



Python

Les Bases

Par Richard BONNAMY



Introduction

Python- Caractéristiques

➤ Un langage interprété

- Python est un langage interprété, ce qui signifie qu'il exécute directement le code ligne par ligne.
- S'il y a des erreurs dans le code du programme, celui-ci s'arrête de fonctionner. Les programmeurs peuvent donc trouver rapidement les erreurs dans le code.

➤ Un langage facile à utiliser

- Python utilise des mots qui ressemblent à l'anglais.
- Contrairement à d'autres langages de programmation, Python n'utilise pas les accolades. Au lieu de cela, il utilise l'indentation.

➤ Un langage à typage dynamique

- Les programmeurs ne doivent pas déclarer les types de variables lors de l'écriture du code, car Python les détermine au moment de l'exécution. Grâce à cela, vous pouvez écrire des programmes Python plus rapidement.

Python- Introduction

➤ Un langage orienté objet

- Python considère tout comme un objet, mais il prend également en charge d'autres types de programmation tels que la programmation fonctionnelle.

Python- Positionnement



Web

Estelle Raffin / Publié le 11 avril 2022 à 10h54

Apr 2022	Apr 2021	Change	Programming Language	Ratings	Change
1	3	▲	Python	13.92%	+2.88%
2	1	▼	C	12.71%	-1.61%
3	2	▼	Java	10.82%	-0.41%
4	4		C++	8.28%	+1.14%
5	5	■	C#	6.82%	+1.91%
6	6		Visual Basic	5.40%	+0.85%
7	7		JavaScript	2.41%	-0.03%
8	8		Assembly language	2.35%	+0.03%
9	10	▲	SQL	2.28%	+0.45%
10	9	▼	PHP	1.64%	-0.19%
11	16	▲	R	1.55%	+0.44%
12	12		Delphi/Object Pascal	1.18%	-0.29%
13	14	▲	Go	1.09%	-0.14%
14	15	▲	Swift	1.00%	-0.19%
15	13	▼	Ruby	0.88%	-0.35%
16	11	▼	Classic Visual Basic	0.83%	-0.71%
17	23	▲	Objective-C	0.82%	+0.15%
18	18		Perl	0.79%	-0.21%
19	37	▲	Lua	0.78%	+0.45%
20	19	▼	MATLAB	0.74%	-0.25%

<https://f.hellomwork.com/blondumoderateur/2022/04/classement-tiobe-avril-2022.inn>

En avril 2022, le langage R gagne 5 places au classement par rapport à l'an dernier. © TIOBE

[Source BDM 11/04/2022](#)

Python- plusieurs IDE disponibles

- **Utilisation en mode console - à partir du site officiel (il est gratuit) :**
<https://www.python.org/downloads/>
 - Python peut s'installer sous Windows, Linux/UNIX et MacOS.

- **IDE Spyder via Distribution Anaconda :**
 - Avec une version récente de Python : 3.11 pour Anaconda3 2023-07-2
 - [Free Download | Anaconda](#)

- **IDE PyCharm de JetBrains (community edition) :**
 - avec une version plus ancienne de Python : 3.8 pour PyCharm 2022.2
 - <https://www.jetbrains.com/fr-fr/edu-products/download/other-PCE.html>

Python- Indentation


- Dans la plupart des langages, la structuration du code se fait via des blocs de code.
- En C++, C#, Java et PHP les blocs de code sont délimités par des accolades. L'indentation du code est recommandée mais facultative.
- En Python, c'est **l'indentation** qui est utilisée pour définir des blocs de code.
- Si on indente mal notre code Python, celui-ci ne s'exécutera tout simplement pas et renverra une erreur.

Python- Indentation

- Le nombre de caractères est arbitraire pour un bloc donné
- Mais, l'indentation d'un bloc doit être homogène


```
a = 12
if a>10:
    print("Je suis un bloc")
    print("Je suis indenté")

print(a)
```



```
a = 12
if a>10:
    print("Je suis un bloc")
    print("Je suis indenté")

print(a)
```



```
Run: bases_test_indentation x
Je suis un bloc
Je suis indenté
12
```

```
Run: bases_test_indentation x
File "C:\Users\RichardBONNAMY\PycharmP
    print("Je suis indenté")
    ^
IndentationError: unexpected indent
```


TP n°1: installez PyCharm et créez un 1^{er} projet



Les bases

Python- Les bases

- Le rôle d'un programme informatique est de manipuler des données, des informations.
- Il existe différents types de données primitives en Python.
- **Principaux types de données :**
 - les nombres : entiers (int), réels (float)
 - les chaînes de caractères (str)
 - les booléens (bool)

Python- Les bases

➤ Exemple de données de chaque type :

- int :	5	127	-16	89652333
- float :	5.0	263.1	-17.98	12.25e8
- str :	"Hello !"	'Hello !'	""""Hello !""""	"""Hello !"""
- bool :	True	False		

Python- Les bases

➤ Les autres types de données en Python :

Catégorie	Type
Texte	<i>str</i>
Nombres	<i>int, float, bool, complex</i>
Séquences	list, tuple, range
Mapping	dict
Set	set, frozenset
Binary	bytes, bytearray, memoryview
None Type	NoneType

Python- Les variables

- Pour manipuler les données, quel que soit le langage, il faut stocker les données dans des **variables**, ce qui revient à nommer les données.
- La **déclaration de variable** utilise l'**opérateur d'affectation** =

```
mon_poids = 75  
mon_nom = "DUPONT"
```

←

Une affectation se lit de droite à gauche : j'affecte la valeur 75 à la variable **mon_poids**

Python- Les variables

➤ Définir des variables :

```
prix_mettre_carre = 3250  
superficie_maison = 98.5
```

➤ Utiliser les variables dans les calculs :

```
prix_maison = prix_mettre_carre * superficie_maison
```

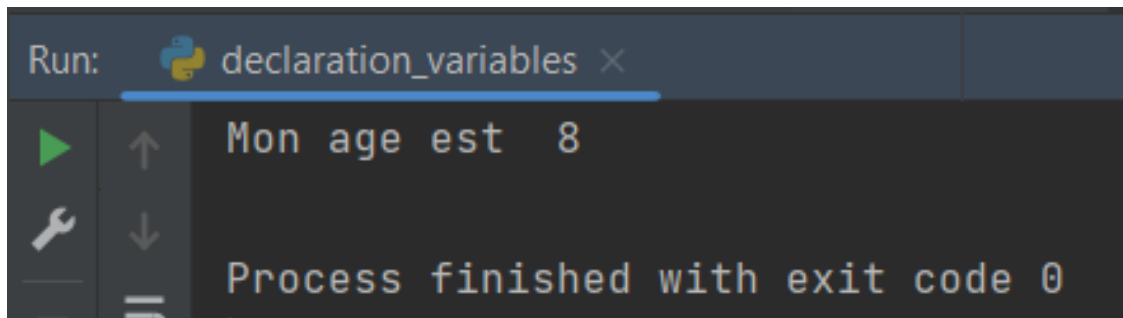
➤ Utiliser les variables dans un appel de fonction :

```
print("Le prix de la maison est de ", prix_maison)
```

Python- Les bases

- On peut modifier (réaffecter) la valeur d'une variable :

```
mon_age = 7  
mon_age = 8  
  
print("Mon age est ", mon_age)
```



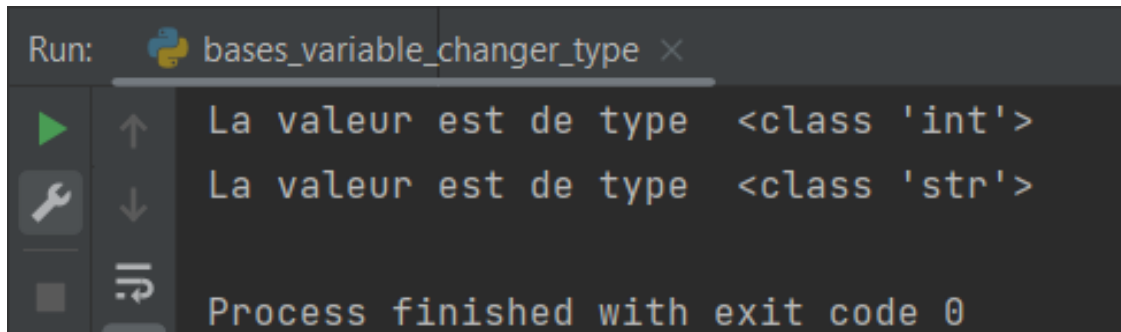
```
Run: declaration_variables ×  
▶ ↑ Mon age est 8  
⚙ ↓  
— Process finished with exit code 0
```


Python- Les bases

- On peut changer le type d'une variable :

```
 valeur = 7
print("La valeur est de type ", type(valeur))

valeur = "Coucou"
print("La valeur est de type ", type(valeur))
```



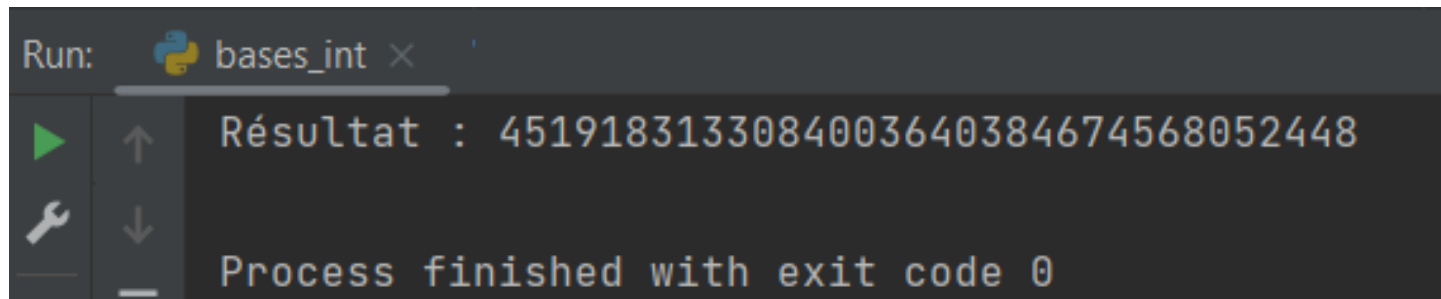
```
Run: bases_variable_changer_type ×
La valeur est de type <class 'int'>
La valeur est de type <class 'str'>
Process finished with exit code 0
```

Python- Les bases

➤ Les nombres entiers – les int :

- En **Python 2** il y avait les types **int** et **long** pour les entiers.
- En **Python 3** il n'y a plus que les **int** pour les entiers
- Il n'y a pas de limite maximum.

```
grosse_valeur = 564897891635500455048084321006556  
print("Résultat :", grosse_valeur * 8)
```



The screenshot shows a terminal window titled 'Run: bases_int'. It contains a green play button icon, a grey up arrow icon, a wrench icon, and a grey down arrow icon. The output of the code is displayed as 'Résultat : 4519183133084003640384674568052448'. Below the output, it says 'Process finished with exit code 0'.

Python- Les bases

➤ Les nombres réels – les float :

- En **Python**, le **caractère décimal** est le **point**.
- Les **floats** sont en double précision, soit jusqu'à **16 chiffres** après la virgule.
- Ils ont une valeur max égale à **1.7976931348623157e+308**.
- Ils ont une valeur min positive égale à **2.2250738585072014e-308**.
- Les **floats** peuvent bien sûr être **négatifs**.

Python- Les bases

➤ Le type str ou chaîne de caractères – construction :

- Les chaînes de caractères sont ce qu'on appelle communément du texte.
- Pour définir une chaîne de caractères ou pour stocker une chaîne de caractères dans une variable, il faudra l'entourer de **guillemets simples** ou **deux doubles** droits.

```
prenom = "Richard"
nom = "BONNAMY"

infos = """
Je suis responsable pédagogique et
formateur Python
"""

print(nom, prenom, infos)
```

Python- Les bases

➤ Le type str - échappement pour les caractères spéciaux:

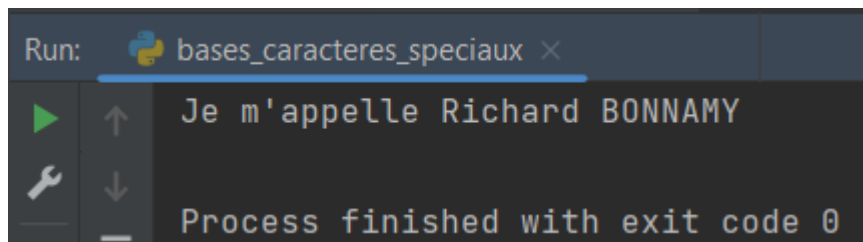
- Si une chaîne de caractères, délimitée par des guillemets simples, contient des guillemets simples, il faut échapper ces derniers afin qu'ils ne soient pas interprétés comme des marqueurs de fin de chaîne. Idem avec guillemets doubles.
- Le caractère d'échappement en Python est l'antislash \.

➤ Exemple incorrect :

```
dire_bonjour = 'Je m'appelle Richard BONNAMY'  
print(dire_bonjour)
```

➤ Exemple correct :

```
dire_bonjour = 'Je m\'appelle Richard BONNAMY'  
print(dire_bonjour)
```



```
Run: bases_caracteres_speciaux x  
▶ ↑ Je m'appelle Richard BONNAMY  
⚙ ↓  
— Process finished with exit code 0
```

Python- Les bases

➤ Le type str - multiligne:

- On peut également utiliser une syntaxe multiligne utilisant des triples guillemets (simples ou doubles) et qui dispense d'avoir à échapper les apostrophes et les guillemets :

```
dire_bonjour1 = """Je m'appelle Jean DUPONT"""  
dire_bonjour2 = '''Je m'appelle Jean DUPONT'''
```

- Le guillemet triple (simple ou double) sert surtout à saisir des chaînes de caractères sur plusieurs lignes.

```
dire_bonjour3 = """  
Je m'appelle  
Jean DUPONT"""
```

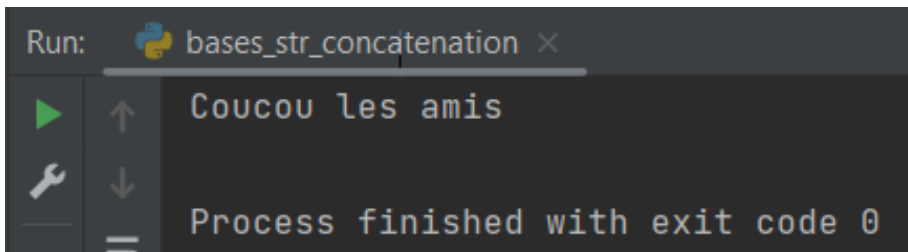
Python- Les bases

➤ Le type str – concaténation :

- Lorsqu'il est utilisé avec deux chaînes, l'opérateur **+** est un opérateur de concaténation et pas d'addition. "Concaténer" signifie "mettre bout à bout".

```
a = "Coucou"  
b = " "  
c = "les amis"  
phrase = a + b + c  
print(phrase)
```

- Dans l'exemple ci-dessus la variable **phrase** contient le résultat de la concaténation des 3 chaînes de caractères a, b et c.
- Python commence par effectuer le calcul à droite du = puis réalise l'affectation une fois le calcul terminé.



```
Run: bases_str_concatenation ×  
Coucou les amis  
Process finished with exit code 0
```

Python- Les bases

➤ La f-string – ou chaine paramétrée :

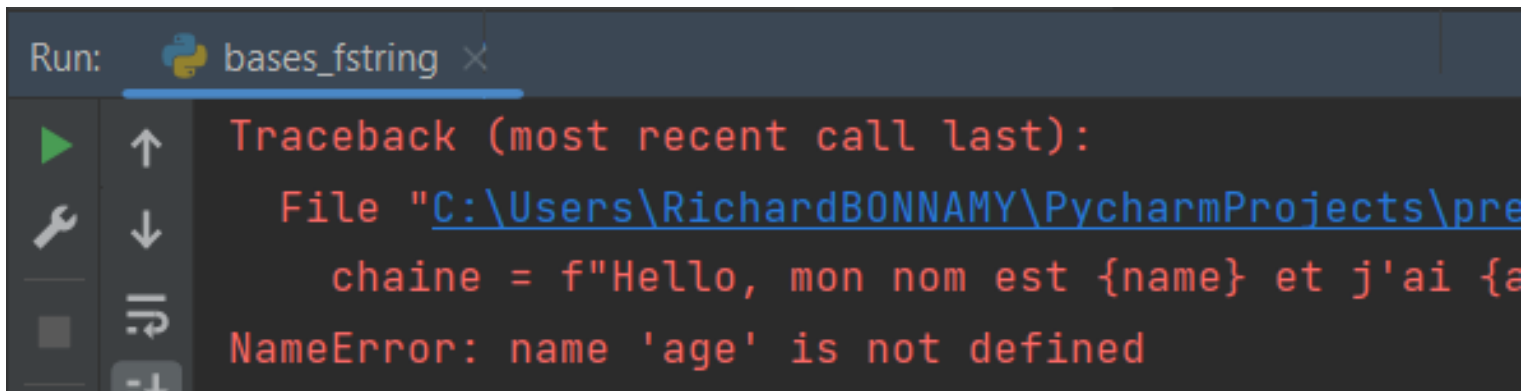
- Apparue en Python 3.6, la **f-string** facilite la construction d'une chaine de caractères et évite les concaténations.
- Pour déclarer une f-string il faut positionner un f à gauche du guillemet ouvrant.
- Elle contient des paramètres nommés entre accolades.

```
name = "Eric"  
age = 27  
chaine = f"Hello, mon nom est {name} et j'ai {age} ans"  
print(chaine)
```


Python- Les bases

- Attention, si un des paramètres de la **f-string**, age, ou name, n'existent pas en tant que variable, alors une erreur est affichée à l'exécution.
- **Exemple :**

```
name = "Eric"  
chaine = f"Hello, mon nom est {name} et j'ai {age} ans"  
print(chaine)
```



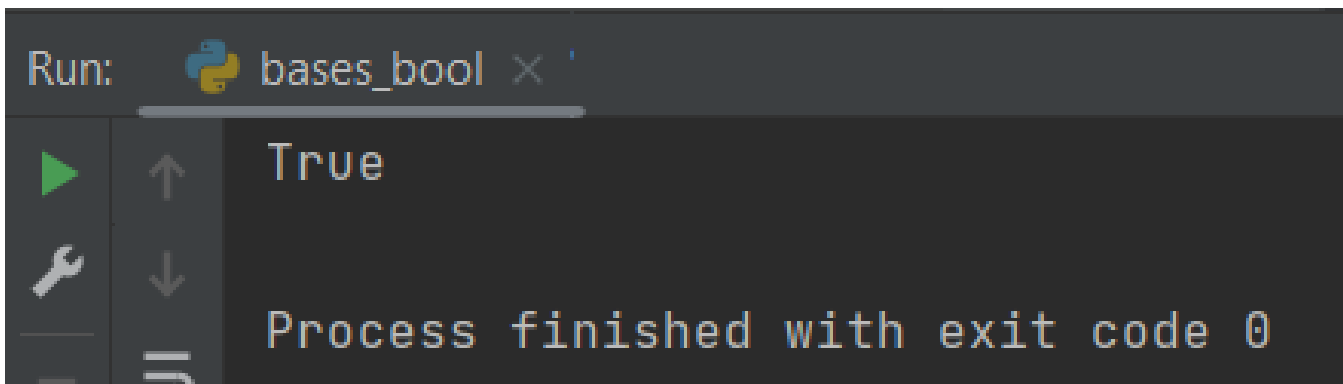
```
Run: bases_fstring ×  
Traceback (most recent call last):  
  File "C:\Users\RichardBONNAMY\PycharmProjects\pre  
    chaine = f"Hello, mon nom est {name} et j'ai {a  
NameError: name 'age' is not defined
```

Python- Les bases

➤ Le type de valeurs bool ou booléen

- Les deux valeurs sont **True** (vrai) et **False** (faux)
- Pas de guillemet !
- Attention à bien utiliser une majuscule pour le premier caractère.

```
b = True  
print(b)
```



Run: bases_bool x

True

Process finished with exit code 0

TP n°2: premier programme en Python

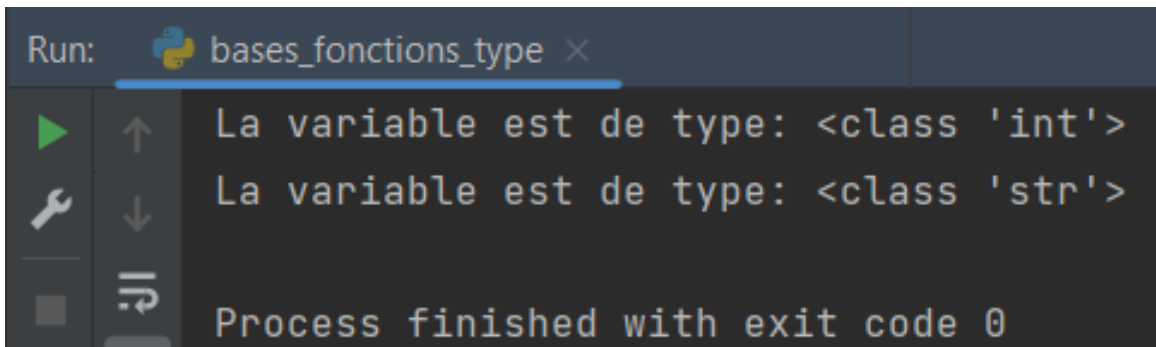
Python- Les bases

➤ Utiliser la fonction `type()` pour connaître le type d'une variable

- Permet de récupérer le type d'une variable.
- On passe la variable à tester en argument de cette fonction.
- La fonction `type(var)` retourne le type de la variable.

```
val1 = 7
print("La variable est de type:", type(val1))

val2 = "Coucou"
print("La variable est de type:", type(val2))
```



```
Run: bases_fonctions_type ×
La variable est de type: <class 'int'>
La variable est de type: <class 'str'>
Process finished with exit code 0
```

Python- Les opérateurs

- Python dispose de nombreux opérateurs qui peuvent être classés selon les catégories suivantes :
 - Les opérateurs arithmétiques ;
 - Les opérateurs d'affectation ou d'assignation ;
 - Les opérateurs de chaînes ;
 - Les opérateurs de comparaison ;
 - Les opérateurs logiques ;
 - Les opérateurs d'identité ;
 - Les opérateurs d'appartenance ;
 - Les opérateurs binaires.

Python- Les opérateurs arithmétiques

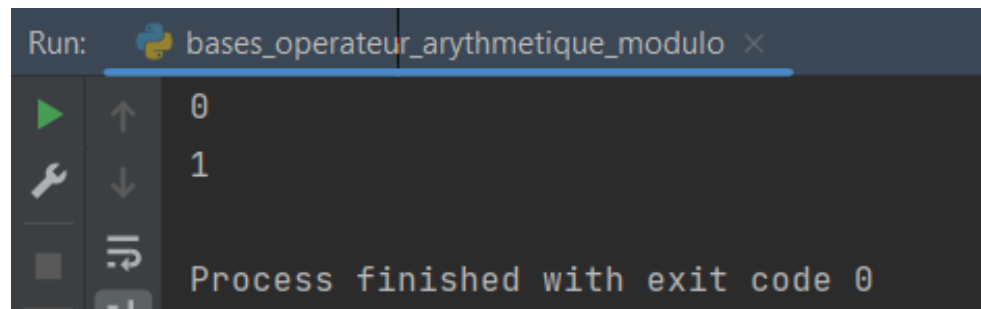
- Les **opérateurs arithmétiques** sont utilisés pour effectuer des opérations mathématiques de base comme des additions, soustractions, multiplication, etc. entre différentes variables contenant des valeurs numériques.
- Python reconnaît et accepte les opérateurs arithmétiques suivants :

Opérateur	Nom
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo
**	Puissance
//	Division entière

Python- Opérateur modulo %

- Le modulo correspond au reste d'une division entière (euclidienne)
- Exemples :
 - **14%2 vaut 0** car lorsque je divise 14 par 2 le résultat est 7 et il reste 0
 - **15%2 vaut 1** car lorsque je divise 15 par 2 le résultat est 7 et il reste 1
 - **Que vaut 14%3 ?**

```
print(14 % 2)  
print(15 % 2)
```



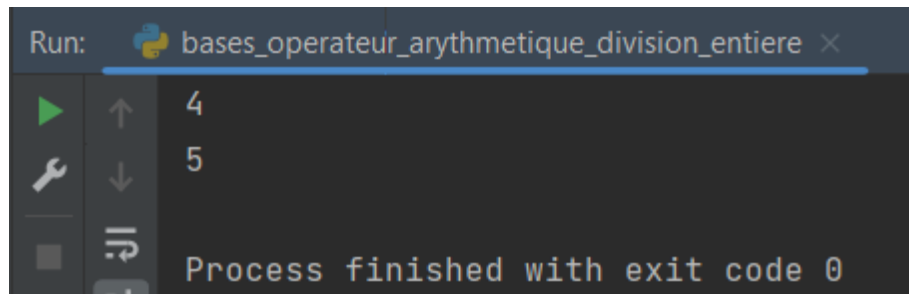
The screenshot shows a terminal window titled "Run: bases_operateur_arithmetique_modulo". It displays the output of the Python code: 0 and 1. Below the output, it says "Process finished with exit code 0".

- Exemple d'application classique du %2 : savoir si un nombre est **pair** ou **impair**

Python- Division entière //

- L'opérateur // permet d'obtenir le résultat entier d'une division (ou la partie entière de ce résultat pour être tout à fait exact).
- Exemples :
 - **14 // 3 vaut 4** car lorsque je divise 14 par 3 le résultat est 4 et il reste 2
 - **15 // 3 vaut 5** car lorsque je divise 15 par 3 le résultat est 5 et il reste 0
 - **Que vaut 16 // 3 ?**

```
print(14 // 3)  
print(15 // 3)
```



```
Run: bases_operateur_arythmetique_division_entiere x  
4  
5  
Process finished with exit code 0
```

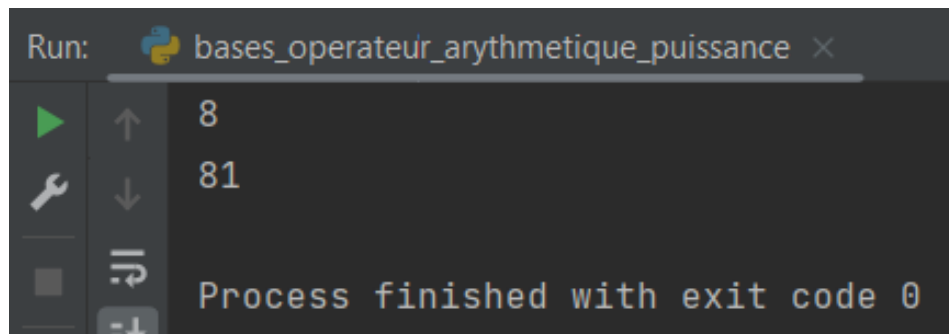

Python- Puissance **

➤ L'opérateur ** permet d'élever un nombre à une certaine puissance.

➤ $a ** b$ équivaut à $\underbrace{a * a * a * \dots * a}_{b \text{ fois}}$

➤ **Exemple :**

```
print(2 ** 3)
print(3 ** 4)
```



```
Run: bases_operateur_arythmetique_puissance x
8
81
Process finished with exit code 0
```

Python- Les opérateurs de chaines

- Les opérateurs de chaines permettent de manipuler des données de type **str** (chaines de caractères) et par extension des variables stockant des données de ce type.
- Python met à notre disposition deux opérateurs de chaine :
 - l'opérateur de **concaténation** +
 - l'opérateur de **répétition** *
- L'opérateur de concaténation permet de mettre bout à bout deux chaines de caractères afin d'en former une troisième, nouvelle.
- L'opérateur de répétition permet de répéter une chaine un certain nombre de fois.

Python- Opérateur de répétition

- L'opérateur * permet de **répéter** une chaîne de caractères n fois.
- **Exemple :**

```
etoile = "*"
chaîne = etoile * 10
print(chaîne)
```

Python- Affectations multiples

- L'opérateur d'affectation permet l'affectation multiple.
- Attention cependant à la lisibilité de votre code

```
name, age = "Eric", 17  
print(name, age)
```

- Dans l'exemple ci-dessus, la première ligne de code correspond à une affectation multiple
- **C'est équivalent au code suivant :**

```
name = "Eric"  
age = 17  
print(name, age)
```

Python- Affectation composée

- Nous connaissons déjà bien l'opérateur d'affectation simple Python `=`. Cet opérateur permet d'affecter ou d'assigner une valeur à une variable.
- Python reconnaît également des opérateurs d'affectation qu'on appelle "composés" et qui combinent l'affectation avec un calcul arithmétique ou logique.
- Les opérateurs `++` et `--` n'existent pas en Python

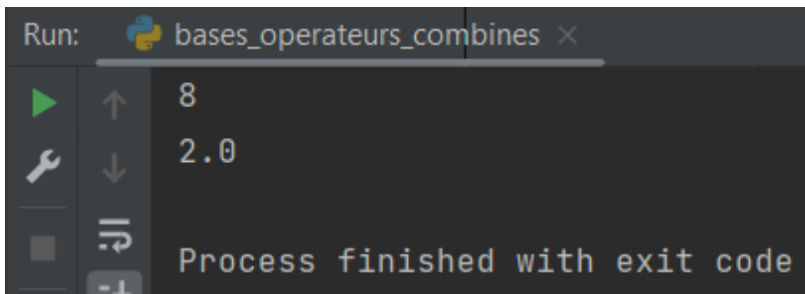
Opérateur	Exemple	Equivalent	Description
<code>=</code>	<code>x = 1</code>	<code>x = 1</code>	Affecte 1 à la variable x
<code>+=</code>	<code>x += 1</code>	<code>x = x + 1</code>	Ajoute 1 à la dernière valeur connue de x et affecte la nouvelle valeur (l'ancienne + 1) à x
<code>-=</code>	<code>x -= 1</code>	<code>x = x - 1</code>	Enlève 1 à la dernière valeur connue de x et affecte la nouvelle valeur à x
<code>*=</code>	<code>x *= 2</code>	<code>x = x * 2</code>	Multiplie par 2 la dernière valeur connue de x et affecte la nouvelle valeur à x
<code>/=</code>	<code>x /= 2</code>	<code>x = x / 2</code>	Divise par 2 la dernière valeur connue de x et affecte la nouvelle valeur à x
<code>%=</code>	<code>x %= 2</code>	<code>x = x % 2</code>	Calcule le reste de la division entière de x par 2 et affecte ce reste à x
<code>//=</code>	<code>x //= 2</code>	<code>x = x // 2</code>	Calcule le résultat entier de la division de x par 2 et affecte ce résultat à x
<code>**=</code>	<code>x **= 4</code>	<code>x = x ** 4</code>	Elève x à la puissance 4 et affecte la nouvelle valeur dans x

Python- Les bases

➤ Exemples avec les opérateurs combinés arithmétiques :

```
x = 5
y = 16
x += 3 # équivalent à x = x + 3
y /= x # équivalent à y = y / x

print(x)
print(y)
```



➤ Les opérateurs d'affectation combinés exotiques

- Il existe également les opérateurs d'affectation combinés plus exotiques: **&=**, **|=**, **^=**, **>>=** et **<<=** qui vont permettre d'effectuer des opérations dont nous discuterons plus tard.

Python- Les opérateurs de comparaison

- Voici ci-dessous les différents opérateurs de comparaison disponibles en Python ainsi que leur signification :

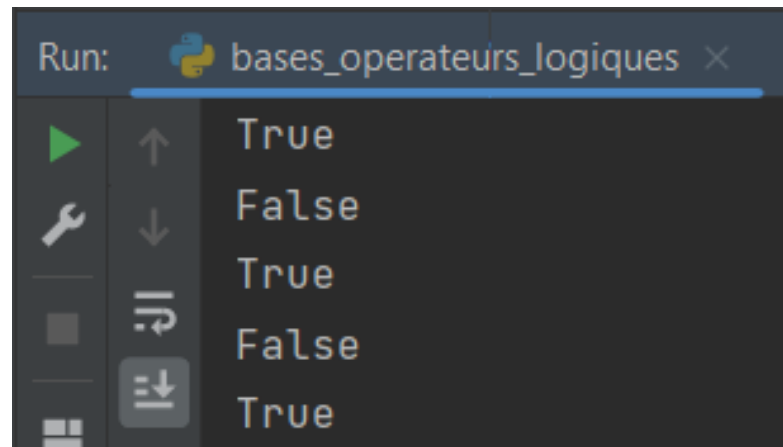
Opérateur	Définition
==	Permet de tester l'égalité en valeur et en type
!=	Permet de tester la différence en valeur ou en type
<	Permet de tester si une valeur est strictement inférieure à une autre
>	Permet de tester si une valeur est strictement supérieure à une autre
<=	Permet de tester si une valeur est inférieure ou égale à une autre
>=	Permet de tester si une valeur est supérieure ou égale à une autre

- Le résultat d'une opération de comparaison est booléenne : **True** ou **False**
- **Une opération de comparaison peut-se lire "est-ce que ?"**

Python- Opérateurs de comparaison - exemples

➤ Exemples :

```
print(4 < 8)
print(4 > 8)
print(4 == 4)
print(4 == "4")
print(5 != 10)
```



➤ Comment lire la première ligne de code ?

- J'affiche le résultat de l'opération de comparaison : Est-ce que 4 est inférieur à 8 ?
- La réponse est Oui donc ce qui équivaut à True pour l'interpréteur Python

Python- Les opérateurs logiques

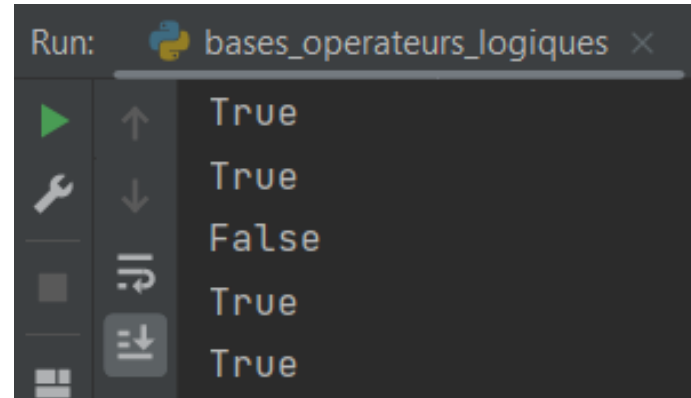
Opérateur	Définition
and	Et logique
or	OU logique
not	NON logique

- Le résultat d'une opération logique est booléenne : **True** ou **False**
- **Une opération logique peut-se lire "est-ce que ?"**

Python- Les opérateurs logiques exemple

➤ Exemples :

```
print(4 < 8 and 2 < 3)
print(4 < 8 or 2 < 3)
print(4 < 8 and 2 > 3)
print(4 < 8 or 2 > 3)
print(4 < 8 and not 2 > 3)
```



➤ Première ligne : est-ce que $4 < 8$ et $2 < 3$?

- Les 2 propositions sont vraies donc l'ensemble est vrai.
- Dans un **ET logique** il faut que les **2 propositions soient vraies** pour que l'ensemble soit vrai

➤ Troisième ligne : Est-ce que $4 < 8$ ou $2 > 3$?

- La seconde proposition est fausse mais la première est vraie donc l'ensemble est vrai
- Dans un **OU logique**, il suffit qu'**une seule des 2 propositions soit vraie** pour que l'ensemble soit vrai. Donc la réponse est bien True.

TP n°3: opérateurs logiques et de comparaison



Les structures de contrôle

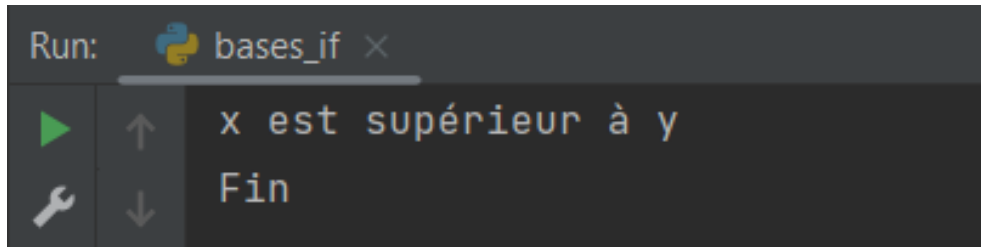
Python- Structure conditionnelle if


- **Structure de contrôle conditionnelle : if**
- Les structures de contrôle conditionnelles (ou plus simplement conditions) permettent d'exécuter un bloc de code ou un autre, selon que des conditions spécifiques sont vérifiées ou pas.
- Python nous fournit les structures conditionnelles suivantes :
 - La condition **if**
 - La condition **if...else**
 - La condition **if...elif...else**

Python- If

- La structure **if** permet d'**exécuter un bloc de code** si et seulement si une **condition est vraie**.
- La structure de contrôle **if** n'exécute le bloc de code que si l'expression vaut True. Dans le cas contraire, le code dans cette structure est ignoré.
- **Exemple :**

```
x = 8
y = 4
if x > y:
    print("x est supérieur à y")
print("Fin")
```



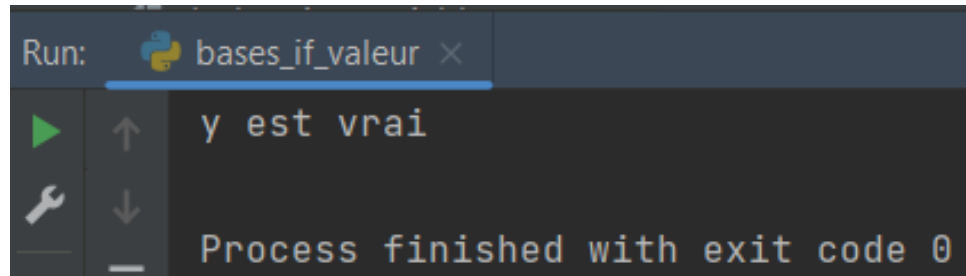
Run:  bases_if ×

▶ ↑ x est supérieur à y
⚙️ ↓ Fin

Python- Test d'une valeur

- Vous pouvez **tester une valeur** directement avec **if**.
- Une valeur "renseignée" est considérée comme valant True.
- Une valeur "non renseignée" est considérée comme False :
 - La valeur 0 (et 0.0)
 - La valeur None
 - Les valeurs chaîne de caractères vide "", liste vide [], dictionnaire vide {} et tuple vide ()
- Exemple :

```
x = 0
y = 1
z = ""
if x:
    print("x est vrai")
if y:
    print("y est vrai")
if z:
    print("z est vrai")
```

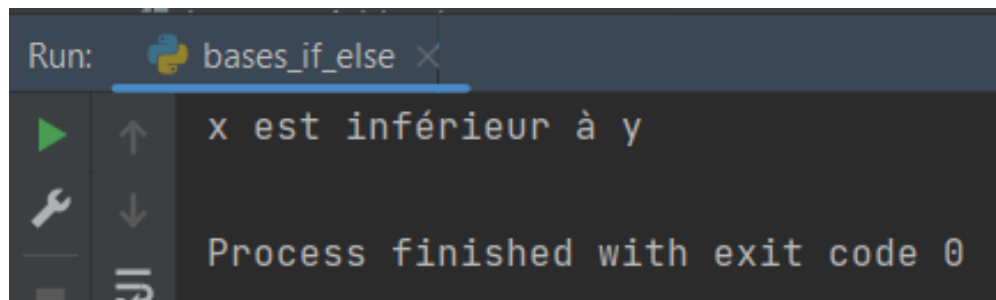


```
Run: bases_if_valeur x
y est vrai
Process finished with exit code 0
```

Python- if else

- La structure conditionnelle **if else** (« si... sinon » en français) permet d'exécuter un premier bloc de code si une expression vaut True ou un autre bloc de code dans le cas contraire.
- La **syntaxe** d'une condition **if else** est la suivante :

```
x = 8
y = 16
if x > y:
    print("x est supérieur à y")
else:
    print("x est inférieur à y")
```

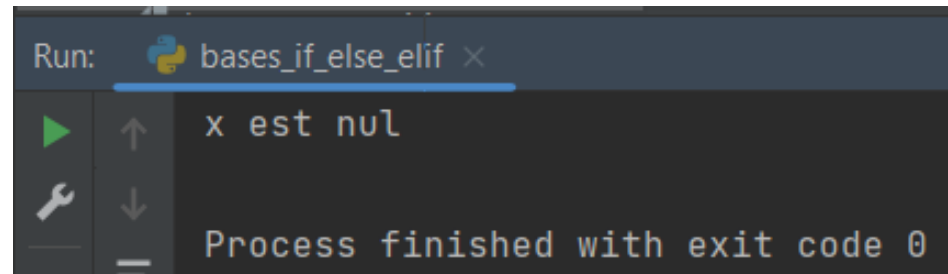


```
Run: bases_if_else x
x est inférieur à y
Process finished with exit code 0
```


Python- if elif else

- La condition **if elif else** (« si...sinon si...sinon ») permet d'effectuer autant d'évaluation que l'on souhaite.
- **elif** permet d'ajouter des **conditions intermédiaires entre la première condition if et le else final**
- Il est possible d'ajouter autant de elif que l'on souhaite.
- Syntaxe :

```
x = 0
if x < 0:
    print("x est négatif")
elif x == 0:
    print("x est nul")
else:
    print("x est positif")
```



Python- Si plusieurs conditions sont vraies

- Si **plusieurs conditions sont vraies** dans une structure **if elif else** alors c'est le bloc de code associé à la **première condition vraie qui sera exécuté**.
- A noter que **else** et **elif** sont facultatifs.

Python- Opérateur ternaire

- L'opérateur ternaire permet de saisir sur une seule ligne la condition suivante :

```
b = 15
a = 0

if b>10:
    a = 1
else:
    a = -1

print(a)
```

- Exemple avec opérateur ternaire :

```
b = 15
a = 0

a = 1 if b > 10 else -1

print(a)
```

Python- Les boucles

- Les **boucles** permettent d'exécuter plusieurs fois un bloc de code tant qu'une condition donnée est vérifiée.
- Utiliser une boucle permet d'écrire le code une seule fois et de le répéter autant qu'on veut.
- On utilise notamment les **boucles pour afficher tous les éléments d'un ensemble** (chaines de caractères, listes, séquences et sets).

Python- Les 2 types de boucles

➤ Les 2 types de boucles :

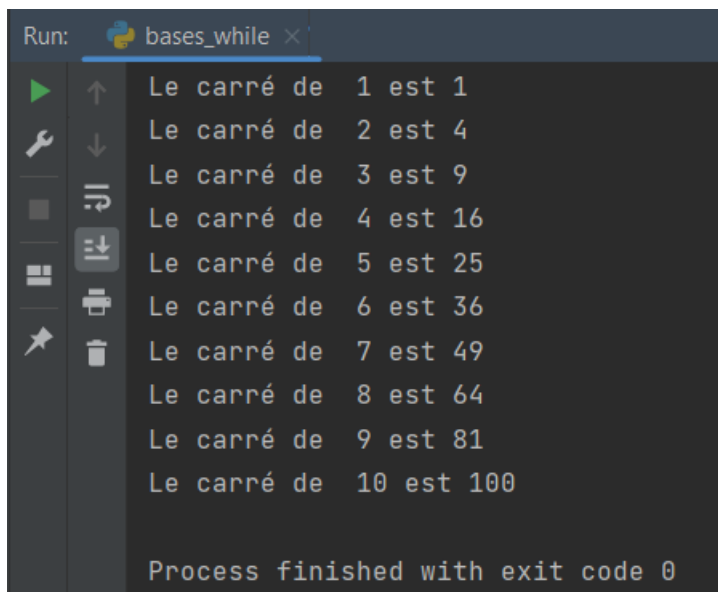
- La boucle **while** ("tant que...") ;
 - La boucle **for** ("pour...").
- Le fonctionnement des boucles est toujours le même : on exécute un bloc de code tant qu'une condition reste vraie.
- Pour éviter de rester bloqué à l'infini dans une boucle, vous pouvez donc déjà noter qu'il faudra que la condition donnée soit fausse à un moment donné (pour pouvoir sortir de la boucle).

Python- La boucle while

- La boucle **while** permet d'exécuter un bloc de code « tant qu'une » condition donnée est vraie.
- Sa syntaxe est la suivante :

```
x = 1
while x <= 10:
    print("Le carré de ", x, "est", x*x)
    x += 1
```

- La variable *x* est initialisée à 1
- Ensuite, **tant que x est inférieur ou égal à 10**, l'exécution du bloc de code est répétée.
- **Pourquoi est-ce que cette boucle while va s'arrêter ?** Parce que *x* augmente de 1 à chaque exécution du bloc.



Run: bases_while x

```
Le carré de 1 est 1
Le carré de 2 est 4
Le carré de 3 est 9
Le carré de 4 est 16
Le carré de 5 est 25
Le carré de 6 est 36
Le carré de 7 est 49
Le carré de 8 est 64
Le carré de 9 est 81
Le carré de 10 est 100

Process finished with exit code 0
```

Python- La boucle for

- La boucle **for** possède une logique et une syntaxe différente de celles des boucles **for** généralement rencontrées dans d'autres langages.
- La boucle **for** permet d'itérer sur les éléments d'une séquence (liste, chaîne de caractères, range, etc.) dans l'ordre de la séquence.
- La condition de sortie dans cette boucle est implicite : on sort de la boucle après avoir parcouru le dernier élément de la séquence.
- **Exemple sur une liste:**

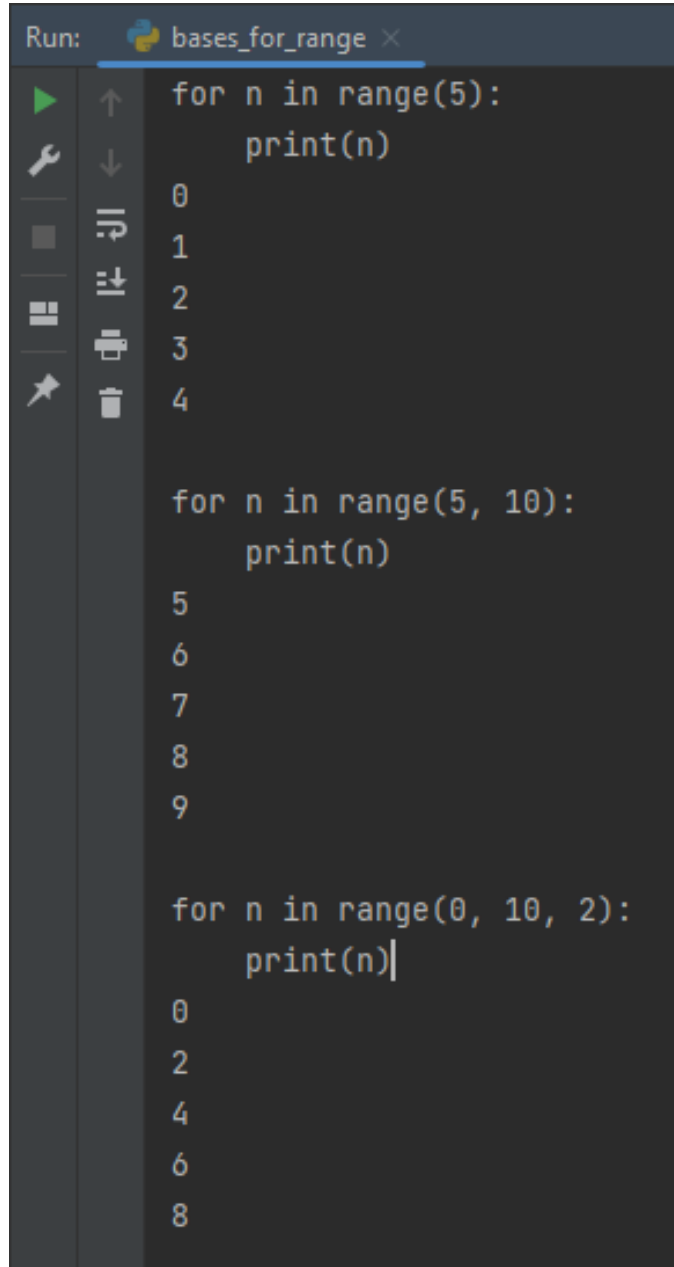
```
liste_languages = ["C", "C#", "Python", "PHP", "Java"]  
for language in liste_languages:  
    print(language)
```

Python- La fonction range()

- On utilise la fonction **range()** pour générer une séquence de nombres entre un **min** et un **max**.
- Attention le **max est exclu**.
- **Exemples :**
 - `range(5)` génère la séquence de 0 à 5 (dernière valeur exclue): 0, 1, 2, 3 et 4.
 - `range(5, 10)` génère la séquence : 5, 6, 7, 8 et 9.
 - `range(0, 10, 2)` génère la séquence de 0 à 10 de 2 en 2 : 0, 2, 4, 6 et 8.
- **Usage courant:**
 - On peut utiliser la fonction **range()** avec **for** pour itérer un certain nombre de fois. Par exemple pour **itérer 100 fois** on utilise **range(100)**.

Python- for avec range - exemples

Exemples :



```
Run: bases_for_range x
for n in range(5):
    print(n)
0
1
2
3
4

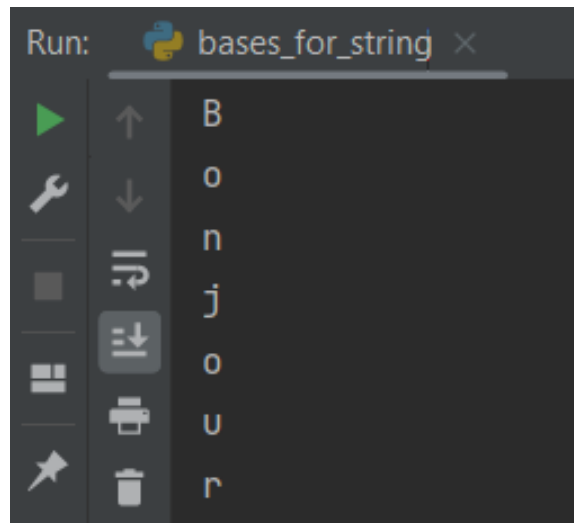
for n in range(5, 10):
    print(n)
5
6
7
8
9

for n in range(0, 10, 2):
    print(n)
0
2
4
6
8
```

Python- for sur une chaine de caractères

- Il est possible d'itérer sur une chaine de caractères
- **Exemple :**

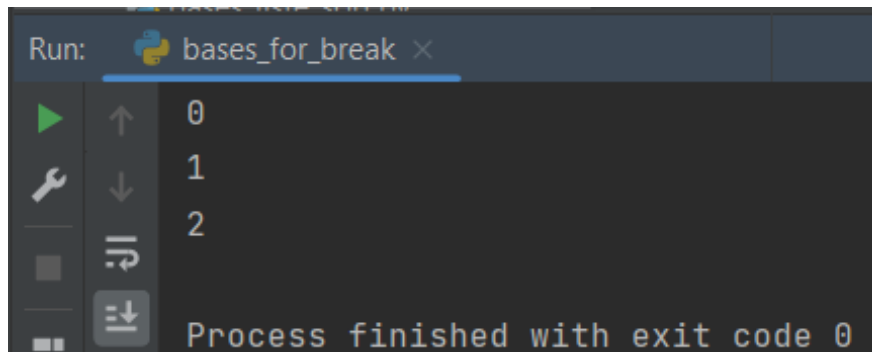
```
chaine = "Bonjour"  
for cc in chaine:  
    print(cc)
```



Python- Instructions break et continue

- L'instruction **break** permet de **stopper l'exécution d'une boucle** lorsqu'une **condition est vérifiée**. On l'inclue souvent dans une **condition de type if**.
- Par exemple, on peut stopper l'exécution d'une boucle lorsqu'une variable contient une valeur en particulier.
- **Exemple :**

```
for i in range(5):  
    if i == 3:  
        break  
    print(i)
```



Run: bases_for_break ×

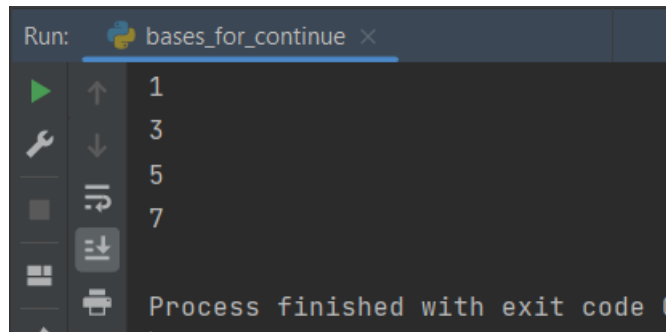
0
1
2

Process finished with exit code 0

Python- L'instruction continue

- L'instruction **continue** permet d'ignorer une itération de la boucle et de passer directement à l'itération suivante.
- **Exemple :**

```
for i in range(9):  
    if i % 2 == 0:  
        continue  
    print(i)
```



```
Run: bases_for_continue x  
1  
3  
5  
7  
Process finished with exit code 0
```

- A chaque fois que *i* est pair, on passe à l'itération suivante, ce qui fait qu'on affiche que les nombres impairs

TP n°4: un peu d'algorithmie



Les listes

Python- Création d'une liste

- Jusqu'à présent, nous n'avons stocké qu'une seule information à la fois dans nos variables.
- Les **listes** peuvent **contenir plusieurs informations**.
- Pour définir une liste, on utilise une paire de **crochets []**.
- Nous plaçons les différents éléments de notre liste entre **crochets** en les séparant par des virgules.
- Exemple d'une **liste de 5 entiers** :

```
liste = [2, -3, 8, 12, -4]  
print(liste)
```

main ×

```
[2, -3, 8, 12, -4]
```

Python- les listes hétérogènes

- On peut stocker n'importe quel type de valeurs dans une liste, comme des chaînes de caractères par exemple :

```
liste = ["C", "C#", "Java", "PHP", "Python"]  
print(liste)
```

```
main ×  
['C', 'C#', 'Java', 'PHP', 'Python']
```

- Les valeurs peuvent aussi contenir des données de types différents, voir une autre liste:

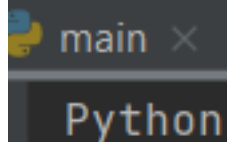
```
liste = [2, "Coucou", -3.5, True, ["Autre", "Liste"]]  
print(liste)
```

```
main ×  
[2, 'Coucou', -3.5, True, ['Autre', 'Liste']]
```


Python- Extraire une valeur via l'index

- Les listes sont **indexées** (i.e. numérotées)
- Les éléments d'une **liste** contenant **n éléments** sont **indexés** de **0** à **n-1**.
- Pour récupérer une valeur dans une liste, on précise le nom de la liste suivi de l'index correspondant entre crochets.

```
liste = ["C", "C#", "Python", "PHP", "Java"]  
print(liste[2])
```



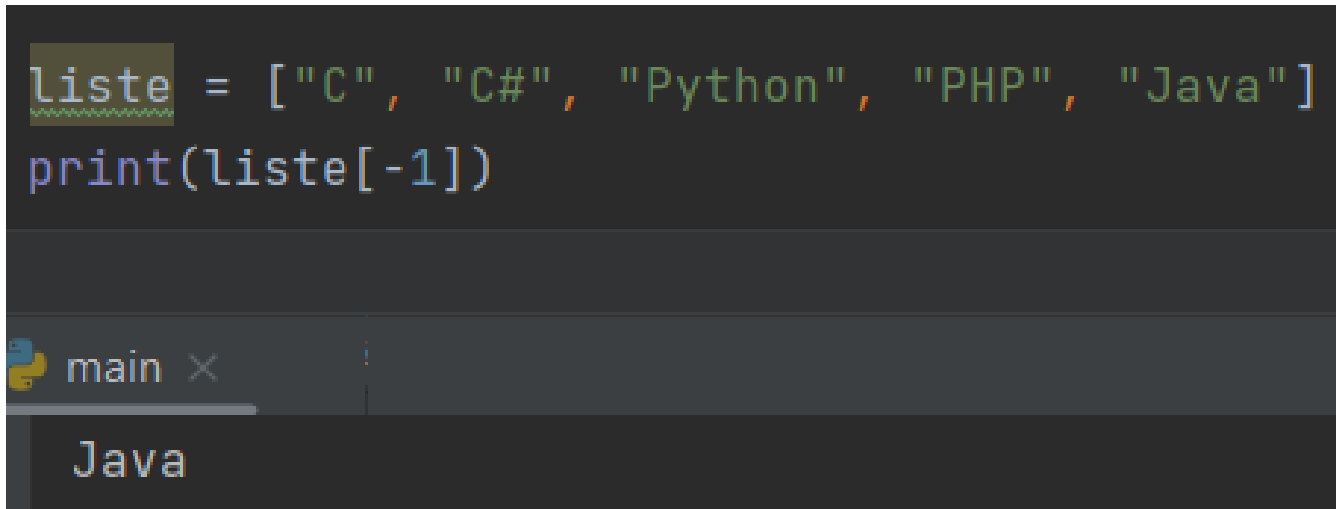
main ×
Python

- Dans la liste ci-dessus, le dernier élément a pour index 4 puisque la liste a 5 éléments.

Python- Index négatif

- Notez que les **index négatifs** sont **acceptés**;
- Dans ce cas on part de la fin de la liste (l'index -1 correspond au dernier élément, -2 à l'avant dernier et etc.).
- **Exemple :**

```
liste = ["C", "C#", "Python", "PHP", "Java"]  
print(liste[-1])
```

A screenshot of a Python IDE. The top part shows a code editor with two lines of Python code: `liste = ["C", "C#", "Python", "PHP", "Java"]` and `print(liste[-1])`. The variable `liste` is highlighted with a yellow background. Below the code editor, there is a terminal window with a tab labeled 'main' and a Python icon. The terminal output shows the word 'Java'.

- Cette utilisation peut intuitive des index est difficile à lire et n'est pas recommandée

Python- Extraire une sous-liste

- On peut récupérer une **sous-liste** à partir d'une liste en précisant les **index min et max**.
- Pour cela, on utilise le **symbole : entre les crochets** avec un index minimum et un index maximum.
 - Si l'index **min** n'est **pas précisé** alors c'est **l'index 0** qui est pris par défaut.
 - Si l'index **max** n'est **pas précisé** alors c'est **l'index le plus grand** qui est pris par défaut..
 - L'index **min** est **inclus**, **max** est **exclus**.

```
liste = ["C", "C#", "Python", "PHP", "Java"]  
print(liste[0:2])  
print(liste[:2])  
print(liste[2:])  
print(liste[:])
```

```
main x  
['C', 'C#']  
['C', 'C#']  
['Python', 'PHP', 'Java']  
['C', 'C#', 'Python', 'PHP', 'Java']
```

Python- Ajouter, modifier, supprimer un élément d'une liste

- Le **contenu** d'une **liste** peut être **modifié**.
- Il est possible **d'ajouter, modifier ou supprimer** une **valeur** de la **liste**.
- Il existe plusieurs techniques pour manipuler le contenu de la liste, notamment l'utilisation de l'opérateur `[]`.
- Les **affectations de tranches** sont possibles :

```
prenoms = ["Khalid", "Laurence", "Pierre", "Coco"]
print(prenoms[2])

# Modification
prenoms[2]="Pierrot"
print(prenoms)

# Ajout d'une tranche vide à partir de l'index 2
prenoms[2:]=[]
print(prenoms)

# Ajout d'une tranche non vide à partir de l'index 2
prenoms[2:4]=["Julia", "Samara", "Axe"]
print(prenoms)

# Effacement du contenu
prenoms[:]=[]
print(prenoms)
```

```
Pierre
['Khalid', 'Laurence', 'Pierrot', 'Coco']
['Khalid', 'Laurence']
['Khalid', 'Laurence', 'Julia', 'Samara', 'Axe']
[]
```

Python- Supprimer un élément avec del

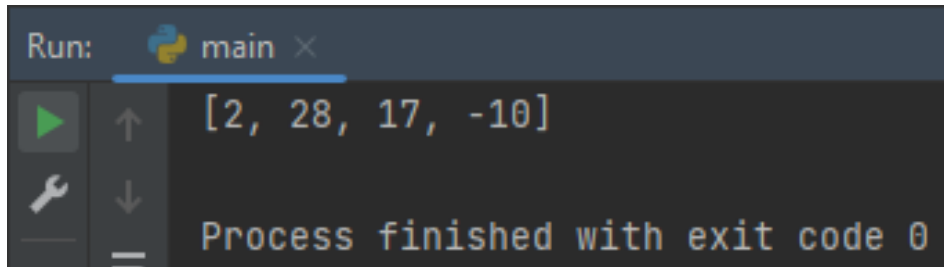
- Il est possible de supprimer un élément dans une liste avec le mot clé **del**

- **Exemple:**

```
liste = [2, 28, -5, 17, -10]
del liste[2]

print(liste)
```

- **Résultat :**



```
Run: main ×
[2, 28, 17, -10]
Process finished with exit code 0
```

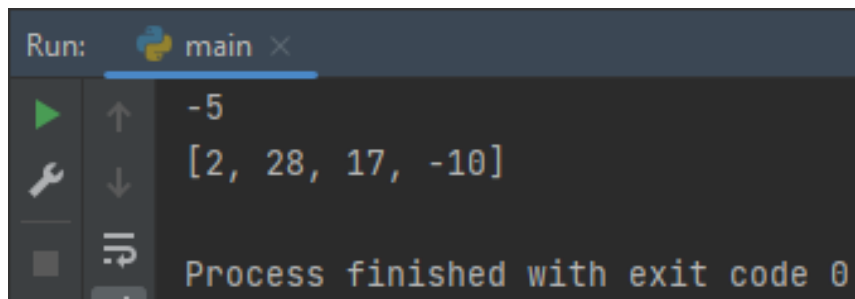
Python- Supprimer un élément avec pop

- Il est possible de supprimer un élément dans la méthode **pop** de la classe list
- Cette méthode extrait également l'élément supprimé
- **Exemple:**

```
liste = [2, 28, -5, 17, -10]
valeur = liste.pop(2)

print(valeur)
print(liste)
```

- **Résultat :**



The screenshot shows a terminal window titled 'Run: main'. It displays the output of the Python code: the value -5 is printed on the first line, followed by the list [2, 28, 17, -10] on the second line. At the bottom, it states 'Process finished with exit code 0'.

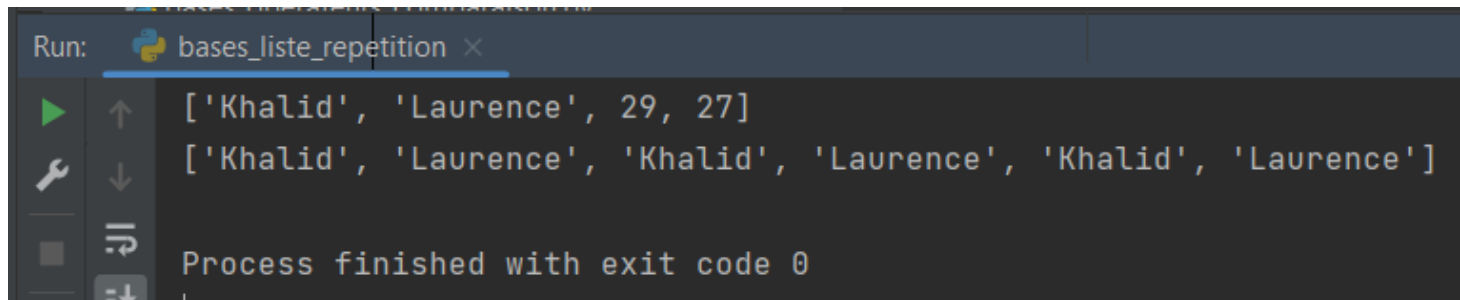
Python- Listes et opérateurs

- On peut utiliser les opérateurs de **concaténation** et de **répétition** avec des listes
- Exemples :

```
prenoms = ["Khalid", "Laurence"]
ages = [29, 27]

# Concaténation
infos = prenoms + ages
print(infos)

# Répétition
prenoms3 = prenoms * 3
print(prenoms3)
```

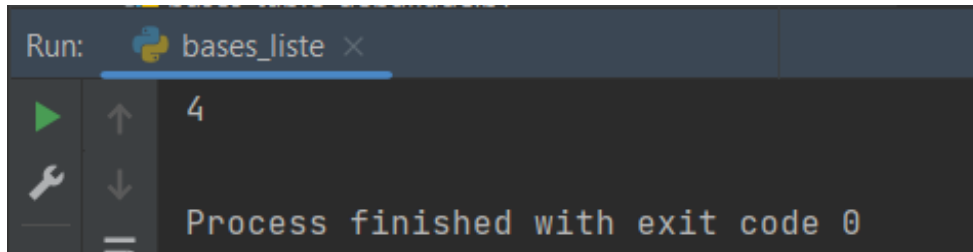


```
Run: bases_liste_repetition x
['Khalid', 'Laurence', 29, 27]
['Khalid', 'Laurence', 'Khalid', 'Laurence', 'Khalid', 'Laurence']
Process finished with exit code 0
```

Python- taille d'une liste

- Pour obtenir la taille d'une liste on utilise la fonction **len(*liste*)** :

```
liste = [17, -2, 14, 28]  
print(len(liste))
```

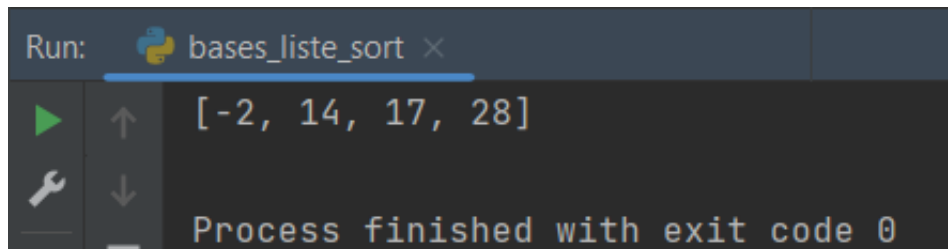


```
Run: bases_liste ×  
4  
Process finished with exit code 0
```


Python- tri d'une liste

- Pour trier une liste on peut utiliser la fonction **sorted(*liste*)**
- Attention **cette fonction ne modifie pas la liste d'origine** mais en crée une nouvelle :

```
liste = [17, -2, 14, 28]
liste_triee = sorted(liste)
print(liste_triee)
```



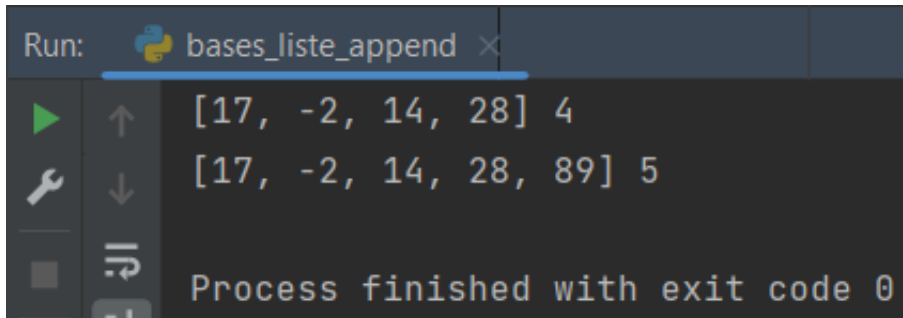
The screenshot shows a terminal window titled "Run: bases_liste_sort". It displays the output of the Python code: `[-2, 14, 17, 28]`. Below the output, it states "Process finished with exit code 0".

Python- les méthodes

- Les listes possèdent des méthodes, c'est-à-dire des fonctions internes appelables à la suite du nom de la liste.
- **Exemple avec la méthode *liste.append(*element*)* pour ajouter un élément :**

```
liste = [17, -2, 14, 28]
print(liste, len(liste))

liste.append(89)
print(liste, len(liste))
```



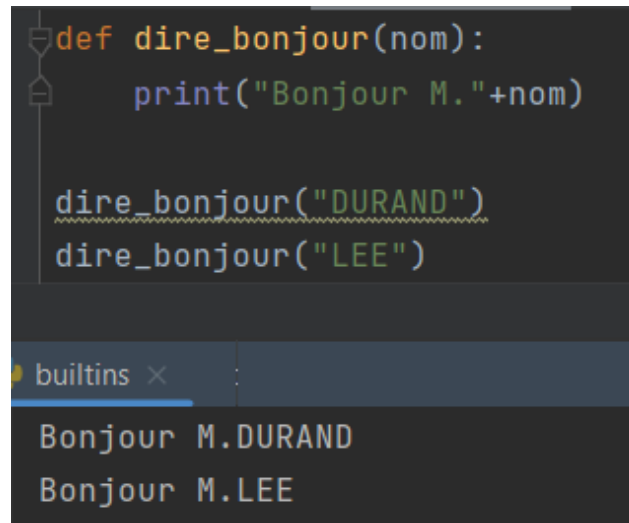
```
Run: bases_liste_append x
[17, -2, 14, 28] 4
[17, -2, 14, 28, 89] 5
Process finished with exit code 0
```



Les fonctions

Python- Définition

- Une fonction est un bloc de code nommé.



```
def dire_bonjour(nom):  
    print("Bonjour M."+nom)  
  
dire_bonjour("DURAND")  
dire_bonjour("LEE")
```

The screenshot shows a Python IDE with a dark background. The code defines a function `dire_bonjour` that takes a parameter `nom` and prints "Bonjour M." followed by the value of `nom`. Below the function definition, the function is called twice: `dire_bonjour("DURAND")` and `dire_bonjour("LEE")`. At the bottom of the IDE, a terminal window shows the output of these calls: "Bonjour M.DURAND" and "Bonjour M.LEE".

- Une **fonction** encapsule **un ensemble d'instructions**.
- Une **fonction** est **appelable** par son **nom**.
- Elle peut posséder ou non des **paramètres entre parenthèses**.
- Il existe deux grands "types" de fonctions en Python : les fonctions prédéfinies (natives ou built-ins) et les fonctions créées par l'utilisateur.

Python- Les fonctions prédéfinies

- Les **fonctions prédéfinies** (built-in ou natives) sont des fonctions mises à disposition par **Python**.
- Dans ce cours, nous avons déjà utilisé des fonctions prédéfinies comme la fonction **print()** ou la fonction **type()** par exemple.
- Il existe de nombreuses fonctions prédéfinies

Python- Les fonctions prédéfinies

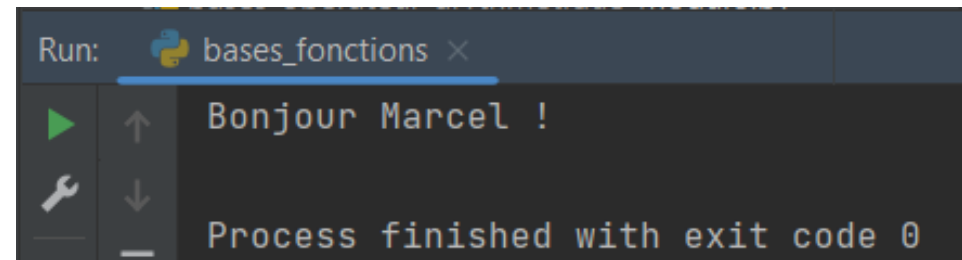
Version 3.7.4

<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryvsiew()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

Python- Les fonctions de l'utilisateur

- Pour définir une fonction, on utilise le mot clé **def**.
- Le mot clé **def** doit être suivi :
 - du nom de la fonction,
 - d'une paire de parenthèses (obligatoires)
 - Éventuellement de paramètres entre parenthèses (facultatifs)
 - et de ":"
- **Exemple :**

```
def dire_bonjour(prenom):  
    print("Bonjour "+prenom+" !")  
  
dire_bonjour("Marcel")
```



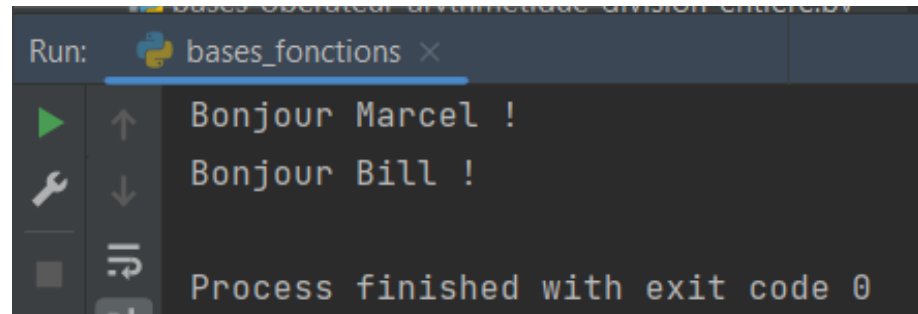
Python- Importance de la casse

- Attention, les **noms de fonctions** sont sensibles à la **casse**,
- Les fonctions `ma_fonction()` et `Ma_fonction()`, sont des fonctions différentes.

Python- Les paramètres

- L'intérêt des fonctions réside dans la possibilité de leur transmettre des données.
- Pour ce faire on peut définir des paramètres (entre parenthèses) dans la signature de la fonction

```
def dire_bonjour(prenom):  
    print("Bonjour "+prenom+" !")  
  
dire_bonjour("Marcel")  
dire_bonjour("Bill")
```



Run: bases_fonctions x

Bonjour Marcel !
Bonjour Bill !

Process finished with exit code 0

- Les parenthèses sont obligatoires.
- Une fonction n'est pas limitée à un unique paramètre. Elle peut en avoir plusieurs.

Python- Plusieurs paramètres

- Si une fonction a plusieurs paramètres, on les sépare des virgules.
- Exemple :

```
def dire_bonjour(prenom, age):  
    print(f"Bonjour {prenom}. Vous avez {age} ans")  
  
dire_bonjour("Marcel", 18)  
dire_bonjour("Bill", 75)
```

```
Run: bases_fonctions_param x  
▶ Bonjour Marcel. Vous avez 18 ans  
⚙ Bonjour Bill. Vous avez 75 ans  
■ Process finished with exit code 0
```

- Attention, dans ce cas **l'appel de la fonction avec 2 valeurs est obligatoire**

Python- La récursivité

- Une **fonction récursive s'appelle elle-même**.
- Pour éviter une boucle infinie il faut une **règle** permettant à la **récursivité de prendre fin**.
- Une telle condition est connue comme une condition de base.
- **Exemple:**

```
1  def factorial(n):
2      if(n == 0):
3          #Define our base case?
4              return 1
5      else:
6          return n*factorial(n-1)
7
8  print(factorial(5))
9
10 #result
11
12 # 120
```

- Dans cet exemple, la fonction **factorial** prend un entier n (positif) en entrée.
- Le factoriel d'un nombre est obtenu en multipliant le nombre par tous les entiers positifs en dessous. Par exemple, $\text{factorial}(3) = 3 \times 2 \times 1$, $\text{factorial}(2) = 2 \times 1$, et $\text{factorial}(0) = 1$.

Python- Cas d'étude - Fibonacci

- Dans une suite de Fibonacci, le **nombre de rang n** est la somme des deux rangs précédents : $f(n) = f(n-1) + f(n-2)$
- La suite de Fibonacci commence par $f(0)=0$ et $f(1)=1$.
- Ensuite $f(2) = f(1) + f(0) = 1 + 0 = 1$ $f(2)=1$
- $f(3) = f(2) + f(1) = 1 + 1 = 2$ $f(3)=2$
- $f(4) = f(3) + f(2) = 2 + 1 = 3$: $f(4)=3, f(5)=5, f(6)=8, f(7)=13$

```
1  def fibonacci(n):
2      if(n == 1):
3          #define Base case 1
4          return 0+1
5      elif(n == 2):
6          #define Base case 1
7          return 1+2
8      else:
9          return fibonacci(n) + fibonacci(n-1)
10
11  print(fibonacci(5))
12
13  #result
14
15  #
```



La portée des variables

Python- La portée des variables

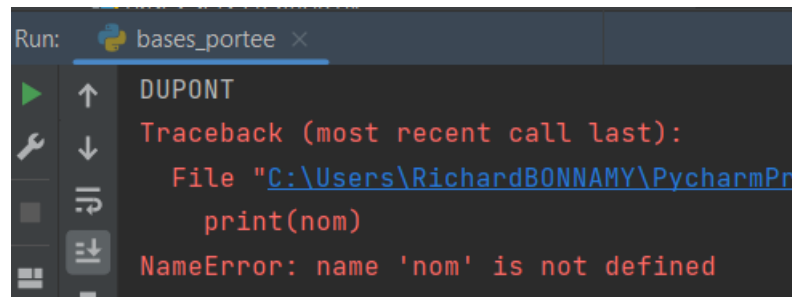
- Il est possible de **déclarer des variables n'importe où** dans notre script : au début du script, à l'intérieur de boucles, au sein de nos fonctions, etc.
- L'endroit où on définit une variable dans le script détermine l'endroit où la variable est utilisable.
- Le terme de **portée des variables** sert à désigner les différents espaces dans le script dans lesquels une variable est utilisable.
- Une variable peut avoir une portée locale ou une portée globale.

Python- Variables globales et locales

- Une variable, déclarée en dehors d'une fonction ou d'un bloc, est globale
- Une variable, déclarée dans une fonction ou un bloc, est locale.
- Les variables locales ne sont utilisables que qu'à l'intérieur de la fonction ou du bloc qui les a définies.
- Tenter d'appeler une variable locale depuis l'extérieur de la fonction qui l'a définie provoque une erreur.

```
def dire_bonjour():  
    nom = "DUPONT"  
    print(nom)
```

```
dire_bonjour()  
print(nom)
```



- Dans l'exemple ci-dessus, l'IDE PyCharm détecte que nom n'est pas définie et le souligne en rouge.

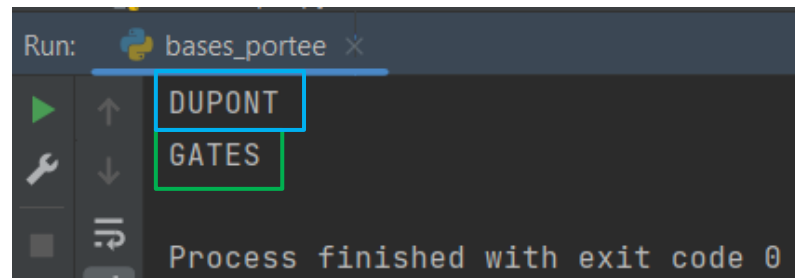
Python- Variables globales et locales

- L'autre erreur classique est de penser qu'on peut modifier une variable globale dans une fonction de la sorte :

```
nom = "GATES"

def dire_bonjour():
    nom = "DUPONT"
    print(nom)

dire_bonjour()
print(nom)
```



Run: bases_portee x

DUPONT

GATES

Process finished with exit code 0

- Dans l'exemple ci-dessus, il y a 2 variables :
 - une **globale** **nom**
 - une **locale** **nom** définie dans la fonction.
- L'affectation dans la fonction a pour effet de créer une variable locale nom
- Ces 2 variables ne sont pas les mêmes et ont des cycles de vie différents.

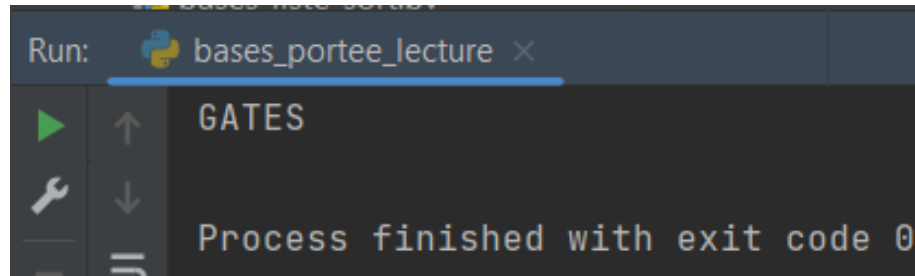
Python- Variables globales et locales

- En revanche une variable globale peut être accédée en lecture dans une fonction

```
nom = "GATES"

def dire_bonjour():
    print(nom)

dire_bonjour()
```



The screenshot shows a terminal window titled 'Run: bases_portee_lecture'. It displays the output 'GATES' and the message 'Process finished with exit code 0'.

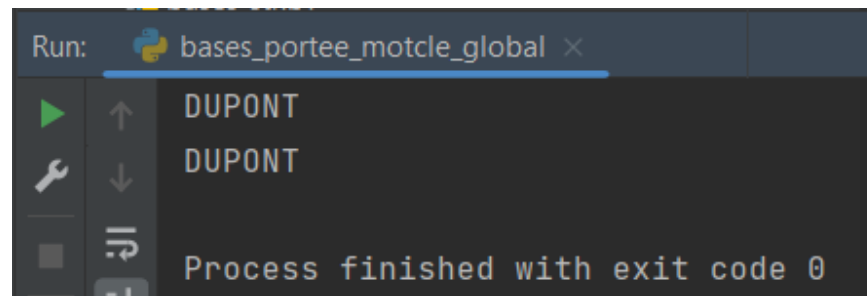
Python- Mot clé global pour l'accès en modification

- Pour modifier la valeur d'une variable globale dans une fonction, il faut utiliser le **mot clé global**.
- **Exemple :**

```
nom = "GATES"

def dire_bonjour():
    global nom
    nom = "DUPONT"
    print(nom)

dire_bonjour()
print(nom)
```



```
Run: bases_portee_motcle_global x
DUPONT
DUPONT
Process finished with exit code 0
```

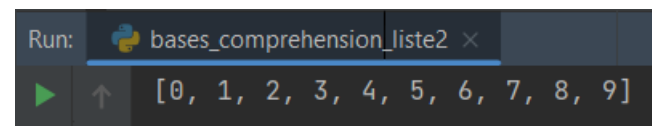


Les compréhensions

Python- Compréhension

- Une **compréhension** est une écriture compacte d'initialisation d'une liste.
- On utilise l'**opérateur crochet []** avec une **syntaxe spécifique** à l'intérieur
- Exemple d'écriture classique pour initialiser une liste avec tous les nombres de 1 à 10 :

```
newlist = []  
  
for x in range(0, 10):  
    newlist.append(x)  
  
print(newlist)
```



Run: bases_comprehension_liste2 ×
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

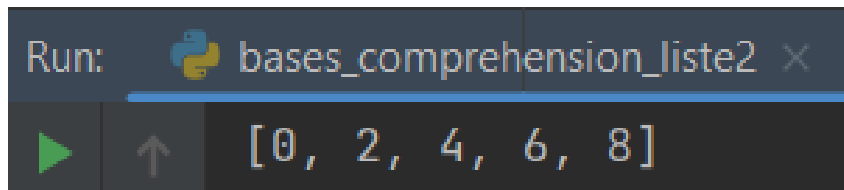
- Exemple équivalent avec une **compréhension** :


```
newlist = [x for x in range(0, 10)]  
  
print(newlist)
```



Python- Compréhension

- Il est possible d'**ajouter un if** pour filtrer les données
- Exemple pour construire une liste ne contenant que des entiers pairs :

```
newlist = [x for x in range(0, 10) if x % 2 == 0]  
  
print(newlist)
```



Run:  bases_comprehension_liste2 ×

  [0, 2, 4, 6, 8]

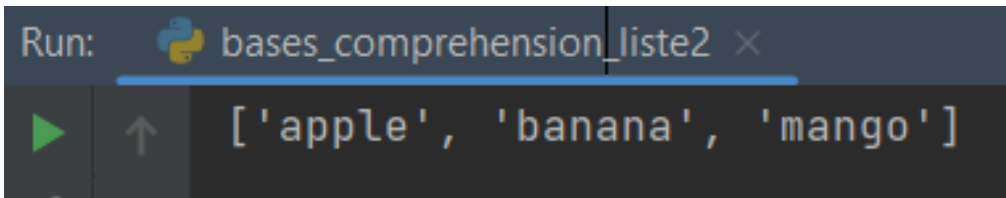
- Syntaxe :


```
newlist = [expression for item in iterable if condition == True]
```



Python- Autres exemples

- Exemple pour ne conserver que les fruits contenant la lettre "a" :

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]  
  
newlist = [x for x in fruits if "a" in x]  
  
print(newlist)
```



Run:  bases_comprehension_liste2 ×

  ['apple', 'banana', 'mango']

Python- Autres exemples

- On peut exécuter une fonction pour chaque item de la liste :

```
def ma_fonction(val):  
    return val*2  
  
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]  
  
newlist = [ma_fonction(x) for x in fruits if "a" in x]  
  
print(newlist)
```



Convention de nommage

Python- Conventions de nommage

PEP : Python Enhancement Proposals

PEP 8 : Style Guide for Python Code <https://peps.python.org/pep-0008/>

A retenir:

- Les noms de variables, de fonctions, de modules sont en **snake case**.
 - Exemple : `mon_poids = 75`
- Les noms de constantes sont en **screaming snake case**.
 - Exemple : `NB_SAISONS = 4`
- Les noms de classes et d'exceptions sont en **pascal case** ou **upper camel case**.
 - Exemple : `PersonnelNavigant`



Complément

Les fonctions
lambdas

Python- Les fonctions lambda

- Une fonction lambda est une fonction "**anonyme**" écrite de manière compacte
- On réserve l'écriture lambda aux fonctions qui n'ont qu'une instruction
- Exemple 1 : une fonction addition classique qui retourne le résultat d'une addition

```
def addition(x, y):  
    return x + y  
  
print(addition(2, 3))
```

- Passage à l'écriture lambda :

```
addition = lambda x, y: x + y  
  
print(addition(2, 3))
```

Python- Les fonctions lambda

- **Etape 1 :** On fait passer le nom de la fonction "addition" sur la gauche et on place un opérateur d'affectation

```
def addition(x, y):  
    return x + y
```

```
print(addition(2, 3))
```



```
addition = def (x, y):  
    return x + y
```



- Evidemment à ce stade le code n'est pas fonctionnel
- **Etape 2 :** on remplace **def** par **lambda**

```
addition = lambda (x, y):  
    return x + y
```

Python- Les fonctions lambda

- **Etape 3 :** on supprime les parenthèses et on passe tout sur une ligne

```
addition = lambda x, y: return x + y  
print(addition(2, 3))
```

- **Etape 4 (facultative):** si l'instruction unique est un return il faut supprimer le mot clé return

```
addition = lambda x, y: x + y  
print(addition(2, 3))
```

- On parle de fonction anonyme mais en réalité il faut la stocker dans une variable pour pouvoir la manipuler. Ici la variable est addition.

Python- Les fonctions lambda

➤ Exemple 2 : fonction sans return

```
def print_addition(x, y):  
    print("Résultat de l'addition = "+str(x + y))  
  
print_addition(2, 3)
```

➤ Transformation en lambda

```
print_addition = lambda x, y: print("Résultat de l'addition = "+str(x + y))  
  
print_addition(2, 3)
```

FIN

MERCI DE VOTRE ATTENTION !