Mathématiques supérieures le 20 Septembre 2019

Corrigé du TP n°2 - Algorithmique et Python

Dans ce TP, nous allons voir les bases de l'algorithmie via le langage Python. Nous allons apprendre à utiliser les concepts de fonctions, de structures conditionnelles et de structures itératives (les boucles). Mais tout d'abord, nous allons commencer par quelques conseils d'utilisation puis nous allons voir comment **doit** se présenter, se structurer un bloc d'instructions en Python.

1. Utilisation de l'éditeur

Comme nous l'avons vu dans le premier TP, l'environnement de développement est essentiellement composé de deux parties : le *shell* que nous avons beaucoup utilisé précédemment et **l'éditeur**. Ce dernier a une fonction très simple : il va nous servir à **enregistrer** nos futurs programmes, codes, etc...

En effet, comme vous pouvez le constater, tout ce que nous avez pu écrire au précédent TP dans la console interactive (shell) a totalement disparu ! Le shell ne "sert" qu'à exécuter nos instructions mais pas à les sauvegarder. Ce n'est évidemment pas très pratique si on veut créer un script ou un logiciel.

Ainsi, pour pouvoir récupérer nos codes pour de futures utilisations, nous allons utiliser l'éditeur. Celui-ci est un simple éditeur de texte : on écrit ce que l'on veut puis on enregistre notre fichier sur le disque dur. Dans l'environnement EduPython, l'éditeur a également plusieurs fonctions supplémentaires accessibles via la barre des menus de la fenêtre de Edupython, via des raccourcis claviers ou via le menu contextuel du clic droit de la souris (dans l'éditeur).

Voyons quelques possibilités importantes sur un exemple :

Écrivons **tout** le code suivant **dans l'éditeur**.

Dans l'éditeur

```
1 # addition de variables
2 L=2
3 M=3
4 print(M,'plus',L,'=',M+L)
```

Comme on peut le voir, l'écriture de ces lignes dans l'éditeur n'a pas d'effet sur la console ! Mais alors comment demander à la console de d'exécuter cette suite d'instructions ?

Et bien on utilise les commandes suivantes, au choix :
Remarque : Attention, il est possible que votre éditeur soit en anglais et que le français ne soit pas disponible ; traduisez alors "run" ou "Execute" par "exécuter".

	Barre d'outils	**Raccourcis**	**Menu contextuel**
Exécuter les lignes sélectionnées	Icône 	Ctrl+F7	Source Code >> Exécuter la sélection
Exécuter tout le fichier en cours	Icône	Ctrl+F9	

Attention ! On choisira le plus souvent l'exécution de la sélection car l'exécution du fichier entier sur Edupython a pour effet de réinitialiser toutes les variables préalablement définie !
Ainsi, lorsque que l'on voudra exécuter tout notre fichier en cous, on choisira de "Tout sélectionner" (Ctrl+A) puis d'"Exécuter la sélection" (Ctrl+F7)

Il y a bien-sûr d'autres options mais nous ne nous en servirons pas pour le moment.

Dès à présent, testons les différentes options ci-dessus sur le code que nous avons écrit dans l'éditeur.

Pour le futur, il est vivement conseillé de connaître les raccourcis claviers pour l'exécution d'instructions écrites dans l'éditeur ! Ils apportent un confort et un gain de temps appréciables !

Concernant le fichier de sauvegarde du code en lui-même ; prenons quelques petites habitudes d'organisation pour aujourd'hui et les prochains TP :

— Dès le début de l'édition d'un fichier dans l'éditeur on indiquera en commentaire (grâce au #), sur la première ligne, une courte description de ce que contiendra le fichier ;

— Dès le début de l'édition d'un fichier, on enregistrera le fichier sur le disque dur en lui donnant un nom **explicite** à propos de ce qu'il contiendra. De plus, on sauvegardera **très, très régulièrement** le fichier en cours : il est toujours possible que survienne une panne, peu importe laquelle. Il serait dommage de perdre une, voire deux heures d'écriture de programmes !

— Pendant l'écriture du code, on **commentera** (toujours grâce au #) les instructions délicates à comprendre. Ainsi, notre code sera compréhensible pour d'autres et même pour soi-même si on revient plus tard lire le fichier. On restera toujours clair et concis dans nos commentaires ; pas la peine d'écrire un roman, quelques mots bien choisis suffisent.

2. Structuration du code

Dans la plupart des langages informatiques, la programmation se fait par blocs d'instructions qui dépendent d'une structure de contrôle (une fonction, une boucle, une structure conditionnelle... nous allons voir tout cela, pas d'inquiétude !). Ainsi, le développeur soucieux de bien présenter son code (afin qu'il soit lisible et plus facilement compréhensible) va séparer les différents blocs de son code. Ce principe à un nom : **l'indentation** du code. Il s'agit de décaler, **indenter**, chaque bloc d'un nombre d'espaces fixe grâce à la touche **Tabulation**.

Pour la plupart des langages, l'indentation du code est une simple règle "de bonne conduite" : ce n'est pas nécessaire, mais cela rend le code beaucoup plus lisible. Mais en Python, **l'indentation est OBLIGATOIRE** ! Elle fait partie de la syntaxe du langage. Ainsi, peu importe qui développe en Python, son code est toujours un minimum lisible !

Voici le principe de l'indentation en Python :

Un bloc est défini par :

— un en-tête qui se termine toujours par un double point (:)

— une suite d'instructions indentées par rapport à l'en-tête

— le retour à l'indentation de l'en-tête (s'il y a encore des instructions ensuite).

Voici quelques exemples pour mieux comprendre :

Un bloc

```
entete:
    instruction 1
    instruction 2
    instruction 3
```

Deux blocs imbriqués

```
entete1:
    instruction 1.1
    instruction 1.2
    entete2:
```

```
5         instruction 2.1
6         instruction 2.2
7     instruction 1.3
8     instruction 1.4
```

Nous allons passer immédiatement à la pratique en découvrant les concepts de fonctions, de structures conditionnelles, et de boucles. Voyons tout d'abord les fonctions.

3. Définir une fonction en Python

a. Définition d'une fonction

Nous avons déjà manipulé quelques fonctions en Python. Par exemple, la fonction **print** qui a pour effet d'exécuter l'instruction qu'elle a en argument et de l'afficher dans le shell ; ou la fonction **len** qui retourne la longueur d'une chaîne de caractères. Mais comment faire pour créer nos propres fonctions ??

En Python, on utilise :

— le mot clé **def** suivi du nom que l'on veut donner à notre fonction, puis suivi, entre parenthèses, de la liste de ses arguments (ou paramètres), et terminé par le double point. Voici pour l'en-tête du bloc qui défini notre fonction.

— Ensuite, on indente chaque instruction composant notre fonction. Avant d'écrire la première instruction, la première ligne du bloc doit être un commentaire appelé *docstring* qui explique ce que fait la fonction, et la nature de chaque argument

Voici un prototype de fonction en Python :

Une fonction en Python

```
1 def nomdelafonction(arg1,arg2,arg3):
2     """ docstring : détails de la fonction """
3     instruction 1
4     instruction 2
5     .
6     .
7     .
```

Pour que la fonction renvoie un résultat après son appel, on utilisera l'instruction **return**.

Testons ce concept sur un exemple. On écrit le code suivant dans l'éditeur :

```
1 def bonjour(prenom):
2     """ Fonction qui renvoie la chaîne de caractères 'Bonjour prenom ! Comment vas-tu ?'
          où l'argument prenom est une chaîne de caractères"""
3     resultat='Bonjour '
4     resultat+=prenom
5     resultat+=' ! Comment vas-tu ?'
6     return resultat
```

Ensuite, on exécute dans le shell ces lignes de code (avec la méthode de notre choix : raccourcis clavier, menu,...).

La fonction que l'on vient de définir a pour nom `bonjour` et pour argument `prenom`. On remarque qu'il est bien précisé dans le docstring que l'argument `prenom` doit être une chaîne de caractères !
Cette fonction renvoie (ou retourne) la chaîne de caractères contenue dans la variable `resultat` dont la valeur est obtenue par la suite d'instructions du bloc de `bonjour`.

Ainsi, en changeant la valeur de l'argument `prenom`, on obtient un résultat différent. On peut comparer cela à une fonction mathématiques :
Si je considère le fonction $f : x \mapsto 2x$; on a $f(0) = 0$ et $f(2) = 4$!

D'accord, mais comment fait-on pour appeler notre fonction et lui donner les arguments que l'on veut ??
Très simple, il suffit de l'appeler par son nom et de lui donner, entre parenthèses, le ou les arguments nécessaires. Essayons avec notre fonction `bonjour`. On peut essayer le code suivant, soit directement dans le shell, soit dans l'éditeur, puis en l'exécutant : mais dans ce cas, attention, il ne faut pas oublier d'enlever l'indentation relative à la fonction que l'on vient d'écrire. En effet, si on écrit notre instruction avec la même indentation que le bloc de la fonction, Python interprétera cela comme la suite de ce même bloc et pas comme une instruction séparée !!

```
1 bonjour('Charles')
```

Bien-sûr, on peut essayer avec d'autres chaînes que `'Charles'`

```
1 bonjour('Jean')
```

Remarque 1.

Pour bien comprendre ce que l'on fait, le bloc interne de la fonction `bonjour` comporte quatre instructions. Mais on aurait pu réduire ceci à une seule instruction, et sans même définir une variable !

```
1 def bonjour(prenom):
2   """ Fonction qui renvoie la chaîne de caractères 'Bonjour prenom ! Comment
        vas-tu ?' où l'argument prenom est une chaîne de caractères"""
3   return 'Bonjour '+prenom+' ! Comment vas-tu ?'
```

Exercice 1.

Définir une fonction `reponse` qui prend pour argument une chaîne de caractère `nom` et qui renvoie la chaîne de caractère `'Très bien ! Et vous Madame '` concaténée avec la chaîne `nom` puis concaténée avec la chaîne `' ?'`.
Par exemple, `reponse('Roberta')` doit retourner la chaîne de caractères :

`'Très bien ! Et vous Madame Roberta ?'`

N'oubliez pas le docstring !!!

Correction.

```
def reponse(nom):
  """ Fonction qui renvoie la chaîne de caractères 'Très bien ! Et vous Madame
      nom ?' où l'argument nom est une chaîne de caractères"""
  return 'Très bien ! Et vous Madame '+nom+' ?'
```

b. Les arguments d'une fonction

Les fonctions précédentes ne possédaient qu'un seul argument. Mais bien-sûr, une fonction peut en posséder plusieurs. Dans ce cas, ils sont séparés par des virgules lors de la définition ou de l'appel de la fonction. Par exemple, définissons une fonction qui calcule la distance euclidienne entre un point (x, y) de $\mathbb{R}^2$ et le point $(0, 0)$:

```
from numpy import sqrt    # On a besoin de la fonction racine carrée ici

def dist(x,y):
  """ Calcule la distance euclidienne entre (x,y) et (0,0) où x,y sont des int ou des
      float """
  return sqrt(x**2+y**2)
```

Testons cette fonction pour différents points $(1, 1)$, $(1000, 200)$, $(3.5, -7.2)$:

```
dist(1,1)
dist(1000,200)
dist(3.5,-7.2)
```

Exercice 2.

Définir une fonction `distance` qui prend pour arguments `x1`,`y1`,`x2`,`y2` et qui retourne la distance euclidienne entre les points (x_1, y_1) et (x_2, y_2) de $\mathbb{R}^2$.

d...dddocstring !!!

Correction.

```
def distance(x1,y1,x2,y2):
  """ Calcule la distance euclidienne entre (x1,y1) et (x2,y2) où x1,y1,x2,y2
      sont des int ou des float """
  return sqrt((x1-x2)**2+(y1-y2)**2)
```

On vient de voir comment passer plusieurs arguments à une fonction. Quand on définit une fonction de cette façon, les arguments demandés sont obligatoires : pour appeler la fonction, il faut lui fournir une

valeur pour chaque argument de la liste sous peine de recevoir un beau message d'erreur de Python (par exemple, on peut essayer `distance(1,1,2)` pour voir que Python sais très bien compter !).
Dans certains cas, on aimerait avoir des arguments optionnels : c'est-à-dire que si on fournit ces arguments à la fonction, elle en tiendra compte, mais si on ne les fournit pas, alors elle prendra à leur place des valeurs prédéfinies à l'avance que l'on appelle valeurs **par défaut**.

En Python, rien de plus facile, il suffit de préciser la valeur par défaut juste après après un argument que l'on veut rendre optionnel en les séparant par un `=`. **ATTENTION :** les arguments optionnels doivent **toujours** être placés **après** les arguments obligatoires dans la définition d'une fonction.

Exemple, une fonction `venucomment` qui prend un paramètre obligatoire `moment` (str) et un paramètre optionnel `transport` (str) de valeur par défaut `'voiture'` et qui renvoie un str `'Ce '+moment+', je suis venu en '+transport` :

```
def venuquandcomment(moment,transport='voiture'):
  """ Retourne le str 'Ce moment, je suis venu en transport' où moment et transport sont
      des str et la valeur par défaut de transport est 'voiture'."""
  return 'Ce '+moment+', je suis venu en '+transport
```

Testons :

```
venuquandcomment('matin')
venuquandcomment('Vendredi',transport='nageant')
```

Remarquez une chose **importante** ici : quand je veux fournir une valeur au paramètre optionnel, je **DOIS** d'abord l'appeler par son nom puis définir sa valeur après un = dans l'appel de la fonction. Ce n'est pas nécessaire ici, mais lorsque la fonction admet plusieurs arguments optionnels, c'est bien-sûr obligatoire (sinon la fonction ne peut pas savoir quel argument optionnel vous avez fourni).

Exercice 3.

Définir une fonction `diagonale`
- qui prend pour arguments des int ou float `l` (c'est un ℓ) et `L` où l'argument `L` est optionnel de valeur par défaut 10 ;
- qui renvoie la mesure d'une diagonale d'un rectangle de largeur `l` et de longueur `L`.

Correction.

```
from numpy import sqrt    # si ce n'est pas déjà fait

def diagonale(l,L=10):
  """ Retourne la mesure d'une diagonale d'un rectangle de largeur l et de
      longueur L où l et L sont des int ou float et L a pour valeur par défaut
      10."""
  return sqrt(l**2+L**2)
```

c. Portée des variables

Comme on l'a vu dans notre toute première fonction `bonjour`, on peut définir et utiliser des variables à l'intérieur d'une fonction. Dans `bonjour`, nous avons utilisé une variable `resultat`. Essayons d'y accéder en tapant dans le shell :

```
1 resultat
```

Que nous dit Python ? Et bien que cette variable n'existe pas ! Pourtant nous l'avions bien définie et utilisée dans `bonjour`. Que s'est-il passé ?

En fait, les variables ont une **portée** : les variables que nous définissons directement dans le shell ou en exécutant une instruction du type `L=2` dans l'éditeur ont une portée `globale` c'est-à-dire que tant que Python est en route, on peut lui demander la valeur de cette variable, il nous la retournera.
Par opposition, les variables que nous définissons dans une fonction ont une portée **locale** : cette variable n'est définie et sa valeur n'est accessible que dans la fonction en question.

Pour mieux comprendre le principe, effectuons les tests suivants :

```
1 L=2
2
3 def f():
4   x=L
5   return x
6
7 f()
8
9 x
```

Que se passe-t-il à l'appel de `x` ?

```
1 def f2():
2   L=5 # définition locale de L
3   return L
4
5 f2()
6
7 L # récuperation de la valeur globale définie dans l'exemple précédent !
```

On voit bien ici que dans une fonction, même si la variable est définie globalement, la définition locale du même nom n'interfère pas !

On peut tout de même créer des variables définies globalement dans une fonction : on utilise l'instruction **`global`** suivi du nom de la variable avant de définir sa valeur :

```
1 def f3():
2   global L
3   L=100
4   return L
5
```

```
f3()

L
```

4. Les structures conditionnelles

a. if ... else

Une des plus simple structure conditionnelle, le "si ... sinon" permet d'exécuter un groupes d'instruction si une condition imposée est vérifiée ou un autre groupe si elle ne l'est pas. Comment utiliser cette structure en Python, et à quoi cela peut-il bien servir ? Commençons par le "comment" :

En anglais, si se dit **if** et sinon se dit **else**. Ainsi, en Python, on utilisera ces mots clés comme en-têtes des blocs d'instructions à exécuter ou non selon la véracité d'un expression booléenne. Voyons ceci sur un prototype de structure conditionnelle **if else** :

blocs if else

```
if expression_booleene:
    instruction_1
    instruction_2
else:
    instruction_1
    .
    .
    .
```

Le fonctionnement est le suivant : si l'expression booléenne s'évalue en `True` alors les instructions du bloc du **if** s'exécutent, sinon, si l'expression s'évalue en `False` et alors ce sont les instructions du bloc du **else** qui s'exécutent.

Donnons tout de suite un exemple : une fonction `plusgrandquecent` qui prend pour argument un nombre entier `n` et qui renvoie 'oui, ce nombre est plus grand que 100' si `n` est plus grand que 100 et "non, ce nombre n'est pas plus grand que 100" sinon.

```
def plusgrandquecent(n):
    """ Retourne 'oui, ce nombre est plus grand que 100' si n est plus grand que 100 et
        "non, ce nombre n'est pas plus grand que 100" sinon, où n est un int"""
    if n >= 100:
      return 'oui, ce nombre est plus grand que 100'
    else:
      return "non, ce nombre n'est pas plus grand que 100"
```

Testons cette fonction :

```
plusgrandquecent(124)
plusgrandquecent(4)
```

Exercice 4.

Définir une fonction div(n,m) qui prend pour argument deux entiers `n` et `m` et qui renvoie 'oui `n` est divisible par `m`' si `n` est divisible par `m` et 'non `n` n'est pas divisible par `m`' sinon.

b. if ... elif ... else

On peut généraliser la structure conditionnelle précédente grâce au mot clé Python **`elif`** que l'on pourrait traduire par "sinon si". Voici le prototype de cette structure :

blocs if elif else

```
if expression_booleene_1:
    instruction 1
    instruction 2
elif expression_booleene_2:
    instruction 1
    instruction 2
else:
    instruction 1
    .
    .
    .
```

Le principe est quasiment le même que la structure if ... else :

— Si `expression_booleenne_1` est évaluée à `True`, on exécute le bloc du **`if`** ;

— Sinon (i.e. Si `expression_booleenne_1` est évaluée à `False`) et si `expression_booleenne_2` est évaluée à `True`, on exécute le bloc du **`elif`** ;

— Sinon on exécute le bloc du else ;

On peut bien-sûr mettre plusieurs blocs elif à la suite.

Exercice 5.

Définir une fonction `ordremots` qui prend la chaîne de caractères `mot` en argument et qui permet de dire où se trouve `mot` par rapport aux mots 'avion', 'serpent' et 'xylophone' dans l'ordre alphabétique.

Correction.

```
def ordremots(mot):
    """ Fonction qui renvoie où se place mot (str) dans l'ordre alphabétique par
         rapport aux mots avion, serpent et xylophone """
    if mot<='avion':
        return mot+" est avant avion dans l'ordre alphabétique"
    elif mot>'avion' and mot<='serpent':
       return mot+" est entre avion et serpent dans l'ordre alphabétique"
    elif mot>'serpent' and mot<='xylophone':
        return mot+" est entre serpent et xylophone dans l'ordre alphabétique"
    else:
        return mot+" est après xylophone dans l'ordre alphabétique"
```

Exercice 6. *(*)*

Définir une fonction `trinombre` qui prend pour arguments trois nombres (int ou float) `p`, `q` et `r` et qui renvoie ces trois valeurs triées par ordre croissant.

Pour retourner ces trois valeurs, on pourra utiliser un "tuple" (un triplet ici) : en Python, l'instruction `p,q,r` renvoie le triplet `(p,q,r)`.

Correction.

```
def trinombre(p,q,r):
    """ Fonction qui renvoie les int ou float p,q,r dans l'ordre croissant"""
    if p<=q and q<=r:
        return p,q,r
    elif q<=p and p<=r:
        return q,p,r
    elif q<=r and r<=p:
        return q,r,p
    elif r<=q and q<=p:
        return r,q,p
    elif r<=p and p<=q:
        return r,p,q
    else:
        return p,r,q
```

5. Instructions itératives

Réaliser une **itération** ou une **boucle**, c'est répéter un certain nombre de fois des instructions semblables qui dépendent d'un variable qui est modifiée (ou non) à chaque répétition.

En Python comme dans la plupart des langages de programmation, il existe deux façons de réaliser une boucle :

— La boucle `for`, que l'on utilise lorsque l'on connaît à l'avance le nombre d'itérations à effectuer dans notre boucle : on dira que c'est une **boucle énumérée** ;

— La boucle `while`, que l'on utilise lorsque le nombre d'itérations dépend de la véracité d'une expression booléenne : on dira que c'est une **boucle conditionnelle**.

a. La boucle énumérée `for`

Avant de parler de la boucle `for`, on va s'intéresser à une nouvelle fonction présente nativement dans le langage Python ; la fonction `range`. Celle-ci se comporte de la façon suivante (on peut également taper `help("range")` pour lire le docstring de cette fonction) :

La fonction `range` prend entre 1 et 3 arguments qui sont des entiers (int) :

- `range(n)` énumère les entiers de 0 à $\mathbf{n-1}$;
- `range(m,n)` énumère les entiers de m à $\mathbf{n-1}$ (et renvoie une énumération vide si $m > n-1$) ;
- `range(m,n,r)` énumère les entiers m, $m+r$, $m+2r$,..., $m+kr$ où k est le plus grand entier tel que $m+kr \leq n-1$.

On remarque qu'en tapant `range(5)` par exemple dans le shell, Python nous retourne une simple mention `range(5)`. Mais alors comment vérifier si notre énumération est bien celle attendue ?

Pour cela, il faut la ranger dans une liste (nous verrons exactement ce que sont les listes et comment les manipuler dans de prochains TP). Testons les codes suivants (directement dans le shell) :

```
1 list(range(45))
2 list(range(4,45))
3 list(range(4,45,3))
4 list(range(3,1))
```

Grâce aux énumérations obtenues avec la fonction `range`, nous allons pouvoir créer un premier type de boucle ... énumérée !

On définit une boucle énumérée avec le mot-clé `for`, qui suit la structure de bloc suivante en Python :

Structure de la boucle `for`

```
1 for ... in range(...):
2     instruction1
3     instruction2
4     .
5     .
6     .
```

Immédiatement après le `for` doit être indiqué le nom d'une variable qui prendra les différentes valeurs de l'énumération donnée par le `range`. Si aucune variable parmi celles déjà déclarées ne porte le nom proposé après le `for`, alors cette variable sera créée sinon, attention, si elle existe déjà, elle sera modifiée par son "passage" dans la boucle !

Pour chacune des valeurs de cette variable, le bloc d'instructions indentées après l'entête `for` sera exécuté en tenant compte, bien-sûr, de la modification de la valeur de notre variable qui contient tour à tour les nombres de l'énumération. Testons notre première boucle `for`

```
for i in range(26):
    print('Le carré de',i,'est égal à',i**2)
```

On peut imbriquer plusieurs boucles les unes dans les autres en respectant toujours les règles d'indentations des blocs :

```
def tableaddition(n):
    """ Fonction qui affiche la table d'addition des nombres compris entre 1 et n où n
        est un int"""
    for i in range(1,n+1):
        print('|',end=' ')
        for j in range(1,n+1):
            print(i+j,end=' ')
        print('|')
```

Essayons cette fonction dans le shell après l'avoir exécutée : `tableaddition(5)` par exemple.

Exercice 7.

Définir une fonction `tablemultiplication` qui prend pour argument un entier `n` et qui affiche la table de multiplication des nombres de 1 à n.

Correction.

```
def tablemultiplication(n):
    """ Fonction qui affiche la table de multiplication des nombres compris
        entre 1 et n où n est un int"""
    for i in range(1,n+1):
        print('|',end=' ')
        for j in range(1,n+1):
            print(i*j,end=' ')
        print('|')
```

Exercice 8. *(*)*

1. Définir une fonction qui prend pour argument un entier `n` et qui calcule et retourne la somme des entiers de 1 à n.
2. Définir une fonction qui prend pour argument deux entiers `n`, `m` et qui calcule et retourne n^m. (on n'aura bien-sûr pas recours à l'opérateur **)
3. Définir une fonction qui prend pour argument une chaîne de caractères `c` et qui la retourne à l'envers. (exemple : si on donne `'avion'` à la fonction, elle renvoie `'noiva'`). (on n'aura bien-sûr pas recours au slicing négatif).

Correction.

1.
```python
def somme(n):
    """ fonction somme des entiers de 1 à n"""
    S=0 #contiendra la somme
    for i in range(1,n+1):
        S+=i
    return S
```

2.
```python
def puissance(n,m):
    """ n^m """
    p=1 #contiendra les produits successifs
    for i in range(1,m+1):
        p*=n
    return p
```

3.
```python
def envers(chn):
    """ renvoie le str chn à l'envers """
    e='' #contiendra le mot à l'envers
    for i in range(0,len(chn)):
        e+=chn[-i-1]
    return e
```

Invariant de boucle

Définition 1. *Invariant de boucle*

On appelle **invariant de boucle**, toute assertion vérifiant les conditions suivantes :

i) **Initialisation** Cette assertion est vraie avant la première itération de la boucle ;

ii) **Conservation** si cette assertion est vraie avant une itération de la boucle, elle le reste avant l'itération suivante ;

iii) **Terminaison** une fois la boucle terminée, l'assertion fournit une propriété utile qui sert à établir, prouver, corriger l'algorithme ;

La troisième propriété d'un invariant de boucle signifie qu'il peut être utilisé pour rédiger l'algorithme sans erreur ; également pour démontrer que l'algorithme se termine bien, et également pour analyser l'algorithme (savoir ce qu'il fait).

Essayons de comprendre tout ceci sur un exemple.

On veut étudier les termes de la suite récurrente $(u_n)_{n\in\mathbb{N}}$ telle que $u_0 = 1$ et pour $n \in \mathbb{N}$, $u_{n+1} = 3*u_n+2$. On veut donc écrire une fonction en Python qui nous calcule et renvoie le terme d'indice n de cette suite.

Considérons l'instruction $(*)$ `u=3*u+2`.

Si la variable `u` contient initialement la valeur u_n, après l'exécution de cette instruction, `u` contiendra u_{n+1}.

On a donc l'idée, pour calculer la valeur de u_{1000} par exemple, d'initialiser une variable `u` en u_0 et d'appliquer 1000 fois l'instruction $(*)$. On obtient alors la fonction suivante qui calcule le terme d'indice `N` de la suite (u_n) :

```
def u(N):
    u=0
    for n in range(N):
        u=3*u+2
    return u
```

On peut alors énoncer l'invariant de boucle suivant :

À l'entrée de la boucle indexée par `n`, la variable `u` prend la valeur de u_n.

Exercice 9.

Prenons le cas du calcul de $n!$. Déterminer un invariant de boucle à respecter pour produire un algorithme itératif de calcul de $n!$. Puis écrire une fonction Python correspondant à cet algorithme.

Correction.

L'invariant de boucle est :

À l'entrée de la boucle indexée par `i`, la variable `p` prend la valeur de $(i-1)!$.

```
def factorielle(n):
    p=1
    for i in range(1,n+1):
        p*=i
    return p
```

Exercice 10.

Déterminer les invariants de boucles pour les algorithmes de l'exercice 8.

Correction.

Les invariants de boucle sont :

À l'entrée de la boucle indexée par `i`, la variable `S` prend la valeur de $1+2+...+i$.

À l'entrée de la boucle indexée par `i`, la variable `p` prend la valeur de n^i.

À l'entrée de la boucle indexée par `i`, la variable `e` prend contient les i dernières lettres de c dans l'ordre inverse.

Exercice 11. *(*)*

Déterminer un invariant de boucle et écrire une fonction Python pour la suite de Fibonacci : $u_{n+2} = u_{n+1} + u_n$ et $u_0 = 0$, $u_1 = 1$.

Correction.

L'invariant de boucle est :

À l'entrée de la boucle indexée par `i`, la variable `u` prend la valeur de u_i et la variable `v` prend la valeur de u_{i-1}.

```
def fibonacci(n):
    u,v=1,0
    for i in range(1,n+1):
        u,v=u+v,u
    return u
```

Parcourir une chaîne de caractères

Avec la boucle `for` dans Python, on peut également énumérer d'autres objets que les énumérations obtenues par la fonction `range`. En fait, il existe une notion d'**énumérable** en Python : ce sont des objets qui possèdent un compteur interne permettant au mot-clé `in` de la boucle `for` de donner successivement à l'itérateur (la variable souvent nommée `i`) les valeurs contenues dans l'objet énumérable. Voyons tout de suite un exemple avec les chaînes de caractères, qui sont comme nous nous en doutons, des énumérables :

Testons la fonction suivante :

```
def epelermot(mot):
    for lettre in mot:
        print(lettre,end='--')
```

Exercice 12. *(*)*

Définir une fonction qui prend comme arguments une chaîne `mot` et un caractère `lettre` (par exemple `lettre='t'`) et qui **retourne** `True` si le caractère `lettre` est contenu dans la chaîne `mot` et `False` sinon.

Correction.

```
def lettredansmot(mot,lettre):
    for caractere in mot:
        if caractere==lettre:
            return lettre+' est dans '+mot
    return lettre+" n'est pas dans "+mot
```

b. La boucle conditionnelle `while`

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine expression booléenne n'est pas vraie (évaluée en `True` donc). Il est donc possible que cette expression booléenne ne soit jamais vraie auquel cas on n'entre pas dans la boucle ; mais aussi que cette expression soit toujours vraie, auquel cas on reste indéfiniment dans la boucle (et ça, ce n'est pas bon !).
Voici la syntaxe du bloc d'une boucle conditionnelle :

```
while expression_booleene:
    instruction1
    instruction2
    .
    .
    .
```

Et voici des exemples que nous allons tester :

```
while 0+0>1:
    print('ahah', end='')
print('fini !')
```

Que se passe-t-il et pourquoi ?

```
while 0+0==0:
    print('ahah', end='')
print('fini !')
```

Que se passe-t-il et pourquoi ?

Ainsi, si une condition initialement vraie n'est pas modifiée dans le corps de la boucle après chaque itération, on aura affaire à des boucles infinies comme la précédente ! Il faut donc considérer une variable dont la valeur changera à chaque tour de boucle et qui permettra à l'expression booléenne de s'évaluer en `False` après un nombre fini de tours de boucle. Essayons un exemple :

```
def comptearebours(n):
    """ affiche le compte à rebours à partie de l'int n """
```

```
    i=n
    while i>=0:
      print(i,end=' ')
      i-=1 # A ne surtout pas oublier, sinon : boucle infinie !
```

Exercice 13.

On considère la situation suivante : on se donne un nombre $p \in \mathbb{N}^*$. Si cet entier est pair, on le divise par 2, s'il est impair, on le multiplie par 3 puis on ajoute 1. Puis on continue avec le résultat, jusqu'à arriver à 1.

— Écrire une fonction qui prend pour arguments deux entiers `p` et `n` et qui retourne la n-ième étape de la situation suivante en commençant avec le nombre p.

— Écrire une fonction qui prend pour argument `p` et qui retourne le plus petit entier n tel qu'à la n-ieme étape, le résultat est égal à 1.

Correction.

```
def suitesyracuse(p,n):
    """ renvoie le n ieme terme de la suite de Syracuse de premier terme p -- où
         n et p sont des int """
    suite=p
    for k in range(n):
        if suite%2==0:
            suite = suite//2
        else:
            suite = 3*suite+1
    return suite


def syracuse(p):
    """ renvoie le premier indice tel que la suite de Syracuse de premier terme
        p arrive en 1 """
    n=1
    while suitesyracuse(p,n)!=1:
        n+=1
    return n
```

Remarque 2.

La suite $(u_n)_{n\in\mathbb{N}}$ définie dans l'exercice précédent est plus connue sous le nom de *suite de Syracuse*. Cette suite, sous ses apparences anodines, a été et est toujours un casse-tête pour de nombreux mathématiciens : même si la question 2 de l'exercice précédent le laisse entendre, on ne sait pas si pour un terme initial donné $u_0 \in \mathbb{N}$, il existe bien un $n \in \mathbb{N}$ tel que $u_n = 1$! Il s'agit d'une conjecture, et elle n'a toujours pas été démontrée (ou infirmée).

Exercice 14. *(*)*

1. Écrire une fonction qui prend comme argument un entier et qui le renvoie à l'envers. Par exemple, si on donne 137369 à la fonction, elle renvoie 963731.
2. Écrire une fonction itérative qui permettent de renvoyer le quotient et le reste d'une division euclidienne.

Correction.

1.
```
def enversentier(n):
    """ Renvoie l'int n à l'envers ! """
    k=0
    envers=0
    while 10**k<=n: # on détermine la plus grande puissance de 10 plus
        petite que n
      k+=1
    for i in range(k):
        envers+=((n//10**i)-10*(n//10**(i+1)))*10**(k-1-i) #on récupère le
            i+1 eme chiffre de n et on le place en k-i eme chiffre dans
            envers

    return envers
```

2.
```
def divisioneuclidienne(a,b):
    q,r=0,a
    while r>=b:
        q+=1
        r=r-b
    print('quotient =',q,'reste =',r)
```

On étudiera dans de prochains TP des méthodes pour prouver nos algorithmes itératifs, mais la prochaine fois, nous mettrons en pratique tout ce que l'on a vu jusqu'à maintenant avec le module de dessin `turtle`