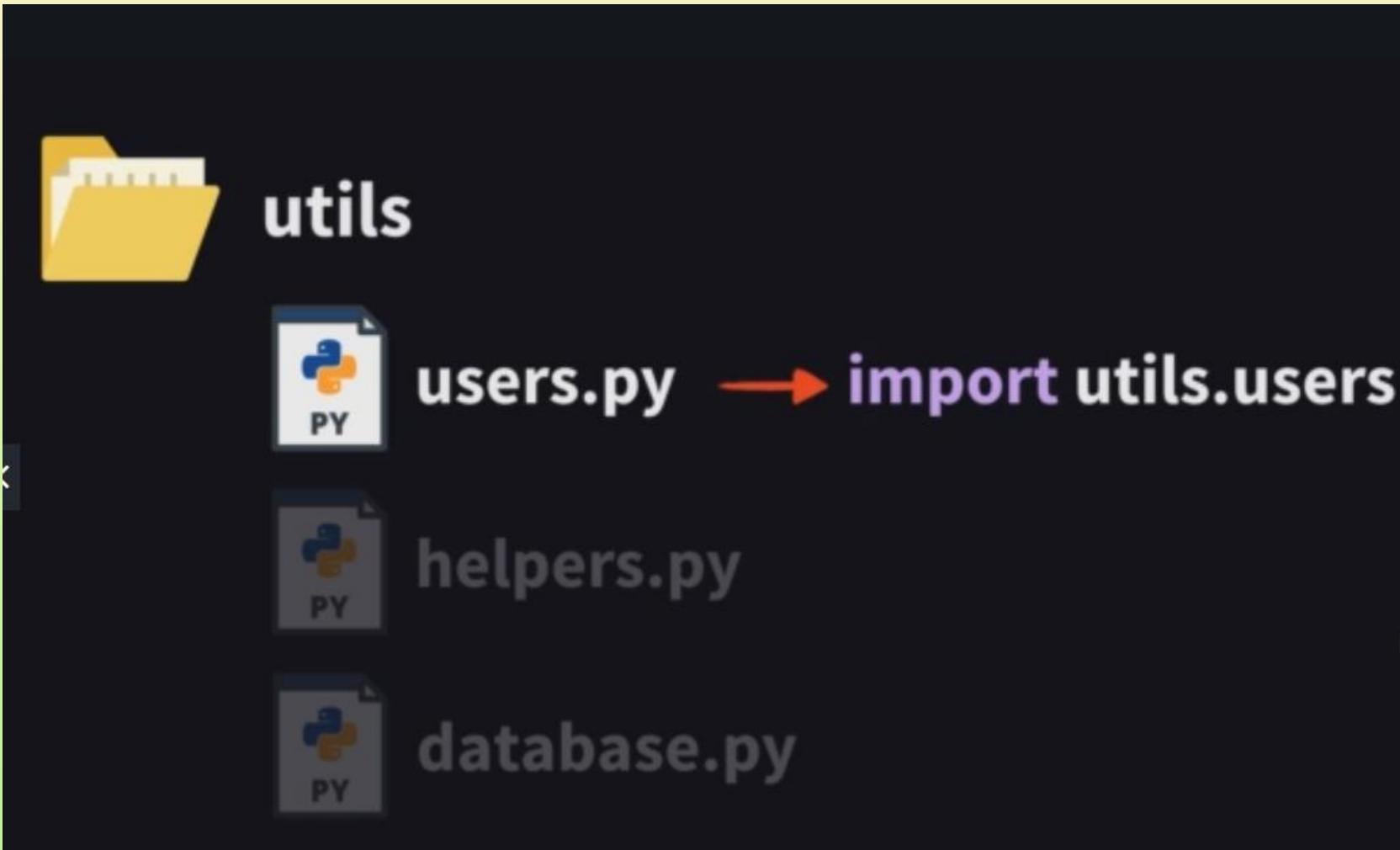
# Package

## Qu'est-ce qu'un package



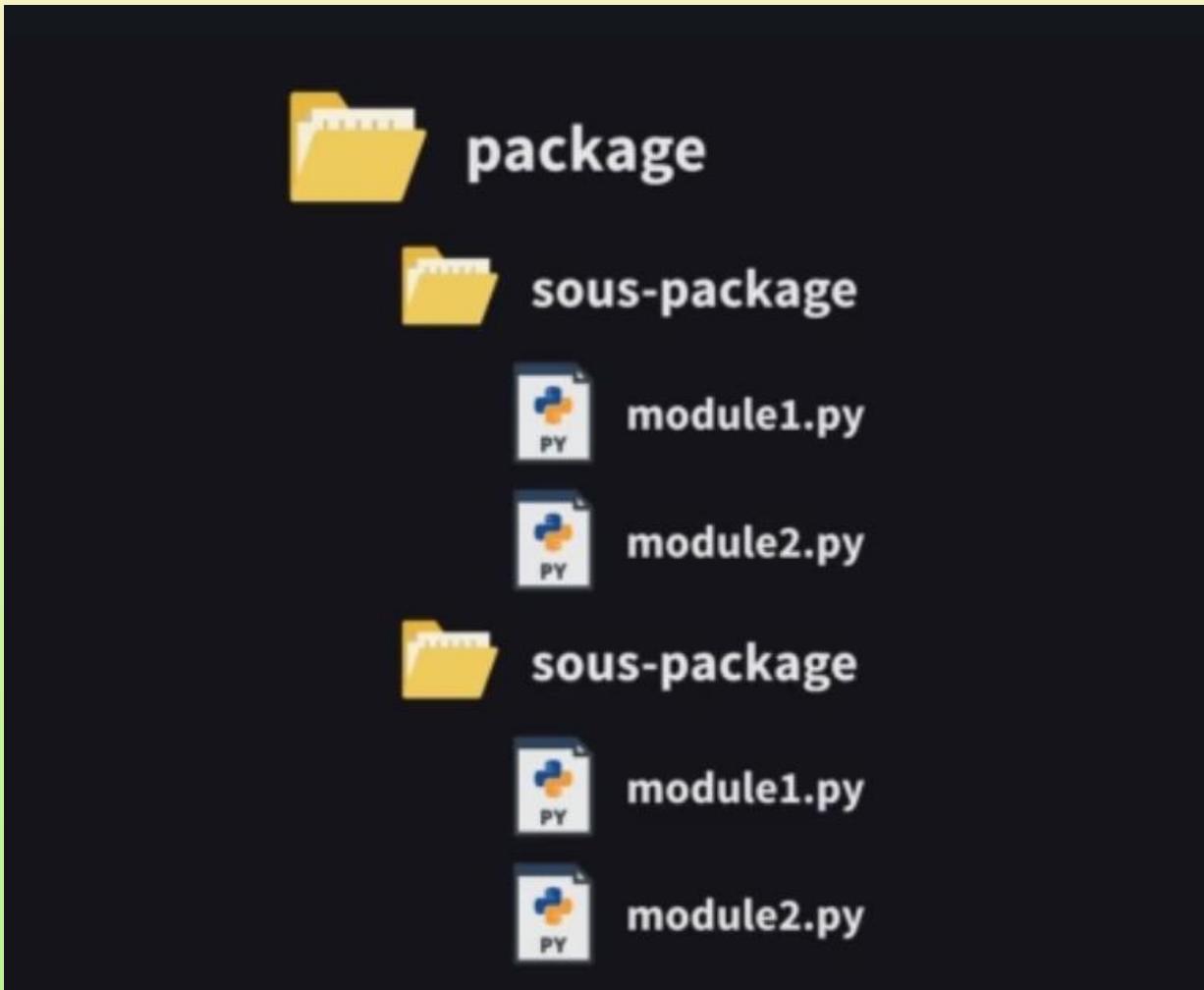
# Package

## Qu'est-ce qu'un package



# Package

## Qu'est-ce qu'un package



# Fichier xlsx

```
import openpyxl as xl
from openpyxl.chart import BarChart, Reference
from pathlib import Path

def list_files():
    path = Path()
    for file in path.glob('*.*'): # liste tous les fichiers
        print(file)

def pickup_your_file(): choosen_file = input('choisissez votre fichier :')
    return choosen_file

def process_workbook(choosen_file):
    wb = xl.load_workbook(choosen_file)
    sheet = wb['Sheet1']

    for row in range(2, sheet.max_row + 1): # we start from 2 as we don't want the first line (line with titles)
        cell = sheet.cell(row, 3)
        corrected_price = cell.value * 0.9
        corrected_price_cell = sheet.cell(row, 4)
        corrected_price_cell.value = corrected_price

    values = Reference(sheet, min_row=2, max_row=sheet.max_row, min_col=4, max_col=4)

    chart = BarChart()
    chart.add_data(values)
    sheet.add_chart(chart, 'e2')
    wb.save('transactions2.xlsx')
```

# Le Logging

# Le Logging

## Logging in the console

```
import logging
logging.basicConfig(format='%(asctime)s %(levelname)-8s %(filename)s:%(lineno)d %(message)s', level=logging.DEBUG)
logger = logging.getLogger('test_logger')

logger.info('This wil not show up.')
logger.warning('This will.')
logger.debug('This is a debug message.')
logger.debug('A critical error occured')

"""
DEBUG
INFO
WARNING
ERROR
CRITICAL
"""

>>> 2021-06-12 17:04:18,053 INFO      [test.py:5] This wil not show up.
>>> 2021-06-12 17:04:18,053 WARNING  [test.py:6] This will.
>>> 2021-06-12 17:16:21,416 DEBUG    [test.py:7] This is a debug message.
>>> 2021-06-12 17:16:21,416 DEBUG    [test.py:8] A critical error occured
```

- -8s: this is old python format and it means the level name is formatted at 8 spaces. So the filename can start at the same position.
- Then within the [] we are going to print the filename in which the log was printed out and the line number as number, what is what the d means
- And then we print the message

# Le Logging

## Logging in a file

```
import logging

logging.basicConfig(format='%(asctime)s %(levelname)-8s [%(filename)s:%(lineno)d] %(message)s',
                    datefmt='%d-%m-%Y %H:%M:%S',
                    level=logging.DEBUG,
                    filename='logs.txt'
                    )

logger = logging.getLogger('books')

logger.info('This wil not show up.')
logger.warning('This will.')
logger.debug('This is a debug message.')
logger.debug('A critical error occured')

"""
DEBUG
INFO
WARNING
ERROR
CRITICAL
"""

# different logger for another section of your program
logger = logging.getLogger('books.database') # this is child from books, so the configuration is inherited

>>> 2021-06-12 17:04:18,053 INFO      [test.py:5] This wil not show up.
>>> 2021-06-12 17:04:18,053 WARNING [test.py:6] This will.
>>> 2021-06-12 17:16:21,416 DEBUG     [test.py:7] This is a debug message.
>>> 2021-06-12 17:16:21,416 DEBUG     [test.py:8] A critical error occured
```

# Itertools

- <https://blog.teclado.com/python-itertools-part-1-product/>



Python Itertools Part 1 - Product.pdf

- <https://blog.teclado.com/python-itertools-part-2-combinations-permutations/>



Python Itertools Part 2 - Combinations & Permutations.pdf

# Further reading

- <https://www.youtube.com/watch?v=F6u5rhUQ6dU>

# Web Scraping

# Understanding HTML with BeautifulSoup

```
from bs4 import BeautifulSoup

SIMPLE_HTML = """<html>
<head></head>
<body>
<h1>This is a title</h1>
<p class="subtitle">Lorem ipsum dolor sit amet. Consectetur edipiscim elit.</p>
<p>Here's another p without a class </p>
<ul>
    <li>Rolf</li>
    <li>Charlie</li>
    <li>Jen</li>
    <li>Jose</li>
</ul>
</body>
</html>"""

simple_soup = BeautifulSoup(SIMPLE_HTML, 'html.parser')

def find_title():
    h1_tag = simple_soup.find('h1')
    print(h1_tag) # it will find the tag h1
    print(h1_tag.string) # it will return "This is a title" string from h1 tag

def find_list_items():
    list_items = simple_soup.find_all('li')
    list_contents = [e.string for e in list_items]
    print(list_contents)

def find_subtitle():
    paragraph = simple_soup.find('p', {'class': 'subtitle'})
    print(paragraph.string)

def find_other_paragraph():
    paragraphs = simple_soup.find_all('p')
    other_paragraph = [p for p in paragraphs if 'subtitle' not in p.attrs.get('class', [])] # we provide [] as second argument as the default one is NONE, and if it returns NONE which it will do in this case this will break: # if 'subtitle not in pa.attrs.get('class') which is a loop.
    print(other_paragraph)

find_title()
find_list_items()
find_subtitle()
find_other_paragraph()
```

# More complex HTML parsing

```
import re

from bs4 import BeautifulSoup

ITEM_HTML = "<html><head></head><body>
<article class='product_pod'>
    <div class='image_container'>
        <a href='catalogue/a-light-in-the-attic_1000/index.html'><img
src='media/cache/2c/da/2cdad67c44b002e7ead0cc35693c0e8b.jpg' alt='A Light in the Attic'
class='thumbnail'></a>
    </div>
    <p class='star-rating Three'>
        <i class='icon-star'></i>
        <i class='icon-star'></i>
        <i class='icon-star'></i>
        <i class='icon-star'></i>
        <i class='icon-star'></i>
    </p>
    <h3><a href='catalogue/a-light-in-the-attic_1000/index.html' title='A Light in the Artic'>A Light in
the ...</a></h3>
    <div class='product_price'>
        <p class='price_color'>£51.77</p>
    <p class='instock availability'>
        <i class='icon-ok'></i>
        In stock
    </p>
    <form>
        <button type='submit' class='btn btn-primary btn-block' data-loading-text='Adding...'>Add to
basket</button>
    </form>
    </div>
</article>
</li>
</body></html>
"""

soup = BeautifulSoup(ITEM_HTML, 'html.parser')
```

```
def find_item_name():
    locator = 'article.product_pod h3 a'
    item_link = soup.select_one(locator)
    item_name = item_link.attrs['title']
    print(item_name)

def find_item_link():
    locator = 'article.product_pod h3 a'
    item_link = soup.select_one(locator).attrs['href']
    print(item_link)

def find_item_price():
    locator = 'article.product_pod p.price_color'
    item_price = soup.select_one(locator).string # £51.77
    pattern = '£([0-9]+\\.[0-9]+)'
    matcher = re.search(pattern, item_price)
    print(matcher.group(0)) # £51.77
    print(float(matcher.group(1))) # 51.77

def find_item_rating():
    locator = 'article.product_pod p.star-rating'
    star_rating_tag = soup.select_one(locator)
    classes = star_rating_tag.attrs['class'] # ['star-rating', 'Three']
    rating_classes = [r for r in classes if r != 'star-rating'] # one way to do it
    # rating_classes = filter(lambda x: x != 'star-rating', classes) # second way to do it
    print(rating_classes[0])

find_item_name()
find_item_link()
find_item_price()
find_item_rating()
```

# More complex HTML parsing

```
import re
from bs4 import BeautifulSoup

ITEM_HTML = "<html><head></head><body>
<article class='product_pod'>
    <div class='image_container'>
        <a href='catalogue/a-light-in-the-attic_1000/index.html'><img
src='media/cache/2c/da/2cdad67c44b002e7ead0cc35693c0e8b.jpg' alt='A Light in the Attic'
class='thumbnail'></a>
    </div>
    <p class='star-rating Three'>
        <i class='icon-star'></i>
        <i class='icon-star'></i>
        <i class='icon-star'></i>
        <i class='icon-star'></i>
        <i class='icon-star'></i>
    </p>
    <h3><a href='catalogue/a-light-in-the-attic_1000/index.html' title='A Light in the Attic'>A
Light in the ...</a></h3>
    <div class='product_price'>
        <p class='price_color'>£51.77</p>
    <p class='instock availability'>
        <i class='icon-ok'></i>
        In stock
    </p>
    <form>
        <button type='submit' class='btn btn-primary btn-block' data-loading-text='Adding...'>Add to
basket</button>
    </form>
    </div>
</article>
</li>
</body></html>
"""

class ParsedItem:
    """
    A class to take in an HTML page (or part of it) and find properties of an item in it.
    """
    def __init__(self, page):
        self.soup = BeautifulSoup(page, 'html.parser')

    @property
    def name(self):
        locator = 'article.product_pod h3 a'
        item_link = self.soup.select_one(locator)
        item_name = item_link.attrs['title']
        return item_name

    @property
    def link(self):
        locator = 'article.product_pod h3 a'
        item_link = self.soup.select_one(locator).attrs['href']
        return item_link

    @property
    def price(self):
        locator = 'article.product_pod p.price_color'
        item_price = self.soup.select_one(locator).string # £51.77
        pattern = '£([0-9]+\.[0-9]+)'
        matcher = re.search(pattern, item_price)
        return float(matcher.group(1)) # 51.77

    @property
    def rating(self):
        locator = 'article.product_pod p.star-rating'
        star_rating_tag = self.soup.select_one(locator)
        classes = star_rating_tag.attrs['class'] # ['star-rating', 'Three']
        rating_classes = [r for r in classes if r != 'star-rating'] # one way to do it
        # rating_classes = filter(lambda x: x != 'star-rating', classes) # second way to do it
        return rating_classes[0]

item = ParsedItem(ITEM_HTML)
print(item.name)
```

# Splitting HTML Locators out of our Python

```
import re
from bs4 import BeautifulSoup

ITEM_HTML = """<html><head></head><body>
<article class="product_pod">
    <div class="image_container">
        <a href="catalogue/a-light-in-the-
attic_1000/index.html"></a>
    </div>
    <p class="star-rating Three">
        <i class="icon-star"></i>
        <i class="icon-star"></i>
        <i class="icon-star"></i>
        <i class="icon-star"></i>
        <i class="icon-star"></i>
    </p>
    <h3><a href="catalogue/a-light-in-the-attic_1000/index.html"
title="A Light in the Attic">A Light in the ...</a></h3>
    <div class="product_price">
        <p class="price_color">£51.77</p>
    <p class="instock availability">
        <i class="icon-ok"></i>
        In stock
    </p>
    <form>
        <button type="submit" class="btn btn-primary btn-block" data-
loading-text="Adding...">Add to basket</button>
    </form>
</div>
</article>
</li>
</body></html>
"""

class ParsedItemLocators:
```

```
"""
Locators for an item in the HTML page.
```

*This allows us to easily see what our code will be looking at as well as change it quickly if we notice it is now different.*

```
NAME_LOCATOR = 'article.product_pod h3 a'
LINK_LOCATOR = 'article.product_pod h3 a'
PRICE_LOCATOR = 'article.product_pod p.price_color'
RATING_LOCATOR = 'article.product_pod p.star-rating'
```

```
class ParsedItem:
```

*A class to take in an HTML page (or part of it) and find properties of an item in it.*

```
def __init__(self, page):
    self.soup = BeautifulSoup(page, 'html.parser')
```

```
@property
```

```
def name(self):
```

```
    locator = ParsedItemLocators.NAME_LOCATOR
    item_link = self.soup.select_one(locator)
    item_name = item_link.attrs['title']
    return item_name
```

```
@property
```

```
def link(self):
```

```
    locator = ParsedItemLocators.LINK_LOCATOR
    item_link = self.soup.select_one(locator).attrs['href']
    return item_link
```

```
@property
```

```
def price(self):
```

```
    locator = ParsedItemLocators.PRICE_LOCATOR
```

```
item_price = self.soup.select_one(locator).string # £51.77
pattern = '£([0-9]+).[0-9]+)'
matcher = re.search(pattern, item_price)
return float(matcher.group(1)) # 51.77
```

```
@property
```

```
def rating(self):
```

```
    locator = ParsedItemLocators.RATING_LOCATOR
    star_rating_tag = self.soup.select_one(locator)
    classes = star_rating_tag.attrs['class'] # ['star-rating', 'Three']
    rating_classes = [r for r in classes if r != 'star-rating'] # one way to
do it
    # rating_classes = filter(lambda x: x != 'star-rating', classes) #
second way to do it
    return rating_classes[0]
```

```
item = ParsedItem(ITEM_HTML)
print(item.name)
```

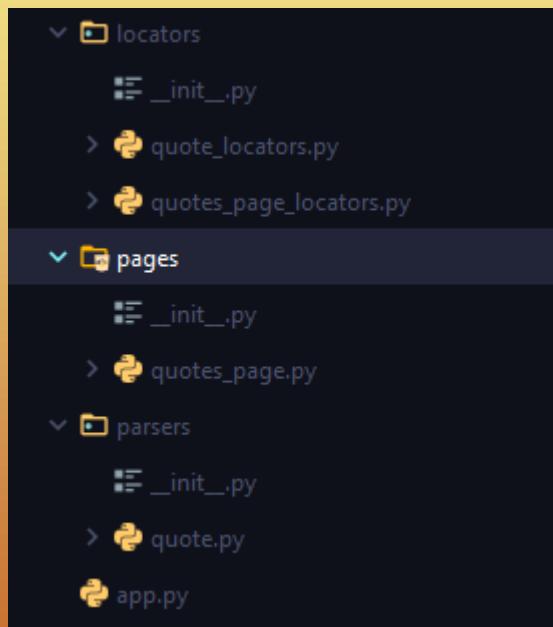
# Scraping our first website Python

```
import requests  
from bs4 import BeautifulSoup  
  
page = requests.get('http://www.example.com')  
soup = BeautifulSoup(page.content, 'html.parser')  
  
print(soup.find('h1').string)  
print(soup.select_one('p a').attrs['href'])
```

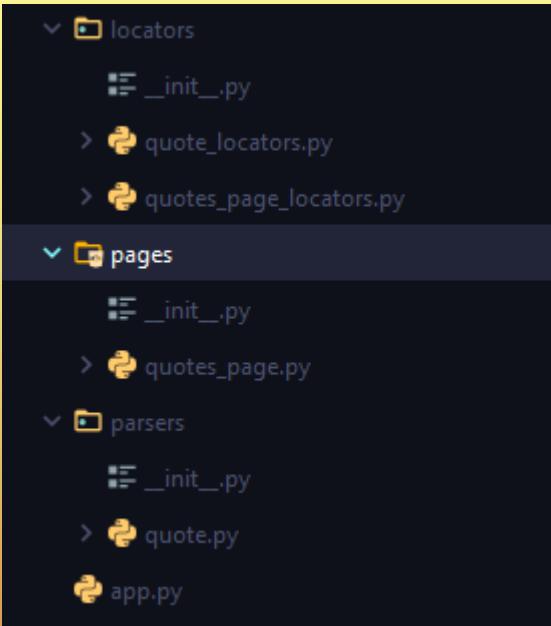
To get the html code from a web page, you need first, to install the “requests” package and secondly you get the page like in the example Above.

# Milestone Project 3: A Quote Scraper

Structure of the project



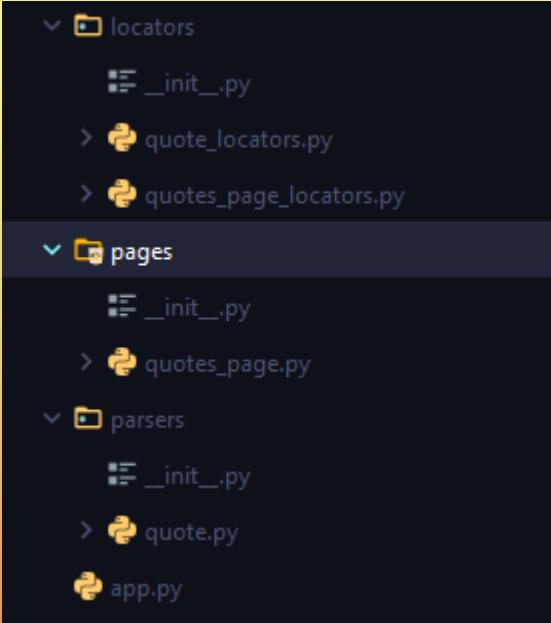
# Milestone Project 3: A Quote Scraper



quote\_locators.py

```
class QuoteLocators:  
    AUTHOR = 'small.author'  
    CONTENT = 'span.text'  
    TAGS = 'div.tags a.tag'
```

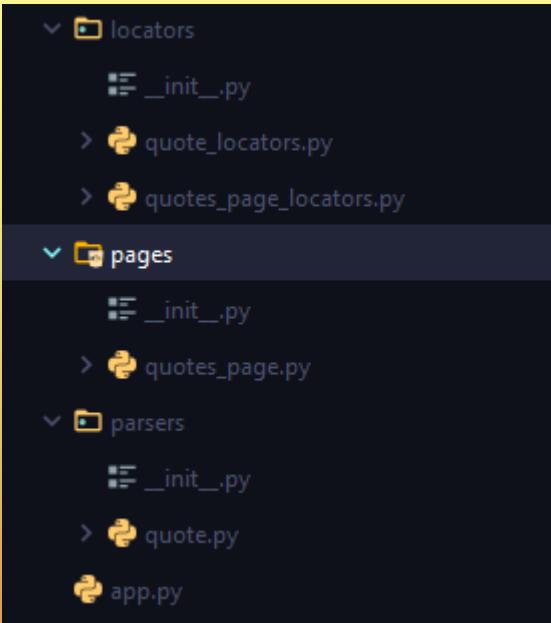
# Milestone Project 3: A Quote Scraper



quotes\_page\_locators.py

```
class QuotesPageLocators:  
    QUOTE = 'div.quote'
```

# Milestone Project 3: A Quote Scraper



quotes\_page.py

```
from bs4 import BeautifulSoup

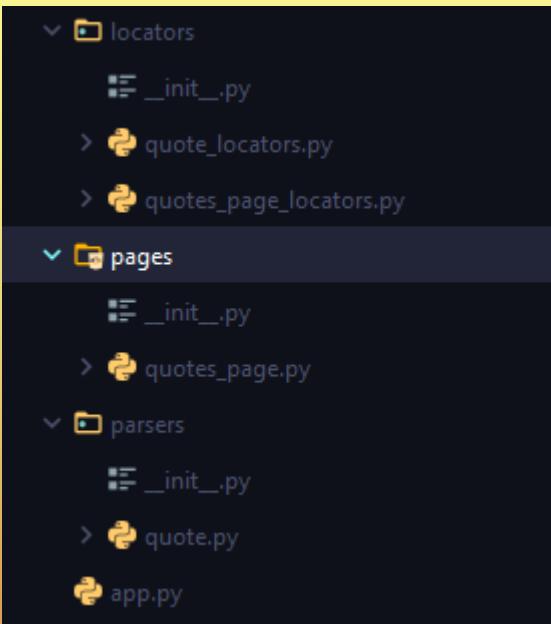
from ScrapingQuotes.locators.quotes_page_locators import QuotesPageLocators
from ScrapingQuotes.parsers.quote import QuoteParser


class QuotePage:
    def __init__(self, page):
        self.soup = BeautifulSoup(page, 'html.parser')

    @property
    def quotes(self):
        locator = QuotesPageLocators.QUOTE
        quote_tags = self.soup.select(locator)
        return [QuoteParser(e) for e in quote_tags]
```

# Milestone Project 3: A Quote Scraper

## quote.py



```
from ScrapingQuotes.locators.quote_locators import QuoteLocators

class QuoteParser:
    """
    Given one of the specific quote divs, find out the data about
    the quote (quote content, quthor, tags).
    """

    def __init__(self, parent):
        self.parent = parent

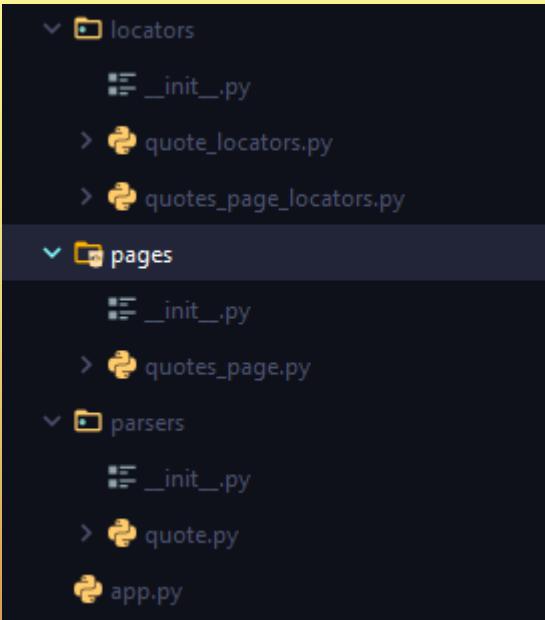
    def __repr__(self):
        return f'<Quote {self.content}, by {self.author}>'

    @property
    def content(self):
        locator = QuoteLocators.CONTENT
        return self.parent.select_one(locator).string

    @property
    def author(self):
        locator = QuoteLocators.AUTHOR
        return self.parent.select_one(locator).string

    @property
    def tags(self):
        locator = QuoteLocators.TAGS
        return [e.string for e in self.parent.select(locator)]
```

# Milestone Project 3: A Quote Scraper



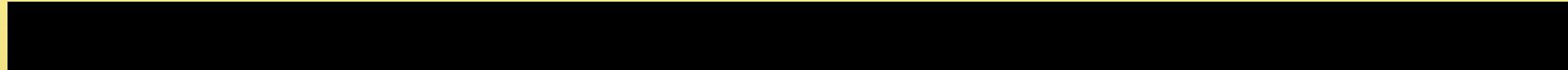
app.py

```
import requests
from ScrapingQuotes.pages.quotes_page import QuotePage

page_content = requests.get('http://quotes.toscrape.com').content
page = QuotePage(page_content)

for quote in page.quotes:
    print(quote.content)
```

# Milestone Project 4: Book Scraper



# Gérez les mots de passe

# Gérez les mots de passe

## Gérez les mots de passe

Dans ce chapitre, nous allons nous intéresser aux mots de passe et à la façon de les gérer en Python, c'est-à-dire de les réceptionner et de les protéger.

Nous allons découvrir deux modules dans ce chapitre : d'abord *getpass* qui permet de demander un mot de passe à l'utilisateur, puis *hashlib* qui permet de chiffrer le mot de passe réceptionné.

# Réceptionner un mot de passe saisi par l'utilisateur

Vous allez me dire, j'en suis sûr, qu'on a déjà une façon de réceptionner une saisie de l'utilisateur. Cette méthode, on l'a vue assez tôt dans le cours : il s'agit naturellement de la fonction input.

Mais input n'est pas très discrète. Si vous saisissez un mot de passe confidentiel, il apparaît de manière visible à l'écran, ce qui n'est pas toujours souhaitable. Quand on tape un mot de passe, c'est même rarement souhaité !

C'est ici qu'intervient le module getpass. La fonction qui nous intéresse porte le même nom que le module. Elle va réagir comme input, attendre une saisie de l'utilisateur et la renvoyer. Mais à la différence d'input, elle ne va pas afficher ce que l'utilisateur saisit.

Faisons un essai :

```
import getpass

user_name = getpass.getuser()
pass_word = getpass.getpass("Enter password: ")

print("User name: ", user_name)
print("Your password :", pass_word)
```

Comme vous le voyez... bah justement on ne voit rien ! Le mot de passe que l'on tape est invisible. Vous appuyez sur les touches de votre clavier mais rien ne s'affiche. Cependant, vous écrivez bel et bien et, quand vous appuyez sur Entrée, la fonction getpass renvoie ce que vous avez saisi.

Ici, on le stocke dans la variable mot\_de\_passe. C'est plus discret qu'input, reconnaissiez-le !

Bon, il reste un détail, mineur certes, mais un détail quand même : le prompt par défaut, c'est-à-dire le message qui vous invite à saisir votre mot de passe, est en anglais. Heureusement, il s'agit tout simplement d'un paramètre facultatif de la fonction :



Pour voir le prompt dans pycharm: Aller dans **run** dans **edit configuration** et cocher **Emulate terminal in output console**

# Réceptionner un mot de passe saisi par l'utilisateur

```
>>> from getpass import getpass  
>>> mot_de_passe = getpass()  
Password:  
>>> mot_de_passe  
'un mot de passe'
```

Comme vous le voyez... bah justement on ne voit rien ! Le mot de passe que l'on tape est invisible. Vous appuyez sur les touches de votre clavier mais rien ne s'affiche. Cependant, vous écrivez bel et bien et, quand vous appuyez sur Entrée, la fonction getpass renvoie ce que vous avez saisi.

Ici, on le stocke dans la variable `mot_de_passe`. C'est plus discret qu'`input`, reconnaisssez-le !

Bon, il reste un détail, mineur certes, mais un détail quand même : le prompt par défaut, c'est-à-dire le message qui vous invite à saisir votre mot de passe, est en anglais. Heureusement, il s'agit tout simplement d'un paramètre facultatif de la fonction :

```
>>> mot_de_passe = getpass("Tapez votre mot de passe : ")  
Tapez votre mot de passe :
```

C'est mieux.

Bien entendu, tous les mots de passe que vous réceptionnerez ne viendront pas forcément d'une saisie directe d'un utilisateur. Mais, dans ce cas précis, la fonction `getpass` est bien utile. À la fin de ce chapitre, nous verrons une utilisation complète de cette fonction, incluant réception et chiffrement de notre mot de passe en prime, deux en un.

# Chiffrer un mot de passe

Cette fois-ci, nous allons nous intéresser au module `hashlib`. Mais avant de vous montrer comment il fonctionne, quelques explications s'imposent.

## Chiffrer un mot de passe ?

La première question qu'on pourrait légitimement se poser est « pourquoi protéger un mot de passe ? ». Je suis sûr que vous pouvez trouver par vous-mêmes pas mal de réponses : il est un peu trop facile de récupérer un mot de passe s'il est stocké ou transmis en clair. Et, avec un mot de passe, on peut avoir accès à beaucoup de choses : je n'ai pas besoin de vous l'expliquer. Cela fait que généralement, quand on a besoin de stocker un mot de passe ou de le transmettre, on le chiffre.

Maintenant, qu'est-ce que le chiffrement ? A priori, l'idée est assez simple : en partant d'un mot de passe, n'importe lequel, on arrive à une seconde chaîne de caractères, complètement incompréhensible.

## Quel intérêt ?

Eh bien, si vous voyez passer, devant vos yeux, une chaîne de caractères comme `b47ea832576a75814e13351dcc97eaa985b9c6b7`, vous ne pouvez pas vraiment deviner le mot de passe qui se cache derrière.

Et l'ordinateur ne peut pas le déchiffrer si facilement que cela non plus. Bien sûr, il existe des méthodes pour déchiffrer un mot de passe mais nous ne les verrons certainement pas ici. Nous, ce que nous voulons savoir, c'est comment protéger nos mots de passe, pas comment déchiffrer ceux des autres !

## Comment fonctionne le chiffrement ?

Grave question. D'abord, il existe plusieurs techniques ou algorithmes de chiffrement. Chiffrer un mot de passe avec un certain algorithme ne donne pas le même résultat qu'avec un autre algorithme.

Ensuite, l'algorithme, quel qu'il soit, est assez complexe. Je serais bien incapable de vous expliquer en détail comment cela marche, on fait appel à beaucoup de concepts mathématiques relativement poussés.

Mais si vous voulez faire un exercice, je vous propose quelque chose d'amusant qui vous donnera une meilleure idée du chiffrement.

Commencez par numérotter toutes les lettres de l'alphabet (de a à z) de 1 à 26. Représentez l'ensemble des valeurs dans un tableau, ce sera plus simple.

# Chiffrer un mot de passe

A (1)	B (2)	C (3)	D (4)	E (5)	F (6) &
G (7)	H (8)	I (9)	J (10)	K (11)	L (12) M (13)
N (14)	O (15)	P (16)	Q (17)	R (18)	S (19) &
T (20)	U (21)	V (22)	W (23)	X (24)	Y (25) Z (26)

Maintenant, supposons que nous allons chercher à chiffrer des prénoms. Pour cela, nous allons baser notre exemple sur un calcul simple : dans le tableau ci-dessus, prenez la valeur numérique de chaque lettre constituant le prénom et additionnez l'ensemble des valeurs obtenues.

Par exemple, partons du prénom Eric. Quatre lettres, cela ira vite. Oubliez les accents, les majuscules et minuscules. On a un E (5), un R (18), un I (9) et un C (3). En ajoutant les valeurs de chaque lettre, on a donc  $5 + 18 + 9 + 3$ , ce qui donne 35.

Conclusion : en chiffrant Eric grâce à notre algorithme, on obtient le nombre 35.

C'est l'idée derrière le chiffrement même si, en réalité, les choses sont beaucoup plus complexes. En outre, au lieu d'avoir un chiffre en sortie, on a généralement plutôt une chaîne de caractères.

Mais prenez cet exemple pour vous amuser, si vous voulez. Appliquez notre algorithme à plusieurs prénoms. Si vous vous sentez d'attaque, essayez de faire une fonction Python qui prenne en paramètre notre chaîne et renvoie un chiffre, ce n'est pas bien difficile.

# Chiffrer un mot de passe

Vous pouvez maintenant vous rendre compte que derrière un nombre tel que 35, il est plutôt difficile de deviner que se cache le prénom Eric !

Si vous faites le test sur les prénoms Louis et Jacques, vous vous rendrez compte... qu'ils produisent le même résultat, 76. En effet :

$$\text{Louis} = 12 + 15 + 21 + 9 + 19 = 76$$

$$\text{Jacques} = 10 + 1 + 3 + 17 + 21 + 5 + 19 = 76$$

C'est ce qu'on appelle une collision : en prenant deux chaînes différentes, on obtient le même chiffrement au final.

Les algorithmes que nous allons voir dans le module `hashlib` essayent de minimiser, autant que possible, les collisions. Celui que nous venons juste de voir en est plein : il suffit de changer de place les lettres de notre prénom et nous retombons sur le même nombre, après tout.

Voilà. Fin de l'exercice, on va se pencher sur le module `hashlib` maintenant.

# Chiffrer un mot de passe

## Chiffrer un mot de passe

On peut commencer par importer le module hashlib :

```
import hashlib
```

On va maintenant choisir un algorithme. Pour nous aider dans notre choix, le module hashlib nous propose deux listes :

- algorithms\_guaranteed : les algorithmes garantis par Python, les mêmes d'une plateforme à l'autre. Si vous voulez faire des programmes portables, il est préférable d'utiliser un de ces algorithmes :

```
hashlib.algorithms_guaranteed
```

```
>>>{'sha256', 'sha512', 'blake2b', 'md5', 'sha3_256',
'sha384', 'sha1', 'blake2s', 'sha3_224', 'shake_256',
'sha224', 'sha3_512', 'sha3_384', 'shake_128'}
```

- algorithms\_available : les algorithmes disponibles sur votre plateforme. Tous les algorithmes garantis s'y trouvent, plus quelques autres propres à votre système.

Dans ce chapitre, nous allons nous intéresser à sha1.

Pour commencer, nous allons créer notre objet SHA1. On va utiliser le constructeur sha1 du module hashlib. Il prend en paramètre une chaîne, mais une chaîne de bytes (octets).

Pour obtenir une chaîne de bytes depuis une chaîne str, on peut utiliser la méthode encode. Je ne vais pas rentrer dans le détail des encodages ici. Pour écrire directement une chaîne bytes sans passer par une chaîne str, vous avez une autre possibilité consistant à mettre un b minuscule avant l'ouverture de votre chaîne :

# Chiffrer un mot de passe

## Chiffrer un mot de passe

On peut commencer par importer le module hashlib :

```
import hashlib
```

On va maintenant choisir un algorithme. Pour nous aider dans notre choix, le module hashlib nous propose deux listes :

- algorithms\_guaranteed : les algorithmes garantis par Python, les mêmes d'une plateforme à l'autre. Si vous voulez faire des programmes portables, il est préférable d'utiliser un de ces algorithmes :

```
hashlib.algorithms_guaranteed
```

```
>>>{'sha256', 'sha512', 'blake2b', 'md5', 'sha3_256',
'sha384', 'sha1', 'blake2s', 'sha3_224', 'shake_256',
'sha224', 'sha3_512', 'sha3_384', 'shake_128'}
```

- algorithms\_available : les algorithmes disponibles sur votre plateforme. Tous les algorithmes garantis s'y trouvent, plus quelques autres propres à votre système.

Dans ce chapitre, nous allons nous intéresser à sha1.

Pour commencer, nous allons créer notre objet SHA1. On va utiliser le constructeur sha1 du module hashlib. Il prend en paramètre une chaîne, mais une chaîne de bytes (octets).

Pour obtenir une chaîne de bytes depuis une chaîne str, on peut utiliser la méthode encode. Je ne vais pas rentrer dans le détail des encodages ici. Pour écrire directement une chaîne bytes sans passer par une chaîne str, vous avez une autre possibilité consistant à mettre un b minuscule avant l'ouverture de votre chaîne :

# Chiffrer un mot de passe

Générons notre mot de passe :

Pour obtenir le chiffrement associé à cet objet, on a deux possibilités :

- la méthode digest, qui renvoie un type bytes contenant notre mot de passe chiffré ;
- la méthode hexdigest, qui renvoie une chaîne str contenant une suite de symboles hexadécimaux (de 0 à 9 et de A à F).

C'est cette dernière méthode que je vais montrer ici, parce qu'elle est préférable pour un stockage en fichier si les fichiers doivent transiter d'une plateforme à l'autre.

```
mot_de_passe.hexdigest()  
>>> 'b47ea832576a75814e13351dcc97eaa985b9c6b7'
```

Et pour déchiffrer ce mot de passe ?

On ne le déchiffre pas. Si vous voulez savoir si le mot de passe saisi par l'utilisateur correspond au chiffrement que vous avez conservé, chiffrer le mot de passe qui vient d'être saisi et comparez les deux chiffrements obtenus :

# Chiffrer un mot de passe

```
import hashlib
from getpass import getpass

chaine_mot_de_passe = b"azerty"
mot_de_passe_chiffre = hashlib.sha1(chaine_mot_de_passe).hexdigest()

verrouille = True
while verrouille:
    entre = getpass("Tapez le mot de passe : ") # azerty
    # On encode la saisie pour avoir un type bytes
    entre = entre.encode()

    entre_chiffre = hashlib.sha1(entre).hexdigest()
    if entre_chiffre == mot_de_passe_chiffre:
        verrouille = False
    else:
        print("Mot de passe incorrect")

print("Mot de passe accepté...")
```

```
>>> Tapez le mot de passe :
>>> Mot de passe accepté...
```

Cela me semble assez clair. Nous avons utilisé l'algorithme sha1, il en existe d'autres comme vous pouvez le voir dans hashlib. Algorithms\_available.

Je m'arrête pour ma part ici ; si vous voulez aller plus loin, je vous redirige vers la documentation de Python sur les modules getpass et hashlib.

# Résumé

## En résumé

Pour demander à l'utilisateur de saisir un mot de passe, on peut utiliser le module **getpass**.

La fonction **getpass** du module **getpass** fonctionne de la même façon que **input**, sauf qu'elle n'affiche pas ce que l'utilisateur saisit.

Pour chiffrer un mot de passe, on va utiliser le module **hashlib**.

Ce module contient en attributs les différents algorithmes pouvant être utilisés pour chiffrer nos mots de passe.

# Gérer les réseaux

# Gérer les réseaux

## Gérez les réseaux

Vaste sujet que le réseau ! Si je devais faire une présentation détaillée, ou même parler des réseaux en général, il me faudrait bien plus d'un chapitre rien que pour la théorie.

Dans ce chapitre, nous allons donc apprendre à faire communiquer deux applications grâce aux sockets, des objets qui permettent de connecter un client à un serveur et de transmettre des données de l'un à l'autre.

Si cela ne vous met pas l'eau à la bouche...

# Brève présentation du réseau

Comme je l'ai dit plus haut, le réseau est un sujet bien trop vaste pour que je le présente en un unique chapitre. On va s'attacher ici à comprendre comment faire communiquer deux applications, qui peuvent être sur la même machine mais aussi sur des machines distantes. Dans ce cas, elles se connectent grâce au réseau local ou à Internet.

Il existe plusieurs protocoles de communication en réseau. Si vous voulez, c'est un peu comme la communication orale : pour que les échanges se passent correctement, les deux (ou plus) parties en présence doivent parler la même langue. Nous allons ici parler du protocole **TCP**.

## Le protocole TCP

L'acronyme de ce protocole signifie *Transmission Control Protocol*, soit « protocole de contrôle de transmission ». Concrètement, il permet de connecter deux applications et de leur faire échanger des informations.

Ce protocole est dit « orienté connexion », c'est-à-dire que les applications sont connectées pour communiquer et que l'on peut être sûr, quand on envoie une information au travers du réseau, qu'elle a bien été réceptionnée par l'autre application. Si la connexion est rompue pour une raison quelconque, les applications doivent rétablir la connexion pour communiquer de nouveau.

Cela vous paraît peut-être évident mais le protocole **UDP** (*User Datagram Protocol*), par exemple, envoie des informations au travers du réseau sans se soucier de savoir si elles seront bien réceptionnées par la cible. Ce protocole n'est pas connecté, une application envoie quelque chose au travers du réseau en spécifiant une cible. Il suffit alors de prier très fort pour que le message soit réceptionné correctement !

Plus sérieusement, ce type de protocole est utile si vous avez besoin de faire transiter beaucoup d'informations au travers du réseau mais qu'une petite perte occasionnelle d'informations n'est pas très handicapante. On trouve ce type de protocole dans des jeux graphiques en réseau, le serveur envoyant très fréquemment des informations au client pour qu'il actualise sa fenêtre. Cela fait beaucoup à transmettre mais ce n'est pas dramatique s'il y a une petite perte d'informations de temps à autre puisque, quelques millisecondes plus tard, le serveur renverra de nouveau les informations.

En attendant, c'est le protocole **TCP** qui nous intéresse. Il est un peu plus lent que le protocole **UDP** mais plus sûr et, pour la quantité d'informations que nous allons transmettre, il est préférable d'être sûr des informations transmises plutôt que de la vitesse de transmission.

# Brève présentation du réseau

## Clients et serveur

Dans l'architecture que nous allons voir dans ce chapitre, on trouve en général un serveur et plusieurs clients. Le serveur, c'est une machine qui va traiter les requêtes du client.

Si vous accédez par exemple à OpenClassrooms, c'est parce que votre navigateur, faisant office de client, se connecte au serveur d'OpenClassrooms. Il lui envoie un message en lui demandant la page que vous souhaitez afficher et le serveur d'OpenClassrooms, dans sa grande bonté, envoie la page demandée au client.

Cette architecture est très fréquente, même si ce n'est pas la seule envisageable.

Dans les exemples que nous allons voir, nous allons créer deux applications : l'**application serveur** et l'**application client**. Le serveur écoute donc en attendant des connexions et les clients se connectent au serveur.

# Brève présentation du réseau

## Les différentes étapes

Nos applications vont fonctionner selon un schéma assez similaire. Voici dans l'ordre les étapes du client et du serveur. Les étapes sont très simplifiées, la plupart des serveurs peuvent communiquer avec plusieurs clients mais nous ne verrons pas cela tout de suite.

Le serveur :

1. attend une connexion de la part du client ;
2. accepte la connexion quand le client se connecte ;
3. échange des informations avec le client ;
4. ferme la connexion.

Le client :

1. se connecte au serveur ;
2. échange des informations avec le serveur ;
3. ferme la connexion.

Comme on l'a vu, le serveur peut dialoguer avec plusieurs clients : c'est tout l'intérêt. Si le serveur d'OpenClassrooms ne pouvait dialoguer qu'avec un seul client à la fois, il faudrait attendre votre tour, peut-être assez longtemps, avant d'avoir accès à vos pages. Et, sans serveur pouvant dialoguer avec plusieurs clients, les jeux en réseau ou les logiciels de messagerie instantanée seraient bien plus complexes.

# Brève présentation du réseau

## Établir une connexion

Pour que le client se connecte au serveur, il nous faut deux informations :

Le nom d'hôte (host name en anglais), qui identifie une machine sur Internet ou sur un réseau local. Les noms d'hôtes permettent de représenter des adresses IP de façon plus claire (on a un nom comme google.fr, plus facile à retenir que l'adresse IP correspondante 74.125.224.84).

Un numéro de port, qui est souvent propre au type d'information que l'on va échanger. Si on demande une connexion web, le navigateur va en général interroger le port 80 si c'est en http ou le port 443 si c'est en connexion sécurisée (https). Le numéro de port est compris entre 0 et 65535 (il y en a donc un certain nombre !) et les numéros entre 0 et 1023 sont réservés par le système. On peut les utiliser, mais ce n'est pas une très bonne idée.

Pour résumer, quand votre navigateur tente d'accéder à OpenClassrooms, il établit une connexion avec le serveur dont le nom d'hôte est fr.openclassrooms.com sur le port 80. Dans ce chapitre, nous allons plus volontiers travailler avec des noms d'hôtes qu'avec des adresses IP.

## Les sockets

Comme on va le voir, les sockets sont des objets qui permettent d'ouvrir une connexion avec une machine locale ou distante et d'échanger avec elle.

Ces objets sont définis dans le module socket et nous allons maintenant voir comment ils fonctionnent.

# Les sockets

Commençons donc, dans la joie et la bonne humeur, par importer notre module socket

```
import socket
```

Nous allons d'abord créer notre serveur puis, en parallèle, un client. Nous allons faire communiquer les deux. Pour l'instant, nous nous occupons du serveur.

## Construire notre socket

Nous allons pour cela faire appel au constructeur socket. Dans le cas d'une connexion TCP, il prend les deux paramètres suivants, dans l'ordre :

socket.AF\_INET : la famille d'adresses, ici ce sont des adresses Internet ;

socket.SOCK\_STREAM : le type du socket, SOCK\_STREAM pour le protocole TCP.

```
connexion_principale = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

## Connecter le socket

Ensuite, nous connectons notre socket. Pour une connexion serveur, qui va attendre des connexions de clients, on utilise la méthode **bind**. Elle prend un paramètre : le tuple (**nom\_hôte, port**).

Attends un peu, je croyais que c'était notre client qui se connectait à notre serveur, pas l'inverse...

Oui mais, pour que notre serveur écoute sur un port, il faut le configurer en conséquence. Donc, dans notre cas, le nom de l'hôte sera vide et le port sera celui que vous voulez, entre 1024 et 65535.

```
connexion_principale.bind("", 12800)
```

# Les sockets

## Faire écouter notre socket

Bien. Notre socket est prêt à écouter sur le port 12800 mais il n'écoute pas encore. On va avant tout lui préciser le nombre maximum de connexions qu'il peut recevoir sur ce port sans les accepter. On utilise pour cela la méthode `listen`. On lui passe généralement 5 en paramètre.

Cela veut dire que notre serveur ne pourra dialoguer qu'avec 5 clients maximum ?

Non. Cela veut dire que si 5 clients se connectent et que le serveur n'accepte aucune de ces connexions, aucun autre client ne pourra se connecter. Mais généralement, très peu de temps après que le client ait demandé la connexion, le serveur l'accepte. Vous pouvez donc avoir bien plus de clients connectés, ne vous en faites pas.

```
connexion_principale.listen(5)
```

# Les sockets

## Accepter une connexion venant du client

Enfin, dernière étape, on va accepter une connexion. Aucune connexion ne s'est encore présentée mais la méthode **accept** que nous allons utiliser va bloquer le programme tant qu'aucun client ne s'est connecté.

Il est important de noter que la méthode **accept** renvoie deux informations :

le socket connecté qui vient de se créer, celui qui va nous permettre de dialoguer avec notre client tout juste connecté ;

un tuple représentant l'adresse IP et le port de connexion du client.

Le port de connexion du client... ce n'est pas le même que celui du serveur ?

Non car votre client, en ouvrant une connexion, passe par un port dit « de sortie » qui va être choisi par le système parmi les ports disponibles. Pour schématiser, quand un client se connecte à un serveur, il emprunte un port (une forme de porte, si vous voulez) puis établit la connexion sur le port du serveur. Il y a donc deux ports dans notre histoire mais celui qu'utilise le client pour ouvrir sa connexion ne va pas nous intéresser.

```
connexion_avec_client, infos_connexion = connexion_principale.accept()
```

Cette méthode, comme vous le voyez, bloque le programme. Elle attend qu'un client se connecte. Laissons cette fenêtre Python ouverte et, à présent, ouvrons-en une nouvelle pour construire notre client.

## Création du client

Commencez par construire votre socket de la même façon :

```
import socket
connexion_avec_serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

# Les sockets

## Connecter le client

Pour se connecter à un serveur, on va utiliser la méthode `connect`. Elle prend en paramètre un tuple, comme `bind`, contenant le nom d'hôte et le numéro du port identifiant le serveur auquel on veut se connecter. Le numéro du port sur lequel on veut se connecter, vous le connaissez : c'est 12800. Vu que nos deux applications Python sont sur la même machine, le nom d'hôte va être `localhost` (c'est-à-dire la machine locale).

```
connexion_avec_serveur.connect(('localhost', 12800))
```

Et voilà, notre serveur et notre client sont connectés !

Si vous retournez dans la console Python abritant le serveur, vous pouvez constater que la méthode `accept` ne bloque plus, puisqu'elle vient d'accepter la connexion demandée par le client. Vous pouvez donc de nouveau saisir du code côté serveur :

```
>>> print(infos_connexion)
('127.0.0.1', 2901)
```

La première information, c'est l'adresse IP du client. Ici, elle vaut `127.0.0.1` c'est-à-dire l'IP de l'ordinateur local. Dites-vous que l'hôte `localhost` redirige vers l'IP `127.0.0.1`. Le second est le port de sortie du client, qui ne nous intéresse pas ici.

## Faire communiquer nos sockets

Bon, maintenant, comment faire communiquer nos sockets ? Eh bien, en utilisant les méthodes `send` pour envoyer et `recv` pour recevoir.

Les informations que vous transmettrez seront des chaînes de bytes, pas des str !

Donc côté serveur :

```
connexion_avec_client.send(b"Je viens d'accepter la connexion")
```

# Les sockets

La méthode *send* vous renvoie le nombre de caractères envoyés.

Maintenant, côté client, on va réceptionner le message que l'on vient d'envoyer. La méthode *recv* prend en paramètre le nombre de caractères à lire. Généralement, on lui passe la valeur 1024. Si le message est plus grand que 1024 caractères, on récupérera le reste après.

Dans la fenêtre Python côté client, donc :

```
msg_recu = connexion_avec_serveur.recv(1024)
msg_recu
>>> b"Je viens d'accepter la connexion"
```

Magique, non ? Vraiment pas ? Songez que ce petit mécanisme peut servir à faire communiquer des applications entre elles non seulement sur la machine locale, mais aussi sur des machines distantes et reliées par Internet.

Le client peut également envoyer des informations au serveur et le serveur peut les réceptionner, tout cela grâce aux méthodes *send* et *recv* que nous venons de voir.  
Fermer la connexion

Pour fermer la connexion, il faut appeler la méthode *close* de notre socket.

Côté serveur :

```
connexion_avec_client.close()
```

Et côté client :

```
connexion_avec_serveur.close()
```

Voilà ! Je vais récapituler en vous présentant dans l'ordre un petit serveur et un client que nous pouvons utiliser. Et pour finir, je vous montrerai une façon d'optimiser un peu notre serveur en lui permettant de gérer plusieurs clients à la fois.

# Le serveur

Pour éviter les confusions, je vous remets ici le code du serveur, légèrement amélioré. Il n'accepte qu'un seul client (nous verrons plus bas comment en accepter plusieurs) et il tourne jusqu'à recevoir du client le message fin.

À chaque fois que le serveur reçoit un message, il envoie en retour le message '5 / 5'.

```
import socket

hote = ""
port = 12800

connexion_principale = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connexion_principale.bind((hote, port))
connexion_principale.listen(5)
print("Le serveur écoute à présent sur le port {}".format(port))

connexion_avec_client, infos_connexion = connexion_principale.accept()

msg_recu = b""
while msg_recu != b"fin":
    msg_recu = connexion_avec_client.recv(1024)
    # L'instruction ci-dessous peut lever une exception si le message
    # Réceptionné comporte des accents
    print(msg_recu.decode())
    connexion_avec_client.send(b"5 / 5")

print("Fermeture de la connexion")
connexion_avec_client.close()
connexion_principale.close()
```

Voilà pour le serveur. Il est minimal, vous en conviendrez, mais il est fonctionnel. Nous verrons un peu plus loin comment l'améliorer.

# Le client

Là encore, je vous propose le code du client pouvant interagir avec notre serveur.

Il va tenter de se connecter sur le port 12800 de la machine locale. Il demande à l'utilisateur de saisir quelque chose au clavier et envoie ce quelque chose au serveur, puis attend sa réponse.

```
import socket

hote = "localhost"
port = 12800

connexion_avec_serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connexion_avec_serveur.connect((hote, port))
print("Connexion établie avec le serveur sur le port {}".format(port))

msg_a_envoyer = b""
while msg_a_envoyer != b"fin":
    msg_a_envoyer = input("> ")
    # Peut planter si vous tapez des caractères spéciaux
    msg_a_envoyer = msg_a_envoyer.encode()
    # On envoie le message
    connexion_avec_serveur.send(msg_a_envoyer)
    msg_recu = connexion_avec_serveur.recv(1024)
    print(msg_recu.decode()) # Là encore, peut planter s'il y a des accents

print("Fermeture de la connexion")
connexion_avec_serveur.close()
```

Que font les méthodes encode et decode ?

# Le client

`encode` est une méthode de `str`. Elle peut prendre en paramètre un nom d'encodage et permet de passer un `str` en chaîne `bytes`. C'est, comme vous le savez, ce type de chaîne que `send` accepte. En fait, `encode` la chaîne `str` en fonction d'un encodage précis (par défaut, **Utf-8**).

`decode`, à l'inverse, est une méthode de `bytes`. Elle aussi peut prendre en paramètre un encodage et elle renvoie une chaîne `str` décodée grâce à l'encodage (par défaut **Utf-8**).

Si l'encodage de votre console est différent d'**Utf-8** (ce sera souvent le cas sur Windows), des erreurs peuvent se produire si les messages que vous encodez ou décodez comportent des accents.

Voilà, nous avons vu un serveur et un client, tous deux très simples. Maintenant, voyons quelque chose de plus élaboré !

# Un serveur plus élaboré

Quel sont les problèmes de notre serveur ?

Si vous y réfléchissez, il y en a pas mal !

- D'abord, notre serveur ne peut accepter qu'un seul client. Si d'autres clients veulent se connecter, ils ne peuvent pas.
- Ensuite, on part toujours du principe qu'on attend le message d'un client et qu'on lui renvoie immédiatement après réception. Mais ce n'est pas toujours le cas : parfois vous envoyez un message au client alors qu'il ne vous a rien envoyé, parfois vous recevez des informations de sa part alors que vous ne lui avez rien envoyé.

Prenez un logiciel de messagerie instantanée : est-ce que, pour dialoguer, vous êtes obligés d'attendre que votre interlocuteur vous réponde ? Ce n'est pas « j'envoie un message, il me répond, je lui réponds, il me répond »... Parfois, souvent même, vous enverrez deux messages à la suite, peut-être même trois, ou l'inverse, qui sait ? Bref, on doit pouvoir envoyer plusieurs messages au client et réceptionner plusieurs messages dans un ordre inconnu. Avec notre technique, c'est impossible (faites le test si vous voulez).

En outre, les erreurs sont assez mal gérées, vous en conviendrez.

## Le module select

Le module *select* va nous permettre une chose très intéressante, à savoir interroger plusieurs clients dans l'attente d'un message à réceptionner, sans paralyser notre programme.

Pour schématiser, *select* va écouter sur une liste de clients et retourner au bout d'un temps précisé. Ce que renvoie *select*, c'est la liste des clients qui ont un message à réceptionner. Il suffit de parcourir ces clients, de lire les messages en attente (grâce à *recv*) et le tour est joué.

Sur Linux, *select* peut être utilisé sur autre chose que des sockets mais, cette fonctionnalité n'étant pas portable, je ne fais que la mentionner ici.

# Un serveur plus élaboré

## En théorie

La fonction qui nous intéresse porte le même nom que le module associé, *select*. Elle prend trois ou quatre arguments et en renvoie trois. C'est maintenant qu'il faut être attentif :

Les arguments que prend la fonction sont :

- *rlist* : la liste des sockets en attente d'être lus ;
- *wlist* : la liste des sockets en attente d'être écrits ;
- *xlist* : la liste des sockets en attente d'une erreur (je ne m'attarderai pas sur cette liste) ;
- *timeout* : le délai pendant lequel la fonction attend avant de retourner. Si vous précisez en *timeout* 0, la fonction retourne immédiatement. Si ce paramètre n'est pas précisé, la fonction retourne dès qu'un des sockets change d'état (est prêt à être lu s'il est dans *rlist* par exemple) mais pas avant.

Concrètement, nous allons surtout nous intéresser au premier et au quatrième paramètre. En effet, *wlist* et *xlist* ne nous intéresseront pas présentement.

Ce qu'on veut, c'est mettre des sockets dans une liste et que *select* les surveille, en retournant dès qu'un socket est prêt à être lu. Comme cela notre programme ne bloque pas et il peut recevoir des messages de plusieurs clients dans un ordre complètement inconnu.

Maintenant, concernant le *timeout* : comme je vous l'ai dit, si vous ne le précisez pas, *select* bloque jusqu'au moment où l'un des sockets que nous écoutons est prêt à être lu, dans notre cas. Si vous précisez un *timeout* de 0, *select* retournera tout de suite. Sinon, *select* retournera au bout du temps que vous indiquez en secondes, ou plus tôt si un socket est prêt à être lu.

En gros, si vous précisez un *timeout* de 1, la fonction va bloquer pendant une seconde maximum. Mais si un des sockets en écoute est prêt à être lu dans l'intervalle (c'est-à-dire si un des clients envoie un message au serveur), la fonction retourne prématurément.

*select* renvoie trois listes, là encore *rlist*, *wlist* et *xlist*, sauf qu'il ne s'agit pas des listes fournies en entrée mais uniquement des sockets « à lire » dans le cas de *rlist*.

Ce n'est pas clair ? Considérez cette ligne (ne l'essayez pas encore) :

```
rlist, wlist, xlist = select.select(clients_connectes, [], [], 2)
```

# Un serveur plus élaboré

Cette instruction va écouter les sockets contenus dans la liste `clients_connectes`. Elle retournera au plus tard dans 2 secondes. Mais elle retournera plus tôt si un client envoie un message. La liste des clients ayant envoyé un message se retrouve dans notre variable `rlist`. On la parcourt ensuite et on peut appeler `recv` sur chacun des sockets.

Si ce n'est pas plus clair, assurez-vous : nous allons voir `select` en action un peu plus bas. Vous pouvez également aller jeter un coup d'œil à [la documentation du module select](#).

## **select** en action

Nous allons un peu travailler sur notre serveur. Vous pouvez garder le même client de test.

Le but va être de créer un serveur pouvant accepter plusieurs clients, réceptionner leurs messages et leur envoyer une confirmation à chaque réception. L'exercice ne change pas beaucoup mais on va utiliser `select` pour travailler avec plusieurs clients.

J'ai parlé de `select` pour écouter plusieurs clients connectés mais cette fonction va également nous permettre de savoir si un (ou plusieurs) clients sont connectés au serveur. Si vous vous souvenez, la méthode `accept` est aussi une fonction bloquante. On va du reste l'utiliser de la même façon qu'un peu plus haut.

Je crois vous avoir donné assez d'informations théoriques. Le code doit parler maintenant

# Un serveur plus élaboré

```
import socket
import select

hote = ""
port = 12800

connexion_principale = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connexion_principale.bind((hote, port))
connexion_principale.listen(5)
print("Le serveur écoute à présent sur le port {}".format(port))

serveur_lance = True
clients_connectes = []
while serveur_lance:
    # On va vérifier que de nouveaux clients ne demandent pas à se connecter
    # Pour cela, on écoute la connexion_principale en lecture
    # On attend maximum 50ms
    connexions_demandees, wlist, xlist = select.select([connexion_principale],
                                                       [], [], 0.05)

    for connexion in connexions_demandees:
        connexion_avec_client, infos_connexion = connexion.accept()
        # On ajoute le socket connecté à la liste des clients
        clients_connectes.append(connexion_avec_client)

    # Maintenant, on écoute la liste des clients connectés
    # Les clients renvoyés par select sont ceux devant être lus (recv)
    # On attend là encore 50ms maximum
    # On enferme l'appel à select.select dans un bloc try
    # En effet, si la liste de clients connectés est vide, une exception
    # Peut être levée
    clients_a_lire = []

connexion_principale.close()
```

# Un serveur plus élaboré

```
try:  
    clients_a_lire, wlist, xlist = select.select(clients_connectes,  
                                                [], [], 0.05)  
except select.error:  
    pass  
else:  
    # On parcourt la liste des clients à lire  
    for client in clients_a_lire:  
        # Client est de type socket  
        msg_recu = client.recv(1024)  
        # Peut planter si le message contient des caractères spéciaux  
        msg_recu = msg_recu.decode()  
        print("Reçu {}".format(msg_recu))  
        client.send(b"5 / 5")  
        if msg_recu == "fin":  
            serveur_lance = False  
  
    print("Fermeture des connexions")  
    for client in clients_connectes:  
        client.close()  
  
    connexion_principale.close()
```

# Un serveur plus élaboré

C'est plus long hein ? C'est inévitable, cependant.

Maintenant notre serveur peut accepter des connexions de plus d'un client, vous pouvez faire le test. En outre, il ne se bloque pas dans l'attente d'un message, du moins pas plus de 50 millisecondes.

Je pense que les commentaires sont assez précis pour vous permettre d'aller plus loin. Ceci n'est naturellement pas encore une version complète mais, grâce à cette base, vous devriez pouvoir facilement arriver à quelque chose. Pourquoi ne pas faire un mini tchat ?

Les déconnexions fortuites ne sont pas gérées non plus. Mais vous avez assez d'éléments pour faire des tests et améliorer notre serveur si cela vous tente.

## Et encore plus

Je vous l'ai dit, le réseau est un vaste sujet et, même en se restreignant au sujet que j'ai choisi, il y aurait beaucoup d'autres choses à vous montrer. Je ne peux tout simplement pas remplacer la documentation et donc, si vous voulez en apprendre plus, je vous invite à jeter un coup d'œil à la page du module [socket](#), de [select](#) et de [socketserver](#).

Le dernier module, socketserver, propose une alternative pour monter vos applications serveur. Il en existe d'autres, dans tous les cas : vous pouvez utiliser des sockets non bloquants (c'est-à-dire qui ne bloquent pas le programme quand vous utilisez leur méthode accept ou recv) ou des threads pour exécuter différentes portions de votre programme en parallèle. Mais je vous laisse vous documenter sur ces sujets s'ils vous intéressent !

# IP address

```
# Python 3 has a std lib
# module for working with
# IP addresses:

import ipaddress

ipaddress.ip_address('192.168.1.2')
>>> IPv4Address('192.168.1.2')

ipaddress.ip_address('2001:af3::')
>>> IPv6Address('2001:af3::')

# Learn more here:
# https://docs.python.org/3/library/ipaddress.html
```

# Résumé

## En résumé

Dans la structure réseau que nous avons vue, on trouve un **serveur** pouvant dialoguer avec plusieurs **clients**.

Pour créer une connexion côté serveur ou client, on utilise le module *socket* et la classe *socket* de ce module.

Pour se connecter à un serveur, le *socket* client utilise la méthode *connect*.

Pour écouter sur un port précis, le serveur utilise d'abord la méthode *bind* puis la méthode *listen*.

Pour s'échanger des informations, les *sockets* client et serveur utilisent les méthodes *send* et *recv*.

Pour fermer une connexion, le *socket* serveur ou client utilise la méthode *close*.

Le module *select* peut être utile si l'on souhaite créer un serveur pouvant gérer plusieurs connexions simultanément ; toutefois, il en existe d'autres.

# Créez des tests unitaires avec unittest

# Créez des tests unitaires avec unittest

## Créez des tests unitaires avec unittest

Tester ! Tout un monde. Vous allez voir dans ce chapitre comment tester le bon fonctionnement de votre programme et apprendre à le rendre aussi stable que possible au fur et à mesure que vous proposerez de nouvelles améliorations.

Si vous pensez que tester ne sert à rien ou que tester ne se fait que quand tout le développement est fini, je vous encourage vivement à lire ce chapitre, ne serait-ce que pour information.

Pour suivre ce chapitre, vous aurez besoin de savoir comment créer des classes et avoir une idée du fonctionnement de l'héritage.

# Pourquoi tester ?

On va parler de tests... mais qu'est-ce qu'on entend par « tester » ?

C'est la première question, et elle est très importante !

Dans ce chapitre, je vais parler de tests (principalement de tests unitaires), qui vérifient que votre code réagit comme il le devrait et qu'il continue à réagir comme il le devrait après de nouvelles améliorations.

Certains développeurs refusent de travailler sur du code qui n'est pas le leur s'il n'a pas de documentation. Pour ce que j'en ai vu, un nombre plus important encore de développeurs refuse de le faire si le code n'a pas de test.

Admettons que vous travaillez sur votre projet qui propose plusieurs fonctions, utilisées par d'autres développeurs ou utilisateurs. Vous pouvez être tout seul sur le projet et ne proposer qu'une dizaine de fonctions, c'est bien suffisant, le plus important c'est que votre code est utilisé par d'autres.

Puis après avoir codé votre dixième fonction, vous commencez à coder votre onzième qui utilise une autre fonction que vous avez déjà développée. Mais vous vous heurtez à un problème : votre nouvelle fonction ne marche pas comme il faut.

Après enquête, vous vous rendez compte que ce n'est pas votre fonction 11 qui pose problème, mais la fonction (1 ou 2) que la fonction 11 appelle. Elle ne répond plus à votre besoin et vous vous dites, naturellement, « je vais la modifier ».

Vous modifiez donc votre fonction 1 ou 2. Votre fonction 11 marche, enfin, sans problème. Vous proposez votre nouvelle version à vos utilisateurs.

Et vous recevez un chœur de protestations : jugez donc ! Ils utilisaient votre fonction 1 ou 2 sans problème, mais avec votre nouvelle version, rien ne marche plus.

Les tests sont une solution possible : pour chaque fonctionnalité de votre programme, il y aura un test et le test va s'assurer que votre programme reste valide même quand vous le modifierez. Ce qui deviendra de plus en plus important au fur et à mesure que votre programme gagnera en fonctionnalités, bien entendu.

Est-ce qu'on doit tester un code quand tout est développé ?

# Pourquoi tester ?

Non ! Si vous pouvez le faire dès le début, dès les premières lignes de code que vous écrivez, c'est mieux. Sachez qu'il peut être assez difficile d'écrire des tests quand votre programme comporte déjà plusieurs centaines de fonctionnalités, il vaut mieux le faire petit à petit.

Il existe aussi plusieurs méthodes de développement, dont le TDD (Test-Driven Development) qui veut que l'on écrive les tests avant d'écrire le code. Je ne rentrerai pas dans le détail ici, mais je vous conseille vivement d'écrire vos tests unitaires même si vous n'avez qu'un tout petit projet avec 4 ou 5 fonctions. Il y a une chance non négligeable que le petit projet devienne grand ; avec des tests à portée de main, vous dormirez plus tranquille.

## Est-ce difficile de tester un programme ?

Une fois que vous maîtrisez une des méthodes de test et que vous l'appliquez à votre programme au fur et à mesure, non ce n'est absolument pas difficile. Vous allez voir dans ce chapitre comment utiliser des tests unitaires. Il existe d'autres méthodes de test proposées par Python, mais c'est celle-ci que je trouve, personnellement, la plus rapide à prendre en main ainsi que la plus flexible. Ce chapitre est là pour vous guider pas à pas vers la création de vos premiers tests unitaires et même vers la gestion de nombreux tests quand votre projet sera plus grand.

## Qui écrit les tests ?

Le développeur, la plupart du temps. Là encore, la méthode de test utilisée peut permettre à d'autres personnes d'écrire les tests, mais les tests unitaires sont souvent écrits par des développeurs (ou des utilisateurs sachant programmer). Comme vous allez le voir, ils ne sont pas très difficiles à écrire, mais vous passerez malgré tout par Python pour ce faire.

Passons à la pratique, la découverte du module unittest !

# Premiers exemples de tests unitaires

## Structure de base d'un test unitaire

Nous le verrons plus loin, un test unitaire peut être constitué de nombreux tests répartis dans plusieurs packages et modules. Pour l'instant, nous n'allons nous intéresser qu'à un test case, la forme la plus simple du test unitaire.

Pour créer un test unitaire, la première chose est de créer une classe héritant de `unittest.TestCase`:

```
import random
import unittest

class RandomTest(unittest.TestCase):
```

On peut définir ensuite un test dans une méthode dont le nom commence par `test`.

Test de la fonction `random.choice`

Voyons pour le premier test, le test de la fonction `choice` :

```
import random
import unittest

class RandomTest(unittest.TestCase):
    """Test case utilisé pour tester les fonctions du module 'random'."""

    def test_choice(self):
        """Test le fonctionnement de la fonction 'random.choice'."""
        liste = list(range(10))
        elt = random.choice(liste)
        # Vérifie que 'elt' est dans 'liste'
        self.assertIn(elt, liste)
```

# Premiers exemples de tests unitaires

- Quelques explications s'imposent pour notre méthode de test :
- D'abord à la première ligne, on crée une liste de 0 à 9 ;
- Ensuite on appelle la fonction random.choice sur notre liste et on récupère le retour ;
- Enfin, on vérifie que notre élément retourné par random.choice se trouve bien dans notre liste. On utilise pour ce faire une méthode assertIn et pas le mot clé assert. En fait, unittest.TestCase propose plusieurs méthodes d'assertion que nous utiliserons dans nos tests unitaires. Une assertion lève une exception qui serait considérée par unittest comme une erreur. Nous verrons plus loin comment les erreurs sont gérées.
- Si vous exécutez ce code dans votre interpréteur... rien ne se passe ! Vous avez créé une classe mais vous n'avez pas demandé au test de se lancer. Pour ce faire vous pouvez exécuter l'instruction :

# Premiers exemples de tests unitaires

Le module `unittest` de la bibliothèque standard de Python inclut le mécanisme des tests unitaires.

Voici la structure que vous rencontrerez le plus souvent :

- Une fonctionnalité codée grâce à un ensemble de fonctions, de classes, de modules, de packages et autre.
- Pour chaque fonctionnalité, un test qui vérifie que la fonctionnalité fait bien ce qu'on lui demande. Par exemple, que si une certaine fonction est appelée avec certains paramètres, elle retourne telle valeur.

Nous allons nous intéresser ici à ce second point dans la liste : comment tester une fonctionnalité.

## Tester une fonctionnalité existante

Pour commencer, nous allons tester une fonctionnalité déjà existante, proposée dans l'un des modules de Python. Je vais reprendre les exemples de la documentation officielle qui sont assez faciles à comprendre.

Pour cet exemple, nous allons nous intéresser au module `random` que nous avons déjà utilisé. Nous allons chercher à tester le fonctionnement en particulier de trois fonctions :

- `random.choice`: cette fonction retourne un élément au hasard de la séquence précisée en paramètre.

```
import random

liste = ["chat", "chien", "renard", "serpent", "cheval", "parapluie"]
random.choice(liste)
>>> 'renard'
```

# Premiers exemples de tests unitaires

- *random.shuffle*: cette fonction mélange une liste. La liste d'origine est modifiée.

```
import random
liste = [1, 2, 3, 4, 5, 6, 7, 8, 9]
random.shuffle(liste)
print(liste)
>>> [3, 4, 7, 1, 8, 6, 5, 9, 2]
```

- *random.sample*: cette fonction prend une séquence et un nombre en paramètres. Elle retourne une nouvelle séquence contenant autant d'éléments que le nombre indiqué, sélectionnés aléatoirement dans la séquence d'origine. Ce n'est pas clair ?

```
import random

liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
random.sample(liste, 5)
>>> ['b', 'a', 'c', 'j', 'e']
# Ou peut-être que cet exemple sera plus clair
random.sample(range(1000), 10)
>>> [389, 406, 890, 955, 837, 401, 971, 716, 954, 862]
```

# Premiers exemples de tests unitaires

```
unittest.main()
```

Et vous devriez obtenir quelque chose comme :

```
.
```

---

```
Ran 1 test in 0.003s
```

```
OK
```

L'appel à `unittest.main` ferme la console Python, soyez prévenu, ce n'est pas une erreur mais bien un comportement attendu.

Le retour affiché se décompose en trois parties :

- D'abord, la première ligne contient un caractère par test exécuté. Les principaux caractères sont un point (".") si le test s'est validé, la lettre F si le test n'a pas obtenu le bon résultat et la lettre E si le test a rencontré une erreur (si une exception a été levée pendant l'exécution de la méthode) ;
- Ensuite se trouve une ligne récapitulative du nombre de tests exécutés ;
- Enfin, la dernière ligne récapitule le nombre de réussites ou échecs ou erreurs. Si tout va bien, cette dernière ligne devrait être simplement "OK".

## Faisons échouer un test

Modifions notre test pour être sûr de provoquer un échec :

```
class RandomTest(unittest.TestCase):  
  
    """Test case utilisé pour tester les fonctions du module 'random'. """  
  
    def test_choice(self):  
        """Test le fonctionnement de la fonction 'random.choice'. """  
        liste = list(range(10))  
        elt = random.choice(liste)  
        self.assertIn(elt, ('a', 'b', 'c'))
```

# Premiers exemples de tests unitaires

*Et après un appel à `unittest.main()`:*

```
.  
-----  
Ran 1 test in 0.003s  
  
OKF  
=====  
FAIL: test_choice (__main__.RandomTest)  
Test le fonctionnement de la fonction 'random.choice'.  
-----  
Traceback (most recent call last):  
File "code.py", line 13, in test_choice  
    self.assertIn(elt, ('a', 'b', 'c'))  
AssertionError: 0 not found in ('a', 'b', 'c')  
  
-----  
Ran 1 test in 0.004s  
  
FAILED (failures=1)
```

Vous voyez que l'on obtient pas mal d'informations sur les tests qui ne marchent pas. D'abord, notez qu'ici, on parle d'échec (failure) et non pas d'erreur (error). Cela signifie que notre assertion ne s'est pas vérifiée (regardez le traceback) mais que notre test s'est correctement exécuté. Vous pouvez essayer de provoquer une erreur dans la méthode de test aussi, pour voir le résultat.

Le traceback est assez détaillé : il donne la ligne de l'erreur avec les appels successifs, si on a besoin de remonter la piste de l'erreur. Le message d'erreur lui-même donne des informations plus précises sur pourquoi le test a échoué (0 not found in ('a', 'b', 'c')).

# Premiers exemples de tests unitaires

## Test de la fonction `random.shuffle`

Intéressons-nous maintenant à la fonction `random.shuffle`. Souvenez-vous, elle prend une liste en paramètre et mélange cette liste aléatoirement.

En vous inspirant du premier exemple, essayez d'écrire la méthode de test correspondante. Il vous faut réfléchir à comment vérifier qu'une liste, après avoir été mélangée, correspond à une liste d'éléments de 0 à 9.

Je vous conseille d'utiliser cette fois la méthode d'assertion `assertEqual` qui prend deux arguments en paramètre et vérifie le test si les arguments sont identiques. Vous trouverez une liste des méthodes d'assertion les plus communes plus bas.

```
class RandomTest(unittest.TestCase):  
  
    """Test case utilisé pour tester les fonctions du module 'random'. """  
  
    # Autres méthodes de test  
    def test_shuffle(self):  
        """Test le fonctionnement de la fonction 'random.shuffle'. """  
        liste = list(range(10))  
        random.shuffle(liste)  
        liste.sort()  
        self.assertEqual(liste, list(range(10)))
```

Comme vous le voyez, on appelle la fonction `random.shuffle` avant de trier de nouveau notre liste. Une fois la liste triée de nouveau, elle devra être identique à notre liste d'origine (`list(range(10))`).

Ici, nous avons utilisé la méthode `assertEqual` qui sera sans doute celle que vous utiliserez le plus souvent. Nous verrons un peu plus loin une liste des méthodes d'assertion proposées par `unittest.TestCase`.

# Premiers exemples de tests unitaires

## Test de la fonction random.sample

Enfin, écrivons notre méthode de test de la fonction random.sample. Souvenez-vous, cette fonction prend deux paramètres : une séquence et un nombre K. Elle retourne une liste contenant K éléments sélectionnés aléatoirement dans notre séquence de base.

```
class RandomTest(unittest.TestCase):  
  
    """Test case utilisé pour tester les fonctions du module 'random'. """  
  
    # Autres méthodes de test  
    def test_sample(self):  
        """Test le fonctionnement de la fonction 'random.sample'."""  
        liste = list(range(10))  
        extrait = random.sample(liste, 5)  
        for element in extrait:  
            self.assertIn(element, liste)
```

Jusqu'ici ce n'est pas bien différent de ce que nous avons fait un peu plus haut.

Avez-vous essayé random.sample en précisant un nombre K plus élevé que la taille de la séquence ?

```
liste = list(range(10))  
random.sample(liste, 20)  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
File "C:\python34\lib\random.py", line 313, in sample  
    raise ValueError("Sample larger than population")  
ValueError: Sample larger than population
```

Citation : PEP 20

Errors should never pass silently.

# Premiers exemples de tests unitaires

Ce comportement est attendu et souhaitable. Autant le tester également :

```
class RandomTest(unittest.TestCase):  
  
    """Test case utilisé pour tester les fonctions du module 'random'. """  
  
    # Autres méthodes de test  
    def test_sample(self):  
        """Test le fonctionnement de la fonction 'random.sample'."""  
        liste = list(range(10))  
        extrait = random.sample(liste, 5)  
        for element in extrait:  
            self.assertIn(element, liste)  
  
        self.assertRaises(ValueError, random.sample, liste, 20)
```

La dernière ligne mérite quelques explications. On utilise encore une méthode d'assertion *assert\** (cette fois, *assertRaises*).

On peut utiliser cette méthode de deux façons :

- Soit, comme on vient de le faire, en précisant d'abord le type de l'exception qui doit être levée, puis la fonction qui doit être appelée (la référence, sans parenthèses) et enfin les paramètres attendus par la fonction ;
- Soit en utilisant un context manager (gestionnaire de contexte) qui rend le code plus facile à lire.

Nous avons vu un **context manager** au moment des fichiers. Rappelez-vous, c'est le bloc d'instructions qui commence par le mot clé `with`.

Voyons comment écrire notre test avec un **context manager**.

# Premiers exemples de tests unitaires

```
class RandomTest(unittest.TestCase):  
  
    """Test case utilisé pour tester les fonctions du module 'random'. """  
  
    # Autres méthodes de test  
    def test_sample(self):  
        """Test le fonctionnement de la fonction 'random.sample'. """  
        liste = list(range(10))  
        extrait = random.sample(liste, 5)  
        for element in extrait:  
            self.assertIn(element, liste)  
  
        with self.assertRaises(ValueError):  
            random.sample(liste, 20)
```

# Premiers exemples de tests unitaires

Comme vous le voyez, cette seconde syntaxe est plus lisible :

1. On appelle un nouveau context manager grâce au mot-clé `with` ouvert sur le retour de la méthode `assertRaises`. Cette fois, on ne passe en paramètre de cette méthode que le type de notre exception ;
2. À l'intérieur de notre bloc se trouve la ligne qui doit lever l'exception `ValueError`. Si le bloc dans le context manager lève bien l'exception, alors le test passe. Sinon il ne passe pas.

Cette seconde syntaxe est plus lisible, à mon sens, mais je vous montre les deux car vous pourriez trouver la première au cours de vos lectures d'autres codes.

## Initialisation des tests

Vous l'avez peut-être remarqué, toutes nos méthodes de test commencent par cette ligne de code :

```
liste = list(range(10))
```

Il existe un moyen pour éviter de répéter cette ligne à chaque fois. Nos méthodes de test partagent un point commun : elles sont définies dans la même classe. Autant en profiter. `unittest.TestCase` nous propose une méthode qui est appelée avant chaque méthode de test. Il serait mieux que la création de notre liste (de 0 à 9) se trouve dans cette méthode. Son nom est `setUp`. Créez-la dans votre classe :

```
class RandomTest(unittest.TestCase):  
  
    """Test case utilisé pour tester les fonctions du module 'random'. """  
  
    def setUp(self):  
        """Initialisation des tests."""  
        self.liste = list(range(10))
```

Comme vous le voyez, on écrit directement notre liste en attribut d'instance de notre test. Cela veut dire qu'il va falloir modifier nos méthodes de test pour qu'elles l'utilisent :

# Premiers exemples de tests unitaires

```
import random
import unittest

class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module 'random'."""

    # Autres méthodes de test
    def test_sample(self):
        """Test le fonctionnement de la fonction 'random.sample'."""
        extrait = random.sample(self.liste, 5)
        for element in extrait:
            self.assertIn(element, self.liste)

        with self.assertRaises(ValueError):
            random.sample(self.liste, 20)
```

Au lieu de créer la liste, on utilise l'attribut d'instance créé dans la méthode setUp. Il existe également une méthode tearDown qui est appelée après chaque test.  
Récapitulatif complet du code de test

Voici le code complet de notre test case et de nos trois méthodes de test.

# Premiers exemples de tests unitaires

```
import random
import unittest

class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module 'random'."""

    def setUp(self):
        """Initialisation des tests."""
        self.liste = list(range(10))

    def test_choice(self):
        """Test le fonctionnement de la fonction 'random.choice'."""
        elt = random.choice(self.liste)
        self.assertIn(elt, self.liste)

    def test_shuffle(self):
        """Test le fonctionnement de la fonction 'random.shuffle'."""
        random.shuffle(self.liste)
        self.liste.sort()
        self.assertEqual(self.liste, list(range(10)))

    def test_sample(self):
        """Test le fonctionnement de la fonction 'random.sample'."""
        extrait = random.sample(self.liste, 5)
        for element in extrait:
            self.assertIn(element, self.liste)

    with self.assertRaises(ValueError):
        random.sample(self.liste, 20)
```

Souvenez-vous, pour tester le code, vous pouvez ajouter l'instruction `unittest.main()` à la fin de votre module. Nous verrons un peu plus loin un autre moyen, plus simple, pour tester un ou plusieurs modules.

# Premiers exemples de tests unitaires

## Les principales méthodes d'assertion

Je vous propose un petit tableau listant les méthodes d'assertion les plus courantes.

Méthode	Explications
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>x is True</code>
<code>assertFalse(x)</code>	<code>x is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assert IsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>
<code>assertRaises(exception, fonction, *args, **kwargs)</code>	Vérifie que la fonction lève l'exception attendue.

Pour une liste complète, consultez la [documentation officielle du module unittest](#).

Nous allons nous intéresser à présent à la découverte automatique des tests par Python.

# La découverte automatique des tests

Lancer les tests avec `unittest.main()` peut s'avérer pratique, mais généralement on fera appel à la découverte automatique des tests. Cette fonctionnalité permet de rechercher tous les tests unitaires contenus dans un package et de les exécuter.

Lancement de tests unitaires depuis un répertoire

Pour commencer, nous allons essayer de lancer les tests unitaires que nous avons créés auparavant depuis un répertoire.

Créez un répertoire où vous mettez généralement votre code Python. Pour moi, ce répertoire s'appelle pytest et se trouve dans Mes Documents ;

Ouvrez la console. Sous Windows, cliquez sur Exécuter... dans le menu démarrer (ou tapez Windows + R) et entrez cmd ;

Déplacez-vous dans le répertoire que vous avez créé :

```
cd pytest
```

# La découverte automatique des tests

Une fois dans le bon dossier, créez le fichier test\_random.py et collez le code que nous avons vu plus haut :

```
import random
import unittest

class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module 'random'."""

    def setUp(self):
        """Initialisation des tests."""
        self.liste = list(range(10))

    def test_choice(self):
        """Test le fonctionnement de la fonction 'random.choice'."""
        elt = random.choice(self.liste)
        self.assertIn(elt, self.liste)

    def test_shuffle(self):
        """Test le fonctionnement de la fonction 'random.shuffle'."""
        random.shuffle(self.liste)
        self.liste.sort()
        self.assertEqual(self.liste, list(range(10)))

    def test_sample(self):
        """Test le fonctionnement de la fonction 'random.sample'."""
        extrait = random.sample(self.liste, 5)
        for element in extrait:
            self.assertIn(element, self.liste)

        with self.assertRaises(ValueError):
            random.sample(self.liste, 20)
```

Sauvegardez ce fichier et revenez dans la console.

Vous devez maintenant exécuter Python avec l'option -m unittest. Sous Windows vous aurez sûrement une commande comme :

```
c:\python34\python.exe -m unittest
```

# La découverte automatique des tests

Sous Linux vous aurez probablement :

```
python3.4 -m unittest
```

Si tout se passe bien vous devriez voir les tests s'exécuter :

```
...
```

```
Ran 3 tests in 0.007s
```

```
OK
```

L'option `-m` permet d'exécuter un module spécifique (ici `unittest`). Quand appelé directement depuis Python, `unittest` cherche les tests unitaires présents dans le dossier courant. Vous pouvez aussi lui donner un chemin de test à exécuter, par exemple `test_random.RandomTest.test_shuffle` :

1. `test_random` est le nom du module (le nom du fichier sans l'extension) ;
2. `RandomTest` est le nom de la classe dans notre module ;
3. `test_shuffle` est le nom de notre méthode à exécuter.

```
c:\python34\python.exe -m unittest test_random.RandomTest.test_shuffle
```

```
.
```

```
Ran 1 test in 0.002s
```

```
OK
```

Vos tests unitaires doivent être indépendants, c'est-à-dire qu'on peut les exécuter tout seul (comme on vient de le faire) ou en groupe (comme on l'a fait plus tôt). En bref, ils ne doivent pas dépendre d'autres tests pour s'exécuter.

## Structure d'un projet avec ses tests

Nous allons ici regarder un projet de taille respectable, CherryPy, qui propose un framework léger pour créer un serveur web. Je vous conseille d'ailleurs de jeter un oeil à ce projet si vous avez le temps.

Si vous téléchargez et décompressez les sources, vous verrez un dossier `cherrypy-version`.

# La découverte automatique des tests

Si vous entrez dedans, vous pouvez lancer les tests unitaires en faisant :

```
python -m unittest
```

Il peut être nécessaire d'installer le package au préalable (exécutez la commande `python setup.py install` pour ce faire).

Si Python trouve les tests unitaires du projet, c'est qu'il explore les répertoires du projet. Il y a notamment le répertoire cherrypy qui contient l'ensemble des sources. Dans ce répertoire se trouve le sous-répertoire test et dans ce sous-répertoire se trouvent les tests de la bibliothèque.

Je ne rentrerai pas dans le détail ici, mais ce qu'il faut comprendre, c'est que la commande `python -m unittest` explore récursivement les packages et modules à la recherche de tests. Tous les packages sont explorés, mais les modules (comme les méthodes de test) doivent commencer par test.

Généralement, vous trouverez une certaine fonctionnalité (disons dans `cherrypy/fonctionnalite.py`) et le test de cette fonctionnalité dans un module spécifique (`cherrypy/test/test_fonctionnalite.py`). Le découpage du dossier test sera souvent le même que le découpage de vos sources (c'est plus une convention qu'une obligation).

Voilà pour ce tour d'horizon des tests unitaires. Là encore, si vous voulez en apprendre plus, rendez-vous sur [la documentation officielle du module unittest](#).

# Résumé

## En résumé

- on peut tester nos applications grâce à plusieurs modules sous Python, les tests unitaires étant supportés par le module unittest ;
- pour créer un test unitaire, il faut créer une classe qui hérite de `unittest.TestCase`. Les méthodes de test ont un nom commençant par `test` ;
- La commande `python -m unittest` permet la découverte automatique des tests dans le répertoire courant.

Faites de la  
programmation parallèle  
avec threading

# Faites de la programmation parallèle avec threading

Faites de la programmation parallèle avec threading

Jusqu'ici, nous avons utilisé Python de façon linéaire : les instructions s'exécutaient dans l'ordre et, pour que la suivante s'exécute, celle d'avant devait être terminée.

Mais Python nous propose dans sa bibliothèque standard plusieurs modules pour faire de la « programmation parallèle », c'est-à-dire que plusieurs instructions de code s'exécuteront en même temps, ou presque en même temps.

Nous allons regarder de plus près le module `threading` qui propose une interface simple pour créer des threads, c'est-à-dire des portions de notre code qui seront exécutées en même temps.

Pour suivre ce chapitre, vous aurez besoin de savoir comment créer des classes et connaître les bases de l'héritage.

# Création de threads

Jusqu'ici, nous avons travaillé avec de la programmation « linéaire ». Considérez ce code :

```
import time
print("Avant le sleep...")
time.sleep(5)
print("Après le sleep.")
```

Si vous exécutez ce code, sans surprise, le premier message Avant le sleep... s'affiche, puis le programme pause pendant 5 secondes. Enfin, le second message Après le sleep. s'affiche.

Les threads permettent d'exécuter plusieurs instructions en même temps. On parle de « programmation parallèle », car au lieu de développer selon un seul flux d'instruction, on développe plusieurs flux en parallèle.

## Premier exemple d'un thread

Voyons un code linéaire pour commencer. Je fais appel à plusieurs fonctions que vous n'avez peut-être jamais vues, mais pas de panique, je commente les lignes en question plus bas :

```
import random
import sys
import time

# Répète 20 fois
i = 0
while i < 20:
    sys.stdout.write("1")
    sys.stdout.flush()
    attente = 0.2
    attente += random.randint(1, 60) / 100
    # attente est à présent entre 0.2 et 0.8
    time.sleep(attente)
    i += 1
```

# Création de threads

D'abord, on importe les modules random, sys et time que nous allons utiliser par la suite ;

ensuite on crée une boucle qui va s'exécuter 20 fois ;

on affiche simplement le chiffre 1. On fait appel à sys.stdout.write() pour afficher le chiffre sur la sortie standard (l'écran, par défaut) et sys.stdout.flush() pour demander à Python d'afficher le chiffre tout de suite. Si vous oubliez cette seconde ligne, les chiffres n'apparaîtront qu'à la fin de l'exécution du programme ;

on crée une variable attente et on la fait varier, grâce à random, entre 0.2 et 0.8 ;

enfin, on appelle time.sleep() qui met en pause notre programme pendant le temps d'attente que nous avons configuré plus haut (c'est-à-dire entre 0,2 et 0,8 seconde).

Si vous exécutez ce code, vous devriez voir apparaître 20 fois le chiffre 1 sur la même ligne, mais entre chaque chiffre le programme se met en pause (la pause est de durée variable).

## Approche parallèle

Maintenant, nous allons créer deux threads qui vont s'exécuter ensemble : le premier affichera des 1 sur l'écran, tandis que le second affichera des 2. Lancé en même temps, vous devriez voir plus clairement la façon dont ils s'exécutent.

Pour créer un thread, il faut créer une classe qui hérite de threading.Thread. On peut redéfinir son constructeur et la méthode run.

Cette seconde méthode est appelée au lancement du thread et contient le code qui doit s'exécuter en parallèle du reste du programme.

# Création de threads

Voyons un exemple :

```
import random
import sys
from threading import Thread
import time

class Afficheur(Thread):

    """Thread chargé simplement d'afficher une lettre dans la console."""

    def __init__(self, lettre):
        Thread.__init__(self)
        self.lettre = lettre

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 20:
            sys.stdout.write(self.lettre)
            sys.stdout.flush()
            attente = 0.2
            attente += random.randint(1, 60) / 100
            time.sleep(attente)
            i += 1
```

Au-dessus se trouve la définition d'un **thread** :

- Le constructeur ne devrait pas trop vous surprendre. Il prend en paramètre la lettre à afficher (nous verrons des exemples plus loin). Il appelle le constructeur parent (`Thread.__init__(self)`) et c'est une étape importante, ne l'oubliez pas quand vous redéfinissez le constructeur de votre **thread** ;
- la méthode `run` est également redéfinie. Le code qu'elle contient vous semble sans doute familier : c'est le code que nous avons utilisé dans notre exemple de programmation linéaire tout à l'heure.

# Création de threads

Une fois encore, si vous exécutez ce code, vous obtenez... rien du tout ! Vous avez défini le thread, mais il nous reste à le créer. Ou plutôt, à les créer, car nous allons essayer de faire deux threads s'exécutant en même temps :

```
# Création des threads  
thread_1 = Afficheur("1")  
thread_2 = Afficheur("2")
```

```
# Lancement des threads  
thread_1.start()  
thread_2.start()
```

```
# Attend que les threads se terminent  
thread_1.join()  
thread_2.join()
```

D'abord, on crée nos deux threads. Les objets Thread sont conservés dans nos variables `thread_1` et `thread_2`. Notez qu'on passe en paramètre de nos deux threads des lettres différentes, pour pouvoir les différencier quand ils commenceront à afficher les informations dans la console ;

ensuite, on appelle `thread_1.start()`. Cette méthode va créer un thread (une partie du code qui va pouvoir s'exécuter en parallèle) et exécuter la méthode `run`. Nos chiffres 1 commencent ainsi à s'afficher dans notre console. Mais la méthode `start` n'attend pas que tous les chiffres soient écrits avant de retourner et on passe tout de suite à la ligne suivante ;

C'est au tour du second thread. Il est également lancé. Les deux threads s'exécutent en même temps ;

Enfin, on appelle la méthode `join()` sur les deux threads. Cette méthode bloque et ne retourne que quand le thread est terminé. Si le programme se termine pendant que des threads tournent, les threads risquent d'être fermés brusquement.

# Création de threads

Pour récapituler, voici le code complet :

```
import random
import sys
from threading import Thread
import time

class Afficheur(Thread):

    """Thread chargé simplement d'afficher une lettre dans la console."""

    def __init__(self, lettre):
        Thread.__init__(self)
        self.lettre = lettre

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 20:
            sys.stdout.write(self.lettre)
            sys.stdout.flush()
            attente = 0.2
            attente += random.randint(1, 60) / 100
            time.sleep(attente)
            i += 1

    # Création des threads
thread_1 = Afficheur("1")
thread_2 = Afficheur("2")

    # Lancement des threads
thread_1.start()
thread_2.start()

    # Attend que les threads se terminent
thread_1.join()
thread_2.join()
```

# Création de threads

Quand vous exécutez ce programme, vous obtenez une ligne similaire :

```
1221121212122121211221122212121221121211
```

Comme vous le voyez, les deux threads s'exécutent en même temps. Puisque le temps de pause est variable, parfois on a un seul chiffre 1 qui s'affiche avant un chiffre 2, parfois on en a plusieurs. Au final, il y en a bien 20 de chaque.

Pour cette fois d'ailleurs, remarquez que le thread\_1 est le plus long à s'exécuter (le dernier chiffre de la ligne est un 1, le dernier 2 est un peu avant). Vous pouvez essayer la même chose en créant plusieurs autres threads, 3 ou 4 ou 5 ou plus, si vous voulez.

La programmation parallèle peut être très pratique, mais elle a aussi ses pièges. Nous allons en voir certains à présent et les méthodes qui existent pour les éviter.

# La synchronisation des threads

Programmer plusieurs flux d'instructions apporte son lot de difficultés. Au premier abord, cela semble très pratique d'avoir plusieurs parties de notre code qui s'exécutent en même temps. Pendant une tâche qui peut prendre longtemps à s'exécuter (peut-être le téléchargement d'une information depuis un site Internet) on peut faire autre chose, pas seulement attendre que la ressource soit téléchargée.

Mais le développement peut être plus compliqué en proportion. Il vous faut garder en tête que les différents flux d'instructions peuvent être avancés à différents points à un moment précis.

## Opérations concurrentes

Considérez ce tout petit exemple :

```
nombre = 1  
nombre += 1
```

C'est la deuxième ligne qui nous intéresse ici :`nombre += 1`. Si vous y faites appel dans un de vos **threads** et que `nombre` est partagé par plusieurs de vos **threads**, vous pourriez avoir des résultats étranges. Pas tout le temps. C'est tout le problème : la plupart du temps vous n'aurez aucun soucis, parfois vous aurez des résultats étranges.

Disons que ce `nombre` serve à compter une information (le nombre de fois où une certaine opération s'exécute, peut-être). Si vous n'avez pas de chance, deux **threads** accéderont à ce code mais `nombre` ne sera augmenté que de 1.

Cela est du au fait que `nombre += 1` fait trois choses :

1. Elle va récupérer la valeur de la variable `nombre` ;
2. Elle va y ajouter 1 ;
3. Elle va écrire le résultat dans la variable `nombre`.

# La synchronisation des threads

Représentez-vous ces étapes sur une feuille. Maintenant, représentez-vous les mêmes étapes pour un second thread.

Admettons que le `thread_1` et le `thread_2` s'exécutent presque en même temps :

Le `thread_1` commence à exécuter l'instruction. Il exécute l'étape 1 et 2 (c'est-à-dire qu'il va récupérer la valeur de la variable `nombre`) mais n'exécute pas encore l'étape 3 (c'est-à-dire que la variable `nombre` n'est pas encore modifiée) ;

et voici `thread_2` qui exécute l'instruction (les trois étapes cette fois). Il récupère `nombre`, y ajoute 1 et écrit le résultat dans la variable ;

et notre `thread_1` exécute l'étape 3 et écrit le résultat dans la variable. Mais cette valeur se base sur l'ancienne valeur de `nombre` (avant que `thread_2` ne soit appelé). Au final, après l'exécution de nos deux **threads**, `nombre` n'a été incrémenté que de 1.

Comme vous le voyez ici, une ligne d'instruction très simple pourra avoir des résultats inattendus si elle est appelée au même moment par différents threads.

# La synchronisation des threads

Cela est du au fait que `nombre += 1` fait trois choses :

1. Elle va récupérer la valeur de la variable `nombre` ;
2. Elle va y ajouter 1 ;
3. Elle va écrire le résultat dans la variable `nombre`.

Représentez-vous ces étapes sur une feuille. Maintenant, représentez-vous les mêmes étapes pour un second **thread**.

Admettons que le `thread_1` et le `thread_2` s'exécutent presque en même temps :

- Le `thread_1` commence à exécuter l'instruction. Il exécute l'étape 1 et 2 (c'est-à-dire qu'il va récupérer la valeur de la variable `nombre`) mais n'exécute pas encore l'étape 3 (c'est-à-dire que la variable `nombre` n'est pas encore modifiée) ;
- et voici `thread_2` qui exécute l'instruction (les trois étapes cette fois). Il récupère `nombre`, y ajoute 1 et écrit le résultat dans la variable ;
- et notre `thread_1` exécute l'étape 3 et écrit le résultat dans la variable. Mais cette valeur se base sur l'ancienne valeur de `nombre` (avant que `thread_2` ne soit appelé). Au final, après l'exécution de nos deux **threads**, `nombre` n'a été incrémenté que de 1.

Comme vous le voyez ici, une ligne d'instruction très simple pourra avoir des résultats inattendus si elle est appelée au même moment par différents **threads**.

## Accès simultané à des ressources

Le problème est encore plus flagrant quand vous voulez accéder à des ressources depuis différents threads. Par exemple, vous voulez écrire dans un fichier (le même fichier depuis différents threads). Comme vous le voyez ici, une ligne d'instruction très simple pourra avoir des résultats inattendus si elle est appelée au même moment par différents threads).

# La synchronisation des threads

Voici le code de nos **threads** un peu modifié pour qu'il affiche des mots complets dans la console au lieu de simples lettres. Regardez surtout la méthode *run* :

```
import random
import sys
from threading import Thread
import time

class Afficheur(Thread):
    """Thread chargé simplement d'afficher un mot dans la console."""

    def __init__(self, mot):
        Thread.__init__(self)
        self.mot = mot

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 5:
            for lettre in self.mot:
                sys.stdout.write(lettre)
                sys.stdout.flush()
                attente = 0.2
                attente += random.randint(1, 60) / 100
                time.sleep(attente)
            i += 1

    # Création des threads
thread_1 = Afficheur("canard")
thread_2 = Afficheur("TORTUE")
```

*# Lancement des threads*  
thread\_1.start()  
thread\_2.start()

*# Attend que les threads se terminent*  
thread\_1.join()  
thread\_2.join()

# La synchronisation des threads

- On veut afficher des mots au lieu de lettres, le constructeur est donc modifié en conséquence ;
- On ne boucle que pendant 5 fois (au lieu de 20), ce sera suffisant pour que vous compreniez l'exemple ;
- À l'intérieur de notre boucle, on boucle sur chaque lettre, l'affiche et fait une pause.

Et quand vous exécutez ce code, vous devriez voir quelque chose comme :

```
cTORanaTUrEdcTaOnRarTdUcEanTaOrRdTcUaEnTaORrdTcanUaErdTORTUE
```

J'ai mis le mot "canard" en minuscule et le mot "TORTUE" en majuscule, ce qui devrait vous aider à les identifier. Comme vous le voyez, nos mots sont complètement mélangés, ce qui n'est pas bien surprenant. Vous pouvez toujours suivre la partie en majuscule ou minuscule et vérifier que les mots s'affichent bien, mais puisque nous écrivons sur la même ressource partagée (la console, ici), le résultat s'affiche mélangé.

## Les locks à la rescousse

Il existe plusieurs moyens de « synchroniser » nos threads, c'est-à-dire de faire en sorte qu'une partie du code ne s'exécute que si personne n'utilise la ressource partagée. Le mécanisme de synchronisation le plus simple est le lock (verrou en anglais).

C'est un objet proposé par threading qui est extrêmement simple à utiliser : au début de nos instructions qui utilisent notre ressource partagée, on dit au lock de bloquer pour les autres threads. Si un autre thread veut faire appel à cette ressource, il doit patienter jusqu'à ce qu'elle soit libérée.

Plutôt qu'un long discours, je vous propose notre code légèrement modifié pour utiliser les locks.

# La synchronisation des threads

```
import random
import sys
from threading import Thread, RLock
import time

verrou = RLock()

class Afficheur(Thread):
    """Thread chargé simplement d'afficher un mot dans la console."""

    def __init__(self, mot):
        Thread.__init__(self)
        self.mot = mot

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 5:
            with verrou:
                for lettre in self.mot:
                    sys.stdout.write(lettre)
                    sys.stdout.flush()
                    attente = 0.2
                    attente += random.randint(1, 60) / 100
                    time.sleep(attente)
            i += 1

    # Création des threads
thread_1 = Afficheur("canard")
thread_2 = Afficheur("TORTUE")

    # Lancement des threads
thread_1.start()
thread_2.start()

    # Attend que les threads se terminent
thread_1.join()
thread_2.join()
```

# La synchronisation des threads

1. On importe RLock du module threading ;
2. on crée un lock que l'on place dans notre variable verrou ;
3. dans notre méthode run, on verrouille une partie de notre **thread**.

with verrou:

```
for lettre in self.mot:  
    sys.stdout.write(lettre)  
    sys.stdout.flush()  
    attente = 0.2  
    attente += random.randint(1, 60) / 100  
    time.sleep(attente)
```

On utilise là encore un **context manager** pour indiquer quand bloquer le lock. Le **lock** se débloque à la fin du bloc with.

La partie verrouillée de notre code ne s'exécute qu'un **thread** à la fois.

1. D'abord, thread\_1 est lancé. Il verrouille le **lock** et commence à afficher les lettres de son mot ("canard") ;
2. thread\_2 est lancé entre temps, mais il bloque au moment d'afficher son propre mot, car le verrou est détenu par thread\_1. Ce n'est que quand thread\_1 relâche le verrou (à la fin du bloc with) qu'il peut commencer à s'exécuter ;
3. ... Et ainsi de suite jusqu'à la fin des deux **threads**.

Si vous exécutez ce code, vous pourrez voir quelque chose comme :

```
canardcanardTORTUETORTUEcanardcanardcanardTORTUETORTUETORTUE
```

# La synchronisation des threads

Comme vous le voyez, cette fois les mots ne sont plus mélangés, mais le reste du code s'exécute bien en parallèle (notez que les mots apparaissent dans un ordre aléatoire, même si il y en a bien 5 de chaque).

Il existe d'autres méthodes de synchronisation et la programmation parallèle en tant que telle mérite plus un livre entier qu'un chapitre d'introduction. Vous avez pu cependant voir ici les bases de ce type de programmation. Si vous voulez plus d'informations sur les mécanismes de synchronisation (ainsi que d'autres informations générales sur les threads), [vous pouvez lire la documentation officielle du module threading](#).

# Résumé

## En résumé

1. Il existe plusieurs mécanismes de programmation parallèle, dont les threads proposés dans le module threading de la bibliothèque standard ;
2. Créer un thread se fait en redéfinissant une classe héritée de `threading.Thread` et en appelant sa méthode `start` ;
3. On peut utiliser les locks pour synchroniser nos threads et faire en sorte que certaines parties de notre code s'exécutent bien à la suite des autres.

# Les bases de données

# Les bases de données

- Les fichiers jasons servent à stocker des données simples (ex: paramètres d'un logiciel: préférence utilisateur de VSCode). Ils sont très facile à mettre en œuvre dans python. Ils sont modifiable depuis un éditeur de texte.
- Les fichiers .db pour les databases sont un peu plus complexe à mettre en œuvre, mais ils permettent de gérer des données complexes. Ils ne sont pas éditables depuis un éditeur de texte.

# Les bases de données fichier JSON

```
1 import json
2
3 fichier = "settings.json"
4
5 with open(fichier, "r") as f:
6     settings = json.load(f)
7
8 settings["fontSize"] = 15
9
10 with open(fichier, "w") as f:
11     json.dump(settings, f, indent=4)
12
```

# Les bases de données SQLITE créer un tableau

```
1 import sqlite3
2
3 conn = sqlite3.connect("database.db")
4 c = conn.cursor()
5 |
6 c.execute("""
7 CREATE TABLE IF NOT EXISTS employees(
8     prenom text,
9     nom text
10 )
11 """)
12 conn.commit()
13 conn.close()
```

# Les bases de données

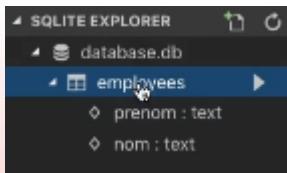
## SQLITE Ajouter des données

On créait un dictionnaire d avec deux clefs:  
prénom et nom contenant respectivement  
le prénom Paul et le nom Dupond.  
Ensuite on passe les clefs prénom et nom  
ainsi que le Dictionnaire d à c.execute.

```
1 import sqlite3
2
3 conn = sqlite3.connect("database.db")
4 c = conn.cursor()
5 c.execute("""
6 CREATE TABLE IF NOT EXISTS employees
7 (
8     prenom text,
9     nom text
10 )
11 """)
12
13 d = {"prenom": "Paul", "nom": "Dupond"}
14 c.execute("INSERT INTO employees VALUES (:prenom, :nom)", d)
15
16 conn.commit()
17 conn.close()
```

# Les bases de données visualiser une base de données dans Visual Studio Code

- Dans VSCode:
- File -> Préférences -> Extensions
- Rechercher Sqlite dans la boite « Search Extensions in Marketplace »
- Installer l'extension SQLite (icone d'une plume)
- Faire **crtl + shift + p** cela ouvre une boite de recherche en haut de VSCode
- Taper « sqlite open » dans le champ texte et sélectionner la base de données désirée
- Un petit sqlite explorer doit s'afficher en bas à gauche, il faut cliquer sur la flèche.



- Pour afficher le contenu de la base, cliquer sur la flèche après employees =>
- Penser à fermer la base de donnée:
  - Sqlite: Close Database

```
SELECT * FROM 'employees';
```

#	prenom	nom
1	Paul	Dupond

1 / 1

# Les bases de données

## SQLITE Récupérer des données

Requête avec une syntaxe classique

```
6 CREATE TABLE IF NOT EXISTS employees
7 (
8     prenom text,
9     nom text
10 )
11 """")
12
13 c.execute("SELECT * FROM employees WHERE prenom='Pierre'")
14 donnees = c.fetchall()
15 print(donnees)
16
17 conn.commit()
18 conn.close()
```

Même requête avec une syntaxe utilisant les dictionnaires

```
6 CREATE TABLE IF NOT EXISTS employees
7 (
8     prenom text,
9     nom text
10 )
11 """
12
13 d = {"a": "Pierre"}
14 c.execute("SELECT * FROM employees WHERE prenom=:a", d)
15 donnees = c.fetchall()
16 print(donnees)
17
18 conn.commit()
19 conn.close()
```

# Les bases de données

## SQLITE Récupérer des données (fetchall)

Il est important de noter que fetchall est un générateur et le générateur une fois que l'on va avoir récupérer les données de la requête il est épuisé. (on ne peut pas le refaire une deuxième fois.) Donc avant de pouvoir refaire un fetchall il faut refaire une requête ex:

```
c.execute("SELECT * FROM employees")
donnees = c.fetchall()
print(donnees)
c.execute("SELECT * FROM employees")
donnees = c.fetchall()
print(donnees)

conn.commit()
conn.close()
```

# Les bases de données

## SQLITE Récupérer des données (fetchone)

Il est important de noter que fetchone va avoir récupérer les données une à une. Une fois qu'on a atteint la fin de la requête il n'y a plus rien (on ne peut pas le refaire une deuxième fois.) Donc avant de pouvoir refaire un fetchone ou un fetchall il faut refaire une requête ex:

```
14 c.execute("SELECT * FROM employees")
15 premier = c.fetchone()
16 print(premier)
17 deuxieme = c.fetchone()
18 print(deuxieme)
19 troisieme = c.fetchone()
20 print(troisieme)
21 quatrieme = c.fetchone()
22 print(quatrieme)
23 cinquieme = c.fetchone()
24 print(cinquieme)
```

```
c.execute("SELECT * FROM employees")
c.fetchall()
cinquieme = c.fetchone()
print(cinquieme)
```

Cela ne fonctionne pas car fetchall à vider tout le contenu

```
('Pierre', 'Durand')
('Julie', 'Delpine')
('Georges', 'Clooney')
('Pierre', 'Dupuis')
None
```

# Les bases de données SQLITE

## Mettre à jour des données (update)

```
5 c.execute(""""
6 CREATE TABLE IF NOT EXISTS employees
7 (
8     prenom text,
9     nom text,
10    salaire int
11 )
12 """
13
14 d = {"salaire": 10000, "prenom": "Paul", "nom": "Dupont"}
15
16 c.execute("""UPDATE employees SET salaire=:salaire
17 WHERE prenom=:prenom AND nom=:nom""", d)
18
19 conn.commit()
20 conn.close()
```

# Les bases de données SQLITE

## Supprimer des données

```
4 c = conn.cursor()
5 c.execute("""
6 CREATE TABLE IF NOT EXISTS employees
7 (
8     prenom text,
9     nom text
10 )
11 """)
12
13 c.execute("DELETE FROM employees WHERE prenom='Paul'")
14
15 conn.commit()
16 conn.close()
```

# Projet

# Projet 'liste'

<https://www.udemy.com/course/the-complete-python-course/learn/lecture/9487198#overview>

## app.py

```
from MilestoneProject2.Utils import database

def prompt_add_book():
    name = input("Nom du livre ?")
    author = input("Auteur du livre ?")
    database.add_book(name, author)
```

```
def list_books():
    books = database.get_all_books()
    for book in books:
        read = 'YES' if book['read'] == '1' else 'NO'
        print(f"{book['name']} by {book['author']}, read: {read}")
```

```
def prompt_read_book():
    name = input("What is the name of the book you read: ")
    database.mark_book_as_read(name)
```

```
def prompt_delete_book():
    name = input("Nom du livre à supprimer?")
    database.delete_book(name)
```

```
USER_CHOICE = ""
```

```
Enter:
```

- 'a' to add a new book
- 'l' to list all books
- 'r' to mark a book as read

- 'd' to delete a book
- 'q' to quit

```
Your choice:""
```

```
user_options = {
    'a': prompt_add_book,
    'l': list_books,
    'r': prompt_read_book,
    'd': prompt_delete_book,
}
```

```
def menu():
    selection = input(USER_CHOICE)
    while selection != 'q':
        if selection in user_options:
            selected_function = user_options[selection]
            selected_function()
        else:
            print('unknown, try again')

    selection = input(USER_CHOICE)
    # def prompt_add_book() ask for book name and author
    # def list_books() show all the books in our list
    # def prompt_read_book() ask for book name and change it to "read" in our list
    # def prompt_delete_book() ask for book name and remove book from list
```

```
menu()
```

# Projet

database.py écriture dans une liste

```
books = []

def create_book_table():
    pass

def get_all_books():
    return books

def add_book(name, author):
    books.append({'name': name, 'author': author, 'read': False})

def mark_book_as_read(name):
    for book in books:
        if book['name'] == name:
            book['read'] = True

def delete_book(name):
    global books
    books = [book for book in books if book['name'] != name]
```

# Projet

## app.py

```
from MilestoneProject2.Utils import database

def prompt_add_book():
    name = input("Nom du livre ?")
    author = input("Auteur du livre ?")
    database.add_book(name, author)

def list_books():
    books = database.get_all_books()
    for book in books:
        read = 'YES' if book['read'] == '1' else 'NO'
        print(f'{book["name"]} by {book["author"]}, read: {read}')

def prompt_read_book():
    name = input("What is the name of the book you read: ")
    database.mark_book_as_read(name)

def prompt_delete_book():
    name = input("Nom du livre à supprimer?")
    database.delete_book(name)

USER_CHOICE = """
Enter:
- 'a' to add a new book
- 'l' to list all books
- 'r' to mark a book as read
"""


```

```
- 'd' to delete a book
- 'q' to quit

Your choice:""""

user_options = {
    'a': prompt_add_book,
    'l': list_books,
    'r': prompt_read_book,
    'd': prompt_delete_book,
}

def menu():
    selection = input(USER_CHOICE)
    while selection != 'q':
        if selection in user_options:
            selected_function = user_options[selection]
            selected_function()
        else:
            print('unknown, try again')
            selection = input(USER_CHOICE)
    # def prompt_add_book() ask for book name and author
    # def list_books() show all the books in our list
    # def prompt_read_book() ask for book name and change it to "read" in our list
    # def prompt_delete_book() ask for book name and remove book from list

menu()
```

# Projet ‘fichier txt’

## database.py écriture dans un fichier txt

```
"""
Concerned with storing and retrieving books from a list.
"""

books_file = 'books.txt'

def add_book(name, author):
    with open('books_file', 'a') as file:
        file.write(f'{name},{author},0\n')

def get_all_books():
    with open('books_file', 'r') as file:
        lines = [line.strip().split(',') for line in file.readlines()]

    return [
        {'name': line[0], 'author': line[1], 'read': line[2]}
        for line in lines
    ]

def mark_book_as_read(name):
    books = get_all_books()
    for book in books:
        if book['name'] == name:
            book['read'] = '1'
            _save_all_books(books)
            break
    else:
        print("Ce livre ne fait pas partie de votre collection.")

def _save_all_books(books): # the function name start with _ as it is a private function, so this
    # function should be call from this file only.
    with open('books_file', 'w') as file:
        for book in books:
            file.write(f'{book["name"]},{book["author"]},{book["read"]}\n')

def delete_book(name):
    books = get_all_books()
    books = [book for book in books if book['name'] != name]
    _save_all_books(books)
```

# Projet

```
└─ MilestoneProject2
    ├─ Correction
    ├─ Utils
    ├─ app.py
    └─ books.json
```

```
from MilestoneProject2.Utils import database

def prompt_add_book():
    name = input("Nom du livre ?")
    author = input("Auteur du livre ?")
    database.add_book(name, author)

def list_books():
    books = database.get_all_books()
    for book in books:
        read = 'YES' if book['read'] else 'NO'
        print(f"{book['name']} by {book['author']}, read: {read}")

def prompt_read_book():
    name = input("What is the name of the book you read: ")
    database.mark_book_as_read(name)

def prompt_delete_book():
    name = input("Nom du livre à supprimer?")
    database.delete_book(name)

USER_CHOICE = """
Enter:
- 'a' to add a new book
- 'l' to list all books
- 'r' to mark a book as read
```

```
- 'd' to delete a book
- 'q' to quit

Your choice:""""

user_options = {
    'a': prompt_add_book,
    'l': list_books,
    'r': prompt_read_book,
    'd': prompt_delete_book,
}

def menu():
    database.create_book_table()
    selection = input(USER_CHOICE)
    while selection != 'q':
        if selection in user_options:
            selected_function = user_options[selection]
            selected_function()
        else:
            print('unknown, try again')
            selection = input(USER_CHOICE)
    # def prompt_add_book() ask for book name and author
    # def list_books() show all the books in our list
    # def prompt_read_book() ask for book name and change it to "read" in our list
    # def prompt_delete_book() ask for book name and remove book from list

menu()
```

# Projet ‘fichier JSON’

## database.py écriture dans un fichier json

```
import json

"""
Concerned with storing and retrieving books from a list.
"""

books_file = 'books.json'

def create_book_table():
    with open(books_file, 'w') as file:
        json.dump([], file)

def add_book(name, author):
    books = get_all_books()
    books.append({'name': name, 'author': author, 'read': False})
    _save_all_books(books)

def get_all_books():
    with open(books_file, 'r') as file:
        return json.load(file)

def _save_all_books(
    books): # the function name start with _ as it is a private function, so this function should
be call from this file only.
    with open(books_file, 'w') as file:
        json.dump(books, file)

def mark_book_as_read(name):
    books = get_all_books()
    for book in books:
        if book['name'] == name:
            book['read'] = True
    _save_all_books(books)

def delete_book(name):
    books = get_all_books()
    books = [book for book in books if book['name'] != name]
    _save_all_books(books)
```

# Projet 'Base de donnees'

<https://www.udemy.com/course/the-complete-python-course/learn/lecture/9445314#overview>

écriture dans une base de donnees

app.py

```
from MilestoneProject2.DB.Utils import database
```

```
def prompt_add_book():
    name = input("Nom du livre : ")
    author = input("Auteur du livre : ")
    database.add_book(name, author)
```

```
def list_books():
    books = database.get_all_books()
    for book in books:
        read = 'YES' if book['read'] else 'NO'
        print(f'{book["name"]} by {book["author"]}, read: {read}')
```

```
def prompt_read_book():
    name = input("What is the name of the book you read: ")
    database.mark_book_as_read(name)
```

```
def prompt_delete_book():
    name = input("Nom du livre à supprimer? ")
    database.delete_book(name)
```

```
USER_CHOICE = """
Enter:
- 'a' to add a new book
- 'l' to list all books
- 'r' to mark a book as read
- 'd' to delete a book
- 'q' to quit
```

```
Your choice:"""
```

```
user_options = {
    'a': prompt_add_book,
    'l': list_books,
    'r': prompt_read_book,
    'd': prompt_delete_book,
}
```

```
def menu():
    database.create_book_table()
    selection = input(USER_CHOICE)
    while selection != 'q':
        if selection in user_options:
            selected_function = user_options[selection]
            selected_function()
        else:
            print('Unknown, try again')
            selection = input(USER_CHOICE)
    # def prompt_add_book() ask for book name and author
    # def list_books() show all the books in our list
    # def prompt_read_book() ask for book name and change it to "read" in our list
    # def prompt_delete_book() ask for book name and remove book from list
```

```
menu()
```

# Projet

## Database.py

```
import sqlite3

"""
Concerned with storing and retrieving books from a list.
"""

def create_book_table():
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()
    cursor.execute('CREATE TABLE IF NOT EXISTS books(name text primary key, author text, read integer)')
    connection.commit()
    connection.close()

def add_book(name, author):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()
    try:
        cursor.execute('INSERT INTO books VALUES(?, ?, 0)', (name, author))
    except:
        print('Ce nom de livre est déjà présent dans la base de données !!!')
    connection.commit()
    connection.close()

def get_all_books():
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM books')

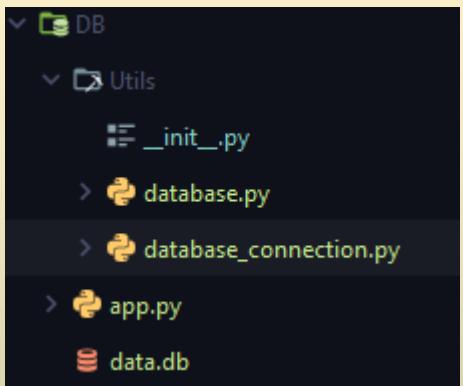
    books = [{'name': row[0], 'author': row[1], 'read': row[2]} for row in cursor.fetchall()] # [(name, author, read)]
    connection.close()
    return books

def mark_book_as_read(name):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()
    cursor.execute('UPDATE books SET read=1 WHERE name=?', (name,))
    connection.commit()
    connection.close()

def delete_book(name):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()
    cursor.execute('DELETE FROM books WHERE name=?', (name,))
    connection.commit()
    connection.close()
```

# Projet

developping our own context manager



Like for files where we use “while open ('filename', 'mode') as file:  
We can create our own context manager for opening the database:

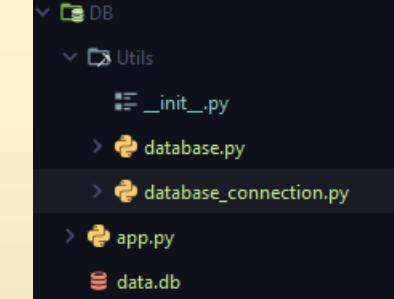
Database\_connection.py

```
import sqlite3

class DatabaseConnection:
    def __init__(self, host):
        self.connection = None
        self.host = host

    def __enter__(self):
        self.connection = sqlite3.connect(self.host)
        return self.connection

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type or exc_val or exc_tb:
            self.connection.close()
        else:
            self.connection.commit()
            self.connection.close()
```



# Projet

developping our own context manager

database.py

```
from MilestoneProject2.DB.Utils.database_connection import
DatabaseConnection
"""
Concerned with storing and retrieving books from a list.
"""

def create_book_table():
    with DatabaseConnection('data.db') as connection:
        cursor = connection.cursor()
        cursor.execute('CREATE TABLE IF NOT EXISTS books(name text primary
key, author text, read integer)')

def add_book(name, author):
    with DatabaseConnection('data.db') as connection:
        cursor = connection.cursor()
        cursor.execute('INSERT INTO books VALUES(?, ?, 0)', (name, author))

def get_all_books():
    with DatabaseConnection('data.db') as connection:
        cursor = connection.cursor()
        cursor.execute('SELECT * FROM books')
        books = [{'name': row[0], 'author': row[1], 'read': row[2]} for row in
cursor.fetchall()] # [(name, author, read)]
    return books

def mark_book_as_read(name):
    with DatabaseConnection('data.db') as connection:
        cursor = connection.cursor()
        cursor.execute('UPDATE books SET read=1 WHERE name=?', (name,))
```

# Projet

developping our own context manager

Further reading:

- [The Curious Case of Python's Context Manager](#), a great overview of context managers and how to define your own.
- [Introduction to Context Managers in Python](#), a nice read to review context managers.
- [SQLite AUTOINCREMENT : Why You Should Avoid Using It](#), Here's why AUTOINCREMENT can be a bad choice in SQLite.

# Typing in Python

# Typing in Python

<https://www.teclado.com/30-days-of-python/python-30-day-28-type-hinting>



Day 28 Type Hinting.pdf

The benefits of type hinting

Type hinting can be very helpful for a number of reasons.

1. It makes it easier to understand our code, because our helpful variable names are now accompanied by a description of the sort of data we expect to be assigned to them.

2. Most modern editors are able to make good use of our type annotations to provide more meaningful hints when we do things like calling functions.

3. We can use tools like mypy to check that our type annotations are being honoured, helping us catch bugs caused by passing around incorrect types.

One thing type annotations cannot do is actually prevent us from breaking the rules we outlined in our annotations. They're a development tool only, and don't have any effect on our code when we run the application.

Now that we have an idea about what type annotations can do for us, let's learn how to actually use them.

Let's look at the return values:

```
def delet_book(name) -> None:  
def get_all_books() -> List[Dict[str, Union[str, int]]]: (here you need to do an import: from typing import List, Dict, Union)
```

We can create our own type:

```
Book = Dict[str, Union[str, int]]
```

Now let's look at the parameters:

```
def add_book(name: str, author: str) -> None:
```

```
# Python 3.5+ supports 'type annotations' that can be  
# used with tools like Mypy to write statically typed Python:
```

```
def my_add(a: int, b: int) -> int:  
    return a + b
```

# Typing in Python

```
from typing import List, Dict, Union
from MilestoneProject2.DB.Utils.database_connection import
DatabaseConnection
"""
Concerned with storing and retrieving books from a list.
"""

def create_book_table() -> None:
    with DatabaseConnection('data.db') as connection:
        cursor = connection.cursor()
        cursor.execute('CREATE TABLE IF NOT EXISTS books(name text primary
key, author text, read integer)')

def add_book(name: str, author: str) -> None:
    with DatabaseConnection('data.db') as connection:
        cursor = connection.cursor()
        cursor.execute('INSERT INTO books VALUES(?, ?, 0)', (name, author))

def get_all_books() -> List[Dict[str, Union[str, int]]]:
    with DatabaseConnection('data.db') as connection:
        cursor = connection.cursor()
        cursor.execute('SELECT * FROM books')
        books = [{'name': row[0], 'author': row[1], 'read': row[2]} for row in
cursor.fetchall()] # [(name, author, read)]
    return books

def mark_book_as_read(name) -> None:
    with DatabaseConnection('data.db') as connection:
        cursor = connection.cursor()
        cursor.execute('UPDATE books SET read=1 WHERE name=?', (name,))
        cursor.execute('DELETE FROM books WHERE name=?', (name,))
```

# Typing in Python

Further reading:

<https://docs.python.org/3/library/typing.html>

<https://www.python.org/dev/peps/pep-0483/>

# Créez des interfaces graphiques avec Tkinter

# TKINTER

## Memento Tkinter avec Python

```
from tkinter import *

Fenêtre
fen1=Tk()
#ouvre une instance de fenêtre
fen1.title('Titre')
#titre
lbl1=Label(fen1,text='Texte',bg='red') # label rouge
lbl1.pack() #ajuste au contenu et affiche
btn1=Button(fen1,text='Quitter', command=fen1.destroy)
btn1.pack()
fen1.mainloop #démarrer la réception d'événements
options couleurs: fg=foreground, bg=background
```

### Widgets containers : Frame et Canvas

Widget Frame= cadre avec couleur et bordure pour contenir des widgets  
 Frame(fen1,bd=2,bg="#d0d0b0",relief=FLAT)  
 reliefs: RAISED (sortant), SUNKEN (entrant), FLAT (plat), RIDGE (crête), GROOVE (sillon), SOLID (bordure)  
Widget Canvas: surface pour dessiner  
 can1=Canvas(fen1 ou Tk(), bg='dark gray',height=200, width=200)  
 can1.pack(side=LEFT)

### Méthodes de positionnements

1.pack(): positionnés dans le flux  
 btn.pack(side=LEFT,padx=5,pady=5)  
 side : LEFT, RIGHT, TOP, BOTTOM  
 padx : espace bord gauche dte  
 pady: espace bord haut bas  
2.Méthode grid: positionnés avec une grille  
 txt1.grid(row=0) #1<sup>e</sup> champ en haut à gche  
 txt2.grid(row=1) #2<sup>e</sup> ligne  
 txt3.grid(row=0,column=1) #1ere ligne 2<sup>eme</sup> col  
 options: sticky N, S, E, O alignment haut, bas...  
 colspan=2 deux colonnes / rowspan=2 2 lignes  
 padx =5 espacement gche dte  
 pady= 5 espacement haut bas  
3.place() en coordonnées

Tracés dans un canevas  
 can1.create\_line(x1,y1,x2,y2,fill='couleur')  
 can1.create\_rectangle(x1,y1,x2,y2)  
 can1.create\_arc(x1,y1,x2,y2,start=0,extent=90)  
 start : angle de départ / extent angle  
ovale dans rectangle fictif du coin sup gche au inf. droit  
 can1.create\_oval(x1,y1,x2,y2,outline='couleur')  
 cercle de centre x,y de rayon r :  
 can1.create\_oval(x-r,y-r,x+r,y+r,outline='blue')  
polygone fermé: dernier pt rejoint le 1er  
 create\_polygon(x1,y1,x2,...)  
texte:  
 can1.create\_text(x,y,text="mon texte", font="Arial 12")  
 modifier par montexte.configure(text="nouveau")  
Paramètres :  
 fill ='couleur' / outline='couleur' / width=1  
Effacer: can1.delete(ALL)  
 fen.update() #force la mise à jour

Widgets de base  
 Button - Label -Entry (champ d'entrée texte) - Text  
 Listbox - Menu - Menubutton (déroulant)  
 Message(texte mis en forme sur largeur hauteur)  
 Checkbutton - Scrollbar (ascenseur)

Widget: Scale (curseur)  
 Scale( Tk(), length=200, orient=HORIZONTAL,  
 label="Echelle", troughcolor='darkgray',  
 sliderlength=20, showvalue=0, \_from=-25, to=125,  
 tickinterval=25, command=nomcommande)

Widget: RadioButton  
 Radiobutton(self,text="texte",variable=var, value=valeur, command=nomcdé)  
 où variable sera la même pour tous les boutons liés  
 on récupère la valeur choisie par: var.get()

Widget Entry (champ d'entrée texte)  
 entree=Entry(fen1)  
 ou entree=Entry(fen1, text variable= var)

Accès au widget Entry  
 entree.get() #récupere la chaîne complète  
 entree.insert(pos,"texte à insérer") #insertion ss-chaîne  
 où pos = entier entre 0 et len -1,  
 0 pour début de chaîne, END pour fin  
 entree.configure(font="Arial 15 Normal") #modif param.

Detection d'entrées clavier  
 #associe la fonction fct par référence sans () au retour  
 entree.bind("<Return>",fct)  
 def fct(event):  
 #interprete un calcul entré au clavier  
 result= eval(entree.get())  
 #affichage sans un Label  
 lbl1.configure(text="Affichage")  
 event.char #contient la lettre pressée

Détection de la souris  
 event transmet ts les événements clavier-souris  
 event.x , event.y coordonnées du click

fen.bind("<Button-1>",fct) #click  
Évènements souris  
 <Button-1> #btn souris gauche droit  
 <Double-1> #double click btn gauche  
 <Button1-Motion> #déplacement bouton gche enfoncé  
 <ButtonRelease-1> #relaché du bouton gauche  
 <Button-2> #click sur molette centrale  
 <Button-3> #bouton droit souris

### Évènements clavier

<KeyPress-A >	#Clavier touche A Majuscule
<KeyPress-a >	#Clavier touche a minuscule
<Control-Shift-KeyPress-A >	#Ctrl Shift A
<Any-KeyPress>=<Key>	#clavier n'importe quelle touche
<KP-Down> <KP-Enter><KP-1>	#clavier numérique
<Delete> <Escape> <F1> <Insert><Tab> <Return> <Pause>	
<Down> <Up><Left><Right>	
<Enter><Leave>	#entrée et sortie de souris sur widget
<Configure>	#Redimensionnement fenêtre

# TKINTER – les classes de widgets

Widget	Description
Button	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque.
Canvas	Un espace pour disposer divers éléments graphiques. Ce widget peut être utilisé pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés.
Checkbutton	Une case à cocher qui peut prendre deux états distincts (la case est cochée ou non). Un clic sur ce widget provoque le changement d'état.
Entry	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque à partir du clavier.
Frame	Une surface rectangulaire dans la fenêtre, où l'on peut disposer d'autres widgets. Cette surface peut être colorée. Elle peut aussi être décorée d'une bordure.
Label	Un texte (ou libellé) quelconque (éventuellement une image).
Listbox	Une liste de choix proposés à l'utilisateur, généralement présentés dans une sorte de boîte. On peut également configurer la Listbox de telle manière qu'elle se comporte comme une série de « boutons radio » ou de cases à cocher.
Menu	Un menu. Ce peut être un menu déroulant attaché à la barre de titre, ou bien un menu « pop up » apparaissant n'importe où à la suite d'un clic.
Menubutton	Un bouton-menu, à utiliser pour implémenter des menus déroulants.
Message	Permet d'afficher un texte. Ce widget est une variante du widget Label, qui permet d'adapter automatiquement le texte affiché à une certaine taille ou à un certain rapport largeur/hauteur.
Radiobutton	Représente (par un point noir dans un petit cercle) une des valeurs d'une variable qui peut en posséder plusieurs. Cliquer sur un bouton radio donne la valeur correspondante à la variable, et « vide » tous les autres boutons radio associés à la même variable.
Scale	Vous permet de faire varier de manière très visuelle la valeur d'une variable, en déplaçant un curseur le long d'une règle.
Scrollbar	Ascenseur ou barre de défilement que vous pouvez utiliser en association avec les autres widgets : Canvas, Entry, Listbox, Text.
Text	Affichage de texte formaté. Permet aussi à l'utilisateur d'édition le texte affiché. Des images peuvent également être insérées.
Toplevel	Une fenêtre affichée séparément, au premier plan.

# Créez des interfaces graphiques avec Tkinter

## Créez des interfaces graphiques avec Tkinter

Nous allons maintenant voir comment créer des interfaces graphiques à l'aide d'un module présent par défaut dans Python : **Tkinter**.

Ce module permet de créer des interfaces graphiques en offrant une passerelle entre Python et la bibliothèque **Tk**.

Vous allez pouvoir apprendre dans ce chapitre à créer des fenêtres, créer des boutons, faire réagir vos objets graphiques à certains événements...

# Présentation de Tkinter

**Tkinter (Tk interface)** est un module intégré à la bibliothèque standard de Python, bien qu'il ne soit pas maintenu directement par les développeurs de Python. Il offre un moyen de créer des interfaces graphiques via Python.

Tkinter est disponible sur Windows et la plupart des systèmes Unix. Les interfaces que vous pourrez développer auront donc toutes les chances d'être portables d'un système à l'autre.

Notez qu'il existe d'autres bibliothèques pour créer des interfaces graphiques. Tkinter a l'avantage d'être disponible par défaut, sans nécessiter une installation supplémentaire.

Pour savoir si vous pouvez utiliser le module Tkinter via la version de Python installée sur votre système, tapez dans l'interpréteur en ligne de commande Python :

```
from tkinter import *
```

Si une erreur se produit, vous devrez aller vous renseigner sur [la page du Wiki Python consacrée à Tkinter](#).

# Votre première interface graphique

Nous allons commencer par voir le code minimal pour créer une fenêtre avec Tkinter. Petit à petit, nous allons apprendre à rajouter des choses, mais commençons par voir la base de code que l'on retrouve d'une interface Tkinter à l'autre.

Étant en Python, ce code minimal est plutôt court :

```
"""Premier exemple avec Tkinter.
```

*On crée une fenêtre simple qui souhaite la bienvenue à l'utilisateur.*

```
"""
```

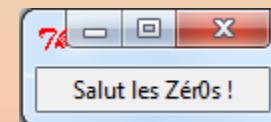
```
# On importe Tkinter  
from tkinter import *
```

*# On crée une fenêtre, racine de notre interface*  
*fenetre = Tk()*

*# On crée un label (ligne de texte) souhaitant la bienvenue*  
*# Note : le premier paramètre passé au constructeur de Label est notre*  
*# interface racine*  
*champ\_label = Label(fenetre, text="Salut les ZérOs !")*

*# On affiche le label dans la fenêtre*  
*champ\_label.pack()*

*# On démarre la boucle Tkinter qui s'interrompt quand on ferme la*  
*fenêtre*  
*fenetre.mainloop()*



Vous pouvez voir le résultat à la figure suivante.

# Votre première interface graphique

Vous pouvez recopier ce code dans un fichier.py(n'oubliez pas de rajouter la ligne spécifiant l'encodage). Vous pouvez ensuite exécuter votre programme, ce qui affiche une fenêtre (simple, certes, mais une fenêtre tout de même).

Comme vous pouvez le voir, la fenêtre est tout juste assez grande pour que le message s'affiche.

Regardons le code d'un peu plus près.

1. On commence par importer **Tkinter**, sans grande surprise.
2. On crée ensuite un objet de la classe **Tk**. La plupart du temps, cet objet sera la fenêtre principale de notre interface.
3. On crée un **Label**, c'est-à-dire un objet graphique affichant du texte
4. On appelle la méthode **pack** de notre **Label**. Cette méthode permet de positionner l'objet dans notre fenêtre (et, par conséquent, de l'afficher).
5. Enfin, on appelle la méthode **mainloop** de notre fenêtre racine. Cette méthode ne retourne que lorsqu'on ferme la fenêtre.

Quelques petites précisions :

- Nos objets graphiques (boutons, champs de texte, cases à cocher, barres de progression...) sont appelés des **widgets**.
- On peut préciser plusieurs options lors de la construction de nos **widgets**. Ici, on définit l'option **text** de notre **Label** à "Salut les ZérOs!".

Il existe d'autres options communes à la plupart des widgets (la couleur de fond **bg**, la couleur du widget **fg**, etc.) et d'autres plus spécifiques à un certain type de widget. Le **Label** par exemple possède l'option **text** représentant le texte affiché par le **Label**.

Comme nous l'avons vu, vous pouvez modifier des options lors de la création du widget. Mais vous pouvez aussi en modifier après :

```
champ_label["text"]
>>> 'Salut les ZérOs !'
champ_label["text"] = "Maintenant, au revoir !"
champ_label["text"]
>>> 'Maintenant, au revoir !'
```

Comme vous le voyez, vous passez entre crochets (comme pour accéder à une valeur d'un dictionnaire) le nom de l'option. C'est le même principe pour accéder à la valeur actuelle de l'option ou pour la modifier. Nous allons voir quelques autres widgets de Tkinter à présent.

# De nombreux widgets

Tkinter définit un grand nombre de widgets pouvant être utilisés dans notre fenêtre. Nous allons en voir ici quelques-uns.

## Les widgets les plus communs

### Les labels

C'est le premier widget que nous avons vu, hormis notre fenêtre principale qui en est un également. On s'en sert pour afficher du texte dans notre fenêtre, du texte qui ne sera pas modifié par l'utilisateur.

```
champ_label = Label(fenetre, text="contenu de notre champ label")
champ_label.pack()
```

N'oubliez pas que, pour qu'un widget apparaisse, il faut :

- qu'il prenne, en premier paramètre du constructeur, la fenêtre principale ;
- qu'il fasse appel à la méthode pack.

La méthode **pack** permet de positionner un objet dans une fenêtre ou dans un cadre, nous verrons plus loin quelques-uns de ses paramètres optionnels.

### Les boutons

Les boutons sont des widgets sur lesquels on peut cliquer et qui peuvent déclencher des actions ou commandes comme nous le verrons ultérieurement plus en détail.

```
bouton_quitter = Button(fenetre, text="Quitter", command=fenetre.quit)
bouton_quitter.pack()
```

J'imagine que vous posez des questions sur le dernier paramètre passé à notre constructeur de Button. Il s'agit de l'action liée à un clic sur le bouton. Ici, c'est la méthode quit de notre fenêtre racine qui est appelée.

Ainsi, quand vous cliquez sur le bouton Quitter, la fenêtre se ferme. Nous verrons plus tard comment créer nos propres commandes.

Si vous faites des tests depuis l'interpréteur Python en ligne de commande, la fenêtre Tk reste ouverte tant que la console reste ouverte. Le bouton Quitter interrompra la boucle mainloop mais ne fermera pas l'interface.

# De nombreux widgets

## Une ligne de saisie

Le widget que nous allons voir à présent est une zone de texte dans lequel l'utilisateur peut écrire. En fait de zone, il s'agit d'une ligne simple. On préférera créer une variable Tkinter associée au champ de texte. Regardez le code qui suit :

```
var_texte = StringVar()  
ligne_texte = Entry(fenetre, textvariable=var_texte, width=30)  
ligne_texte.pack()
```

À la ligne 1, nous créons une variable Tkinter. En résumé, c'est une variable qui va ici contenir le texte de notre Entry. Il est possible de lier cette variable à une méthode de telle sorte que la méthode soit appelée quand la variable est modifiée (l'utilisateur écrit dans le champ Entry).

Pour en savoir plus, je vous renvoie à la méthode trace de la variable.

Comme vous l'avez peut-être remarqué, le widget Entry n'est qu'une zone de saisie. Pour que l'utilisateur sache ce qu'il doit y écrire, il pourrait être utile de lui mettre une indication auprès du champ. Le widget Label est le plus approprié dans ce cas.

Notez qu'il existe également le widget Text qui représente un champ de texte à plusieurs lignes.

Les cases à cocher

Les cases à cocher sont définies dans la classe Checkbutton. Là encore, on utilise une variable pour surveiller la sélection de la case.

Pour surveiller l'état d'une case à cocher (qui peut être soit active soit inactive), on préférera créer une variable de type IntVar plutôt que StringVar, bien que ce ne soit pas une obligation.

# De nombreux widgets

```
var_texte = StringVar()  
ligne_texte = Entry(fenetre, textvariable=var_texte, width=30)  
ligne_texte.pack()
```

Vous pouvez ensuite contrôler l'état de la case à cocher en interrogeant la variable :

```
var_case.get()
```

# De nombreux widgets

Si la case est cochée, la valeur renvoyée par la variable sera1. Si elle n'est pas cochée, ce sera0.

Notez qu'à l'instar d'un bouton, vous pouvez lier la case à cocher à une commande qui sera appelée quand son état change.

## Les boutons radio

Les boutons radio (radio buttons en anglais) sont des boutons généralement présentés en groupes. C'est, à proprement parler, un ensemble de cases à cocher mutuellement exclusives : quand vous cliquez sur l'un des boutons, celui-ci se sélectionne et tous les autres boutons du même groupe se désélectionnent.

Ce type de bouton est donc surtout utile dans le cadre d'un regroupement.

Pour créer un groupe de boutons, il faut simplement qu'ils soient tous associés à la même variable (là encore, une variable Tkinter). La variable peut posséder le type que vous voulez.

Quand l'utilisateur change le bouton sélectionné, la valeur de la variable change également en fonction de l'option **value** associée au bouton. Voyons un exemple :

```
var_choix = StringVar()

choix_rouge = Radiobutton(fenetre, text="Rouge", variable=var_choix, value="rouge")
choix_vert = Radiobutton(fenetre, text="Vert", variable=var_choix, value="vert")
choix_bleu = Radiobutton(fenetre, text="Bleu", variable=var_choix, value="bleu")

choix_rouge.pack()
choix_vert.pack()
choix_bleu.pack()
```

Pour récupérer la valeur associée au bouton actuellement sélectionné, interrogez la variable :

```
var_choix.get()
```

# De nombreux widgets

## Les listes déroulantes

Ce widget permet de construire une liste dans laquelle on peut sélectionner un ou plusieurs éléments. Le fonctionnement n'est pas tout à fait identique aux boutons radio. Ici, la liste comprend plusieurs lignes et non un groupe de boutons.

Créer une liste se fait assez simplement, vous devez commencer à vous habituer à la syntaxe :

```
liste = Listbox(fenetre)
liste.pack()
```

On insère ensuite des éléments. La méthode `insert` prend deux paramètres :

la position à laquelle insérer l'élément ;

l'élément même, sous la forme d'une chaîne de caractères.

Si vous voulez insérer des éléments à la fin de la liste, utilisez la constante `END` définie par **Tkinter** :

```
liste.insert(END, "Pierre")
liste.insert(END, "Feuille")
liste.insert(END, "Ciseau")
```

Pour accéder à la sélection, utilisez la méthode `curselection` de la liste. Elle renvoie un tuple de chaînes de caractères, chacune étant la position de l'élément sélectionné.

Par exemple, si `liste.curselection()` renvoie `('2',)`, c'est le troisième élément de la liste qui est sélectionné (*Ciseau* en l'occurrence).

## Organiser ses widgets dans la fenêtre

Il existe plusieurs widgets qui peuvent contenir d'autres widgets. L'un d'entre eux se nomme **Frame**. C'est un cadre rectangulaire dans lequel vous pouvez placer vos widgets... ainsi que d'autres objets **Frame** si besoin est.

# De nombreux widgets

Si vous voulez qu'un widget apparaisse dans un cadre, utilisez le Frame comme parent à la création du widget :

```
cadre = Frame(fenetre, width=768, height=576, borderwidth=1)
cadre.pack(fill=BOTH)

message = Label(cadre, text="Notre fenêtre")
message.pack(side="top", fill=X)
```

Comme vous le voyez, nous avons passé plusieurs arguments nommés à notre méthode *pack*. Cette méthode, je vous l'ai dit, sert à placer nos widgets dans la fenêtre (ici, dans le cadre).

En précisant *side="top"*, on demande à ce que le widget soit placé en haut de son parent (ici, notre cadre).

Il existe aussi l'argument nommé *fill* qui permet au widget de remplir le widget parent, soit en largeur si la valeur est *X*, soit en hauteur si la valeur est *Y*, soit en largeur et hauteur si la valeur est *BOTH*.

D'autres arguments nommés existent, bien entendu. Si vous voulez une liste exhaustive, rendez-vous [sur le chapitre consacré à Tkinter dans la documentation officielle de Python](#).

Une partie est consacrée au **packer** et à la méthode *pack*.

Notez qu'il existe aussi le widget *Labelframe*, un cadre avec un titre, ce qui nous évite d'avoir à placer un label en haut du cadre. Il se construit comme un *Frame* mais peut prendre en argument, à la construction, le texte représentant le titre : *cadre = Labelframe(..., text="Titre du cadre")*

## Bien d'autres widgets

Vous devez vous en douter, ceci n'est qu'une approche très sommaire de quelques widgets de **Tkinter**. Il en existe de nombreux autres et ceux que nous avons vus ont bien d'autres options.

Il est notamment possible de créer une barre de menus avec ses menus imbriqués, d'afficher des images, des **canvas** dans lequel vous pouvez dessiner pour personnaliser votre fenêtre... bref, il vous reste bien des choses à voir, même si ce chapitre ne peut pas couvrir tous ces widgets et options.

Je vous propose pour l'heure d'aller jeter un coup d'œil sur les commandes que nous avons effleurées jusqu'ici sans trop nous pencher dessus.

# Les commandes

Nous avons vu très brièvement comment faire en sorte qu'un bouton ferme une fenêtre quand on clique dessus :

```
bouton_quitter = Button(fenetre, text="Quitter", command=fenetre.quit)
```

C'est le dernier argument qui est important ici. Il a pour nom *command* et a pour valeur la méthode *quit* de notre fenêtre.

Sur ce modèle, nous pouvons créer assez simplement des commandes personnalisées, en écrivant des méthodes.

Cependant, il y a ici une petite subtilité : la méthode que nous devons créer ne prend aucun paramètre. Si nous voulons qu'un clic sur le bouton modifie le bouton lui-même ou un autre objet, nous devons placer nos widgets dans un corps de classe.

D'ailleurs, à partir du moment où on sort du cadre d'un test, il est préférable de mettre le code dans une classe.

On peut la faire hériter de *Frame*, ce qui signifie que notre classe sera un widget elle aussi. Voyons un code complet que j'expliquerai plus bas :

# Les commandes

```
from tkinter import *

class Interface(Frame):
    """Notre fenêtre principale.
    Tous les widgets sont stockés comme attributs de cette fenêtre."""

    def __init__(self, fenetre, **kwargs):
        Frame.__init__(self, fenetre, width=768, height=576, **kwargs)
        self.pack(fill=BOTH)
        self.nb_clic = 0

    # Création de nos widgets
    self.message = Label(self, text="Vous n'avez pas cliqué sur le bouton.")
    self.message.pack()

    self.bouton_quitter = Button(self, text="Quitter", command=self.quit)
    self.bouton_quitter.pack(side="left")

    self.bouton_cliquer = Button(self, text="Cliquez ici", fg="red",
                                 command=self.cliquer)
    self.bouton_cliquer.pack(side="right")

    def cliquer(self):
        """Il y a eu un clic sur le bouton.

        On change la valeur du label message."""

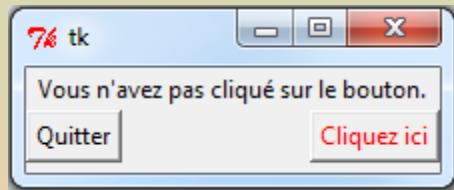
        self.nb_clic += 1
        self.message["text"] = "Vous avez cliqué {} fois.".format(self.nb_clic)
```

Et pour créer notre interface :

```
fenetre = Tk()
interface = Interface(fenetre)

interface.mainloop()
interface.destroy()
```

La figure suivante vous montre le résultat obtenu.



Dans l'ordre :

1. On crée une classe qui contiendra toute la fenêtre. Cette classe hérite de Frame, c'est-à-dire d'un cadre Tkinter.
2. Dans le constructeur de la fenêtre, on appelle le constructeur du cadre et on pack(positionne et affiche) le cadre.
3. Toujours dans le constructeur, on crée les différents widgets de la fenêtre. On les positionne et les affiche également.
4. On crée une méthode `bouton_cliquer`, qui est appelée quand on clique sur le bouton `cliquer`. Elle ne prend aucun paramètre. Elle va mettre à jour le texte contenu dans le labelself.message pour afficher le nombre de clics enregistrés sur le bouton.
5. On crée la fenêtre Tk qui est l'objet parent de l'interface que l'on instancie ensuite.
6. On rentre dans la boucle mainloop. Elle s'interrompra quand on fermera la fenêtre.
7. Ensuite, on détruit la fenêtre grâce à la méthode `destroy`.

#### Pour conclure

Ceci n'est qu'un survol, j'insiste sur ce point. Tkinter est une bibliothèque trop riche pour être présentée en un chapitre. Vous trouverez de nombreux exemples d'interfaces de par le Web, si vous cherchez quelque chose de plus précis.

- Tkinter est un module intégré à la bibliothèque standard et permettant de créer des interfaces graphiques.
- Les objets graphiques (boutons, zones de texte, cases à cocher...) sont appelés des **widgets**.
- Dans Tkinter, les **widgets** prennent, lors de leur construction, leur objet parent en premier paramètre.
- Chaque **widget** possède des options qu'il peut préciser comme arguments nommés lors de sa construction.
- On peut également accéder aux options d'un widget ainsi :*widget["nom\_option"]*

# TKinter

# Tkinter - buttons

```
from tkinter import *
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
ttk.Label(root, text="Hello, World!", padding=(30, 10)).pack()
```

```
import tkinter as tk
from tkinter import ttk

def greet():
    print("Hello, World!")

root = tk.Tk()

greet_button = ttk.Button(root, text="Greet", command=greet)
greet_button.pack(side="left", fill="y")

quit_button = ttk.Button(root, text="Quit", command=root.destroy)
quit_button.pack(side="left")

root.mainloop()
```



```
import tkinter as tk
from tkinter import ttk

def greet():
    print("Hello, World!")

root = tk.Tk()

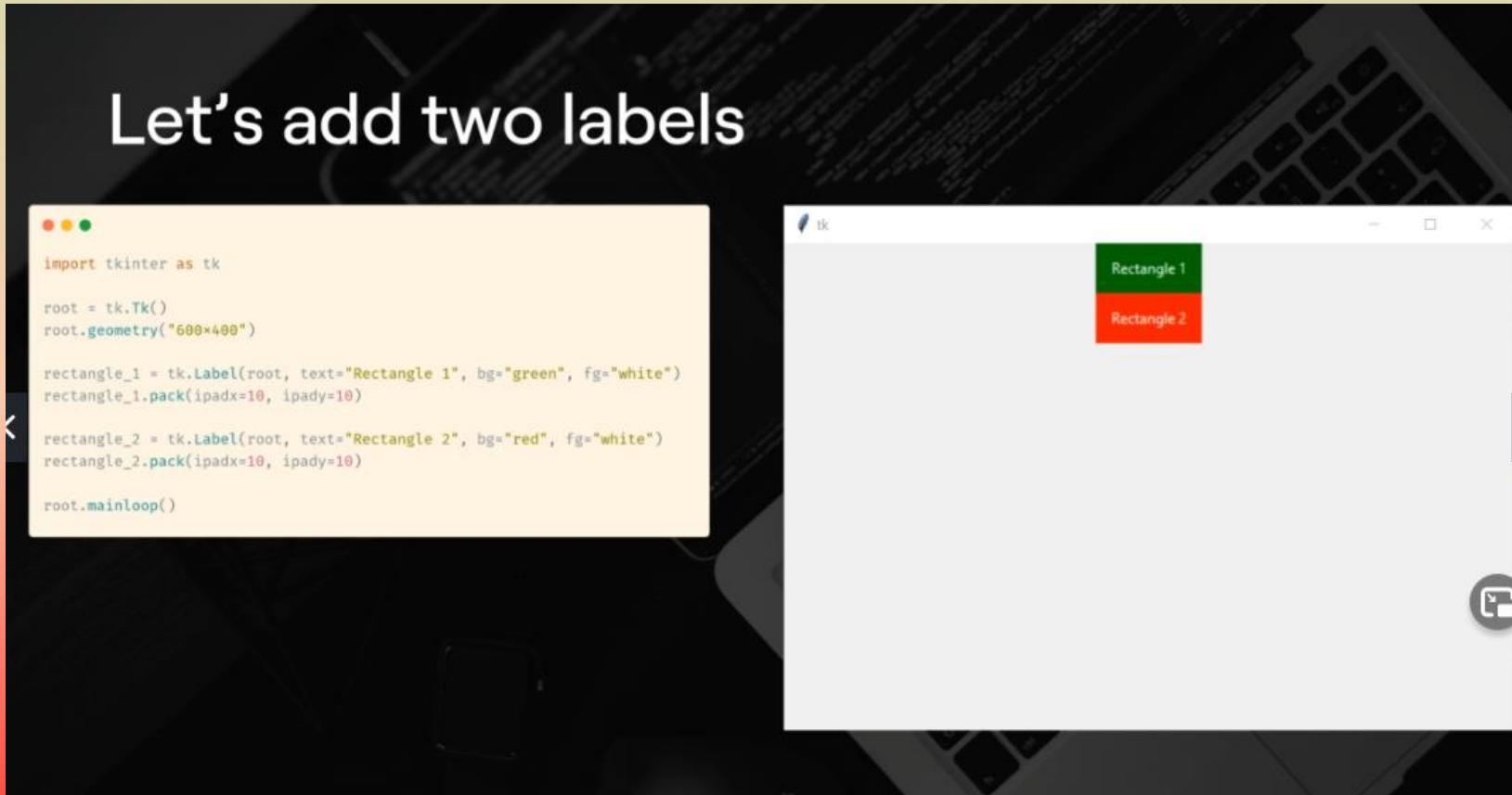
greet_button = ttk.Button(root, text="Greet", command=greet)
greet_button.pack(side="left", fill="x", expand=True)

quit_button = ttk.Button(root, text="Quit", command=root.destroy)
quit_button.pack(side="left")

root.mainloop()
```



# Packing components in tkinter



# Packing components in tkinter

Add `fill="x"`

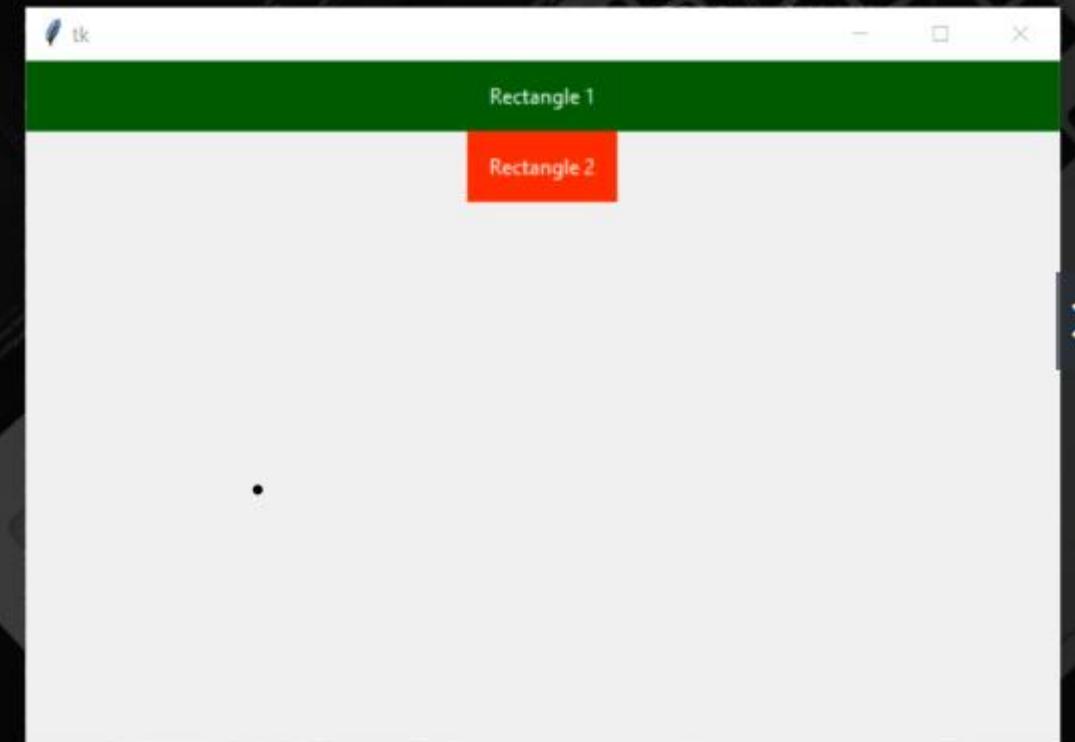
```
import tkinter as tk

root = tk.Tk()
root.geometry("600x400")

rectangle_1 = tk.Label(root, text="Rectangle 1", bg="green", fg="white")
rectangle_1.pack(ipadx=10, ipady=10, fill="x")

rectangle_2 = tk.Label(root, text="Rectangle 2", bg="red", fg="white")
rectangle_2.pack(ipadx=10, ipady=10)

root.mainloop()
```



# Packing components in tkinter

But add `fill="y"`...

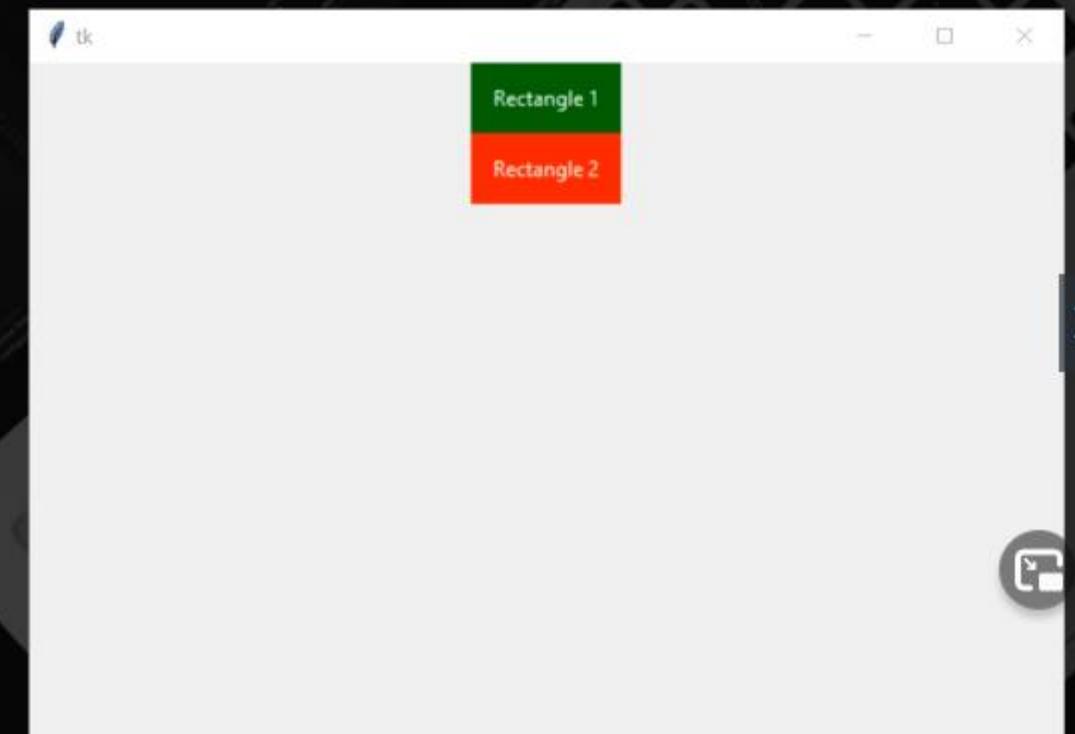
```
import tkinter as tk

root = tk.Tk()
root.geometry("600x400")

rectangle_1 = tk.Label(root, text="Rectangle 1", bg="green", fg="white")
rectangle_1.pack(ipadx=10, ipady=10, fill="y")

rectangle_2 = tk.Label(root, text="Rectangle 2", bg="red", fg="white")
rectangle_2.pack(ipadx=10, ipady=10)

root.mainloop()
```



With `fill="y"` nothing change, this is because these widgets have a default side of top, when side is top widgets get as much vertical space as they need for their content.

# Packing components in tkinter

## Using expand

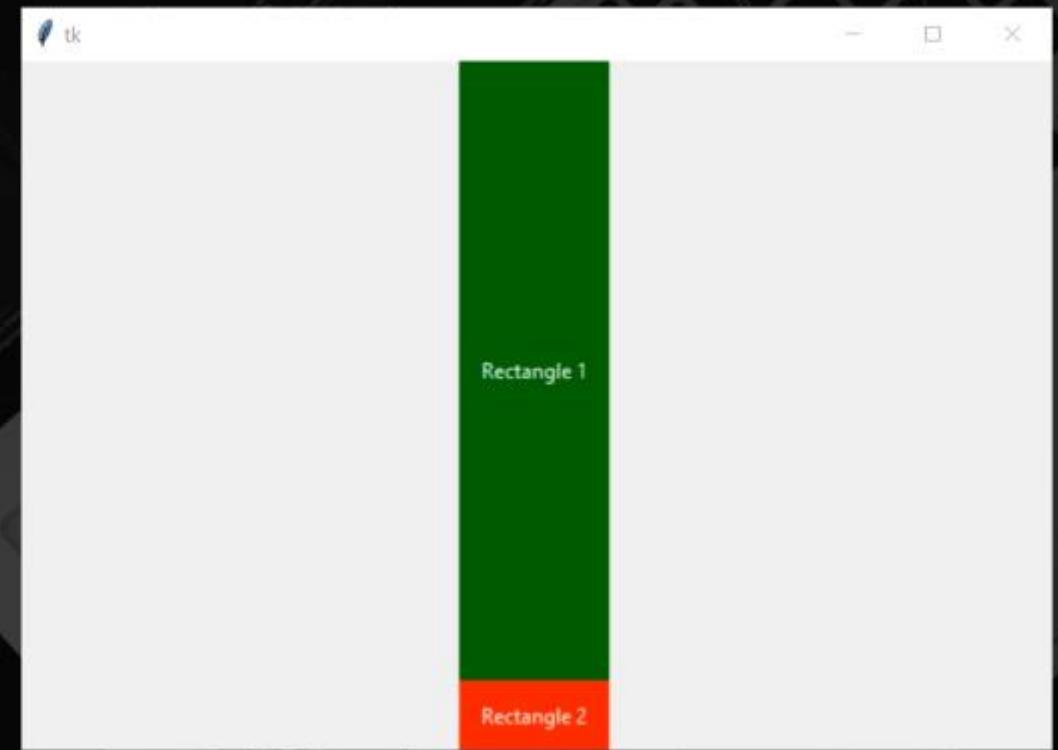
```
import tkinter as tk

root = tk.Tk()
root.geometry("600x400")

rectangle_1 = tk.Label(root, text="Rectangle 1", bg="green", fg="white")
rectangle_1.pack(ipadx=10, ipady=10, fill="y", expand=True)

rectangle_2 = tk.Label(root, text="Rectangle 2", bg="red", fg="white")
rectangle_2.pack(ipadx=10, ipady=10)

root.mainloop()
```



# Packing components in tkinter

## Using expand and fill="both"

```
import tkinter as tk

root = tk.Tk()
root.geometry("600x400")

rectangle_1 = tk.Label(root, text="Rectangle 1", bg="green", fg="white")
rectangle_1.pack(ipadx=10, ipady=10, fill="both", expand=True)

rectangle_2 = tk.Label(root, text="Rectangle 2", bg="red", fg="white")
rectangle_2.pack(ipadx=10, ipady=10)

root.mainloop()
```



# Packing components in tkinter

Using expand without fill="both"

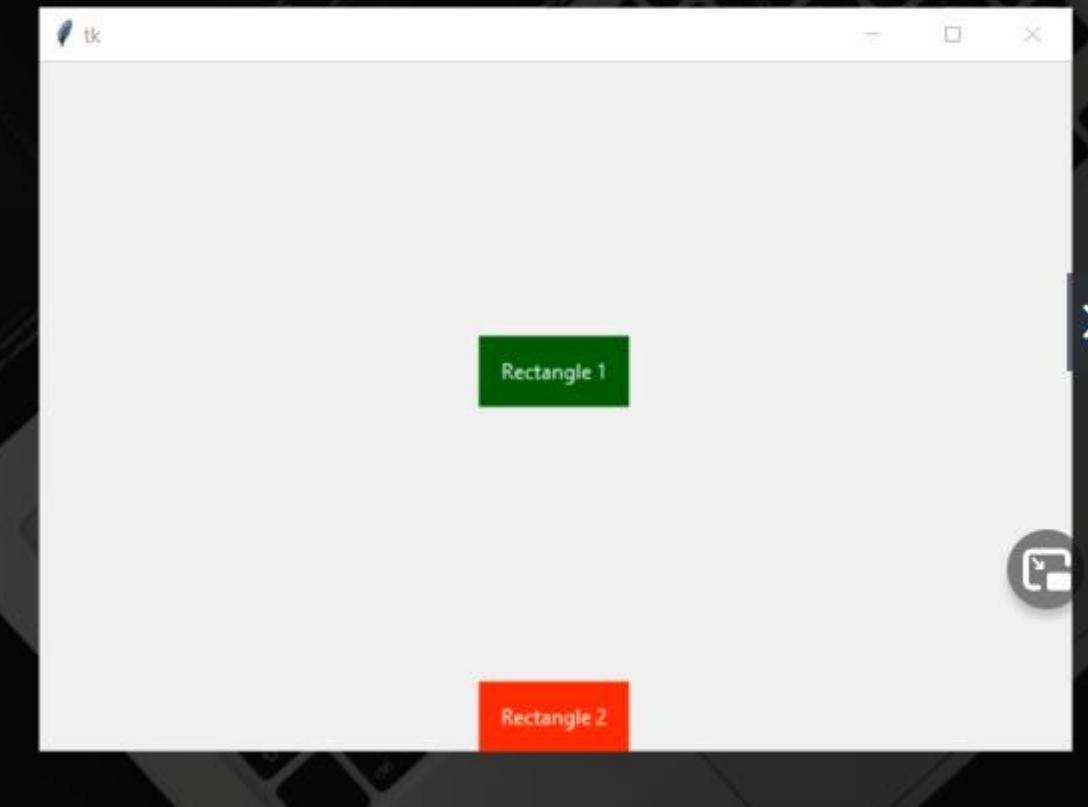
```
import tkinter as tk

root = tk.Tk()
root.geometry("600x400")

rectangle_1 = tk.Label(root, text="Rectangle 1", bg="green", fg="white")
rectangle_1.pack(ipadx=10, ipady=10, expand=True)

rectangle_2 = tk.Label(root, text="Rectangle 2", bg="red", fg="white")
rectangle_2.pack(ipadx=10, ipady=10)

root.mainloop()
```



# Packing components in tkinter

Using expand on both rectangles

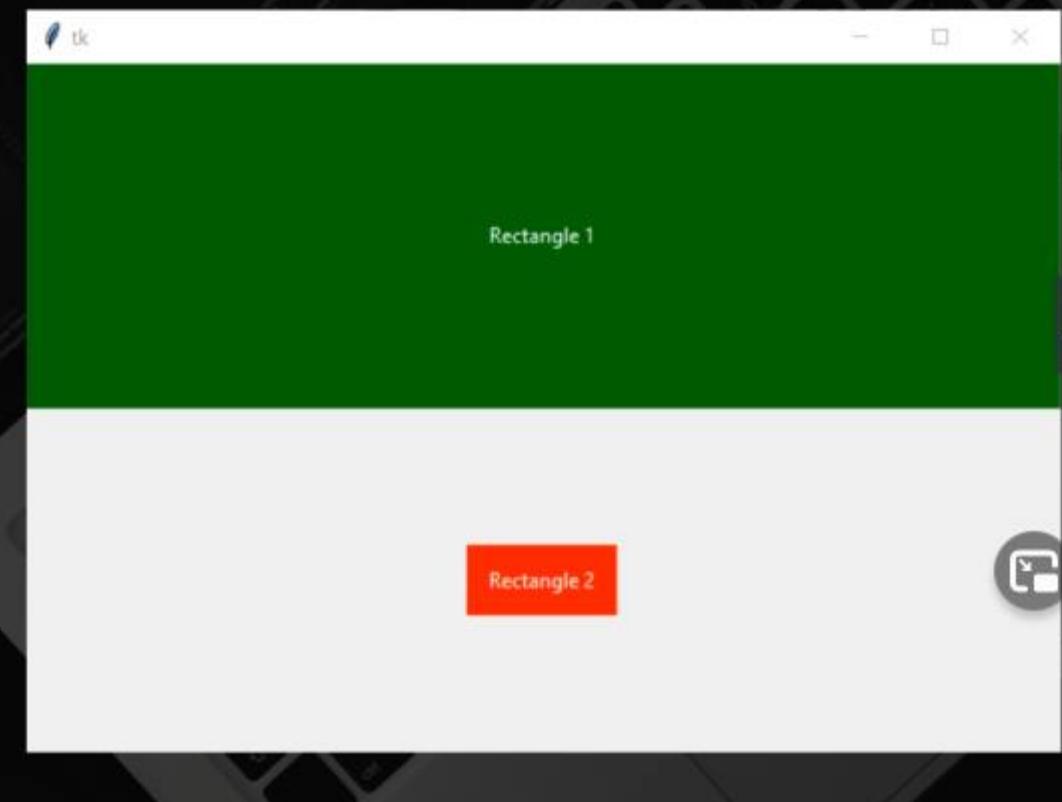
```
import tkinter as tk

root = tk.Tk()
root.geometry("600x400")

rectangle_1 = tk.Label(root, text="Rectangle 1", bg="green", fg="white")
rectangle_1.pack(ipadx=10, ipady=10, fill="both", expand=True)

rectangle_2 = tk.Label(root, text="Rectangle 2", bg="red", fg="white")
rectangle_2.pack(ipadx=10, ipady=10, expand=True)

root.mainloop()
```



# Packing components in tkinter

## Finally, side values

- ✓ We can specify which side to anchor components on.
- ✓ The default is side="top"
- ✓ When side="top" or side="bottom", widgets take up all the horizontal space
- ✓ When side="left" or side="right", widgets take up all the vertical space
- ✓ Widgets are pushed in the direction of their anchor point as far as possible within the available space

# Packing components in tkinter

Let's have a side="left" and side="top"

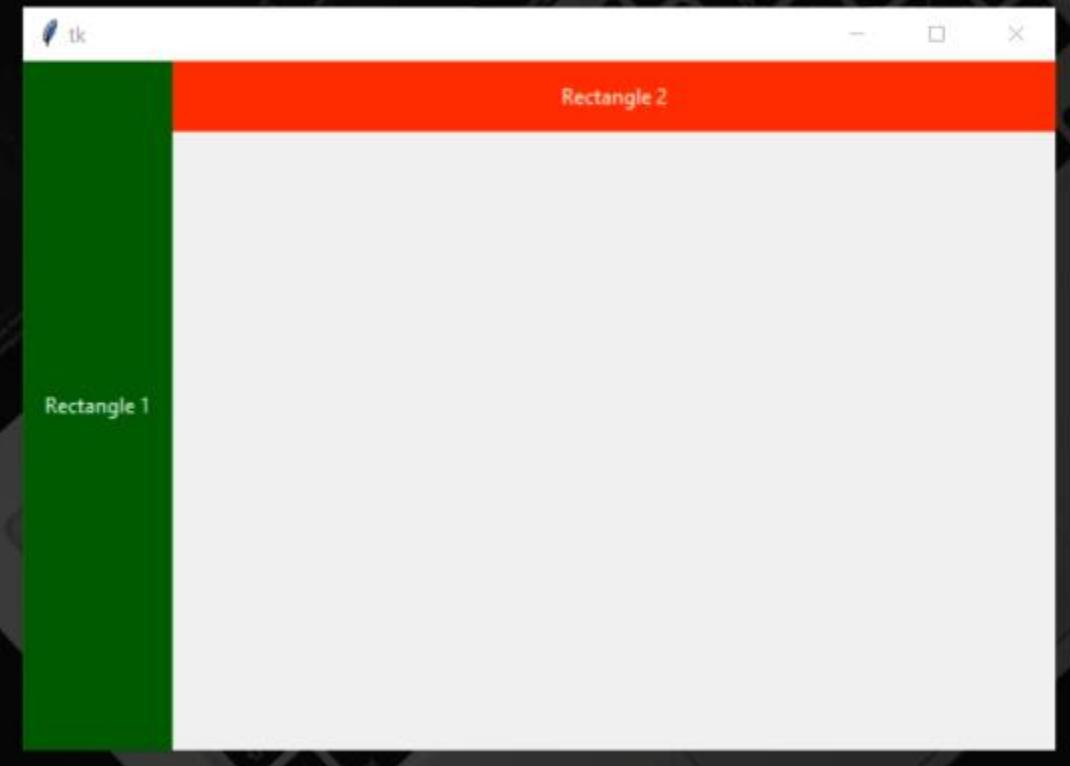
```
import tkinter as tk

root = tk.Tk()
root.geometry("600x400")

rectangle_1 = tk.Label(root, text="Rectangle 1", bg="green", fg="white")
rectangle_1.pack(side="left", ipadx=10, ipady=10, fill="both")

rectangle_2 = tk.Label(root, text="Rectangle 2", bg="red", fg="white")
rectangle_2.pack(ipadx=10, ipady=10, fill="both")

root.mainloop()
```



# Packing components in tkinter

## Expansion priority

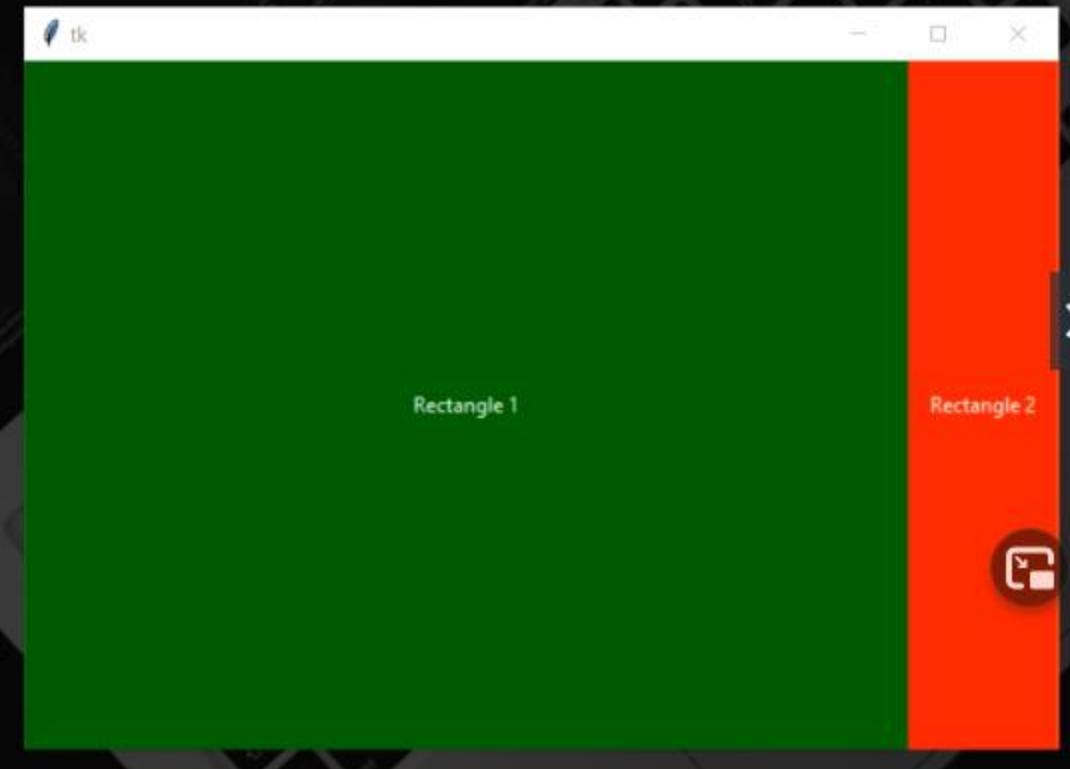
```
import tkinter as tk

root = tk.Tk()
root.geometry("600x400")

rectangle_1 = tk.Label(root, text="Rectangle 1", bg="green", fg="white")
rectangle_1.pack(side="left", ipadx=10, ipady=10, fill="both", expand=True)

rectangle_2 = tk.Label(root, text="Rectangle 2", bg="red", fg="white")
rectangle_2.pack(ipadx=10, ipady=10, fill="both", expand=True)

root.mainloop()
```



**side = left** was the first to be added to the window so **side = left** gets priority

# Packing components in tkinter

## Expansion priority

- ✓ This causes some very interesting things to happen when you add more widgets into the mix...
- ✓ For example, first `side="left"`, then `side="top"`, then another `side="left"` all with `expand=True`
- ✓ The `side="left"` ones get expansion priority because the first widget added with `expand=True` had that side value

# Packing components in tkinter

## Expansion priority

```
import tkinter as tk

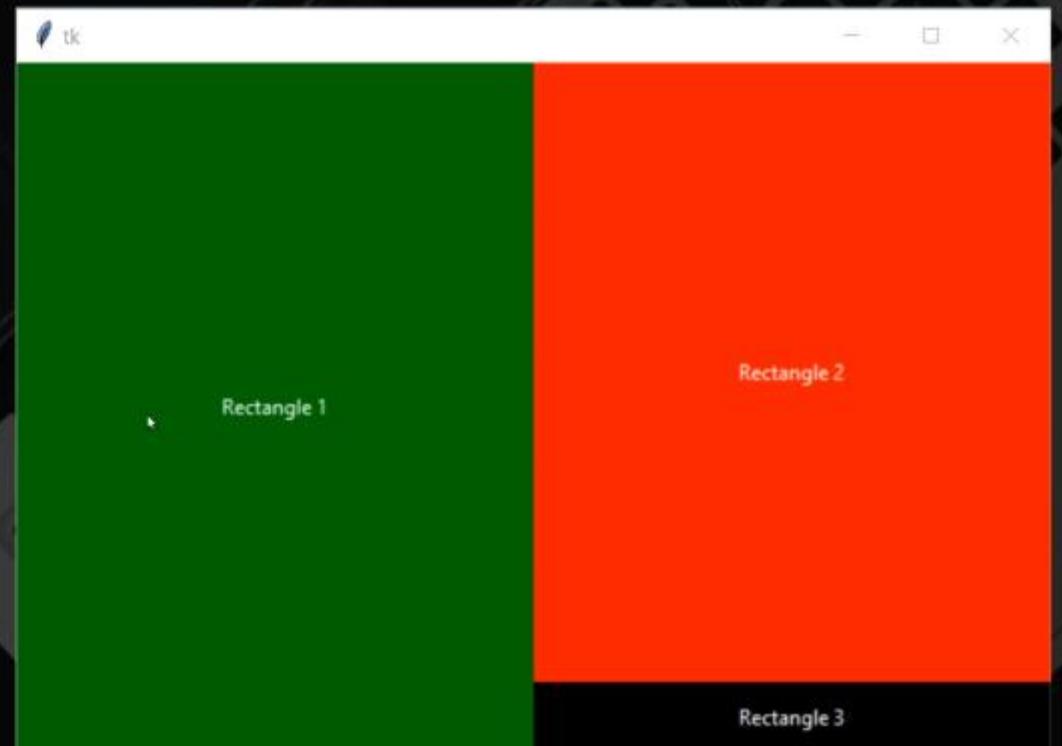
root = tk.Tk()
root.geometry("600x400")

rectangle_1 = tk.Label(root, text="Rectangle 1", bg="green", fg="white")
rectangle_1.pack(side="left", ipadx=10, ipady=10, fill="both", expand=True)

rectangle_2 = tk.Label(root, text="Rectangle 2", bg="red", fg="white")
rectangle_2.pack(side="top", ipadx=10, ipady=10, fill="both", expand=True)

rectangle_3 = tk.Label(root, text="Rectangle 3", bg="black", fg="white")
rectangle_3.pack(side="left", ipadx=10, ipady=10, fill="both", expand=True)

root.mainloop()
```

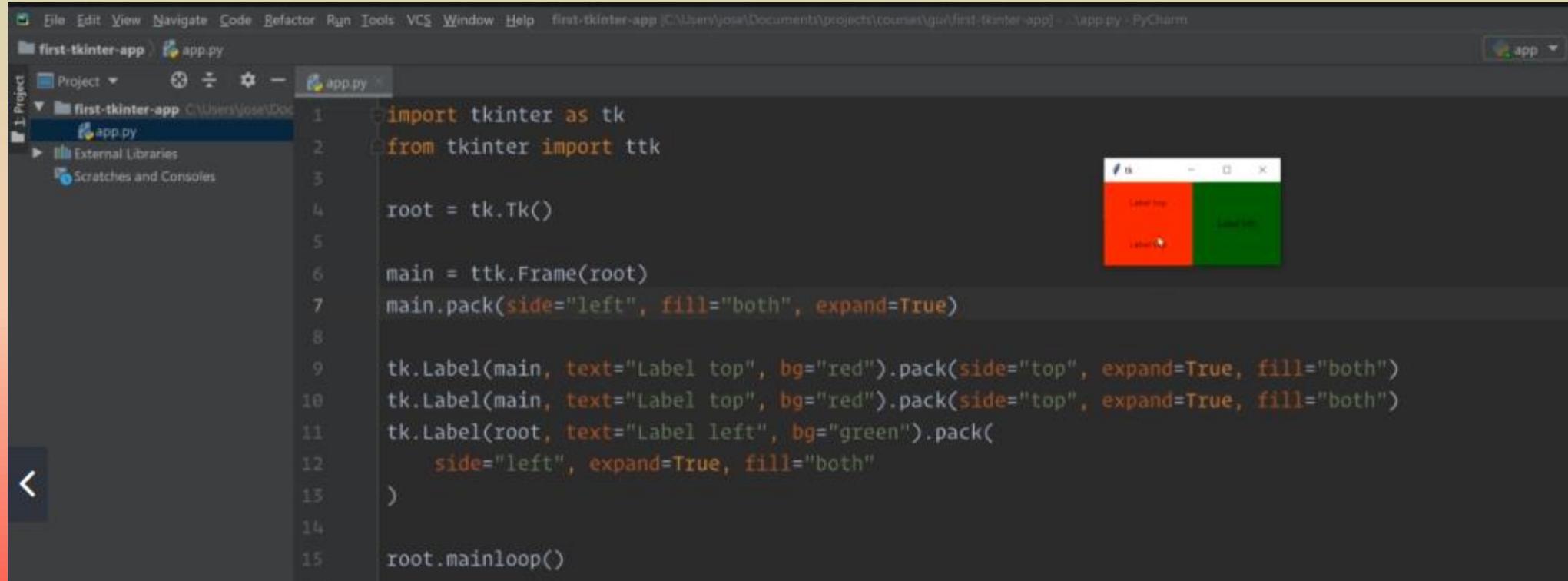


# Packing components in tkinter

## Pack is weird, but it's good

- ✓ It takes time to fully understand what's going to happen when you use pack
- ✓ But when you do, pack is really quick and easy to use
- ✓ For more complicated layouts, there are other algorithms that can be more suitable, such as grid
- ✓ Play around with pack and try things out, it'll make more sense the more you use it!

# Frame components in tkinter



The screenshot shows a PyCharm interface with a code editor and a preview window. The code editor displays Python code for creating a Tkinter application with frames and labels. The preview window shows the resulting application window with two red labels.

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help first-tkinter-app [C:\Users\jose\Documents\projects\courses\gui\first-tkinter-app]\app.py : PyCharm
first-tkinter-app app.py
Project External Libraries Scratches and Consoles
first-tkinter-app C:\Users\jose\Doc
app.py
app.py
1 import tkinter as tk
2 from tkinter import ttk
3
4 root = tk.Tk()
5
6 main = ttk.Frame(root)
7 main.pack(side="left", fill="both", expand=True)
8
9 tk.Label(main, text="Label top", bg="red").pack(side="top", expand=True, fill="both")
10 tk.Label(main, text="Label top", bg="red").pack(side="top", expand=True, fill="both")
11 tk.Label(root, text="Label left", bg="green").pack(
12     side="left", expand=True, fill="both"
13 )
14
15 root.mainloop()
```

The application window preview shows a window with three labels:

- A red label at the top left with the text "Label top".
- A red label at the top right with the text "Label top".
- A green label on the left side with the text "Label left".

# Frame components in tkinter

```
import tkinter as tk
from tkinter import ttk

def greet():
    # The get() method is used to fetch the value of a StringVar() instance.
    # If user_name is empty, print Hello, World!
    print(f"Hello, {user_name.get() or 'World'}!")

root = tk.Tk()
root.title("Greeter")

# Here we create instances of the StringVar() class, which is to track the content of widgets
user_name = tk.StringVar()

# We define two frames to keep the input on different lines. In the next version we will switch to grid() geometry.
# Padding accepts a tuple of up to four values. Clockwise like CSS.
input_frame = ttk.Frame(root, padding=(20, 10, 20, 0))
input_frame.pack(fill="both")

name_label = ttk.Label(input_frame, text="Name: ")
name_label.pack(side="left", padx=(0, 10)) ←
name_entry = ttk.Entry(input_frame, width=15, textvariable=user_name)
name_entry.pack(side="left")
name_entry.focus()

buttons = ttk.Frame(root, padding=(20, 10))
buttons.pack(fill="both")

greet_button = ttk.Button(buttons, text="Greet", command=greet)
greet_button.pack(side="left", fill="x", expand=True)
quit_button = ttk.Button(buttons, text="Quit", command=root.destroy)
quit_button.pack(side="right", fill="x", expand=True)

root.mainloop()
```

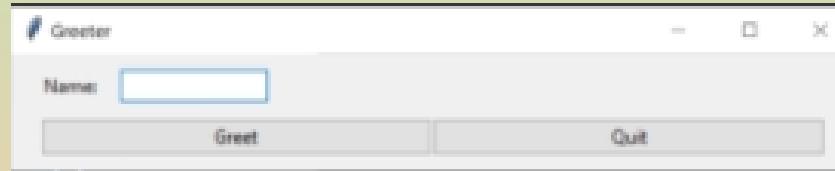


The padding apply clockwise order starting at the left.

When you have 2 values they applied left and right

# The Tkinter Grid Geometry manager

```
1 import tkinter as tk
2 from tkinter import ttk
3
4 root = tk.Tk()
5 root.title("Greeter")
6
7 user_name = tk.StringVar()
8
9 main = ttk.Frame(root, padding=(20, 10, 20, 0))
10 main.grid()
11
12 root.columnconfigure(0, weight=1)
13 root.rowconfigure(1, weight=1)
14
15 name_label = ttk.Label(main, text="Name:")
16 name_label.grid(row=0, column=0, padx=(0, 10))
17 name_entry = ttk.Entry(main, width=15, textvariable=user_name)
18 name_entry.grid(row=0, column=1)
19 name_entry.focus()
20
21 buttons = ttk.Frame(root, padding=(20, 10))
22 buttons.grid(row=1, column=0, sticky="EW")
23
24 buttons.columnconfigure(0, weight=1)
25 buttons.columnconfigure(1, weight=1)
26
27 greet_button = ttk.Button(buttons, text="Greet")
28 greet_button.grid(row=0, column=0, sticky="EW")
29 quit_button = ttk.Button(buttons, text="Quit", command=root.destroy)
30 quit_button.grid(row=0, column=1, sticky="EW")
31
32 root.mainloop()
```



# Label's in Tkinter

```
01_Labels.py
1 import tkinter as tk
2 from tkinter import ttk
3 from windows import set_dpi_awareness
4
5 set_dpi_awareness()
6
7
8 root = tk.Tk()
9 root.geometry("600x400")
10 root.resizable(False, False)
11 root.title("Widget Examples")
12
13 label = ttk.Label(root, text="Hello, world!", padding=20)
14 label.pack()
15
16 root.mainloop()
```



Let's change the font size which is too small and include an image

```
import tkinter as tk
from tkinter import ttk
from PIL import Image, ImageTk
from windows import set_dpi_awareness

set_dpi_awareness()

root = tk.Tk()
root.geometry("600x400")
root.resizable(False, False)
root.title("Widget Examples")

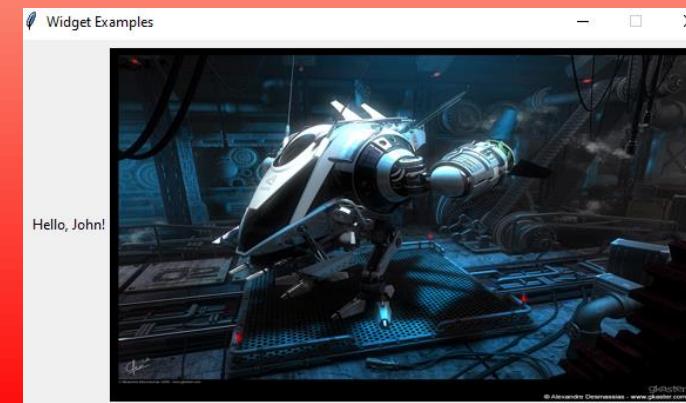
greeting = tk.StringVar()

image = Image.open("160.jpg").resize((500, 300)) ←
photo = ImageTk.PhotoImage(image)
label = ttk.Label(root, textvariable=greeting, image=photo, padding=5, compound="right")
label.pack()

greeting.set("Hello, John!")

root.mainloop()
```

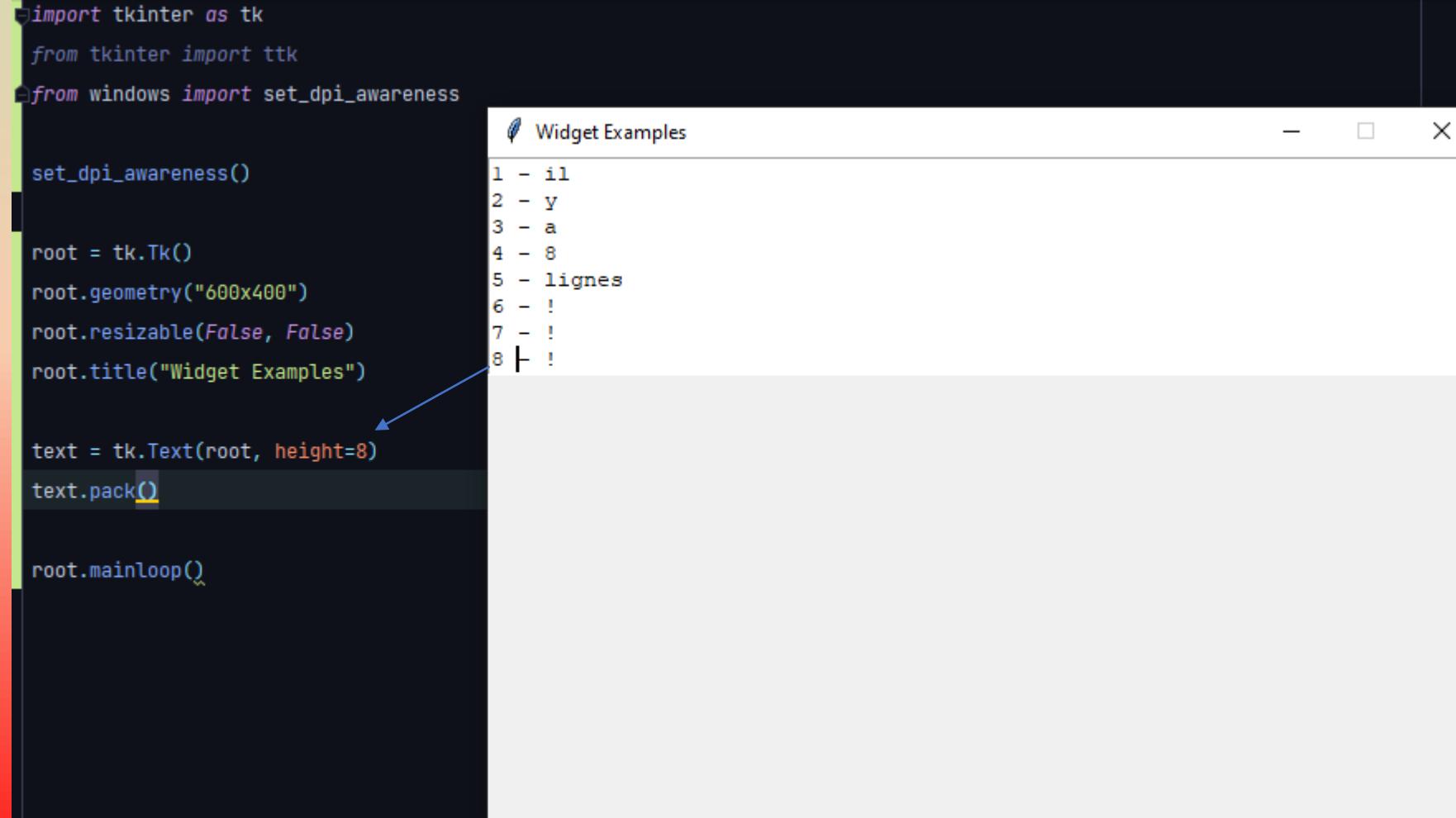
Resize the image to get it fit within the root window



# Label's in Tkinter

- ✓ Change a label's font  
with `font=("Font name", 20)`
- ✓ Add an image by first opening it:  
`image = Image.open("image.png")`  
Then creating an ImageTk:  
`ImageTk.PhotoImage(image)`  
And applying it to the label  
with `image=photo`
- ✓ Adjust text and image positioning  
with `compound="right"`

# The text widget in Tkinter



A screenshot of a Python code editor and a running Tkinter application. The code editor on the left shows a script with the following content:

```
import tkinter as tk
from tkinter import ttk
from windows import set_dpi_awareness

set_dpi_awareness()

root = tk.Tk()
root.geometry("600x400")
root.resizable(False, False)
root.title("Widget Examples")

text = tk.Text(root, height=8)
text.pack()

root.mainloop()
```

An arrow points from the line `text.pack()` in the code editor to the text widget in the application window. The application window is titled "Widget Examples" and contains the following text in a list-like format:

- 1 - il
- 2 - y
- 3 - a
- 4 - 8
- 5 - lignes
- 6 - !
- 7 - !
- 8 - !

# The text widget in Tkinter

```
import tkinter as tk
from tkinter import ttk
from windows import set_dpi_awareness

set_dpi_awareness()

root = tk.Tk()
root.geometry("600x400")
root.resizable(False, False)
root.title("Widget Examples")

text = tk.Text(root, height=8)
text.pack()

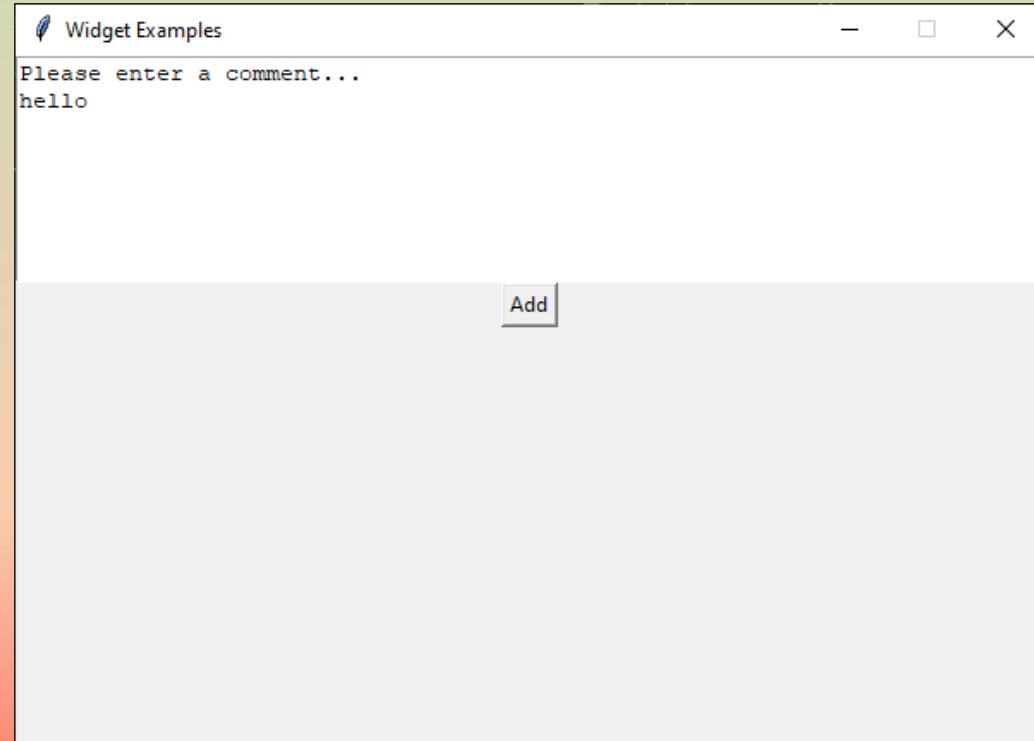
def Add():
    text_content1 = text.get("1.0", "end")
    print(text_content1)

button1 = tk.Button(root, text='Add', command=Add)
button1.pack()

text.insert("1.0", "Please enter a comment...") # the 1 refers to the line number
# the 0 refers to the character number
# We go to the first line and 0 character
text["state"] = "normal" # normal or disabled depending if you want the user to write inside the text field

text_content2 = text.get("1.0", "end") # First line character 0 till the end
print(text_content2)

root.mainloop()
```



# The text widget in Tkinter

- Create a text widget with `tk.Text`.  
The `height` property is the number of rows.
- The starting position is `"1.0"` and the ending position is `"end"`
- Insert with `.insert("1.0", "Text...")` and retrieve with `.get("1.0", "end")`
- `text["state"] = "disabled"` will prevent typing. Otherwise use `"normal"`

<https://blog.teclado.com/tkinter-placeholder-entry-field/>



How to add placeholders to Tkinter Entry fields.pdf

# Scrollbars in Tkinter

```
import tkinter as tk
from tkinter import ttk
from windows import set_dpi_awareness

set_dpi_awareness()

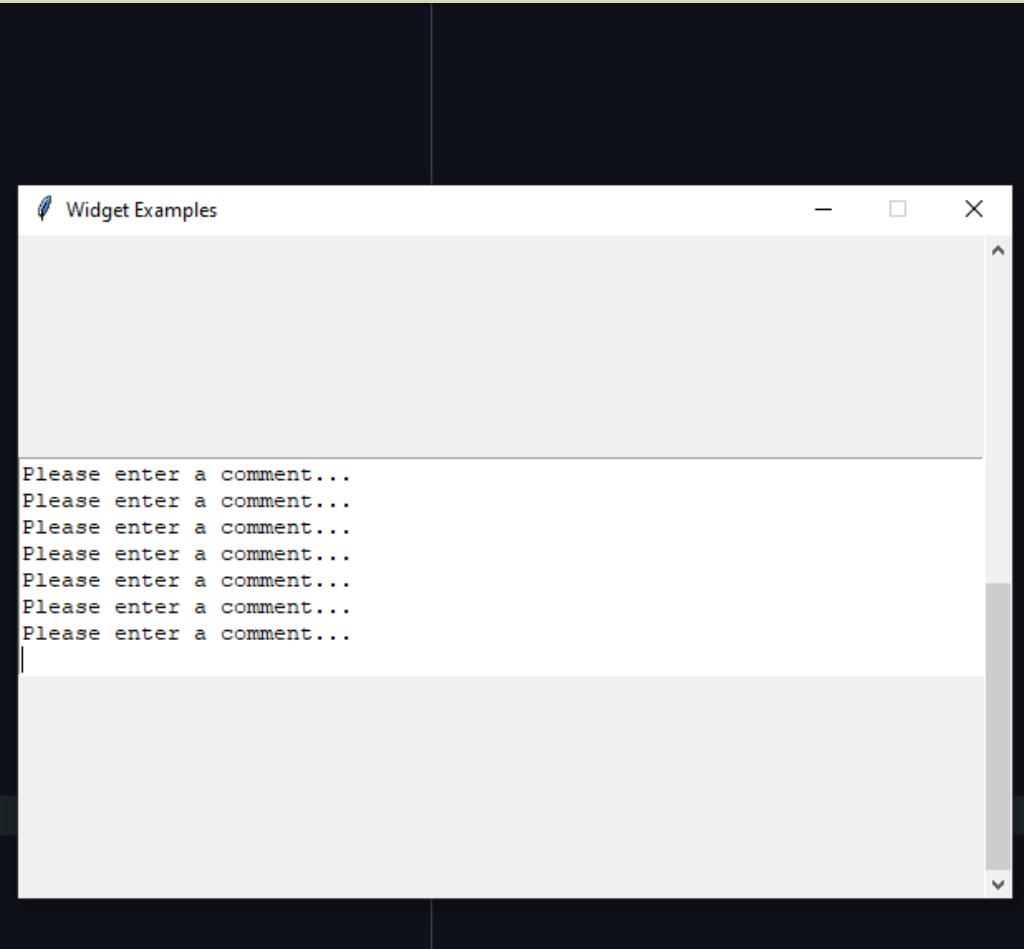
root = tk.Tk()
root.geometry("600x400")
root.resizable(False, False)
root.title("Widget Examples")

root.grid_columnconfigure(0, weight=1)
root.grid_rowconfigure(0, weight=1)

text = tk.Text(root, height=8)
text.grid(row=0, column=0, sticky="ew")
text.insert("1.0", "Please enter a comment...")

text_scroll= ttk.Scrollbar(root, orient="vertical", command=text.yview)
text_scroll.grid(row=0, column=1, sticky="ns")
text[ "yscrollcommand"] = text_scroll.set

root.mainloop()
```



# Scrollbars in Tkinter

- ✓ Create a scrollbar  
with `ttk.Scrollbar(orient, command)`
  
- ✓ `orient` can be "vertical" or "horizontal"
  
- ✓ `command` will usually be the `yview` or `xview`  
property of the widget you want to  
control with this scrollbar.
  
- ✓ Set the `yscrollcommand` property of the  
controlled widget so it links to the  
scrollbar.

# Separators in Tkinter

```
import tkinter as tk
from tkinter import ttk
from windows import set_dpi_awareness

set_dpi_awareness()

root = tk.Tk()
root.geometry("600x400")
root.resizable(False, False)
root.title("Widget Examples")

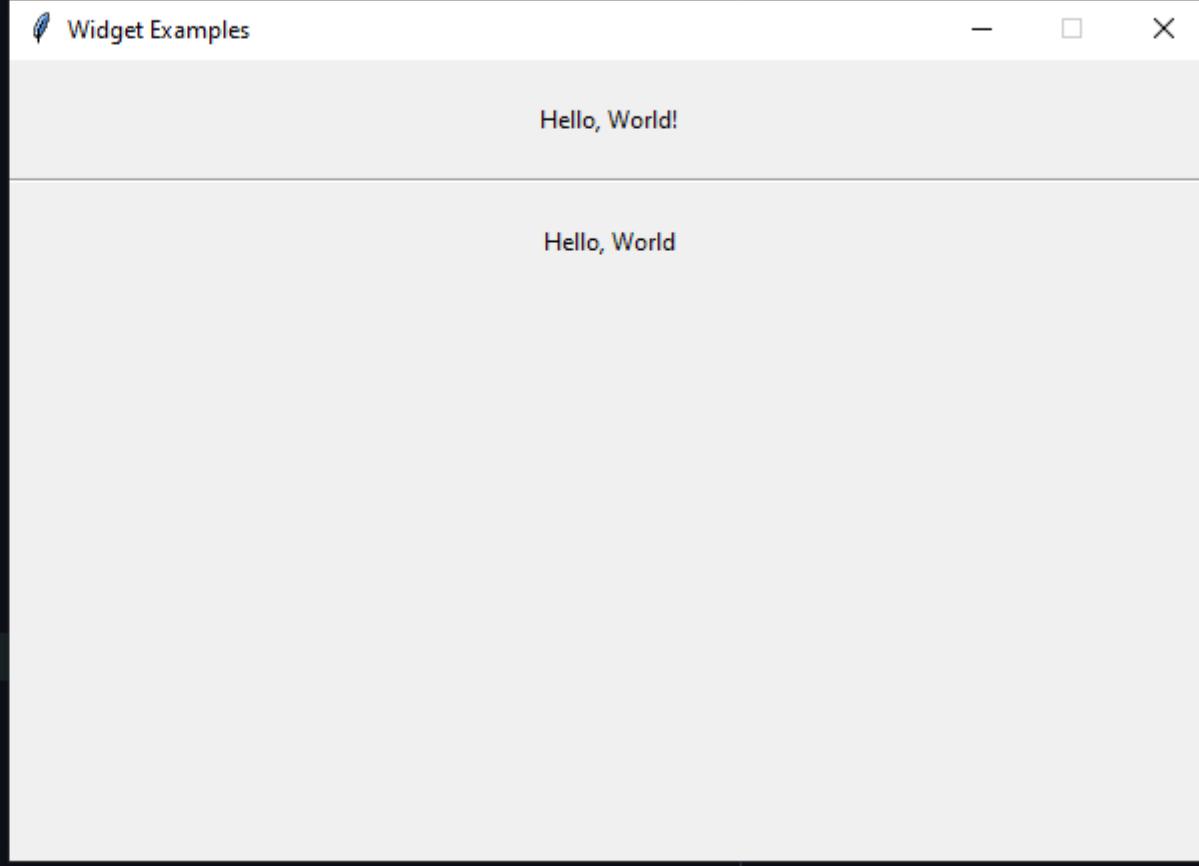
root.grid_columnconfigure(0, weight=1)
root.grid_rowconfigure(0, weight=1)

ttk.Label(root, text="Hello, World!", padding=20).pack()

ttk.Separator(root, orient="horizontal").pack(fill="x")

ttk.Label(root, text="Hello, World", padding=20).pack()

root.mainloop()
```



# Separators in Tkinter

- ✓ Create a separator  
with `ttk.Separator(orient)`
- ✓ `orient` can be `"vertical"` or `"horizontal"`
- ✓ By default, separators are 1px in size.  
Remember to set the `fill` or `sticky`  
property to adjust their size

# Check Buttons in Tkinter

```
from windows import set_dpi_awareness

set_dpi_awareness()

root = tk.Tk()
root.geometry("600x400")
root.resizable(False, False)
root.title("Widget Examples")

selected_option = tk.StringVar()

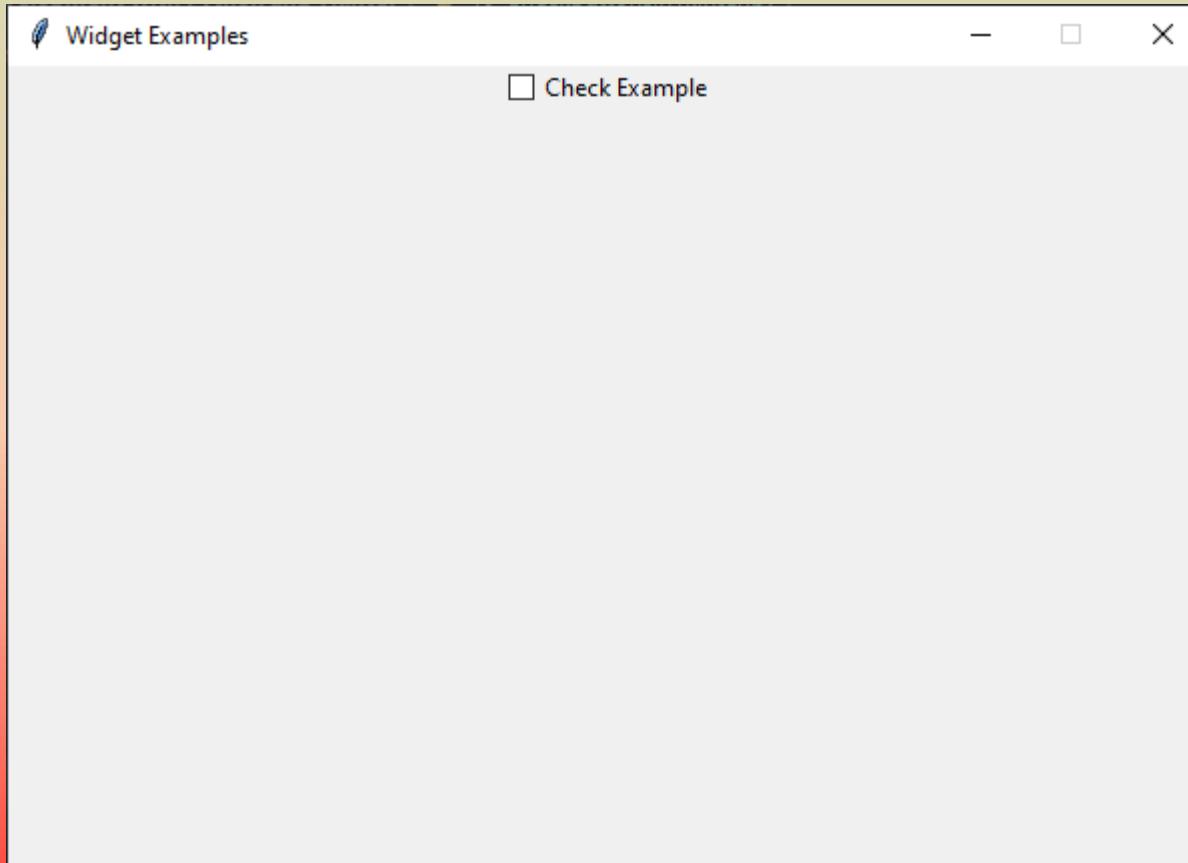
def print_current_option():
    print(selected_option.get())

check = ttk.Checkbutton(
    root,
    text="Check Example",
    variable=selected_option,
    command=print_current_option,
    onvalue="On",
    offvalue="Off"
)

check.pack()

# check_button = ttk.Checkbutton(root, text="Check me!")
# check_button.pack()
# check_button["state"] = "enabled" # "disabled"

root.mainloop()
```



# Check Buttons in Tkinter

## Widget Overview: Check buttons

- ✓ Create a check button with `ttk.Checkbutton(text, variable)`
- ✓ `variable` should be a `tk.StringVar()`
- ✓ Can attach a `command`, a function that runs when the button is checked or unchecked.
- ✓ `onvalue` and `offvalue` determine what value the `variable` will take

# Radio Buttons in Tkinter

```
import tkinter as tk
from tkinter import messagebox
from tkinter import ttk
from windows import set_dpi_awareness

set_dpi_awareness()

root = tk.Tk()
root.geometry("600x400")
root.resizable(False, False)
root.title("Widget Examples")

storage_variable = tk.StringVar()

def viewSelected():
    choice = storage_variable.get()
    if choice == "First option":
        output = "Science"
    elif choice == "Second option":
        output = "Commerce"
    elif choice == "Third option":
        output = "Art"
    else:
        output = "Invalid selection"

    return messagebox.showinfo('PythonGuides', f'You Selected {output}.')

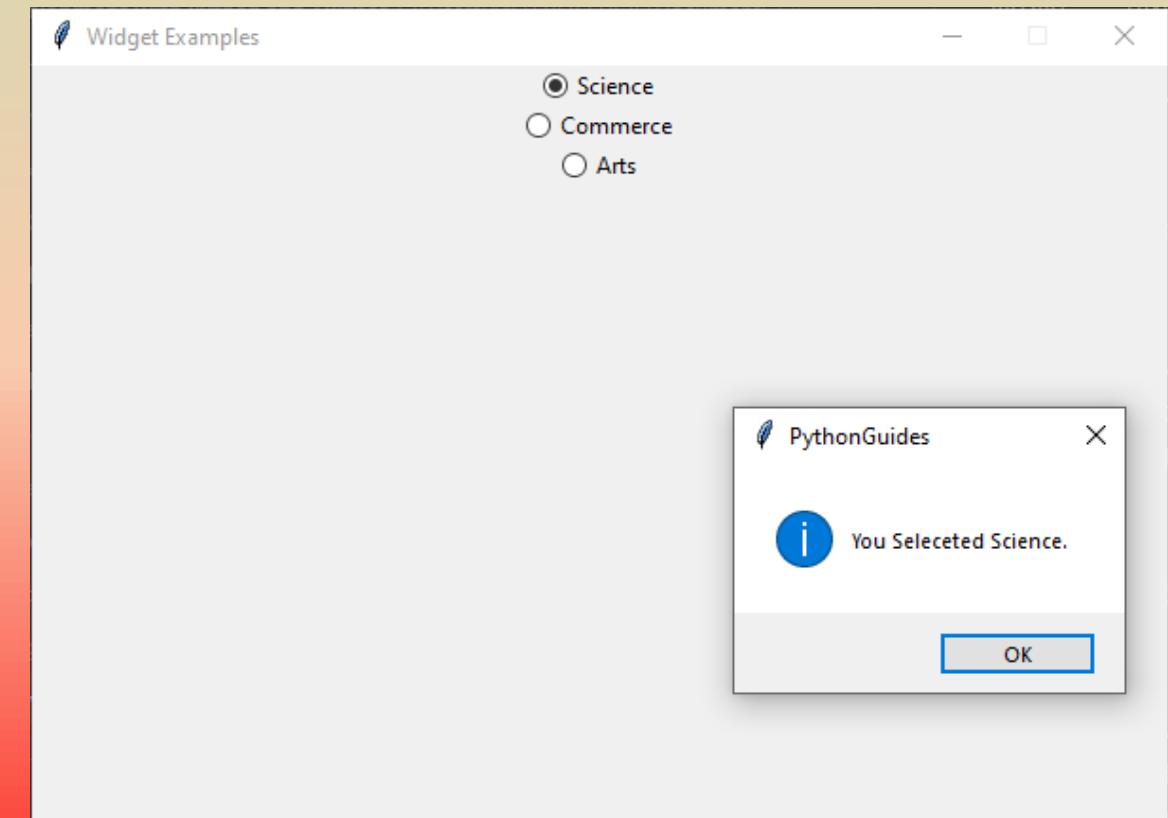
option_one = ttk.Radiobutton(
    root,
    text="Science",
    variable=storage_variable,
    value="First option",
    command=viewSelected
)

option_two = ttk.Radiobutton(
    root,
    text="Commerce",
    variable=storage_variable,
    value="Second option",
    command=viewSelected
)

option_three = ttk.Radiobutton(
    root,
    text="Arts",
    variable=storage_variable,
    value="Third option",
    command=viewSelected
)

option_one.pack()
option_two.pack()
option_three.pack()

root.mainloop()
```



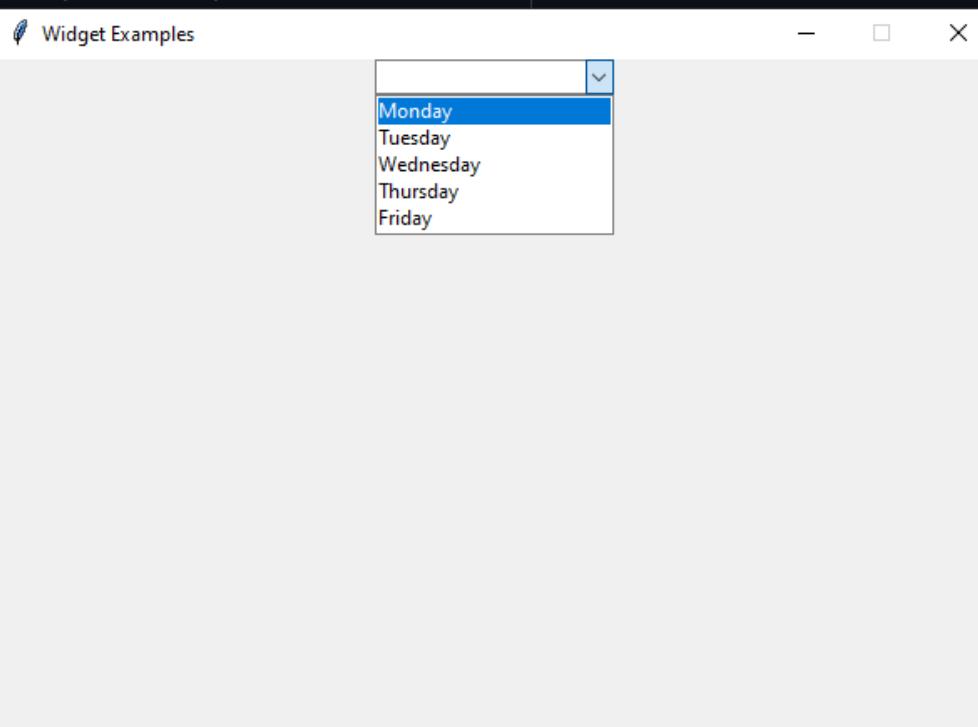
# Radio Buttons in Tkinter

## Widget Overview: Radio buttons

- ✓ Create a radio button with `ttk.Radiobutton(text, variable)`
- ✓ `variable` should be a `tk.StringVar()`
- ✓ If multiple radio buttons have the same `variable`, they become a group. Only one can be checked in a group at a time.
- ✓ `value` determines what value the `variable` will take

# Comboboxes in Tkinter

```
 5 set_dpi_awareness()
 6
 7 root = tk.Tk()
 8 root.geometry("600x400")
 9 root.resizable(False, False)
10 root.title("Widget Examples")
11
12 selected_weekday = tk.StringVar()
13 weekday = ttk.Combobox(root, textvariable=selected_weekday)
14 weekday["values"] = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
15 weekday["state"] = "readonly" # to avoid the user write in the weekday combobox, the counterpart is "normal"
16 weekday.pack()
17
18 def handle_selection(event):
19     print("Today is", selected_weekday.get())
20     print("But we're gonna change it to Friday")
21     selected_weekday.set("Friday")
22     print(weekday.current())
23
24
25 weekday.bind("<<ComboboxSelected>>", handle_selection)
26
27 root.mainloop()
28
29 print(selected_weekday)
```



# Comboboxes in Tkinter

## Widget Overview: Combo boxes

- ✓ Create a radio button with `ttk.Radiobutton(textvariable, values)`
- ✓ `values` should be a tuple, and as with all properties it can be defined in another line
- ✓ Set the `state` property to `"readonly"` so users can't type custom values
- ✓ To run a function when the value changes, bind a function to `"<<ComboboxSelected>>"`

# Listboxes in Tkinter

```
root.title("Widget Examples")

programming_languages = ("C", "Go", "JavaScript", "Perl", "Python", "Rust")

langs = tk.StringVar(value=programming_languages)
langs_select = tk.Listbox(root, listvariable=langs, height=6) # Listbox is part of tkinter is not a themable tkinter
# widget so we have to use tk and not ttk

# To activate multiple lines choice the two lines below are equivalent
# langs_select = tk.Listbox(root, listvariable=langs, height=6, selectmode="extended")
langs_select["selectmode"] = "extended" # allow the user to select multiple line. # "browse" is the counterpart
langs_select.pack()

def handle_selection_change(event):
    selected_indices = langs_select.curselection()
    for i in selected_indices:
        print(langs_select.get(i))

langs_select.bind("<<ListboxSelect>>", handle_selection_change)

root.mainloop()
```



# Listboxes in Tkinter

## Widget Overview: List boxes

- ✓ Create a listbox  
with `tk.Listbox(listvariable, height)`
- ✓ `listvariable` should be  
a `tk.StringVar(value=your_tuple)`
- ✓ Set the `selectmode` property to "extended"  
to allow multiple selection. Otherwise  
use "browse"
- ✓ To run a function when the  
value changes, bind a function  
to "`<<ListboxSelect>>`"

# Spinboxes in Tkinter

```
import tkinter as tk
from tkinter import ttk
from windows import set_dpi_awareness

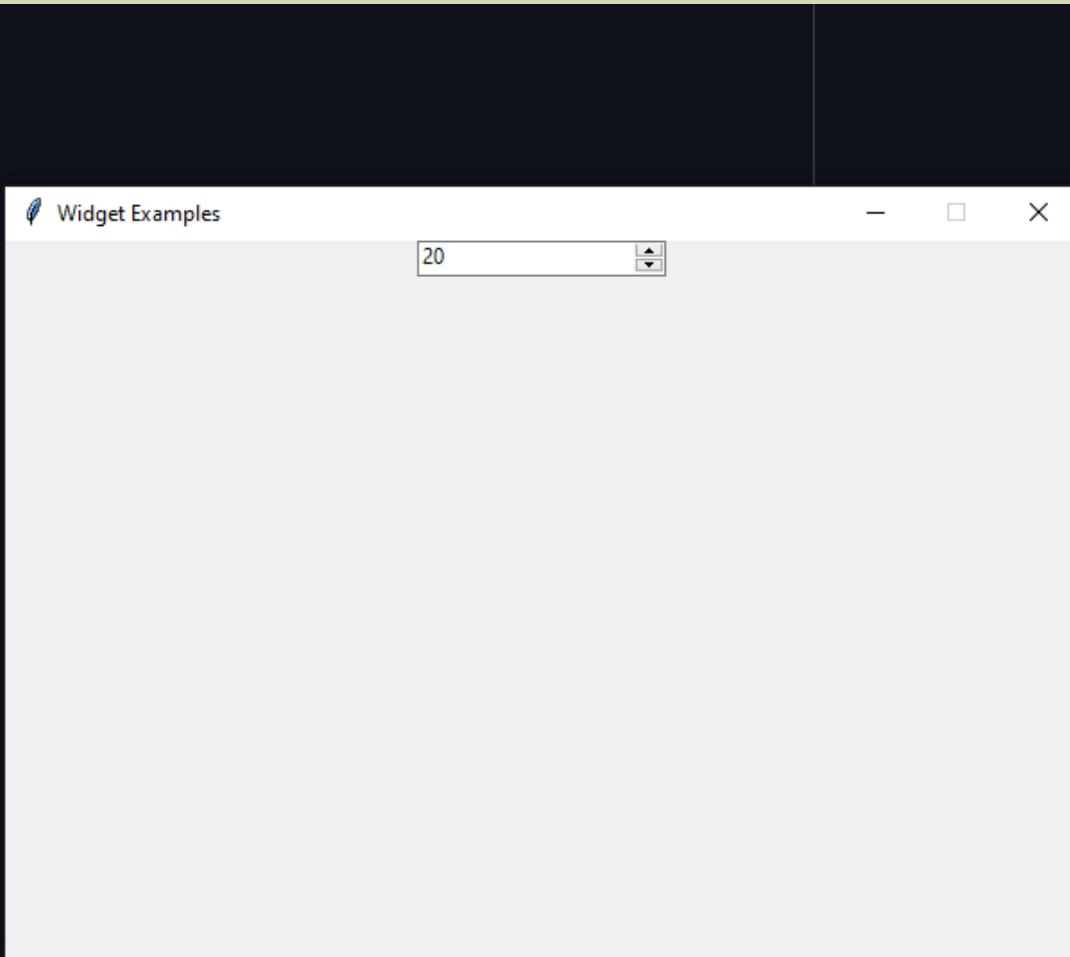
set_dpi_awareness()

root = tk.Tk()
root.geometry("600x400")
root.resizable(False, False)
root.title("Widget Examples")

initial_value = tk.StringVar(value=20)
spin_box = ttk.Spinbox(
    root,
    # from_=0,
    # to=30,
    # instead of from and to, you can pass values
    values=(5, 10, 15, 20, 25, 30),
    textvariable=initial_value,
    wrap=False
)
spin_box.pack()

print(spin_box.get())

root.mainloop()
```



# Spinboxes in Tkinter

## Widget Overview: Spin boxes

- ✓ Create a spin box with `tk.Spinbox(from_, to, textvariable)`
- ✓ `textvariable` should be a `tk.StringVar()`
- ✓ Instead of `from_` and `to`, set the `values` to a tuple of numbers for discrete steps
- ✓ Set `wrap=True` for overflow after the maximum value to go back to the minimum

# Scales in Tkinter

```
root = tk.Tk()
root.geometry("600x400")
root.resizable(False, False)
root.title("Widget Examples")

def handle_scale_change(event):
    print(scale.get())

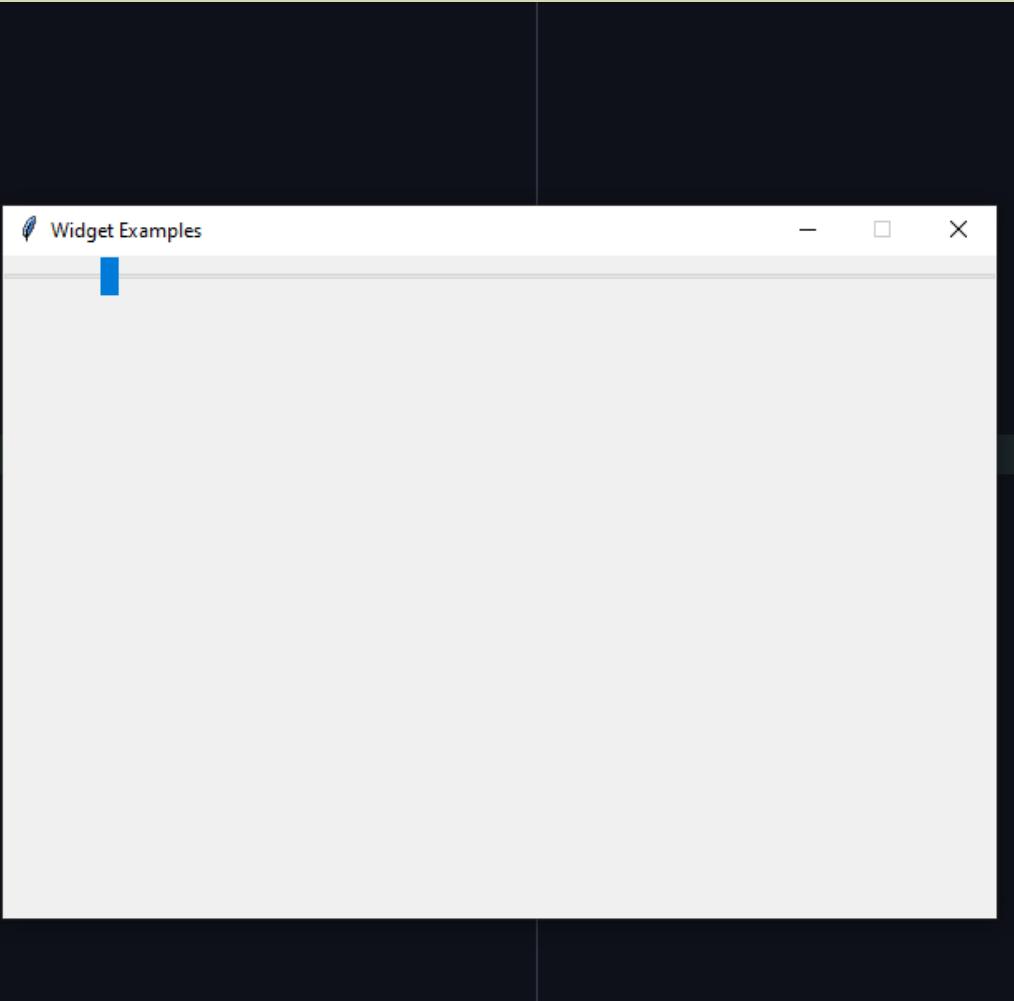
current_value = tk.DoubleVar()

scale = ttk.Scale(root,
                  orient="horizontal",
                  from_=0, to=10,
                  command=handle_scale_change,
                  variable=current_value)
scale.pack(fill="x")

scale["state"] = "normal" # "disabled" is the counterpart

root.mainloop()

print(current_value.get())
```



Run:	15-ScalesInTkinter
▶	0.8191126279863481
▼	0.870307167235495
⟳	0.9215017064846416
➕	0.9556313993174061
🖨️	0.9726962457337884
✖	0.9897610921501707
	0.9897610921501707

# Scales in Tkinter

## Widget Overview: Scales

- ✓ Create a scale with `ttk.Scale(orient, from_, to)`
- ✓ `scale.get()` can be used to retrieve the current value
- ✓ A `command` can be set to run when the value changes. However, this can cause performance problems.

# Object-Oriented Programming with Tkinter

```
import tkinter as tk
from tkinter import ttk
from windows import set_dpi_awareness

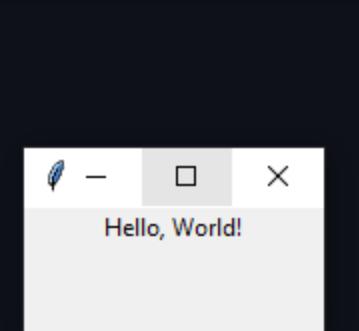
set_dpi_awareness()

class HelloWorld(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Hello, World!")

        ttk.Label(self, text="Hello, World!").pack()

root = HelloWorld()
root.mainloop()
```



# An Object-Oriented Frame

```
import tkinter as tk
```

```
from tkinter import ttk
```

```
from windows import set_dpi_awareness
```

```
set_dpi_awareness()
```

```
class UserInputFrame(ttk.Frame):
```

```
    def __init__(self, container):  
        super().__init__(container)
```

```
        self.user_input = tk.StringVar()
```

```
        label = ttk.Label(self, text="Enter your name:")
```

```
        entry = ttk.Entry(self, textvariable=self.user_input)
```

```
        button = ttk.Button(self, command=self.greet, text="Valider")
```

```
        label.pack(side="left")
```

```
        entry.pack(side="left")
```

```
        button.pack(side="left")
```

```
    def greet(self):
```

```
        print(f"Hello, {self.user_input.get()}!")
```

```
root = tk.Tk()
```

```
frame = UserInputFrame(root)
```

```
frame.pack()
```



# A Full Tkinter Object-Oriented App

```
import tkinter as tk
from tkinter import ttk
from windows import set_dpi_awareness

set_dpi_awareness()

class HelloWorld(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Hello world!")

        UserInputFrame(self).pack()

class UserInputFrame(ttk.Frame):
    def __init__(self, container):
        super().__init__(container)

        self.user_input = tk.StringVar()

        label = ttk.Label(self, text="Enter your name:")
        entry = ttk.Entry(self, textvariable=self.user_input)
        button = ttk.Button(self, command=self.greet, text="Valider")

        label.pack(side="left")
        entry.pack(side="left")
        button.pack(side="left")

    def greet(self):
        print(f"Hello, {self.user_input.get()}!")

root = HelloWorld()
root.mainloop()
```



# Distance Converter Object-Oriented Programming

```
import tkinter as tk
from tkinter import ttk, font
from windows import set_dpi_awareness

set_dpi_awareness()

class DistanceConverter(tk.Tk):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.title("Distance Converter")

        frame = MetresToFeet(self, padding=(60, 30))
        frame.grid()

        self.bind("<Return>", frame.calculate_feet)
        self.bind("<KP_Enter>", frame.calculate_feet)

    class MetresToFeet(ttk.Frame):
        def __init__(self, container, **kwargs):
            super().__init__(container, **kwargs)

            self.metres_value = tk.StringVar()
            self.feet_value = tk.StringVar(value="Feet shown here.")

            metres_label = ttk.Label(self, text="Metres:")
            metres_input = ttk.Entry(self, width=10, textvariable=self.metres_value, font=("Segoe UI", 15))
            feet_label = ttk.Label(self, text="Feet:")
            feet_display = ttk.Label(self, textvariable=self.feet_value)
            calc_button = ttk.Button(self, text="Calculate", command=self.calculate_feet)

            metres_label.grid(column=0, row=0, stick="W")
            metres_input.grid(column=1, row=0, stick="EW")
            metres_input.focus()

            feet_label.grid(column=0, row=1, stick="W")
            feet_display.grid(column=1, row=1, stick="EW")

            calc_button.grid(column=0, row=2, columnspan=2, stick="EW")

            for child in self.winfo_children():
                child.grid_configure(padx=15, pady=15)

        def calculate_feet(self, *args):
            try:
                metres = float(self.metres_value.get())
                feet = metres * 3.28084
                self.feet_value.set(f"{feet:.3f}")
            except ValueError:
                print("saisir un entier")

root = DistanceConverter()

font.nametofont("TkDefaultFont").configure(size=15)

root.columnconfigure(0, weight=1)
root.mainloop()
```

```
feet_label.grid(column=0, row=1, stick="W")
feet_display.grid(column=1, row=1, stick="EW")

calc_button.grid(column=0, row=2, columnspan=2, stick="EW")

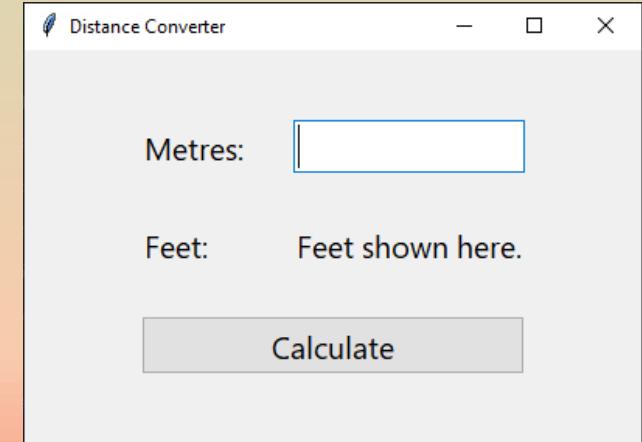
for child in self.winfo_children():
    child.grid_configure(padx=15, pady=15)

def calculate_feet(self, *args):
    try:
        metres = float(self.metres_value.get())
        feet = metres * 3.28084
        self.feet_value.set(f"{feet:.3f}")
    except ValueError:
        print("saisir un entier")

root = DistanceConverter()

font.nametofont("TkDefaultFont").configure(size=15)

root.columnconfigure(0, weight=1)
root.mainloop()
```



# Creating a FeetToMeter frame

```
import tkinter as tk
from tkinter import ttk, font
from windows import set_dpi_awareness

set_dpi_awareness()

class DistanceConverter(tk.Tk):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.title("Distance Converter")

        container = ttk.Frame(self)
        container.grid(padx=60, pady=30, sticky="EW")

        # frame = MetresToFeet(container)
        frame = FeetToMetres(container)
        frame.grid(column=0, row=0, sticky="NSEW")

        self.bind("<Return>", frame.calculate)
        self.bind("<KP_Enter>", frame.calculate)

    class MetresToFeet(ttk.Frame):
        def __init__(self, container, **kwargs):
            super().__init__(container, **kwargs)

            self.metres_value = tk.StringVar()
            self.feet_value = tk.StringVar(value="Feet shown here.")

            metres_label = ttk.Label(self, text="Metres:")
            metres_input = ttk.Entry(self, width=10, textvariable=self.metres_value, font=("Segoe UI", 15))
            feet_label = ttk.Label(self, text="Feet:")
            feet_display = ttk.Label(self, textvariable=self.feet_value)
            calc_button = ttk.Button(self, text="Calculate", command=self.calculate)

            metres_label.grid(column=0, row=0, stick="W")
            metres_input.grid(column=1, row=0, stick="EW")
            metres_input.focus()

            feet_label.grid(column=0, row=1, stick="W")
            feet_display.grid(column=1, row=1, stick="EW")

            calc_button.grid(column=0, row=2, columnspan=2, stick="EW")

            for child in self.winfo_children():
                child.grid_configure(padx=15, pady=15)

        def calculate(self, *args):
            try:
                metres = float(self.metres_value.get())
                feet = metres * 3.28084
                self.feet_value.set(f"{feet:.3f}")
            except ValueError:
                print("saisir un entier")

    class FeetToMetres(ttk.Frame):
        def __init__(self, container, **kwargs):
            super().__init__(container, **kwargs)

            self.metres_value = tk.StringVar()
            self.feet_value = tk.StringVar(value="Feet shown here.")

            feet_label = ttk.Label(self, text="Feet:")
            feet_input = ttk.Entry(self, width=10, textvariable=self.feet_value, font=("Segoe UI", 15))
            metres_label = ttk.Label(self, text="Metres:")
            metres_display = ttk.Label(self, textvariable=self.metres_value)
            calc_button = ttk.Button(self, text="Calculate", command=self.calculate)

            feet_label.grid(column=0, row=0, stick="W")
            metres_display.grid(column=1, row=1, stick="EW")

            calc_button.grid(column=0, row=2, columnspan=2, stick="EW")

            for child in self.winfo_children():
                child.grid_configure(padx=15, pady=15)

        def calculate(self, *args):
            try:
                feet = float(self.feet_value.get())
                metres = feet / 3.28084
                self.metres_value.set(f"{metres:.3f}")
            except ValueError:
                print("saisir un entier")

    def calculate(self, *args):
        try:
            feet = float(self.feet_value.get())
            metres = feet / 3.28084
            self.metres_value.set(f"{metres:.3f}")
        except ValueError:
            print("saisir un entier")

root = DistanceConverter()

font.nametofont("TkDefaultFont").configure(size=15)

root.columnconfigure(0, weight=1)
root.mainloop()
```

# How to switch between frames in a Tkinter app 1/2

```
import tkinter as tk
from tkinter import ttk, font
from windows import set_dpi_awareness

set_dpi_awareness()

class DistanceConverter(tk.Tk):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.title("Distance Converter")
        self.frames = dict()

        container = ttk.Frame(self)
        container.grid(padx=60, pady=30, sticky="EW")

        # # frame = FeetToMetres(container)
        # feet_to_metres = FeetToMetres(container, self)
        # feet_to_metres.grid(column=0, row=0, sticky="NSEW")
        #
        # # frame = MetresToFeet(container)
        # metres_to_feet = MetresToFeet(container, self)
        # metres_to_feet.grid(column=0, row=0, sticky="NSEW")
        #
        # self.frames[FeetToMetres] = feet_to_metres
        # self.frames[MetresToFeet] = metres_to_feet

        # the above commented section of code is replaced by the for loop below. Shorter and nicer

        for FrameClass in (MetresToFeet, FeetToMetres):
            frame = FrameClass(container, self)
            self.frames[FrameClass] = frame
            frame.grid(column=0, row=0, sticky="NSEW")

        self.show_frame(MetresToFeet)

    def show_frame(self, container):
        frame = self.frames[container]
```

```
        self.bind("<Return>", frame.calculate)
        self.bind("<KP_Enter>", frame.calculate)
        frame.tkraise()

class MetresToFeet(ttk.Frame):
    def __init__(self, container, controller, **kwargs):
        super().__init__(container, **kwargs)

        self.metres_value = tk.StringVar(value="Metre shown here.")
        self.feet_value = tk.StringVar()

        metres_label = ttk.Label(self, text="Metres:")
        metres_input = ttk.Entry(self, width=15, textvariable=self.metres_value, font=("Segoe UI", 8))
        feet_label = ttk.Label(self, text="Feet:")
        feet_display = ttk.Label(self, textvariable=self.feet_value)
        calc_button = ttk.Button(self, text="Calculate", command=self.calculate)
        switch_page_button = ttk.Button(
            self,
            text="Switch to feet conversion",
            command=lambda: controller.show_frame(FeetToMetres)
        )

        metres_label.grid(column=0, row=0, stick="W")
        metres_input.grid(column=1, row=0, stick="EW")
        metres_input.focus()

        feet_label.grid(column=0, row=1, stick="W")
        feet_display.grid(column=1, row=1, stick="EW")

        calc_button.grid(column=0, row=2, columnspan=2, stick="EW")
        switch_page_button.grid(column=0, row=3, columnspan=2, sticky="EW")

        for child in self.winfo_children():
            child.grid_configure(padx=15, pady=15)

    def calculate(self, *args):
        try:
            metres = float(self.metres_value.get())
```

# How to switch between frames in a Tkinter app 2/2

```
feet = metres * 3.28084
self.feet_value.set(f"{feet:.3f}")

except ValueError:
    print("saisir un entier")

class FeetToMetres(ttk.Frame):
    def __init__(self, container, controller, **kwargs):
        super().__init__(container, **kwargs)

        self.metres_value = tk.StringVar()
        self.feet_value = tk.StringVar(value="Feet shown here.")

        feet_label = ttk.Label(self, text="Feet:")
        feet_input = ttk.Entry(self, width=15, textvariable=self.feet_value, font=("Segoe UI", 8))
        metres_label = ttk.Label(self, text="Metres:")
        metres_display = ttk.Label(self, textvariable=self.metres_value)
        calc_button = ttk.Button(self, text="Calculate", command=self.calculate)
        switch_page_button = ttk.Button(
            self,
            text="Switch to metres conversion",
            command=lambda: controller.show_frame(MetresToFeet)
        )

        feet_label.grid(column=0, row=0, stick="W")
        feet_input.grid(column=1, row=0, stick="EW")
        feet_input.focus()

        metres_label.grid(column=0, row=1, stick="W")
        metres_display.grid(column=1, row=1, stick="EW")

        calc_button.grid(column=0, row=2, columnspan=2, stick="EW")
        switch_page_button.grid(column=0, row=3, columnspan=2, sticky="EW")

        for child in self.winfo_children():
            child.grid_configure(padx=15, pady=15)
```

```
def calculate(self, *args):
    try:
        feet = float(self.feet_value.get())
        metres = feet / 3.28084
        self.metres_value.set(f"{metres:.3f}")
    except ValueError:
        print("saisir un entier")

root = DistanceConverter()

font.nametofont("TkDefaultFont").configure(size=15)

root.columnconfigure(0, weight=1)
root.mainloop()
```

# Tkinter themes, and how to change theme

## ttk Style themes

We can access the style database by creating an instance of `ttk.Style()`:

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
style = ttk.Style(root)

...
```

This allows us to access and customize styles for the Tk application passed, in this case `root`.

## Themes

Every Operating System comes with themes. A theme is a collection of styles for Tkinter widgets, so that if you enable one theme, those styles will be applied.

You can see which themes are available like so:

```
style = ttk.Style(root)
print(style.theme_names())
```

And you can change the theme like so:

```
style = ttk.Style(root)
style.theme_use("classic") # Use a theme available in your system
```

```
import tkinter as tk
from tkinter import ttk, font
from windows import set_dpi_awareness

set_dpi_awareness()

def greet(*args):
    print(f"Hello, {user_name.get()}")

root = tk.Tk()
root.resizable(False, False)
root.title("Greeter")

style = ttk.Style(root)
print(style.theme_names())
print(style.theme_use("clam"))

main = ttk.Frame(root, padding=(40, 20))
main.grid()

user_name = tk.StringVar()

name_label = ttk.Label(main, text="Name:")
name_label.grid(column=0, row=0, padx=(0, 10))
name_entry = ttk.Entry(main, width=15, textvariable=user_name)
name_entry.grid(column=1, row=0, padx=0)
name_entry.focus()

greet_button = ttk.Button(main, text="Greet", command=greet)
greet_button.grid(column=2, row=0, stick="EW", padx=10)

root.mainloop()
```

# Finding a Tkinter widget's style class

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
style = ttk.Style(root)

name = ttk.Label(root, text="Hello, world!")
entry = ttk.Entry(root, width=15)
name.pack()

print(name.winfo_class())

root.mainloop()
```

# How to change and configure a Tkinter style

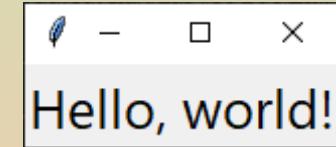
```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
style = ttk.Style(root)

# name = ttk.Label(root, text="Hello, world!", style="TLabel")
# or
name = ttk.Label(root, text="Hello, world!")
ttk.Label(root, text="Hello, world!")
entry = ttk.Entry(root, width=15)
entry["style"] = "CustomEntryStyle" # Either the style is included in ttk.Label like above (commented line)
# or we define the style on a sparated line like here
name.pack()

style.configure("TLabel", font=("Segoe UI", 20))

root.mainloop()
```



# Find out what properties you can change in a Tkinter style

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
style = ttk.Style(root)

name = ttk.Label(root, text="Hello, world!")
name.pack()

print(style.layout("TLabel"))

print(style.element_options("Label.border"))
print(style.element_options("Label.padding"))
print(style.element_options("Label.label"))

print(style.lookup("TLabel", "font")) # retrieve value of style properties
print(style.lookup("TLabel", "foreground")) # retrieve value of style properties
print(style.lookup("TLabel", "compound")) # no value returned because it is not set

style.theme_use("clam") # we are setting a theme and look at the setting again

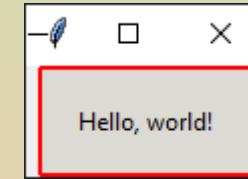
print(style.layout("TLabel"))

print(style.element_options("Label.border"))
print(style.element_options("Label.padding"))
print(style.element_options("Label.label"))

print(style.lookup("TLabel", "font")) # retrieve value of style properties
print(style.lookup("TLabel", "foreground")) # retrieve value of style properties
print(style.lookup("TLabel", "compound")) # no value returned because it is not set

# Let's create a border with the clam them (you can't do it with windows style)
style.configure("TLabel", bordercolor="#f00")
style.configure("TLabel", borderwidth=20) # in clam theme borderwidth can't exceed 2 pixels
style.configure("TLabel", relief="solid")

root.mainloop()
```



# How to create new inherited styles in Tkinter

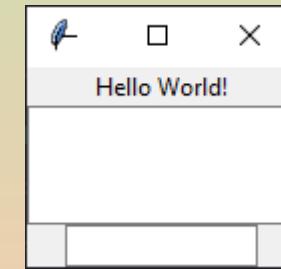
```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
style = ttk.Style(root)

style.configure("CustomEntryStyle.TEntry", padding=20)

name = ttk.Label(root, text="Hello World!")
entry = ttk.Entry(root, width=15)
entry["style"] = "CustomEntryStyle.TEntry"
name.pack()
entry.pack()

entry2 = ttk.Entry(root, width=15)
entry2.pack()
root.mainloop()
```



# How to configure state-specific options in a Tkinter style

```
import tkinter as tk
from tkinter import ttk

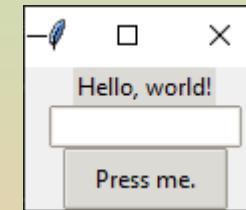
root = tk.Tk()
style = ttk.Style(root)

style.theme_use("clam")

style.map("CustomButton.TButton",
    foreground=[("pressed", "red"), ("active", "blue")], # we change the foreground color when we press the button
    background=[("pressed", "!disabled", "black"), ("active", "black")], # we change the background color ...
    font=[("pressed", ("TkDefaultFont", 15))] # We change the font size ...
    # to get the background working it is necessary to change the theme as windows theme (by default)
    # doesn't allow you to change background.
)

name = ttk.Label(root, text="Hello, world!")
entry = ttk.Entry(root, width=15)
button = ttk.Button(root, text="Press me.", style="CustomButton.TButton")
name.pack()
entry.pack()
button.pack()

root.mainloop()
```



# Tkinter themes, and how to change theme

## Changing `ttk.Entry` field font with `ttk.Style`

The gist of is, you can't. There might be a bug in the Tk engine which means that applying a `ttk.Style()` which uses the `font=` argument on a `ttk.Entry` results in the font change not being shown on the entry field.

Therefore, for `ttk.Entry` fields you must pass the `font=` argument directly to the constructor.

In other words, this won't work on entry fields. Or at least, the font will continue being the default font:

```
style = ttk.Style()
style.configure("LargeEntry.TEntry", font=("Segoe UI", 15))
entry = ttk.Entry(root, style="LargeEntry.TEntry")
entry.pack()
```

However, this *will* change the font:

```
entry = ttk.Entry(root, font=("Segoe UI", 15))
entry.pack()
```

## Using the global entry font

The next lecture in this course talks how to use the global entry font, and how that can greatly simplify your font-setting on `ttk.Entry` fields. However, it is partially limited in that it will change the styling for *all* entry fields:

```
import tkinter.font as font

...
Font.nametofont("TkEntryFont").configure(size=15)
...
entry = ttk.Entry(root) # The font is 15 pixels
entry.pack()
```

# Can you change the entry field font using styles?

## Changing `ttk.Entry` field font with `ttk.Style`

The gist of is, you can't. There might be a bug in the Tk engine which means that applying a `ttk.Style()` which uses the `font=` argument on a `ttk.Entry` results in the font change not being shown on the entry field.

Therefore, for `ttk.Entry` fields you must pass the `font=` argument directly to the constructor.

In other words, this won't work on entry fields. Or at least, the font will continue being the default font:

```
style = ttk.Style()
style.configure("LargeEntry.TEntry", font=("Segoe UI", 15))
entry = ttk.Entry(root, style="LargeEntry.TEntry")
entry.pack()
```

However, this *will* change the font:

```
entry = ttk.Entry(root, font=("Segoe UI", 15))
entry.pack()
```

## Using the global entry font

The next lecture in this course talks how to use the global entry font, and how that can greatly simplify your font-setting on `ttk.Entry` fields. However, it is partially limited in that it will change the styling for *all* entry fields:

```
import tkinter.font as font

...
font.nametofont("TkEntryFont").configure(size=15)
...
entry = ttk.Entry(root) # The font is 15 pixels
entry.pack()
```

# Can you change the entry field font using styles?

```
import tkinter as tk
from tkinter import ttk
import tkinter.font as font

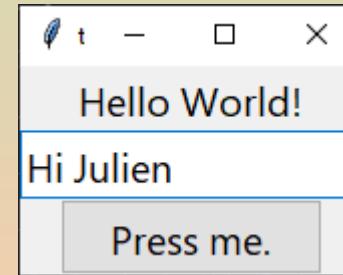
root = tk.Tk()
style = ttk.Style(root)

font.nametofont("TkDefaultFont").configure(size=15)
font.nametofont("TkTextFont").configure(size=15)

name = ttk.Label(root, text="Hello World!")
entry = ttk.Entry(root, width=15)
button = ttk.Button(root, text="Press me.")

name.pack()
entry.pack()
button.pack()

root.mainloop()
```



# Can you change the entry field font using styles?

## Tkinter specially named fonts

Font name	Description
TkDefaultFont	Default for items not otherwise specified.
TkTextFont	Used for entry widgets, listboxes, etc.
TkFixedFont	A standard fixed-width font.
TkMenuFont	The font used for menu items.
TkHeadingFont	Font for column headings in lists and tables.
TkCaptionFont	A font for window and dialog caption bars.
TkSmallCaptionFont	A smaller caption font for tool dialogs.
TkIconFont	A font for icon captions.
TkTooltipFont	A font for tooltips.

## Using named fonts

The most flexible approach is to define custom fonts (that can inherit from `TkEntryFont` or `TkDefaultFont`), and then manually pass them to each entry field. We also look at that in this course, in the lecture after the next.

# Can you change the entry field font using styles?

## Named fonts

Read the Tk Docs for more: <https://tkdocs.com/tutorial/fonts.html>

A couple of things are important to know:

The "family" specifies the font name; the names Courier, Times, and Helvetica are guaranteed to be supported (and mapped to an appropriate monospaced, serif, or sans-serif font).

## Create your own font

Creating your own font is easy enough. It gets stored in a variable and can be reused anywhere where our known font tuple could be passed:

```
import tkinter.font as font

warningLabelFont = font.Font(family="Helvetica", size=14, weight="bold")
```

## Re-use the default font

If you do this:

```
import tkinter.font as font

warningLabelFont = font.nametofont("TkDefaultFont")
warningLabelFont.configure(size=15)
```

You'll have problems, because the `warningLabelFont` is the default font. That means that all widgets which use the default font will change size.

Instead, `.copy()` the font to only use it in certain widgets:

```
import tkinter.font as font

warningLabelFont = font.nametofont("TkDefaultFont").copy()
warningLabelFont.configure(size=15)
```

# How to use named fonts in Tkinter

```
import tkinter as tk
from tkinter import ttk
import tkinter.font as font

root = tk.Tk()
style = ttk.Style(root)

# What family are available in the OS
print(font.families())

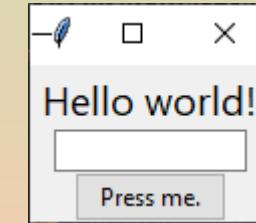
# Helvetica, Courier, Times are supported by all OS. The other family name are not necessary.
warningLabelFont = font.Font(family="Helvetica", size=14, weight="bold")

# To change the font
# warningLabelFont = font.nametofont("TkDefaultFont") # will change the fonts across everything (button and label are
# both large.
warningLabelFont = font.nametofont("TkDefaultFont").copy() # here you get a copy of default font, so only the label is
# in size 15 and bold while the button use the default font.
warningLabelFont.configure(size=15)

name = ttk.Label(root, text="Hello world!", font=warningLabelFont)
entry = ttk.Entry(root, width=15)
button = ttk.Button(root, text="Press me.")

name.pack()
entry.pack()
button.pack()

root.mainloop()
```



# PySide2

# PySide2(or Qt) vs Tkinter 1/9

```
notes.txt
1 TKinter.
2
3 Avantages.
4     - Natif avec Python, pas besoin d'installer de module supplémentaire
5     - Beaucoup de ressources en ligne pour débuter
6
7 Inconvénients.
8     - Interfaces difficilement personnalisable
9     - On atteint rapidement les limites
10    - Peu de développement / nouveautés
11
12
13 Qt.
14 <
15 Avantages.
16     - Multi-Plateforme
17     - Très complet
18     - Qt Designer pour faire nos interfaces avec un éditeur WYSIWYG
19     - Très utilisé et donc beaucoup de documentation
20     - En constante évolution
21     - Très utilisé dans le milieu professionnel
22     - Très utilisé dans d'autres logiciels
23
24 Inconvénients.
25     - Un peu plus complexe à utiliser
26
```

# PyQt vs PySide 2/9



*PyQt est gratuit pour les projets open source mais dès que l'on désire vendre un programme utilisant PyQt l'achat d'une License est nécessaire ~ 500 US dollars.  
Avec PySide pas besoin de License.*

# PySide2

## Création de la fenêtre principale 3/9

Import de la librairie PySide2 et de la fonction  
QtWidgets

Création de la class App qui hérite de  
QtWidgets.QWidget

Initialisation de la class enfant App

Initialisation de la class parent  
QtWidget.Qwidget

On crée une application à l'intérieur de laquelle  
On aura nos fenêtres (dans ce cas 1 fenêtre)

Instance de ma class App

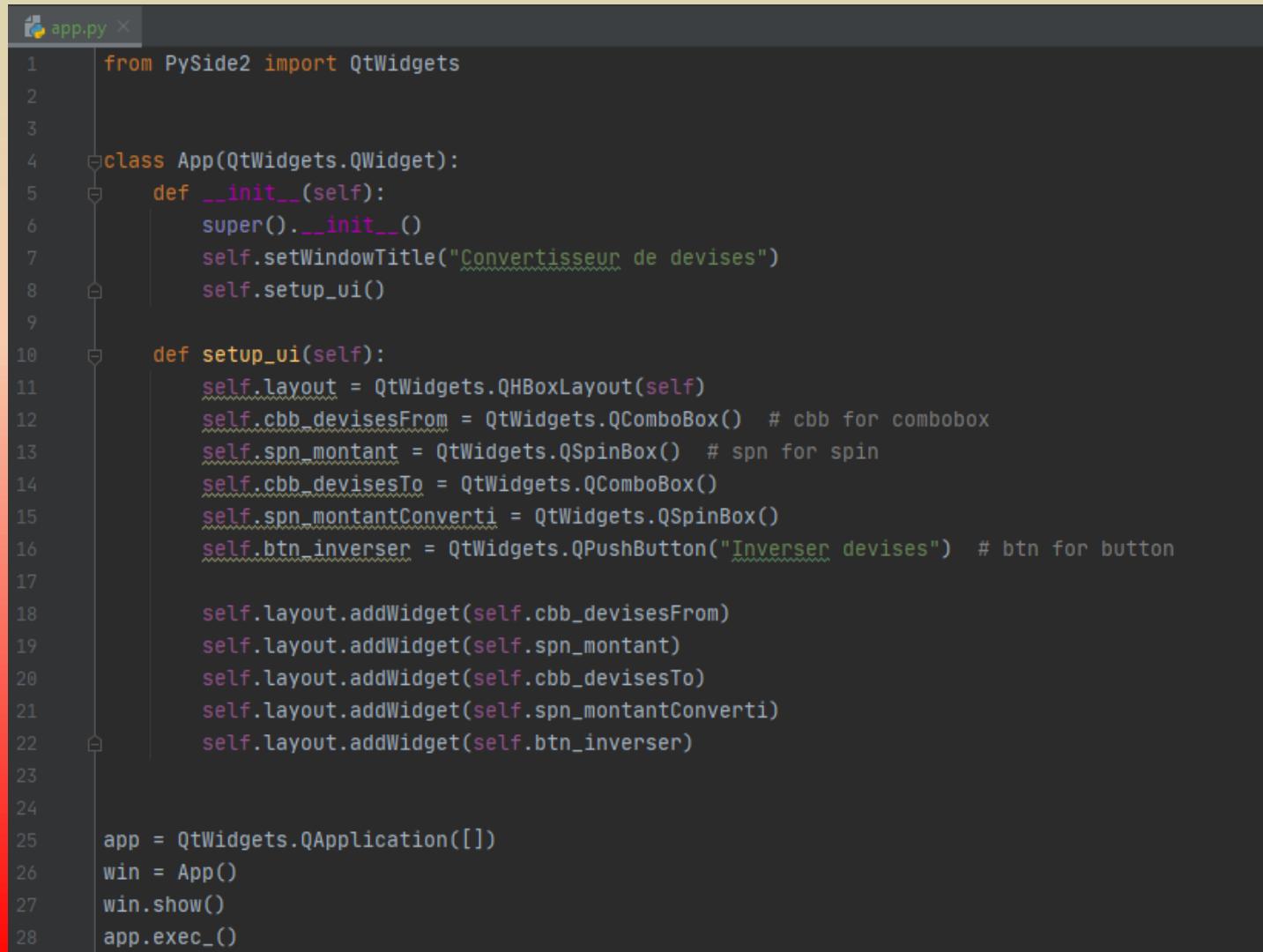
Afficher la fenêtre avec la méthode show()

app.exec\_() permet d'exécuter l'application

```
1 from PySide2 import QtWidgets
2
3 class App(QtWidgets.QWidget):
4     def __init__(self):
5         super().__init__()
6         self.setWindowTitle("Convertisseur
7                         de devises")
8
9     app = QtWidgets.QApplication([])
10    win = App()
11    win.show()
12    app.exec_()
```

# PySide2

## Création de l'interface 4/9



The screenshot shows a code editor window with a dark theme. The file is named "app.py". The code implements a simple GUI application using PySide2's QtWidgets module. It defines a class "App" that inherits from QWidget. The \_\_init\_\_ method sets the window title to "Convertisseur de devises" and calls setup\_ui(). The setup\_ui method creates five widgets: two QComboBoxes (cbb\_devisesFrom and cbb\_devisesTo), one QSpinBox (spn\_montant), and one QPushButton (btn\_inverser). It then adds these widgets to a horizontal layout. Finally, it creates an QApplication instance, creates an App object, shows it, and starts the application loop.

```
from PySide2 import QtWidgets

class App(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Convertisseur de devises")
        self.setup_ui()

    def setup_ui(self):
        self.layout = QtWidgets.QHBoxLayout(self)
        self.cbb_devisesFrom = QtWidgets.QComboBox() # cbb for combobox
        self.spn_montant = QtWidgets.QSpinBox() # spn for spin
        self.cbb_devisesTo = QtWidgets.QComboBox()
        self.spn_montantConverti = QtWidgets.QSpinBox()
        self.btn_inverser = QtWidgets.QPushButton("Inverser devises") # btn for button

        self.layout.addWidget(self.cbb_devisesFrom)
        self.layout.addWidget(self.spn_montant)
        self.layout.addWidget(self.cbb_devisesTo)
        self.layout.addWidget(self.spn_montantConverti)
        self.layout.addWidget(self.btn_inverser)

app = QtWidgets.QApplication([])
win = App()
win.show()
app.exec_()
```

# PySide2

## Définir des valeurs par défaut 5/9

L'ordre d'appel des fonctions est important !

L'ordre de définition des fonctions n'est pas important. Par convention on les places dans l'ordre alphabétique.

```
app.py x
1  from PySide2 import QtWidgets
2  import currency_converter
3
4  class App(QtWidgets.QWidget):
5      def __init__(self):
6          super().__init__()
7          self.c = currency_converter.CurrencyConverter()
8          self.setWindowTitle("Convertisseur de devises")
9          self.setup_ui()           # Appel de la methode setup_ui()
10         self.set_default_values() # Appel de la methode set_default_value()
11
12     def set_default_values(self):
13         self.cbb_devisesFrom.addItems(sorted(list(self.c.currencies))) # on transform le set en list et on trie
14         self.cbb_devisesTo.addItems(sorted(list(self.c.currencies)))
15         self.cbb_devisesFrom.setCurrentText("EUR")
16         self.cbb_devisesTo.setCurrentText("EUR")
17
18         self.spn_montant.setRange(1, 1000000)
19         self.spn_montantConvertis.setRange(1, 1000000)
20
21         self.spn_montant.setValue(100)
22         self.spn_montantConvertis.setValue(100)
23
24     def setup_ui(self):
25         self.layout = QtWidgets.QHBoxLayout(self)
26         self.cbb_devisesFrom = QtWidgets.QComboBox() # cbb for combobox
27         self.spn_montant = QtWidgets.QSpinBox() # spn for spin
28         self.cbb_devisesTo = QtWidgets.QComboBox()
29         self.spn_montantConvertis = QtWidgets.QSpinBox()
30         self.btn_inverser = QtWidgets.QPushButton("Inverser devises") # btn for button
31
32         self.layout.addWidget(self.cbb_devisesFrom)
33         self.layout.addWidget(self.spn_montant)
34         self.layout.addWidget(self.cbb_devisesTo)
35         self.layout.addWidget(self.spn_montantConvertis)
36         self.layout.addWidget(self.btn_inverser)
37
38
39     app = QtWidgets.QApplication([])
40     win = App()
41     win.show()
42     app.exec_()
```

# PySide2 Connecter les widgets aux méthodes 6/9

Connection entre les widgets et les méthodes



```
app.py x
1  from PySide2 import QtWidgets
2  import currency_converter
3
4  class App(QtWidgets.QWidget):
5      def __init__(self):
6          super().__init__()
7          self.c = currency_converter.CurrencyConverter()
8          self.setWindowTitle("Convertisseur de devises")
9          self.setup_ui() # Appel de la methode setup_ui()
10         self.set_default_values() # Appel de la methode set_default_value()
11         self.setup_connections() # connect les widgets aux methodes
12
13
14     def compute(self):
15         print("Compute")
16
17     def inverser_devise(self):
18         print("Inverser devise")
19
20     def setup_connections(self):
21         self.cbb_devisesFrom.activated.connect(self.compute)
22         self.cbb_devisesTo.activated.connect(self.compute)
23         self.spn_montant.valueChanged.connect(self.compute)
24         self.btn_inverser.clicked.connect(self.inverser_devise)
25
26     def set_default_values(self):
27         self.cbb_devisesFrom.addItems(sorted(list(self.c.currencies))) # on transform le set en list et on trie
28         self.cbb_devisesTo.addItems(sorted(list(self.c.currencies)))
29         self.cbb_devisesFrom.setCurrentText("EUR")
30         self.cbb_devisesTo.setCurrentText("EUR")
31
32         self.spn_montant.setRange(1, 1000000)
33         self.spn_montantConverti.setRange(1, 1000000)
34
35         self.spn_montant.setValue(100)
36         self.spn_montantConverti.setValue(100)
37
38     def setup_ui(self):
39         self.layout = QtWidgets.QHBoxLayout(self)
40         self.cbb_devisesFrom = QtWidgets.QComboBox() # cbb for combobox
41         self.spn_montant = QtWidgets.QSpinBox() # spn for spin
42         self.cbb_devisesTo = QtWidgets.QComboBox()
43         self.spn_montantConverti = QtWidgets.QSpinBox()
44         self.btn_inverser = QtWidgets.QPushButton("Inverser devises") # btn for button
```

```
        self.layout.addWidget(self.cbb_devisesFrom)
        self.layout.addWidget(self.spn_montant)
        self.layout.addWidget(self.cbb_devisesTo)
        self.layout.addWidget(self.spn_montantConverti)
        self.layout.addWidget(self.btn_inverser)

    )
    app = QtWidgets.QApplication([])
    win = App()
    win.show()
    app.exec_()
```

# PySide2 Convertir la devise et afficher le résultat 7/9

```
1  from PySide2 import QtWidgets
2  import currency_converter
3
4  class App(QtWidgets.QWidget):
5      def __init__(self):
6          super().__init__()
7          self.c = currency_converter.CurrencyConverter()
8          self.setWindowTitle("Convertisseur de devises")
9          self.setup_ui()          # Appel de la méthode setup_ui()
10         self.set_default_values() # Appel de la méthode set_default_value()
11         self.setup_connections() # connect les widgets aux méthodes
12
13     def compute(self):
14         montant = self.spn_montant.value()
15         devise_from = self.cbb_devisesFrom.currentText()
16         devise_to = self.cbb_devisesTo.currentText()
17         resultat = self.c.convert(montant, devise_from, devise_to)
18         self.spn_montantConverti.setValue(resultat)
19
20     def inverser_devise(self):
21         devise_from = self.cbb_devisesFrom.currentText()
22         devise_to = self.cbb_devisesTo.currentText()
23
24         self.cbb_devisesFrom.setCurrentText(devise_to)
25         self.cbb_devisesTo.setCurrentText(devise_from)
26
27         self.compute()
28
29     def setup_connections(self):
30         self.cbb_devisesFrom.activated.connect(self.compute)
31         self.cbb_devisesTo.activated.connect(self.compute)
32         self.spn_montant.valueChanged.connect(self.compute)
33         self.btn_inverser.clicked.connect(self.inverser_devise)
34
35     def set_default_values(self):
36         self.cbb_devisesFrom.addItems(sorted(list(self.c.currencies))) # on transform le set en list et on trie
37         self.cbb_devisesTo.addItems(sorted(list(self.c.currencies)))
38         self.cbb_devisesFrom.setCurrentText("EUR")
39         self.cbb_devisesTo.setCurrentText("EUR")
40
41         self.spn_montant.setRange(1, 1000000)
42         self.spn_montantConverti.setRange(1, 1000000)
43
44         self.spn_montant.setValue(100)
45         self.spn_montantConverti.setValue(100)
```

```
48     def setup_ui(self):
49         self.layout = QtWidgets.QHBoxLayout(self)
50         self.cbb_devisesFrom = QtWidgets.QComboBox() # cbb for combobox
51         self.spn_montant = QtWidgets.QSpinBox() # spn for spin
52         self.cbb_devisesTo = QtWidgets.QComboBox()
53         self.spn_montantConverti = QtWidgets.QSpinBox()
54         self.btn_inverser = QtWidgets.QPushButton("Inverser devises") # btn for button
55
56         self.layout.addWidget(self.cbb_devisesFrom)
57         self.layout.addWidget(self.spn_montant)
58         self.layout.addWidget(self.cbb_devisesTo)
59         self.layout.addWidget(self.spn_montantConverti)
60         self.layout.addWidget(self.btn_inverser)
61
62
63     app = QtWidgets.QApplication([])
64     win = App()
65     win.show()
66     app.exec_()
```

# PySide2

## Gérer les erreurs 8/9

```
1  from PySide2 import QtWidgets
2  import currency_converter
3
4  class App(QtWidgets.QWidget):
5      def __init__(self):
6          super().__init__()
7          self.c = currency_converter.CurrencyConverter()
8          self.setWindowTitle("Convertisseur de devises")
9          self.setup_ui()          # Appel de la methode setup_ui()
10         self.set_default_values() # Appel de la methode set_default_value()
11         self.setup_connections()  # connect les widgets aux methodes
12
13
14     def compute(self):
15         montant = self.spn_montant.value()
16         devise_from = self.cbb_devisesFrom.currentText()
17         devise_to = self.cbb_devisesTo.currentText()
18
19         try:
20             resultat = self.c.convert(montant, devise_from, devise_to)
21         except currency_converter.currency_converter.RateNotFoundError:
22             print("La conversion n'a pas fonctionné.")
23         else:
24             self.spn_montantConverti.setValue(resultat)
25
26     def inverser_devise(self):
27         devise_from = self.cbb_devisesFrom.currentText()
28         devise_to = self.cbb_devisesTo.currentText()
29
30         self.cbb_devisesFrom.setCurrentText(devise_to)
31         self.cbb_devisesTo.setCurrentText(devise_from)
32
33         self.compute()
34
35     def setup_connections(self):
36         self.cbb_devisesFrom.activated.connect(self.compute)
37         self.cbb_devisesTo.activated.connect(self.compute)
38         self.spn_montant.valueChanged.connect(self.compute)
39         self.btn_inverser.clicked.connect(self.inverser_devise)
40
41     def set_default_values(self):
42         self.cbb_devisesFrom.addItems(sorted(list(self.c.currencies))) # on transform le set en list et on trie
43         self.cbb_devisesTo.addItems(sorted(list(self.c.currencies)))
44         self.cbb_devisesFrom.setCurrentText("EUR")
45         self.cbb_devisesTo.setCurrentText("EUR")
```

```
47         self.spn_montant.setRange(1, 1000000)
48         self.spn_montantConverti.setRange(1, 1000000)
49
50
51         self.spn_montant.setValue(100)
52         self.spn_montantConverti.setValue(100)
53
54     def setup_ui(self):
55         self.layout = QtWidgets.QHBoxLayout(self)
56         self.cbb_devisesFrom = QtWidgets.QComboBox() # cbb for combobox
57         self.spn_montant = QtWidgets.QSpinBox() # spn for spin
58         self.cbb_devisesTo = QtWidgets.QComboBox()
59         self.spn_montantConverti = QtWidgets.QSpinBox()
60         self.btn_inverser = QtWidgets.QPushButton("Inverser devises") # btn for button
61
62         self.layout.addWidget(self.cbb_devisesFrom)
63         self.layout.addWidget(self.spn_montant)
64         self.layout.addWidget(self.cbb_devisesTo)
65         self.layout.addWidget(self.spn_montantConverti)
66         self.layout.addWidget(self.btn_inverser)
67
68     app = QtWidgets.QApplication([])
69     win = App()
70     win.show()
71     app.exec_()
```

# PySide2

# changer le style de l'interface 9/9

Pour modifier le style on utilise le CSS. Attention ce n'est pas du 1 pour 1 avec le CSS web.

```
1  from PySide2 import QtWidgets
2  import currency_converter
3
4  class App(QtWidgets.QWidget):
5      def __init__(self):
6          super().__init__()
7          self.c = currency_converter.CurrencyConverter()
8          self.setWindowTitle("Convertisseur de devises")
9          self.setup_ui()      # Appel de la methode setup_ui()
10         self.set_default_values()    # Appel de la methode set_default_value()
11         self.setup_css()
12         self.setup_connections()    # connect les widgets aux methodes
13
14
15     def compute(self):
16         montant = self.spn_montant.value()
17         devise_from = self.cbb_devisesFrom.currentText()
18         devise_to = self.cbb_devisesTo.currentText()
19
20         try:
21             resultat = self.c.convert(montant, devise_from, devise_to)
22         except currency_converter.currency_converter.RateNotFoundError:
23             print("La conversion n'a pas fonctionné.")
24         else:
25             self.spn_montantConverti.setValue(resultat)
26
27     def inverser_devise(self):
28         devise_from = self.cbb_devisesFrom.currentText()
29         devise_to = self.cbb_devisesTo.currentText()
30
31         self.cbb_devisesFrom.setCurrentText(devise_to)
32         self.cbb_devisesTo.setCurrentText(devise_from)
33
34         self.compute()
35
36     def setup_connections(self):
37         self.cbb_devisesFrom.activated.connect(self.compute)
38         self.cbb_devisesTo.activated.connect(self.compute)
39         self.spn_montant.valueChanged.connect(self.compute)
40         self.btn_inverser.clicked.connect(self.inverser_devise)
41
42     def setup_css(self):
43         self.setStyleSheet("""
44             background-color: rgb(30, 30, 30);
45             color: rgb(240, 240, 240);
46             border: none;
47             """)
```

Distribuer facilement  
nos programmes avec  
`cx_freeze`

# Distribuer facilement nos programmes avec cx\_freeze

Comme nous l'avons vu, Python nous permet de générer des exécutables d'une façon assez simple. Mais, si vous en venez à vouloir distribuer votre programme, vous risquez de vous heurter au problème suivant : pour lancer votre code, votre destinataire doit installer Python ; qui plus est, la bonne version. Et si vous commencez à utiliser des bibliothèques tierces, il doit aussi les installer !

Heureusement, il existe plusieurs moyens pour produire des fichiers exécutables que vous pouvez distribuer et qui incluent tout le nécessaire.

Sur Windows, il faut enfermer vos fichiers à l'extension .py dans un .exe accompagné de fichiers.dll. Cx\_freeze est un des outils qui permet d'atteindre cet objectif.

# En théorie

L'objectif de ce chapitre est de vous montrer comment faire des programmes dits standalone, que l'on peut traduire très littéralement par "se tenir seul". Comme vous le savez, pour que vos fichiers.py s'exécutent, il faut que Python soit installé sur votre machine. Mais vous pourriez vouloir transmettre votre programme sans obliger vos utilisateurs à installer Python sur leur ordinateur.

Une version standalone de votre programme contient, en plus de votre code, l'exécutable Python et les dépendances dont il a besoin.

Sur Windows, vous retrouverez avec un fichier.exeet plusieurs fichiers compagnons, bien plus faciles à distribuer et, pour vos utilisateurs, à exécuter.

Le programme résultant ne sera pas sensiblement plus rapide ou plus lent. Il ne s'agit pas de compilation, Python reste un langage interprété et l'interpréteur sera appelé pour lire votre code, même si celui-ci se trouvera dans une forme un peu plus compressée.

## Avantages de cx\_Freeze

- Portabilité : cx\_Freeze est fait pour fonctionner aussi bien sur Windows que sur Linux ou Mac OS ;
- Compatibilité : cx\_Freeze fonctionne sur des projets Python de la branche 2.X ou 3.X ;
- Simplicité : créer son programme standalone avec cx\_Freeze est simple et rapide ;
- Souplesse : vous pouvez aisément personnaliser votre programme standalone avant de le construire.

Il existe d'autres outils similaires, dont le plus célèbre est py2exe, que vous pouvez télécharger sur le site officiel de py2exe.

Il a toutefois l'inconvénient de ne fonctionner que sur Windows et, à l'heure où j'écris ces lignes du moins, de ne pas proposer de version compatible avec Python 3.X.

Nous allons à présent voir comment installer cx\_Freeze et comment construire nos programmes standalone.

# En pratique

Il existe plusieurs façons d'utiliser cx\_Freeze. Il nous faut dans tous les cas commencer par l'installer.

## Installation

### Sur Windows

Rendez-vous sur le site [sourceforge](#), où est hébergé le projet cx\_Freeze.

et téléchargez le fichier correspondant à votre version de Python.

Après l'avoir téléchargé, lancez l'exécutable et laissez-vous guider. Rien de trop technique jusqu'ici !

### Sur Linux

Je vous conseille d'installer cx\_Freeze depuis les sources.

Commencez par vous rendre sur le site de téléchargement ci-dessus et sélectionnez la dernière version de cx\_Freeze (Source Code only).

Téléchargez et décompressez les sources :

```
tar -xvf cx_Freeze_version.tar.gz
```

Rendez-vous dans le dossier décompressé puis lancez l'installation en tant qu'utilisateur **root**:

```
$ cd cx_Freeze_version  
$ python3.4 setup.py build  
$ sudo python3.4 setup.py install
```

Si ces deux commandes s'exécutent convenablement, vous disposerez de la commande `cxfreeze`:

# En pratique

```
cxfreeze  
cxfreeze: error: script or a list of modules must be specified
```

## Utiliser le script cxfreeze

Pour les utilisateurs de Windows, je vous invite à vous rendre dans la ligne de commande (Démarrer>Exécuter... >cmd).

Rendez-vous dans le sous-dossier scripts de votre installation Python (chez moi, C:\python34\scripts).

```
cd \  
cd C:\python34\scripts
```

Sur Linux, vous devez avoir accès au script directement. Vous pouvez le vérifier en tapant cxfreeze dans la console. Si cette commande n'est pas disponible mais que vous avez installé cxfreeze, vous pourrez trouver le script dans le répertoire bin de votre version de Python.

Sur Windows ou Linux, la syntaxe du script est la même : cxfreeze fichier.py.

Faisons un petit programme pour le tester. Créez un fichiersalut.py (sur Windows, mettez-le dans le même dossier que le script cxfreeze, ce sera plus simple pour le test). Vous pouvez y placer le code suivant :

```
"""Ce fichier affiche simplement une ligne grâce  
à la fonction print."""  
  
import os  
  
print("Salut le monde !")  
  
# Sous Windows il faut mettre ce programme en  
pause (inutile sous Linux)  
os.system("pause")
```

N'oubliez pas la ligne spécifiant l'encodage.

# En pratique

À présent, lancez le script **cxfreeze** en lui passant en paramètre le nom de votre fichier :`cxfreeze salut.py`.

Si tout se passe bien, vous vous retrouvez avec un sous-dossier `dist` qui contient les bibliothèques dont votre programme a besoin pour s'exécuter... et votre programme lui-même.

Sur Windows, ce sera `salut.exe`. Sur Linux, ce sera simplement `salut`.

Vous pouvez lancer cet exécutable : comme vous le voyez, votre message s'affiche bien à l'écran.

Formidable ! Ou pas...

Au fond, vous ne voyez sans doute pas de différence avec votre programme `salut.py`. Vous pouvez l'exécuter, lui aussi, il n'y a aucune différence.

Sauf que l'exécutable que vous trouvez dans le sous-dossier `dist` n'a pas besoin de Python pour s'exécuter : il contient lui-même l'interpréteur Python.

Vous pouvez donc distribuer ce programme à vos amis ou le mettre en téléchargement sur votre site, si vous le désirez.

Une chose importante à noter, cependant : veillez à copier, en même temps que votre programme, tout ce qui se trouve dans le dossier `dist`. Sans quoi, votre exécutable pourrait ne pas se lancer convenablement.

Le script **cxfreeze** est très pratique et suffit bien pour de petits programmes. Il comporte certaines options utiles que vous pouvez retrouver dans [la documentation de cx\\_Freeze](#).

Nous allons à présent voir une seconde méthode pour utiliser **cx\_Freeze**.

# En pratique

## Le fichier `setup.py`

La seconde méthode n'est pas bien plus difficile mais elle peut se révéler plus puissante à l'usage. Cette fois, nous allons créer un fichier `setup.py` qui se charge de créer l'exécutable de notre programme.

Un fichier `setup.py` basique contient ce code :

```
"""Fichier d'installation de notre script salut.py."""

from cx_Freeze import setup, Executable

# On appelle la fonction setup
setup(
    name = "salut",
    version = "0.1",
    description = "Ce programme vous dit bonjour",
    executables = [Executable("salut.py")],
)
```

Tout tient dans l'appel à la fonction `setup`. Elle possède plusieurs arguments nommés :

- `name`: le nom de notre futur programme.
- `version`: sa version.
- `description`: sa description.
- `executables`: une liste contenant des objets de type `Executable`, type que vous importez de `cx_Freeze`. Pour se construire, celui-ci prend en paramètre le chemin du fichier `.py`(ici, c'est notre fichier `salut.py`).

Maintenant, pour créer votre exécutable, vous lancez `setup.py` en lui passant en paramètre la commande `build`.

Sur Windows, dans la ligne de commande :`C:\python34\python.exe setup.py build`.

Et sur Linux :`$ python3.4 setup.py build`.

# En pratique

## Pour conclure

Ceci n'est qu'un survol de cx\_Freeze. Vous trouverez plus d'informations dans la documentation indiquée plus haut, si vous voulez connaître les différentes façons d'utiliser cx\_Freeze.

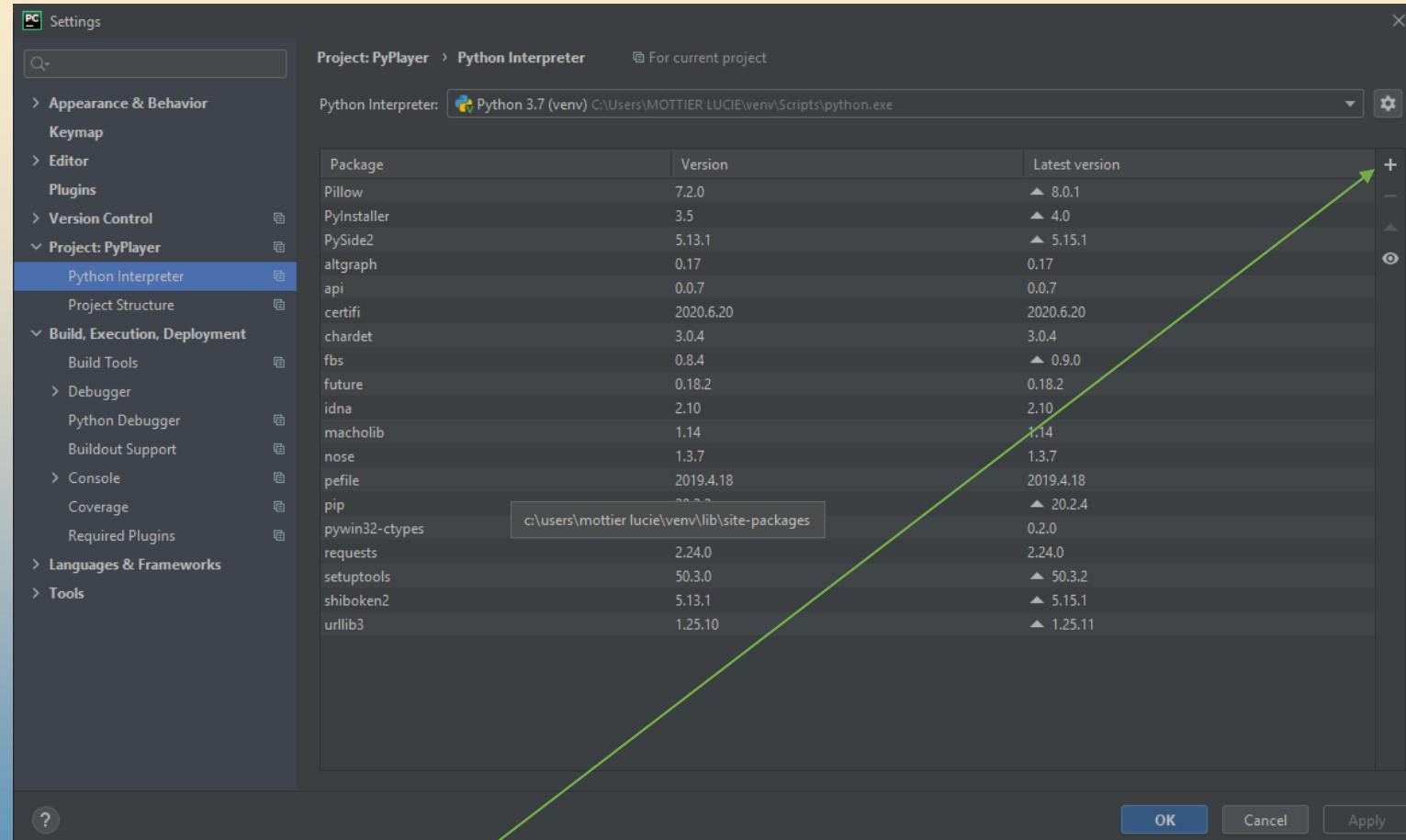
# Résumé

## En résumé

- cx\_Freeze est un outil permettant de créer des programmes Python standalone.
- Un programme standalone signifie qu'il contient lui-même les dépendances dont il peut avoir besoin, ce qui rend sa distribution plus simple.
- cx\_Freeze installe un script qui permet de créer nos programmes standalone très rapidement.
- On peut arriver à un résultat analogue en créant un fichier appelé traditionnellement `setup.py`.

Distribuer facilement  
nos programmes avec  
**FBS**

# FBS installation de l'environnement



Installer FBS

Dans PyCharm:

File -> Settings -> Python Interpréter

Cliquer sur le + à gauche de l'écran puis chercher fbs sélectionner install package.

# FBS créer une application FBS 1/

Dans commander ou le terminal de pycharm, aller dans le répertoire de votre projet, pour l'explication j'utilise un environnement virtuel et cmder.

- 1ere étape source mon environnement virtuel

```
source ~/venv/Scripts/activate
```

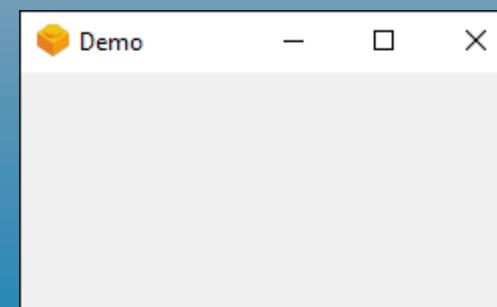
- 2ème étape on va dans le répertoire de travail

```
(venv) MOTTIER LUCIE@DESKTOP-23A2N2I ~/Documents/GitHub/DemoFBS (master)
λ fbs startproject
App name [MyApp] : Demo
Author [Palleau Julien] : juju
Mac bundle identifier (eg. com.julien.demo, optional):
```

Created the src/ directory. If you have PySide2 installed, you can now  
do:

```
fbs run
Created
```

Ensuite on peut faire « fbs run » et cela va lancer notre programme.



# FBS créer une application FBS 2/

Dans commander ou le terminal de pycharm, aller dans le répertoire de votre projet, pour l'explication j'utilise un environnement virtuel.

- 1ere étape source mon environnement virtuel

```
λ pwd  
/c/Users/MOTTIER LUCIE/Documents/GitHub/DemoFBS  
λ ls  
src/  
  
λ cd src  
λ ls  
build      main  
  
λ cd main  
λ ls  
Icons      python  
  
λ cd python  
λ ls  
main.py  
  
#####  
  
λ pwd  
/c/Users/MOTTIER LUCIE/Documents/GitHub/DemoFBS  
λ fbs freeze  
Done. You can now run `target\Demo\Demo.exe`. If that doesn't work,  
see https://build-system.fman.io/troubleshooting.  
  
λ cd target/  
(venv) MOTTIER LUCIE@DESKTOP-23A2N2I ~/Documents/GitHub/Udemy/PySide2/PyPlayer/target (master)  
λ ls  
Demo/ PyInstaller/ # L'executable se trouve dans le repertoire Demo
```

# Installation d'un logiciel d'installateur/

Télécharger le logiciel nullsoft, sur nsis.sourceforge.io et l'installer sous windows.

Ajouter le chemin ou NSIS a été installé:

1. Copier le path ou l'exécutable nsis se trouve, pour moi c'est "C:\Program Files (x86)\NSIS\NSIS.exe"
2. Clique droit sur la tuile windows, Recherche, puis tapez enviro, Windows devrait vous proposer « Modifier les variable d'environnement systèmes », cliquez dessus
3. Cliquez sur Variable d'environnement
4. Choisir Path
5. Cliquer sur Modifier
6. Cliquer sur Nouveau en haut à gauche
7. Coller le path de l'executable
8. Cliquer sur Ok, Ok, Ok

```
λ pwd  
/c/Users/MOTTIER LUCIE/Documents/GitHub/DemoFBS  
λ ls  
src/          target/  
  
λ fbs installer  
Created target\DemoSetup.exe.  
  
λ cd target/  
(venv) MOTTIER LUCIE@DESKTOP-23A2N2I ~/Documents/GitHub/Udemy/PySide2/PyPlayer/target (master)  
λ ls  
Demo/ DemoSetup.exe* installer/ PyInstaller/
```

Nom	Modifié le	Type	Taille
Demo	08/11/2020 17:40	Dossier de fichiers	
installer	08/11/2020 17:56	Dossier de fichiers	
PyInstaller	08/11/2020 17:39	Dossier de fichiers	
DemoSetup.exe	08/11/2020 17:57	Application	38 946 Ko

Distribuer facilement  
nos programmes avec  
Pyinstaller

# Packaging and Distributing executables

Install pyinstaller (in pycharm go to File, Settings, Python interpreter, then click on + and search pyinstaller and add it)

Pyinstaller reads a Python script written by you. It analyzes your code to discover every other module and library your script needs in order to execute. Then it collects copies of all those files

- Including the Python interpreter!
- And puts them with your script in a single folder, or optionally in a single executable file.
- Important !!! Pyinstaller can only generate an executable for the platform on which it runs. Windows exe if you generate your package on windows, mac application if you generate your package on mac and so on so forth.

Two ways to bundle apps into executables with Python

There are two main ways to bundle your application using pyinstaller: into one folder, together with all the assets, or compressed into one file.

One folder makes everything easier.

One file makes it more secure (your users don't have access to assets), but a bit slower (it has to be decompressed before running) and might require a bit more work. That's because when running the "one file" application, a temporary folder has to be created prior to starting the application.

The official documentation does a good job explaining the difference in depth:

- [The one folder docs](#)
- [The one file docs](#)

# Building a standalone app

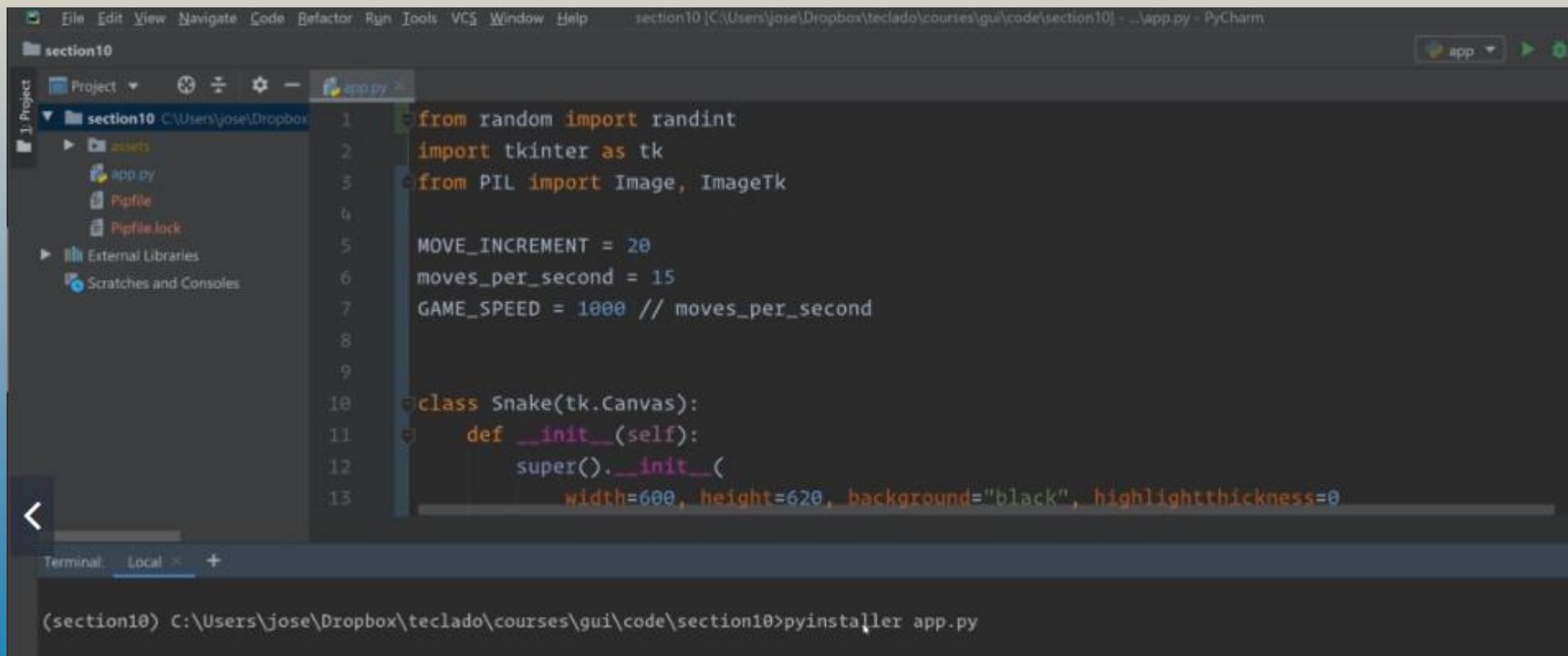
Open a terminal in PyCharm and run **pyinstaller test.py --onefile**.

# How to include data files with an executable Python app

<https://www.udemy.com/course/desktop-gui-python-tkinter/learn/lecture/16738726#search>

Open a terminal in PyCharm and run **pyinstaller app.py**

app.py is the name of your python program



The screenshot shows the PyCharm IDE interface. The top bar displays the menu: File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help. The title bar shows the project name: section10 [C:\Users\jose\Dropbox\teclado\courses\gui\code\section10] - ..\app.py - PyCharm. The left sidebar shows the Project structure with a file named app.py selected. The main editor area contains Python code for a Snake game:

```
from random import randint
import tkinter as tk
from PIL import Image, ImageTk

MOVE_INCREMENT = 20
moves_per_second = 15
GAME_SPEED = 1000 // moves_per_second

class Snake(Canvas):
    def __init__(self):
        super().__init__(
            width=600, height=620, background="black", highlightthickness=0
        )
```

Below the editor is a terminal window titled "Local" showing the command: (section10) C:\Users\jose\Dropbox\teclado\courses\gui\code\section10>pyinstaller app.py

# How to include data files in a –onefile executable

<https://www.udemy.com/course/desktop-gui-python-tkinter/learn/lecture/16738732#search>

First thing to do is to import 2 packages:

```
import sys  
from os import path
```

So below is an example of what I did for Snake game:

```
def load_assets(self):  
    try:  
        bundle_dir = getattr(sys, "_MEIPASS", path.abspath(path.dirname(__file__)))  
        path_to_snake = path.join(bundle_dir, "assets", "snake.png")  
  
        self.snake_body_image = Image.open(path_to_snake)  
        self.snake_body = ImageTk.PhotoImage(self.snake_body_image)  
  
        path_to_food = path.join(bundle_dir, "assets", "food.png")  
        self.food_image = Image.open(path_to_food)  
        self.food = ImageTk.PhotoImage(self.food_image)  
    except IOError as error:  
        print(error)  
        root.destroy()
```

MEIPASS is the variable we get in our system that tells you where the bundle is.

This is a default value in case MEIPASS is not there, which is the case when you run the script from pycharm. So, your script can be turned into an executable or run from pycharm

This is the path to snake and food images.

We are copying the content of the assets folder and paste them into a directory inside our bundle folder. It's going to create an assets folder into our bundle folder and pull out the contents inside there.

```
(section10) C:\Users\jose\Dropbox\teclado\courses\gui\code\section10>pyinstaller app.py --onefile --add-data "assets;assets"
```

# How to hide the console window when packaging applications

<https://www.udemy.com/course/desktop-gui-python-tkinter/learn/lecture/16738740#search>

```
(section10) C:\Users\jose\Dropbox\teclado\courses\gui\code\section10>pyinstaller app.pyw --add-data "assets;assets" --windowed
```



-windowed option open the game without opening a terminal window in addition of the game. To make it short with this option you only get the game on your screen.

In addition if there is an error when launching the application you'll get a native error window telling you that there was an error.

# When things go wrong packaging Python app

## When things go wrong

Pyinstaller has a lengthy section of their documentation detailing what to do when things go wrong:<https://pyinstaller.readthedocs.io/en/stable/when-things-go-wrong.html>

In addition to that, I'd like to give you some practical tips.

Keeping the Windows console when an error happens

By default, the Windows console will automatically close when the application crashes. This can make it really difficult to see an error that's printed out there.

The first time you encounter an error, you should surround your entire code with a try-except block. In there, print the traceback and also include an `input()` statement so that the Windows console does not close until you enter something (to satisfy the input() function).

Like this:

```
try:  
    root = tk.Tk()  
    root.title("Snake")  
    root.resizable(False, False)  
  
    board = Snake()  
    board.pack()  
  
    root.mainloop()  
except:  
    import traceback  
    traceback.print_exc()  
    input("Press Enter to end...")
```

The last few lines in the except block are important, as they'll allow you to see what the original exception was!

# Building Python apps for multiple platforms

You can go to [circleci.com](https://circleci.com), which provide multiple platforms in order to package your application, however it is not free!

# Les bonnes pratiques

# Les bonnes pratiques

Nous allons à présent nous intéresser à quelques bonnes pratiques de codage en Python.

Les conventions que nous allons voir sont, naturellement, des propositions. Vous pouvez coder en Python sans les suivre.

Toutefois, prenez le temps de considérer les quelques affirmations ci-dessous. Si vous vous sentez concernés, ne serait-ce que par une d'entre elles, je vous invite à lire ce chapitre :

- Un code dont on est l'auteur peut être difficile à relire si on l'abandonne quelque temps.
- Lire le code d'un autre développeur est toujours plus délicat.
- Si votre code doit être utilisé par d'autres, il doit être facile à reprendre (à lire et à comprendre).

# Pourquoi suivre les conventions des PEP ?

Vous avez absolument le droit de répondre en disant que personne ne lira votre code de toute façon et que vous n'aurez aucun mal à comprendre votre propre code. Seulement, si votre code prend des proportions importantes, si l'application que vous développez devient de plus en plus utilisée ou si vous vous lancez dans un gros projet, il est préférable pour vous d'adopter quelques conventions clairement définies dès le début. Et, étant donné qu'il n'est jamais certain qu'un projet, même démarré comme un amusement passager, ne devienne pas un jour énorme, ayez les bons réflexes dès le début !

En outre, vous ne pouvez jamais être sûrs à cent pour cent qu'aucun développeur ne vous rejoindra, à terme, sur le projet. Si votre application est utilisée par d'autres, là encore, ce jour arrivera peut-être lorsque vous n'aurez pas assez de temps pour poursuivre seul son développement.

Quoi qu'il en soit, je vais vous présenter plusieurs conventions qui nous sont proposées au travers de PEP (Python Enhancement Proposal : proposition d'amélioration de Python). Encore une fois, il s'agit de propositions et vous pouvez choisir d'autres conventions si celles-ci ne vous plaisent pas.

# La PEP 20 : tout une philosophie

La PEP 20, intitulée The Zen of Python, nous donne des conseils très généraux sur le développement. Elle est disponible sur le site de Python.

Bien entendu, ce sont davantage des conseils axés sur « comment programmer en Python » mais la plupart d'entre eux peuvent s'appliquer à la programmation en général.

Je vous propose une traduction de cette PEP :

- Beautiful is better than ugly : le beau est préférable au laid ;
- Explicit is better than implicit : l'explicite est préférable à l'implicite ;
- Simple is better than complex : le simple est préférable au complexe ;
- Complex is better than complicated : le complexe est préférable au compliqué ;
- Flat is better than nested : le plat est préférable à l'imbriqué. Moins littéralement, du code trop imbriqué (par exemple une boucle imbriquée dans une boucle imbriquée dans une boucle...) est plus difficile à lire ;
- Sparse is better than dense : l'aéré est préférable au compact ;
- Readability counts : la lisibilité compte ;
- Special cases aren't special enough to break the rules : les cas particuliers ne sont pas suffisamment particuliers pour casser la règle ;
- Although practicality beats purity : même si l'aspect pratique doit prendre le pas sur la pureté. Moins littéralement, il est difficile de faire un code à la fois fonctionnel et « pur » ;
- Errors should never pass silently : les erreurs ne devraient jamais passer silencieusement ;
- Unless explicitly silenced : à moins qu'elles n'aient été explicitement réduites au silence ;
- In the face of ambiguity, refuse the temptation to guess : en cas d'ambiguïté, résistez à la tentation de deviner ;
- There should be one -- and preferably only one -- obvious way to do it : il devrait exister une (et de préférence une seule) manière évidente de procéder ;

# La PEP 20 : tout une philosophie

- Although that way may not be obvious at first unless you're Dutch : même si cette manière n'est pas forcément évidente au premier abord, à moins que vous ne soyez Néerlandais ; % il faudrait peut-être indiquer que c'est une blague ?
- Now is better than never : maintenant est préférable à jamais ;
- Although never is often better than \*right\* now : mais jamais est parfois préférable à immédiatement ;
- If the implementation is hard to explain, it's a bad idea : si la mise en œuvre est difficile à expliquer, c'est une mauvaise idée ;
- If the implementation is easy to explain, it may be a good idea : si la mise en œuvre est facile à expliquer, ce peut être une bonne idée ;
- Namespaces are one honking great idea -- let's do more of those : les espaces de noms sont une très bonne idée (faisons-en plus !).

Comme vous le voyez, c'est une liste d'aphorismes très simples. Ils donnent des idées sur le développement Python mais, en les lisant pour la première fois, vous n'y voyez sans doute que peu de conseils pratiques.

Cependant, cette liste est vraiment importante et peut se révéler très utile. Certaines des idées qui s'y trouvent couvrent des pans entiers de la philosophie de Python.

Si vous travaillez sur un projet en équipe, un autre développeur pourra contester la mise en œuvre d'un extrait de code quelconque en se basant sur l'un des aphorismes cités plus haut.

Quand bien même vous travailleriez seul, il est toujours préférable de comprendre et d'appliquer la philosophie d'un langage quand on l'utilise pour du développement.

Je vous conseille donc de garder sous les yeux, autant que possible, cette synthèse de la philosophie de Python et de vous y référer à la moindre occasion. Commencez par lire chaque proposition. Les lignes sont courtes, prenez le temps de bien comprendre ce qu'elles veulent dire.

# La PEP 20 : tout une philosophie

- Sans trop détailler ce qui se trouve au-dessus (cela prendrait trop de temps), je signale à votre attention que plusieurs de ces aphorismes parlent surtout de l'allure du code. L'idée qui semble se dissimuler derrière, c'est qu'un code fonctionnel n'est pas suffisant : il faut, autant que possible, faire du « beau code ». Qui fonctionne, naturellement... mais ce n'est pas suffisant !
- Maintenant, nous allons nous intéresser à deux autres PEP qui vous donnent des conseils très pratiques sur votre développement :
  - la première nous donne des conseils très précis sur la présentation du code ;
  - la seconde nous donne des conseils sur la documentation au cœur de notre code.

# La PEP 8 : des conventions précises

Maintenant que nous avons vu des directives très générales, nous allons nous intéresser à une autre proposition d'amélioration, la PEP 8. Elle nous donne des conseils très précis sur la forme du code. Là encore, c'est à vous de voir : vous pouvez appliquer la totalité des conseils donnés ici ou une partie seulement. Vous pouvez retrouver [la PEP 8 sur le site de Python](#).

Je ne vais pas reprendre tout ce qui figure dans cette PEP mais je vais expliquer la plupart des conseils en les simplifiant. Par conséquent, si l'une des propositions présentées dans cette section manque d'explications à vos yeux, je vous conseille d'aller faire un tour sur la PEP originale. Ce qui suit n'est pas une traduction complète, j'insiste sur ce point.

## Introduction

L'une des convictions de Guido (Guido Van Rossum, créateur et BDFL, Benevolent Dictator For Life soit « dictateur bienveillant à vie » de Python) est que le code est lu beaucoup plus souvent qu'il n'est écrit. Les conseils donnés ici sont censés améliorer la lisibilité du code. Comme le dit la PEP 20, la lisibilité compte !

Un guide comme celui-ci parle de cohérence. La cohérence au cœur d'un projet est importante. La cohérence au sein d'une fonction ou d'un module est encore plus importante.

Mais il est encore plus essentiel de savoir « quand » être incohérent (parfois, les conseils de style donnés ici ne s'appliquent pas). En cas de doute, remettez-vous-en à votre bon sens. Regardez plusieurs exemples et choisissez celui qui semble le meilleur.

Il y a deux bonnes raisons de ne pas respecter une règle donnée :

1. Quand appliquer la règle rend le code moins lisible.
2. Dans un soucis de cohérence avec du code existant qui ne respecte pas cette règle non plus. Ce cas peut se produire si vous utilisez un module ou une bibliothèque qui ne respecte pas les mêmes conventions que celles définies ici.

# La PEP 8 : des conventions précises

## Forme du code

Indentation : utilisez 4 espaces par niveau d'indentation.

Tabulations ou espaces : ne mélangez jamais, dans le même projet, des indentations à base d'espaces et d'autres à base de tabulations. À choisir, on préfère généralement les espaces mais les tabulations peuvent également utilisées pour marquer l'indentation.

Longueur maximum d'une ligne : limitez vos lignes à un maximum de 79 caractères. De nombreux éditeurs favorisent des lignes de 79 caractères maximum. Pour les blocs de texte relativement longs (docstrings, par exemple), limitez-vous de préférence à 72 caractères par ligne.

Quand cela est possible, découpez vos lignes en utilisant des parenthèses, crochets ou accolades plutôt que l'anti-slash\|. Exemple :

```
appel_d_une_fonction(parametre_1, parametre_2,  
                      parametre_3, parametre_4):
```

Si vous devez découper une ligne trop longue, faites la césure après l'opérateur, pas avant.

```
# Oui  
un_long_calcul = variable + \  
                  taux * 100  
  
# Non  
un_long_calcul = variable \  
                  + taux * 100
```

Sauts de ligne : séparez par deux sauts de ligne la définition d'une fonction et la définition d'une classe.

Les définitions de méthodes au cœur d'une classe sont séparées par une ligne vide. Des sauts de ligne peuvent également être utilisés, parcimonieusement, pour délimiter des portions de code

Encodage : à partir de Python 3.0, il est conseillé d'utiliser, dans du code comportant des accents, l'encodage Utf-8.

# La PEP 8 : des conventions précises

## Directives d'importation

- Les directives d'importation doivent préférentiellement se trouver sur plusieurs lignes. Par exemple :

```
import os  
import sys
```

plutôt que :

```
import os, sys
```

Cette syntaxe est cependant acceptée quand on importe certaines données d'un module :

```
from subprocess import Popen, PIPE
```

Les directives d'importation doivent toujours se trouver en tête du fichier, sous la documentation éventuelle du module mais avant la définition de variables globales ou de constantes du module.

Les directives d'importation doivent être divisées en trois groupes, dans l'ordre :

les directives d'importation faisant référence à la bibliothèque standard ;

les directives d'importation faisant référence à des bibliothèques tierces ;

les directives d'importation faisant référence à des modules de votre projet.

Il devrait y avoir un saut de ligne entre chaque groupe de directives d'importation.

Dans vos directives d'importation, utilisez des chemins absolus plutôt que relatifs. Autrement dit :

```
from paquet.souspaquet import module
```

```
# Est préférable à  
from . import module
```

# La PEP 8 : des conventions précises

## Le signe espace dans les expressions et instructions

- Évitez le signe espace dans les situations suivantes :

- Au cœur des parenthèses, crochets et accolades :

```
# Oui
spam(ham[1], {eggs: 2})
```

```
# Non
spam( ham[ 1 ], { eggs: 2 } )
```

- Juste avant une virgule, un point-virgule ou un signe deux points :

```
# Oui
if x == 4: print x, y; x, y = y, x
```

```
# Non
if x == 4 : print x , y ; x , y = y , x
```

- Juste avant la parenthèse ouvrante qui introduit la liste des paramètres d'une fonction :

```
# Oui
spam(1)
```

```
# Non
spam (1)
```

- Juste avant la parenthèse ouvrante qui introduit la liste des paramètres d'une fonction :

```
# Oui
dict['key'] = list[index]
```

```
# Non
dict ['key'] = list [index]
```

# La PEP 8 : des conventions précises

- Plus d'un espace autour de l'opérateur d'affectation=(ou autre) pour l'aligner avec une autre instruction :

```
# Oui
x = 1
y = 2
long_variable = 3

# Non
x      = 1
y      = 2
long_variable = 3
```

- Toujours entourer les opérateurs suivants d'un espace (un avant le symbole, un après) :

- affectation :=,+=,-=, etc. ;
- comparaison :<,>,<=, ...,in,not in,is,is not;
- booléens :and,or,not;
- arithmétiques :+,-,\* , etc.

```
# Oui
i = i + 1
submitted += 1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)

# Non
i=i+1
submitted +=1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

# La PEP 8 : des conventions précises

Attention : n'utilisez pas d'espaces autour du signe=si c'est dans le contexte d'un paramètre ayant une valeur par défaut (définition d'une fonction) ou d'un appel de paramètre (appel de fonction).

```
# Oui
def fonction(parametre=5):
    ...
fonction(parametre=32)
```

```
# Non
def fonction(parametre = 5):
    ...
fonction(parametre = 32)
```

- Il est déconseillé de mettre plusieurs instructions sur une même ligne :

```
# Oui
if foo == 'blah':
    do_bla_bla_thing()
do_one()
do_two()
do_three()
```

```
# Plutôt que
if foo == 'blah': do_bla_bla_thing()
do_one(); do_two(); do_three()
```

Les commentaires qui contredisent le code sont pires qu'une absence de commentaire. Lorsque le code doit changer, faites passer parmi vos priorités absolues la mise à jour des commentaires !

# La PEP 8 : des conventions précises

- Les commentaires doivent être des phrases complètes, commençant par une majuscule. Le point terminant la phrase peut être absent si le commentaire est court.
- Si vous écrivez en anglais, les règles de langue définies par Strunk and White dans « *The Elements of Style* » s'appliquent.
- À l'attention des codeurs non-anglophones : s'il vous plaît, écrivez vos commentaires en anglais, sauf si vous êtes sûrs à 120% que votre code ne sera jamais lu par quelqu'un qui ne comprend pas votre langue (ou que vous ne parlez vraiment pas un mot d'anglais !).

# La PEP 8 : des conventions précises

## Conventions de nommage

### Noms à éviter

N'utilisez jamais les caractères suivants de manière isolée comme noms de variables :l(L minuscule),O(o majuscule) et l (i majuscule). L'affichage de ces caractères dans certaines polices fait qu'ils peuvent être aisément confondus avec les chiffres 0 ou 1.

### Noms des modules et packages

Les modules et packages doivent avoir des noms courts, constitués de lettres minuscules. Les noms de modules peuvent contenir des signes\_(souligné). Bien que les noms de packages puissent également en contenir, la PEP 8 nous le déconseille.

### Noms de classes

Sans presque aucune exception, les noms de classes utilisent la convention suivante : la variable est écrite en minuscules, exceptée la première lettre de chaque mot qui la constitue. Par exemple :MaClasse.

### Noms d'exceptions

Les exceptions étant des classes, elles suivent la même convention. En anglais, si l'exception est une erreur, on fait suivre le nom du suffixeError(vous retrouvez cette convention dansSyntaxError,IndexError...).

### Noms de variables, fonctions et méthodes

La même convention est utilisée pour les noms de variables (instances d'objets), de fonctions ou de méthodes : le nom est entièrement écrit en minuscules et les mots sont séparés par des signes soulignés (\_). Exemple :nom\_de\_fonction.

### Constantes

Les constantes doivent être écrites entièrement en majuscules, les mots étant séparés par un signe souligné (\_). Exemple :NOM\_DE\_MA\_CONSTANTE.

# La PEP 8 : des conventions précises

## Conventions de programmation

### Comparaisons

Les comparaisons avec des singletons (comme `None`) doivent toujours se faire avec les opérateurs `is` et `is not`, jamais avec les opérateurs `==` ou `!=`.

```
# Oui
if objet is None:
...
# Non
if objet == None:
...
```

Quand cela est possible, utilisez l'instruction `if objet:` si vous voulez dire `if objet is not None: .`

La vérification du type d'un objet doit se faire avec la fonction `isinstance`:

```
# Oui
if isinstance(variable, str):
...
# Non
if type(variable) == str:
...
```

Quand vous comparez des séquences, utilisez le fait qu'une séquence vide est `False`.

```
if liste: # La liste n'est pas vide
```

# La PEP 8 : des conventions précises

Enfin, ne comparez pas des booléens à True ou False:

```
# Oui
if boolean: # Si boolean est
vrai
...
if not boolean: # Si
boolean n'est pas vrai
...

# Non
if boolean == True:
...

# Encore pire
if boolean is True:
...
```

## Conclusion

Voilà pour la PEP 8 ! Elle contient beaucoup de conventions et toutes ne figurent pas dans cette section. Celles que j'ai présentées ici, dans tous les cas, sont moins détaillées. Je vous invite donc à faire un tour du côté du texte original si vous désirez en savoir plus.

# La PEP 257 : de belles documentations

Nous allons nous intéresser à présent à la PEP 257 qui définit d'autres conventions concernant la documentation via les docstrings. Consultez-la sur le site de Python.

```
def fonction(parametre1, parametre2):
    """Documentation de la fonction."""

```

La ligne 2 de ce code, que vous avez sans doute reconnue, est une docstring. Nous allons voir quelques conventions autour de l'écriture de ces docstrings (comment les rédiger, qu'y faire figurer, etc.).

Une fois de plus, je vais prendre quelques libertés avec le texte original de la PEP. Je ne vous proposerai pas une traduction complète de la PEP mais je reviendrai sur les points que je considère importants.

## Qu'est-ce qu'une docstring ?

La docstring (chaîne de documentation, en français) est une chaîne de caractères placée juste après la définition d'un module, d'une classe, fonction ou méthode. Cette chaîne de caractères devient l'attribut spécial `__doc__` de l'objet.

```
fonction.__doc__
'Documentation de la fonction.'
```

Tous les modules doivent être documentés grâce aux docstrings. Les fonctions et classes exportées par un module doivent également être documentées ainsi. Cela vaut aussi pour les méthodes publiques d'une classe (y compris le constructeur `__init__`). Un package peut être documenté via une docstring placée dans le fichier `__init__.py`.

Pour des raisons de cohérence, utilisez toujours des guillemets triples `"""` autour de vos docstrings. Utilisez `"""chaîne de documentation"""` si votre chaîne comporte des anti-slash\.

On peut trouver les docstrings sous deux formes :

- sur une seule ligne ;
- sur plusieurs lignes.

# La PEP 257 : de belles documentations

## Notes

- Les guillemets triples sont utilisés même si la chaîne tient sur une seule ligne. Il est plus simple de l'étendre par la suite dans ces conditions.
- Les trois guillemets """ fermant la chaîne sont sur la même ligne que les trois guillemets qui l'ouvrent. Ceci est préférable pour une docstring d'une seule ligne.
- Il n'y a aucun saut de ligne avant ou après la docstring.
- La chaîne de documentation est une phrase, elle se termine par un point..
- La docstring sur une seule ligne ne doit pas décrire la signature des paramètres à passer à la fonction/méthode, ou son type de retour. N'écrivez pas :

```
def fonction(a, b):
    """fonction(a, b) -> list"""
```

Cette syntaxe est uniquement valable pour les fonctions C (comme les built-ins). Pour les fonctions Python, l'introspection peut être utilisée pour déterminer les paramètres attendus. L'introspection ne peut cependant pas être utilisée pour déterminer le type de retour de la fonction/méthode. Si vous voulez le préciser, incluez-le dans la docstring sous une forme explicite :

```
"""Fonction faisant cela et renvoyant une liste."""
```

## Les docstrings sur plusieurs lignes

Les docstrings sur plusieurs lignes sont constituées d'une première ligne résumant brièvement l'objet (fonction, méthode, classe, module), suivie d'un saut de ligne, suivi d'une description plus longue. Respectez autant que faire se peut cette convention : une ligne de description brève, un saut de ligne puis une description plus longue.

# La PEP 257 : de belles documentations

La première ligne de la docstring peut se trouver juste après les guillemets ouvrant la chaîne ou juste en-dessous.  
Dans tous les cas, le reste de la docstring doit être indenté au même niveau que la première ligne :

```
class MaClasse:  
    def __init__(self, ...):  
        """Constructeur de la classe MaClasse  
  
        Une description plus longue...  
        sur plusieurs lignes...  
  
        """
```

Insérez un saut de ligne avant et après chaque docstring documentant une classe.

La docstring d'un module doit généralement dresser la liste des classes, exceptions et fonctions, ainsi que des autres objets exportés par ce module (une ligne de description par objet). Cette ligne de description donne généralement moins d'informations sur l'objet que sa propre documentation. La documentation d'un package (la docstring se trouvant dans le fichier `__init__.py`) doit également dresser la liste des modules et sous-packages qu'il exporte.

La documentation d'une fonction ou méthode doit décrire son comportement et documenter ses arguments, sa valeur de retour, ses effets de bord, les exceptions qu'elle peut lever et les restrictions concernant son appel (quand ou dans quelles conditions appeler cette fonction). Les paramètres optionnels doivent également être documentés.

```
def complexe(reel=0.0, image=0.0):  
    """Forme un nombre complexe.  
  
    Paramètres nommés :  
    reel -- la partie réelle (0.0 par défaut)  
    image -- la partie imaginaire (0.0 par défaut)  
  
    """  
    if image == 0.0 and reel == 0.0: return complexe_zero  
    """
```

# La PEP 257 : de belles documentations

La documentation d'une classe doit, de même, décrire son comportement, documenter ses méthodes publiques et ses attributs.

Le BDFL nous conseille de sauter une ligne avant de fermer nos docstrings quand elles sont sur plusieurs lignes. Les trois guillemets fermant la docstring sont ainsi sur une ligne vide par ailleurs.

```
def fonction():
    """Documentation brève sur une ligne.

    Documentation plus longue...

    """
```

# Docstring

- Docstring: Documentation String
- Les différents formats de Docstring:
  - Epytext
  - reST
  - Google

Epytext :

```
"""Docstring de style Epytext

@param param1: un premier paramètre
@param param2: un autre paramètre
@return: description de ce qui est retourné
"""
```

reST:

```
"""Docstring de type reST

:param param1: un premier paramètre
:param param2: un autre paramètre
:returns: description de ce qui est retourné
"""
```

Google:

```
"""Docstring de style Google

Args:
    param1: un premier paramètre
    param2: un autre paramètre

Returns:
    Description de ce qui est retourné
"""
```

Pour finir et bien  
continuer

# Pour finir et bien continuer

La fin de ce cours sur Python approche. Mais si ce langage vous a plu, vous aimeriez probablement concrétiser vos futurs projets avec lui. Je vous donne ici quelques indications qui devraient vous y aider.

Ce sera cependant en grande partie à vous d'explorer les pistes que je vous propose. Vous avez à présent un bagage suffisant pour vous lancer à corps perdu dans un projet d'une certaine importance, tant que vous vous en sentez la motivation.

Nous allons commencer par voir quelques-unes des ressources disponibles sur Python, pour compléter vos connaissances sur ce langage.

Nous verrons ensuite plusieurs bibliothèques tierces spécialisées dans certains domaines, qui permettent par exemple de réaliser des interfaces graphiques.

# Quelques références

Dans cette section, je vais surtout parler des ressources officielles que l'on peut trouver sur [le site de Python](#).

Il en existe bien entendu d'autres, certaines d'entre elles sont en français. Mais les ressources les plus à jour concernant Python se trouvent sur le site de Python lui-même.

En outre, les ressources mises à disposition sont clairement expliquées et détaillées avec assez d'exemples pour comprendre leur utilité. Elles n'ont qu'un inconvénient : elles sont en anglais. Mais c'est le cas de la majeure partie des documentations en programmation et il faudra bien envisager, un jour où l'autre, de s'y mettre pour aller plus loin !

## La documentation officielle

Nous avons déjà parlé de la documentation officielle dans ces pages. Nous allons maintenant voir comment elle se décompose exactement.

Commencez par vous rendre sur le site de Python.

Dans le menu de navigation, vous pourrez trouver plusieurs liens (notamment le lien de téléchargement, [DOWNLOAD](#), sur lequel vous avez probablement cliqué pour obtenir Python). Il s'y trouve également le lien [DOCUMENTATION](#) et c'est sur celui-ci que je vous invite à cliquer à présent.

Dans la nouvelle page qui s'affiche figurent deux éléments intéressants :

Sous le lien DOCUMENTATION du menu, il y a à présent un sous-menu contenant les liens Current Docs, License, Help, etc.

La partie centrale de la page contient maintenant des informations sur les documentations de Python, classées suivant les versions. Par défaut, seules les deux versions les plus récentes de Python (dans les branches 2.X et 3.X) sont visibles mais vous pouvez afficher toutes les versions en cliquant sur le lien the complete list of documentation by Python version.

Nous allons d'abord nous intéresser au sous-menu.

## L'index des PEP (Python Enhancement Proposal)

Dans ce sous-menu, vous pouvez également trouver un lien intitulé PEP Index. Si vous cliquez dessus, vous accédez à un tableau, ou plutôt à un ensemble de tableaux reprenant les PEPs classées par catégories. Comme vous pouvez le constater, il y en a un paquet et, dans ce livre, je n'ai pu vous en présenter que quelques-unes. Libre à vous de parcourir cet index et de vous pencher sur certaines des PEP en fonction des sujets qui vous intéressent plus particulièrement.

# Quelques références

## La documentation par version

À présent, revenez sur la page de documentation de Python.

Cliquez sur le lien correspondant à la version de Python installée sur votre machine (Browse Python 3.4.0 Documentation pour moi).

Sur la nouvelle page qui s'affiche sous vos yeux, vous trouvez les grandes catégories de la documentation. En voici quelques-unes :

**Tutorial** : le tutoriel. Selon toute probabilité, les bases de Python vous sont acquises ; il est néanmoins toujours utile d'aller faire un tour sur cette page pour consulter la table des matières.

**Library Reference** : la référence de la bibliothèque standard, nous reviendrons un peu plus loin sur cette page. Le conseil donné me paraît bon à suivre : Keep this under your pillow, c'est-à-dire, « gardez-la sous votre oreiller ».

**Language Reference** : cette page décrit d'une façon très explicite la syntaxe du langage.

**Python HOWTOs** : une page regroupant des documents d'aide traitant de sujets très précis, par exemple comment bien utiliser les sockets.

Vous pouvez aussi trouver un classement par index que je vous laisse découvrir. Vous pourrez y voir, notamment, le lien permettant d'afficher la table des matières complète de la documentation. Vous y trouverez également un glossaire, utile dans certains cas.

## La référence de la bibliothèque standard

Vous vous êtes peut-être déjà rendus sur cette page. Je l'espère, en vérité. Elle comporte la documentation des types prédéfinis par Python, des fonctions *built-in* et exceptions, mais aussi des modules que l'on peut trouver dans la bibliothèque standard de Python. Ces modules sont classés par catégories et il est assez facile (et parfois très utile) de survoler la table des matières pour savoir ce que Python nous permet de faire sans installer de bibliothèque tierce.

C'est déjà pas mal, comme vous pouvez le voir !

Cela dit, il existe certains cas où des bibliothèques tierces sont nécessaires. Nous allons voir quelques-uns de ces cas dans la suite de ce chapitre, ainsi que quelques bibliothèques utiles dans ces circonstances.

# Des bibliothèques tierces

La bibliothèque standard de Python comporte déjà beaucoup de modules et de fonctionnalités. Mais il arrive, pour certains projets, qu'elle ne suffise pas.

Si vous avez besoin de créer une application avec une interface graphique, la bibliothèque standard vous propose un module appelé **tkinter**. Il existe toutefois d'autres moyens de créer des interfaces graphiques, en faisant appel à des bibliothèques tierces.

Ces bibliothèques se présentent comme des packages ou modules que vous installez pour les rendre accessibles depuis votre interpréteur Python.

À l'heure où j'écris ces lignes, toutes les bibliothèques dont je parle ne sont pas nécessairement compatibles avec Python 3.X.

Les développeurs desdites bibliothèques ont généralement comme projet, à plus ou moins long terme, de passer leur code en Python 3.X. Si certains ont déjà franchi le pas, d'autres attendent encore et ce travail est plus ou moins long en fonction des dépendances de la bibliothèque.

Bref, tout cela évolue et si je vous dis que telle bibliothèque n'est pas compatible avec Python 3.X, il faudra attendre pour l'instant. À l'heure où vous lisez ces lignes, il est bien possible qu'une version compatible soit parue. Le changement se fait, lentement mais sûrement.

Ceci étant posé, examinons quelques bibliothèques tierces.

## Pour créer une interface graphique

Nous avons parlé de **tkinter**. Il s'agit d'un module disponible par défaut dans la bibliothèque standard de Python. Il se base sur la bibliothèque Tk et permet de développer des interfaces graphiques.

Il est cependant possible que ce module ne corresponde pas à vos besoins. Il existe plusieurs bibliothèques tierces qui permettent de développer des interfaces graphiques, parfois en proposant quelques bonus.

En voici trois parmi d'autres :

**PyQT** : une bibliothèque permettant le développement d'interfaces graphiques, actuellement en version 4. En outre, elle propose plusieurs packages gérant le réseau, le SQL (bases de données), un kit de développement web... et bien d'autres choses. Soyez vigilants cependant : PyQt est distribuée sous plusieurs licences, commerciales ou non. Vous devrez tenir compte de ce fait si vous commencez à l'utiliser.

**PyGTK** : comme son nom l'indique, c'est une bibliothèque faisant le lien entre Python et la bibliothèque GTK / GTK+. Elle est distribuée sous licence LGPL.

**wx Python** : une bibliothèque faisant le lien entre Python et la bibliothèque WxWidget.

Ces informations ne vous permettent pas de faire un choix immédiat entre telle ou telle bibliothèque, j'en ai conscience. Aussi, je vous invite à aller jeter un coup d'œil du côté des sites de ces différents projets. Ces trois bibliothèques ont l'avantage d'être multiplateformes et, généralement, assez simples à apprendre. En fonction de vos besoins, vous vous tournez plutôt vers l'une ou l'autre, mais je ne peux certainement pas vous aider dans ce choix. Je vous invite donc à rechercher par vous-mêmes si vous êtes intéressés.

# Des bibliothèques tierces

## Dans le monde du Web

Il existe là encore de nombreuses bibliothèques, bien que je n'en présente ici que deux. Elles permettent de créer des sites web et concurrencent des langages comme PHP. Django

La première, dont vous avez sans doute entendu parler auparavant, est Django.

Django est une bibliothèque, ou plutôt un framework, permettant de développer votre site dynamique en Python. Il propose de nombreuses fonctionnalités que vous trouverez, je pense, aussi puissantes que flexibles si vous prenez le temps de vous pencher sur le site du projet.

À l'heure où j'écris ces lignes, certains développeurs tentent de proposer des patches pour adapter Django à Python 3. L'équipe du projet, cependant, ne prévoit pas dans l'immédiat de développer de branche pour Python 3.

Si vous tenez réellement à Django et qu'aucune version stable n'existe encore sous Python 3 à l'heure où vous lisez ces lignes, je vous encourage à tester cette bibliothèque sous Python 2.X. Rassurez-vous, vous n'aurez pas à apprendre toute la syntaxe pour programmer dans ce langage : la liste des changements est très clairement affichée sur le site de Python et ceux-ci ne représentent pas un obstacle insurmontable pour qui est motivé !

## CherryPy

[CherryPy](#) est une bibliothèque permettant de construire tout votre site en Python. Elle n'a pas la même vocation que Django puisqu'elle permet de créer la hiérarchie de votre site dynamiquement mais vous laisse libres des outils à employer pour faire quelque chose de plus complexe.

## Un peu de réseau

Pour finir, nous allons parler d'une bibliothèque assez connue, appelée [Twisted](#).

Cette bibliothèque est orientée vers le réseau. Elle prend en charge de nombreux protocoles de communication réseau (TCP et UDP, bien entendu, mais aussi HTTP, SSH et de nombreux autres). Si vous voulez créer une application utilisant l'un de ces protocoles, Twisted pourrait être une bonne solution.

Là encore, je vous invite à jeter un coup d'œil sur le site du projet pour plus d'informations. À l'heure où j'écris, Twisted n'est utilisable que sous la branche 2.X de Python. Cependant, on peut trouver une future version, en cours d'élaboration, portant Twisted sur la branche 3.X.

# Des bibliothèques tierces

## Dans le monde du Web

Il existe là encore de nombreuses bibliothèques, bien que je n'en présente ici que deux. Elles permettent de créer des sites web et concurrencent des langages comme PHP. Django

La première, dont vous avez sans doute entendu parler auparavant, est Django.

Django est une bibliothèque, ou plutôt un framework, permettant de développer votre site dynamique en Python. Il propose de nombreuses fonctionnalités que vous trouverez, je pense, aussi puissantes que flexibles si vous prenez le temps de vous pencher sur le site du projet.

À l'heure où j'écris ces lignes, certains développeurs tentent de proposer des patches pour adapter Django à Python 3. L'équipe du projet, cependant, ne prévoit pas dans l'immédiat de développer de branche pour Python 3.

Si vous tenez réellement à Django et qu'aucune version stable n'existe encore sous Python 3 à l'heure où vous lisez ces lignes, je vous encourage à tester cette bibliothèque sous Python 2.X. Rassurez-vous, vous n'aurez pas à apprendre toute la syntaxe pour programmer dans ce langage : la liste des changements est très clairement affichée sur le site de Python et ceux-ci ne représentent pas un obstacle insurmontable pour qui est motivé !

## CherryPy

[CherryPy](#) est une bibliothèque permettant de construire tout votre site en Python. Elle n'a pas la même vocation que Django puisqu'elle permet de créer la hiérarchie de votre site dynamiquement mais vous laisse libres des outils à employer pour faire quelque chose de plus complexe.

## Un peu de réseau

Pour finir, nous allons parler d'une bibliothèque assez connue, appelée [Twisted](#).

Cette bibliothèque est orientée vers le réseau. Elle prend en charge de nombreux protocoles de communication réseau (TCP et UDP, bien entendu, mais aussi HTTP, SSH et de nombreux autres). Si vous voulez créer une application utilisant l'un de ces protocoles, Twisted pourrait être une bonne solution.

Là encore, je vous invite à jeter un coup d'œil sur le site du projet pour plus d'informations. À l'heure où j'écris, Twisted n'est utilisable que sous la branche 2.X de Python. Cependant, on peut trouver une future version, en cours d'élaboration, portant Twisted sur la branche 3.X.

# Pour conclure

Ce ne sont là que quelques bibliothèques tierces, il en existe de nombreuses autres, certaines dédiées à des projets très précis. Je vous invite à faire des recherches plus avancées si vous avez des besoins plus spécifiques. Vous pouvez commencer avec la liste de bibliothèques qui se trouve ci-dessus, avec les réserves suivantes :

- Je ne donne que peu d'informations sur chaque bibliothèque et elles ne s'accordent peut-être plus avec celles disponibles sur le site du projet. En outre, la documentation de chaque bibliothèque reste et restera, dans tous les cas, une source plus sûre et actuelle.
- Ces projets évoluent rapidement. Il est fort possible que les informations que je fournis sur ces bibliothèques ne soient plus vraies à l'heure où vous lisez ces lignes. Pour mettre à jour ces informations, il n'y a qu'une seule solution imparable : allez sur le site du projet !

Une dernière petite parenthèse avant de vous quitter : je me suis efforcé de présenter, tout au long de ce livre, des données utiles et à jour sur le langage de programmation Python, dans sa branche 3.X. Il vous reste encore de nombreuses choses à découvrir sur le langage et ses bibliothèques, mais vous êtes désormais capables de voler de vos propres ailes. Bonne route ! ;-)

# Installer des packages supplémentaires avec PIP

# Installer des packages supplémentaires avec PIP

## A quoi sert pip ?

PIP (Pip Installs Packages qui est un acronyme récursif, il y a des fous partout) est un petit utilitaire qu'on appelle 'gestionnaire de paquets' et qui nous permet d'installer des 'paquets' (packages en anglais).

On peut donc l'utiliser pour télécharger facilement en ligne de commande des packages qui sont hébergés sur le site <http://pypi.org>.

Avec PIP, c'est un monde de possibilités qui s'offre à vous et qui vous permettra de télécharger des librairies bien connues comme Django, requests, BeautifulSoup et j'en passe.

**Petit point à noter :** j'utilise les termes 'module', 'librairie' et 'package' pour parler des 'paquets' qu'on peut installer (packages en anglais).

Ces trois termes sont interchangeables, car un package peut parfois consister en une seul fichier (on parle donc dans ce cas de module), et le terme librairie quant à lui est souvent utilisé pour parler de packages.

# Installer des packages supplémentaires avec PIP

Pour utiliser PIP sous windows:

- Git bash
- Depuis PyCharm, Visual Studio Code, GitBash ou cmder

# Installer des packages supplémentaires avec PIP

## Chercher des modules sur PyPI et avec pip

Site web: <http://pypi.org>

# Installer des packages supplémentaires avec PIP

## Installer un package avec pip

Pour connaitre la version de pip:

```
-----  
pip --version  
pip 20.2.3 from C:\Users\MOTTIER LUCIE\AppData\Roaming\Python\Python38\site-packages\pip (python 3.8)  
⇒ Nous obtenons la version de pip 20.2.3 et pour qu'elle version de python ici 3.8
```

Pour mettre pip à jour:

```
-----  
pip3.8 install –upgrade pip # on upgrade pip pour la version de python 3.8
```

Pour installer un package:

```
-----  
pip3.8 install request # install le package request pour la version de python 3.8
```

# Installer des packages supplémentaires avec PIP

Lister les packages installés avec pip

```
pip list
```

Désinstaller un package avec pip

```
pip uninstall requests
```

# Maitriser le système de versioning des paquets

Le moyen le plus simple pour installer une version de paquet Python consiste à utiliser l'opérateur == . Par exemple, pip install requests==2.1.0 installera, comme vous vous en doutez, la version 2.1.0 . Il existe toutefois plusieurs façons différentes de préciser la version du paquet. Par exemple :

pip install requests~=2.2 installera la version la plus élevée disponible au-dessus de 2.2! , mais pas 3.0 ni les versions ultérieures.

pip install requests~=2.1.0 installera la version la plus élevée disponible au-dessus de 2.1.0 , mais pas la version 2.2.0 ni les versions ultérieures.

pip install requests>2.5.0 installera la version la plus élevée disponible au-dessus de 2.5.0 .

pip install "requests>2.4.0,<2.6.0" installera la version la plus élevée disponible supérieure à 2.4.0 , mais inférieure à 2.6.0 .

# Un fichier de dépendance

Imaginons à présent que le programme soit fini. Vous le publiez sur Github afin que toute personne intéressée puisse y accéder.

Comment ces dernières peuvent-elles connaître les différentes librairies utilisées dans le projet ? Elles n'auront pas accès à votre environnement virtuel. Bien sûr, elles peuvent regarder en haut des différents fichiers et chercher les imports. Mais cela est fastidieux et, surtout, il sera impossible de connaître les versions utilisées.

Les librairies qui ne sont pas standards, qui doivent par conséquent être installées, sont ce que nous appelons des dépendances. En effet, le projet est dépendant de leur installation et ne peut pas fonctionner sans.

Afin de remédier à ce souci, il existe une convention : créer un fichier **requirements.txt** qui liste les différentes librairies utilisées ainsi que leur version.

Créer un fichier de dépendances

Pypi peut le faire automatiquement pour vous ! Pour cela, exécutez la commande suivante.

```
pip freeze > requirements.txt
```

Cette commande "gèle" les librairies utilisées ainsi que leur version en les listant dans un document.

Si vous utilisez PowerShell, exécutez la commande suivante :

```
pip list > requirements.txt
```

Voici ce qui a été généré :

requirements.txt

```
appdirs==1.4.3
packaging==16.8
pyparsing==2.2.0
six==1.10.0
```

Cette commande est très utile mais peut s'avérer restrictive. Pip indique en effet toutes les librairies utilisées, y compris des librairies fondamentales telles que packaging ou six que vous n'avez pas besoin d'installer. C'est pourquoi ce que nous préconisons est d'ajouter à la main les dépendances explicitement requises par un projet.

requirements.txt

```
requests
pandas
```

# Un fichier de dépendance

## Installer des dépendances

Si un fichier requirements.txt existe déjà, vous pouvez facilement installer toutes les librairies en exécutant la commande suivante :

```
pip install -r requirements.txt
```

# QT Designer

- Une fois l'interface créer avec QT Designer, enregistrer le fichier ce qui donnera un fichier avec une extension ui
- Dans l'explorateur si vous êtes à l'endroit où mon\_fichier.ui est généré, taper dans le chemin de l'explorateur(sous les icônes) cmd. Cela ouvre une fenêtre dos directement au bonne emplacement
- Dans le terminal de pyCharm pyuic5 est disponible
- Puis dans la fenêtre dos tapez:  
`pyuic5 -x mon_fichier.ui -o mon_fichier.py`

# disutils.util

True

```
import os
import distutils.util

cur_dir = os.path.realpath(os.path.dirname(__file__))
fichier = os.path.join(cur_dir, "variable.txt")

with open(fichier, "r") as f:
    variable = f.read()

if variable:
    print("La variable est un boolean True")
<
vraie_variable = bool(distutils.util strtobool(variable))
print(type(vraie_variable))
print(vraie_variable)
if vraie_variable:
    print("La variable est un boolean True")
```

Ce que nous montre l'exemple ci-dessus, c'est que si on récupère le texte True ou False dans un fichier et que l'on stock dans une variable, la variable sera de type texte et non pas boolean. Pour avoir un boolean il faut la transformer avec le package disutils.util.

# Les environnements virtuels

<https://blog.teclado.com/python-virtual-environments-complete-guide/>



Working with Python virtual environments the complete guide.pdf

# Les environnements virtuels création d'un environnement virtuel

```
# Virtual Environments ("virtualenvs") keep
# your project dependencies separated.
# They help you avoid version conflicts
# between packages and different versions
# of the Python runtime.
# Before creating & activating a virtualenv:
# `python` and `pip` map to the system
# version of the Python interpréter
# (e.g. Python 2.7)$ which python/usr/local/bin/python
# Let's create a fresh virtualenv using
# another version of Python (Python 3):$ python3 -m venv ./venv
# A virtualenv is just a "Python
# environment in a folder":$ ls ./venv bin include lib pyvenv.cfg
# Activating a virtualenv configures the
# current shell session to use the python
# (and pip) commands from the virtualenv
# folder instead of the global environment:$ source ./venv/bin/activate
# Note how activating a virtualenv modifies
# your shell prompt with a little note
# showing the name of the virtualenv folder:(venv) $ echo "wee!"«
# With an active virtualenv, the `python`
# command maps to the interpreter binary
# *inside the active virtualenv*:(venv) $ which python/Users/dan/my-project/venv/bin/python3
# Installing new libraries and frameworks# with `pip` now installs them *into the# virtualenv
# sandbox*, leaving your global
# environment (and any other virtualenvs)
# completely unmodified (venv) $ pip install requests
# To get back to the global Python
# environment, run the following command:(venv) $ deactivate
# (See how the prompt changed back
# to "normal" again?)$ echo "yay!"«
# Deactivating the virtualenv flipped the
# `python` and `pip` commands back to
# the global environment:$ which python/usr/local/bin/python
```

# Les environnements virtuels

## création d'un environnement virtuel

```
# Creons un dossier pour travailler puis creons notre environnement virtuel
mkdir dossier_test
cd dossier_test

python -m venv env

ls
env/

cd env/
ls
Include/      Lib/          pyenv.cfg      Scripts/
cd Scripts
ls
activate      deactivate.bat    pip.exe        python.exe
activate.bat  easy_install.exe  pip3.7.exe    python.exe
activate.ps1   easy_install-3.7.exe  pip3.exe

# Activation de l'environnement
source activate
(env)

# Desactivation
deactivate
```

# Cmder

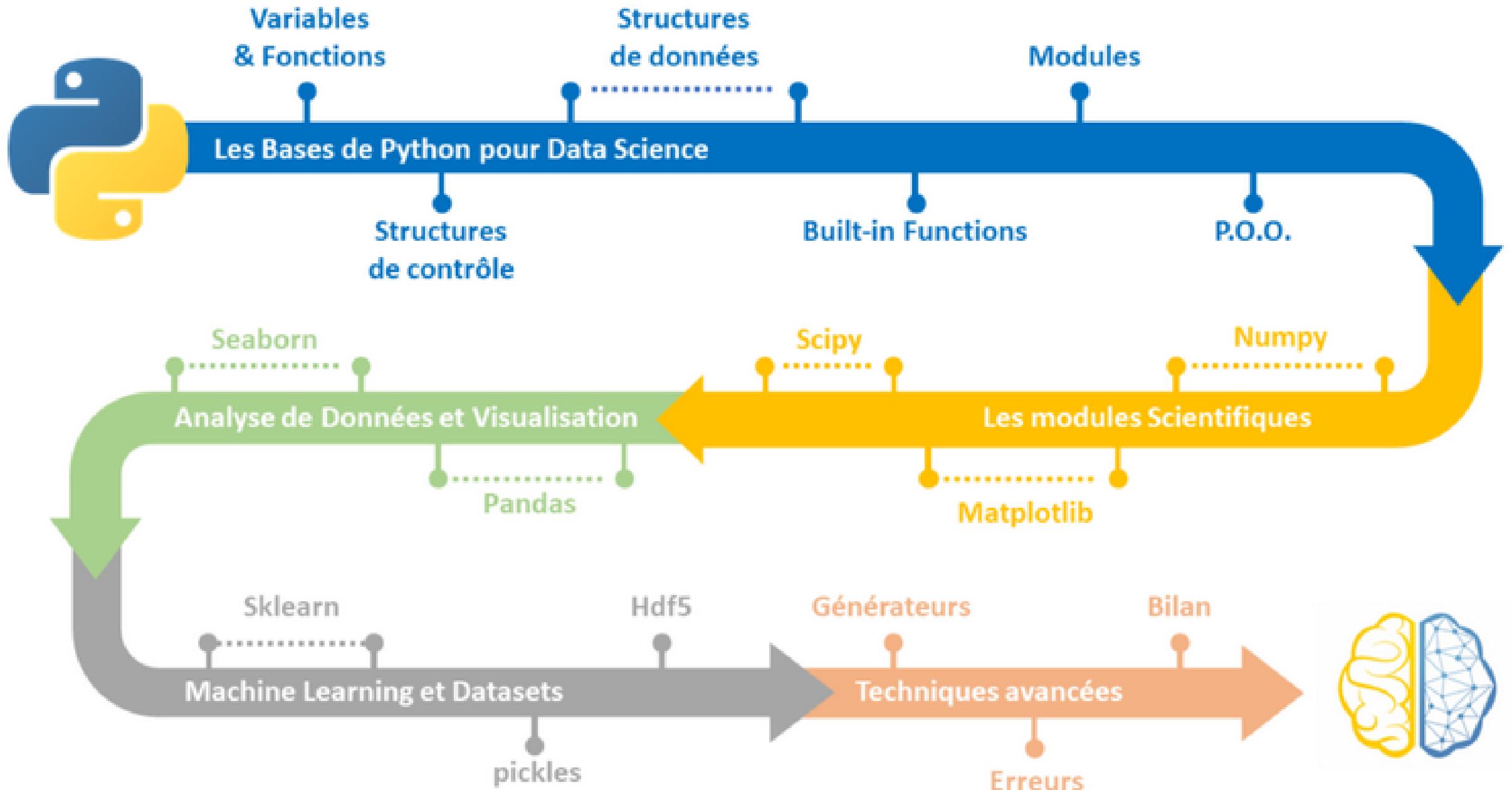
Apres la création de l'environnement virtuel il est nécessaire de mettre à jour l'environnement dans cmder:

1. Lancer cmder, vous devriez vous retrouver dans un dossier cmder/

```
MOTTIER LUCIE@DESKTOP-23A2N2I /c/Program Files/cmder
λ ls
bin/ Cmder.exe* config/ icons/ LICENSE opt/ vendor/ 'Version 1.3.16.1035'
MOTTIER LUCIE@DESKTOP-23A2N2I /c/Program Files/cmder
λ cd config/
λ vi user_profile.sh
alias python='/c/Program\ Files/Python39/python' # redefinir l'alias comme necessaire
```

# Machine Learnia

[https://www.youtube.com/playlist?list=PLO\\_fdPEVlfKqMDNmCFzQISI2H\\_nJcEDJq](https://www.youtube.com/playlist?list=PLO_fdPEVlfKqMDNmCFzQISI2H_nJcEDJq)



# SEQUENCES LISTES

<https://machinelearnia.com>

Liste =	'Paris'	'Berlin'	'Londres'	'Bruxelles'
(index)	0	1	2	3

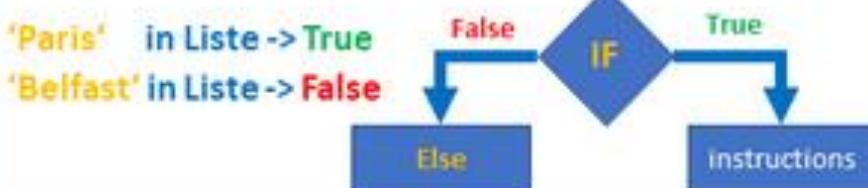
*Indexing:* Pour accéder à un élément.

Liste[#index] = élément

*Slicing:* Pour accéder à une mini-séquence

Liste[#début : #fin] = mini-séquence

## If/Else



## Boucle For

For i in Liste:

-> 'Paris' - 'Berlin' - 'Londres' - 'Bruxelles'

For index, valeur in enumerate(Liste):

-> 0 'Paris' - 1 'Berlin' - 2 'Londres' - 3 'Bruxelles'

For A, B in zip(Liste\_A, Liste\_B):

-> A0 B0 - A1 B1 - A2 B3 - ...

## Méthodes

Liste.sort(reverse=False)

'Berlin'	'Bruxelles'	'Londres'	'Paris'
0	1	2	3

Liste.count(truc)

'Paris'	'Berlin'	'Paris'	'Bruxelles'
0	1	2	3

= 2

len(liste)

'Paris'	'Berlin'	'Paris'	'Bruxelles'
0	1	2	3

= 4

Liste.append(truc)

'Paris'	'Berlin'	'Londres'	'Bruxelles'
0	1	2	3

'Madrid'

Liste.extend(liste)

'Paris'	'Berlin'	'Londres'	'Bruxelles'
0	1	2	3

'Madrid'

'Rome'

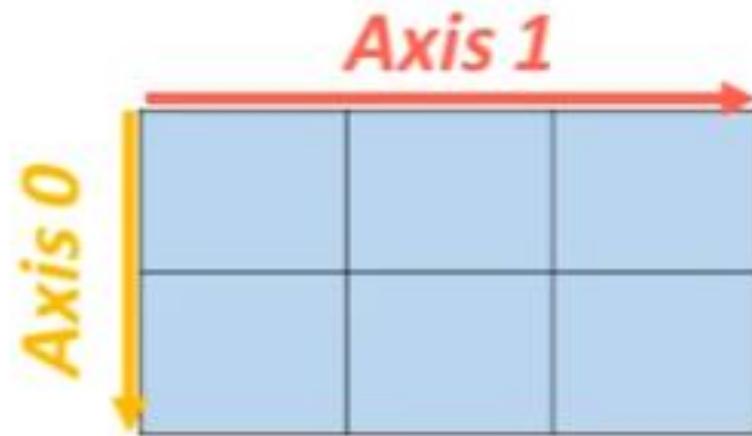
Liste.insert(index, truc)

'Paris'	'Berlin'	'Londres'	'Bruxelles'
0	1	2	3

'Madrid'

# NUMPY - AXES

2D array

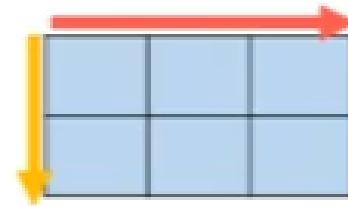


**shape = (2, 3)**

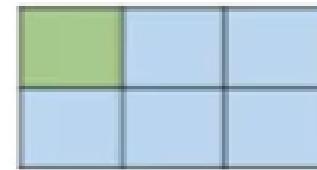


# NUMPY - INDEXING

A =



A[*ligne, colonne*]



A[0, 0]



A[0, 1]

```
import numpy as np  
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
A
```

```
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

A[0, 0]

I



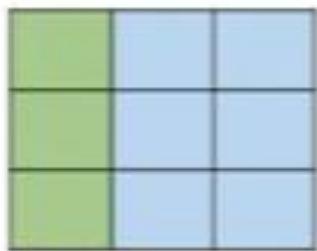
# NUMPY - SLICING

A =

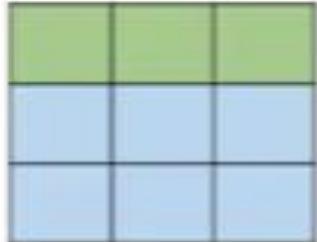


A[*début : fin*, *début : fin*]

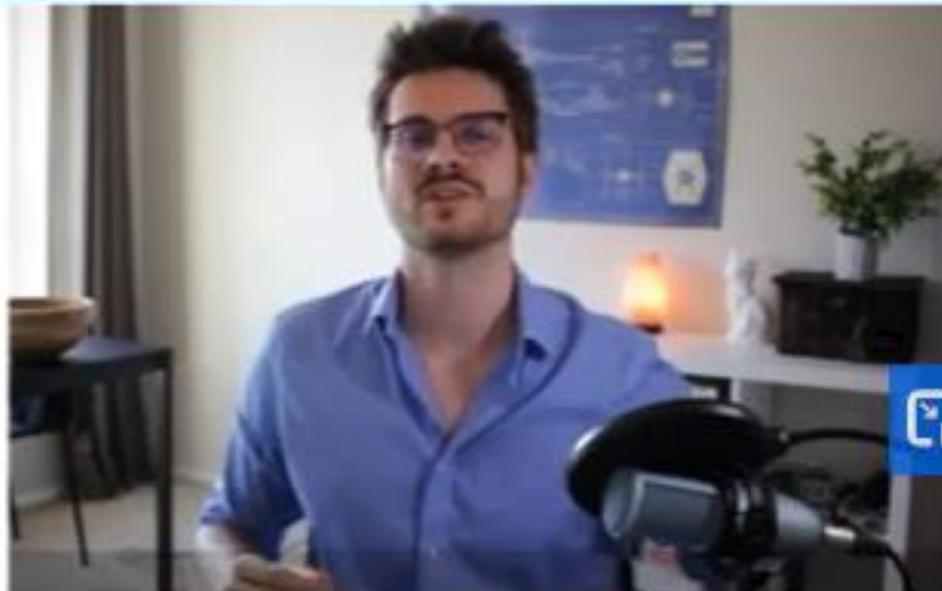
A[:, 0]



A[0, :]

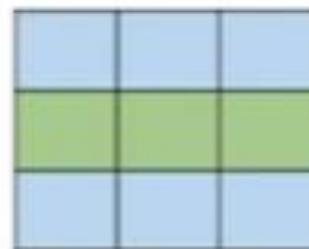


A[0]

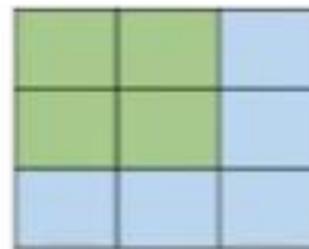


# NUMPY – SLICING

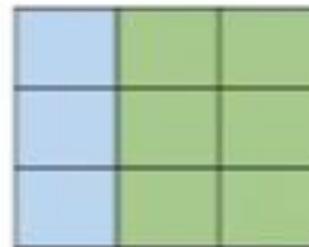
**A[*début : fin*, *début : fin*]**



**A[*1, :*]**



**A[*0:2, 0:2*]**



**A[*?, ?*]**

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
A
```

array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])

A[*1, 1*]

5

A[*2*]

array([7, 8, 9])



# NUMPY – MATHÉMATIQUES

axis 0	axis 1
5	0
3	7
3	9

A.sum(*axis=0*) =

5	0	3
3	7	9
=	=	=
8	7	12



A.sum(*axis=1*) =

5	0	3
3	7	9

= 8  
= 19



# NUMPY – METHODES NDARRAY

Quelques méthodes de ndarray:

A.sum(*axis*), A.cumsum(*axis*)

A.prod(*axis*), A.cumprod(*axis*)

A.min(*axis*), A.max(*axis*)

A.argmin(*axis*), A.argmax(*axis*)

A.sort(*axis*), A.argsort(*axis*)



# MATPLOTLIB CYCLE DE VIE

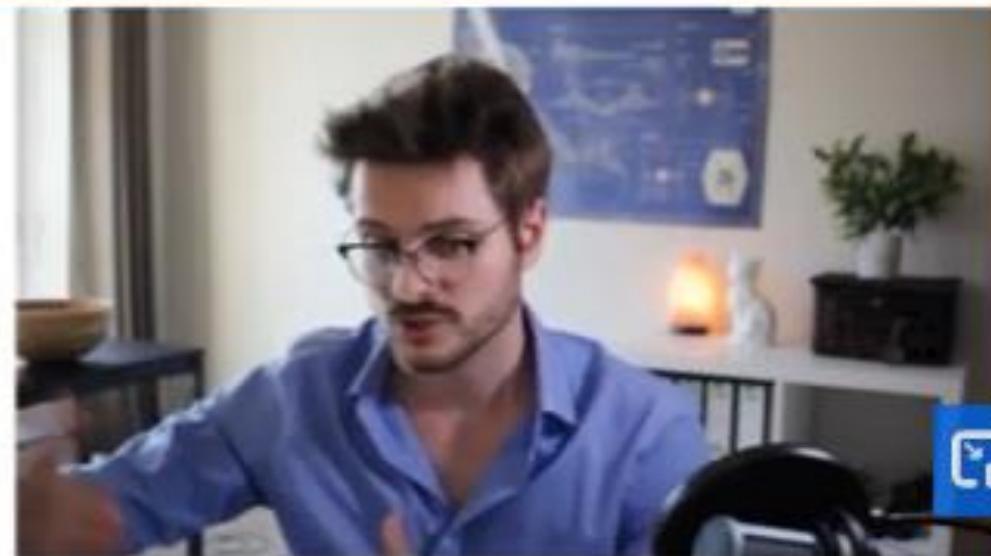
`plt.figure()` <- Début de la figure

`plt.plot(..., ...)`  
`plt.plot(..., ...)`  
`plt.xlabel('texte')`  
`plt.title('texte')`  
`plt.legend()`

Contenu

`plt.show()` <- Affiche la figure

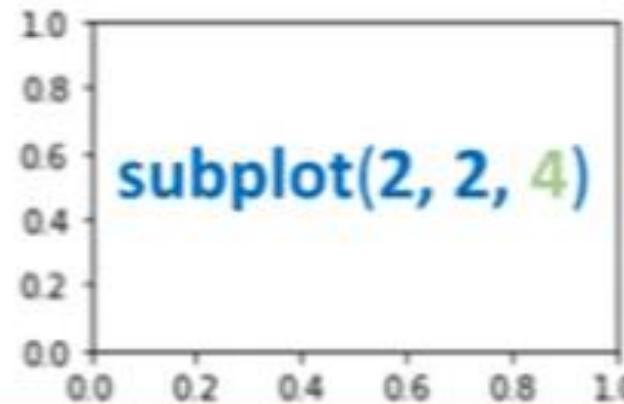
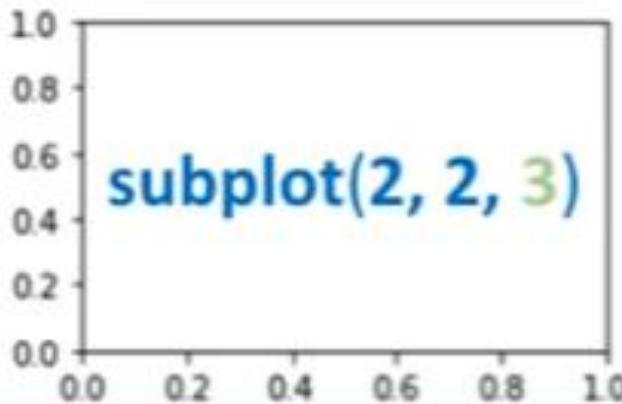
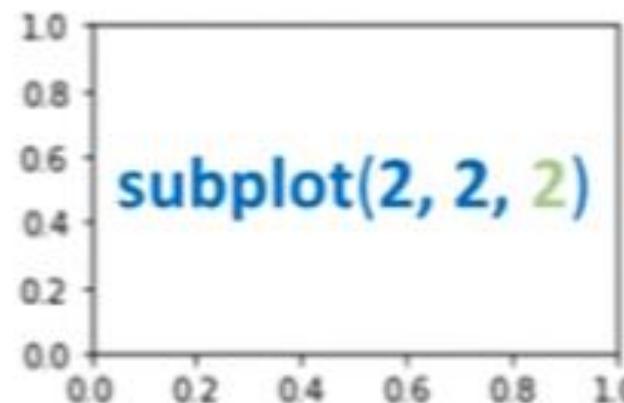
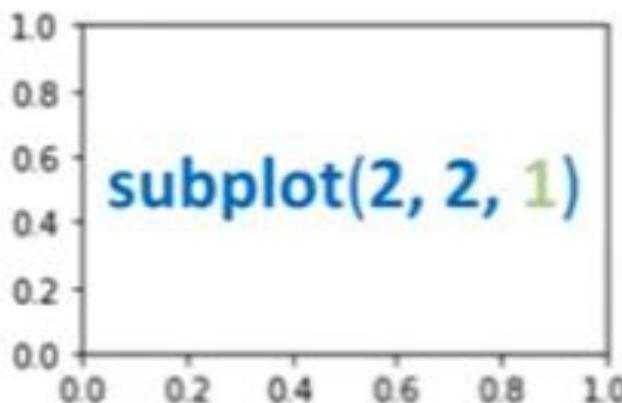
`plt.savefig('text.png')`



# MATPLOTLIB SUBPLOT

Une grille de graphiques :

`plt.subplot(lignes, colonnes, position)`



# MATPLOTLIB CYCLE DE VIE

`plt.figure()`    <- Début de la figure

`plt.subplot(2,1,1)`

`plt.plot(..., ...)`    }    Graphique 1  
...  
`plt.title()`

`plt.subplot(2,1,2)`

`plt.plot(..., ...)`    }    Graphique 2  
...  
`plt.title()`

`plt.show()`    <- Affiche la figure

`plt.figure()`

I



# MATPLOTLIB OBJECT ORIENTED

```
fig, ax = plt.subplots(n)
ax[0].plot(x, y)
ax[1].plot(x, y)
plt.show()
```

**fig** est un **objet**.

**ax** est un tableau **ndarray** qui  
contient des **objets**.

On utilise donc des **méthodes**.

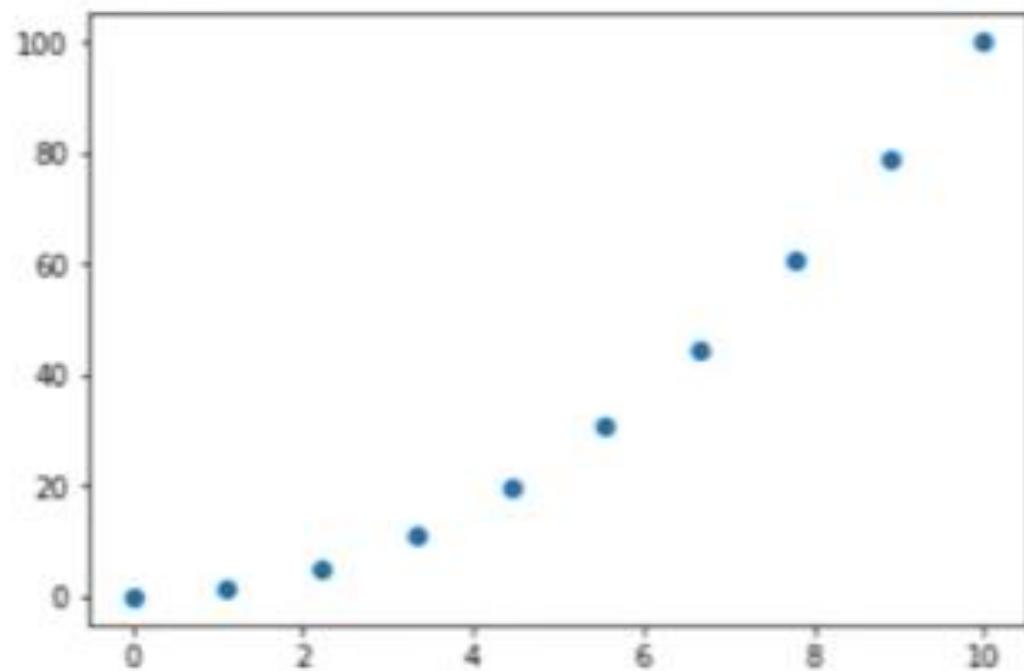
-> **Plus professional, pour les Devs**



# 1) Interpolation

```
In [4]: x = np.linspace(0, 10, 10)
y = x**2
plt.scatter(x, y)
```

```
Out[4]: <matplotlib.collections.PathCollection at 0x2b60e676f98>
```



```
In [3]: from scipy.interpolate import interp1d
```



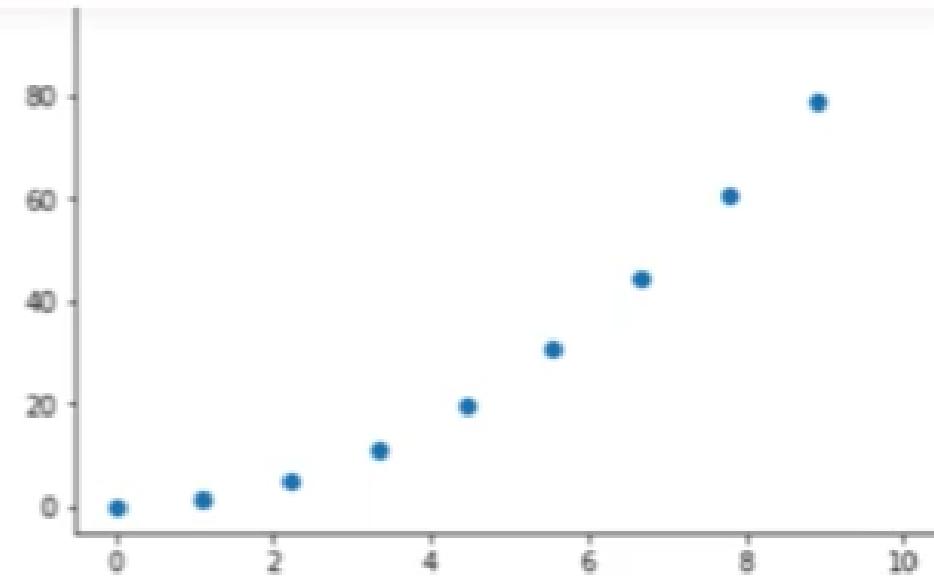
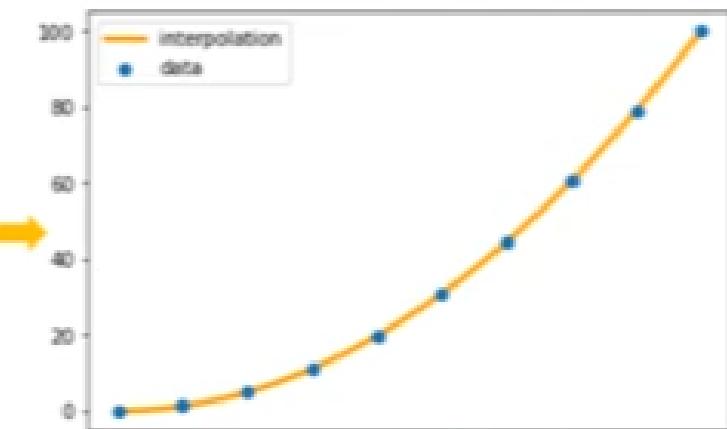
# Scipy Interpolate

Interp1d nous retourne  
une fonction :  $f(\dots) = \dots$

On peut utiliser cette fonction  
comme on veut :

`x = np.linspace(0, 10, 30)`

$f(x)$



```
from scipy.interpolate import interp1d
```

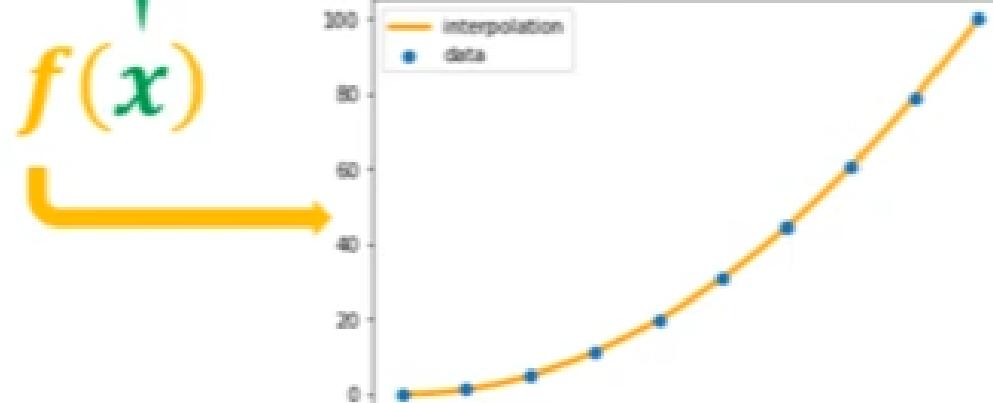
```
f = interp1d(x, y, kind='linear')
```



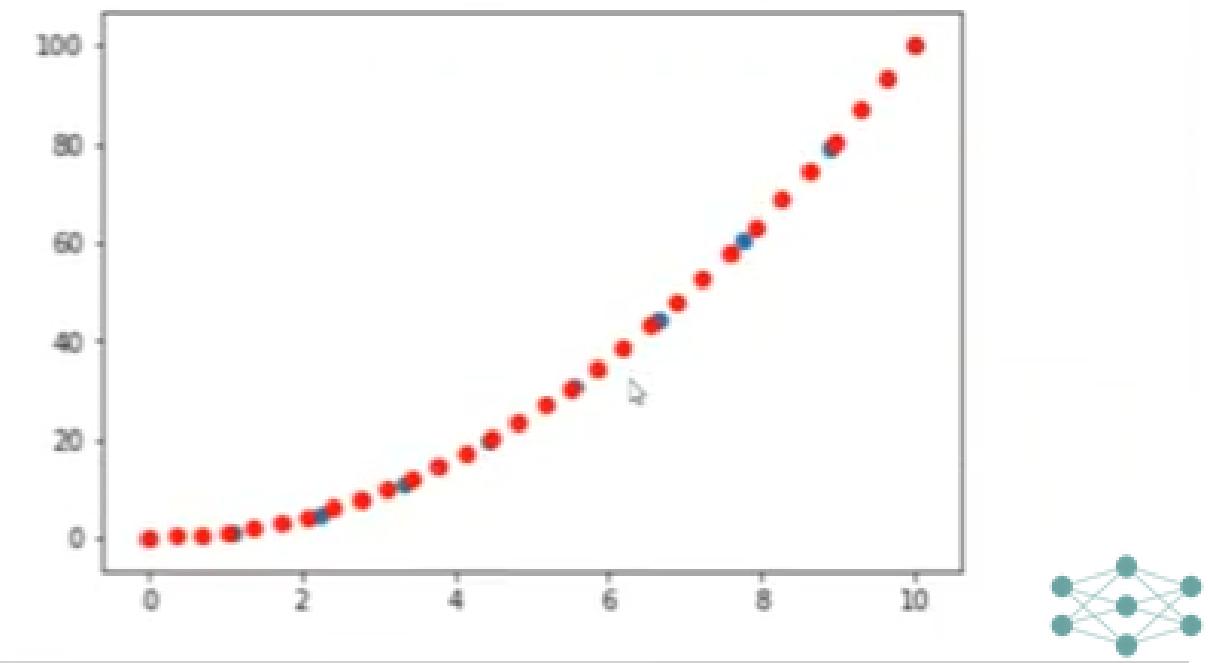
Interp1d nous retourne  
une fonction :  $f(\dots) = \dots$

On peut utiliser cette fonction  
comme on veut :

$x = np.linspace(0, 10, 30)$



```
f = interp1d(x, y, kind='linear')  
  
new_x = np.linspace(0, 10, 30)  
result = f(new_x)  
  
plt.scatter(x, y)  
plt.scatter(new_x, result, c='r')  
  
<matplotlib.collections.PathCollection at 0x2b60ee60160>
```



<https://docs.scipy.org/doc/scipy/reference/index.html>

# SEABORN

Les fonctions Seaborn ont presque toutes la même structure:

**sns.fonction(x, y, data, hue, size, style)**

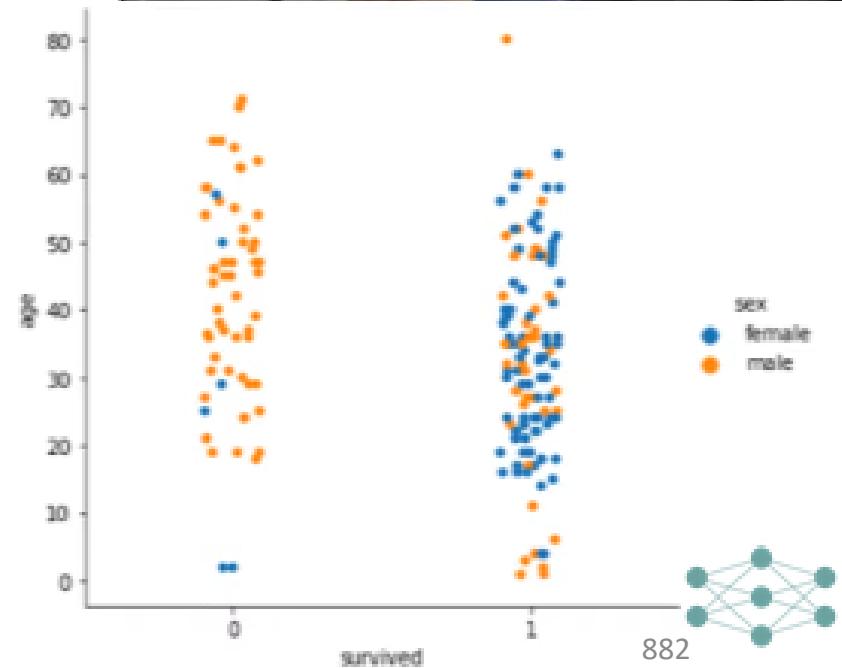
*Les données  
à afficher*

*Options de  
segmentation*

**Exemple:**

```
sns.catplot(x='survived', y='age', data=titanic, hue='sex')
```

<https://seaborn.pydata.org/tutorial.html>



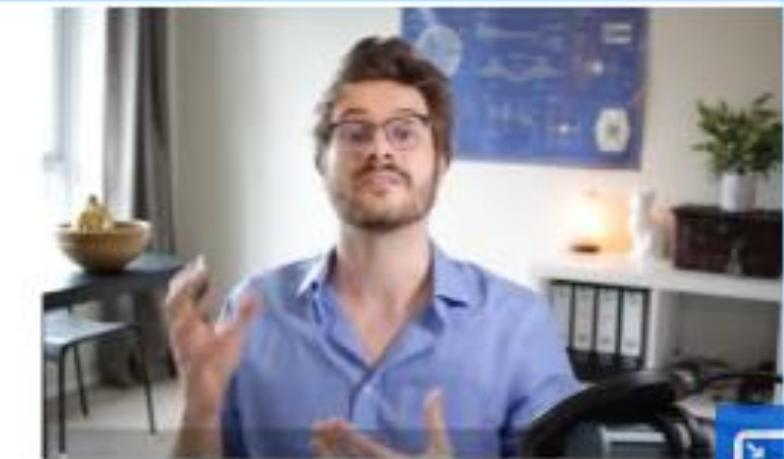
## Les fonctions les plus utiles

1. **Pairplot()**
2. **Caplot()**
3. **Boxplot()**
4. **Distplot()**
5. **Jointplot()**
6. **Heatmap()**



# SEABORN Vs MATPLOTLIB

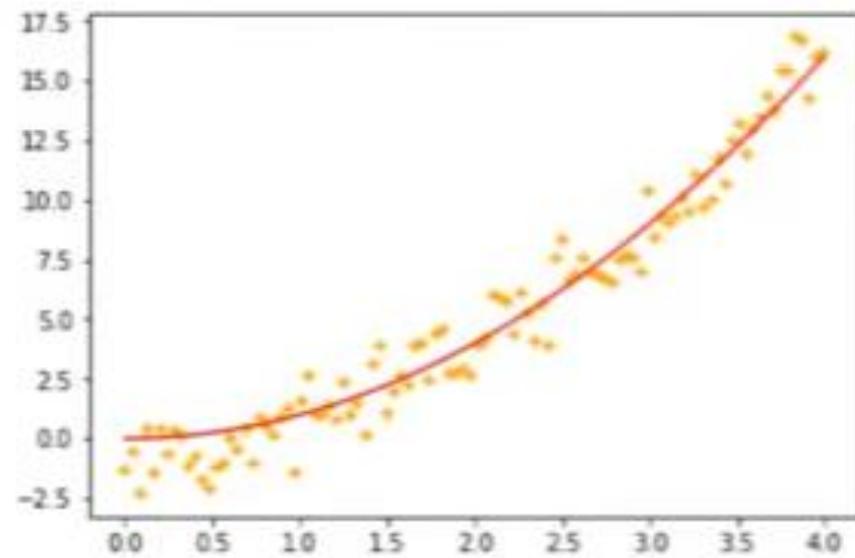
SEABORN	MATPLOTLIB
Data	Fonctions, matrices, etc.
Exploration Statistique	Mathématique, science, ingénierie, etc.
Vision globale	Graphique spécialisé



# MACHINE LEARNING

« Donner à une machine la **capacité d'apprendre** sans la programmer de façon explicite. » Arthur Samuel

→ Développer un **modèle** à partir de **données**



# MACHINE LEARNING

« *Donner à une machine la capacité d'apprendre sans la programmer de façon explicite.* » Arthur Samuel



→ Développer un modèle à partir de données



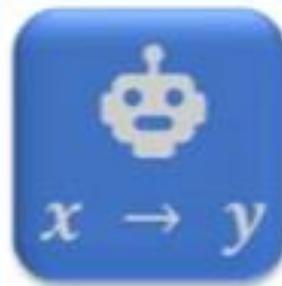
Apprentissage  
Non-supervisé



Apprentissage  
Par renforcement



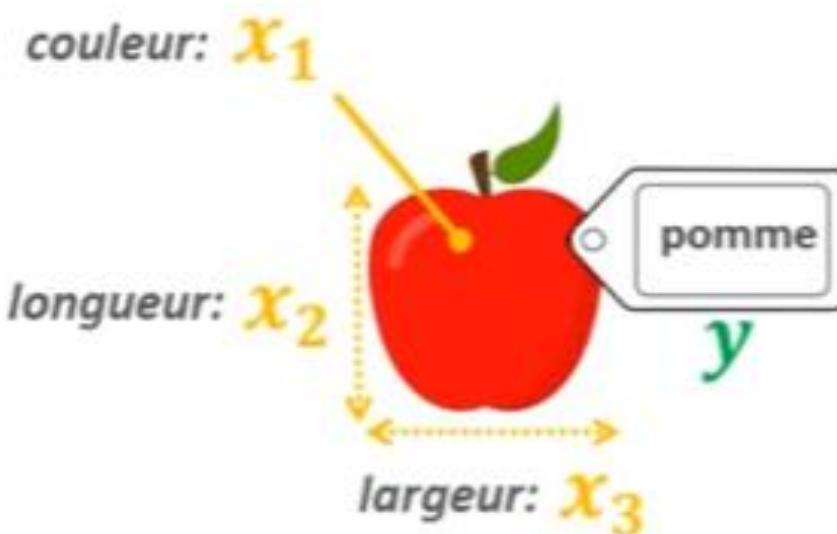
# SUPERVISED LEARNING



La machine reçoit des données  
caractérisées par des variables  $x$   
et annotées d'une variable  $y$

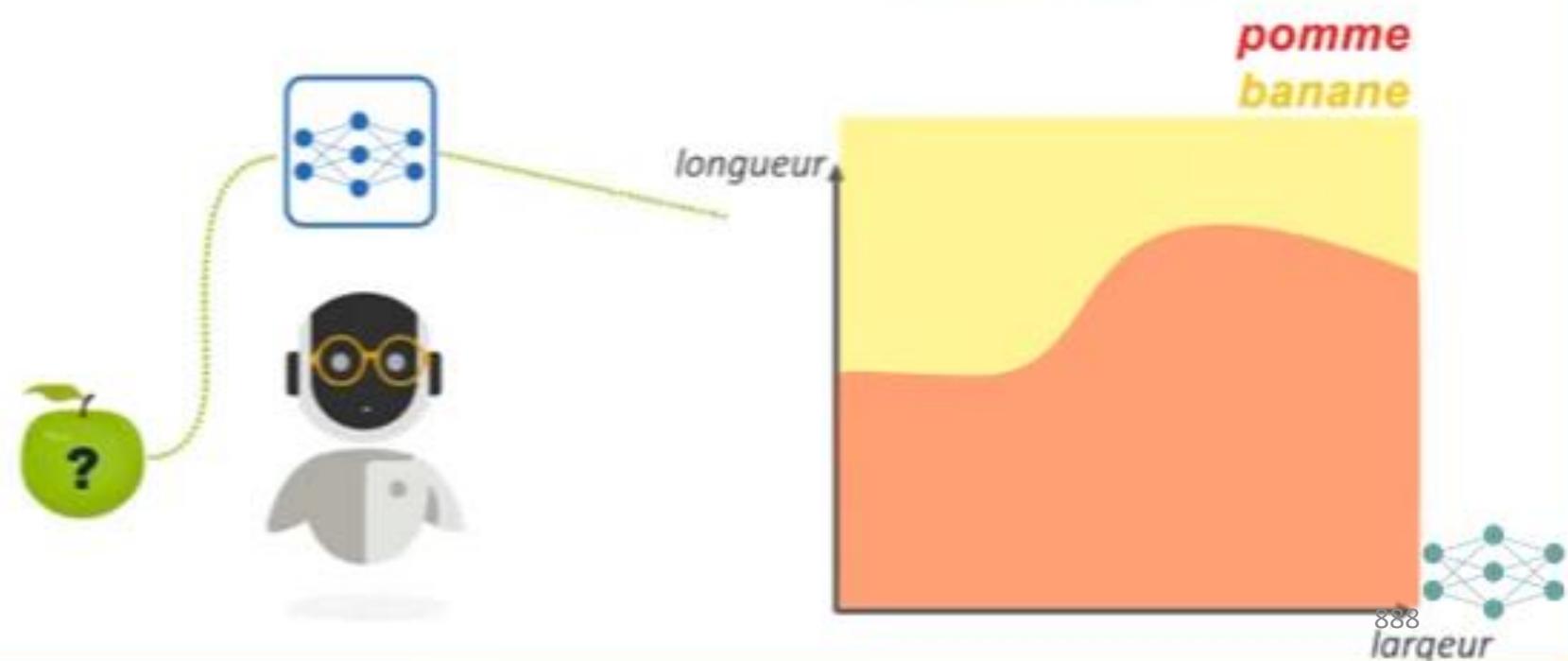
Prédire  $y$   
en fonction de  $x$

$x$  *Features*  
 $y$  *Label / Target*



# SUPERVISED LEARNING

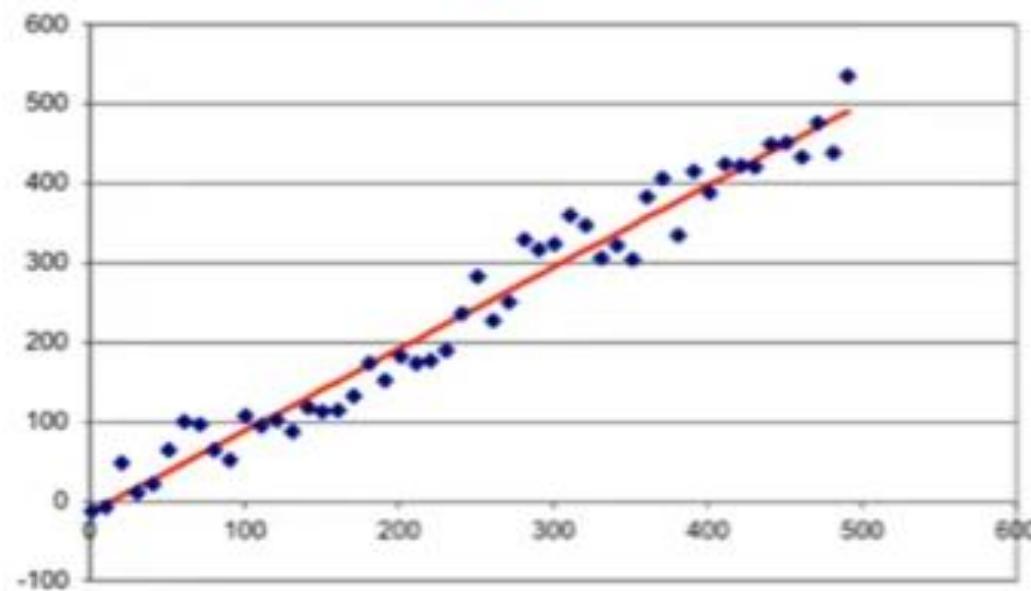
C'est fini ! Le modèle de machine Learning est prêt à être utilisé.



# SUPERVISED LEARNING

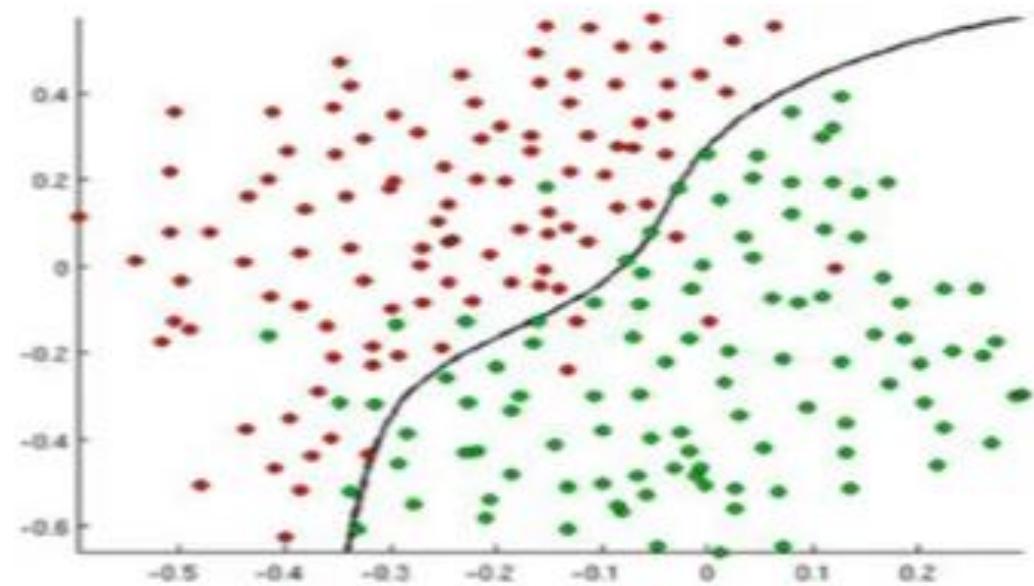
Permet de résoudre 2 types de problèmes:

## Régression



Prix d'un appartement  
(y est **continue**)

## Classification



Email Spam / non Spam  
(y est **discrete**)



# SUPERVISED LEARNING



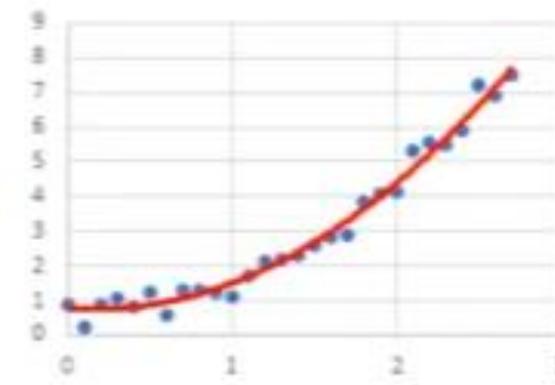
*Dataset*



*Entrainement*

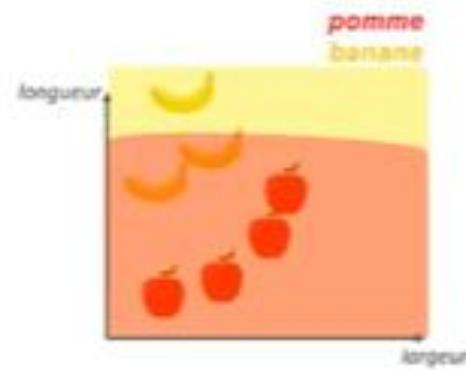


*Modèle*



# SKLEARN

*Avec 3 méthodes présentes dans toutes les classes !*



# INTERFACE DE SKLEARN

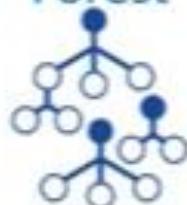
Linear  
Regression



Decision  
Tree



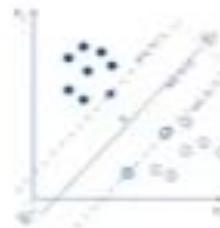
Random  
Forest



K-NN



SVM



Neural  
Network



*Différents mécanismes*

*Mais la même interface*



*fit*



*score*



*predict*



# SKLEARN SUPERVISED LEARNING

1. Sélectionner un **estimateur** et préciser ses **hyperparamètres** :

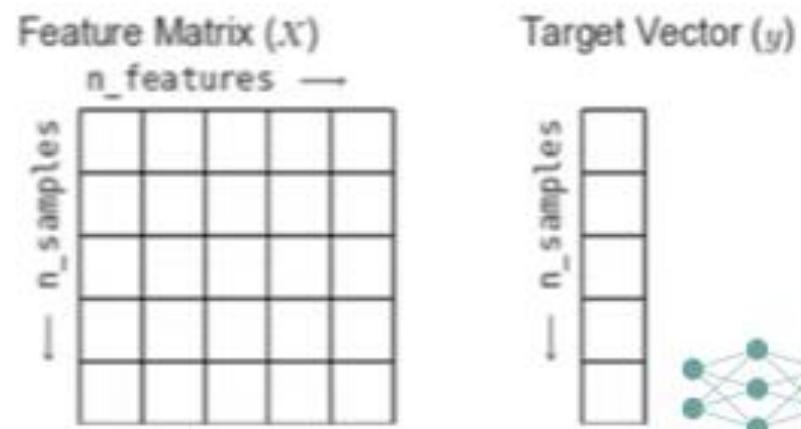
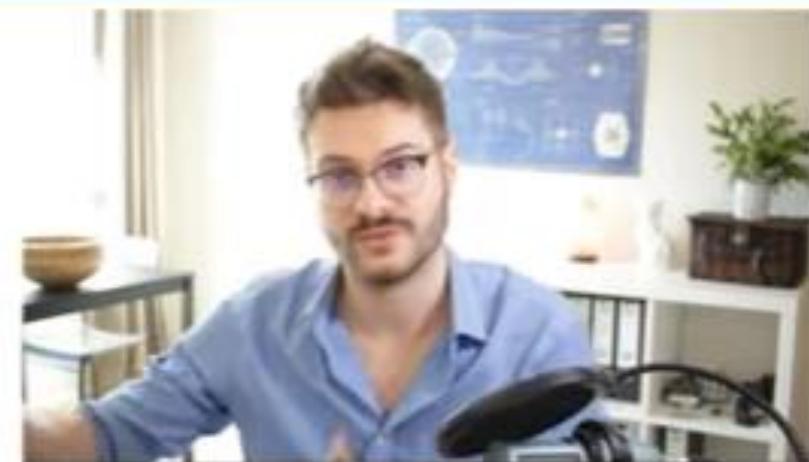
*model = LinearRegression(.....)*

2. Entrainer le modèle sur les données **X, y**  
(divisées en 2 tableaux **Numpy**)

*model.fit(X, y)*

⇒ X et y doivent avoir **2 dimensions** !

*[n\_samples, n\_features]*



# SKLEARN SUPERVISED LEARNING

1. Sélectionner un **estimateur** et préciser ses **hyperparamètres** :

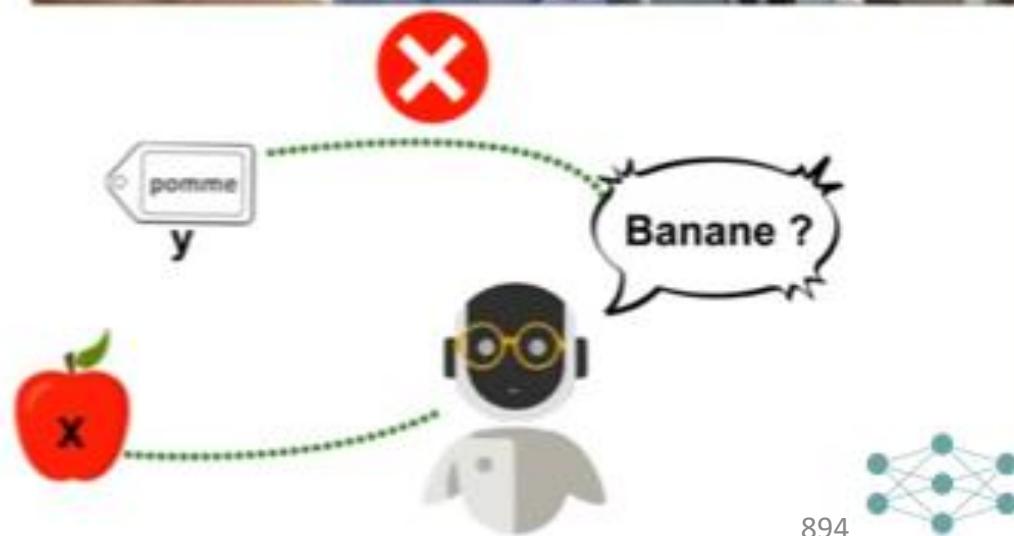
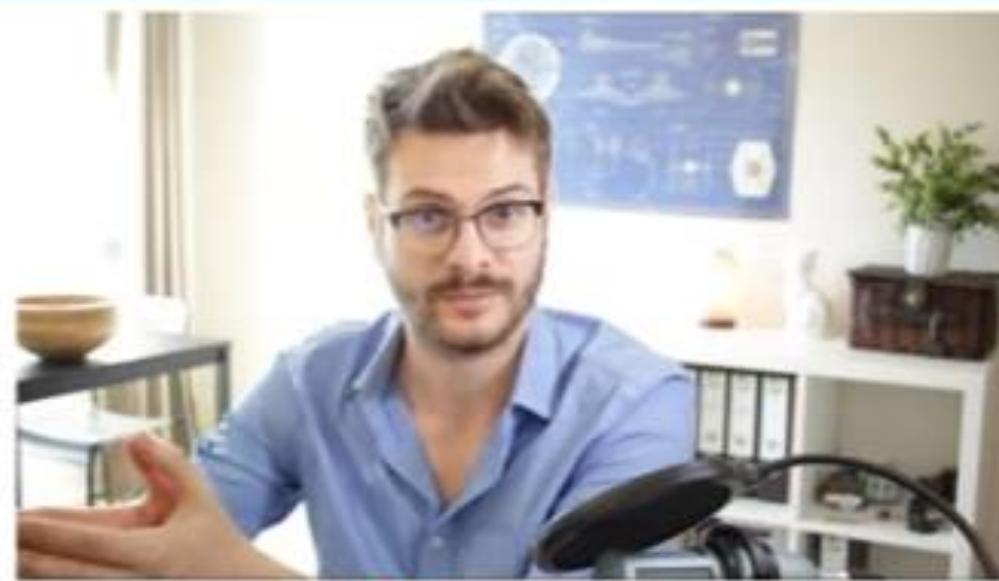
```
model = LinearRegression(.....)
```

2. Entrainer le modèle sur les données **X, y**  
(divisées en 2 tableaux **Numpy**)

```
model.fit(X, y)
```

3. Évaluer le modèle

```
model.score(X, y)
```



# SKLEARN SUPERVISED LEARNING

1. Sélectionner un estimateur et préciser ses hyperparamètres :

*model = LinearRegression(.....)*

2. Entrainer le modèle sur les données  $X$ ,  $y$   
*(divisées en 2 tableaux Numpy)*

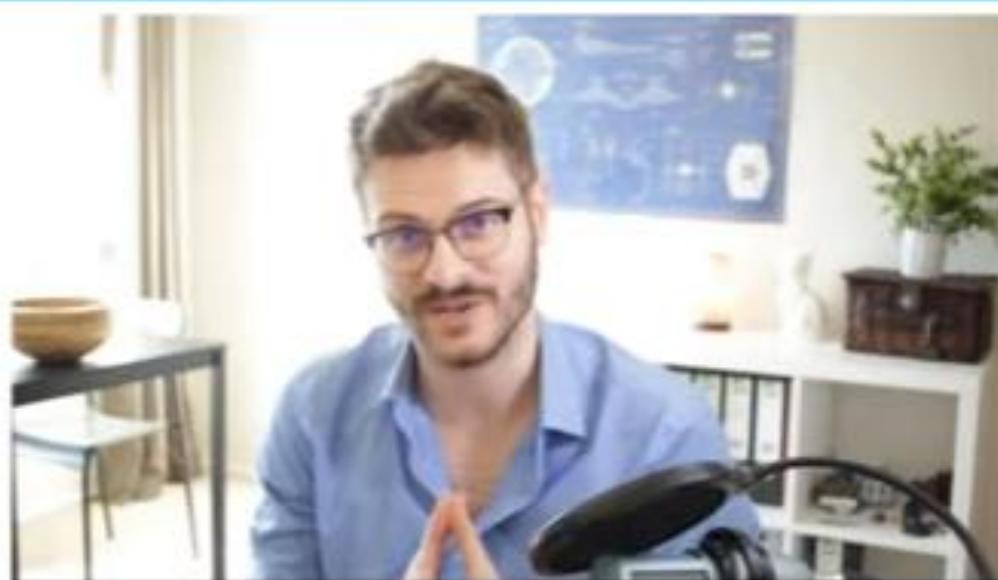
*model.fit(X, y)*

3. Évaluer le modèle

*model.score(X, y)*

4. Utiliser le modèle

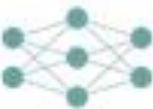
*model.predict(X)*



# SKLEARN EN 4 LIGNES

## Régression Linéaire:

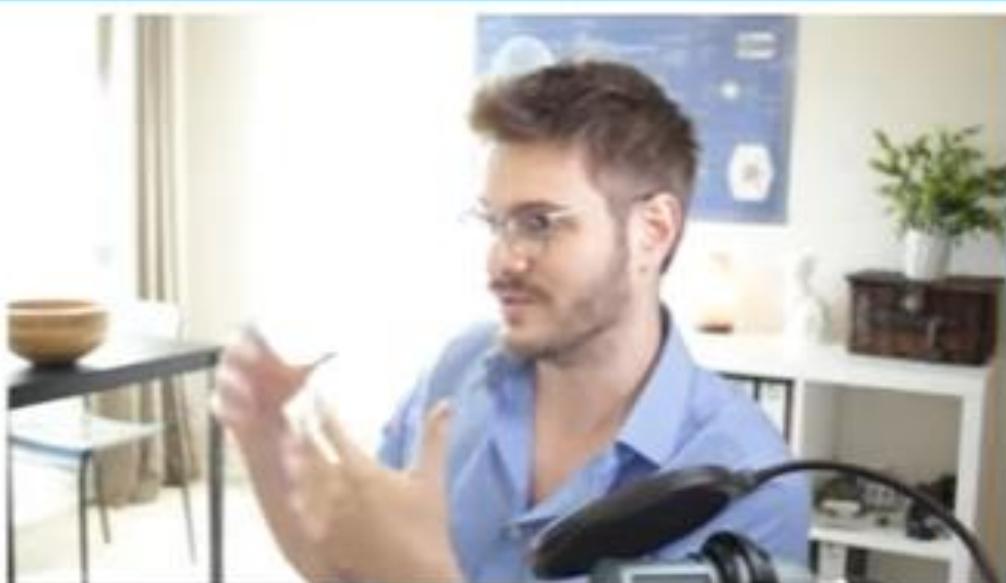
```
model = LinearRegression()  
model.fit(X, y)  
model.score(X, y)  
model.predict(X)
```



# SKLEARN EN 4 LIGNES

## Régression Logistique:

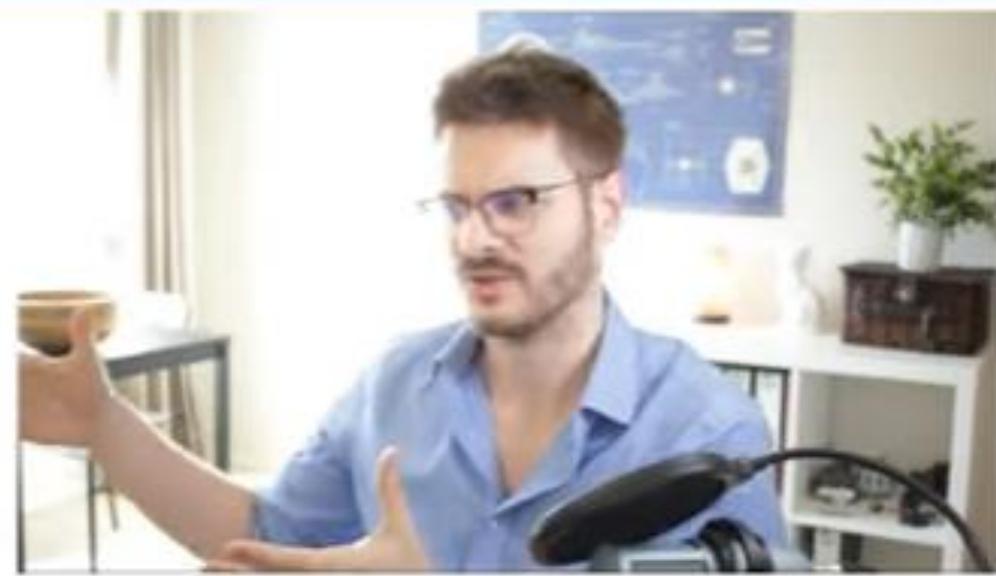
```
model = LogisticRegression()  
model.fit(X, y)  
model.score(X, y)  
model.predict(X)
```



# SKLEARN EN 4 LIGNES

## Support Vector Machine:

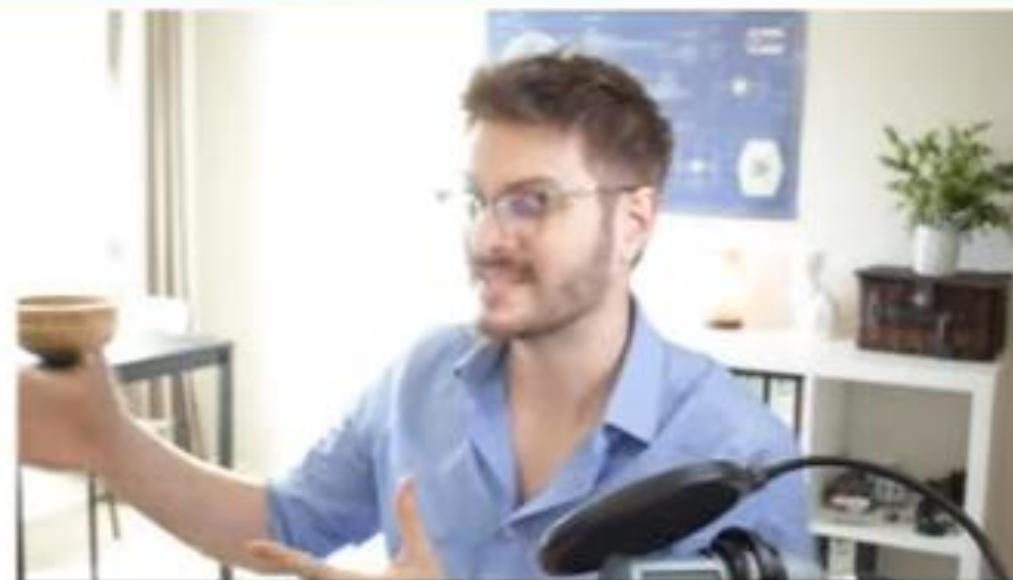
```
model = SVC()  
model.fit(X, y)  
model.score(X, y)  
model.predict(X)
```



# SKLEARN EN 4 LIGNES

## Random Forest:

```
model = RandomForestClassifier()  
model.fit(X, y)  
model.score(X, y)  
model.predict(X)
```



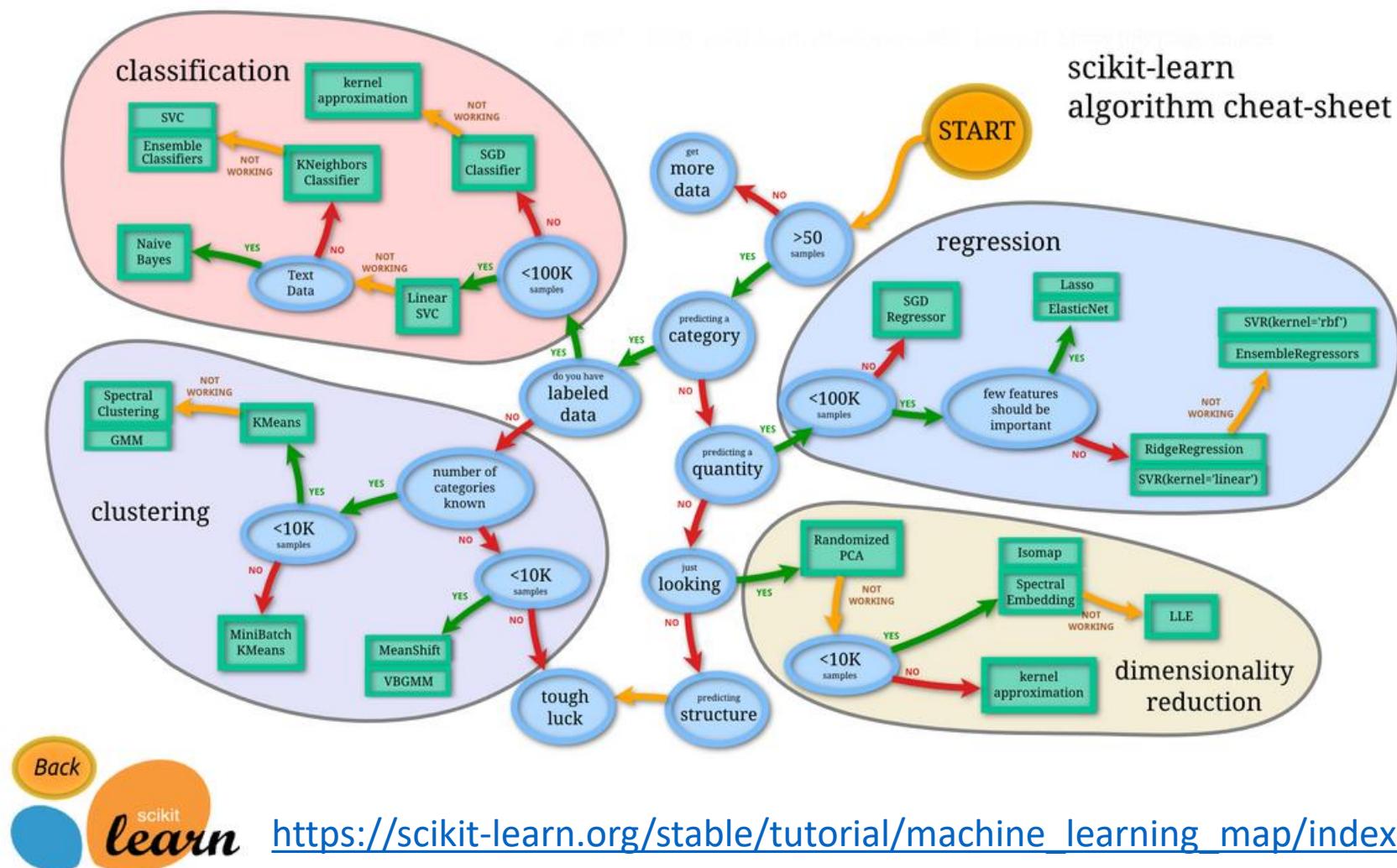
# Choosing the right estimator

Often the hardest part of solving a machine learning problem can be finding the right estimator for the job.

Different estimators are better suited for different types of data and different problems.

The flowchart below is designed to give users a bit of a rough guide on how to approach problems with regard to which estimators to try on your data.

Click on any estimator in the chart below to see its documentation.



# SKLEARN SUPERVISED LEARNING

1. Sélectionner un **estimateur** et préciser ses **hyperparamètres** :

```
model = LinearRegression(.....)
```

objet      Constructeur      Hyperparamètres

2. Entrainer le modèle sur les données **X, y** (divisées en 2 tableaux **Numpy**)

```
model.fit(X, y)
```

3. Évaluer le modèle

```
model.score(X, y)
```

4. Utiliser le modèle

```
model.predict(X)
```

