Constructions élémentaires en Python

1 Pourquoi un langage de programmation ?

L'Informatique s'articule autour de quatre notions :

1. **Algorithme = séquence d'actions ou de calculs** : Par exemple l'algorithme de la multiplication égyptienne qui existe depuis l'Antiquité...
2. **Machine** Pourquoi ne pas créer des machines capables d'exécuter les algorithmes à notre place ?
3. **Langage** : Comment communiquer avec les machines ? **A travers un langage de programmation**
4. **Information** Un texte, une image, un son , une vidéo sont numérisés puis transformés avec des algorithmes.

2 Langage de programmation

Il existe beaucoup de langages de programmation.

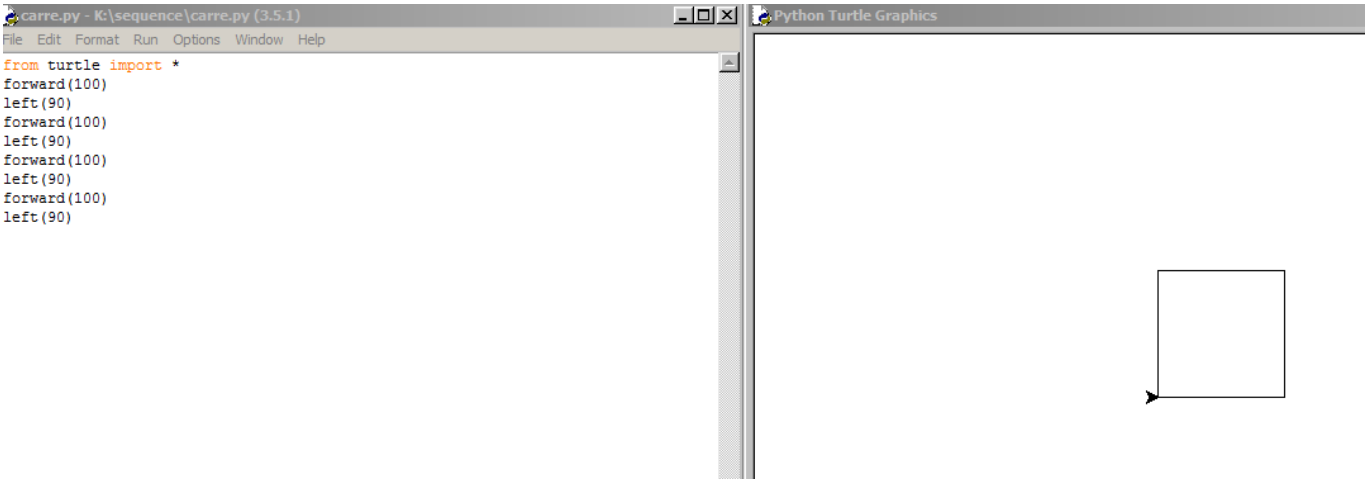
Les plus connus sont : C, C++, Java, JavaScript, OCaml, Rust et Python qu'on va étudier au Lycée pour sa simplicité d'utilisation.

Les briques élémentaires d'un langage de programmation **impératif** comme Python sont :

1. La séquence d'instructions
2. La notion de variable et d'affectation
3. Les tests (if)
4. Les boucles bornées (boucle for)
5. Les boucles non bornées (boucle while)

3 Séquence d'instructions

1. Un programme est une suite d'instructions (exécutées dans l'ordre)
   Voici un exemple de programme Python utilisant le module turtle
2. Chaque ligne du programme correspond à une action de la Tortue, **les actions ou instructions sont exécutées en séquence dans l'ordre**
3. La dernière ligne semble inutile mais elle a deux intérêts : elle remet la tortue dans son état initial, et on voit mieux la répétition
4. On peut régler la vitesse de déplacement de la tortue et la régler au minimum en ajoutant l'instruction `speed(1)`. Prendre le temps de bien comprendre le déplacement de la tortue en lien avec le code, quitte à exécuter plusieurs fois le programme.
5. Peut on changer l'ordre des instructions ?
   Devinez ce que fait ce programme **avant de l'exécuter**




```
from turtle import *

speed(1)

left(90)
forward(100)

left(90)
forward(100)

left(90)
forward(100)

left(90)
forward(100)
```

4 Notion de fonction

La notion de fonction n'est pas fondamentale pour la "puissance de calcul" d'un langage de programmation mais **elle aide à la lisibilité d'un programme** en **factorisant le code**

Que signifie cette expression ?

Imaginons que l'on veuille à plusieurs endroits du programme faire dessiner à la Tortue un carré de côté 100 pixels, autant donner un nom à cette action par exemple `DESSINER UN CARRE de 100 PIXELS`

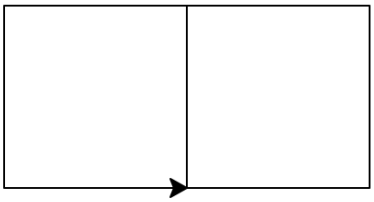
En Python comme en mathématiques on définit d'abord la fonction avec le mot réservé **def** pour define, puis ensuite on l'applique par exemple pour construire deux carrés

```
from turtle import *
def dessine_carre_100():
```

```
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)

dessine_carre_100()
goto(100,0)
dessine_carre_100()
```

On observe alors



On insiste sur la manière de définir une fonction :

1. On isole la séquence d'instructions qui définie la fonction, on dit que cette séquence forme le **corps de la fonction** ou **bloc de la fonction**
2. Ce bloc commence juste après les deux points : et est isolé par le décalage ou **indentation** vers la droite en général de quatre caractères
3. Toute instruction qui n'est pas alignée sur la ligne du bloc (ligne bleue) ne fait donc pas partie du bloc ainsi l'instruction `dessine_carre_100()` ne fait pas partie du bloc de la fonction

```
def dessine_carre_100():

    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)
    forward(100)
    left(90)

dessine_carre_100()
goto(100,0)
dessine_carre_100()
```

Le programme suivant a des erreurs :

```
from turtle import *
```

```
def dessin()
forward(70)
left(216)
forward(70)
left(216)
forward(70)
left(216)
forward(70)
left(216)
forward(70)
left(216)
```

Si on l'exécute un premier message d'erreur apparaît :

```
    def dessin()
                ^
SyntaxError: invalid syntax
```

Il manque les deux points, donc on corrige

```
 from turtle import *
def dessin():
forward(70)
left(216)
forward(70)
left(216)
forward(70)
left(216)
forward(70)
left(216)
forward(70)
left(216)
```

Si on l'exécute un deuxième message d'erreur apparaît :

```
    forward(70)
    ^
IndentationError: expected an indented
block
```

On a oublié **d'indenter le bloc de la fonction**, on corrige et on exécute

```
from turtle import *
def dessin():
    forward(70)
    left(216)
    forward(70)
    left(216)
    forward(70)
    left(216)
    forward(70)
```

```
    left(216)
    forward(70)
    left(216)
```

Cette fois ci il n'y a pas d'erreur mais rien ne se passe

C'est normal **on a oublié d'exécuter la fonction**, d'où

```
from turtle import *
def dessin():
    forward(70)
    left(216)
    forward(70)
    left(216)
    forward(70)
    left(216)
    forward(70)
    left(216)
    forward(70)
    left(216)
ht()
dessin()
```

Et finalement on obtient

5 Notion de variable et d'affectation

La fonction précédente est trop "rigide". On aimerait pouvoir tracer un carré de côté "**variable**"

On introduit **une variable** dans la fonction, comme en mathématiques où on écrit $f(x)$ on écrit `dessine_carre(cote)` où `cote` est une variable

Ce qui donne

```
from turtle import *
def dessine_carre(cote):

    forward(cote)
    left(90)
```

```
forward(cote)
left(90)
forward(cote)
left(90)
forward(cote)
left(90)
```

Une variable est un **identificateur** (une étiquette) par exemple **cote** associée à une **référence** que l'on peut représenter par une boîte dans un souci de simplification) contenant une **valeur** (par exemple 100). Cette valeur peut changer au cours de l'exécution du programme.

On peut comparer une variable à un nom , la référence à une boîte aux lettres et la valeur au contenu de la boîte aux lettres, **attention ! plusieurs noms peuvent être écrits sur la même boîte aux lettres**

Pour **faire évoluer la référence associée à une variable** on dispose d'une opération importante appelée **l'affectation** :

L'**affectation** correspond à une instruction de la forme :

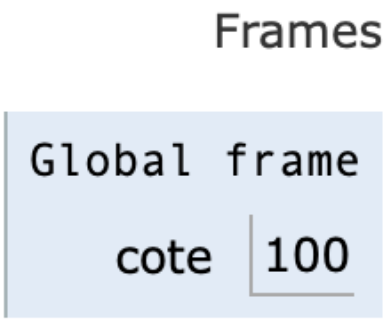
```
variable = expression
```

une expression est formée à partir des nombres, des variables et des opérations comme $+$, $\times$, etc...

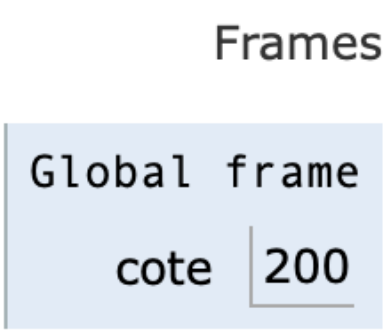
par exemple :

```
cote = 100
cote = 2*cote
```

Dans la référence associée à la variable `cote` il y a eu dans un premier temps la valeur 100 ce que l'on peut **visualiser** ainsi grâce au site suivant `http://pythontutor.com/visualize.html#mode=display`



puis cette valeur a été multipliée par 2 puis le résultat, c'est à dire 200 a été remis dans la mémoire à l'endroit référencé par la variable nommée `cote`



En pseudo-code le symbole d'affectation est ←, on voit mieux ainsi dans quel sens lire l'instruction de la droite vers la gauche et le programme précédent en pseudo-code est :

début
  cote ← 100
  cote ← 2*cote
fin

6 Entrées et Sorties

Une affectation peut prendre en compte une entrée au clavier par l'utilisateur du programme.

Ainsi dans le programme suivant on laisse à l'utilisateur le soin de préciser la longueur d'un côté. Comment ?

```
longueur_cote = int(input("Quelle est la longueur du côté du carré?"))
dessine_carre(longueur_cote)
```

La fonction `input(".....")` affiche à l'écran ce qu'il y a entre guillemets par exemple ici la question Quelle est la longueur du côté du carré ?

Imaginons que l'utilisateur tape au clavier 50, pour 50 pixels. Pour l'utilisateur 50 est un nombre entier, **mais pour la machine ce qui est tapé au clavier est codé sous la forme d'une chaîne de caractères**, voilà pourquoi il faut convertir cette chaîne de caractères en nombre entier en utilisant la fonction `int()`

(int pour integer qui signifie nombre entier en anglais)

Si on veut afficher à l'écran le contenu d'une variable ou la valeur d'une expression on utilise la fonction `print()`

7 Fonction qui renvoie un résultat

Si dans un programme à plusieurs reprises on doit calculer l'aire d'un carré de côté `cote` variable, il est judicieux de définir une fonction `aire(cote)` ainsi

```
def aire(cote):
    return cote*cote
```

Le mot réservé `return` signifie qu'une fois calculé cote*cote le résultat est renvoyé à l'endroit où a été appelée la fonction

Regardons cela sur un exemple où on a trois carrés de côtés 100, 145 et 175 et on veut calculer les aires des carrés

Si on veut mémoriser les résultats on peut procéder ainsi

```
cote = 100
aire_1 = aire(cote)
cote = 145
aire_2 = aire(cote)
cote = 175
aire_3 = aire(cote)
```

L'instruction `aire_1 = aire(cote)` se lit de la droite vers la gauche :

La fonction `aire(cote)` est exécutée avec la variable `cote` ayant pour valeur 100, le résultat est **renvoyée** puis affecté à la variable `aire_1`

On a vu l'importance du symbole : et de l'indentation pour délimiter le bloc de la fonction, dans d'autres langages comme JavaScript, ce sont des accolades qui délimitent le bloc de la fonction, l'indentation n'a qu'un rôle esthétique mais ne fait pas partie de la **syntaxe** du langage :

```
function somme(x,y,z){
/* param     : trois entiers x,y,z
   résultat  : un entier

   renvoie la somme de x,y et z */
   return x+y+z;
}
```

8 Exercices

Ex 1

1. Quelles sont les valeurs des variables X et Y à la fin de l'algorithme suivant
2. Si on permute les deux dernières instructions aura-t-on le même résultat ?

---

début
  X ← 10
  Y ← 2
  Y ← X + Y
  X ← X * Y
fin

Ex 2

1. Quelles sont les valeurs des variables X et Y à la fin de l'algorithme suivant
2. Si on permute les deux dernières instructions aura-t-on le même résultat ?

---

début
  X ← 5
  Y ← 4
  Y ← 2*Y
  X ← X * X
fin

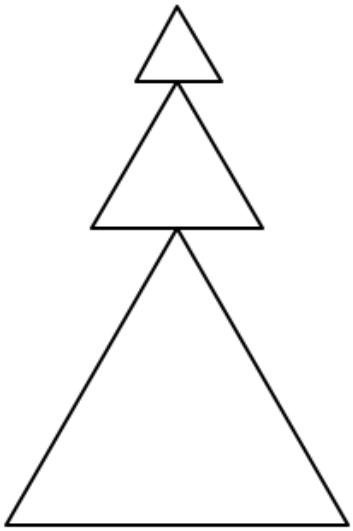
Ex 3

La fonction suivante est censée dessiner un carré, corriger toutes les erreurs

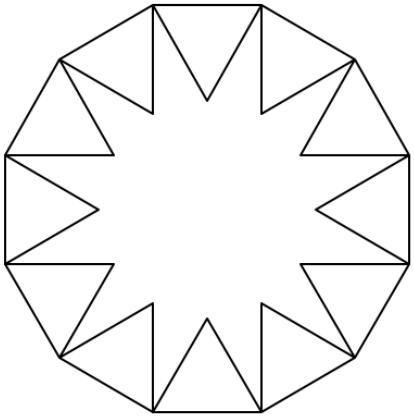
```
from turtle import *
def dessine_carre(cote)
forward(cote)
    left(90)
        forward(cote)
            left(90)
    forward(cote)
left(90)
    forward(cote)
        left(90)
```

Ex 4

1. Définir une fonction python `dessine_triangle(cote)` qui dessine un triangle équilatéral de côté `cote`
2. Ecrire un programme avec le module Turtle pour réaliser le dessin suivant



3. idem



Ex 5

L'aire d'un parallélogramme est défini par la moyenne de son petit côté et de son grand côté (les deux côtés parallèles) multipliée par la hauteur égale à la distance entre les deux côtés parallèles

1. Définir une fonction Python `aire_parallelogramme(petit_cote,grand_cote)` qui renvoie l'aire d'un parallélogramme
2. Ecrire un programme où on demande à l'utilisateur les petit côté, grand côté et hauteur d'un parallélogramme et qui affiche le message, par exemple si le petit côté vaut 3, le grand 5 et la hauteur 6 :

   ```
   l'aire du parallélogramme vaut : 24
   ```

Ex 6

Faire la même chose avec le volume d'un cylindre de hauteur h et dont la base a pour rayon r

Ex 7

Corriger l'erreur

```
Traceback (most recent call last):
  File "/Users/vallon/Python/erreurs.
py", line 1, in <module>
    forward(100)
NameError: name 'forward' is not defi
ned
```