CHAPITRE 6
PROCÉDURES ET FONCTIONS

Programmation en Python

Procédures et fonctions

- Une procédure consiste en une série d'instructions regroupées sous un même nom, mais qui ne retourne pas de valeur particulière.
  - La procédure effectue des instructions, mais ne retourne rien en particulier.
- Une fonction consiste en une série d'instructions regroupées sous un même nom et qui retourne une valeur particulière.
  - La fonction effectue des instructions et retourne une certaine valeur.

Procédures et fonctions

- En programmation, il arrive que des mêmes blocs d'instructions doivent être réutilisés plusieurs fois.
- Dans ce cas, il convient de regrouper ces blocs d'instructions sous un même nom, et d'appeler ensuite ce nom à chaque fois qu'on veut faire appel à ces instructions.
- Les procédures et fonctions consistent en de tels blocs d'instructions regroupés sous un même nom, afin de pouvoir être réutilisés.

Procédures et fonctions

- L'implémentation de fonctions et procédures de plus en plus complexes, elle-même utilisant d'autres fonctions et procédures, etc., est au cœur de tout processus de programmation.

Procédures et fonctions

- En Python, les syntaxes pour définir une procédure ou une fonction sont les suivantes :

```
def nom_procedure():
    bloc_instructions

def nom_fonction():
    bloc_instructions
    return valeur
```

Procédures et fonctions

Programme

```
def message():
    print "Ceci est un message imprimé automatiquement à
chaque appel de la procedure"


message()
message()
```

Exécution

```
Ceci est un message imprimé automatiquement à chaque appel
de la procedure
Ceci est un message imprimé automatiquement à chaque appel
de la procedure
```

Procédures et fonctions

Programme

```
def message():
    print "Ceci est un message imprimé automatiquement à
chaque appel de la procedure."


message() + "Ok"
# le résultat d'une procédure ne peut pas être utilisé
# comme une valeur, car il n'y a pas de valeur de retour
```

Exécution

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType'
and 'str'
```

Procédures et fonctions

Programme

```
def pi():
    return 3.14159

print pi()
print pi() + 2
# le résultat d'une fonction peut être utilisé comme
# une valeur, car c'est une valeur de retour
```

Exécution

```
3.14159
5.14159
```

Procédures et fonctions

- Il est possible de documenter les procédures et fonctions en ajoutant une chaîne de caractère qui décrit leur comportement entre triple guillemets. La syntaxe est la suivante :

```
def nom_procedure_ou_fonction():
    """description..."""
    bloc_instructions
```

Procédures et fonctions

Programme

```
def message():
    """Documentation de la fonction"""
    print "Petit message"

message()
print message.__doc__
```

Exécution

```
Petit message
Documentation de la fonction
```

Procédures et fonctions

Programme

```
def pi():
    """Cette fonction retourne pi"""
    return 3.14159

print pi()
print pi.__doc__
```

Exécution

```
3.14159
Cette fonction retourne pi
```

Procédures et fonctions

- Il est possible, et c'est là que ces concepts prendront toute leur puissance, de définir des procédures et fonctions qui possèdent des arguments. La syntaxe est :

```
def nom_procedure(args):
    bloc_instructions

def nom_fonction(args):
    bloc_instructions
    return valeur
```

Programme (faire démo)

- On avait écrit un programme qui permet de résoudre les équations du second degré.
- On va transformer ce programme en une procédure avec arguments qui imprime les solutions désirées.
- On pourra alors réutiliser cette procédure à souhait.

Programme (faire démo)


```
from math import * # importation de la librairie math

def resolution_eq(a,b,c):
    """Cette procédure résout les équations du 2ème degré
de paramètres a, b, c et imprime les solutions"""
    delta = b**2 - 4*a*c
    if delta < 0:
        print "Les solutions sont complexes:"
        aux = 0 + (sqrt(-delta))*1j
        print (-b + aux) / (2*a)
        print (-b - aux) / (2*a)
```

Programme (faire démo)


```
elif delta == 0:
    print "Une seule solution:"
    print -b / (2*a)
else:
    print "Deux solutions:"
    print (-b + sqrt(delta)) / (2*a)
    print (-b - sqrt(delta)) / (2*a)
```

Programme (faire démo)


```
resolution_eq(3,-1,4)
resolution_eq(1,4,4)
resolution_eq(4,7,-2)
```

Exécution

```
Les solutions sont complexes:
(0.16666666666666666+1.1426091000668406j)
(0.16666666666666666-1.1426091000668406j)
Une seule solution:
-2.0
Deux solutions:
0.25
-2.0
```

Programme (faire démo)

- On va transformer ce programme en une fonction avec arguments, et non une procédure, qui retourne (au lieu d'imprimer) les solutions désirées.
- On pourra ensuite utiliser cette fonction à souhait dans d'autres programmes.

Programme (faire démo)


```
from math import * # importation de la librairie math

def solution_eq(a,b,c):
    """Cette fonction résout les équations du 2ème degré de
paramètres a, b, c et retourne les solutions"""
    delta = b**2 - 4*a*c
    if delta < 0:
        aux = 0 + (sqrt(-delta))*1j
        return ((-b + aux) / (2*a), (-b - aux) / (2*a))
```

Programme (faire démo)


```
elif delta == 0:
    return (-b / (2*a))
else:
    return ((-b + sqrt(delta)) / (2*a), \
            (-b - sqrt(delta)) / (2*a))
```

Programme (faire démo)


```
s = solution_eq(3,7,4)
print s[0]
print s[1]
```

Exécution

```
-1.0
-1.3333333333333333
```

Procédures et fonctions

- Lorsque l'on déclare une fonction ou une procédure, il est possible de donner des noms et des valeurs par défaut aux arguments.
- Ainsi, il n'y plus besoin de donner les arguments dans le bon ordre, pour autant qu'on les nomme par leur nom correctement.
- Si on omet de spécifier la valeur d'un argument, c'est la valeur par défaut qui sera considérée.

Procédures et fonctions

- La syntaxe est la suivante :

```
def nom_procedure(arg_1=v_1,...,arg_n=v_n):
     bloc_instructions

def nom_fonction(arg_1=v_1,...,arg_n=v_n):
     bloc_instructions
     return valeur
```

Programme (faire démo)

- On avait écrit un programme qui générait le cours aléatoire d'une action sur 1000 pas de temps à partir d'une certaine valeur et en fluctuant de +/- 2 à chaque pas de temps. On avait ensuite implémenté une stratégie d'achat et de vente en fonction du cours de l'action.
- On va transformer ce programme en une procédure où les arguments sont nommés et possèdent des valeurs par défaut.

Programme (faire démo)


```
from math import * # importation de la librairie math
from random import * # importation de la librairie random

# on génère le cours d'une action de 1000 valeurs
# le cours varie de +/- 2 à chaque pas de temps

def procedure_action(temps=1000, v_init=80,
                     v_achat=75, v_vente=85):
    """procedure qui implémente de cous aléatoire d'une
action ainsi qu'une stratégie d'achat et de vente de cette
action"""
```

Programme (faire démo)


```
# on initialise le 1er élément de la liste
cours = [v_init]
# on crée les 999 éléments restants de la liste
for i in range(temps-1):
    cours.append( cours[-1] + randint(-2,2) )
```

Programme (faire démo)


```
# on implémente une stratégie d'achat et de de vente
# on suppose qu'on a acheté les actions au départ
ordre = "achat"
for i in range(temps-1):
    # si l'action monte en-dessus de 85, on vend
    # (pour autant qu'on avait acheté auparavant)
    if (cours[i] >= v_vente and ordre == "achat"):
        print "Vendre au temps", i
        ordre = "vente"
    # si l'action baisse en-dessous de 75, on achète
    # (pour autant qu'on avait vendu auparavant)
    elif (cours[i] <= v_achat and ordre == "vente"):
        print "Acheter au temps", i
        ordre = "achat"
```

Programme (faire démo)


```
procedure_action()
print ""
procedure_action(v_achat = 73, v_vente = 88)
print ""
procedure_action(temps = 2000)
```

Programme (faire démo)

Exécution

```
Vendre au temps 25
Acheter au temps 49
Vendre au temps 422
Acheter au temps 739

Vendre au temps 152
Acheter au temps 199
Vendre au temps 531

Vendre au temps 83
Acheter au temps 149
Vendre au temps 405
```

Traitement des arguments

- Le traitement des arguments avec valeurs par défaut peut être très subtil, suivant que le type de l'argument est modifiable ou non modifiable.
- Pour les arguments de types non modifiables, la valeur par défaut donnée lors de la déclaration de la fonction ne sera jamais modifiée.
- Pour les arguments de types modifiables, la valeur par défaut donnée lors de la déclaration de la fonction peut être modifiée dans le corps de la fonction elle-même.

Traitement des arguments

- Dans tous les cas, les arguments par défaut sont des variables locales et toute réaffectation globale de ces arguments n'aura jamais aucune répercussion locale.

Traitement des arguments


```
>>> # Exemple : argument de type non modifiable
>>> def ma_fonction(n = 1):
    # n prend la valeur locale par défaut 1
        n = n + 1 # n est apparemment localement modifié
        return n

>>> ma_fonction(n = 5)
# n prend la valeur locale 5
6
>>> ma_fonction()
# n possède toujours la valeur locale par défaut 1
2
```

Traitement des arguments


```
>>> ma_fonction(n = 3)
# n prend la valeur locale 3
4
>>> ma_fonction()
# n possède toujours la valeur locale par défaut 1
2
>>> n = 5
# n est modifié globalement, extérieurement à la fct
>>> ma_fonction()
# n possède toujours la valeur locale par défaut 1
2
>>> print n # valeur globale de n
5
```

Traitement des arguments


```
>>> # Exemple : argument de type modifiable
>>> def ma_fonction(l = [1,2,3]):
    # l prend la valeur locale par défaut [1,2,3]
        l.append("a") # l modifiée localement en [1,2,3,"a"]
        return l

>>> ma_fonction()
# l possède la valeur locale par défaut [1,2,3]
[1,2,3,"a"]
>>> ma_fonction()
# l possède la valeur locale par défaut [1,2,3,"a"]
[1,2,3,"a","a"]
```

Traitement des arguments


```
>>> ma_fonction(l = [1,2,3,4])
# l prend la valeur locale [1,2,3,4]
[1,2,3,4,"a"]
>>> l = [1,1,1]
# l est modifiée globalement, extérieurement à la fct
>>> ma_fonction()
# l possède la valeur locale par défaut [1,2,3,"a","a"]
[1,2,3,"a","a","a"]
>>> print l # valeur globale de l
[1,1,1]
```

Variables locales et globales

- Le traitement des variables locales et globales est également subtil.
- Une variable locale est une variable définie dans le corps d'une procédure ou fonction.
- Une variable globale est une variable définie en dehors du corps de toute procédure ou fonction.
- Une variable peut donc être à la fois locale et globale, ou, plus précisément, posséder à la fois des déclarations locales et d'autres globales.

Variables locales et globales

- Plus précisément, il se peut qu'une procédure ou fonction fasse appel localement à une variable qui a déjà été définie globalement (i.e., en dehors de la procédure ou fonction elle-même).
- Si la variable externe est de type non modifiable, alors sa valeur globale ne sera jamais modifiée.
- Si la variable externe est de type modifiable, alors sa valeur globale peut être modifiée.

Variables locales et globales


```
>>> # variables locales et globales
>>> n = 2                  # n de type non modifiable (int)
>>> l1 = [1,2,3]           # l1 de type modifiable (list)
>>> l2 = ["a","b","c”] # l2 de type modifiable (list)

>>> def ma_fonction():
        n = 5               # réaffectation locale de n
        l1.append(4)        # modification locale de l1
        l2 = [0,0,0]        # réaffectation locale de l2
        print n
        print l1
        print l2
```

Variables locales et globales


```
>>> ma_fonction()
5
[1, 2, 3, 4]
[0, 0, 0]
>>> print n  # n n'est pas modifié globalement
2
>>> print l1 # l1 est modifié globalement
[1, 2, 3, 4]
>>> print l2 # l2 n'est pas modifié globalement
['a', 'b', 'c']
# car une réaffectation locale n'est pas compris comme une
modification (subtil)...
```

Variables locales et globales

- Dans le cas d'une variable de type non modifiable, on a vu qu'une réaffectation ou modification locale n'affectait pas la valeur globale (contrairement aux variables de types modifiables).
- Si on désire que la réaffectation locale se répercute globalement, on utilise la syntaxe `global var`.

Variables locales et globales


```
>>> n1 = 1 # n1 de type non modifiable
>>> n2 = 2 # n2 de type non modifiable
>>> def ma_fonction():
        n1 = 5      # réaffectation locale de n1
        global n2
        n2 = 5      # réaffectation globale de n2
        return(n1,n2)

>>> print ma_fonction()
(5,5)
>>> print n1 # valeur globale de n1 non modifiée
1
>>> print n2 # par contre, valeur globale de n2 modifiée!
5
```

Importation de modules

- Le grand avantage de définir des procédures et des fonctions est de pouvoir les réutiliser à souhait.
- Pour cela, il convient d'écrire nos fonctions dans des fichiers séparés, appelés « scripts » ou « modules », puis d'importer les fonctions de ces fichiers lorsque vous en avez besoin.
- On est là au cœur de toute pratique de programmation. Les programmes complexes sont toujours constitués de plusieurs modules qui dépendent les uns des autres, et s'appellent entre eux...

Importation de modules

- Pour appeler les procédures ou fonctions f, g, h, etc. d'un fichier mon_fichier.py, on utilise l'instruction :

```
from mon_fichier import f, g, h
```

- Si l'on désire importer toutes les procédures et fonctions du module mon_fichier.py d'un coup, on utilise une des deux instructions :

```
import mon_fichier
from mon_fichier import *
```

Programme (faire démo)

- Au cours de ce chapitre, on a implémenté trois procédures et fonctions permettant de résoudre les équations du second degré ou de générer le cours aléatoire d'une action.
- Regroupons ce code dans un fichier nommé Ch6_ex4.py.
- On peut alors importer ce module et disposer de nos fonctions à souhait.

Programme (faire démo)


```
>>> from Ch6_ex4 import solution_eq
>>> solution_eq(1,3,2)
(-1.0, -2.0)
>>> from Ch6_ex4 import *
>>> resolution_eq(1,3,2)
Deux solutions:
-1.0
-2.0
>>> procedure_action()
Vendre au temps 6
Acheter au temps 92
Vendre au temps 285
Acheter au temps 297
Vendre au temps 375
Acheter au temps 386
```