Introduction à la Programmation des Algorithmes

5.1. Python – Syntaxe générale

François Fleuret
https://fleuret.org/11x001/

Nous avons vu dans 1.2. "Introduction – Langages de programmation" que le langage Python est **interprété**, c'est à dire que le fichier source sera lu par un **interpréteur Python** qui est un exécutable.

Si le fichier `truc.py` contient le code source suivant:

```
1  for k in range(5):
2      print(k)
```

exécuter dans un terminal

```
python truc.py
```

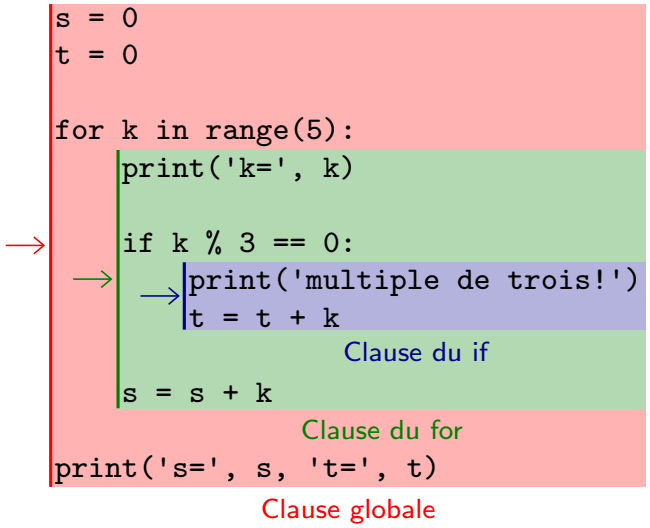
affiche

```
0
1
2
3
4
```


Contrairement à la plupart des langages, dont le C, l'indentation est primordiale en Python.

Une clause n'est pas définie par un groupe d'instructions délimité par des `{ }` ou un `;` mais par **un bloc d'instructions indentées de manière identique**.

Contrairement au C où l'indention facilite la lecture du code source pour les humains sans que cela ne change le sens du programme, en Python elle module la manière dont l'interpréteur "comprend" le programme.



L'interpréteur Python détecte seul automatiquement combien le programme utilise d'espaces pour indenter un bloc (le standard est 4) et indique une erreur si l'indentation est incohérente.

```
1  s = 3
2
3  for k in range(10):
4      s += k
5
6  print(s)
```

```
1  s = 3
2
3  for k in range(10):
4   s += k
5
6  print(s)
```

```
1  s = 3
2
3  for k in range(10):
4      s += k
5
6   print(s)
```

```
python test.py
  File "test.py", line 6
    print(s)
          ^
IndentationError: unindent does not match any outer indentation level
```


Comme en C, il est possible de rajouter des commentaires qui seront ignorés lors de l'exécution du programme. On indique qu'une partie de ligne est un commentaire en la préfixant avec un `#`.

```
1  print(3)
2
3  # je viens d'afficher 3 et je vais afficher 42!
4
5  print(42)
```

Python possède de manière native des types complexes pour gérer des listes ou des collections. Nous y reviendrons ultérieurement.

Les types de base sont:

- `int` représente un entier similaire au type `int` en C,
- `float` représente un réel similaire au type `float` en C,
- `str` représente une chaîne de caractères,
- `bool` représente une valeur booléenne.

Python possède un type supplémentaire `NoneType` qui ne peut prendre que la valeur `None` et permet par exemple de représenter une grandeur qui n'est pas encore disponible.

Python possède donc de manière native des types dont la manipulation demandent des opérations sophistiquées en mémoire (allocation, copies) et cache cette complexité au programmeur.

Il permet donc en particulier de manipuler les chaînes de caractères de façon infiniment plus simple qu'en C.

Cette complexité sous-jacente fait que Python est un langage très lent. En pratique les opérations coûteuses sont faites à l'aide de librairies écrites dans un langage de bas niveau comme le C.

Les chaînes de caractères en Python sont encodées en Unicode dont le standard actuel (08.2021) inclut 140'000 caractères, dont toutes les lettres accentuées, les alphabets chinois, cyrillique, etc. ainsi que les émojis.

Une chaîne littérale en Python peut être délimitée avec " ou '.

```
In [1]: a = "Une chaîne normale, avec un accent"

In [2]: b = "Это русский"

In [3]: c = "🎂🍾"

In [4]: print(a, b, c)
        Une chaîne normale, avec un accent Это русский 🎂🍾

In [5]: d = a + b + c

In [6]: d
Out[6]: 'Une chaîne normale, avec un accentЭто русский🎂🍾'
```

Il est possible de noter une chaîne de caractères littérale sur plusieurs lignes en la délimitant avec """.

```
a = """ceci
est une chaînes sur
plusieurs lignes!!!
"""
```

Et la fonction `len` retourne la longueur d'une chaîne

```
>>> len("123456")
6
```


Nous utiliserons dans les exemples la fonction `print` qui affiche simplement ses arguments dans l'ordre, suivis d'un retour à la ligne.

Nous verrons plus tard comment faire des formatages plus sophistiqués.

L'existence des chaînes de caractères gérées automatiquement permet en particulier de faire facilement des saisis.

La fonction `input` retourne une chaîne de caractères entrée interactivement par l'utilisateur.

```
1  nom = input("Entrez votre nom: ")
2  print("Bonjour chère/cher", nom)
```

affiche (en vert ce que tape l'utilisateur)

```
Entrez votre nom: Francois
Bonjour chère/cher Francois
```


Contrairement au langage C, Python ne demande pas de déclarer une variable avant de l'utiliser.

Une variable est créée lors de la première affectation. De plus son type peut changer à la suite d'une autre affectation.

La fonction `type` retourne le type d'une expression.

```
1  a = 3
2  print(a, type(a))
3  a = "toto"
4  print(a, type(a))
```

affiche

```
3 <class 'int'>
toto <class 'str'>
```

Il est possible de faire plusieurs affectations d'un coup en séparant les variables et leurs valeurs par des virgules:

```
1  a, b = 3, 4
2  print(a, b)
```

affiche

```
3 4
```


Comme le C, Python fait des conversions de types numériques implicitement pour garder la précision autant que possible:

```
1  a = 3
2  b = 4.2
3  print(a, type(a))
4  print(b, type(b))
5  print(a + b, type(a + b))
```

affiche

```
3 <class 'int'>
4.2 <class 'float'>
7.2 <class 'float'>
```

Et Python permet des conversions explicites avec des fonctions associées aux différents types.

```
1  x = 3
2  print(x, type(x))
3
4  x = float(x)
5  print(x, type(x))
6
7  x = str(x)
8  print(x, type(x))
9
10 x = float(x)
11 print(x, type(x))
```

affiche

```
3 <class 'int'>
3.0 <class 'float'>
3.0 <class 'str'>
3.0 <class 'float'>
```


Le langage Python offre des opérateurs similaires à ceux du C. Les principales différences sont:

- les opérateurs de post/pré-incrément/décrément ++ et -- n'existent pas,
- l'opérateur / fait toujours le calcul en virgule flottante, et Python a // pour la division euclidienne,

```
1  a = 14
2  b = 5
3  print(a / b, a // b)
```

affiche

```
2.8 2
```

- les opérateurs booléens sont and, or, et not.

```
1  a = True
2  b = 4 < 5
3  print((not a) or b)
```

affiche

```
True
```

Python possède aussi des opérateurs de concaténation de chaînes de caractères + et +=.

```
1  a = "le chat"
2  b = "chasse"
3  c = "la souris"
4
5  s = a + " " + b
6  print(s)
7
8  s += " " + c
9  print(s)
```

affiche

```
le chat chasse
le chat chasse la souris
```


De plus l'opérateur [] permet d'extraire un caractère (en réalité une sous-chaîne de longueur 1) d'une chaîne de caractères

```
1  s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2  print(s[0], s[1], s[2], s[24], s[25])
```

affiche

```
A B C X Y
```

Cet opérateur permet également d'extraire une sous-chaîne à l'aide du symbole : pour spécifier les indexes du premier caractères à prendre et l'indexe qui suit le dernier caractère à inclure.

Si le premier est absent, la valeur par défaut est 0.

Si le second est absent, la valeur par défaut est la longueur de la chaîne complète, et s'il est négatif, la valeur est la longueur complète diminuée d'autant.

```
1  s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2  t = s[3:5]
3  u = s[:5]
4  r = s[16:]
5  v = s[16:-2]
6  print(type(t), t, u, r, v)
```

affiche

```
<class 'str'> DE ABCDE QRSTUVWXYZ QRSTUVWX
```


La manière la plus simple et standard de formater des valeurs à afficher consiste à utiliser une chaîne de caractères formatée, aussi appelées *f-string*.

Il suffit de la préfacer par `f` et de spécifier les expressions à substituer entre `{}` dans le corps de la chaîne.

```
1  a = 3
2  b = 4.5
3  c = "Bob"
4
5  print(f"Nous avons {a} et {a + b} et aussi {c}")
```

affiche

```
Nous avons 3 et 7.5 et aussi Bob
```

Il est possible de définir plus précisément le format (notation scientifique, nombre de chiffre après la virgule, etc.)

Contrôle du flot

Python dispose d'instructions de contrôle du flot similaires à celles du C:

- `if`,
- `for`,
- `while`,
- `break`, et `continue`.

En revanche Python n'a pas l'équivalent du `do-while` et du `switch-case`.

La structure `if` en Python accepte en option autant de clauses `elif` que nécessaire et/ou une clause `else` finale.

```
if condition_1:
    clause_1
elif condition_2:
    clause_2
    ...
elif condition_k:
    clause_k
else:
    defaut
```

La clause `elif` permet d'éviter une accumulation d'indentations.

La structure `for` en Python est différente du C. Au lieu de spécifier une clause de départ et d'itération et une condition, il faut indiquer une ou plusieurs variables de boucles et un élément "itérable".

```
for var_1, var_2, ..., var_k in iterateur:
  clause
```

L'itérateur le plus simple est `range` qui permet de définir une série de valeurs entières.

```
for k in range(5):
    print(k)
```

affiche

```
0
1
2
3
4
```

La création d'un itérateur avec `range` peut se faire avec un, deux, ou trois arguments entiers:

- `range(borne)`
- `range(premier, borne)`
- `range(premier, borne, increment)`

`borne` n'est **jamais** atteint.

Par exemple

- `range(6)` $= \{0, 1, 2, 3, 4, 5\}$,
- `range(3, 10)` $= \{3, 4, 5, 6, 7, 8, 9\}$,
- `range(3, 10, 3)` $= \{3, 6, 9\}$.

Une chaîne de caractères est itérable caractère par caractère:

```
1  for c in "blah":
2      print(c)
```

affiche

```
b
l
a
h
```

Finalement, le mot clé `pass` permet de spécifier qu'une clause est vide, l'équivalent de {} en C.

```
1  a = 2
2
3  if a > 3:
4      pass
5  else:
6      print("Plus petit ou égal à trois")
```

affiche

```
Plus petit ou égal à trois
```


Exemples

Nous avions écrit ce programme en C pour estimer la racine carrée de 2:

```c
float epsilon = 1e-6, x = 2;
float a = 0, b = 1 + x, c;

while(b - a >= epsilon) {
  c = (a + b) / 2;
  if (c * c >= x) {
    b = c;
  } else {
    a = c;
  }
}

printf("sqrt(%f) ~ %f\n", x, c);
```


La version Python est très similaire:

```python
epsilon = 1e-6
x = 2
a = 0
b = 1 + x

while b - a >= epsilon:
    c = (a + b) / 2
    if (c * c >= x):
        b = c
    else:
        a = c

print(f"sqrt{x} ~ {c}")
```

Nous avions écrit ce programme en C pour estimer π:

```c
float x, y, N = 0, M = 0, delta = 5e-4;

for(x = -1; x <= 1; x += delta) {
  for(y = -1; y <= 1; y += delta) {
    N++;
    if(x * x + y * y <= 1) M++;
  }
}

printf("PI ~ %f\n", 4 * M / N);
```

affiche

```
PI ~ 3.141493
```


La version Python est très similaire:

```python
N = 0
M = 0
delta = 5e-4

x = 0
while x <= 1:
    y = 0
    while y <= 1:
        N += 1
        if x * x + y * y <= 1: M += 1
        y += delta
    x += delta

print(f"PI ~ {4 * M / N}")
```

affiche

```
PI ~ 3.140445769119438
```

Lister des nombres premiers:

```c
int n, m;
for(n = 2; n < 25; n++) {
  m = 2;
  while(m < n && n % m > 0) m++;
  if(m == n) printf("%d\n", n);
}
```

affiche

```
2
3
5
7
11
13
17
19
23
```



```python
for n in range(2, 25):
    m = 2
    while m < n and n % m > 0: m += 1
    if m == n: print(n)
```

affiche

```
2
3
5
7
11
13
17
19
23
```