Définition

- Une fonction permet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés en fragments plus petits etc.
- Une fonction est réutilisable
- Une fonction apparaît sous la forme d'un nom associé à des parenthèses : print()
- Dans les parenthèses, on transmet à la fonction un ou plusieurs arguments
- La fonction fournit une valeur de retour

Fonctions prédéfinies

print()

input ()

Révision

Contrôle du flux : utilisation d'une liste simple

```
# Utilisation d'une liste et de branchements conditionnels

print("Ce script recherche le plus grand de trois nombres")
print("Veuillez entrer trois nombres séparés par des virgules : ")
ch =input()
# Note : l'association des fonctions eval() et list() permet de convertir
# en liste toute chaîne de valeurs séparées par des virgules
nn = list(eval(ch))
max, index = nn[0], 'premier'
if nn[1] > max:                          # ne pas omettre le double point !
    max = nn[1]
    index = 'second'
if nn[2] > max:
    max = nn[2]
    index = 'troisième'
print("Le plus grand de ces nombres est", max)
print("Ce nombre est le", index, "de votre liste.")
```

Boucle *while* – instructions imbriquées

```
 1#  # Instructions composées <while> - <if> - <elif> - <else>
 2#
 3#  print('Choisissez un nombre de 1 à 3 (ou 0 pour terminer) ', end=' ')
 4#  ch = input()
 5#  a = int(ch)              # conversion de la chaîne entrée en entier
 6#  while a:                 # équivalent à : < while a != 0: >
 7#      if a == 1:
 8#          print("Vous avez choisi un :")
 9#          print("le premier, l'unique, l'unité ..."
10#      elif a == 2:
11#          print("Vous préférez le deux :")
12#          print("la paire, le couple, le duo ...")
13#      elif a == 3:
14#          print("Vous optez pour le plus grand des trois :")
15#          print("le trio, la trinité, le triplet ...")
16#      else :
17#          print("Un nombre entre UN et TROIS, s.v.p.")
18#      print('Choisissez un nombre de 1 à 3 (ou 0 pour terminer) ', end =' ')
19#      a = int(input())
20#  print("Vous avez entré zéro :")
21#  print("L'exercice est donc terminé.")
```

Fonctions originales

Définir une fonction

```
def nomDeLaFonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
```

- Nom
  - N'importe quel nom sauf des mots réservés du langage, aucun caractère spécial (sauf _)
  - Conseillée : tout en minuscules (maj. Réservé aux classes)
  - Nom très explicite
- Paramètres
  - la liste de paramètres spécifie quelles informations il faut fournir en guise d'arguments
  - Les parenthèses restent vides si la fonction ne nécessite pas d'arguments
- Appel de fonction
  - Nom de la fonction suivi de parenthèses vides ou avec les arguments

Fonction simple sans paramètres

```
>>> def table7():
...     n = 1
...     while n <11 :
...         print(n * 7, end =' ')
...         n = n +1
...
```

```
>>> table7()
```

provoque l'affichage de :

```
7 14 21 28 35 42 49 56 63 70
```

Répéter la réutilisation de la fonction

```
>>> def table7triple():
...     print('La table par 7 en triple exemplaire :')
...     table7()
...     table7()
...     table7()
...
```

```
>>> table7triple()
```

l'affichage résultant devrait être :

```
La table par 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Fonction avec paramètre

- Si l'on veut afficher maintenant une table de multiplication par 9 et après par 13

$\Rightarrow$Définir une fonction qui affiche n'importe quelle table de multiplication à la demande

$\Rightarrow$Si l'on appelle cette nouvelle fonction, on doit indiquer quelle table on veut afficher

$\Rightarrow$ Cette information s'appelle un **argument**

$\Rightarrow$Il faut prévoir une variable particulière pour recevoir l'argument transmis, elle s'appelle un **paramètre**

```
>>> def table(base):
...     n = 1
...     while n <11 :
...         print(n * base, end =' ')
...         n = n +1
```

argument

- Pour tester cette nouvelle fonction, on l'appelle avec un argument

```
>>> table(13)
13 26 39 52 65 78 91 104 117 130

>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

Utilisation d'une variable comme argument

- L'argument de la fonction précédente est toujours une constante (9, 13, etc.), mais il peut être une variable

```
>>> a = 1
>>> while a <20:
...     table(a)
...     a = a +1
...
```

Fonction avec plusieurs paramètres

```
>>> def tableMulti(base, debut, fin):
...     print('Fragment de la table de multiplication par', base, ':')
...     n = debut
...     while n <= fin :
...         print(n, 'x', base, '=', n * base)
...         n = n +1
```

```
>>> tableMulti(8, 13, 17)
```

ce qui devrait provoquer l'affichage de :

```
Fragment de la table de multiplication par 8 :
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Variables locales / globales

- Variable locales
  - Ne sont accessibles qu'à la fonction elle-même
  - Le cas des variables *base, debut, fin*
  - Chaque fois que la fonction *tableMulti()* est appelé, Python réserve pour elle un nouvel espace de noms qui est inaccessible depuis l'extérieur de la fonction

```
>>> print(base)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'base' is not defined
```

Variables locales / globales

- Variable globales
  - Sont définies à l'extérieur d'une fonction
  - Leur contenu est « visible » de l'intérieur, mais la fonction ne peut pas le modifier

```
>>> def mask():
...     p = 20
...     print(p, q)
...
>>> p, q = 15, 38
>>> mask()
20 38
>>> print(p, q)
15 38
```

Modifier une variable globale

- L'instruction **global** : permet d'indiquer à l'intérieur de la définition d'une fonction quelles sont les variables à traiter globalement

```
>>> def monter():
...     global a
...     a = a+1
...     print(a)
...
>>> a = 15
>>> monter()
16
>>> monter()
17
>>>
```

Vraies fonctions

- Une vraie fonction doit renvoyer une valeur
- Modifier la fonction **table ()** pour qu'elle renvoie une valeur

```
>>> def table(base):
...     resultat = []              # resultat est d'abord une liste vide
...     n = 1
...     while n < 11:
...         b = n * base
...         resultat.append(b)     # ajout d'un terme à la liste
...         n = n +1
...     return resultat
...
```

```
>>> ta9 = table(9)
```

```
>>> print(ta9)
[9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
>>> print(ta9[0])
9
>>> print(ta9[3])
36
>>> print(ta9[2:5])
[27, 36, 45]
>>>
```

Vraies fonctions

```
>>> def table(base):
...     resultat = []                # resultat est d'abord une liste vide
...     n = 1
...     while n < 11:
...         b = n * base
...         resultat.append(b)       # ajout d'un terme à la liste
...         n = n +1
...     return resultat
...
```

- Une méthode est une fonction qui est associée à un objet
- Sous Python, les listes constituent une classe particulière d'objets

Utilisation des fonctions dans un script

```
def cube(n):
   return n**3

def volumeSphere(r):
   return 4 * 3.1416 * cube(r) / 3

r = input('Entrez la valeur du rayon : ')
print('Le volume de cette sphère vaut', volumeSphere(float(r)))
```

Utilisation des fonctions dans un script

- **Dans un script, la définition des fonctions doit précéder leur utilisation**

Intervertissez cet ordre (en plaçant par exemple le corps principal du programme au début).

Utilisation des fonctions dans un script : _ _main_ _

- Le corps principal d'un programme Python constitue lui-même une entité toujours reconnue dans le fonctionnement interne de l'interpréteur sous le nom réservé **__main__**
- L'exécution d'un script commence toujours avec la première instruction de cette entité **__main__**, ou qu'elle puisse se trouver dans le listing.
- Au lieu de passer à l'instruction suivante, l'interpréteur exécute la fonction appelée puis revient au programme appelant pour continuer le travail interrompu
- Pour que ce mécanisme puisse fonctionner, il faut que l'interpréteur ait pu lire la définition de la fonction avant l'entité **_ _main_ _**, et celle-ci sera donc placée en général à la fin du script.

Utilisation des commentaires dans une fonction

- Une chaine de car. ne joue aucun rôle fonctionnel dans le script : elle est traitée par Python comme un simple commentaire
- Python place cette chaine dans une variable spéciale dont le nom est __doc__ et qui est associée à l'objet fonction comme étant l'un de ses attributs

```
>>> def essai():
...     "Cette fonction est bien documentée mais ne fait presque rien."
...     print("rien à signaler")
...
>>> essai()
rien à signaler
>>> print(essai.__doc__)
Cette fonction est bien documentée mais ne fait presque rien.
```

Typage des paramètres

- Le typage des variables est un typage dynamique : le type d'une variable est défini au moment où on lui affecte une valeur, idem pour les paramètres d'une fonction

```
>>> def afficher3fois(arg):
...     print(arg, arg, arg)
...

>>> afficher3fois(5)
5 5 5

>>> afficher3fois('zut')
zut zut zut

>>> afficher3fois([5, 7])
[5, 7] [5, 7] [5, 7]

>>> afficher3fois(6**2)
36 36 36
```

Valeurs par défaut pour les paramètres

- Définir un argument par défaut pour chacun des paramètres permet d'obtenir une fonction qui peut être appelée avec une partie seulement des arguments attendus

```
>>> def politesse(nom, vedette ='Monsieur'):
...     print("Veuillez agréer ,", vedette, nom, ", mes salutations cordiales.")
...

>>> politesse('Dupont')
Veuillez agréer , Monsieur Dupont , mes salutations cordiales.

>>> politesse('Durand', 'Mademoiselle')
Veuillez agréer , Mademoiselle Durand , mes salutations cordiales.
```

- les paramètres sans valeur par défaut doivent précéder les autres dans la liste.

- Cette définition est incorrecte

```
>>> def politesse(vedette ='Monsieur', nom):
```

Arguments avec étiquettes

- Les arguments que l'on fournit lors de l'appel d'une fonction doivent être fournis exactement dans le même ordre que celui des paramètres qui leur correspondent dans la définition de la fonction
- Si les paramètres annoncés dans la définition de la fonction ont reçu chacun une valeur par défaut, on peut faire appel à la fonction en fournissant les arguments correspondants dans n'importe quel ordre

```
>>> def oiseau(voltage=100, etat='allumé', action='danser la java'):
...     print('Ce perroquet ne pourra pas', action)
...     print('si vous le branchez sur', voltage, 'volts !')
...     print("L'auteur de ceci est complètement", etat)
...

>>> oiseau(etat='givré', voltage=250, action='vous approuver')
Ce perroquet ne pourra pas vous approuver
si vous le branchez sur 250 volts !
L'auteur de ceci est complètement givré

>>> oiseau()
Ce perroquet ne pourra pas danser la java
si vous le branchez sur 100 volts !
L'auteur de ceci est complètement allumé
```

Signature d’une fonction

- La signature d’une fonction en Python est son nom
- On ne peut donc pas définir deux fonctions du même nom (dans le cas inverse : l’ancienne définition est écrasée par la nouvelle)

Module de fonctions

- Module : un bout de code que l'on a enfermé dans un fichier
- Il est conseillé de placer fréquemment les définitions de fonctions dans un module Python, et le programme qui les utilise dans un autre.
- On peut l'utiliser dans n'importe quel autre script
- Si l'on veut travailler avec les fonctionnalités prévues par le module et utiliser ensuite toutes les fonctions et variables prévues, on importe le module

Importer un module de fonctions

- Les **modules** sont des fichiers qui regroupent des ensembles de fonctions
- Module *math* : fonctions mathématiques *sinus, cosinus, racine carrée, etc.*

```
>>> import math
>>> math.sqrt(16)
4
```

Module de fonction

- Aide pour avoir des informations sur le module :

```
>>> help("math")

>>> help("math.sqrt")
```

Importer un module de fonctions

- Autre méthode d'importation

from math import …

Utile si l'on veut une/des fonctions particulières du module :

```
>>>from math import fabs   #fonction renvoyant la valeur absolue d'une  var.
>>>fabs(-5)
5
>>>fabs(2)
2
```

Si l'on veut toutes les fonctions :

from math import *

S’amuser un peu : module *turtle*

- Ecrire des scripts qui réalisent des dessins suivant un modèle imposé à l’avance.
- Les principales fonctions :

reset()	On efface tout et on recommence
goto(x, y)	Aller à l’endroit de coordonnées x, y
forward(distance)	Avancer d’une distance donnée
backward(distance)	Reculer
up()	Relever le crayon (pour pouvoir avancer sans dessiner)
down()	Abaisser le crayon (pour recommencer à dessiner)
color(couleur)	*couleur* peut être une chaîne prédéfinie (’red’, ’blue’, etc.)
left(angle)	Tourner à gauche d’un angle donné (exprimé en degrés)
right(angle)	Tourner à droite
width(épaisseur)	Choisir l’épaisseur du tracé
fill(1)	Remplir un contour fermé à l’aide de la couleur sélectionnée
write(texte)	*texte* doit être une chaîne de caractères

- Essayez :

```
>>> from turtle import *
>>> forward(120)
>>> left(90)
>>> color('red')
>>> forward(80)
```

L'exercice est évidemment plus riche si l'on utilise des boucles :

```
>>> reset()
>>> a = 0
>>> while a <12:
        a = a +1
        forward(150)
        left(150)
```