I Fonctions

1. Définition d'une fonction

On crée une fonction avec le mot clé `def` suivi du nom de la fonction (celui-ci ne doit pas être déjà utilisé) et de la liste des arguments entre parenthèses et séparés d'une virgule, puis de `:`. Les parenthèses sont obligatoires même s'il n'y a pas d'arguments. Le corps de la fonction doit être indenté par rapport à ces éléments.

```
>>> def fct(n) :                    # indenter la suite !!!!!
        """calcule 2 puissance n""" # commentaire affiché par help(fct)
        return 2**n                 # résultat de la fonction
>>> fct(4)
16
```

```
>>> def autre() :                   # fonction sans argument
        n = input()                 # demande de saisie
        n = int(n)                  # en Python 3, input renvoie une chaîne de caractères
        print(3**n)                 # affichage simple du résultat
>>> autre()
3                                   # on prend n=3
27
```

Arguments par défaut :

```
>>> def diff(x = 1,y = 2) :
        return y-x
>>> diff(4,2)
-2
>>> diff(4)   # y = 2 par défaut
-2
>>> diff(y = 1) # ici c'est x = 1 par défaut
0
```

Avec un nombre d'arguments variable :

```
>>> def somme(*arg) :      # rajouter *
        s = 0
        for k in arg :     # indenter le bloc de commandes
            s += k         # équivalent à s = s + k
        return s           # en dehors du bloc for ...
>>> somme(1,2,3,4,5)
15
```

2. La commande `return`

En Python, la commande `return` interrompt la fonction, il est donc possible de programmer une fonction en utilisant cette possibilité pour interrompre une boucle `for` par exemple. La fonction suivante recherche par exemple si `0` est un élément de la liste `L`

```
def zero(L) :
    for elt in L :
        if elt == 0 :
            return True
    return False
```

Mais cette solution doit être évitée lorsqu'il est possible de faire autrement (sans que cela ne devienne trop compliqué) ; il est en général suffisant de remplacer la boucle `for` par une boucle `while` et l'introduction d'un booléen :

```
def ZERO(L) :
    trouve = False
    k = 0
    while not trouve and k < len(L) :
        if L[k] == 0 :
            trouve = True
        k += 1
    return trouve
```

Il s'agit d'éviter d'interrompre une boucle `for`, utiliser plusieurs fois la commande `return` dans une seule fonction n'est pas à éviter systématiquement :

```
def valAbs(x) :
    if x > 0 :
        return x
    else :
        return -x
```

La commande `return` n'est pas toujours nécessaire. Comparons deux fonctions :

```
def f(x):
    return x**2

def g(x):
    print(x**2)
```

Lors de l'exécution, ces deux fonctions semblent avoir le même effet :

```
>>> f(3)
9
>>> g(3)
9
```

Cherchons maintenant à *utiliser* le résultat de la fonction, par exemple pour le mettre en mémoire :

```
>>> y = f(3)
>>> y
9
>>> type(y)
<class 'int'>
>>> z = g(3)
9
>>> z
>>> type(z)
<class 'NoneType'>
>>> z == None
True
```

La fonction `f` renvoie une valeur, ici un entier, alors que la fonction `g` ne renvoie rien (elle affiche juste le résultat), ce qui est pour Python l'objet `None`. Python renvoie un message d'erreur si on cherche à prendre le carré d'un tel objet :

```
>>> f(z)
in f
    return x**2
TypeError: unsupported operand type(s) for ** : 'NoneType'
```

L'objet `None` est néanmoins... un objet, et peut être l'argument d'une fonction dans laquelle il est détecté par un test :

```
def h(x):
    if x == None:
        return 0
    else:
        return 1

>>> h(z)
0
```

L'objet `None` en tant qu'argument d'une fonction n'est donc pas équivalent à une absence d'argument :

```
>>> h()
TypeError: h() missing 1 required positional argument: 'x'
```

La commande `return` est donc nécessaire dans une fonction pour que le résultat de cette fonction puisse être utilisé par la suite.

3. Fonctions agissant « sur place »

Pour des structures de données mutables comme les listes (ou les tableaux), certaines fonctions peuvent modifier la liste passée en argument et peuvent très bien ne rien renvoyer. Prenons par exemple une fonction `permutation` qui effectue une permutation circulaire sur une liste :

```
def permutation(L):
    n = len(L)
    a = L[-1]
    for i in range(n-1,0,-1):
        L[i] = L[i-1]
    L[0] = a

>>> L = [1,2,3,4,5,0]
>>> permutation(L)
>>> L
[0,1,2,3,4,5]
```

La liste L a été modifiée, on peut lui appliquer de nouveau la fonction :

```
>>> permutation(L)
>>> L
[5,0,1,2,3,4]
```

En revanche, `permutation` ne renvoyant rien, on ne peut pas effectuer deux permutations circulaires successives par cette commande :

```
>>> permutation(permutation(L))
in permutation
    n = len(L)
TypeError: object of type 'NoneType' has no len()
```

Une fonction modifiant sur place une liste `L` peut très bien ne rien renvoyer.
Par la suite, on n'utilisera alors pas le résultat de cette fonction qui est l'objet `None`, mais la liste `L` elle même.

On peut parler de *procédure* lorsqu'une telle suite d'instructions ne renvoie pas de valeur et de *fonction* lorsqu'une valeur est renvoyée.

On peut bien sûr aussi rajouter `return(L)` à la fin de la fonction si on veut qu'elle renvoie `L` en plus de modifier `L` sur place.

Signalons que la méthode `pop` renvoie le dernier élément d'une liste tout en le supprimant de la liste (elle modifie la liste sur place)

```
>>> L = [0,1,2,3]
>>> L.pop()
3
>>> L
[0,1,2]
>>> a = L.pop() # pas d'affichage car le résultat de la méthode est utilisé
>>> a
2
```

4. Variables globales et locales

Une variable *globale* est définie *en dehors* d'une fonction et peut être utilisée par cette dernière.

```
y = 3
def f(x):
    return x+y
```

```
>>> f(5)
8
```

La fonction cherche à utiliser une variable `y` qui n'est pas définie dans la fonction. Python recherche alors dans la table des variables dites globales, définies hors de la fonction, la valeur de `y`.
Si on définit maintenant une variable *à l'intérieur* d'une fonction, elle devient une variable *locale*, indépendante d'une éventuelle variable globale du même nom :

```
y = 3
def f(x):
    y = 2
    return x+y
>>> f(5)
7
>>> y
3 # c'est la variable locale y, dont la valeur est 2, qui est utilisée par la fonction,
    la variable globale de même nom n'est pas modifiée
```

L'utilisation de deux variables, l'une globale, l'autre, locale, de même nom est à éviter :

```
y = 3
def f(x):
    y = y+1
    return x+y
>>> f(5)
in f
    y = y+1
UnboundLocalError: local variable 'y' referenced before assignment
```

la commande `y =...` à l'intérieur de la fonction implique la création d'une variable locale `y`. C'est elle qui sera utilisée dans la fonction et pas la variable globale `y`. La commande `y = y+1` n'est pas valide puisque `y` n'est pas affectée. Il faut alors *déclarer* `y` comme variable *globale* pour pouvoir la modifier :

```
y = 3
def f(x):
    global y
    y = y+1
    return x+y
>>> f(5)
9
>>> y
4
```

Pour éviter les erreurs, il vaut mieux passer une variable en *argument* de la fonction plutôt que d'utiliser une variable globale...

```
def f(x,y):
    return x+y
```

... et éviter aussi de définir une variable locale du même nom qu'un argument de la fonction :

```
def f(L):
    L = L+[1]
    return L
>>> L = [4]
>>> f(L)
[4,1]
>>> L
[4]
```

Cette fois-ci la commande `L = L+[1]` est valide puisque `L` est un argument de la fonction `f`, mais `L+[1]` est affectée à une variable locale `L` du même nom que l'argument `L`. Cette variable locale n'existant pas hors de la fonction, si le but était de modifier la liste `L` sur place, il n'est pas atteint... Il le sera par exemple ainsi :

```
def f(L):
    L.append(1) # on travaille sur la liste L entrée en argument sans créer de variable
        locale du même nom
```

```
    return L
>>> L = [4]
>>> f(L)
[4,1]
>>> L
[4,1]
```

II Eléments de programmation

1. Boucle inconditionnelle `for`

La syntaxe est la suivante :

```
for i in range(début,fin,pas) :
    bloc de commandes
```

La boucle sera inconditionnellement exécutée en entier, sauf si elle comporte des tests aboutissant à un `return`.

2. Le test `if`

La commande `if` permet de distinguer plusieurs situations conduisant à un comportement différent du programme. La syntaxe est la suivante :

```
if cond 1 :
    bloc de commandes 1
elif cond 2 :
    bloc de commandes 2
elif cond 3 :
    bloc de commandes 3
...
elif cond n :
    bloc de commandes n
else :
    bloc de commandes n+1
```

Les conditions de test s'écrivent avec les opérateurs == ,!= , < , > , <= , >= , les opérateurs logiques `and` , `or` et `not`, la commande `in` ...
Si cond 1 est réalisée, le bloc de commande 1 est exécuté, sinon la cond 2 est testée et le bloc de commande 2 est exécuté si cond 2 est réalisée, ... Le bloc de commande n+1 n'est exécuté que si aucune des conditions 1 à n n'est réalisée. Le bloc de commande i < n+1 est donc executé si toutes les cond j < i ne sont pas réalisées et si cond i est réalisée.

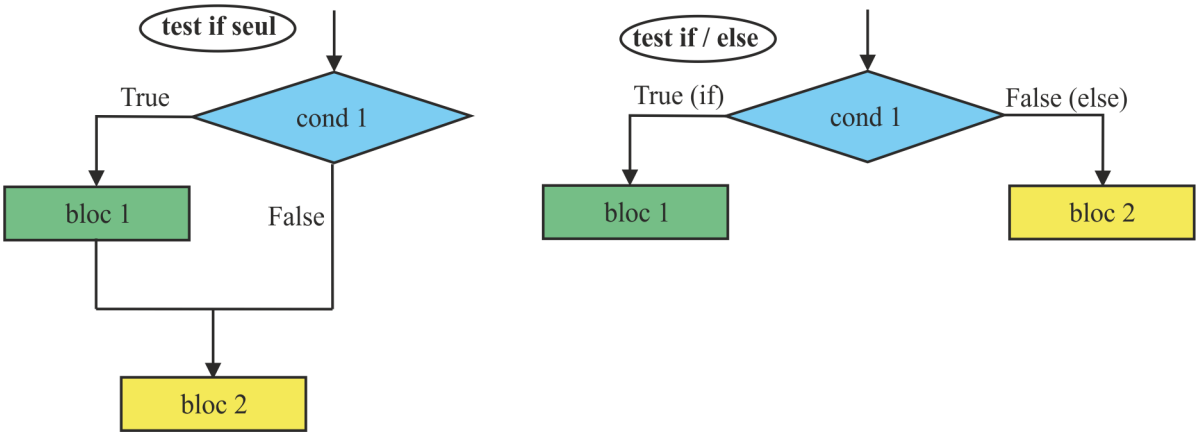
Il faut bien distinguer un simple test `if` :

```
if cond 1 :
    bloc de commandes 1
bloc de commandes 2
```

d'un test `if/else` :

```
if cond 1 :
    bloc de commandes 1
else :
    bloc de commandes 2
```

Dans le premier cas, le bloc de commandes 2 est exécuté que la cond 1 soit réalisée ou pas, alors que dans le second cas, il n'est exécuté que si cond 1 n'est pas réalisée.

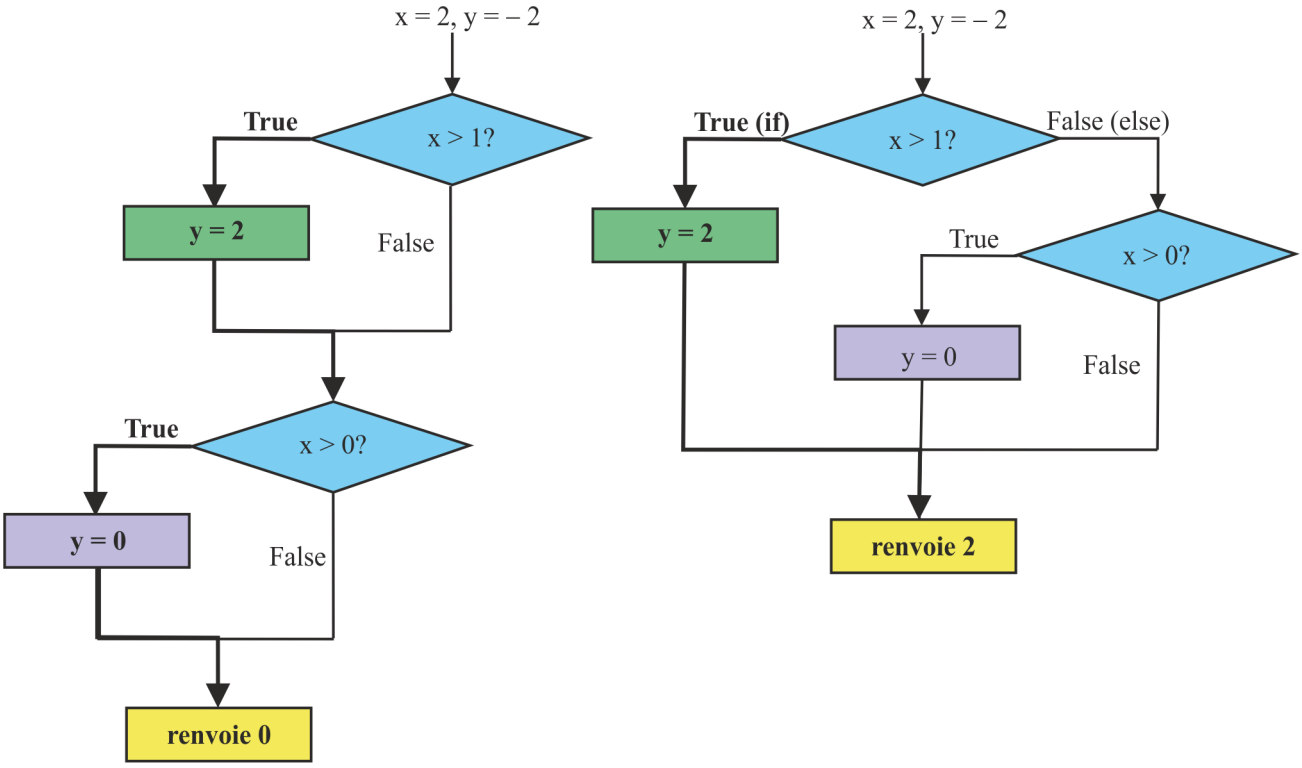


Prenons quelques exemples.

```
def fonction1(x):
    y = -x
    if x > 1:
        y = x
    if x > -1:
        y = 0
    return y

def fonction2(x):
    y = -x
    if x > 1:
        y = x
    elif x > -1:
        y = 0
    return y
```

Il n'y a pas de différence entre les résultats des deux fonctions quand $x \leqslant 1$. En revanche, dans le cas où $x > 1$, `fonction1` renvoie 0 car le deuxième test `if` est exécuté, alors qu'il ne l'est pas pour `fonction1` qui renvoie alors `x` :



Un autre exemple, avec un test « composé » :

```
>>> L = list(range(10)) + ['a','b','c']
>>> print(L)
[0,1,2,3,4,5,6,7,8,9,'a','b','c']
>>> L1 = list(range(1,11,3))
>>> print(L1)
[1,4,7,10]
>>> for elt in L :
        if elt in L1 or elt == 'b' :
            L.remove(elt)
>>> print(L)
[0,2,3,5,6,8,9,'a','c']
```

3. Boucle conditionnelle `while`

La syntaxe est la suivante :

```
while condition :
    bloc de commandes
```

Le bloc de commande est exécuté tant que la condition est réalisée. Il faut faire attention que la condition soit réalisée au départ pour que le bloc de commande soit exécuté au moins une fois et penser à modifier les paramètres de la condition dans le bloc de commande pour que le programme termine.

```
>>> def fact(n) :
        r = 1
        k = 1
        while k <= n :
            r = r * k
            k += 1
        return r
>>> fact(4)
24
>>> fact(1)
1
```

III Quelques exercices

1. Manipulation des listes

Dans cet exercice, L est une liste non vide.

1. Ecrire une fonction `MaxListe(L)`, d'argument L qui renvoie le maximum de L.
2. Ecrire une fonction `CompteMaxListe(L)` d'argument L qui renvoie le nombre d'occurences du plus grand élément de L.
3. Ecrire une fonction `ToutSurMax(L)` qui renvoie une liste sous la forme [maximum, occurences, liste des positions].

2. Nombres narcissiques (ENSAM PSI 2015)

1. Tester `list(str(1027))`.
   renvoie `['1','0','2','7']`.
2. Montrer que 93084 est un nombre narcissique, c'est-à-dire un nombre égal à la somme des puissances $p^{\text{ème}}$ de ses chiffres, où p est le nombre de ses chiffres (ici $p = 5$).
3. Ecrire une fonction qui teste si un nombre est narcissique.
4. Afficher tous les nombres narcissiques $\leqslant 10000$.
5. Ecrire une fonction qui renvoie tous les nombres narcissiques entre `n` et `N`, arguments de la fonction.

IV Quelques exercices : corrigés

1. Manipulations de listes

1.

```
def MaxListe(L) :
    max = L.pop()
    while len(L) > 0 :
        elt = L.pop()
        if elt > max :
            max = elt
    return max
```

2.

```
def CompteMaxListe(L) :
    max = L.pop()
    k = 1
    while len(L) > 0 :
        elt = L.pop()
        if elt > max :
            max = elt
            k = 1
        elif elt == max :
            k += 1
    return k
```

3.

```
def ToutSurMax(L) :
    k = 0
    max = L[0]
    pos = [0]
    nb = 1
    while k < len(L)-1 :
        k += 1
        elt = L[k]
        if elt > max :
            max = elt
            nb = 1
            pos = [k]
        elif elt == max :
            nb += 1
            pos.append(k)
    return [max,nb,pos]
```

2. Nombres narcissiques

1.

```
93084 == 9**5 + 3**5 + 0**5 + 8**5 + 4**5
```

2.

```
def narcissique(n) :
    L = list(str(n))
    p = len(L)
    s = 0
    for elt in L :
        s += int(elt)**p
    return n == s
```

3.

```
for k in range(10001) :
    if narcissique(k) :
        print(k)
```

4.

```
def Narcisse(n,N) :
    for k in range(n,N+1) :
        if narcissique(k) :
            print(k)
```