Apprentissage de la programmation

avec Python 3

Christian VINCENT

LETTRES SORBONNE UNIVERSITÉ

L1 - Semestre 1 - année 2023-2024

Premiers pas, types et variables

- Chapitre I -

Face à un problème posé, il faut savoir :

Face à un problème posé, il faut savoir :

- par où commencer

Face à un problème posé, il faut savoir :

- par où commencer
- de quoi j'ai besoin

Face à un problème posé, il faut savoir :

- par où commencer
- de quoi j'ai besoin
- comment implémenter un algorithme qui réalise la tâche visée

Face à un problème posé, il faut savoir :

- par où commencer
- de quoi j'ai besoin
- comment implémenter un algorithme qui réalise la tâche visée

Définition

Un algorithme peut être vu comme une suite d'étapes à parcourir en vue de réaliser une tâche, ceci en un temps acceptable (donc fini).

Face à un problème posé, il faut savoir :

- par où commencer
- de quoi j'ai besoin
- comment implémenter un algorithme qui réalise la tâche visée

Définition

Un algorithme peut être vu comme une suite d'étapes à parcourir en vue de réaliser une tâche, ceci en un temps acceptable (donc fini).

- Un algorithme étant trouvé, on peut ensuite passer à l'étape de la programmation.

Face à un problème posé, il faut savoir :

- par où commencer
- de quoi j'ai besoin
- comment implémenter un algorithme qui réalise la tâche visée

Définition

Un algorithme peut être vu comme une suite d'étapes à parcourir en vue de réaliser une tâche, ceci en un temps acceptable (donc fini).

- Un algorithme étant trouvé, on peut ensuite passer à l'étape de la programmation.

Concevoir un algorithme et savoir l'implémenter (passer à la programmation) nécessitent une certaine habitude et ne s'apprend qu'essentiellement par la pratique (comme le vélo !)

Exemple :

Exemple :

- Problème : dans la phrase $P =$ **"quand passe-t-on dans l'action ?"**, combien y a-t-il de voyelles ?

Exemple :

- Problème : dans la phrase $P =$ **"quand passe-t-on dans l'action ?"**, combien y a-t-il de voyelles ?
- Posons le problème :
  - on connaît les voyelles $V =$ "a e i o u y"

Exemple :

- Problème : dans la phrase $P =$ **"quand passe-t-on dans l'action ?"**, combien y a-t-il de voyelles ?
- Posons le problème :
  - on connaît les voyelles $V =$ "a e i o u y"
  - on connaît la phrase P

Exemple :

- Problème : dans la phrase $P =$ **"quand passe-t-on dans l'action ?"**, combien y a-t-il de voyelles ?
- Posons le problème :
  - on connaît les voyelles $V =$ "a e i o u y"
  - on connaît la phrase P
  - on peut lire chaque élément (lettres, espaces, ponctuation) de P, de "q" à " ?", et vérifier pour chacun si il est dans V.

Exemple :

- Problème : dans la phrase $P =$ **"quand passe-t-on dans l'action ?"**, combien y a-t-il de voyelles ?
- Posons le problème :
  - on connaît les voyelles $V =$ "a e i o u y"
  - on connaît la phrase P
  - on peut lire chaque élément (lettres, espaces, ponctuation) de P, de "q" à " ?", et vérifier pour chacun si il est dans V.
  - Si c'est le cas on ajoute 1 au nombre de voyelles trouvées, sinon on passe à l'élément suivant de P.

Exemple :

- Problème : dans la phrase $P =$ **"quand passe-t-on dans l'action ?"**, combien y a-t-il de voyelles ?
- Posons le problème :
  - on connaît les voyelles $V =$ "a e i o u y"
  - on connaît la phrase P
  - on peut lire chaque élément (lettres, espaces, ponctuation) de P, de "q" à " ?", et vérifier pour chacun si il est dans V.
  - Si c'est le cas on ajoute 1 au nombre de voyelles trouvées, sinon on passe à l'élément suivant de P.

Maintenant, l'essentiel du travail est fait ! Formalisons-le !

Formalisation : il s'agit

Formalisation : il s'agit

- d'énumérer les **données en entrée** : P, V

Formalisation : il s'agit

- d'énumérer les **données en entrée** : P, V
- de repérer les structures algorithmiques connues : parcourir chaque lettre de P (boucle), tester si la lettre est dans V (test)

Formalisation : il s'agit

- d'énumérer les **données en entrée** : P, V
- de repérer les structures algorithmiques connues : parcourir chaque lettre de P (boucle), tester si la lettre est dans V (test)
- d'assurer le stockage des données intermédiaires : compter le nombre de voyelles

Formalisation : il s'agit

- d'énumérer les **données en entrée** : P, V
- de repérer les structures algorithmiques connues : parcourir chaque lettre de P (boucle), tester si la lettre est dans V (test)
- d'assurer le stockage des données intermédiaires : compter le nombre de voyelles
- de s'assurer qu'une réponse au problème est affichée

Formalisation : il s'agit

- d'énumérer les **données en entrée** : P, V
- de repérer les structures algorithmiques connues : parcourir chaque lettre de P (boucle), tester si la lettre est dans V (test)
- d'assurer le stockage des données intermédiaires : compter le nombre de voyelles
- de s'assurer qu'une réponse au problème est affichée

Entrée [1]:

```
1 P = "quand passe-t-on dans l'action"
2 V = "aeiouy"
3 nb_de_voyelle = 0              # compteur pour le nombre de voyelle
4 for lettre in P:               # pour chaque lettre de P
5     if lettre in V:            # la lettre est-elle dans V ? si oui,
6         nb_de_voyelle = nb_de_voyelle + 1  # on ajoute 1 au compteur
7 print("Dans la phrase, il y a", nb_de_voyelle, "voyelle(s)")
```

```
Dans la phrase, il y a 9 voyelle(s)
```

Ce qu'il faut retenir...

- les noms donnés aux objets utilisés sont des *variables*

Ce qu'il faut retenir...

- les noms donnés aux objets utilisés sont des *variables*
- la répétition effectuée sur chaque lettre est une *boucle* (ici avec *for*)

Ce qu'il faut retenir...

- les noms donnés aux objets utilisés sont des *variables*
- la répétition effectuée sur chaque lettre est une *boucle* (ici avec *for*)
- le vérification d'appartenance à V est un *test* (avec *if*)

Ce qu'il faut retenir...

- les noms donnés aux objets utilisés sont des *variables*
- la répétition effectuée sur chaque lettre est une *boucle* (ici avec *for*)
- le vérification d'appartenance à V est un *test* (avec *if*)
- l'écriture de la réponse se fait avec *print()*.

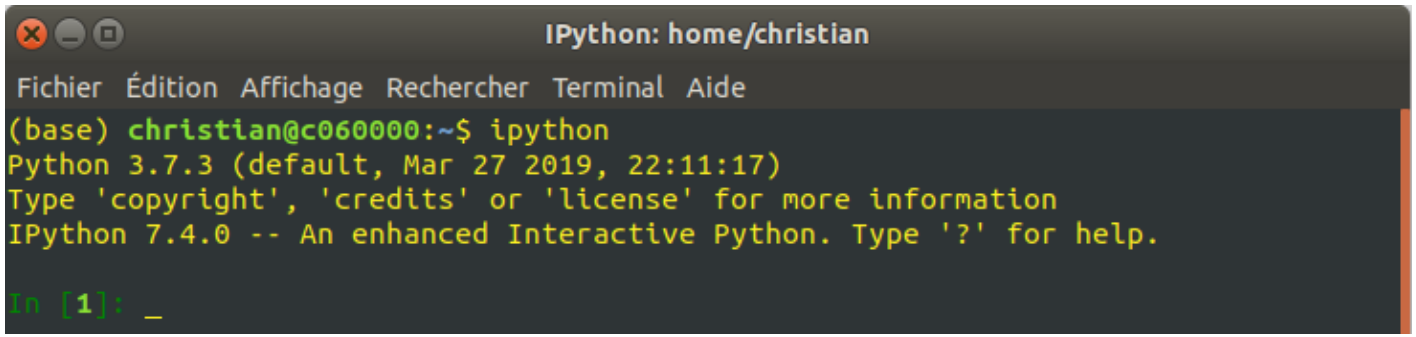
Ce qu'il faut retenir...

- les noms donnés aux objets utilisés sont des *variables*
- la répétition effectuée sur chaque lettre est une *boucle* (ici avec *for*)
- le vérification d'appartenance à V est un *test* (avec *if*)
- l'écriture de la réponse se fait avec *print()*.

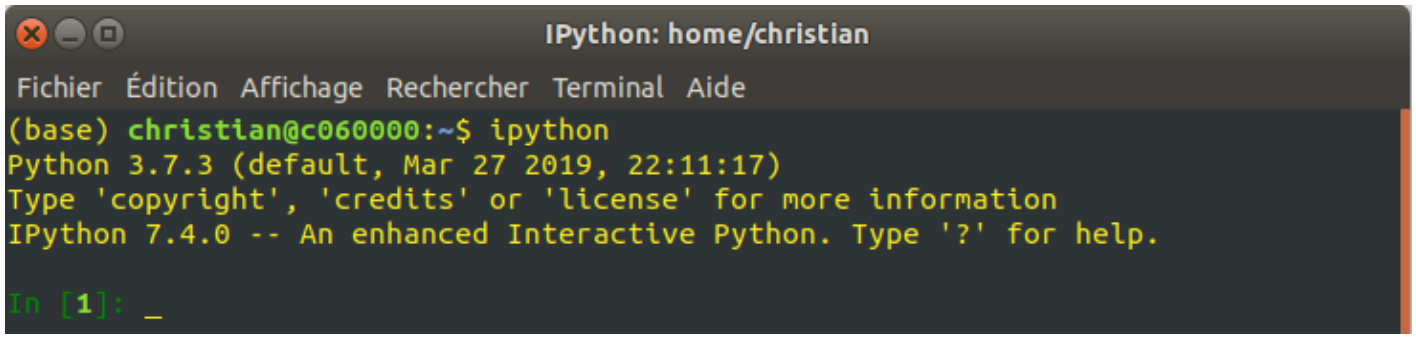
Reste maintenant à utiliser *Python* avec un IDE !

Pour illuster la première utilisation possible, nous allons lancer *ipython* (soit par le menu démarrer de Windows ou en tapant *ipython* dans un terminal pour les autres systèmes).

Pour illuster la première utilisation possible, nous allons lancer *ipython* (soit par le menu démarrer de Windows ou en tapant *ipython* dans un terminal pour les autres systèmes).

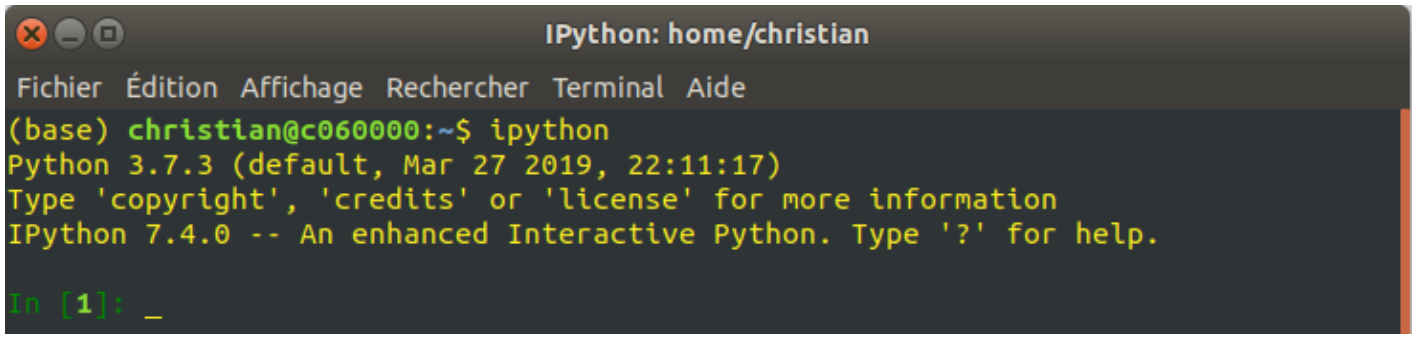


Pour illuster la première utilisation possible, nous allons lancer *ipython* (soit par le menu démarrer de Windows ou en tapant *ipython* dans un terminal pour les autres systèmes).



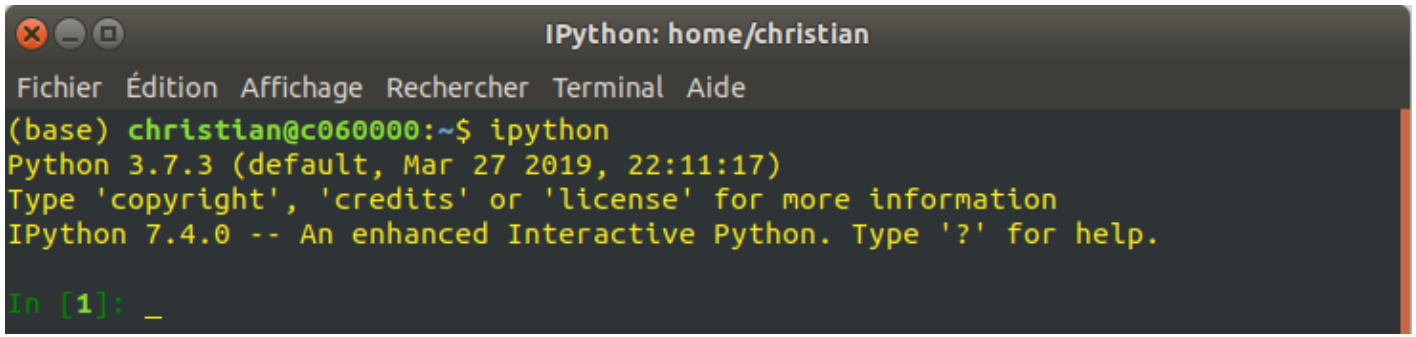
Le **in[1]** qui apparait signifie qu'*ipython* est en situation d'attente d'une entrée...

Pour illuster la première utilisation possible, nous allons lancer *ipython* (soit par le menu démarrer de Windows ou en tapant *ipython* dans un terminal pour les autres systèmes).



Le **in[1]** qui apparait signifie qu'*ipython* est en situation d'attente d'une entrée...
À partir de là, *Python* peut se comporter comme une "calculatrice" de bureau.

Pour illuster la première utilisation possible, nous allons lancer *ipython* (soit par le menu démarrer de Windows ou en tapant *ipython* dans un terminal pour les autres systèmes).



Le **in[1]** qui apparait signifie qu'*ipython* est en situation d'attente d'une entrée...
À partir de là, *Python* peut se comporter comme une "calculatrice" de bureau.
Essayons : 30 + 50 * 2

ipython

```
In [1] : 30 + 50 * 2
Out[1] : 130
```

ipython

```
In [1] : 30 + 50 * 2
Out[1] : 130
```

- les espaces dans les expressions n'ont pas d'importance

ipython

In [1] : 30 + 50 * 2
Out[1] : 130

- les espaces dans les expressions n'ont pas d'importance
- la multiplication est notée *

ipython

In [1] : 30 + 50 * 2
Out[1] : 130

- les espaces dans les expressions n'ont pas d'importance
- la multiplication est notée *
- la division "à virgule" est notée /

ipython

In [1] : 30 + 50 * 2
Out[1] : 130

- les espaces dans les expressions n'ont pas d'importance
- la multiplication est notée *
- la division "à virgule" est notée /
- la division euclidienne (en entier) est notée //

ipython

In [1] : 30 + 50 * 2
Out[1] : 130

- les espaces dans les expressions n'ont pas d'importance
- la multiplication est notée *
- la division "à virgule" est notée /
- la division euclidienne (en entier) est notée //
- l'exponentiation (élévation à la puissance) est notée **

ipython

In [1] : 30 + 50 * 2
Out[1] : 130

- les espaces dans les expressions n'ont pas d'importance
- la multiplication est notée *
- la division "à virgule" est notée /
- la division euclidienne (en entier) est notée //
- l'exponentiation (élévation à la puissance) est notée **
- la virgule est un point .

Ce qui donne par exemple :

Ce qui donne par exemple :

ipython

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

Ce qui donne par exemple :

ipython

```
In [1] : 3 +          4
Out[1] : 7
In [2] : 2 * 3 * 4
Out[2] : 24
```

Ce qui donne par exemple :

ipython

```
In [1] : 3 +          4
Out[1] : 7
In [2] : 2 * 3 * 4
Out[2] : 24
In [3] : 15 / 2
Out[3] : 7.5
In [4] : 14 / 2
Out[4] : 7.0
```

Ce qui donne par exemple :

ipython

```
In [1] : 3 +          4
Out[1] : 7
In [2] : 2 * 3 * 4
Out[2] : 24
In [3] : 15 / 2
Out[3] : 7.5
In [4] : 14 / 2
Out[4] : 7.0
In [5] : 7 // 2
Out[5] : 3
```

Ce qui donne par exemple :

ipython

```
In [1] : 3 +        4
Out[1] : 7
In [2] : 2 * 3 * 4
Out[2] : 24
In [3] : 15 / 2
Out[3] : 7.5
In [4] : 14 / 2
Out[4] : 7.0
In [5] : 7 // 2
Out[5] : 3
In [6] : 3 ** 2
Out[6] : 9
```

Ce qui donne par exemple :

ipython

```
In [1] : 3 +          4
Out[1] : 7
In [2] : 2 * 3 * 4
Out[2] : 24
In [3] : 15 / 2
Out[3] : 7.5
In [4] : 14 / 2
Out[4] : 7.0
In [5] : 7 // 2
Out[5] : 3
In [6] : 3 ** 2
Out[6] : 9
In [7] : 3.7 + 1.3
Out[7] : 5.0
```

Deux types de commentaires sont possibles dans le code :

Deux types de commentaires sont possibles dans le code :

- commentaire précédé d'un dièse #.

Deux types de commentaires sont possibles dans le code :

- commentaire précédé d'un dièse #. Le texte qui suit un dièse n'est pas utilisé par *Python*.

Deux types de commentaires sont possibles dans le code :

- commentaire précédé d'un dièse #. Le texte qui suit un dièse n'est pas utilisé par *Python*.
  Le programmeur commente son code pour s'y retrouver plus facilement.

Deux types de commentaires sont possibles dans le code :

- commentaire précédé d'un dièse #. Le texte qui suit un dièse n'est pas utilisé par *Python*.
  Le programmeur commente son code pour s'y retrouver plus facilement.
- commentaire situé entre deux séries de trois guillemets ''' (simples) ou """ (doubles).

Deux types de commentaires sont possibles dans le code :

- commentaire précédé d'un dièse #. Le texte qui suit un dièse n'est pas utilisé par *Python*.
  Le programmeur commente son code pour s'y retrouver plus facilement.
- commentaire situé entre deux séries de trois guillemets ''' (simples) ou """ (doubles). Ce qui est entre les deux séries peut être utilisé comme documentation par *Python* (avec ?).

Deux types de commentaires sont possibles dans le code :

- commentaire précédé d'un dièse #. Le texte qui suit un dièse n'est pas utilisé par *Python*.
  Le programmeur commente son code pour s'y retrouver plus facilement.
- commentaire situé entre deux séries de trois guillemets ''' (simples) ou """ (doubles). Ce qui est entre les deux séries peut être utilisé comme documentation par *Python* (avec ?).

ipython

```
In [1] : 3 + 4 # c'est une addition
```

Deux types de commentaires sont possibles dans le code :

- commentaire précédé d'un dièse #. Le texte qui suit un dièse n'est pas utilisé par *Python*.
  Le programmeur commente son code pour s'y retrouver plus facilement.
- commentaire situé entre deux séries de trois guillemets ''' (simples) ou """ (doubles). Ce qui est entre les deux séries peut être utilisé comme documentation par *Python* (avec ?).

ipython

```
In [1] : 3 + 4 # c'est une addition
Out[1] : 7
```

Deux types de commentaires sont possibles dans le code :

- commentaire précédé d'un dièse #. Le texte qui suit un dièse n'est pas utilisé par *Python*.
  Le programmeur commente son code pour s'y retrouver plus facilement.
- commentaire situé entre deux séries de trois guillemets ''' (simples) ou """ (doubles). Ce qui est entre les deux séries peut être utilisé comme documentation par *Python* (avec ?).

ipython

```
In [1] : 3 + 4 # c'est une addition
Out[1] : 7
In [1] : def carre(n) : # fonction carrée
             """renvoie le carré de n"""
             return n ** 2
```

- *ipython*, comme *jupyter notebook* utilise la coloration syntaxique.

- *ipython*, comme *jupyter notebook* utilise la coloration syntaxique.
- Cela signifie que les "mots clés" de *Python*, les opérateurs, les variables, les fonctions (nous allons voir ce que tout cela est !), les commentaires, ... ont une couleur attitrée, pour mieux s'y retrouver.

- *ipython*, comme *jupyter notebook* utilise la coloration syntaxique.
- Cela signifie que les "mots clés" de *Python*, les opérateurs, les variables, les fonctions (nous allons voir ce que tout cela est !), les commentaires, ... ont une couleur attitrée, pour mieux s'y retrouver.

- Dans l'image précédente, on voit une "Entrée" qui prend trois lignes.

- Dans l'image précédente, on voit une "Entrée" qui prend trois lignes.
- Il est clair que l'on ne va pas utiliser *Python* que, comme une calculatrice.

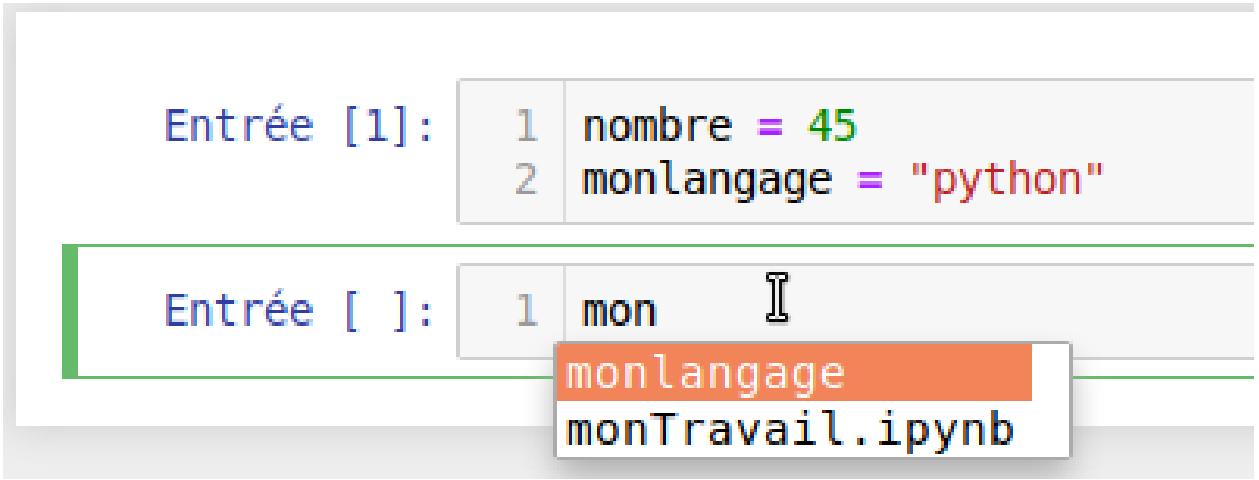
- Dans l'image précédente, on voit une "Entrée" qui prend trois lignes.
- Il est clair que l'on ne va pas utiliser *Python* que, comme une calculatrice. On va écrire des programmes (qui contiennent souvent des algorithmes) de plusieurs lignes. On parle alors de script *Python*

- Dans l'image précédente, on voit une "Entrée" qui prend trois lignes.
- Il est clair que l'on ne va pas utiliser *Python* que, comme une calculatrice. On va écrire des programmes (qui contiennent souvent des algorithmes) de plusieurs lignes. On parle alors de script *Python*
- Pour cela *jupyter notebook* est plus approprié pour une première approche.

- *ipython*, comme *jupyter notebook* utilise la complétion automatique.

- *ipython*, comme *jupyter notebook* utilise la complétion automatique.
- Cela signifie qu'il suffit d'écrire les premières lettres d'un mot que l'on a défini ou d'un mot clé de *Python* (on verra plus tard ces mots clés), d'appuyer ensuite sur la touche de tabulation du clavier (touche avec ce symbole $\leftrightarrows$ pour se voir proposer tous les mots possibles commençant par ces quelques lettres.

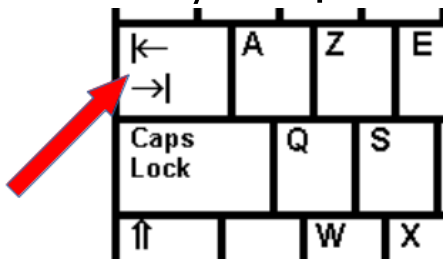
- *ipython*, comme *jupyter notebook* utilise la complétion automatique.
- Cela signifie qu'il suffit d'écrire les premières lettres d'un mot que l'on a défini ou d'un mot clé de *Python* (on verra plus tard ces mots clés), d'appuyer ensuite sur la touche de tabulation du clavier (touche avec ce symbole $\leftrightarrows$ pour se voir proposer tous les mots possibles commençant par ces quelques lettres.
- Attention : les mots doivent avoir été définis auparavant et/ou la cellule qui contient ce mot exécutée au moins une fois.


Entrée [1]:
1 nombre = 45
2 monlangage = "python"
Entrée [ ]:
1 mon
monlangage
monTravail.ipynb

```
Entrée [1]:  1 nombre = 45
             2 monlangage = "python"

Entrée [ ]:  1 mon
               monlangage
               monTravail.ipynb
```

À partir de 3 lettres entrées au clavier, si vous appuyez sur la touche de tabulation, le *notebook* (comme *ipython*) vous propose tous les mots (=mot-clé, mots déjà définis) commençant par ces lettres.



- Dans le mode "calculatrice", à chaque fois qu'une ligne est validée (ou un groupe de lignes que l'on ne peut scinder), la ligne est interprétée et exécutée.

- Dans le mode "calculatrice", à chaque fois qu'une ligne est validée (ou un groupe de lignes que l'on ne peut scinder), la ligne est interprétée et exécutée.
- Dès que l'on veut écrire plusieurs lignes de code, on va le faire dans un éditeur (fichier distinct ou le notebook), et on demande ensuite à *Python* d'exécuter l'ensemble du fichier.

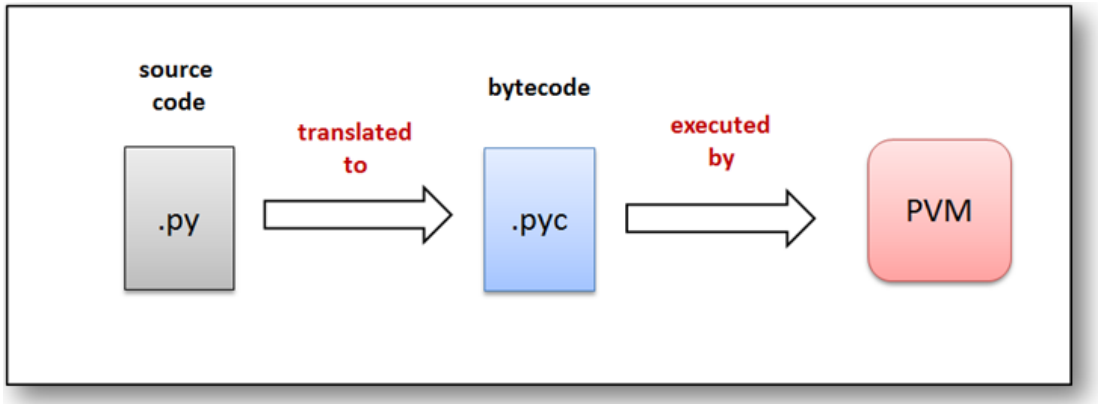
- Dans le mode "calculatrice", à chaque fois qu'une ligne est validée (ou un groupe de lignes que l'on ne peut scinder), la ligne est interprétée et exécutée.
- Dès que l'on veut écrire plusieurs lignes de code, on va le faire dans un éditeur (fichier distinct ou le notebook), et on demande ensuite à *Python* d'exécuter l'ensemble du fichier.
- L'avantage est que le code peut être modifié (et corrigé) à souhait, et l'ensemble réexécuté autant de fois que l'on veut.

- Le programme (les lignes de code) s'appelle le *source*.

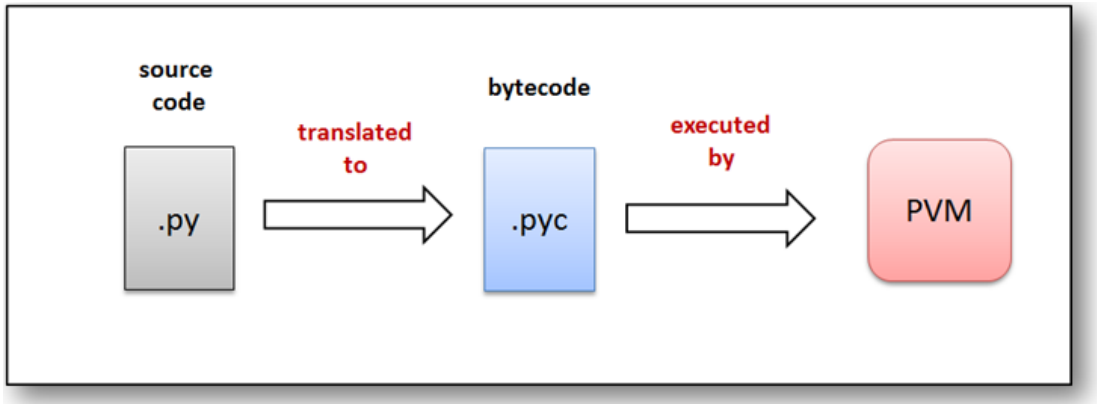
- Le programme (les lignes de code) s'appelle le *source*.
- Le source va être dans un premier temps analysé par *Python*

- Le programme (les lignes de code) s'appelle le *source*.
- Le source va être dans un premier temps analysé par *Python*
- Si il n'y a pas d'erreur, le source est compilé et *Python* renvoie du *bytecode* (plus rapidement exécutable).

- Le programme (les lignes de code) s'appelle le *source*.
- Le source va être dans un premier temps analysé par *Python*
- Si il n'y a pas d'erreur, le source est compilé et *Python* renvoie du *bytecode* (plus rapidement exécutable).



- Le programme (les lignes de code) s'appelle le *source*.
- Le source va être dans un premier temps analysé par *Python*
- Si il n'y a pas d'erreur, le source est compilé et *Python* renvoie du *bytecode* (plus rapidement exécutable).



- PVM est la machine virtuelle *Python*. Le bytecode est interprétable sur toute PVM quelque soit le système.

Tous les objets manipulés par *Python* doivent être nommés.

Tous les objets manipulés par *Python* doivent être nommés.

- Ce nom s'appelle l'*identificateur*.

Tous les objets manipulés par *Python* doivent être nommés.

- Ce nom s'appelle l'*identificateur*.
- Contraintes du nom :

Tous les objets manipulés par *Python* doivent être nommés.

- Ce nom s'appelle l'*identificateur*.
- Contraintes du nom :
  - posséder au moins un caractère

Tous les objets manipulés par *Python* doivent être nommés.

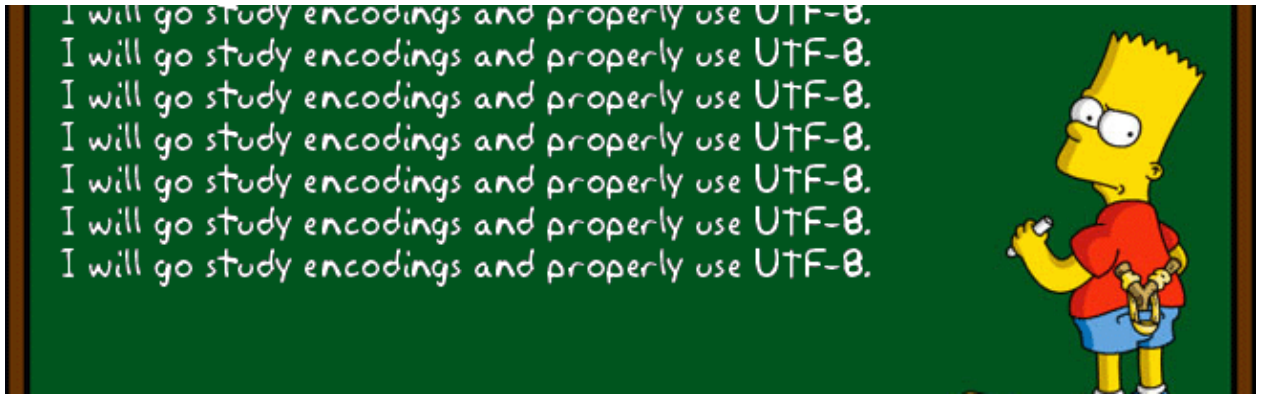
- Ce nom s'appelle l'*identificateur*.
- Contraintes du nom :
  - posséder au moins un caractère
  - la 1^e lettre doit être une lettre ou le souligné _

Tous les objets manipulés par *Python* doivent être nommés.

- Ce nom s'appelle l'*identificateur*.
- Contraintes du nom :
  - posséder au moins un caractère
  - la 1^e lettre doit être une lettre ou le souligné _
  - la suite éventuelle des caractères est quelconque parmi : lettre, chiffres, souligné.

Tous les objets manipulés par *Python* doivent être nommés.

- Ce nom s'appelle l'*identificateur*.
- Contraintes du nom :
  - posséder au moins un caractère
  - la 1^e lettre doit être une lettre ou le souligné _
  - la suite éventuelle des caractères est quelconque parmi : lettre, chiffres, souligné.
- En général, on évite les caractères accentués pour un identificateur (risque d'erreur).



Convention de nommage en *Python*.

Convention de nommage en *Python*.

- les constantes en majuscules, ex : *TVA*

Convention de nommage en *Python*.

- les constantes en majuscules, ex : *TVA*
- les variables en minuscules, ex : *compteur*

Convention de nommage en *Python*.

- les constantes en majuscules, ex : *TVA*
- les variables en minuscules, ex : *compteur*
- notation *mixedCase* pour les fonctions et méthodes, ex : *maFonction*

Convention de nommage en *Python*.

- les constantes en majuscules, ex : *TVA*
- les variables en minuscules, ex : *compteur*
- notation *mixedCase* pour les fonctions et méthodes, ex : *maFonction*
- notation *CamelCase* pour les classes et les exceptions, ex : *MaClasse*

Convention de nommage en *Python*.

- les constantes en majuscules, ex : *TVA*
- les variables en minuscules, ex : *compteur*
- notation *mixedCase* pour les fonctions et méthodes, ex : *maFonction*
- notation *CamelCase* pour les classes et les exceptions, ex : *MaClasse*

Attention

Évitez de donner des noms qui commencent par un ou deux soulignés, et surtout qui commencent et finissent par deux soulignés (usage spécial en *Python*) :

_aaaaa ___aaaaa ___aaaaa___

Certains identifiants sont réservés au langage *Python*.

Certains identifiants sont réservés au langage *Python*.
Il s'agit de :

Certains identifiants sont réservés au langage *Python*.
Il s'agit de :

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

Certains identifiants sont réservés au langage *Python*.
Il s'agit de :

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

Attention

Les identifiants sont sensibles à la casse ! (les votres, comme les réservés, ex : True $\neq$ true, compteur $\neq$ CompTeuR)

Les données manipulées par *Python* ont besoin d'un nom (identificateur).

Les données manipulées par *Python* ont besoin d'un nom (identificateur).
L'affectation est l'opération qui consiste à :

Les données manipulées par *Python* ont besoin d'un nom (identificateur).
L'affectation est l'opération qui consiste à :

- choisir un nom (intelligemment) et stocker ce nom en mémoire

Les données manipulées par *Python* ont besoin d'un nom (identificateur).
L'affectation est l'opération qui consiste à :

- choisir un nom (intelligemment) et stocker ce nom en mémoire
- stocker la donnée en mémoire et la relier au nom

Les données manipulées par *Python* ont besoin d'un nom (identificateur).
L'affectation est l'opération qui consiste à :

- choisir un nom (intelligemment) et stocker ce nom en mémoire
- stocker la donnée en mémoire et la relier au nom
- attribuer un **type** à ce nom

Les données manipulées par *Python* ont besoin d'un nom (identificateur).
L'affectation est l'opération qui consiste à :

- choisir un nom (intelligemment) et stocker ce nom en mémoire
- stocker la donnée en mémoire et la relier au nom
- attribuer un **type** à ce nom

On dit alors que l'on a affecté une *valeur* à une *variable*.

Les données manipulées par *Python* ont besoin d'un nom (identificateur).
L'affectation est l'opération qui consiste à :

- choisir un nom (intelligemment) et stocker ce nom en mémoire
- stocker la donnée en mémoire et la relier au nom
- attribuer un **type** à ce nom

On dit alors que l'on a affecté une *valeur* à une *variable*. La **variable** est le nom et la **valeur** est la donnée.

Les données manipulées par *Python* ont besoin d'un nom (identificateur).
L'affectation est l'opération qui consiste à :

- choisir un nom (intelligemment) et stocker ce nom en mémoire
- stocker la donnée en mémoire et la relier au nom
- attribuer un **type** à ce nom

On dit alors que l'on a affecté une *valeur* à une *variable*. La **variable** est le nom et la **valeur** est la donnée.

En *Python*	nombre = 45
En pseudo langage	*nombre* ← 45
Informatiquement	

Définition

Une variable est un identificateur associé à une valeur. En *Python*, c'est une référence d'objet.

Définition

Une variable est un identificateur associé à une valeur. En *Python*, c'est une référence d'objet.

- *nombre* = 45 : signifie que la valeur affichée à droite est associée au nom situé à gauche.

Définition

Une variable est un identificateur associé à une valeur. En *Python*, c'est une référence d'objet.

- *nombre* = 45 : signifie que la valeur affichée à droite est associée au nom situé à gauche.
- On remarque que l'affectation utilise le signe "=" (rien à voir avec l'égalité mathématique).

Définition

Une variable est un identificateur associé à une valeur. En *Python*, c'est une référence d'objet.

- *nombre* = 45 : signifie que la valeur affichée à droite est associée au nom situé à gauche.
- On remarque que l'affectation utilise le signe "=" (rien à voir avec l'égalité mathématique).
- Les noms donnés doivent être en rapport avec la donnée contenue (c'est mieux...).

Définition

Une variable est un identificateur associé à une valeur. En *Python*, c'est une référence d'objet.

- *nombre* = 45 : signifie que la valeur affichée à droite est associée au nom situé à gauche.
- On remarque que l'affectation utilise le signe "=" (rien à voir avec l'égalité mathématique).
- Les noms donnés doivent être en rapport avec la donnée contenue (c'est mieux...).
- Il est possible de réaffecter une autre valeur à une variable existante.

Exemple :

Exemple :

Entrée [1]:

```
1  a = 3      # que représente "a" ?
2  b = 7      # que représente "b" ?
3  print(a * b)   # que représente ce produit ?
4  largeur = 3
5  longueur = 7
6  print(longueur * largeur)   # qui ne comprend pas ?
7  largeur = 4
8  print(longueur * largeur)
```

```
21
21
28
```

Exemple :

Entrée [1]:

```
1  a = 3      # que représente "a" ?
2  b = 7      # que représente "b" ?
3  print(a * b)   # que représente ce produit ?
4  largeur = 3
5  longueur = 7
6  print(longueur * largeur)   # qui ne comprend pas ?
7  largeur = 4
8  print(longueur * largeur)
```

```
21
21
28
```

Python donne le **type** "entier" (*int* en *Python*) aux variables *a*, *b*, *largeur* et *longueur*.

Si maintenant, on reprend la cellule précédente et affecte à "a" la valeur **textuelle** "lapin", voici ce que l'on obtient.

Si maintenant, on reprend la cellule précédente et affecte à "a" la valeur **textuelle** "lapin", voici ce que l'on obtient.

Entrée [2]:

```
1  a = "lapin"
2  b = 7
3  print(a * b)
4  largeur = 3
5  longueur = 7
6  print(longueur * largeur)   # qui ne comprend pas ?
7  largeur = 4
8  print(longueur * largeur)
```

```
lapinlapinlapinlapinlapinlapinlapin
21
28
```

Si maintenant, on reprend la cellule précédente et affecte à "a" la valeur **textuelle** "lapin", voici ce que l'on obtient.

Entrée [2]:

```
1  a = "lapin"
2  b = 7
3  print(a * b)
4  largeur = 3
5  longueur = 7
6  print(longueur * largeur)   # qui ne comprend pas ?
7  largeur = 4
8  print(longueur * largeur)
```

```
lapinlapinlapinlapinlapinlapinlapin
21
28
```

Python réclame que les valeurs textuelles (string ou str en *Python*) soient entre guillemets simples, doubles, ou triple simple ou doubles.

Par exemple :

Par exemple :

Entrée [3]:

```
1 animal1 = 'lapin'
2 animal2 = "cerf"
3 animal3 = """cochon"""
4 print(animal1, animal2, animal3)
```

```
lapin cerf cochon
```

Voici quelques variantes :

Par exemple :

Entrée [3]:

```
1 animal1 = 'lapin'
2 animal2 = "cerf"
3 animal3 = """cochon"""
4 print(animal1, animal2, animal3)
```

```
lapin cerf cochon
```

Voici quelques variantes :

Entrée [6]:

```
1 texte1 = "j'ai faim"
2 texte2 = """Il dit : "j'ai faim"."""
3 texte3 = 'j\'ai faim'
4 print(texte1, texte2, texte3)
```

```
j'ai faim Il dit : "j'ai faim". j'ai faim
```

▸ Exercice 1

Le fait que la variable "a" contenait la valeur entière 3 puis que "a" contenait ensuite la valeur textuelle "lapin" montre que *Python* affecte à la volée le type requis.

Le fait que la variable "a" contenait la valeur entière 3 puis que "a" contenait ensuite la valeur textuelle "lapin" montre que *Python* affecte à la volée le type requis.

Définition

Le typage des variables sous *Python* est un typage dynamique (par opposition à d'autres langages (comme le C) qui possède un typage statique).

Définition

La fonction *Python* qui donne le type d'une variable (d'un objet plus largement) est **type()**.

Regardez bien la structure du premier **print()**, les 2 suivants étant là pour afficher le type des 2 variables.

Regardez bien la structure du premier **print()**, les 2 suivants étant là pour afficher le type des 2 variables.

Entrée [8]:

```
1 animal = 'lynx'
2 nombre = 4
3 print("Il y a", nombre, animal, "sur la colline")
4 print(type(animal))
5 print(type(nombre))
```

```
Il y a 4 lynx sur la colline
<class 'str'>
<class 'int'>
```

Regardez bien la structure du premier **print()**, les 2 suivants étant là pour afficher le type des 2 variables.

Entrée [8]:

```
1 animal = 'lynx'
2 nombre = 4
3 print("Il y a", nombre, animal, "sur la colline")
4 print(type(animal))
5 print(type(nombre))
```

```
Il y a 4 lynx sur la colline
<class 'str'>
<class 'int'>
```

Pour un nombre "à virgule" ou écrit en notation scientifique, le type est *float*.

Regardez bien la structure du premier **print()**, les 2 suivants étant là pour afficher le type des 2 variables.

Entrée [8]:

```
1 animal = 'lynx'
2 nombre = 4
3 print("Il y a", nombre, animal, "sur la colline")
4 print(type(animal))
5 print(type(nombre))
```

```
Il y a 4 lynx sur la colline
<class 'str'>
<class 'int'>
```

Pour un nombre "à virgule" ou écrit en notation scientifique, le type est *float*.

Entrée [10]:

```
1 resultat = 15 / 2
2 print(resultat, type(resultat))
```

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

- Écrire *animal* = "*cerf*" est une *affectation simple*.

- Écrire *animal* = "*cerf*" est une *affectation simple*.
- On peut aussi utiliser l'*affectation parallèle* :

- Écrire *animal* = "*cerf*" est une *affectation simple*.
- On peut aussi utiliser l'*affectation parallèle* :

Entrée [11]:
```
1 boy, girl = "Lucien", "Mia"
2 print(boy, 'et', girl)
```
```
Lucien et Mia
```

- Écrire *animal* = "*cerf*" est une *affectation simple*.
- On peut aussi utiliser l'*affectation parallèle* :

```
Entrée [11]:
1 boy, girl = "Lucien", "Mia"
2 print(boy, 'et', girl)

Lucien et Mia
```

"Lucien" est affecté à la variable *boy* et "Mia" est affectée à la variable *girl*.
Les 2 affectations ont lieu "simultanément".

- Écrire *animal* = "*cerf*" est une *affectation simple*.
- On peut aussi utiliser l'*affectation parallèle* :

```
Entrée [11]:
1 boy, girl = "Lucien", "Mia"
2 print(boy, 'et', girl)

Lucien et Mia
```

"Lucien" est affecté à la variable *boy* et "Mia" est affectée à la variable *girl*.
Les 2 affectations ont lieu "simultanément".
Pour bien comprendre l'affectation parallèle, voyons un autre exemple.

Entrée [4]:

```
1 n1 = 3
2 n2 = 5
3 n1, n2 = n1 * 2, n2 + n1
4 print("n1 =", n1, "et n2 =", n2)
```

```
n1 = 6 et n2 = 8
```

Entrée [4]:

```
1 n1 = 3
2 n2 = 5
3 n1, n2 = n1 * 2, n2 + n1
4 print("n1 =", n1, "et n2 =", n2)
```

```
n1 = 6 et n2 = 8
```

$n1$ va prendre l'ancienne valeur de $n1 \times 2$ et en même temps, $n2$ va se voir affecter l'ancienne valeur de $n1$ plus l'ancienne valeur de $n2$, d'où le résultat.

Entrée [4]:

```
1 n1 = 3
2 n2 = 5
3 n1, n2 = n1 * 2, n2 + n1
4 print("n1 =", n1, "et n2 =", n2)
```

```
n1 = 6 et n2 = 8
```

$n1$ va prendre l'ancienne valeur de $n1 \times 2$ et en même temps, $n2$ va se voir affecter l'ancienne valeur de $n1$ plus l'ancienne valeur de $n2$, d'où le résultat. Ce qui est différent de :

Entrée [5]:

```
1 n1 = 3
2 n2 = 5
3 n1 = n1 * 2
4 n2 = n2 + n1
5 print("n1 =", n1, "et n2 =", n2)
```

```
n1 = 6 et n2 = 11
```

- On peut utiliser l'*affectation augmentée ou diminuée*.

- On peut utiliser l'*affectation augmentée ou diminuée*.
  C'est un raccourci d'écriture : au lieu d'écrire $a = a + 1$, on peut écrire plus simplement $a += 1$. Ceci est utilisé dans plusieurs langages.

- On peut utiliser l'*affectation augmentée ou diminuée*.
  C'est un raccourci d'écriture : au lieu d'écrire $a = a + 1$, on peut écrire plus simplement $a + = 1$. Ceci est utilisé dans plusieurs langages.

Entrée [16]:

```
1  a, b = 5, 7
2  a += 1   # remplace a = a + 1
3  b -= 1   # remplace b = b - 1
4  a *= 3   # remplace a = a * 3
5  b /= 2   # remplace b = b / 2
6  print("a =", a, "et b =", b)
```

```
a = 18 et b = 3.0
```

- On peut utiliser l'*affectation augmentée ou diminuée*.
  C'est un raccourci d'écriture : au lieu d'écrire $a = a + 1$, on peut écrire plus simplement $a + = 1$. Ceci est utilisé dans plusieurs langages.

```
Entrée [16]:
1 a, b = 5, 7
2 a += 1   # remplace a = a + 1
3 b -= 1   # remplace b = b - 1
4 a *= 3   # remplace a = a * 3
5 b /= 2   # remplace b = b / 2
6 print("a =", a, "et b =", b)

a = 18 et b = 3.0
```

Vous utiliserez ce genre de notation surtout dans les *boucles* (plus loin dans le cours)...

- Enfin on peut utiliser l'*affectation multiple*.

- Enfin on peut utiliser l'*affectation multiple*.

Entrée [6]:

```
1 a = b = c = 1
2 print("a =", a)
3 print("b =", b)
4 print("c =", c)
```

```
a = 1
b = 1
c = 1
```

- Enfin on peut utiliser l'*affectation multiple*.

Entrée [6]:

```
1 a = b = c = 1
2 print("a =", a)
3 print("b =", b)
4 print("c =", c)
```

```
a = 1
b = 1
c = 1
```

Cela est parfois utile quand on veut initialiser plusieurs variables avec une même valeur.

Il y a un dernier type de base à connaître, c'est le *booléen* (en référence au mathématicien *George Boole*).

Il y a un dernier type de base à connaître, c'est le *booléen* (en référence au mathématicien *George Boole*).

Il y a un dernier type de base à connaître, c'est le *booléen* (en référence au mathématicien *George Boole*).

- Un booléen ne prend que 2 valeurs : *True* ou *False*.

Il y a un dernier type de base à connaître, c'est le *booléen* (en référence au mathématicien *George Boole*).

- Un booléen ne prend que 2 valeurs : *True* ou *False*.
- Une entité vraie (ex : $2 + 1 < 10$), un nombre non nul, une chaîne caractères non vide, une liste non vide, ... sont considérés comme *True*.

Il y a un dernier type de base à connaître, c'est le *booléen* (en référence au mathématicien *George Boole*).

- Un booléen ne prend que 2 valeurs : *True* ou *False*.
- Une entité vraie (ex : $2 + 1 < 10$), un nombre non nul, une chaîne caractères non vide, une liste non vide, ... sont considérés comme *True*.
- Au contraire, $2 + 1 > 10$, 0, "", ... sont considérés comme *False*

Comment obtient-on un booléen ?

Comment obtient-on un booléen ?

- par affectation directe :

Entrée [1]:

```
1 a = True
2 b = False
3 print("a =", a, "b =", b)
```

```
a = True b = False
```

Comment obtient-on un booléen ?

- par affectation directe :

Entrée [1]:
```
1 a = True
2 b = False
3 print("a =", a, "b =", b)
```
```
a = True b = False
```

- par évaluation d'une expression :

Entrée [8]:
```
1 print(1 == 2)
2 print(2 == 1 + 1)
```
```
False
True
```

Comment obtient-on un booléen ?

- par affectation directe :

```
Entrée [1]:
1 a = True
2 b = False
3 print("a =", a, "b =", b)

a = True b = False
```

- par évaluation d'une expression :

```
Entrée [8]:
1 print(1 == 2)
2 print(2 == 1 + 1)

False
True
```

Remarque : 1 == 2 signifie 1 est-il égal à 2 ? Rien à voir avec le = de l'affectation !

- par évaluation d'une variable :

Entrée [12]:

```
1 inconnu = 3 < 7
2 print(inconnu)
```

```
True
```

- par évaluation d'une variable :

```
Entrée [12]:
1 inconnu = 3 < 7
2 print(inconnu)

True
```

Un calcul peut s'évaluer par rapport à un booléen, avec *True* = 1 et *False* = 0.

```
Entrée [14]:
1 nombre = 1
2 print(5 * (nombre < 2) + 7 * (nombre > 2))
3 nombre = 4
4 print(5 * (nombre < 2) + 7 * (nombre > 2))

5
7
```

Quels sont les opérateurs de comparaison ? (qui renvoie un booléen)

Quels sont les opérateurs de comparaison ? (qui renvoie un booléen)

- l'égalité avec ==

Quels sont les opérateurs de comparaison ? (qui renvoie un booléen)

- l'égalité avec ==
- la différence avec !=

Quels sont les opérateurs de comparaison ? (qui renvoie un booléen)

- l'égalité avec ==
- la différence avec !=
- la comparaison avec :

Quels sont les opérateurs de comparaison ? (qui renvoie un booléen)

- l'égalité avec ==
- la différence avec !=
- la comparaison avec :
  - inférieur strictement <

Quels sont les opérateurs de comparaison ? (qui renvoie un booléen)

- l'égalité avec ==
- la différence avec !=
- la comparaison avec :
  - inférieur strictement <
  - supérieur strictement >

Quels sont les opérateurs de comparaison ? (qui renvoie un booléen)

- l'égalité avec ==
- la différence avec !=
- la comparaison avec :
  - inférieur strictement <
  - supérieur strictement >
  - inférieur ou égal <=

Quels sont les opérateurs de comparaison ? (qui renvoie un booléen)

- l'égalité avec ==
- la différence avec !=
- la comparaison avec :
  - inférieur strictement <
  - supérieur strictement >
  - inférieur ou égal <=
  - supérieur ou égal >=

Quels sont les opérateurs de comparaison ? (qui renvoie un booléen)

- l'égalité avec ==
- la différence avec !=
- la comparaison avec :
  - inférieur strictement <
  - supérieur strictement >
  - inférieur ou égal <=
  - supérieur ou égal >=

Exemple :

```
Entrée [15]: 1 print(2 == 3, 2 < 3, 2 > 3, 2 != 3, 2 <= 3, 2 >= 3)
             False True False True True False
```

Quelles opérations avec des booléens ? (qui renvoie un booléen)

Quelles opérations avec des booléens ? (qui renvoie un booléen)

- le *not*

Quelles opérations avec des booléens ? (qui renvoie un booléen)

- le *not*
  si $a = True$ alors *not* $a = False$.

Quelles opérations avec des booléens ? (qui renvoie un booléen)

- le *not*
  si $a = True$ alors *not* $a = False$.
  si $a = False$ alors *not* $a = True$

Quelles opérations avec des booléens ? (qui renvoie un booléen)

- le *not*
  si $a = True$ alors *not* $a = False$.
  si $a = False$ alors *not* $a = True$
- le *and* et le *or*

Quelles opérations avec des booléens ? (qui renvoie un booléen)

- le *not*
  si $a = True$ alors *not* $a = False$.
  si $a = False$ alors *not* $a = True$
- le *and* et le *or*
  si $a = True$ et $b = True$ alors a and $b = True$, a or $b = True$

Quelles opérations avec des booléens ? (qui renvoie un booléen)

- le *not*
  si $a = True$ alors *not* $a = False$.
  si $a = False$ alors *not* $a = True$
- le *and* et le *or*
  si $a = True$ et $b = True$ alors a and $b = True$, a or $b = True$
  si $a = True$ et $b = False$ alors a and $b = False$, a or $b = True$

Quelles opérations avec des booléens ? (qui renvoie un booléen)

- le *not*
  si $a = True$ alors *not* $a = False$.
  si $a = False$ alors *not* $a = True$
- le *and* et le *or*
  si $a = True$ et $b = True$ alors a and $b = True$, a or $b = True$
  si $a = True$ et $b = False$ alors a and $b = False$, a or $b = True$
  si $a = False$ et $b = True$ alors a and $b = False$, a or $b = True$

Quelles opérations avec des booléens ? (qui renvoie un booléen)

- le *not*
  si $a = True$ alors *not* $a = False$.
  si $a = False$ alors *not* $a = True$
- le *and* et le *or*
  si $a = True$ et $b = True$ alors a and $b = True$, a or $b = True$
  si $a = True$ et $b = False$ alors a and $b = False$, a or $b = True$
  si $a = False$ et $b = True$ alors a and $b = False$, a or $b = True$
  si $a = False$ et $b = False$ alors a and $b = False$, a or $b = False$

Quelles opérations avec des booléens ? (qui renvoie un booléen)

- le *not*
  si $a = True$ alors *not* $a = False$.
  si $a = False$ alors *not* $a = True$
- le *and* et le *or*
  si $a = True$ et $b = True$ alors a and $b = True$, a or $b = True$
  si $a = True$ et $b = False$ alors a and $b = False$, a or $b = True$
  si $a = False$ et $b = True$ alors a and $b = False$, a or $b = True$
  si $a = False$ et $b = False$ alors a and $b = False$, a or $b = False$

Ce qui précéde s'appelle une *table de vérité*.

Exemple :

Exemple :

Entrée [4]:

```
1 a = 4
2 b = 5
3 print("1 :\t", a > 0 and b > 0, "\t", a > 0 or b > 0)
4 print("2 :\t", a < 0 and b < 0, "\t", a < 0 or b < 0)
5 print("3 :\t", a > 0 and b < 0, "\t", a > 0 or b < 0)
6 print("4 :\t", a < 0 and b > 0, "\t", a < 0 or b > 0)
```

```
1 :      True    True
2 :      False   False
3 :      False   True
4 :      False   True
```

Exemple :

Entrée [4]:

```
1 a = 4
2 b = 5
3 print("1 :\t", a > 0 and b > 0, "\t", a > 0 or b > 0)
4 print("2 :\t", a < 0 and b < 0, "\t", a < 0 or b < 0)
5 print("3 :\t", a > 0 and b < 0, "\t", a > 0 or b < 0)
6 print("4 :\t", a < 0 and b > 0, "\t", a < 0 or b > 0)
```

```
1 :      True    True
2 :      False   False
3 :      False   True
4 :      False   True
```

On remarque que, pour une raison esthétique de présentation, il y a un "**t**" dans le **print()**. Cela induit une *tabulation horizontale*.

▸ Exercice 2

- En licence "Sciences du Langage", la place du texte a une position prépondérante sur celui des calculs. C'est pour cela que vous allez aborder en profondeur une majorité de fonctionnalités de *Python* liées aux chaînes de caractères.

- En licence "Sciences du Langage", la place du texte a une position prépondérante sur celui des calculs. C'est pour cela que vous allez aborder en profondeur une majorité de fonctionnalités de *Python* liées aux chaînes de caractères.
- Les chaînes de caractères (lettre(s), mots, phrases, textes, et plus) sont représentées par la classe *string* (str en *Python*).

- En licence "Sciences du Langage", la place du texte a une position prépondérante sur celui des calculs. C'est pour cela que vous allez aborder en profondeur une majorité de fonctionnalités de *Python* liées aux chaînes de caractères.
- Les chaînes de caractères (lettre(s), mots, phrases, textes, et plus) sont représentées par la classe *string* (str en *Python*).
- Une chaîne de caractères peut être composée de caractères alphanumériques, symboles divers (dont la ponctuation), et d'espaces.

Entrée [1]:
```
1 chaine1 = "3 lapins !!! et ils crient %#*&%$£ !!!"
2 print(chaine1)
```
```
3 lapins !!! et ils crient %#*&%$£ !!!
```

- Les chaînes de caractères (ou *string*) sont entourées de guillemets qui autorisent l'inclusion d'un type dans un autre (simple dans double ou le contraire).

- Les chaînes de caractères (ou *string*) sont entourées de guillemets qui autorisent l'inclusion d'un type dans un autre (simple dans double ou le contraire).
  Exemple :

- Les chaînes de caractères (ou *string*) sont entourées de guillemets qui autorisent l'inclusion d'un type dans un autre (simple dans double ou le contraire).
  Exemple :

Entrée [2]:

```
1 chaine1 = "j'ai dit : stop !"
2 chaine2 = 'il dit :"ça suffit !"'
3 print(chaine1)
4 print(chaine2)
```

```
j'ai dit : stop !
il dit :"ça suffit !"
```

- Pour gérer toutes les langues du monde (ou presque), les strings doivent être encodés en *Unicode* (façon dont les caractères sont implémentés en 0 et 1 sur une mémoire).

- Les chaînes de caractères (ou *string*) sont entourées de guillemets qui autorisent l'inclusion d'un type dans un autre (simple dans double ou le contraire).
  Exemple :

Entrée [2]:

```
1 chaine1 = "j'ai dit : stop !"
2 chaine2 = 'il dit :"ça suffit !"'
3 print(chaine1)
4 print(chaine2)
```

```
j'ai dit : stop !
il dit :"ça suffit !"
```

- Pour gérer toutes les langues du monde (ou presque), les strings doivent être encodés en *Unicode* (façon dont les caractères sont implémentés en 0 et 1 sur une mémoire).
  L'**utf-8** est une implementations de l'Unicode gèrer par défaut par *Python*.

Une chaîne de caractères est d'un type *immutable*, c'est-à-dire qu'une fois affectée à une variable, on ne peut plus la changer.

Une chaîne de caractères est d'un type *immutable*, c'est-à-dire qu'une fois affectée à une variable, on ne peut plus la changer. Sauf en créant une nouvelle chaîne et en la réaffectant à la même variable.

Une chaîne de caractères est d'un type *immutable*, c'est-à-dire qu'une fois affectée à une variable, on ne peut plus la changer. Sauf en créant une nouvelle chaîne et en la réaffectant à la même variable.

Explication : une chaîne de caractères peut être vue comme un tableau de caractères.

Une chaîne de caractères est d'un type *immutable*, c'est-à-dire qu'une fois affectée à une variable, on ne peut plus la changer. Sauf en créant une nouvelle chaîne et en la réaffectant à la même variable.

Explication : une chaîne de caractères peut être vue comme un tableau de caractères.

str	L	A	P	I	N
rang positif	0	1	2	3	4
rang négatif	−5	−4	−3	−2	−1

Chaque caractère a un rang positif de 0 à la longueur de la chaîne moins un !.

Une chaîne de caractères est d'un type *immutable*, c'est-à-dire qu'une fois affectée à une variable, on ne peut plus la changer.
Sauf en créant une nouvelle chaîne et en la réaffectant à la même variable.
Explication : une chaîne de caractères peut être vue comme un tableau de caractères.

str	L	A	P	I	N
rang positif	0	1	2	3	4
rang négatif	−5	−4	−3	−2	−1

Chaque caractère a un rang positif de 0 à la longueur de la chaîne moins un !.
ou un rang négatif, de −1 à − longueur de la chaîne.

Observez le code !

Observez le code !

Entrée [1]:

```
1 chaine = 'LAPIN'
2 print(chaine[0])    # 1e lettre
3 print(chaine[-1])   # dernière lettre
4 # on veut maintenant changer une lettre
5 chaine[2] = "H"     # j'affecte le lettre H en remplacement de la lettre P
6 print("Résultat :", chaine)
```

```
L
N

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-d84b4dc98d30> in <module>
      3 print(chaine[-1])   # dernière lettre
      4 # on veut maintenant changer une lettre
----> 5 chaine[2] = "H"     # j'affecte le lettre H en remplacement de la lettre P
      6 print("Résultat :", chaine)

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

Observez le code !

Entrée [1]:

```
1 chaine = 'LAPIN'
2 print(chaine[0])    # 1e lettre
3 print(chaine[-1])   # dernière lettre
4 # on veut maintenant changer une lettre
5 chaine[2] = "H"     # j'affecte le lettre H en remplacement de la lettre P
6 print("Résultat :", chaine)
```

```
L
N

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-d84b4dc98d30> in <module>
      3 print(chaine[-1])   # dernière lettre
      4 # on veut maintenant changer une lettre
----> 5 chaine[2] = "H"     # j'affecte le lettre H en remplacement de la lettre P
      6 print("Résultat :", chaine)

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

Python a relevé une erreur. Le type *str* ne supporte pas la modification d'un élément (3^e lettre ici) qui compose ce str...

Vous avez vu précédemment une séquence particulière "\t". C'est une *séquence d'échappement*.

Vous avez vu précédemment une séquence particulière "\t". C'est une *séquence d'échappement*. Elles sont préfixées par un anti-slash \.

Vous avez vu précédemment une séquence particulière "\t". C'est une *séquence d'échappement*. Elles sont préfixées par un anti-slash \.
Voici les principales à connaître :

Vous avez vu précédemment une séquence particulière "\t". C'est une *séquence d'échappement*. Elles sont préfixées par un anti-slash \.
Voici les principales à connaître :

\n	Saut de ligne

Vous avez vu précédemment une séquence particulière "\t". C'est une *séquence d'échappement*. Elles sont préfixées par un anti-slash \.
Voici les principales à connaître :

\n	Saut de ligne
\t	Tabulation horizontale

Vous avez vu précédemment une séquence particulière "\t". C'est une *séquence d'échappement*. Elles sont préfixées par un anti-slash \.
Voici les principales à connaître :

\n	Saut de ligne
\t	Tabulation horizontale
\v	Tabulation verticale

Vous avez vu précédemment une séquence particulière "\t". C'est une *séquence d'échappement*. Elles sont préfixées par un anti-slash \.
Voici les principales à connaître :

\n	Saut de ligne
\t	Tabulation horizontale
\v	Tabulation verticale
\'	Apostrophe

Vous avez vu précédemment une séquence particulière "\t". C'est une *séquence d'échappement*. Elles sont préfixées par un anti-slash \.
Voici les principales à connaître :

\n	Saut de ligne
\t	Tabulation horizontale
\v	Tabulation verticale
\'	Apostrophe
\"	Guillemet

Vous avez vu précédemment une séquence particulière "\t". C'est une *séquence d'échappement*. Elles sont préfixées par un anti-slash \.
Voici les principales à connaître :

\n	Saut de ligne
\t	Tabulation horizontale
\v	Tabulation verticale
\'	Apostrophe
\"	Guillemet

L'\ peut aussi servir à indiquer à *Python* qu'une ligne est trop longue et se poursuit sur la ligne suivante. Voir les exemples qui suivent.

Entrée [6]:

```
1 chaine1 = 'J\'ai très faim\net soif\t\t et finalement les 2'
2 chaine2 = "Il dit : \"dehors !\", il\ncourt vers\tlui"
3 print("Voici la première phrase :", chaine1, "\nsuivie de la \
4 deuxième phrase :", chaine2)
```

```
Voici la première phrase : J'ai très faim
et soif          et finalement les 2
suivie de la deuxième phrase : Il dit : "dehors !", il
court vers      lui
```

Entrée [6]:

```
1 chaine1 = 'J\'ai très faim\net soif\t\t et finalement les 2'
2 chaine2 = "Il dit : \"dehors !\", il\ncourt vers\tlui"
3 print("Voici la première phrase :", chaine1, "\nsuivie de la \
4 deuxième phrase :", chaine2)
```

```
Voici la première phrase : J'ai très faim
et soif          et finalement les 2
suivie de la deuxième phrase : Il dit : "dehors !", il
court vers      lui
```

Notez l'effet du \n qui provoque un saut de ligne à l'affichage, du \t qui provoque une(ou des) tabulation(s).

Entrée [6]:

```
1 chaine1 = 'J\'ai très faim\net soif\t\t et finalement les 2'
2 chaine2 = "Il dit : \"dehors !\", il\ncourt vers\tlui"
3 print("Voici la première phrase :", chaine1, "\nsuivie de la \
4 deuxième phrase :", chaine2)
```

```
Voici la première phrase : J'ai très faim
et soif          et finalement les 2
suivie de la deuxième phrase : Il dit : "dehors !", il
court vers      lui
```

Notez l'effet du \n qui provoque un saut de ligne à l'affichage, du \t qui provoque une(ou des) tabulation(s).
Observez bien la(il n'y en a qu'une) ligne du **print()**.

Entrée [6]:

```
1 chaine1 = 'J\'ai très faim\net soif\t\t et finalement les 2'
2 chaine2 = "Il dit : \"dehors !\", il\ncourt vers\tlui"
3 print("Voici la première phrase :", chaine1, "\nsuivie de la \
4 deuxième phrase :", chaine2)
```

```
Voici la première phrase : J'ai très faim
et soif          et finalement les 2
suivie de la deuxième phrase : Il dit : "dehors !", il
court vers      lui
```

Notez l'effet du \n qui provoque un saut de ligne à l'affichage, du \t qui provoque une(ou des) tabulation(s).
Observez bien la(il n'y en a qu'une) ligne du **print()**. Comme la ligne était trop longue, on a mis un \ pour indiquer à *Python* que la ligne du print() se poursuit sur la ligne suivante. Ceci n'est pas obligatoire.

Voici ce que l'on aurait obtenu sinon :

Voici ce que l'on aurait obtenu sinon :

Entrée [6]:

```
1 chaine1 = 'J\'ai très faim\net soif\t\t et finalement les 2'
2 chaine2 = "Il dit : \"dehors !\", il\ncourt vers\tlui"
3 print("Voici la première phrase :", chaine1, "\nsuivie de la de
```

```
Voici la première phrase : J'ai très faim
et soif          et finalement les 2
suivie de la deuxième phrase : Il dit : "dehors !", il
court vers      lui
```

un ascenceur horizontal, mais pas très lisible...

Voici ce que l'on aurait obtenu sinon :

Entrée [6]:
```
1 chaine1 = 'J\'ai très faim\net soif\t\t et finalement les 2'
2 chaine2 = "Il dit : \"dehors !\", il\ncourt vers\tlui"
3 print("Voici la première phrase :", chaine1, "\nsuivie de la de
```

```
Voici la première phrase : J'ai très faim
et soif          et finalement les 2
suivie de la deuxième phrase : Il dit : "dehors !", il
court vers      lui
```

un ascenceur horizontal, mais pas très lisible...

Entrée [6]:
```
1 \net soif\t\t et finalement les 2'
2 rs !\", il\ncourt vers\tlui"
3 hrase :", chaine1, "\nsuivie de la deuxième phrase :", chaine2)
```

```
Voici la première phrase : J'ai très faim
et soif          et finalement les 2
suivie de la deuxième phrase : Il dit : "dehors !", il
court vers      lui
```

Quelles opérations sur les strings ?

Quelles opérations sur les strings ?

- La concaténation, le $+$:
  $p1 =$ "*Alain* ", $p2 =$ "*Terrieur*", $p1 + p2$ donne "*Alain Terrieur*".

Quelles opérations sur les strings ?

- La concaténation, le $+$:
  $p1 =$ "*Alain* ", $p2 =$ "*Terrieur*", $p1 + p2$ donne "*Alain Terrieur*".
- La répétition avec * :
  $p1 =$ "*Alain* ", "*Alain* " $* 3$, donne "*Alain Alain Alain* ".

Quelles opérations sur les strings ?

- La concaténation, le $+$:
  $p1 =$ "*Alain* ", $p2 =$ "*Terrieur*", $p1 + p2$ donne "*Alain Terrieur*".
- La répétition avec * :
  $p1 =$ "*Alain* ", "*Alain* " $* 3$, donne "*Alain Alain Alain* ".
- L'appartenance avec *in* :
  $p1 =$ "*Alain*", et *lettre* = "*a*", *lettre in* $p1$ renvoie le booléen *True*.

Quelles opérations sur les strings ?

- La concaténation, le $+$:
  $p1 =$ "*Alain* ", $p2 =$ "*Terrieur*", $p1 + p2$ donne "*Alain Terrieur*".
- La répétition avec * :
  $p1 =$ "*Alain* ", "*Alain* " $* 3$, donne "*Alain Alain Alain* ".
- L'appartenance avec *in* :
  $p1 =$ "*Alain*", et *lettre* = "*a*", *lettre in* $p1$ renvoie le booléen *True*.
- La longueur d'une chaîne avec *len*() (fonction)

Exemples :

Exemples :

Entrée [9]:

```
1  p1 = "je cours "      # notez l'espace après le s
2  p2 = "dans le champ"
3  print(p1 + p2)        # la concaténation colle les chaînes
4  print(p1 * 4)
5  print("test1", "a" in p1)
6  print("text2", "J" in p1)
7  motif = "cha"
8  print("test3", motif in p1 + p2)
9  print("longueur de p2 :", len(p2))
10 print("A-t-on len(p1)+len(p2)==len(p1+p2) ?", \
11       len(p1)+len(p2)==len(p1+p2))
```

```
je cours dans le champ
je cours je cours je cours je cours
test1 False
text2 False
test3 True
longueur de p2 : 13
A-t-on len(p1)+len(p2)==len(p1+p2) ? True
```

Le slicing : le slicing repose sur l'indexation de chacun des caractères qui composent une chaîne.

Le slicing : le slicing repose sur l'indexation de chacun des caractères qui composent une chaîne.

Slicing

Le slicing est l'opérateur qui permet l'extraction de tranche de caractères (contigus ou pas). La syntaxe utilisée est une notation avec crochet qui suit la chaîne de caractères.

Exemple :

Entrée [1]:

```
1 chaine = 'SORBONNE-UNIVERSITE'
2 print(chaine[3:11])
3 print(chaine[5::2])
4 print(chaine[::-1])
```

```
BONNE-UN
NEUIEST
ETISREVINU-ENNOBROS
```

La syntaxe exacte est la suivante :

La syntaxe exacte est la suivante :
chaîne[début : fin : pas] où

La syntaxe exacte est la suivante :
chaîne[début : fin : pas] où

- début est l'indice de départ de l'extraction

La syntaxe exacte est la suivante :
chaîne[début : fin : pas] où

- début est l'indice de départ de l'extraction
- fin est l'indice de fin de l'extraction

La syntaxe exacte est la suivante :
chaîne[début : fin : pas] où

- début est l'indice de départ de l'extraction
- fin est l'indice de fin de l'extraction
- pas spécifie de combien en combien de caractères on extrait..

La syntaxe exacte est la suivante :
chaîne[début : fin : pas] où

- début est l'indice de départ de l'extraction
- fin est l'indice de fin de l'extraction
- pas spécifie de combien en combien de caractères on extrait..
- notez que le pas peut être négatif (on extrait alors de la droite vers la gauche)

La syntaxe exacte est la suivante :
chaîne[début : fin : pas] où

- début est l'indice de départ de l'extraction
- fin est l'indice de fin de l'extraction
- pas spécifie de combien en combien de caractères on extrait..
- notez que le pas peut être négatif (on extrait alors de la droite vers la gauche)
- si début n'est pas spécifié, on commence de l'indice 0 (pas positif) ou du dernier indice (pas négatif).

La syntaxe exacte est la suivante :
chaîne[début : fin : pas] où

- début est l'indice de départ de l'extraction
- fin est l'indice de fin de l'extraction
- pas spécifie de combien en combien de caractères on extrait..
- notez que le pas peut être négatif (on extrait alors de la droite vers la gauche)
- si début n'est pas spécifié, on commence de l'indice 0 (pas positif) ou du dernier indice (pas négatif).
- si fin n'est pas spécifié, on termine au dernier indice (pas positif) ou à l'indice 0 (pas négatif)

La syntaxe exacte est la suivante :
chaîne[début : fin : pas] où

- début est l'indice de départ de l'extraction
- fin est l'indice de fin de l'extraction
- pas spécifie de combien en combien de caractères on extrait..
- notez que le pas peut être négatif (on extrait alors de la droite vers la gauche)
- si début n'est pas spécifié, on commence de l'indice 0 (pas positif) ou du dernier indice (pas négatif).
- si fin n'est pas spécifié, on termine au dernier indice (pas positif) ou à l'indice 0 (pas négatif)
- si pas n'est pas spécifié, il est de 1 par défaut.

Application :

Application :

Entrée [6]:

```
1 chaine = 'ABCDEFGHIJKLMNOP'
2 print(chaine[3:])
3 print(chaine[:3])
4 print(chaine[::-2])
5 print(chaine[3:14:3])
6 newchaine = chaine[:]
7 print(newchaine)
8 chaine == newchaine
```

```
DEFGHIJKLMNOP
ABC
PNLJHFDB
DGJM
ABCDEFGHIJKLMNOP
```

Out[6]: True

Application :

Entrée [6]:

```
1 chaine = 'ABCDEFGHIJKLMNOP'
2 print(chaine[3:])
3 print(chaine[:3])
4 print(chaine[::-2])
5 print(chaine[3:14:3])
6 newchaine = chaine[:]
7 print(newchaine)
8 chaine == newchaine
```

```
DEFGHIJKLMNOP
ABC
PNLJHFDB
DGJM
ABCDEFGHIJKLMNOP
```

Out[6]: True

Le slicing est en fait du découpage de chaîne de caractères. Très utile pour travailler efficacement sur les strings.

Une méthode sur un string est une fonction **attachée** à ce string.

Une méthode sur un string est une fonction **attachée** à ce string. C'est pour cela que l'application d'une méthode se fait avec la **notation pointée**.

Une méthode sur un string est une fonction **attachée** à ce string.
C'est pour cela que l'application d'une méthode se fait avec la **notation pointée**.
Exemple : on applique une mise en majuscule à la chaîne "lapin"

Une méthode sur un string est une fonction **attachée** à ce string.
C'est pour cela que l'application d'une méthode se fait avec la **notation pointée**.
Exemple : on applique une mise en majuscule à la chaîne "lapin"

Mise en majuscule

```
"lapin".upper()
```

Une méthode sur un string est une fonction **attachée** à ce string.
C'est pour cela que l'application d'une méthode se fait avec la **notation pointée**.
Exemple : on applique une mise en majuscule à la chaîne "lapin"

Mise en majuscule

```
"lapin".upper()
```

La méthode upper() attachée (par le point) à la chaîne "lapin" la met en majuscule. Elle renvoie une nouvelle chaîne !

Une méthode sur un string est une fonction **attachée** à ce string. C'est pour cela que l'application d'une méthode se fait avec la **notation pointée**.
Exemple : on applique une mise en majuscule à la chaîne "lapin"

Mise en majuscule

```
"lapin".upper()
```

La méthode upper() attachée (par le point) à la chaîne "lapin" la met en majuscule. Elle renvoie une nouvelle chaîne !

```
1 print('lapin'.upper())
2 chaine = 'coq'
3 print(chaine.upper())
```

```
LAPIN
COQ
```

Méthodes qui renvoient une nouvelle chaîne.

Méthodes qui renvoient une nouvelle chaîne.

codes	renvoie...	Explications
"Lapin".lower()	lapin	passe en minuscule

Méthodes qui renvoient une nouvelle chaîne.

codes	renvoie...	Explications
"Lapin".lower()	lapin	passe en minuscule
"Lapin".upper()	LAPIN	passe en majuscule

Méthodes qui renvoient une nouvelle chaîne.

codes	renvoie...	Explications
"Lapin".lower()	lapin	passe en minuscule
"Lapin".upper()	LAPIN	passe en majuscule
"laPiN".swapcase()	LApIn	inverse majuscule et minuscule

Méthodes qui renvoient une nouvelle chaîne.

codes	renvoie...	Explications
"Lapin".lower()	lapin	passe en minuscule
"Lapin".upper()	LAPIN	passe en majuscule
"laPiN".swapcase()	LApIn	inverse majuscule et minuscule
"Lapin".rjust(10,"-")	----Lapin	justifie à droite avec - en 10 caractères

Méthodes qui renvoient une nouvelle chaîne.

codes	renvoie...	Explications
"Lapin".lower()	lapin	passe en minuscule
"Lapin".upper()	LAPIN	passe en majuscule
"laPiN".swapcase()	LApIn	inverse majuscule et minuscule
"Lapin".rjust(10,"-")	----Lapin	justifie à droite avec - en 10 caractères
"Lapin".ljust(10,"*")	lapin*****	justifie à gauche avec* en 10 caractères

Suite des méthodes ...

Suite des méthodes ...

"Lapin".center(15,"+")	+++++Lapin+++++	centre en 15 caractères avec +

Suite des méthodes ...

"Lapin".center(15,"+")	+++++Lapin+++++	centre en 15 caractères avec +
"Lapin.lstrip("Lap")	in	suppression des caractères Lap à gauche

Suite des méthodes ...

"Lapin".center(15,"+")	+++++Lapin+++++	centre en 15 caractères avec +
"Lapin.lstrip("Lap")	in	suppression des caractères Lap à gauche
"Lapin ".rstrip(" ")	Lapin	suppression d'espaces à droite

Suite des méthodes ...

"Lapin".center(15,"+")	+++++Lapin+++++	centre en 15 caractères avec +
"Lapin.lstrip("Lap")	in	suppression des caractères Lap à gauche
"Lapin ".rstrip(" ")	Lapin	suppression d'espaces à droite
" Lapin ".strip()	Lapin	suppression des espaces des 2 cotés

Suite des méthodes ...

"Lapin".center(15,"+")	+++++Lapin+++++	centre en 15 caractères avec +
"Lapin.lstrip("Lap")	in	suppression des caractères Lap à gauche
"Lapin ".rstrip(" ")	Lapin	suppression d'espaces à droite
" Lapin ".strip()	Lapin	suppression des espaces des 2 cotés
"un lapin".split()	['un', 'lapin']	découpe selon les espaces

Suite des méthodes ...

"Lapin".center(15,"+")	+++++Lapin+++++	centre en 15 caractères avec +
"Lapin.lstrip("Lap")	in	suppression des caractères Lap à gauche
"Lapin ".rstrip(" ")	Lapin	suppression d'espaces à droite
" Lapin ".strip()	Lapin	suppression des espaces des 2 cotés
"un lapin".split()	['un', 'lapin']	découpe selon les espaces
"un lapin".split('n')	['u', ' Lapi', '']	découpe selon le caractère n

Méthodes qui renvoient un booléen.

Méthodes qui renvoient un booléen.

On peut tester si une chaîne contient ou pas :

Méthodes qui renvoient un booléen.

On peut tester si une chaîne contient ou pas :

- que des majuscules

Méthodes qui renvoient un booléen.

On peut tester si une chaîne contient ou pas :

- que des majuscules
- que des minuscules

Méthodes qui renvoient un booléen.

On peut tester si une chaîne contient ou pas :

- que des majuscules
- que des minuscules
- que des mots qui commencent avec une majuscule

Méthodes qui renvoient un booléen.

On peut tester si une chaîne contient ou pas :

- que des majuscules
- que des minuscules
- que des mots qui commencent avec une majuscule
- que des caractères alphabétiques

Méthodes qui renvoient un booléen.

On peut tester si une chaîne contient ou pas :

- que des majuscules
- que des minuscules
- que des mots qui commencent avec une majuscule
- que des caractères alphabétiques
- que des caractères numériques, etc.

Méthodes qui renvoient un booléen.

On peut tester si une chaîne contient ou pas :

- que des majuscules
- que des minuscules
- que des mots qui commencent avec une majuscule
- que des caractères alphabétiques
- que des caractères numériques, etc.

On peut tester si une chaîne :

Méthodes qui renvoient un booléen.

On peut tester si une chaîne contient ou pas :

- que des majuscules
- que des minuscules
- que des mots qui commencent avec une majuscule
- que des caractères alphabétiques
- que des caractères numériques, etc.

On peut tester si une chaîne :

- commence par ...

Méthodes qui renvoient un booléen.

On peut tester si une chaîne contient ou pas :

- que des majuscules
- que des minuscules
- que des mots qui commencent avec une majuscule
- que des caractères alphabétiques
- que des caractères numériques, etc.

On peut tester si une chaîne :

- commence par ...
- finit par ...

Méthodes	renvoie ...	explications

Méthodes	renvoie ...	explications
"un lapin".islower()	True	est-ce en minuscule ?

Méthodes	renvoie ...	explications
"un lapin".islower()	True	est-ce en minuscule ?
"Un LAPIN".isupper()	False	est-ce en majuscule ?

Méthodes	renvoie ...	explications
"un lapin".islower()	True	est-ce en minuscule ?
"Un LAPIN".isupper()	False	est-ce en majuscule ?
"Un Vieux Lapin".istitle()	True	Mots commencent par une majuscule ?

Méthodes	renvoie ...	explications
"un lapin".islower()	True	est-ce en minuscule ?
"Un LAPIN".isupper()	False	est-ce en majuscule ?
"Un Vieux Lapin".istitle()	True	Mots commencent par une majuscule ?
"3 lapins".isalpha()	False	Ne contient que de l'alphabétique ?

Méthodes	renvoie ...	explications
"un lapin".islower()	True	est-ce en minuscule ?
"Un LAPIN".isupper()	False	est-ce en majuscule ?
"Un Vieux Lapin".istitle()	True	Mots commencent par une majuscule ?
"3 lapins".isalpha()	False	Ne contient que de l'alphabétique ?
"2020".isdigit()	True	Ne contient que des chiffres ?

Méthodes	renvoie ...	explications
"un lapin".islower()	True	est-ce en minuscule ?
"Un LAPIN".isupper()	False	est-ce en majuscule ?
"Un Vieux Lapin".istitle()	True	Mots commencent par une majuscule ?
"3 lapins".isalpha()	False	Ne contient que de l'alphabétique ?
"2020".isdigit()	True	Ne contient que des chiffres ?
"lapin".startswith('z')	False	Commence par z ?

Méthodes	renvoie ...	explications
"un lapin".islower()	True	est-ce en minuscule ?
"Un LAPIN".isupper()	False	est-ce en majuscule ?
"Un Vieux Lapin".istitle()	True	Mots commencent par une majuscule ?
"3 lapins".isalpha()	False	Ne contient que de l'alphabétique ?
"2020".isdigit()	True	Ne contient que des chiffres ?
"lapin".startswith('z')	False	Commence par z ?
"lapin".endswith('in')	True	Finit par 'in'

Méthodes qui recherche une chaîne dans une chaîne.

Méthodes qui recherche une chaîne dans une chaîne.

Deux méthodes sont disponibles qui se déclinent chacune en deux variantes.

Méthodes qui recherche une chaîne dans une chaîne.
Deux méthodes sont disponibles qui se déclinent chacune en deux variantes.

code find

find("motif", début, fin)

- qui cherche la chaîne "*motif*" depuis l'index *début* et jusqu'à l'index *fin*. Si rien n'est spécifié, début = 0 et fin = la fin de la chaîne. La recherche s'effectue de gauche à droite.

Méthodes qui recherche une chaîne dans une chaîne.
Deux méthodes sont disponibles qui se déclinent chacune en deux variantes.

code find

find("motif", début, fin)

- qui cherche la chaîne "*motif*" depuis l'index *début* et jusqu'à l'index *fin*. Si rien n'est spécifié, début = 0 et fin = la fin de la chaîne. La recherche s'effectue de gauche à droite.

code rfind

rfind("motif", début, fin)

- qui fait la même chose mais de droite à gauche.

La méthode **find()** (et **rfind()**) renvoie -1 si rien n'est trouvé.

La méthode **find()** (et **rfind()**) renvoie -1 si rien n'est trouvé.

code index

index("motif", début, fin)

- qui fait la même chose que find()

La méthode **find()** (et **rfind()**) renvoie -1 si rien n'est trouvé.

code index

index("motif", début, fin)

- qui fait la même chose que find()

code rindex

rindex("motif", début, fin)

- qui fait la même chose que rfind().

La méthode **find()** (et **rfind()**) renvoie -1 si rien n'est trouvé.

code index

index("motif", début, fin)

- qui fait la même chose que find()

code rindex

rindex("motif", début, fin)

- qui fait la même chose que rfind().

La différence réside dans le fait que index (et rindex) renvoie une erreur si rien n'est trouvé.

Exemples :

Exemples :

Entrée [31]:

```
1 texte = "un chien court après un chat dans le champ"
2 motif1 = "chien"
3 motif2 = "lapin"
4 print("Motif trouvé à l'indice :",texte.find(motif1))
5 print("Motif trouvé à l'indice :",texte.find(motif2, 5, 25))
6 print("-"*10)
7 print("Motif trouvé à l'indice :",texte.index(motif1))
8 print("Motif trouvé à l'indice :",texte.index(motif2))
```

```
Motif trouvé à l'indice : 3
Motif trouvé à l'indice : -1
----------
Motif trouvé à l'indice : 3

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-31-e6d67be5f14a> in <module>
      6 print("-"*10)
      7 print("Motif trouvé à l'indice :",texte.index(motif1))
----> 8 print("Motif trouvé à l'indice :",texte.index(motif2))

ValueError: substring not found
```

▸ Exercice 3

Formatage de chaînes.

Formatage de chaînes.

Pour afficher une chaîne de caractères avec print(), plusieurs solutions existent pour son formatage, c'est-à-dire la façon dont la chaîne à afficher va être construite.

Formatage de chaînes.
Pour afficher une chaîne de caractères avec print(), plusieurs solutions existent pour son formatage, c'est-à-dire la façon dont la chaîne à afficher va être construite.

- Opérateur % :

Formatage de chaînes.
Pour afficher une chaîne de caractères avec print(), plusieurs solutions existent pour son formatage, c'est-à-dire la façon dont la chaîne à afficher va être construite.

- Opérateur % :
  syntaxe :

Opérateur %

```
"J'ai %d %s dans mon %s" % (3, "lapins", "chapeau.")
```

donne

Formatage de chaînes.
Pour afficher une chaîne de caractères avec print(), plusieurs solutions existent pour son formatage, c'est-à-dire la façon dont la chaîne à afficher va être construite.

- Opérateur % :
  syntaxe :

Opérateur %

```
"J'ai %d %s dans mon %s" % (3, "lapins", "chapeau.")
```

donne
J'ai 3 lapins dans mon chapeau.
(%d, %s, ... sont des spécificateurs et il y en a plein d'autres...)

- Avec *format* et *print()*

- Avec *format* et *print()*
  Chaque paire d'accolades (numérotées ou pas) désigne un champ

- Avec *format* et *print()*
  Chaque paire d'accolades (numérotées ou pas) désigne un champ

format

```
print("Un {} {} vers {}".format("lapin", "court", "moi")
```

donne :

- Avec *format* et *print()*
  Chaque paire d'accolades (numérotées ou pas) désigne un champ

format

```
print("Un {} {} vers {}".format("lapin", "court", "moi")
```

donne :
Un lapin court vers moi

- Avec *format* et *print()*
  Chaque paire d'accolades (numérotées ou pas) désigne un champ

format

```
print("Un {} {} vers {}".format("lapin", "court", "moi")
```

donne :
Un lapin court vers moi

format

```
print("Un {2} {0} vers {1}".format("court", "le chat", "chien")
```

donne :

- Avec *format* et *print()*
  Chaque paire d'accolades (numérotées ou pas) désigne un champ

format

```
print("Un {} {} vers {}".format("lapin", "court", "moi")
```

donne :
Un lapin court vers moi

format

```
print("Un {2} {0} vers {1}".format("court", "le chat", "chien")
```

donne :
Un chien court vers le chat

- Avec *format* et *print()*
  Chaque paire d'accolades (numérotées ou pas) désigne un champ

format

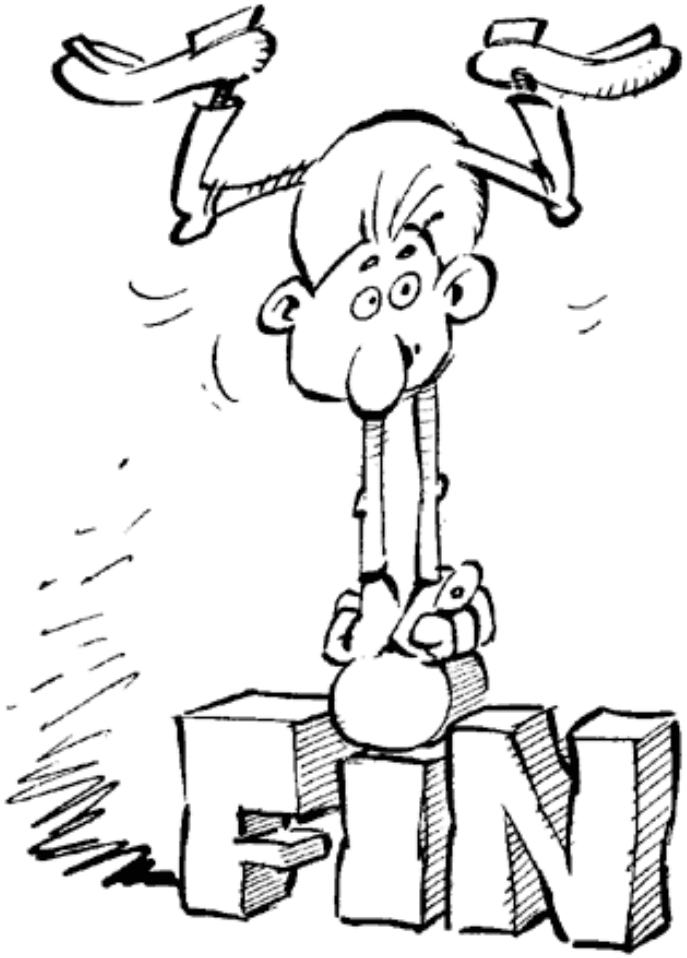
```
print("Un {} {} vers {}".format("lapin", "court", "moi")
```

donne :
Un lapin court vers moi

format

```
print("Un {2} {0} vers {1}".format("court", "le chat", "chien")
```

donne :
Un chien court vers le chat
Les capacités de *format* sont encore plus étendues (voir la documentation attitrée)


FIN

Exercice :

Exercice :

Que fait le code suivant :

Exercice :

Que fait le code suivant :

exo 1-a

```
nb1 = 5
nb2, tva = 7, 0.2
print("Le montant est", (nb1 + nb2)*(1 + tva))
```

Exercice :

Que fait le code suivant : affiche "Le montant est 14.4"

exo 1-a

```
nb1 = 5
nb2, tva = 7, 0.2
print("Le montant est", (nb1 + nb2)*(1 + tva))
```

et celui-ci :

Exercice :

Que fait le code suivant : affiche "Le montant est 14.4"

exo 1-a

```
nb1 = 5
nb2, tva = 7, 0.2
print("Le montant est", (nb1 + nb2)*(1 + tva))
```

et celui-ci :

exo 1-b

```
cote = 6
nom = "carré"
print("Le", nom, "a pour aire", cote*cote)
```

Exercice :

Que fait le code suivant : affiche "Le montant est 14.4"

exo 1-a

```
nb1 = 5
nb2, tva = 7, 0.2
print("Le montant est", (nb1 + nb2)*(1 + tva))
```

et celui-ci : affiche "Le carré a pour aire 36"

exo 1-b

```
cote = 6
nom = "carré"
print("Le", nom, "a pour aire", cote*cote)
```

Retour au cours

Exercice :

Exercice :

Que fait le code suivant :

Exercice :

Que fait le code suivant :

exo 2-a

```
2 + 3 != 7 - 2
```

Exercice :

Que fait le code suivant : renvoie *False* car 5 == 5

exo 2-a

```
2 + 3 != 7 - 2
```

et celui-ci :

Exercice :

Que fait le code suivant : renvoie *False* car 5 == 5

exo 2-a

```
2 + 3 != 7 - 2
```

et celui-ci :

exo 2-b

```
print("lapin" > "tigre")
print( 0 == False)
```

Exercice :

Que fait le code suivant : renvoie *False* car 5 == 5

exo 2-a

```
2 + 3 != 7 - 2
```

et celui-ci : renvoie *False* puis *True*

exo 2-b

```
print("lapin" > "tigre")
print( 0 == False)
```

et enfin :

Exercice :

Que fait le code suivant : renvoie *False* car 5 == 5

exo 2-a

```
2 + 3 != 7 - 2
```

et celui-ci : renvoie *False* puis *True*

exo 2-b

```
print("lapin" > "tigre")
print( 0 == False)
```

et enfin :

exo 3-c

```
not ("lapin" <= "laponie") == (3 + 5 != 4 * 2)
```

Exercice :

Que fait le code suivant : renvoie *False* car 5 == 5

exo 2-a

```
2 + 3 != 7 - 2
```

et celui-ci : renvoie *False* puis *True*

exo 2-b

```
print("lapin" > "tigre")
print( 0 == False)
```

et enfin : renvoie *True*

exo 3-c

```
not ("lapin" <= "laponie") == (3 + 5 != 4 * 2)
```

Retour au cours

Exercice :

Exercice : Que fait le code suivant :

Exercice : Que fait le code suivant :

exo 3-a

```
text1 = "animal"
text1[2 :4].startswith("i")
```

Exercice : Que fait le code suivant : renvoie *True*

exo 3-a

```
text1 = "animal"
text1[2 :4].startswith("i")
```

et celui-ci :

Exercice : Que fait le code suivant : renvoie *True*

exo 3-a

```
text1 = "animal"
text1[2 :4].startswith("i")
```

et celui-ci :

exo 3-b

```
text2 = "éléphant"
print(("é" in text2) and len(text2) == 8)
```

Exercice : Que fait le code suivant : renvoie *True*

exo 3-a

```
text1 = "animal"
text1[2 :4].startswith("i")
```

et celui-ci : renvoie *True*

exo 3-b

```
text2 = "éléphant"
print(("é" in text2) and len(text2) == 8)
```

et enfin :

Exercice : Que fait le code suivant : renvoie *True*

exo 3-a

```
text1 = "animal"
text1[2 :4].startswith("i")
```

et celui-ci : renvoie *True*

exo 3-b

```
text2 = "éléphant"
print(("é" in text2) and len(text2) == 8)
```

et enfin :

exo 3-c

```
text3 = "On rentre du boulot !".upper()
print(text3.split())
```

Exercice : Que fait le code suivant : renvoie *True*

exo 3-a

```
text1 = "animal"
text1[2 :4].startswith("i")
```

et celui-ci : renvoie *True*

exo 3-b

```
text2 = "éléphant"
print(("é" in text2) and len(text2) == 8)
```

et enfin : renvoie ["ON","RENTRE","DU","BOULOT"," !"]

exo 3-c

```
text3 = "On rentre du boulot !".upper()
print(text3.split())
```

Retour au cours