# Programmation en Python
# (II)

Akka Zemmari
Hervé Hocquard
herve.hocquard@u-bordeaux.fr

LaBRI, Université de Bordeaux - CNRS

17 septembre, 2023

université de BORDEAUX

# Plan

Ce que nous verrons dans ce cours :

- Opérateurs de base
- Types de variables
- Nombres
- Chaines de caractères
- Listes
- Tuples
- Dictionnaire
- Branchements conditionnels
- Boucles
- Fonctions
- Modules
- Fichiers I/O
- Exceptions
- Classes et Objets

université de BORDEAUX

# Plan

Ce que nous verrons dans ce cours :

- Opérateurs de base
- Types de variables
- Nombres
- Chaines de caractères
- Listes
- Tuples
- Dictionnaire
- Branchements conditionnels
- Boucles
- Fonctions
- Modules
- Fichiers I/O
- Exceptions
- Classes et Objets

# Caractéristiques de Python

# Caractéristiques de Python

- Facile à apprendre : peu de mots clés, structure simple, syntaxe assez claire

# Caractéristiques de Python

- Facile à apprendre : peu de mots clés, structure simple, syntaxe assez claire
- Facile à lire : assez proche de "texte naturel"

# Caractéristiques de Python

- Facile à apprendre : peu de mots clés, structure simple, syntaxe assez claire
- Facile à lire : assez proche de "texte naturel"
- Facile à maintenir

# Caractéristiques de Python

- Facile à apprendre : peu de mots clés, structure simple, syntaxe assez claire
- Facile à lire : assez proche de "texte naturel"
- Facile à maintenir
- Mode interactif possible

# Caractéristiques de Python

- Facile à apprendre : peu de mots clés, structure simple, syntaxe assez claire
- Facile à lire : assez proche de "texte naturel"
- Facile à maintenir
- Mode interactif possible
- Types de données dynamiques de très haut niveau

# Plus d'information

Une seule adresse :

`https://www.python.org/`

# Opérateurs

# Opérateurs

- Opérateurs arithmétiques

# Opérateurs

- Opérateurs arithmétiques
- Opérateurs de comparaison

# Opérateurs

- Opérateurs arithmétiques
- Opérateurs de comparaison
- Opérateurs d'affectation

# Opérateurs

- Opérateurs arithmétiques
- Opérateurs de comparaison
- Opérateurs d'affectation
- Opérateurs logiques

# Opérateurs

- Opérateurs arithmétiques
- Opérateurs de comparaison
- Opérateurs d'affectation
- Opérateurs logiques
- Opérateurs ensemblistes

# Opérateurs arithmétiques (sur des nombres)

On suppose que $a$ vaut 10 et que $b$ vaut 20 :

# Opérateurs arithmétiques (sur des nombres)

On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| Opérateur | Description | Exemple |
|---|---|---|
| $+$ | Addition | $a + b$ vaut 30 |

# Opérateurs arithmétiques (sur des nombres)

On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| Opérateur | Description | Exemple |
|---|---|---|
| $+$ | Addition | $a + b$ vaut 30 |
| $-$ | Soustraction | $a - b$ vaut $-20$ |

# Opérateurs arithmétiques (sur des nombres)

On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| Opérateur | Description | Exemple |
|---|---|---|
| $+$ | Addition | $a + b$ vaut 30 |
| $-$ | Soustraction | $a - b$ vaut $-20$ |
| $*$ | Multiplication | $a * b$ vaut 200 |

# Opérateurs arithmétiques (sur des nombres)

On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| Opérateur | Description | Exemple |
|---|---|---|
| $+$ | Addition | $a + b$ vaut 30 |
| $-$ | Soustraction | $a - b$ vaut $-20$ |
| $*$ | Multiplication | $a * b$ vaut 200 |
| $/$ | Division | $b/a$ vaut 2 |

# Opérateurs arithmétiques (sur des nombres)

On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| Opérateur | Description | Exemple |
|---|---|---|
| $+$ | Addition | $a + b$ vaut 30 |
| $-$ | Soustraction | $a - b$ vaut $-20$ |
| $*$ | Multiplication | $a * b$ vaut 200 |
| $/$ | Division | $b/a$ vaut 2 |
| % | Modulo | $b\%a$ vaut 0 et $a\%b$ vaut 10 |

# Opérateurs arithmétiques (sur des nombres)

On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| Opérateur | Description | Exemple |
|---|---|---|
| $+$ | Addition | $a + b$ vaut 30 |
| $-$ | Soustraction | $a - b$ vaut $-20$ |
| $*$ | Multiplication | $a * b$ vaut 200 |
| $/$ | Division | $b/a$ vaut 2 |
| % | Modulo | $b\%a$ vaut 0 et $a\%b$ vaut 10 |
| $**$ | Puissance | $b * *a$ vaut $20^{10}$ |

# Opérateurs arithmétiques (sur des nombres)

On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| Opérateur | Description | Exemple |
|---|---|---|
| $+$ | Addition | $a + b$ vaut 30 |
| $-$ | Soustraction | $a - b$ vaut $-20$ |
| $*$ | Multiplication | $a * b$ vaut 200 |
| $/$ | Division | $b/a$ vaut 2 |
| % | Modulo | $b\%a$ vaut 0 et $a\%b$ vaut 10 |
| $**$ | Puissance | $b * *a$ vaut $20^{10}$ |
| $//$ | Division entière | $b//a$ vaut 2 |

# Opérateurs de comparaison (résultat booléen)

Une expression qui ne peut prendre que les valeurs **True** ou **False** est appelée expression booléenne.

# Opérateurs de comparaison (résultat booléen)

Une expression qui ne peut prendre que les valeurs **True** ou **False** est appelée expression booléenne.
On suppose que *a* vaut 10 et que *b* vaut 20 :

# Opérateurs de comparaison (résultat booléen)

Une expression qui ne peut prendre que les valeurs **True** ou **False** est appelée expression booléenne.
On suppose que *a* vaut 10 et que *b* vaut 20 :

| **Opérateur** | **Description** | **Exemple** |
|---|---|---|
| == | Égalité | *a* == *b* vaut *False* |

# Opérateurs de comparaison (résultat booléen)

Une expression qui ne peut prendre que les valeurs **True** ou **False** est appelée expression booléenne.
On suppose que *a* vaut 10 et que *b* vaut 20 :

| **Opérateur** | **Description** | **Exemple** |
|---|---|---|
| $==$ | Égalité | $a == b$ vaut *False* |
| $!=$ | Différent | $a != b$ vaut *True* |

# Opérateurs de comparaison (résultat booléen)

Une expression qui ne peut prendre que les valeurs **True** ou **False** est appelée expression booléenne.
On suppose que *a* vaut 10 et que *b* vaut 20 :

| **Opérateur** | **Description** | **Exemple** |
|---|---|---|
| $==$ | Égalité | $a == b$ vaut *False* |
| $!=$ | Différent | $a != b$ vaut *True* |
| $<$ | Inférieur strictement | $a < b$ vaut *True* |

# Opérateurs de comparaison (résultat booléen)

Une expression qui ne peut prendre que les valeurs **True** ou **False** est appelée expression booléenne.
On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| **Opérateur** | **Description** | **Exemple** |
|---|---|---|
| $==$ | Égalité | $a == b$ vaut *False* |
| $!=$ | Différent | $a != b$ vaut *True* |
| $<$ | Inférieur strictement | $a < b$ vaut *True* |
| $>$ | Supérieur strictement | $a > b$ vaut *False* |

# Opérateurs de comparaison (résultat booléen)

Une expression qui ne peut prendre que les valeurs **True** ou **False** est appelée expression booléenne.
On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| **Opérateur** | **Description** | **Exemple** |
|---|---|---|
| $==$ | Égalité | $a == b$ vaut *False* |
| $!=$ | Différent | $a != b$ vaut *True* |
| $<$ | Inférieur strictement | $a < b$ vaut *True* |
| $>$ | Supérieur strictement | $a > b$ vaut *False* |
| $\leq$ | Inférieur ou égal | $a \leq b$ vaut *True* |

# Opérateurs de comparaison (résultat booléen)

Une expression qui ne peut prendre que les valeurs **True** ou **False** est appelée expression booléenne.
On suppose que $a$ vaut 10 et que $b$ vaut 20 :

| **Opérateur** | **Description** | **Exemple** |
|---|---|---|
| $==$ | Égalité | $a == b$ vaut *False* |
| $!=$ | Différent | $a != b$ vaut *True* |
| $<$ | Inférieur strictement | $a < b$ vaut *True* |
| $>$ | Supérieur strictement | $a > b$ vaut *False* |
| $\leq$ | Inférieur ou égal | $a \leq b$ vaut *True* |
| $\geq$ | Supérieur ou égal | $a \geq b$ vaut *False* |

# Opérateurs logiques (entre booléens, résultat booléen)

On suppose que *a* vaut *True* et que *b* vaut *False* :

# Opérateurs logiques (entre booléens, résultat booléen)

On suppose que *a* vaut *True* et que *b* vaut *False* :

| Opérateur | Description | Exemple |
|---|---|---|
| *or* | ou (logique) | *a or b* vaut *True* |

# Opérateurs logiques (entre booléens, résultat booléen)

On suppose que *a* vaut *True* et que *b* vaut *False* :

| Opérateur | Description | Exemple |
|---|---|---|
| *or* | ou (logique) | *a or b* vaut *True* |
| *and* | et (logique) | *a and b* vaut *False* |

# Opérateurs logiques (entre booléens, résultat booléen)

On suppose que *a* vaut *True* et que *b* vaut *False* :

| Opérateur | Description | Exemple |
|---|---|---|
| *or* | ou (logique) | *a or b* vaut *True* |
| *and* | et (logique) | *a and b* vaut *False* |
| *not* | non (logique) | *not b* vaut *True* |

- En Python, le « ou » est fainéant, c'est-à-dire que si la 1ère expression vaut vrai, la deuxième n'est pas évaluée ( 2 == 1 + 1 ) or (a > = 5) ne provoque pas d'erreur même si a n'existe pas, le résultat vaut vrai ( 3 == 1 + 1 ) or (a > = 5) provoque une erreur si a n'existe pas.
- En Python, le « et » est fainéant, c'est-à-dire que si la 1ère expression vaut faux, la deuxième n'est pas évaluée ( 2 > 8 ) and (a > = 5) ne provoque pas d'erreur même si a n'existe pas, le résultat vaut faux ( 2 < 8 ) and (a > = 5) provoque une erreur si a n'existe pas.

# Notion de variable

- on doit être capable de stocker des informations en mémoire centrale durant l'exécution d'un programme
- on veut éviter d'avoir à manipuler directement les adresses

# Notion de variable

- on doit être capable de stocker des informations en mémoire centrale durant l'exécution d'un programme
- on veut éviter d'avoir à manipuler directement les adresses
- $\rightarrow$ on manipule des variables
- le programmeur donne aux variables des noms de son choix
- les variables désignent une ou plusieurs cases mémoires.

# Affectation des valeurs à des variables =

Syntaxe :

```
1  nom_variable = val
```

L'opérande (nom_variable) à gauche du symbole = est le nom de la variable alors que l'opérande (val) à sa droite est la valeur à stocker dans la variable :

```
1  counter = 100 # Affectation d'un entier
2  miles = 1000.0 # Affectation d'un réel
3  name = "John" # Une chaîne de caractères
4  print(counter,miles,name,sep="\n")
```

À l'exécution, on obtient :

```
100
1000.0
John
```

# Affectation des valeurs à des variables =

Que se passe-t-il lorsqu'on stocke une valeur dans une variable ?

```
1  n=10
2  print(id(n))
3  m=n
4  print(id(m))
5  m=10.0
6  print(id(m))
```

À l'exécution, on obtient :

```
1352264336
1352264336
66820992
```

université de BORDEAUX

# Types de données standards

Python définit cinq types de données standards :

1. Numbers
2. String
3. List
4. Tuple
5. Dictionary

# Numbers

- Python définit plusieurs types de "Numbers" : `int, float, complex`
- L'instruction `x=3` crée une variable de type `int` initialisée à 3
  tandis que `y=3.0` crée une variable de type `float` initialisée à 3.0

```
1  x = 3
2  y = 3.0
3  print("x =", x,type(x))
4  print("y =", y,type(y))
```

À l'exécution, on obtient :

```
x = 3 <class 'int'>
y = 3.0 <class 'float'>
```

# String

- Chaîne de caractères = String en anglais = suite finie de caractères = texte.

# String

- Chaîne de caractères = String en anglais = suite finie de caractères = texte.
- Ce texte est compris entre deux guillemets ou deux apostrophes, ces deux symboles sont interchangeables.
- Le signe + correspond à la concaténation, le signe $*$ à la répétition
- Créer une chaîne de caractères : affecter une valeur à une variable :

# String

- Chaîne de caractères = String en anglais = suite finie de caractères = texte.
- Ce texte est compris entre deux guillemets ou deux apostrophes, ces deux symboles sont interchangeables.
- Le signe + correspond à la concaténation, le signe ∗ à la répétition
- Créer une chaîne de caractères : affecter une valeur à une variable :

```
1    ch1 = 'Hello world!'
2    ch2 = "Bonjour"
```

# String

```
1  str = 'Hello World!'
2  print(str)           # Affiche la chaîne de caractère en entier
3  print(str[0])        # Affiche le premier caractère de la chaîne
4  print(str[2:5])      # Affiche les caractères entre les positions 3 et 5
5  print(str[2:])       # Affiche les caractères à partir de la position 3
6  print(str * 2)       # Affiche la chaîne deux fois
7  print(str + "Test")  # Affiche la concaténation des deux chaînes
```

À l'exécution, on obtient :

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!Test
```

# Caractères d'échappement

Liste de caractères non imprimables :

| | | |
|---|---|---|
| $\backslash b$ | 0x08 | Backspace |
| $\backslash e$ | 0x1b | Escape |
| $\backslash n$ | 0x0a | Newline |
| $\backslash r$ | 0x0d | Carriage return |
| $\backslash s$ | 0x20 | Space |
| $\backslash t$ | 0x09 | Tab |
| $\backslash v$ | 0x0b | Vertical tab |
| $\backslash x$ | | Character x |
| $\backslash xnn$ | | Notation hexadecimale, $n$ entre 0 et 9, $a$ et $f$ ou $A$ e $F$ |

# String : opérateurs de formatage

```
1  print("My name is %s and I am %d years old." % ('Totolabricot', 21))
```

À l'exécution, on obtient :

```
My name is Totolabricot and I am 21 years old.
```

| Format Symbol | Conversion |
|---|---|
| %s | string conversion via `str()` prior to formatting |
| %i | signed decimal integer |
| %d | signed decimal integer |
| %f | floating point real number |

# String

Longueur d'une chaîne de caractères :

```
1 fruit = 'pomme'
2 print(len(fruit))
```

À l'exécution, on obtient :

5

# String Library

D'autres fonctions existent :

```
1 greet = 'Hello Bob'
2 zap = greet.lower()
3 print(zap)
4 print(greet)
5 print('Hi There'.lower())
```

À l'exécution, on obtient :

```
hello bob
Hello Bob
hi there
```

# String Library

D'autres fonctions existent :

```
1  str.capitalize()                 str.title(width[])
2  str.join()                       str.find(sub[, start[, end]])
3  str.replace(old, new[, count])   str.lower()
4  str.rstrip([chars])              str.upper()
```

# String : Comparaison

```
1  if word == 'banana':
2      print('All right, bananas.')
3  if word < 'banana':
4      print('Your word,' + word + ', comes before banana.')
5  elif word > 'banana':
6      print('Your word,' + word + ', comes after banana.')
7  else:
8      print('All right, bananas.')
```

# String

**Chercher des sous-chaînes :**

- On utilise la fonction `find()` pour chercher une sous-chaîne dans une chaîne
- `find()` trouve la première occurrence de la sous-chaine
- Si la sous-chaîne n'est pas trouvée, `find()` retourne `-1`
- Rappel : dans une chaîne, le premier caractère se trouve à la position zero

| p | o | m | m | e |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

```
1  fruit = 'pomme'
2  pos = fruit.find('om')
3  print(pos)
4  aa = fruit.find('z')
5  print(aa)
```

À l'exécution, on obtient :

```
1
-1
```

# String

**Rechercher et remplacer :**

- La fonction `replace()` agit comme l'opération "chercher et remplacer" dans word
- Elle remplace toutes les occurrences de la chaîne recherchée par la chaîne de remplacement

```
1  greet = 'Hello Bob'
2  nstr = greet.replace('Bob','Jane')
3  print(nstr)
4  nstr = greet.replace('o','X')
5  print(nstr)
```

À l'exécution, on obtient :

```
Hello Jane
HellX BXb
```

# Conversion de types

Parfois, il est nécessaire de convertir un type vers un autre :

| | |
|---|---|
| `int(x)` | Converts x to an integer. |
| `long(x)` | Converts x to a long integer |
| `float(x)` | Converts x to a floating-point number. |
| `complex(real, imag)` | Creates a complex number. |
| `str(x)` | Converts object x to a string representation. |
| `tuple(s)` | Converts s to a tuple. |
| `list(s)` | Converts s to a list. |
| `set(s)` | Converts s to a set. |
| `dict(d)` | Creates a dictionary. |
| `hex(x)` | Converts an integer to a hexadecimal string. |

# Les entrées/sorties

On a généralement besoin de pouvoir interagir avec un programme :

- pour lui fournir les données à traiter, par exemple au clavier : **entrées**.
- pour pouvoir connaître le résultat d'exécution ou pour que le programme puisse écrire ce qu'il attend de l'utilisateur, par exemple, texte écrit à l'écran : **sorties**.

# Les entrées : la fonction input()

- À l'exécution, l'ordinateur :
  - interrompt l'exécution du programme,
  - affiche éventuellement un message à l'écran,
  - attend que l'utilisateur entre une donnée au clavier et appuie sur la touche Entrée.
- C'est une saisie en mode texte :
  - la valeur saisie est vue comme une chaîne de caractères,
  - on peut ensuite changer le type.

# Les sorties : la fonction print()

À l'exécution, l'ordinateur :

- affiche la représentation textuelle de n'importe quel nombre de valeurs fournies entre les parenthèses et séparées par des virgules.
- à l'affichage, ces valeurs sont séparées par un espace.
- l'ensemble se termine par un retour à la ligne (modifiable en utilisant sep et/ou end).

Possibilité d'insérer :

- des sauts de ligne en utilisant $\backslash n$ et
- des tabulations avec $\backslash t$.