

Quelques rappels

Polytech Marseille

Séverine Dubuisson, Simon Vilmin

`severine.dubuisson@univ-amu.fr`,

`simon.vilmin@univ-amu.fr`

2024 - 2025

amU Aix
Marseille
Université



Au programme

Premiers pas

- Commentaires

- Instructions

- Types et variables

- Variables

- Saisie de valeurs avec `input()`

- Affichage avec `print()`

- Résumé

Un peu plus technique

- Chaînes formatées

- Mémoire et Python

- Résumé

Premiers pas

Premiers pas

- Commentaires

- Instructions

- Types et variables

- Variables

- Saisie de valeurs avec `input()`

- Affichage avec `print()`

- Résumé

Un peu plus technique


Commentaires


 **Syntaxe** : deux manières de faire des commentaires

- commentaires *courts* : ligne commençant par #
- commentaires *longs* : bloc de lignes encadré par `"""`

```
# un commentaire court
```

```
""" Un commentaire long, treeeeeeeeeeeeeeeeeeeeeeeeeeee  
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee  
long. """
```

 **Question** : pourquoi faire ?

 **Réponse** : pour rendre le code plus lisible, y compris pour vous.

Validité des instructions

Une instruction (ou un bloc d'instructions) doit être correcte :


- *syntactiquement* : elle doit respecter la *syntaxe* du langage
 - nombre de paramètres des opérateurs,
 - syntaxe des mots clés,
 - parenthèses au niveau des fonctions, etc


```
>>> 3 + 4 + # mauvaise syntaxe : '+' attend 2 parametres
SyntaxError: invalid syntax
```

- *sémantiquement* : elle doit pouvoir être *évaluée*

```
>>> "b" / 12 # syntaxe ok mais 'b' / 12 n'a pas de sens
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```


Types de base

 **Question** : Qu'est-ce qui différencie 12 et "b" ?

 **Réponse** : le *type* de ces valeurs. C'est leur *nature*, il définit les opérations qu'on peut leur appliquer.

5 principaux types de base en Python :

- **int** : nombres entiers, (ex : 4, -8, 36, 725648791)
- **float** : nombres réels, (ex : -3.14, 16.758, 2e56)
- **str** : chaînes de caractère, (ex : "pouet", 'a_B27 458+-')
- **bool** : booléens (**True**, **False**)
- **NoneType** : le type de la constante **None** représentant la valeur « nulle »

 **Remarque** : en Python, le typage est *dynamique*. Les types sont déterminés automatiquement et non pas spécifié par l'utilisateur·ice.

Type et conversion (cast)

On peut obtenir le type d'une valeur avec la fonction `type()` :

```
>>> type("La question elle est vite repondue")  
<class 'str'>
```

On peut, dans certains cas, *convertir* le type d'une valeur : c'est un *cast*

```
>>> int(4.57)  
4
```

```
>>> float('4.57')  
4.57
```

```
>>> int("A")  
ValueError: invalid literal for int() with base 10: 'A'
```

Quelques opérateurs

Opérateurs arithmétiques (sur des nombres) :

- `+`, `-`, `*` : addition, soustraction, produit
- `**` : puissance
- `/`, `%`, `//` : division, modulo, division entière

Opérateurs de comparaison (résultat booléen !) :

- `==`, `!=` : est égal à, est différent de
- `<`, `<=`, `>=`, `>` : plus petit, plus petit ou égal, plus grand ou égal, plus grand

Opérateurs logiques (entre booléens, résultat booléen ou `NoneType`) :

- `and`, `or`, `not`

Quelques opérateurs sur les chaînes

Certaines opérations sont également définies sur les chaînes de caractères :

- `+` : *concaténation* de deux chaînes de caractères
- `*` : *répétition* d'une chaîne de caractère
- `len()` : renvoie la *longueur* d'une chaîne

```
>>> "bon" + "jour"
```

```
bonjour
```

```
>>> 4 * "ah "
```

```
ah ah ah ah
```

```
>>> len("crocodile")
```

```
9
```

Exercice

⚙️ **Exercice** : donner le résultat des instructions suivantes :

```
(1 + 2)**3
```

```
"bla" + 4
```

```
"bla" * 2
```

```
"bla" * 2 - "bla"
```

```
"1" + "1"
```

```
5 // 2
```

```
5 % 2
```

```
str(4) * int("3")
```

```
float(int(3.14))
```

```
float(int("3.14"))
```

```
str(3) * float("3.2")
```

```
str(3/4) * 2
```

```
int(float("3.14"))
```

```
type(int(float(int(3.14))) * str(float(2)) == 6)
```

Variables : identificateurs


 **Définition** : une variable est un *conteneur d'information*. Le nom d'une variable est son *identificateur*.

 **Syntaxe** : identificateurs : `[_]lettre[lettre/chiffre/_]`

- sensibles à la casse (pouet != POUET)
- doivent être différents des mots-clés du langage (`if`, `else`, `return`, ...)

Exemples :

```
corrects : pouet, _a, var256, le_chameau, WaOuH  
incorrects : 2B3, 2-B-3, nom var, pie*thon, (a
```

 **Remarque** : voir [PEP8](#) pour les bonnes pratiques de nommage, e.g. :

- nom porteur de sens (`var_age > lhkvfdbg`)
- pas d'accents

Variables : affectation

 **Syntaxe** : pour mémoriser une valeur dans une variable, *affectation* avec = :

```
var = val # la variable nommée var mémorise la valeur val  
var = expression # var stocke le résultat de l'expression
```

Exemples :

```
>>> prenom = "Mirabelle"  
>>> prenom  
'Mirabelle'  
  
>>> a = 4 + 5 * 2  
>>> b = 3 * a  
>>> b  
42
```

 **Remarque** : la déclaration d'une variable se fait en même temps que sa première affectation. C'est l'*initialisation* de la variable.

Plus sur les variables

! Attention : ne pas confondre = (affectation) et == (comparaison) !

```
>>> a = 4 # affectation
>>> a == 2 # comparaison
False
```

i Remarque : on peut appliquer aux variables ce qu'on a appliqué aux valeurs : type, cast, opérations, etc

```
>>> type(a)
<class 'int'>
>>> b = str(a + 7)
>>> b
'11'
```

Un détail sur les chaînes de caractères

On peut accéder aux éléments d'une chaîne de caractères par leurs *indices*.

```
>>> chaine = "bonjour"
>>> chaine[0]
"b"
```

! Attention : on ne peut pas modifier les caractères d'une chaîne : **str** est un *type immutable*. Pour modifier une variable **str**, il faut la réaffecter.

```
>>> animal = "morse"
>>> animal[0] = "M"
TypeError: 'str' object does not support item assignment

>>> animal = "Morse"
```

Exercice

⚙️ **Exercice :** Que vaut bcdj à la fin du programme suivant ?

```
bcdj = 2
bjcd = int(str(bcdj) * 2) - 16
ouf = bjcd
bdjc = ouf
bjcd = ouf + bcdj
bcdj = bdbc + bjcd
ouf = (bcdj + bjcd) // (ouf + bdbc)
bjcd = float(str(bcdj) * int(bjcd) + str(ouf) * int(bdbc)) - 13.75
bcdj = (int((str(ouf) * 2) + str(bcdj // 2)) + 1) / (2 + bdbc - 6)
```

⚙️ **Exercice :** Que calcule le programme suivant (que représente densite) ?


```
masse = 2.7182
vol = 3.1459
densite = ((vol * masse**3) + (masse * vol**3)) / (masse * vol)
densite = densite / densite**(0.5)
```

Saisie de valeurs avec `input()`

 **Syntaxe :** `input`(message a afficher)

```
>>> variable = input("entrez une valeur : ")
entrez une valeur : # mettons 196
>>> variable
'196' # pas un entier !!!

>>> type(variable)
<class 'str'>
```

 **Attention :** `input()` renvoie une chaîne de caractères !!!

```
>>> variable = int(input("entrez une valeur : ")) # cast
>>> variable
196
```


Affichage avec `print()`

 **Syntaxe :** `print(*objets, sep='', end='\n')`

- `*objets` : le(s) élément(s) à afficher
- `sep=''` : séparateur entre les éléments à afficher (`' '` par défaut)
- `end='\n'` : fin de l'affichage (`\n` par défaut, soit le retour à la ligne)

Remarque :

- la syntaxe au dessus est une version simplifiée de la doc Python
- la notation `param=valeur` signifie que le paramètre `param` est *optionnel* et qu'il a la valeur `valeur` par défaut

Exemple

```
# utilisation basique
```

```
>>> print("bonjour")
```

```
bonjour
```

```
# utilisation avec plusieurs elements a afficher
```

```
>>> age = 29
```

```
>>> taille = 1.78
```

```
>>> print("J'ai", age, "ans et je mesure", taille, "m")
```

```
J'ai 29 ans et je mesure 1.78 m
```

```
# utilisation ou on affiche plusieurs elements et ou on change
```

```
# les valeurs par default de sep et end
```

```
>>> print("age", age, "taille", taille, sep=" ", end=".\\n AH")
```


```
age, 28, taille, 1.78.
```

```
AH
```

Résumé

Rappel :

- les commentaires sont des lignes commençant par #
- une variable est un *conteneur d'information* avec un nom
- les variables et les valeurs ont un *type* qui définit leur nature et les opérations qu'on peut effectuer avec

 **Attention :** = affectation, == comparaison

```
>>> nb_1, nb_2 = 63.5, 4
>>> type(nb_1 + nb_2)
<class 'float'>

>>> nb_1 == 67
False
```

Résumé

Rappel :

- `input()` pour faire saisir des valeurs par l'utilisateur·ice
- `print()` pour afficher des valeurs, des messages, etc

! Attention : la saisie via `input()` ne produit que des chaînes de caractères, penser à convertir en nombre (ou autre) si besoin !

```
>>> temp = float(input("temperature de l'eau : "))
temperature de l'eau : 28.5


>>> print("l'eau est a", 100 - temp,
          "degres d'etre en ebullition", sep=" !! ", end=".")
l'eau est a !! 71.5 !! degres d'etre en ebullition.
```

Un peu plus technique

Premiers pas

Un peu plus technique
Chaînes formatées
Mémoire et Python
Résumé


Chaînes de caractères formatées

 **Remarque :** on peut afficher des messages complexes avec des *chaînes de caractères formatées*, les *f-strings*.

Syntaxe :

```
f" <texte> {<expr> <opt:form>} <texte> {<expr> <opt:form>} ..."
```

- f avant les guillemets
- <texte> : texte qui fera parti de la chaîne
- <expr> : expression qui sera évaluée puis insérée dans la chaîne
- <opt:form> : un format (optionnel) à appliquer au résultat de l'expression
- accolades autour des expressions et leur format

 **Astuce :** plus d'informations dans la [doc](#) et dans une fiche sur [AMeTICE](#) !

Exemple

```
>>> prenom = "Madeleine"
>>> age = 22
>>> taille = 1.64786513

# premier test simple
>>> print(f"Bonjour {prenom} !")
Bonjour Madeleine !


# test avec des expressions plus complexes
>>> print(f"Tu as {(age + 4) / (age - 2)} ans et tu fais {taille}m, c'est ca ?")
Tu as 1.3 ans et tu mesures 1.64756513m, c'est ca ?

# test avec un format sur la taille,
# .2f n'affiche que deux chiffres significatifs
>>> print(f"Euh non, j'ai {age + 0.5} ans et en gros je fais {taille:5.2f}m")
Euh non, j'ai 22.5 ans et en gros je fais 1.65m
```

Exercice

⚙️ **Exercice** : écrire un programme qui

- lit un temps en secondes et le décompose en heure, minute, seconde.
- affiche ce temps sous le format hh:mm:ss.

 **Astuce** : le format 02d permet d'afficher des entiers (d) sur au moins 4 caractères (4) qui ont par défaut la valeur 0 (0) :

```
>>> n = 5
>>> m = 23
>>> print(f"{n:04d}, {m:04d}")
0005, 0023
```


Une vue de la mémoire

1

bit : plus petit élément de la mémoire

2 valeurs possibles (0 ou 1)

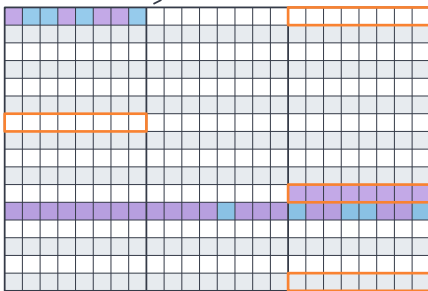
0 1 1 0 1 0 0 1

Octet : mot de 8 bits

$2^8 = 256$ valeurs possibles

adr 2

adr 18



adr 47

- Mémoire : immense suite d'octets
- Chaque octet a une **adresse**
- Les programmes viennent stocker des valeurs dans la mémoire

valeur 2201 stockée à partir de l'**adr 32**
au format binaire

Une histoire de représentation

❌ **Problème** : donc en mémoire, tout est représenté en binaire ! ALED !

Représentations classiques :

- booléens : 0 (**False**) ou 1 (**True**) (codé sur un octet)
- chaînes de caractères : normes (UTF-8, Latin-1, code ASCII, ...)
 - ASCII : un octet par caractère, ne code pas les accents

'a' -> 97 'b' -> 98 ...

'A' -> 65 'B' -> 66 ...

'0' -> 48 '1' -> 49 ...

- entiers relatifs : *complément à 2*
- nombres réels : norme *IEEE 754*

Le rapport avec Python ?

i Remarque : en Python, les types de base sont une « surcouche » à ces représentations

? Question : mais en quoi ça nous touche tout ça ?

```
>>> 0.125 + 0.125 == 0.25
```

```
True
```

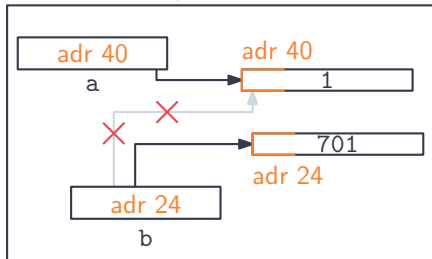
```
>>> 0.1 + 0.1 + 0.1 == 0.3
```

```
False
```

! Attention : à cause de la précision finie de la représentation, certaines égalités « naturelles » pour nous ne sont pas vérifiées par la machine !

Qu'est-ce qu'une variable ?

schéma simplifié de la mémoire



```
>>> a = 1
>>> b = a
>>> b = b + 700
```

i Remarque :

- en Python, une variable ne contient pas directement une valeur, mais l'*adresse mémoire* à laquelle est stockée la valeur
- `b = a` signifie que `b` « pointe » sur la même zone mémoire que `a`
- *pour les types de bases* modifier `b` crée un nouvel emplacement mémoire

Résumé



Rappel :

- quand on fait `a = b`, `a` « pointe » sur la même zone mémoire que `b`
- on peut faire des `str` élaborées via les *f-strings*



Attention : à cause de leurs représentations machine, il est risqué de comparer des nombres réels (`float`)

```
>>> 0.1 + 0.1 + 0.1 == 0.3
```

```
False
```

```
>>> print(f"J'ai {108 + 5:08d} elephants")
```

```
J'ai 00000113 elephants
```