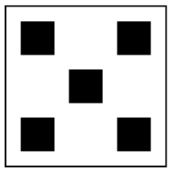
Afin de répéter ou d'adapter à différentes valeurs une portion de code ou simplement afin de rendre un programme plus lisible, on utilise les fonctions.

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 « dessiner 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 code réutilisables 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 la fonction soit exécutée, il faut l'appeler (ce qui peut 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 un 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(<valeur de 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 des programmeurs.

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 valeurs 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 vues jusque maintenant permettent d'exécuter des blocs d'instructions mais pas d'effectuer des calculs. En effet, on verra dans la partie suivante que les variables utilisées 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 fonctions. 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, en Python, 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 Portées des variables

4.1 Pile d'appel et variables locales

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 valeurs 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'environnements mémoire qui contiennent les variables accessibles à un instant donné. Plus précisément, lorsqu'une fonction est appelée, un nouvel état ne comprenant a priori comme variables uniquement les arguments de la fonction initialisées aux valeurs données est créé. 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'interpréteur est dans un premier état E1 donnant les informations suivantes :

— 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 E2 ne contenant initialement que deux variables `a` et `b` (noms 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.

E2 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, après la ligne 7, l'interpréteur efface l'état E2 car l'appel de la fonction est terminé puis revient dans l'état E1 en y ajoutant la valeur retournée par `addition` dans la variable `s` :

— la variable `nombre1` contient la valeur 2 ;

— la variable `nombre2` contient la valeur 7 ;

— la variable `s` contient la valeur 9.

On peut résumer ces étapes dans le tableau suivant :

Ligne	État	Variable	Valeur
5	E1	`nombre1`	2
6	E1	`nombre1` `nombre2`	2 7
1	E2	`a` `b`	2 7

2	E2	`a`
		`b`
		`res`
			2
			7
			9

7	E1	`nombre1`
		`nombre2`
		`s`
			2
			7
			9

Comme on le voit, les variables `a`, `b` et `res` n'existent que dans l'état mémoire E2, lors de l'exécution de la fonction `addition`. On dit que ce sont des variables **locales** à cette fonction. À l'inverse, les variables `nombre1` et `nombre2` n'existent que dans l'état E1 est ne sont donc pas accessibles lors de l'exécution de la fonction. Les portions de code dans lesquels une variable est accessible s'appellent **la portée de cette variable**.

4.2 Variables globales et effets de bord

Tester le programme suivant :

```
1 def incrementer(x)
2     x = x+1
3
4 a=7
5 incrementer(a)
6 print(a)
```

Le résultat est assez étonnant : en effet, la valeur de la variable `a` n'a pas été modifiée par la fonction `incrémenter` mais en décrivant l'exécution du programme, on comprend :

Ligne	État	Variable	Valeur
4	E1	`a`	7
1	E2	`x`	7
2	E2	`x`	8
5	E1	`a`	7

Lors de l'appel de la fonction `incrementer`, un nouvel état E2 de la pile est créé comprenant une variable `x` initialisée à 7 puis passant à 8. Cependant cet état mémoire est effacé en sortie de la fonction et on retourne dans E1 sans donc avoir modifié la valeur de `a`.

Si on veut pouvoir modifier une variable (pour les type de bases `int`, `float`, `bool`) à l'intérieur d'une fonction, deux remarques doivent être prises en compte :

— La variable ne peut pas être donnée en argument car l'argument effectif pourrait être une valeur. Or, si modifier une variable est concevable, modifier une valeur n'a aucun sens.

— Il faut demander à l'interpréteur d'ajouter cette variable à l'état mémoire de la fonction lors de son appel, c'est le but de la syntaxe `global`.

Pour qu'une variable existant en dehors d'une fonction puisse être modifiée par elle, celle-ci doit être déclarée **variable globale** dans la fonction :

```
1 def nom_fonction(<arguments>):
2     global <variable exterieure pouvant etre modifiee par la fonction>
3     <corp de la fonction>
4     return <resultat>
```

Attention : une variable servant d'argument ne peut pas être globale.

Reprenons notre fonction `incrementer` :

```
def incrementer()
    global x
    x = x+1

x=7
incrementer()
print(x)
```

Cette fois, la valeur de `x` a bien augmenté de 1. Examinons l'exécution de ce programme :

Ligne	État	Variable	Valeur
5	E1	x	7
1	E2	∅	∅
2	E2	x	7
3	E2	x	8
6	E1	x	8

Le fait que la fonction `incremente` modifie l'état mémoire dans lequel elle est appelée est appelé un **effet de bord** de la fonction. Il est fortement déconseillé d'en abuser car on peut rapidement perdre le contrôle de ces effets. On reparlera de ces effets dans le chapitre suivant sur les tableaux.

5 Sortie prématurée d'une fonction

Tester le programme suivant :

```
def temps_restant_avant_le_bac(annee):
    if annee<=2021:
        return 2021-annee
    print("felicitations (sans doute)")
    return 0

print(temps_restant_avant_le_bac(2020))
print(temps_restant_avant_le_bac(2022))
```

On voit ici que lorsque le test `annee<=2021` est positif, le reste de la fonction n'est pas exécuté alors qu'il est pas dans un `else`. Cela provient du fait que l'exécution d'une fonction s'arrête au premier `return` rencontré par l'interpréteur, que ce soit dans un branchement conditionnel ou dans une boucle :

```
def gamin():
    for i in range(1000):
        reponse = input("On est binetot arriveeee ? ")
        if reponse == "oui":
            return "Youpi !"
    return "zzzZzzzZZ"

print(gamin())
```

Parfois, utiliser une sortie prématurée de fonction peut en simplifier largement le code. Il vaut mieux cependant ne pas trop en abuser et s'en tenir, quand ce n'est pas beaucoup plus difficile à écrire à un seul `return` à la fin du corps de la fonction.

Aussi, il vaut mieux (même si cela ne pose pas de problème à l'interpréteur Python) s'arranger pour que quelque soit l'exécution de la fonction, un `return` ait toujours lieu.