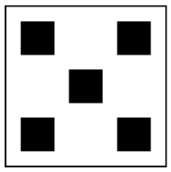
Afin de répéter ou d'adapter à différentes valeurs une portion de code ou simplement afin de rendre un programme plus lisible, on utilise les fonctions.

1 Problème : dessiner une face d'un dé

Supposons que l'on souhaite dessiner une face de dé avec Turtle :



Nul besoin d'écrire le code pour savoir que celui-ci va être assez long et répétitif, dessiner le carré extérieur, bouger, dessiner un carré noir, etc. Cependant, la description que l'on vient de faire est assez concise et lisible, détaillons donc en quoi consiste « dessiner le carré extérieur » et dessiné un carré noir :

```
#dessiner le carre exterieur (deplacement deja fait)
for i in range(4):
    forward(100)
    right(90)


#dessiner un carre noir (deplacement deja fait)
begin_fill()
for i in range(4):
    forward(20)
    right(90)
end_fill()
```

Avec les fonctions, nous allons voir comment rendre ces portions de codes réutilisable afin d'écrire de manière concise le dessin de la figure.

2 Définir et appeler une fonction

2.1 Fonction sans argument

Pour **définir** une fonction on utilise la syntaxe suivante :

```
def nom_fonction():
    <code reutilisable>
```

Il faut bien noter que le code de la fonction (on parle de son **corps**) est indenté par rapport au mot clé `def`. La définition prend fin quand l'indentation revient au niveau initial.

Une fois la fonction définie, il est possible de **l'appeler**, c'est à dire exécuter son code en écrivant à l'endroit du programme voulu :

```
nom_fonction()
```

Par exemple, dans le cadre de notre problème, on peut définir la fonction suivante :

```
def carre_plein():
    down()
    begin_fill()
    for i in range(4):
        forward(20)
        right(90)
    end_fill()
    up()
```

De cette manière, à chaque fois que l'on voudra tracer un carré plein de 20 unités de côté, il suffira d'écrire `carre_plein()`. Tester cette fonction.

Remarque : Il est normal qu'il ne se passe rien lorsqu'on exécute un programme dans lequel on n'a fait que définir une fonction ! Pour que le de le la fonction soit exécuté, il faut l'appeler (qui que être fait dans le programme ou éventuellement dans le mode interactif).

2.2 Fonction avec argument

Il est assez dommage que notre fonction `carre_plein` ne soit capable de tracer que des carrés de côté 20. On aimerait plutôt pouvoir spécifier à l'appel de la fonction de quelle longueur doit être le côté du carré tracé. Pour cela on ajoute un **argument** à notre fonction.

Pour définir une fonction avec un argument on utilise la syntaxe suivante :

```
def nom_fonction(argument):
    <code reutilisable utilisant l argument>
```

Pour appeler la fonction il faudra alors donner une valeur à l'argument.

```
nom_fonction(<valeur de l argument>):
```

Dans la définition, l'argument représente un trou dans le code qui sera rempli uniquement à l'appel de la fonction. Modifions notre fonction :

```
def carre_plein(cote):
    down()
    begin_fill()
    for i in range(4):
        forward(cote)
        right(90)
    end_fill()
    up()
```

La variable `cote` n'est pas une variable étant définie avant dans le code, elle représente le trou devant être rempli (par une valeur ou une variable) à l'appel de la fonction.

```
#dans la definition, on fixe le "code a trou" de la fonction
down()
begin_fill()
for i in range(4):
    forward(.....)
    right(90)
end_fill()
up()
```

```
#code execute a l appel de carre_plein(30)
down()
begin_fill()
for i in range(4):
    forward(30)
    right(90)
end_fill()
up()
```

```
#code execute a l appel de carre_plein(x)
down()
begin_fill()
for i in range(4):
    forward(x)
    right(90)
end_fill()
up()
```

Attention : Il ne faut pas confondre les arguments des fonctions avec les `input` qui sont des moyens de communiquer avec un utilisateur. De manière générale, on ne mettra pas d'input dans des fonctions sauf dans le cas précis où la fonction en question sers à dialoguer avec un utilisateur.

On peut retenir que les `input` servent d'interface entre le programmeur (qui code) et l'utilisateur (qui ne code pas) alors que les fonctions ne sont que des raccourcis dans le code et sont donc créées et utilisées par un programmeur.

2.3 Arguments multiples

On peut encore améliorer notre fonction `carre_plein`. En effet, il pourrait être intéressant de choisir lors de l'appel de la fonction les coordonnées à partir desquelles le carré est tracé (coin supérieur gauche). Pour cela, il est possible dans la définition de donner plusieurs arguments.

Pour définir une fonction avec plusieurs arguments on utilise la syntaxe suivante :

```
def nom_fonction(argument1, argument2, ...):
    <code reutilisable utilisant les arguments>
```

À l'appel de la fonction il faudra alors donner des valeur à ces arguments en respectant l'ordre donné par la définition.

```
nom_fonction(<valeur de l argument 1>, <valeur de l argument 2>, ...):
```

Modifions encore notre fonction en lui ajoutant deux arguments `x` et `y` représentant les coordonnées de départ pour le tracé du carré :

```
def carre_plein(x, y, cote):
    up()
    goto(x,y)
```

```
    down()
    begin_fill()
    for i in range(4):
        forward(cote)
        right(90)
    end_fill()
    up()
```

Ainsi la commande `carre_plein(15, 30, 40)` trace un carré plein de côté 40 en partant des coordonnées (15,30).

Remarque : Pour désigner les arguments d'une fonction on parlera aussi souvent de paramètres ou variables de la fonction mais le terme argument est le plus précis. Pour encore plus de précision, on pourra distinguer les **arguments formels** désignant les arguments lors de la définition (exemple : `x, y, cote`) des **arguments effectifs** désignant les arguments lors de l'appel de la fonction (exemple : `15, 30, 40`).

2.4 Résolution du problème

Voici un programme répondant au problème :

```
#definition des fonctions
def carre(x,y,cote):
    up()
    goto(x,y)
    down()
    for i in range(4):
        forward(cote)
        right(90)
    up()

def carre_plein(x,y,cote):
    up()
    goto(x,y)
    down()
    begin_fill()
    for i in range(4):
        forward(cote)
        right(90)
    end_fill()
    up()

#trace de la face 5
carre(-50, 50, 100)
carre_plein(-40, 40, 20)
carre_plein(20, 40, 20)
carre_plein(-40, -20, 20)
carre_plein(20, -20, 20)
carre_plein(-10, 10, 20)
```

3 Renvoyer un résultat

Les fonctions que l'on a vu jusque maintenant ne permettent d'exécuter des blocs d'instructions. Cependant, elles ne permettent pas d'effectuer des calculs. En effet, on verra dans la partie suivante que les variables utilisée dans le corps d'une fonction sont en quelque sorte prisonnières de cette dernière et ne peuvent pas en sortir : on parle de variables locales.

Si l'on veut faire sortir un **résultat** d'une fonction, on utilise la commande `return` :

```
def nom_fonction(<arguments>)
  <corps de la fonction>
  return <resultat de la focntion>
```

Pour avoir accès à la valeur du résultat de la fonction, il suffit alors de l'appeler :

```
#afficher un resultat :
print(nom_fonction(<valeurs des arguments>))

#stocker un resultat :
nom_variable = nom_fonction(<valeurs des arguments>)
```

Voyons un exemple, on crée une fonction `somme` calculant la somme des $p + ... + n$ où p et n sont les arguments de la focntion et on affiche le résultat $10 + ... + 17$:

```
def somme(p,n):
  acc = 0
  for i in range(p, n+1)
    acc = acc+i
  return acc

print(somme(10, 17))
```

Remarque : tout comme les `input`, les `print` sont à voir comme des fonctions d'interface avec l'utilisateur. Aussi, sauf dans des fonctions dont le but est l'affichage de données, on préférera ne pas en inclure dans les corps des fonction. Si on veut cependant afficher un résultat on procédera plutôt comme dans l'exemple précédant : la fonction effectue et renvoie le résultat du calcul, résultat que l'on affiche en dehors de celle-ci.

Typage des focntions : on parle de typage des fonctions lorsque le langage de programmation force le programmeur à déclarer le type (`int`, `float`, `bool`, etc.) de chaque argument et le type du résultat à sa définition. Python ne fait pas de typage des fonctions mais permet à titre informatif de l'indiquer. Tout ce passe dans la première ligne de la définition :

```
def nom_fonction(<arg1>:<type arg1>, <arg2>:<type arg2>, ...) -> <type resultat> :
  <corps de la fonction>
  return <resultat de la focntion>
```

Il faut bien noter que ce typage n'est qu'indicatif et que rien n'oblige la fonction ni à être appelée sur des arguments effectifs du bon type ni même d'avoir un résultat du bon type.

On peut par exemple préciser dans la fonction `somme` les types impliqués :

```
def somme(p:int, n:int) -> int:
  acc = 0
  for i in range(p, n+1)
    acc = acc+i
  return acc
```

4 Portée des variables

Comme on l'a vu dans le chapitre $P1$, les variables stockent des valeurs dans la mémoire de la machine et permettent d'accéder à ces valeur dans le programme via leur nom. En fait l'histoire est plus complexe : lors de l'exécution d'un programme, chaque appel de fonction remplit ce qu'on appelle la **pile d'exécution** du programme. Il s'agît d'une pile d'états des différentes variables présentes dans le programme et, lorsqu'une fonction est exécutée, un nouvel état est créé ne comprenant a priori comme variable uniquement les arguments de la fonction initialisées aux valeurs données à l'appel de la fonction. Prenons le programme suivant en exemple :

```
1 def addition(a, b):
2     res = a+b
3     return res
4
5 nombre1 = 2
6 nombre2 = 7
7 s = somme(nombre1, nombre2)
8 print(s)
```

Après la ligne 6, l'état de l'interpréteur est le suivant :

— la variable `nombre1` contient l'entier 2 ;

— la variable `nombre2` contient la valeur 7.

Pour l'exécution de la fonction `addition` à la ligne 7, l'interprète se crée un nouvel état ne contenant initialement que deux variables `a` et `b` (nom donnés aux arguments de la fonction) ayant pour valeurs 2 et 7. L'état est donc le suivant :

— la variable `a` contient l'entier 2 ;

— la variable `b` contient la valeur 7.

Cet état est mis à jour avec la variable `res` :

— la variable `a` contient l'entier 2 ;

— la variable `b` contient la valeur 7 ;

— la variable `res` contient la valeur 9.

Enfin, à la ligne 7, l'interpréteur efface l'état interne à la fonction et revient au précédent, avec en plus la valeur retournée par `addition` dans la variable `s` :

— la variable `nombre1` contient l'entier 2 ;

— la variable `nombre2` contient la valeur 7.

— la variable `s` contient la valeur 9.

Comme on le voit dans cette description, les variables `a`, `b` et `res` n'existent que lors de l'exécution de la fonction `addition`, on dit que ce sont des variable **locales**. De même, toujours lors de l'exécution de la fonction `addition`, les variables `nombre1` et `nombre2` ne sont pas accessibles.