FONCTIONS

I) Fonctions

Notion de fonction

Lorsque l'on veut automatiser une suite d'instructions dans certaines situations ou lorsque l'on veut ajouter de nouvelles fonctions (au sens mathématique) à l'environnement utilisé, on peut créer des fonctions (ou procédures dans certains langages).

Contrairement aux fonctions en mathématiques qui prennent comme arguments un ou plusieurs nombres (ou des éléments d'un certain ensemble) et retournent un ou plusieurs nombres (ou des éléments d'un autre ensemble), les fonctions informatiques peuvent retourner des objets de différents types : des nombres, des affichages, des graphes, rien….

Elles peuvent également combiner certains de ces résulats : rendre un résultat et faire un changement dans une variable ou une fenêtre de graphique par exemple.

Dans Python, la syntaxe de définition d'une fonction utilise le mot clé **def** et s'écrit

def nom_de_la_fonction (*arguments*) :
corps de la procédure

Evidemment les : et l'indentation sont essentiels…

Par exemple, la fonction **affiche_carré_cube** écrite ci-dessous, va prendre un nombre qui lui sera donné et va afficher la phrase : "le carré du nombre … donné est … et son cube est …" où les … remplacent les valeurs correspondantes

```
def affiche_carré_cube(x) :
    a = x * x
    b = a * x
    print("le carré du nombre ", x ," donné est ", a , " et son cube est ", b)
```

```
>>> affiche_carré_cube(10)
le carré du nombre  10  donné est  100  et son cube est  1000
>>> affiche_carré_cube(1+3j)
le carré du nombre  (1+3j)  donné est  (-8+6j)  et son cube est  (-26-18j)
>>>
```

Retour de valeur

Dans la fonction précédente, on calcule le carré et le cube du nombre donné en argument mais ces valeurs sont juste affichées et ne sont plus récupérables par Python à l'extérieur de la définition de la fonction. Ce serait d'ailleurs la même chose si, au lieu de l'instruction : **print**("le carré du nombre ", x ," donné est ", a , " et son cube est ", b) on avait mis l'instruction : **print**(a)

Pour que Python comprenne que l'on veut effectivement utiliser le ou les résultats de la procédure, il faut utiliser le mot clé **return.**

Par exemple, la fonction **calcule_carré_cube** écrite ci-dessous, va prendre un nombre qui lui sera donné et va prendre pour valeur le tuple constitué du carré et du cube de la valeur entrée en argument.

```
def calcule_carré_cube(x) :
    a = x * x
    b = a * x
    return( a , b)
```

```
>>> calcule_carré_cube(1.2)
(1.44, 1.728)
>>> t = affiche_carré_cube(1.2)
le carré du nombre  1.2  donné est  1.44  et son cube est  1.728
>>> t[1]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    t[1]
TypeError: 'NoneType' object is not subscriptable
>>> t = calcule_carré_cube(1.2)
>>> t[1]
1.728
```

Il faut bien distinguer l'affichage et l'affectation d'un résultat

Par exemple, si on définit la fonction **fexemple** ainsi

```
def fexemple(x) :
    a = x * x
    b = a * x
    print (a)
    return( b)
```

et que l'on effectue l'instruction >>> t = fexemple(2) , l'affichage réponse de Python est 4 alors que t est affectée à la valeur 8

II) Portée des variables

Variables locales et globales

Dans le corps d'une fonction, on peut utiliser des noms de variables utilisés ou non ailleurs dans le corps du programme. Une variable locale, est une variable qui n'est connue ou changée qu'à l'intérieur de la fonction alors qu'une variable globale est une variable existant avant l'appel à la fonction et après cet appel et qui est éventuellement modifiée par l'appel à cette fonction.

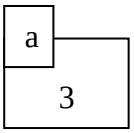
D'une façon générale, toutes les variables qui subiront une affectation dans le corps de la procédure seront considérées come locale, et les autres variables seront considérées comme globales (et en fait seront considérées comme des paramètres de la fonction au même titre que ses arguments).

Voyons quelques exemples illustrant ces différences

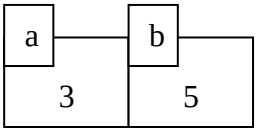
```
>>> a = 3
>>> def f1():
        b = a + 2
        return(a,b)

>>> f1()
(3, 5)
>>> a
3
>>> b
Traceback (most recent call last):
  File "<pyshell#118>", line 1, in <modul
    b
NameError: name 'b' is not defined
```

Ici, la fonction f1 utilise la variable locale b et la variable globale a
Au début de l'appel de la fonction l'état est



Puis, durant l'appel à f1(), on passe à un état intermédiaire



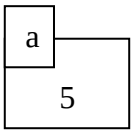
Et enfin, la variable b est effacée avant que Python ne donne le résultat de f1()

Un appel à f1() avant l'initialisation de a, aurait conduit à un message d'erreur

```
>>> a = 3
>>> def f1():
        b = a + 2
        return(a,b)

>>> a = 5
>>> f1()
(5, 7)
```

Ici, la fonction f1 utilise la variable locale b et la variable globale a
Au début de l'appel de la fonction l'état est

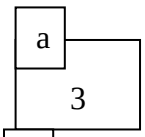


C'est bien l'état de la variable au moment de l'appel à la fonction f1 qui compte et non l'état de la variable au moment où on l'a écrite

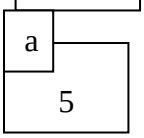
```
>>> a= 3
>>> def f2():
        a = 5
        return(a)

>>> f2()
5
>>> a
3
>>>
```

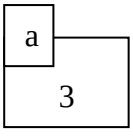
Ici, la fonction f2 utilise la variable locale a
Au début de l'appel de la fonction l'état est



Puis, durant l'appel à f2(), on passe à un état intermédiaire



Et enfin, la variable locale a est effacée avant que Python ne donne le résultat de f2() et on revient à l'état précédant l'appel à f2()




```
def f3() :
    b = a + 2
    a = 5
    return (a,b)
```

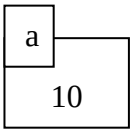
Tout appel à f3() conduit à un message d'erreur car Python repère a comme étant locale (elle est au moins une fois à gauche de =).. mais elle est utilisée une fois avant d'avoir été initialisée.

Si on veut interdire à Python de repérer une variable comme locale, on utilise le mot clé **global**

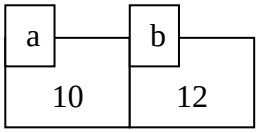
```
def f4() :
    global a
    b = a + 2
    a = 5
    return (a,b)

>>> a = 10
>>> f4()
(5, 12)
>>> f4()
(5, 7)
```

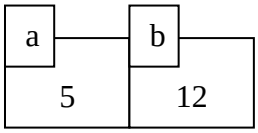
Avant le premier appel à f4(), on est dans l'état



Durant ce premier appel, on passe par le premier état intermédiaire (après la première affectation b = a + 2)

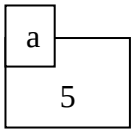


puis par l'état



après l'instruction a = 5

A la fin du premier appel, la variable b étant locale, est desaffectée et l'état devient
état servant alors de départ pour le second appel à la fonction f4()



Comme vous pouvez le constater l'appel à une fonction contenant des variables globales engendre des résultats différents selon l'état des variables au moment de l'appel à cette fonction. Il vaudra donc mieux éviter d'utiliser des variables globales dans une fonction car l'application de cette fonction dépendra de l'état des variables. En fait on n'utilisera ces variables globales que pour des objets importants comme des tableaux ou des graphes que l'on voudra modifier par l'appel de la fonction…

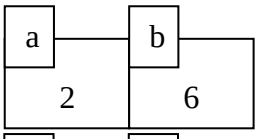
Arguments

Pour une fonction définie avec des arguments, les noms de ces différents arguments sont considérés comme des variables locales initialisées aux valeurs des arguments passés.

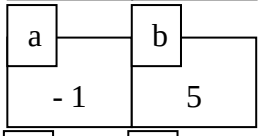
```
def f5(a, b) :
    c = b + 2
    b = a * c
    return (b)

>>> a, b = 2, 6
>>> f5(-1, 5)
- 7
>>> a, b
(2, 6)
```

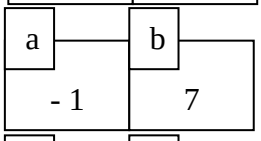
Avant le premier appel à f5, on est dans l'état



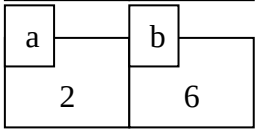
Au début de l'appel à f5(-1,5), Python affecte provisoirement les variables dans l'état intermédiaire :



A la fin du calcul, avant le retour du résultat de f5, on arrive à l'état



Puis à la sortie de la fonction f5, on revient à l'état global initial



III) Manipulations avancées

Fonctions anonymes

Pour les fonctions simples (celles qui à un nombre associent un autre nombre), Python possède une syntaxe particulière utilisant le mot clé **lambda**

Par exemple : **lambda** x: x*x désigne la fonction qui à x associe x^2

Valeurs par défaut pour les paramètres

Il est possible de donner des valeurs par défaut aux arguments d'une fonction. Dans ce cas les paramètres sans valeur par défaut doivent écrits avant les autres et ceux avec une valeur par défaut sont optionnels.

Par exemple, en supposant toutes les fonctions du module math chargées, si on considère la fonction suivante :

```
def nbre_de_chiffres(n, base = 10):
    return ( floor(log(n)/log(base)) + 1)
```

nbre_de_chiffres(36) donne 2 nbre_de_chiffres(36, 2) donne 6

Arguments avec étiquettes

Lorsque tous les paramètres d'une fonction ont une valeur par défaut, on peut à l'appel de cette fonction, entrer les arguments de la fonction dans l'ordre que l'on veut mais en associant les paramètres à leurs valeurs.

Par exemple, la fonction suivante permet de calculer le n-ième terme de la suite de premier terme u0 et définie par la relation de récurrence $u_{n+1} = f(u_n)$

```
def suite(f = lambda x: x, u0 = 0, n = 10) :
    u = u0
    for k in range(n):   u = f(u)
    return (u)
```

L'appel suite(u0 = 1.1 , f = lambda x: x**2) donnera le 10-ième terme de la suite définie par $u_0 = 1.1$ et pour tout n $u_{n+1} = (u_n)^2$

Documenter une fonction

Lorsque l'on écrit une fonction, et surtout si on a plusieurs fonctions à écrire pour un projet, il est conseillé d'ajouter des commentaires (après le symbole #) pour expliquer ce qui se passe voire, mieux, d'écrire une aide pour la fonction (encadrée par """ """).

```
def suite(f = lambda x: x, u0 = 0, n = 10) :
    """ calcule le nième terme de la suite définie par u0 et la fonction f """
   u = u0    # on initialise la variable à u0
   for k in range(n): u = f(u)    # on effectue n fois l'affectation u = f(u)
   return (u)   # on a récupéré dans la variable u le terme u(n) de la suite
```

Un appel à **help**(suite) donne la syntaxe de la fonction suite et reprend la phrase écrite entre """ """

Fonctions d'une bibliothèque

La plupart des logiciels de programmation possède de nombreuses fonctions qu'il n'est pas utile de charger complétement pour ne pas saturer la mémoire vive de l'ordinateur. Ces fonctions sont regroupées dans certains modules (ou bibliothèques) que l'on charge au besoin. Nous avons déjà rencontré les modules **math**, **cmath** par exemple et on en utilisera encore d'autres **numpy**, **matplotlib**, **random**, **time**, **tkinter**, … Pour charger les fonctions on a plusieurs options, chacune ayant des avantages et des inconvénients.

syntaxe	effet	avantages inconvénients
import module	charge toutes les fonctions du module	les noms des fonctions sont composés. Par exemple si on charge le module math, la fonction cosinus est appelée avec la syntaxe : math.cos(x)
import module as alias	charge toutes les fonctions et les renomme en alias.nom	Même remarque mais les noms sont un peu moins longs … il suffit de prendre le bon alias..
from module import fonction1, fonction2	Ne charge que les fonctions désignées	L'appel d'une fonction chargée ne nécessite pas le préfixe module. Par contre il peut y avoir concurrence entre deux fonctions de même nom dans des modules différents. cf exemple ci -dessous
from module import *	Charge toutes les fonctions du module	

Le fait de charger toutes les fonctions en enlevant le préfixe, semble interessant mais peut engendrer des soucis lorsque 2 fonctions de même nom sont dans deux modules différents.
Par exemple la fonction cos existe dans les modules math et cmath.
Un appel à math.cos(0) donne 1.0 alors qu'un appel à cmath.cos(0) donne (1 – 0j)

IV) Exercices

1) Ecrire une fonction **moyenne** prenant comme argument une liste L d'entiers ou de flottants et qui donne la moyenne de ces nombres

2) Ecrire une fonction **f** prenant pour argument un couple (i,j) et qui donne i si i < j, j si j < i et 3 si i = j

3) Ecrire une fonction **plusgrand** prenant comme argument une liste L d'entiers ou de flottants et qui donne le plus grand de ces nombres

4) *Algorithme de Syracuse.* L'algorithme de Syracuse est le suivant : étant donné un entier n, on remplace n par n/2 si n est pair et par 3n+1 si n est impair. On réitère alors le procédé jusqu'à obtenir 1 (On ne sait toujours pas si on arrive toujours à 1, mais on n'a pas non plus trouvé de contre-exemple). Par exemple si n = 5, on obtient succesivement : 5, 16, 8, 4, 2, 1.
   - **a)** Ecrire une fonction **syracuse** recevant comme argument un entier n et qui détermine la liste des termes successifs jusqu'à l'obtention du terme 1.
     Par exemple : syracuse(5) doit rendre [5, 16, 8, 4, 2, 1]
   - **b)** Ecrire une procédure **nombresyracuse** recevant comme argument un entier n et qui détermine le nombre d'itérations pour obtenir 1

5) *Triangle pseudo-isocèle.* On appelle triangle rectangle pseudo-isocèle (TRPI) tout triangle rectangle dont les cotés ont pour longueurs des entiers de la forme a, a+1, c où c est la longueur de l'hypoténuse. On admet qu'il existe une infinité de TRPI et on classe les TRPI par ordre croissant des valeurs de a. Le triangle de cotés a=3, a+1=4 et c=5 est le premier TRPI. Déterminer les deuxième et troisième TRPI.

6) Ecrire une fonction **dicho** prenant comme argument une fonction f et 3 réels mini, maxi et epsi, et qui recherche, par dichotomie, sur l'intervalle [mini, maxi], une valeur approchée à moins de epsi d'une racine de la fonction f

V) Correction des exercices

<table>
<tr><td>1)
Remarquons que "somme" est une fonction locale</td><td>def moyenne(L):
  def somme(L):
  s=0
  for k in L: s += k
  return(s)
  return(somme(L)/len(L))</td><td>2)</td><td>def f(i,j):
  if i < j : return (i)
  elif i > j : return(j)
  else : return (3)</td></tr>
<tr><td>3)</td><td>def plusgrand(L):
  s = L[0]
  for k in L:
  if k > s : s = k
  return(s)</td><td>4)
pour nbre syracuse on peut utiliser len ou un compteur</td><td>def syracuse(n):
  m = n
  s = [n]
  while m > 1:
  if m % 2 == 0 : m = m // 2
  else : m = 3*m + 1
  s += [m]
  return(s)</td></tr>
<tr><td>5)</td><td>from math import sqrt, floor
def TRPI(a = 2):
  f = lambda x : floor(sqrt((x**2)+(x+1)**2))
  a,c = a, f(a)
  while c**2 != (a**2) + (a+1)**2 :
  a , c = a + 1, f(a+1)
  return(a, a+1, c)</td><td colspan="2">On trouve les premiers TRPI
(3, 4, 5), (20, 21, 29)
(119, 120, 169), (696, 697, 985)
(4059, 4060, 5741) , (23660, 23661, 33461)
On constate que les quotients de a(n+1)/a(n) et c(n+1)/c(n) convergent vers la plus grande racine de $X^2 – 6X + 1 = 0$</td></tr>
<tr><td>6)</td><td colspan="3">from math import *
def dicho(f = lambda x: x**2 - 2, mini = 1, maxi = 2,epsi = 0.1):
  a, b = mini, maxi
  while b - a > epsi:
  c = (a + b)/2
  if f(a) * f(c) < 0: b = c
  else: a = c
  return (a,b)</td></tr>
</table>