Les bases de Python

1. Bloc d'instruction

En Python chaque ligne donne une **instruction**. Pour séparer certains blocs (fonctions, conditions, boucles etc.) on utilise l'**indentation** (4 espaces). Les commentaires sont précédés d'un `#` et ne sont pas exécutés.

```
note = 20                                   #  instruction
if note == 20:                              #  condition
    print("super note !")                   #  n'est effectué que si
x vaut 20
print("au revoir")                          #  sera toujours effectué
```

La syntaxe Python impose de respecter l'indentation. Chaque bloc qui est indenté est précédé du symbole `:`

Opérations de base :

Exemple	résultat	opération
`3 + 2`	`5`	somme
`3 - 2`	`-1`	soustraction
`3 * 2`	`6`	produit
`3 ** 2`	`9`	puissance
`3 / 2`	`1.5`	division
`3 // 2`	`1`	quotient
`3 % 2`	`1`	reste

Certaines agissent différemment selon l'objet : `'a' + 'b' == 'ab'`

2. Affecter

Pour affecter une valeur à une variable on utilise la notation `variable = valeur`.

Par exemple `note = 20`. On peut inclure n'importe quel objet Python dans une variable.

3. Types de base

- `int` : les entiers naturels. Ex: `2`
- `float` : les nombres à virgule flottantes. Ex: `3.14`
- `boolean` : les booléens. Ex `True, False, a != b`

On obtient le type d'un objet avec la fonction type :

```
>>> type(4)
<class 'int'>
>>> type(1 != 2)
<class 'bool'>
```

4. Types complexes

On a déjà rencontré quatre types complexes : `str`, `list`, `tuple` et `dict`.

a) Les chaînes de caractères. Type `str`

Ce sont des séries de caractères affichables à l'écran (simplification abusive). En Python 3 ils sont *encodés* en `utf-8`. On en reparlera. On les note avec des apostrophes ou des guillemets.

Chaque caractère est numéroté et on peut l'atteindre avec son *indice*.

```
>>> chaine = "Bonjour"
>>> chaine[0]
'B'
```

Les chaînes de caractères ne sont *pas mutables.* Mais on peut opérer dessus !

```
>>> "Bonjour" + "Papy"              # concaténation
'BonjourPapy'
>>> "Bonjour {}".format("Diego")    # formatage, version 1
'Bonjour Diego'
>>> nom = "Robert"
>>> f"Bonjour {nom}"                # formatage, version 2
'Bonjour Robert'
```

b) Les tableaux ou listes. Type `list`

Ce sont des tableaux où les objets sont indexés à partir de 0. On peut atteindre un objet depuis son indice :

```
>>> L = ["a", "b", "c"]          # Les listes sont notées entre [
]
>>> L[2]                         # le TROISIEME élément de L
'c'
```

Les listes sont **mutables**. On peut les modifier **modifier**

```
>>> L.pop(0)                     # renvoyer l'élément 0 et
l'effacer de la liste
'a'
>>> L
['b', 'c']                       # la liste a bien été modifiée.
>>> L.append("d")                # ajouter un élément à la fin de
la liste
>>> L
['b', 'c', 'd']
```

c) Les tuples

Ce sont des séries indexées d'objets. Ils sont notés avec des parenthèses. Les tuples **ne sont pas mutables.** C'est comme les listes, mais pas mutable et plus rapide.

```
>>> tup = (1, 2, 3)
(1, 2, 3)
>>> tup.pop(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'pop'
```

d) Les dictionnaires (tables de hashage), type `dict`

Ce sont des tables d'associations entre une clé et une valeur. On les notes entre accolades. Attention, dans les accolades on utilise `:`.

Les dictionnaires sont mutables.

```
>>> dict = {"DS": 5, "DM": 1}    # remarquez les :
>>> dict["DM"]                   # obtenir un élément d'un
dictionnaire
1
>>> dict["IE"] = 2               # ajouter une nouvelle paire
clé:valeur
>>> del dict["DM"]               # del permet d'effacer un
élément d'un mutable
>>> dict
{"DS": 5, "IE": 2}
```

Les conditions : `if elif else`

- Permettent de réaliser des choix ou de tester des résultats.
- s'utilisent dans cet ordre : if, elif (optionnel), else (optionnel)

La syntaxe est :

```
if condition_1:                              # les : sont
obligatoires !
    bloc execute si condition_1 est True     # indentation !!!
elif condition_2:
    bloc execute si condition_1 est False et condition_2 True
else:
    bloc execute si condition_1 est False et condition_2 False
bloc toujours execute car pas indente
```

- Les comparaisons entre nombres sont : `==`, `>`, `<`, `>=`, `<=`, `!=`
- On peut tester si un objet est dans un autre avec `in`

```
>>> 4 in [1, 2, 3]
False
>>> "DM" in {"DS": 5, "DM": 1}
True
```

- On peut tester plusieurs conditions grace aux opérateurs booléens `and` et `or`

  Attention c'est un simplification et `and` et `or` sont en réalité plus compliqués.

Les boucles : `for` et `while`

Il existe deux types de boucles :

- `for` quand on veut parcourir un objet ou qu'on connait le nombre d'étapes.
- `while` quand on ne connait pas le nombre d'étapes.

Boucle `for`

En Python, `for` parcourt toujours une collection d'objet (un *itérable*).

On construit une série de nombres avec `range(debut, fin, pas)`. Attention on s'arrête toujours *avant* `fin`.

```
>>> for k in range(0, 3, 1): # k varie de 0 à 2 avec un pas de
1
...     k                     # indentation !!!
...
0
1
2
```

On peut parcourir une liste, un tuple, une chaîne ou un dictionnaire (vu plus tard).

```
>>> for nombre in [0, 2, 4]:
...     nombre ** 2
...
0
4
16
>>> for lettre in 'abc':
...    lettre
'a'
'b'
'c'
```

Boucle `while`

Effectuer quelque chose *tant qu'une condition est vraie*.

Pour arrêter une boucle `while` **il faut que la condition devienne FAUSSE**.

Exemple : chaque mois je gagne 100€. Combien de mois avant d'avoir 2.000€ ?

```
>>> mois = 0                    # sera mon compteur
>>> somme = 0                   # je vais ajouter mes revenus dans
somme
>>> while somme < 2000:         # tant que ma somme est plus
petite que 2000...
...     somme = somme + 100     # je cumule ma somme
...     mois = mois + 1         # et je compte
...
>>> mois                        # remarquez que mois contient la
bonne valeur...
20
```

Boucle infinie

- On exécute une boucle *infinie* avec `while True:`
- Pour l'arrêter, on peut utiliser le mot clé `break`

Fonctions

Qu'est ce qu'une fonction ?

Les fonctions sont des morceaux de code qu'on peut *appeler* quand on le souhaite.

On doit d'abord définir la fonction avant de l'appeler.

Les fonctions, comme en maths, prennent des *paramètres d'entrée* et *retournent* toujours une valeur.

Si elle font autre chose, comme afficher du texte ou modifier la mémoire, on parle d'*effet de bord*.

Exemples

Par exemple, `len` est une fonction qui renvoie la longueur d'un itérable.

```
>>> len("abcd")
4
```

- Son paramètre d'entrée est un itérable (`list, tuple, dict, str` etc.)
- Sa valeur de retour est un `int`
- À ma connaissance elle n'a pas d'effet de bord.

Autre exemple, `print` est une fonction qui prend une chaîne ou n'importe quel objet qu'on peut écrire à l'écran et qui l'affiche. Elle renvoie `None` (rien).

- Son paramètre d'entrée est une chaîne ou n'importe quel objet affichable
- Sa valeur de retour est `None` (rien…)
- Son effet de bord et d'afficher à l'écran quelque chose.

Définir une fonction

- Une fonction se définit avec `def nom(parametre):` suivi d'un bloc indenté.
- On retourne une valeur avec `return truc_de_sortie`

```
def carre(x):
  '''
  documentation : détaillé ci-dessous
  '''
  return x ** 2
```

Documentation

Quand vous définissez une fonction, **vous devez toujours la documenter.**

La documentation est dans une *chaîne sur plusieurs lignes* entre triples apostrophes. On y trouve :

- La description rapide de la fonction
- les paramètre d'entrée :
- les paramètres retournés :
- un exemple, quand c'est faisable :

```
'''
Renvoie le carré d'un nombre
@param x: (int ou float)
@return: (int ou float)
>>> carre(2)
4
'''
```

Indications de types

On peut considérablement abréger la documentation en indiquant les types.

Plutôt que d'écrire :

```
def somme_trois_nombres(a, b, c):
    """
    Renvoie la somme des trois nombres.

    @param a: (int)
    @param b: (int)
    @param c: (int)
    @return: (int)
    """
    return a + b + c
```

On peut se contenter de :

```
def somme_trois_nombres(a: int, b: int, c: int) -> int:
    """Renvoie la somme des trois nombres."""
    return a + b + c
```

entrée != saisie et sortie != affichage

- utiliser `input` permet de demander à l'utilisateur d'un programme de taper une valeur. Ce n'est pas le paramètre d'entrée d'une fonction
- utiliser `print` affiche du texte, ce n'est pas le paramètre de sortie d'une fonction. Paramètre de sortie : `return`

Les erreurs ou exceptions

Lire les messages d'erreur

Quand un programme génère une erreur, Python *lève une exception*.

Il affiche alors un message commençant par `Traceback` et se lisant de bas en haut :

```
>>> l = [0, 1, 2]
>>> l[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- Le type d'exception : `IndexError`
- la ligne : 1
- d'autres informations (par exemple quelle série d'appels de fonctions a provoqué cette erreur)

Les erreurs courantes sont :

Erreurs produises quand Python *lit* le fichier .py :

- `IndentationError` : mélange entre espaces et tabulations ou indentation incorrecte
- `SyntaxError` : oublier des parenthèses, oublier `:` après `if`, `print 3`,

Erreurs produites durant l'exécution du programme lui même :

- `ZeroDivisionError` : division par 0,
- `IndexError` : aller chercher un élément inexistant dans une liste,
- `ValueError` : racine carrée d'un nombre négatif etc.
- `TypeError` : réaliser une opération impossible sur ce type (`len(print)`),
- `AssertionError` : quand un *test* avec `assert` est faux

```
>>> assert 2 == 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

`try: ... except Exception as e: ...`

On peut *attraper* une exception prévisible avec `try except`. C'est à la limite du programme mais vous le rencontrerez souvent dans le code.

C'est un procédé courant, en particulier pour gérer les saisies d'utilisateur.