

Premiers pas en Python

Polytech Marseille, GII, 3A

Séverine Dubuisson, Simon Vilmin

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

2023 - 2024



Au programme

Instructions, types de bases et variables

- Premières instructions et types de base

- Variables

- Résumé

Entrées/sorties et chaînes de caractères

- Saisie de valeurs avec `input()`

- Affichage avec `print()` et `format()`

- Résumé

Mémoire et représentation des données

- Mémoire

- Représentation des données

- Gestion de la mémoire en Python

Instructions, Types de Bases et Variables

Instructions, types de bases et variables

Premières instructions et types de base

Variables

Résumé

Entrées/sorties et chaînes de caractères

Mémoire et représentation des données

HALTE!!! Commentaires

Toute première notion : les *commentaires*

- lignes commençant par # ou bloc de lignes encadré de `"""` (aussi appelé *docstring*)
- lignes pas exécutées, purement explicatives

```
# un commentaire en une ligne
```

```
# un autre
```

```
""" Un long commentaire ou je raconte ma vie et ou tout  
le monde meurt d'ennui trois fois parce que c'est pas  
non plus super fascinant mais au moins j'ai bien documenté  
mon code """
```

? Question : pourquoi faire? Pour rendre le code lisible, pour tout le monde (vous y compris).

Premiers tests avec l'interpréteur

```
In []: (3 + 45.0) * (75.43 / 7) # un commentaire !
```

```
Out []: 517.2342857142858
```

```
In []: 2 * "AH !" == "AH !AH !"
```


```
Out []: True
```

```
In []: 3 + 4 +
```

```
Out []: SyntaxError: invalid syntax
```

```
In []: "b" / 12
```

```
Out []: TypeError: unsupported operand type(s)  
        for /: 'str' and 'int'
```

 **Question :** pourquoi des instructions fonctionnent et pas d'autres ?

Validité des instructions

Pour être exécutée sans erreur, une instruction (ou un bloc d'instructions) doit être correcte :

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

```
In []: 3 + 4 + # mauvaise syntaxe : '+' attend 2 parametres
```


```
Out []: SyntaxError: invalid syntax
```


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

```
In []: "b" / 12 # syntaxe ok mais 'b' / 12 n'a pas de sens
```

```
Out []: TypeError: unsupported operand type(s)  
        for /: 'str' and 'int'
```


Notion de type

 **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()` :

```
In []: type("La question elle est vite repondue")
Out[]: str
```

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

```
In []: int(4.57)
Out[]: 4
```

```
In []: float('4.57')
Out[]: 4.57
```

```
In []: int("A")
Out[]: 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) :

- `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

```
In []: "bon" + "jour"
```

```
Out []: bonjour
```

```
In []: 4 * "ah "
```

```
Out []: ah ah ah ah
```

```
In []: len("crocodile")
```

```
Out []: 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

❌ **Problème** : On ne va pas programmer bien loin avec seulement des valeurs « constantes »...

💡 **Idée** : On va utiliser des *variables*

📖 **Définition** : une variable est un *conteneur d'information*, formellement une zone de la mémoire qui contient une valeur. Une variable a un nom que l'on appelle *identificateur*.

Identificateurs (= noms de variables)

Syntaxe d'un identificateur : `[_]lettre[lettre/chiffre/_]`

- sensibles à la casse (pouet != POUET)
- ne peuvent pas être 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 : quelques bonnes pratiques

- nom porteur de sens (var_age > lhkvfdbg)
- pas d'accents
- nom_var ou nomvar si nom pas trop complexe
- voir peps.python.org/pep-0008/#naming-conventions !

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 :

```
In []: prenom = "Mirabelle"
```

```
In []: prenom
```

```
Out []: "Mirabelle"
```

```
In []: a = 4 + 5 * 2
```

```
In []: b = 3 * a
```

```
In []: b
```

```
Out []: 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) !

```
In []: a = 4 # affectation
```

```
In []: a == 2 # comparaison
```

```
Out []: False
```

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

```
In []: type(a)
```

```
Out []: int
```

```
In []: b = str(a + 7)
```

```
IN []: b
```

```
Out []: '11'
```

Un détail sur les chaînes de caractères

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

```
In []: chaine = "bonjour"
```

```
In []: chaine[0]
```

```
Out []: "b"
```

```
In []: chaine[1]
```

```
Out []: "o"
```

! 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.

```
In []: animal = "morse"
```

```
In []: animal[0] = "M"
```

```
Out []: TypeError: 'str' object does not support item assignment
```

```
In []: animal = "Morse"
```


Exercice

⚙️ **Exercice** : Parmi les noms suivants, lesquels sont des noms de variables valide ?

```
morse, 4phoque, GIRAFE, (a), s0l31l, 50l31l, u"i", __var__,  
_var-, _var_, /ours/, variable!
```

⚙️ **Exercice** : Que vaut bcdj à la fin de ces instructions ?

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

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

```
In []: nb_1 = 63.5
In []: nb_2 = 4
In []: type(nb_1 + nb_2)
Out[]: float

In []: nb_1 == 67
Out[]: False
```

Entrées/Sorties et Chaînes de Caractères

Instructions, types de bases et variables

Entrées/sorties et chaînes de caractères

- Saisie de valeurs avec `input()`

- Affichage avec `print()` et `format()`

- Résumé

Mémoire et représentation des données

De quoi ? Pourquoi faire ?

On a souvent besoin d'interagir avec un programme :

- saisir des données : les *entrées*
- afficher des résultats ou des messages pour l'utilisateur.ice : les *sorties*


En Python, les fonctions associées sont `input()` et `print()`

Entrées avec `input()`

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

```
In []: variable = input("entrez une valeur : ")
entrez une valeur : # mettons 196
In []: variable
Out[]: '196' # pas un entier !!!

In []: type(variable)
Out[]: str
```

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

```
In []: variable = int(input("entre une valeur : ")) # cast
In []: variable
Out []: 196
```

Sorties 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 est *optionnel* et qu'il a la valeur `valeur` par défaut

Exemple

```
# utilisation basique
```

```
In []: print("bonjour")
```

```
bonjour
```

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

```
In []: age = 28
```

```
In []: taille = 1.78
```

```
In []: print("J'ai", age, "ans et je mesure", taille, "m")
```

```
J'ai 28 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
```

```
In []: print("age", age, "taille", taille, sep=", ", end=".\n AH")
```

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

```
AH
```

Chaînes de caractères formatées

i Remarque : on peut aussi 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

Les formats possibles sont dispos dans la plus dans la [doc](#) !

Exemple

```
In []: prenom = "Madeleine"
```

```
In []: age = 22
```

```
In []: taille = 1.64786513
```

```
# premier test simple
```

```
In []: print(f"Bonjour {prenom} !")
```

```
Bonjour Madeleine !
```

```
# test avec des expressions plus complexes
```

```
print(f"Tu as {(age + 4) / (age - 2)} ans et  
      tu mesures {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  
      on peut dire que je mesure {taille:5.2f}m")
```

```
Euh non, j'ai 22.5 ans et on peut dire que je mesure 1.65m
```

Exercice

⚙️ **Exercice** : écrire un programme qui saisit deux valeurs dans des variables a et b et qui inverse ces variables (b devient a et a devient b).

⚙️ **Exercice** : écrire un programme qui

- lit un temps en seconde et le décompose en heure, minute, seconde.
- affiche ce temps sous le format hh:mm:ss.
- et qui partant de cet affichage, retrouve le temps en seconde et l'affiche pour valider le calcul.

🐾 **Astuce** : le format 02d permet d'afficher des entiers (d) sur au moins deux caractères (2) qui ont par défaut la valeur 0 (0) :

```
In []: n = 5
In []: m = 11
In []: print(f"{n:02d}, {m:02d}")
Out[]: 05, 11
```

Résumé



Rappel :

- `input()` pour faire saisir des valeurs par l'utilisateur.ice
- `print()` pour afficher des valeurs, des messages, etc
- les *f-strings* pour créer des chaînes formatées avec des variables, valeurs, etc



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

```
In []: temp_eau = float(input("temperature de l'eau : "))  
temperature de l'eau : 27.673
```

```
In []: print(f"l'eau est a {100 - temp_eau:.2f} degrees  
        d'etre en ebullition")
```

```
l'eau est a 72.33 degrees d'etre en ebullition
```

Mémoire et représentation des données

Instructions, types de bases et variables

Entrées/sorties et chaînes de caractères

Mémoire et représentation des données

- Mémoire

- Représentation des données

- Gestion de la mémoire en Python

i Remarque : un peu de culture générale (avec conséquences pratiques).
Pas à l'examen mais ce sont des choses importantes à savoir !

Une vue de la mémoire

1

bit : plus petit élément de la mémoire

2 valeurs possibles (0 ou 1)

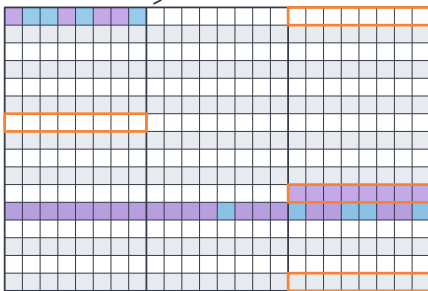
0 1 1 0 1 0 0 1

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 idée sur la représentation des données

❌ **Problème** : la machine n'utilise que des 0 et des 1 en vrai, il faut donc trouver un moyen de tout représenter en binaire ... ALED !

- 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

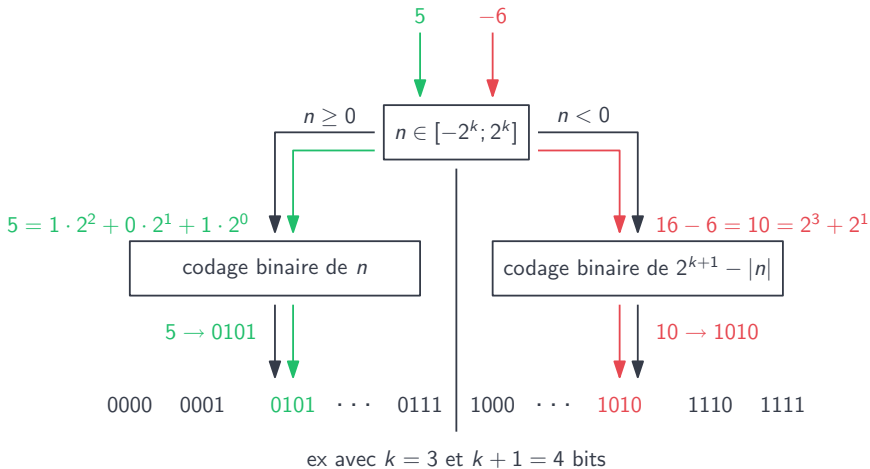
```
'a' -> 97  'b' -> 98 ...  
'A' -> 65  'B' -> 66 ...  
'0' -> 48  '1' -> 49 ...
```

ℹ **Remarque** : on peut donc *comparer* des chaînes avec <, <= etc !

```
In []: "morse" < "phoque" # ordre alphabetique en fait !  
Out []: True
```

Les nombres entiers relatifs

Principe du *complément à 2* : $k + 1$ bits pour coder les entiers de $[-2^k; 2^k]$



Le cas des nombres réels

? Question : les entiers c'est plié, mais comment coder exactement π ?

✗ Problème : cé cui ! Pour les *réels* il faut une *représentation approchée*

Représentation *IEEE 754* : $\pm 2^e \times m$

- \pm : 1 bit de signe
- e : un *exposant* (*biaisé*) en binaire
- m : une *mantisse*, nombre binaire étant une somme de $1/2^i$

$$\pi \quad \simeq \quad \begin{array}{c} + \\ \boxed{0} \\ \text{signe} \end{array} \quad \begin{array}{c} 2^{128} \\ \boxed{10000000} \\ \text{exposant} \\ \text{(biaisé)} \end{array} \times \begin{array}{c} 1.5707963705062866 \\ \boxed{10010010000111111011011} \\ \text{mantisse} \end{array}$$

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 ?

```
In []: 0.125 + 0.125 == 0.25
```

```
Out []: True
```

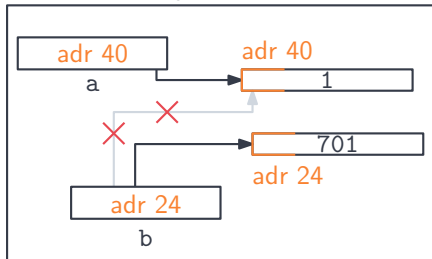
```
In []: 0.1 + 0.1 + 0.1 == 0.3
```

```
Out []: 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



```
In []: a = 1
```

```
In []: b = a
```

```
In []: 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 :

- la mémoire est une suite d'octets, chacun avec une *adresse*
- en Python, les variables contiennent l'adresse à laquelle est stockée sa valeur et non pas directement la valeur
- quand on fait `a = b`, `a` « pointe » sur la même zone mémoire que `b`
- on peut *comparer des chaînes de caractères*



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

```
In []: 0.1 + 0.1 + 0.1 == 0.3
```

```
Out []: False
```

```
In []: 0.1 + 0.1 + 0.1
```

```
Out []: 0.30000000000000004
```