Chapitre I

Les bases de Python et de sa syntaxe

Table des matières

Partie A : Débuter avec Python **2**
1. Pratique du langage Python . . . 2
2. Installation . . . 2
3. L'environnement de développement . . . 3
4. Console et instructions de base . . . 3
5. L'éditeur . . . 7
6. Les bases du langage Python . . . 7

Partie B : Spécificités et syntaxe de Python **16**
1. Utilisation de l'éditeur . . . 16
2. Structuration de code . . . 17
3. Les fonctions . . . 18
4. Les structures conditionnelles . . . 25
5. Les boucles for/while . . . 29

Partie C : Mise en oeuvre avec les modules `random` et `turtle` **37**
1. Le module `random` . . . 37
2. Le module `turtle` . . . 39
3. T.P. n°1 : `random` et `turtle` . . . 47

Partie A

Débuter avec Python

En I.T.C. (Informatique Tronc Commun), les algorithmes que nous allons étudier seront implémentés grâce au langage **Python**. Python est un langage de haut-niveau - c'est-à-dire un langage qui permet à l'utilisateur d'écrire des programmes informatique en terme de ce qu'il veut faire (et non pas en terme de ce que la machine sait faire) et sa syntaxe est très naturelle (si, comme pour tout langage, on maîtrise un anglais basique).
Python a vu sa première version sortir en 1991 et a été créé par Guido Van Rossum.
Il s'agit d'un langage facile a appréhender qui permet de nombreuses possibilités. On étend ses fonctionnalités grâce à des bibliothèques (pour simplifier : des collections de fonctions déjà codées) qui facilitent le travail des développeurs : quand on travail sur un sujet précis, on ne veut pas avoir à coder des fonctions basiques comme par exemple le cosinus ou la partie entière !

Mais alors, à terme, que peut-on faire avec Python :

- Dans un premier temps, programmer des petits scripts qui peuvent automatiser des tâches sur notre ordinateur (renommer des fichiers en masse par exemple)
- Puis une fois très à l'aise, créer des logiciels, des jeux, etc...

1. Pratique du langage Python

Avant de voir comment installer, je vous conseille de commencer à vous exercer sur le langage Python via des sites spécialisés dans son apprentissage. Voici quelques propositions de site qui propose des parcours et tutoriels progressifs sur Python :

- France IOI - https://www.france-ioi.org/algo/chapters.php.
- Coursera - https://www.coursera.org/courses?query=python.
- Code Academy - https://www.codecademy.com/enrolled/courses/learn-python-3.

Il existe bien-sûr beaucoup d'autres ! Dans l'idéal, il faudra alterner entre la lecture de ce cours et la pratique sur ces tutoriels pour progresser efficacement.

2. Installation

Alors maintenant, comment installer et utiliser Python ?

Nous allons utiliser une distribution qui devra inclure :

- Python 3
- numpy : c'est une bibliothèque de fonctions pour effectuer des calculs numériques
- scipy : c'est une bibliothèque de fonctions pour effectuer du calcul scientifique
- matplotlib : c'est une bibliothèque de fonctions graphiques

et l'environnement de développement. Bon, dit comme cela, cela peut paraître très abrupte mais, pas d'inquiétude, on verra dans la suite ce que tout cela signifie.

En attendant, installons le nécessaire :

Le plus simple est d'intaller, sur votre ordinateur personnel, le tout-en-un EduPython. Son avantage est sa simplicité d'installation ; son désavantage... il n'est plus maintenu par ses auteurs. Mais, EduPython reste fonctionnel et suffisant pour l'usage qu'on en fera

https://edupython.tuxfamily.org/.

Pour ceux qui veulent avoir un environnement à jour et plus que complet, je vous recommande d'installer Anaconda qui vous proposera, après renseignement d'une adresse e-mail, une installation complète distribution Python récente + environnement (par défaut, Spyder - le même qu'EduPython). Attention, il faut 10Go d'espace disque pour l'installer ! Voici l'adresse :

`https://www.anaconda.com/download/success`.

Il existe beaucoup d'autres environnements de développement pour Python, par exemple Pyzo, Jupyter, Eric, PyCharm ou encore Spyder mais EduPython à quelques avantages intéressants, notamment, il fournit un environnement ET une distribution (c'est-à-dire l'interprète Python3 et une sélection de modules) : ainsi, notre environnement est fonctionnel en une seule installation !

3. L'environnement de développement

Un environnement de développement est composé de deux parties principales que nous allons étudier individuellement dans la suite :

— *La console interactive ou shell :* le code à évaluer est écrit dans l'interprète de commande après **l'invite de commande** ou **prompt** :

`>>>`

Le **résultat** (s'il y en a un) du code est affiché sur la ligne en dessous sans préfixe.
Attention : Toute commande inscrite via la console **ne peut pas être sauvegardée** ! On l'utilisera principalement pour effectuer des calculs ou des tests de parties de nos programmes mais **jamais** pour implémenter nos algorithmes !

— *L'éditeur :* il permet d'écrire des programmes et de les enregistrer dans un fichier. On peut alors exécuter le code inscrit dans l'éditeur par l'interpréteur Python : le résultat (s'il y en a un) de l'exécution est affiché directement dans la console interactive.

4. Console et instructions de base

a. Invite de commande

Après les informations relatives à la version de Python, l'interprète de commande commence par une invite de commande (ou prompt). Comme il a été dit plus haut, elle est symbolisée par `>>>`. On entre nos lignes de code une par une après chaque invite de commande en les validant avec la touche "Entrée". Le résultat (s'il y en a un) s'affiche alors sous chacune des lignes validées : on remarque que la ligne le contenant ne commence pas par l'invite de commande ; en effet, il s'agit d'un résultat, et non d'une instruction. Puis encore en dessous, une nouvelle invite apparaît, afin que l'on puisse de nouveau écrire une instruction.

On commence par répondre à la première invite `>>>` par `1+1` et en validant par "Entrée" ; on doit obtenir :

code Python

```
>>>1+1
2
>>>
```

L'interprète de commande a exécuté notre instruction et retourné le résultat de celle-ci. Une fois la tâche terminée, le prompt nous invite de nouveau à taper une instruction.

> **Astuce :** en utilisant les flèches du haut et du bas du clavier, on peut récupérer les précédentes instructions exécutées dans la console.

b. Effet de bord et résultat d'une instruction

Attention, il convient de bien différencier ce que peut produire une instruction donnée à Python. Essayons l'instruction suivante : `print('Hello world !')`. On obtient :

code Python

```
>>>print('Hello world !')
Hello world !
>>>
```

Du point de vue de l'affichage, on ne voit pas de différence avec le code précédent, pourtant il y en a une !

— l'instruction `1+1` a comme résultat `2` et n'a pas d'effet sur l'environnement,
— l'instruction `print('Hello world !')` a un résultat particulier qui n'est pas affiché (`None`) et a pour effet d'écrire une chaîne de caractère dans le shell.

Oulah, mais quelle est la différence entre résultat et effet sur l'environnement ?
Même si la console n'indique aucune différence, la voici :

Définition 1.

— un **effet de bord** est une interaction avec un périphérique de sortie ou une modification de la mémoire *(on le verra avec la déclaration de variable)* ;
— un **résultat** est le retour d'une instruction *(on s'y attardera plus longuement dans l'étude des fonctions).*

Pour s'en convaincre, testons les deux intructions suivantes :

- `1+(1+1)`
- `1+print(1+1)`

La première instruction nous retourne le réusltat 3 et la seconde affiche 2 puis une erreur ! En fait, la fonction `print` a bien affiché ce qu'on lui a demandé, à savoir 2 mais son résultat étant `None`, on a alors demandé le calcul : `1+None` ce que Python n'apprécie pas et nous verrons pourquoi dans la partie sur les types.
Cette distinction de comportement fera toute la différence quand nous définirons des fonctions : pour retourner un résultat utilisable par la suite, on n'utilisera jamais la fonction `print`.

c. Opérations arithmétiques de base en Python

Nous allons effectuer quelques tests dans la console en proposant comme instructions divers calculs arithmétiques simples.

Essayons les instructions suivantes ligne par ligne et observons le résultat.

code python

```
7+1
5-1
2*4
2**10
8/4
10/3
3//4
10//3
25%7
```

On comprend avec ces quelques exemples à quoi correspondent ces différents opérateurs :

Définition 2.

+	l'addition usuelle
-	la soustraction usuelle
*	la multiplication usuelle
**	l'exponentiation (puissance)
/	la division usuelle
//	le quotient division euclidienne
%	le reste de la division euclidienne

La priorité des opérations est la même qu'en mathématiques. Pour s'assurer de la priorité d'une opération par rapport à une autre, on utilise, toujours comme en mathématiques, les parenthèses.

Si on veut effectuer des calculs plus poussés, Python n'est pas armé par défaut pour permettre l'utilisation de nos fonctions usuelles comme le cosinus, le logarithme ou la racine. Pour cela, il faut faire appel à une bibliothèque (ou module) nommée `math` qui définit ces fonctions pour nous. Une fois importée grâce à l'instruction **`import`**, on peut faire usage d'une multitude de fonctions supplémentaires pour nos calculs.

On importe la bibliothèque `math` et on lui donne le surnom `m` pour simplifier nos futurs appels de fonctions de cette bibliothèque :

```
import math as m
```

Le nom des fonctions dans la bibliothèque `math` est volontairement très proche du nom de leur pendant mathématique : on trouve par exemple `exp`, `cos`, `tan`, etc... Il faut cependant faire référence à la bibliothèque avant l'appel de la fonction : il faut indiquer à l'interpréteur où aller la chercher puisqu'elle n'existe pas pour python à la base !

Voici comment calculer la racine de 8 grâce à la bibliothèque `math` (que l'on a surnommée `m`) :

```
m.sqrt(8)
```

On peut également importer un module en utilisant la syntaxe suivante **`from`** `nomdumodule` **`import`** `*` ; dans le cas de `math` cela donne :

```
from math import *
```

L'avantage de cet appel est que l'on a plus besoin du "surnom", on appelle directement les fonctions du module :

```
sqrt(8)
```

Ce code donnera le même résultat que le précédent calcul.

Mais alors pourquoi s'embêter avec les "surnoms", ça alourdi considérablement la syntaxe ! En effet, mais un souci peut se poser si on importe plusieurs modules : si deux modules contiennent des fonctions ayant le même nom, python ne saura pas faire la différence, et il considérera que c'est le dernier module appelé qui imposera sa fonction. En fait, la première fonction est "écrasée" par la deuxième portant le même nom.

On prendra donc l'habitude, quand on importe plusieurs modules, d'utiliser la première méthode pour ne pas avoir de mauvaises surprises...

Revenons au module `math`. Comment savoir quelles fonctions sont incluses dans ce module ? Pour tout connaître d'un module/bibliothèque, il suffit de taper `help("nomdumodule")` dans l'invite de commande. Donc pour connaître le module `math` (après l'avoir importé), on tape :

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

Voici un premier exercice pour appréhender les calculs numériques avec Python.

Exercice 1.

Écrire en Python les calculs mathématiques suivants et afficher le résultat :

1. le quotient et le reste de la division euclidienne de 10000 par 123 ;
2. $\sqrt{4064256}$;
3. $\cos(\frac{2\pi}{3})$

   Remarque : le nombre π est implémenté sous le nom `pi` dans le module `math` que nous avons surnommé `m` : on retrouve donc π en Python (enfin une approximation, bien sûr) avec `m.pi`.
4. $\mathrm{ch}(0)$
5. $3 \times 2^2 + \frac{5+11^4}{20} - (64-5)^{-2} + 3,8$ (Attention la virgule pour les nombres décimaux devient en point en Python).

Correction.

```
#1.
10000//123 #quotient
10000%123 #reste
#2.
4064256**(1/2) # ou sqrt(4064256) après avoir importé sqrt via le module math
#3
from math import cos, pi
cos(2*pi/3)
#4
from math import cosh
cosh(0)
#5
3* 2**2 +(5+11**4)/20-(64-5)**(-2)+3.8
```

5. L'éditeur

L'éditeur - la fenêtre à côté de la shell - permet d'écrire plusieurs lignes de code à la fois ! Mais celles-ci ne sont pas interprétées directement : il faut exécuter le code inscrit dans l'éditeur pour qu'il soit interprété dans le shell. Mais attention, quand on exécute un code provenant de l'éditeur, le résultat ne s'affiche pas dans la console, seuls les effets des instructions peuvent permettre d'y afficher quelque chose après exécution.
L'intérêt principal de l'éditeur n'est donc pas d'effectuer des calculs : il nous permettra d'écrire des programmes, qui sont des successions d'instructions ; et surtout, il nous permettra d'enregistrer ces programmes : l'éditeur offre la possibilité de sauvegarder dans un fichier d'extension .py (pour être directement reconnu et utilisable comme module par Python). C'est ce que nous ferons lorsque nous écrirons nos premiers programmes.

Nous étudierons plus en détail l'éditeur dans la prochaine partie.

6. Les bases du langage Python

Dans les langages de programmation, le concept de **type** est très important. En Python, tous les objets (entiers, fonctions,...) ont un type : ce type caractérise la manière dont l'objet est représenté en mémoire. Le type représente la nature de l'objet : c'est essentiel pour l'ordinateur de savoir quelle est la nature des objets qu'il manipule. Pour nous, humains, on sait très bien qu'on ne peut pas ajouter une pomme et une voiture : ces objets sont de type complètement différents. Mais pour l'ordinateur, tous les objets sont représentés en mémoire par une suite de 0 et de 1 : il pourrait ajouter sans problème deux telles suites, et ce, qu'elles représentent de la musique et une image par exemple !
C'est pourquoi la notion de type est essentielle : elle permet à l'ordinateur de différencier les objets manipulés, et en cas de mauvaise manipulation par l'utilisateur, au lieu d'ajouter inutilement des objets de natures différentes, il lui renvoie une erreur !
Testons le type de différents objets pour Python grâce à la fonction ... `type` et observons le résultat :

```
type(12)
type('Bonjour monde !')
type(3.7)
type(4+3j)
```

```
type(None)
type(m.pi)
type(m.sqrt)
type((1,2))
type([1,2])
type({1,2})
type({'a':1,'b':2})
type(type)
```

Nous verrons les types `list` et `builtin_function_or_method` plus tard même s'il est aisé de comprendre quelle est la nature des objets qui ont ces types. Intéressons nous aux types des nombres :

a. Les types numériques en Python

Python reconnaît trois types de nombres :

Définition 3.

- Le type `int` : les nombres entiers tels que 6 par exemple ;
- Le type `float` : les nombres décimaux tels que 6.12 par exemple ;
- Le type `complex` : les nombres complexes tels que $5 + 3i$ par exemple ;

Un même nombre, 3 par exemple, a trois représentations mémoires différentes selon son type : 3 pour le type `int` ; 3.0 pour le type `float` et $3 + 0j$ pour le type `complex`.

Dans certains langages, dits fortement typés, il est impossible de faire une opération avec des nombres de types différents !! En Python, heureusement, ce n'est pas le cas : si on multiplie un `int` et un `float` par exemple, Python aura "la présence d'esprit" de convertir les types automatiquement. La conversion se fait toujours de `int` vers `float` vers `complex`. Observons sur ces exemples :

```
2+2
2+2.0
2+(2+0j)
2.0+(2+0j)
```

On peut également demander manuellement demander la conversion d'un type de nombre vers un autre. Testons les codes suivant et voyons ce qui est possible ou pas :

```
float(34)
int(45.97)
complex(3)
complex(2.3)
float(2+0j)
```

b. Les Booléens

Comme en Mathématiques, la logique est un aspect essentiel de l'Informatique. En Python, il existe un type pour désigner les "V" (=vrai), "F" (=faux) utilisés en logique mathématique. Il s'agit du type

`bool` et ce type ne contient que deux objets : `True` et `False` : **Vrai** et **Faux** donc ! Trois opérateurs sont associés à ce type : `not`, `or` et `and` qui correspondent exactement au "*non*", "*ou*" et "*et*" logique. On a donc par exemple : `not True = False` et `not False = True` - Faisons le test dans le shell !

Exercice 2.

Donner les tables des opérations `or` et `and` sur les deux objets de type `bool`.

Correction.

Table de `or`

```
>>>True or True
True
>>>True or False
True
>>>False or True
True
>>>False or False
False
```

Table de `and`

```
>>>True and True
True
>>>True and False
False
>>>False and True
False
>>>False and False
False
```

Il existe d'autres opérateurs, dit opérateurs de comparaison, qui permettent de comparer de objets de type similaire et qui renvoient un objet de type `bool` : `True` ou `False`. Les voici :

Définition 4.

Soit `o`, `o'` des objets de même type (ou du moins de types qui peuvent être convertis vers l'un ou l'autre)

`o == o'`	retourne `True` si `o`=`o'` ; `False` sinon
`o >= o'`	retourne `True` si `o`$\geq$`o'` ; `False` sinon
`o <= o'`	retourne `True` si `o`$\leq$`o'` ; `False` sinon
`o > o'`	retourne `True` si `o`$>$`o'` ; `False` sinon
`o < o'`	retourne `True` si `o`$<$`o'` ; `False` sinon
`o != o'`	retourne `True` si `o`$\neq$`o'` ; `False` sinon

Testons tout ceci :

```
4+5 == 3+6
5 == 3
2+1 == 3 or 6>=8
'avion'>='fusee'
'a'<'b' and not 'd'<'c'
(1==2) != (2==2)
```

c. Les Variables

Pour mettre nos données en mémoire, nous aurons besoin des **variables**. Une variable est définie par un nom auquel on associe une valeur grâce à l'opérateur d'affectation =. Attention ! = n'a pas la même signification que == :

Définition 5.

- le = permet de **définir** ou **redéfinir** une variable en lui associant une valeur ;
- le == est un "test" : c'est, comme on l'a vu plus haut, un opérateur qui à deux objets associe une booléen : `True` s'ils sont égaux ; `False` sinon.

Une fois déclarée, on peut récupérer la valeur d'une variable en faisant appel au nom de cette variable. Voyons comment cela fonctionne sur un exemple - on remarquera au passage que la casse est importante, une minuscule est différente d'une majuscule dans le nom d'une variable :

```
l = 5
L = 12
p = 2*(5+12)
print("Le périmètre d'un rectangle de largeur",l,"et de longueur",L,"est égal à",p)
```

Essayons de comprendre la ligne suivante en observant le résultat de l'appel de la variable :

```
L=L*2
L
```

On remarque que lors de l'affectation d'une nouvelle valeur à une variable, Python garde en mémoire (et c'est le cas de le dire) l'ancienne valeur de la variable et n'associe la nouvelle qu'après avoir effectué les calculs ! C'est extrêmement pratique !

Au niveau de la syntaxe, on a des écritures raccourcis pour certaines déclarations auto-référentes, ce qui permet un gain de temps lors de l'écriture de nos programmes :

- `L=L*2` est équivalent à `L*=2`
- `L=L+1` est équivalent à `L+=1`
- `L=L/4` est équivalent à .....

— `L=L*2` est équivalent à .....

Petit problème concernant les variables :

Exercice 3.

On déclare deux variables `x` et `y`. Donner une suite d'instructions qui permet d'échanger leur valeur.

Correction.

```
>>>z=x
>>>x=y
>>>y=z
```

On verra grâce au type **`tuple`** que Python permet aisément ce type d'affectations simultanées sans perdre de place en mémoire avec l'introduction de variables annexes.

d. Les chaînes de caractères

Comme on l'a vu pendant nos tests de la fonction **`type`**, les données alphanumériques - c'est-à dire des suites de lettres/nombres/symboles - sont de type **`str`** en Python. On appelle ce type de données **chaînes de caractères**. En Python, il existe plusieurs moyens de définir une chaînes de caractères ; voici les deux principaux :

Pour définir une chaînes de caractères en Python, on encadre les caractères voulus par :

— des guillemets simples `'` (touche 4 du clavier en mode minuscule), ou
— des guillemets doubles `"` (touche 3 du clavier en mode minuscule).

D'accord, mais comment choisir ? Et bien cela dépend du contenu de la chaîne considérée : si celle-ci contient un symbole `'`, on l'encadrera par des guillemets doubles `"`, et réciproquement :
Testons les codes suivants dans la console :

```
"Le TP d'Informatique"
'est vraiment "cool"'
'n'est-ce pas ?'
```

Que se passe-t-il lorsqu'on exécute la troisième ligne ? Comment l'éviter ?

D'accord, mais si ma chaîne de caractères contient les deux sortes de guillemets ? Alors on les *échappe* grâce au *caractère d'échappement*, le *backslash* : `\`.

Testons :

```
'n\'est-ce pas ? "oui !"'
"\"Citation\""
```

Le backslash sert également à définir des caractères spéciaux comme le `\n` qui est interprété comme un passage à la ligne :

```
print("Allez, on passe\n à la ligne")
```

On peut réaliser diverses opérations sur les chaînes de caractères. Testons les instructions suivantes pour déterminer le comportement des opérations sur les chaînes.

Au passage, vous remarquerez l'utilisation dans le code suivant du caractère `#` suivi d'une explication : il s'agit d'un commentaire du code en Python. Tout se qui se trouve après un `#` ne sera pas interprété par Python. Lorsque nous écrirons des programmes dans l'éditeur, nous prendrons l'habitude de commenter nos instructions de manière à rendre compréhensible (sans trop s'appesantir) nos différentes lignes de code pour une autre personne (ou pour soi-même pour une future relecture).

```
'Infor'+'matique'
s="Sainte Croix "
s+="Saint Euverte " # équivalent à s=s+"Saint Euverte "
print(s)
s*5                 #équivalent à s+s+s+s+s
```

L'opération `+` pour des chaînes de caractères n'a pas la même signification que pour les objets de type `int` pour lesquels il représente l'addition.

Définition 6.

Pour les chaînes de caractères, comme on a pu s'en rendre compte grâce aux instructions précédentes, le symbole `+` correspond à la **concaténation** de deux chaînes i.e. `+` met bout-à-bout les deux chaînes qu'on lui donne.

On comprend mieux la différence entre la concaténation de chaînes et l'addition de nombres grâce aux tests suivants :

```
'78'+'83'
78+83
'78'+83    #produit une erreur : les types sont incompatibles pour +.
```

On peut accéder aux caractères d'une chaîne individuellement : chaque caractère est accessible grâce à son rang (son indice) dans la chaîne. On utilise les crochets pour désigner le rang d'un caractère dans un chaîne.

Attention : l'indice du 1er caractère d'une chaîne est 0 ! L'indice du 2eme caractère est 1, etc...

Testons :

```
chaine="Mathématiques Supérieures"
chaine[0]                                   # retourne le 1er caractère de chaine, ici "
    M"
chaine[6]                                   # retourne le 7eme caractère de chaine, ici
    "a"
chaine[13]                                  # un espace compte comme un caractère !
chaine[14]+chaine[15]+chaine[16]            # concatène les 15,16 et 17eme caractères
chaine[0]+chaine[1]+chaine[18]+chaine[14]
```

On peut aussi accéder aux caractères dans le sens inverse de la chaîne (i.e. de droite à gauche) en utilisant des indices négatifs : la dernier caractère a pour indice -1 ; l'avant dernier -2 ; etc...

```
chaine[-1]                # retourne le dernier caractère de chaine, ici "s"
chaine[-5]                # 5eme caractère de chaîne en partant de la droite, ici "e"
chaine[-25]*2+chaine[-11]
```

Exercice 4.

En utilisant la variable `chaine="Mathématiques Supérieures"`, écrire des instructions qui permettent de renvoyer les chaînes de caractères suivantes :

1. `"Maths"` ;
2. `"Ma ma"` ;
3. `"Super"` ;
4. `"Maaaaaaaaaaaaaaaaaaaaaaaaaths"` ;

Correction.

```
#1.
chaine[0]+chaine[1]+chaine[2]+chaine[3]+chaine[-1]+chaine[1]
#2.
chaine[0]+chaine[1]+chaine[13]+chaine[5]+chaine[6]
#3.
chaine[14]+chaine[15]+chaine[16]+chaine[-2]+chaine[-3]
#4.
chaine[0]+chaine[1]*25+chaine[2]+chaine[3]+chaine[-1]+chaine[1]
```

On peut également accéder à la longueur d'une chaîne c'est-à-dire le nombre de caractères dans la chaîne grâce à la fonction **`len`** :

```
len(chaine) # retourne la longueur de chaine
```

Exercice 5.

Expliquer ce que va retourner l'instruction suivante pour une chaîne de caractères affectée à une variable `chn` :

```
chn[len(chn)-1]
```

Correction.

Cette instruction renvoie le dernier caractère de la chaîne `chn` : en effet, comme il y a exactement `len(chn)` caractères dans `chn`, leurs indices vont de 0 à `len(chn)-1`.
Remarque : cette instruction est donc équivalente à `chn[-1]`

Définition 7.

On peut aller plus loin dans l'accès individuel aux caractères : on utilise la technique du **slicing** (découpage en tranches) pour extraire une suite de caractères dans une chaîne.

On doit préciser l'indice de début `m` et l'indice de fin `n` de la tranche à extraire, séparer par un double point `:`. Attention, l'indice `n` de fin est toujours **exclu** de la tranche ! Ainsi le nombre de caractères affichés par la tranche `[m:n]` est égal à `n-m`.

```
chaine[5:8]  # retourne du 6eme (indice 5) jusqu'au 8eme (indice 7) caractère
chaine[1:-1] # retourne du 2eme (indice 1) jusqu'à l'avant dernier (indice -2)
    caractère
```

De plus, si l'un des indices de début ou de fin de tranche n'est pas indiqué, Python considérera que tous les indices avant (ou après selon le cas) sont demandés. Observons :

```
chaine[3:] # retourne la chaîne du 4eme (indice 3) jusqu'au dernier caractère
chaine[:8] # retourne la chaîne du 1er jusqu'au 8eme (indice 7) caractère
chaine[:4]+chaine[-1]+chaine[13:17]
```

Exercice 6.

On considère la variable `chn="J'aime pas l'Info !"`. En utilisant le slicing sur la variable `chn`, écrire une instruction qui renvoie la chaîne de caractères :

`"J'aime l'Info "`

Correction.

```
chn = "J'aime pas l'Info"
chn[:6]+chn[-7:] #ou encore chn[:6]+chn[10:]
```

Exercice 7.

Comment retourner les 5 derniers caractères de `chaine` ? Ou plus généralement, comment afficher les k derniers caractères d'une chaîne ?

Correction.

```
chn[-5:] #ou encore chn[len(chn)-5:]
chn[-k:] #ou encore chn[len(chn)-k:]
```

Et pour finir notre étude de l'accès individuel des caractères d'une chaîne, on notera une dernière méthode de slicing plus poussée :

> `chaine[m:n:p]` retourne tous les caractères dont les indices sont compris entre `m` (inclus) et `n` (exclu) et séparés d'un pas de `p`.

Par exemple :

```
chaine[2:9:3] # retourne la chaîne du 3,6 et 9eme caractères (indices 2,5,8)
chaine[:14:5]
chaine[::2]   # expliquer cette instruction !
```

En mettant une valeur `k` de pas négative, on peut inverser simplement la chaîne de caractères :

```
chaine[::-1] # Inverse l'ordre des caractères de chaine
```

Partie B

Spécificités et syntaxe de Python

Dans la partie suivante, nous allons voir revoir 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 revoir 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 ! L'exécution du fichier entier sur Edupython a pour effet de réinitialiser toutes les variables préalablement définie !
Si on veut exécuter tout notre fichier en cours sans réinitialiser les variables prédéfinies, 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 :

- 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** - *raccourci clavier très utile : Ctrl+S* - 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 de 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...). 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

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

```
4     instruction 3
```

Deux blocs imbriqués

```
1 entete1:
2     instruction 1.1
3     instruction 1.2
4     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. Les fonctions

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 :

```
def bonjour(prenom):
    """ Fonction qui renvoie la chaîne de caractères 'Bonjour prenom ! Comment vas-tu
        ?' où l'argument prenom est une chaîne de caractères"""
    resultat='Bonjour '
    resultat+=prenom
    resultat+=' ! Comment vas-tu ?'
    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), grâce au mot-clé **`return`**, 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 !

```
>>>bonjour('Charles')
'Bonjour Charles ! Comment vas-tu ?'
```

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

```
>>>bonjour('Jean')
'Bonjour Jean ! Comment vas-tu ?'
```

Remarque.

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 !

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

Exercice 8.

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

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

Correction.

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

Très important ! Pour **retourner** un résultat, on utilisera quasiment exclusivement le mot-clé `return` suivi de l'objet à retourner et non la fonction `print`.
Comme expliqué dans le premier chapitre, la fonction `print` affiche ce qu'on lui donne mais ne renvoie aucun résultat - en fait, elle renvoie l'objet `None` qui symbolise le vide, l'absence de valeur.
A l'inverse, le mot-clé `return` permet de renvoyer la valeur qui le suit lors de l'appel d'une fonction.
Remarque : `return` n'est pas une fonction, on le separera de la valeur à renvoyer par un espace et on ne mettra donc pas de parenthèses.
Voici un exemple de ces comportements :

Dans l'éditeur

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

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

Dans la console

```
>>>1+f(3)
6
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
>>>1+g(3)
7
```

on pourra remarquer que l'objet None cité dans l'erreur est renvoyé par la fonction f et non pas par la fonction print : en l'absence de retour indiqué, une fonction renvoie None par défaut

On peut faire le parallèle avec une fonction Mathématiques : pour la fonction $g : x \mapsto 2x$, $g(3)$ **vaut** 6 de même qu'ici `g(3)` vaut 3 car elle renvoie 3 et `f(3)` ne vaut rien car elle ne renvoie rien.

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 $(0, 0)$:

```
1 from numpy import sqrt    # On a besoin de la fonction racine carrée ici
2
3 def dist(x,y):
4   """ Calcule la distance euclidienne entre (x,y) et (0,0) où x,y sont des int ou des
        float """
5   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 9.

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 $\mathbb{R}^2$.

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 10.

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 :

```
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 :

```
L=2

def f():
  x=L
  return x
```

```
>>>f()

>>>x
```

Question.

Que se passe-t-il à l'appel de `x` ?

```
L=2

def f2():
  L=5 # définition locale de L
  return L
```

```
>>>f2()

>>>L # récuperation de la valeur globale définie précédemment !
```

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 :

```
def f3():
  global L
  L=100
  return L
```

```
>>>f3()

>>>L
```

d. Appel de fonction par valeur

En Python, pour une instruction du type `f(x)` où `f` est une fonction et `x` est une variable (*non mutable*), l'exécution de celle-ci :

- évalue d'abord `x` ;
- exécute `f` en lui passant en argument la valeur calculée (de `x`)

On désigne de comportement par **l'appel d'une fonction par *valeur***. Observons ce comportement sur un exemple :

```
y=0

def f(x):
    x=1
    return x
```

```
>>>f(y)
```

```
>>>y
```

Question.

Que se passe-t-il à l'appel de `y` ?

Par contraste, d'autres langages de programmation choisissent d'effectuer les appels de fonctions par *variable* : dans le code précédent, l'instruction `y` aurait renvoyé la valeur `1`.
À noter que, en Python, si on passe un objet *mutable* en argument à une fonction, le comportement est différent comme on le verra avec le type `list`.

e. Annotations de type

Comme on l'a déjà vu, la notion de type est très importante pour les langages informatiques afin de savoir différencier les données.
Mais selon les langages, le typage est géré de manière propre : en particulier, Python est un langage dont le typage est dit **dynamique** c'est-à-dire que l'interpréteur détermine le type *à la volée* de l'exécution du code.
Dans un langage au typage statique (par exemple C ou Java), c'est le programmateur qui doit définir dans le code le type de ses variables.

Ainsi, en Python, lorsqu'on lit une portion de code où apparait une variable, on ne peut pas savoir quel est son type si on ne voit pas où elle a été définie (mais le contexte peut tout de même aider à le deviner !)
Encore pire, lorsqu'on lit la définition d'une fonction, on ne peut pas savoir le type des arguments que l'auteur du code a voulu leur donner !
Pour compenser ce potentiel problème, Python autorise ce qu'on appelle les **annotations de type** pour faciliter la lecture des fonctions d'un code tierse (ou même de son propre code !) Voyons la syntaxe des annotations de type sur un exemple :

Fonction avec la syntaxe usuelle

```
def machin(x,m):
    if len(m)==x:
        return 'youpi'
```

la même fonction avec annotations de type

```
def machin(x:int,m:str) -> str:
    if len(m)==x:
        return 'youpi'
```

Certains concours utilisent les annotations dans leur sujet (Centrale par exemple), il est ainsi bon de s'y habituer. Par contre, ce n'est pas une exigence du programme donc vous n'êtes pas contraints de les utiliser dans vos rédactions de code.

4. Les structures conditionnelles

a. `if ... else`

Une des plus simples structures conditionnelles, 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 **affiche** '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):
    """ Affiche '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:
        print('oui, ce nombre est plus grand que 100')
    else:
        print("non, ce nombre n'est pas plus grand que 100")
```

Testons cette fonction :

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

Exercice 11.

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

Correction.

```
def div(n,m):
    if n%m==0:
        print('oui',n,'est divisible par',m)
    else:
        print('non',n,"n'est pas divisible par",m)
```

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 12.

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

Correction.

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

Exercice 13. *(*)*

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
```

c. Caractère paresseux de `and` et `or`

Les opérateurs **`and`** et **`or`** sont dits "paresseux" : cela signifie qu'ils utilisent les particularités des tables de véracité du "ET" et du "OU" pour eviter d'évaluer une expression pour rien et même mieux, éviter d'évaluer une expression qui produirait une erreur.
Voyons exactement le principe :

— Dans la table du "ET", on remarque que dans l'expression (P ET Q), si P est fausse, peu importe la valeur de Q, (P ET Q) est fausse.
Ainsi, pour une expression `P and Q`, si le booléen `P` vaut `False`, alors `P and Q` sera évalué à `False` **sans que le booléen `Q` ne soit évalué**.
— De même pour "OU", on remarque que dans l'expression (P OU Q), si P est vraie, peu importe la valeur de Q, (P OU Q) est vraie.
Ainsi, pour une expression `P or Q`, si le booléen `P` vaut `True`, alors `P or Q` sera évalué à `True` **sans que le booléen `Q` ne soit évalué**.

Voyons l'utilité de ce caractère paresseux sur un exemple :

```
def f(x):
    if type(x)==int and x+2==2:
        print('x vaut 0 !')
    else:
        print('oups !')

def g(x):
    if x+2==2 and type(x)==int:
        print('x vaut 0 !')
    else:
        print('oups !')
```

```
>>>f('truc')

>>>g('truc')
```

On voit ainsi que contrairement à leur pendant mathématiques, `and` et `or` ne sont pas commutatifs ! Nous utiliserons assez régulièrement la "paresse" de ces opérateurs lorsque nous aborderons les listes.

5. Les boucles for/while

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 :

Définition 8.

— 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. Boucles `for` et fonction `range`

Avant de parler de la boucle `for`, on va s'intéresser à une 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 convertir en liste (nous verrons exactement ce que sont les listes et comment les manipuler dans le prochain chapitre). Testons les codes suivants (directement dans le shell) :

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

```
for ... in range(...):
    instruction1
    instruction2
    .
    .
    .
```

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 14.

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 15. *(*)*

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
```

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 :

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

Exercice 16. *(*)*

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. Boucles 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 !')
```

Question.

Que se passe-t-il et pourquoi ?

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

Question.

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 17. *(*)*

1. Écrire une fonction `puissance2(n)` qui prend en argument un entier strictement positif `n` et qui **renvoie** la plus grande puissance de 2 plus petite ou égale à `n`.
2. Écrire une fonction `divisioneuclidienne(a,b)` qui prend en arguments deux entier `a,b` et qui **renvoie** le quotient et le reste d'une division euclidienne de `a` par `b` (bien-sûr, sans utiliser `//` ni `%`).
3. Écrire une fonction `valuation(p,n)` qui prend en arguments deux entiers strictement positifs `p,n` et qui **renvoie** la valuation `p`-adique de `n` i.e. l'exposant de la plus grande puissance de `p` qui divise `n`

Correction.

1.
```
def puissance2(n):
    k=0
    while 2**k<=n:
        k+=1
    return 2**(k-1)
# ou utilisant moins de "puissance" :
def puissance2(n):
    m=1
    while m<=n:
        m=m*2
    return m//2
```

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

3.

```python
def valuation(p,n):
    m=n
    k=0
    while m%p==0:
        m=m//p
        k+=1
    return k
```

Exercice 18.

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.

```python
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.

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).

Partie C

Mise en oeuvre avec les modules `random` et `turtle`

Dans cette partie, nous allons mettre en pratique ce que nous avons vu précedemment au travers d'un T.P. en nous appuyant sur deux modules : `random` et `turtle`.
Avant de manipuler, commençons par découvrir les bases de ces modules.

Rappelons les manières d'importer un module :

```
>>>import nom_du_module as mod # imporation avec un surnom
>>>from nom_du_module import * # imporation sans surnom
>>>from nom_du_module import f1, f2 # imporation de fonctions spécifiques
```

1. Le module `random`

```
>>>from random import *
```

Le module `random` est un module qui permet de générer des nombres "pseudo"-aléatoires en Python. Ce module fournit plusieurs fonctions ; nous n'en présenterons ici que deux principales :

a. La fonction `random()`

La première fonction que nous allons voir est éponyme du module et "engendre" la plupart des fonctions incluse dans `random` :

> **Fonction** `random()` ⇝ `random() -> float`
> Cette fonction ne prend par d'argument et retourne un `float` "aléatoire" uniformément dans l'intervalle $[0, 1[$.

Testez plusieurs fois l'instruction suivante dans la console :

```
>>>random()
```

b. La fonction `random()`

La seconde, qui "découle" de la première (nous allons d'ailleurs la programmer dans un prochain exercice) permet d'obtenir un entier "aléatoire" compris entre deux bornes entières :

> **Fonction** `randint(a,b)` ⇝ `random(a:int, b:int) -> int`
> Cette fonction prend pour arguments `a` et `b` tous deux de type `int` et retourne un `int` "aléatoire" uniformément entre `a` et `b` tous deux inclus.

Testez plusieurs fois l'instruction suivante dans la console :

```
>>>randint(1,6)
```

c. Exercices

Exercice 19.

1. Ecrire une fonction `bernouilli(p)` qui prend en argument un **`float`** dans l'intervalle $[0,1]$ et qui renvoie `True` avec probabilité `p` et `False` avec probabilité $1-$`p`. *On aura alors modélisé une variable aléatoire suivant une loi de Bernouilli de paramètre* `p`.
2. Ecrire une fonction `binomiale(n,p)` qui prend en arguments `n` de type **`int`** et `p` de type **`float`** compris dans l'intervalle $[0,1]$ et qui renvoie le nombre de fois où l'instruction `bernouilli(p)` renvoie `True` après `n` appels de celle-ci. *On aura alors modélisé une variable aléatoire suivant une loi de binomiale de paramètres* `n`,`p`.
3. On joue au jeu suivant : on lance 100 fois une caillou qui présente deux faces : une rouge et une bleue. Après les 100 lancers, on gagne autant d'euros qu'on a obtenu de fois la face bleue. Ce caillou n'est pas vraiment régulier et il a une chance sur 20 de tomber sur la face bleue.
   Déterminer "numéricempiriquement" quel est le gain moyen lorsqu'on joue à ce jeu.

Correction.

1.
```
def bernouilli(p):
    return random()<p
```
2.
```
def binomiale(n,p):
    S=0
    for k in range(n):
        if bernouilli(p):
            S+=1
    return S
```
3.

```
def gain_moyen(n_tentatives):
    G=0
    for k in range(n_tentatives):
        G+=binomiale(100,1/20)
    return G/n_tentatives
```

Exercice 20.

Ecrire une fonction `monrandint(a,b)` qui reproduit le comportement de la fonction `randint` en utilisant la fonction `random` du module `random` (et aucune autre dans ce module bien-sûr).

Correction.

```
def monrandint(a,b):
    return int(random()*(b-a+1)+a)
```

2. Le module `turtle`

Nous allons utiliser le module nommé `turtle` qui va nous permettre, avec des instructions minimalistes, de tracer des dessins séquentiellement dans une fenêtre indépendante.

Commençons par importer le module en question :

```
>>>from turtle import *
```

a. Premiers tests du module

Maintenant, il nous faut ouvrir la fenêtre dans laquelle nous allons tracer nos dessins : tapons puis exécutons la commande suivante, toujours dans le shell :

```
>>>reset()
```

Ainsi, une nouvelle fenêtre dont le contenu est blanc s'affiche avec un curseur noir en son centre : ce point représente la "tortue" (oui, il faut de l'imagination !) qui va, en se déplaçant sur nos instructions - c'est le cas de le dire - laisser sa "trace". La tortue est située au point de coordonnées $(0,0)$ de la fenêtre et sa tête est dirigée vers la droite : elle forme un angle de $0°$ par rapport à l'horizontale.

Testons, dans la console, les commandes suivantes, en les exécutant une par une - afin de bien comprendre ce qu'elle produisent au niveau de la tortue :

```
>>>forward(100)
>>>left(120)
>>>forward(100)
>>>left(120)
>>>forward(100)
>>>circle(100)
>>>circle(200,45)
>>>forward(-100)
>>>right(60)
```

Que produisent ces instructions prises séparément dans la fenêtre de dessin ?

b. Commandes de base

Tracés basiques :

- La fonction `forward(distance)` permet de déplacer du nombre de pixels indiqué en argument la tortue dans la direction indiquée par sa tête. Par exemple, `forward(123)` fera avancer la tortue de 123 pixels dans la direction indiquée par la tête au moment de l'exécution.
  En indiquant un nombre de pixels négatif en argument de la fonction `forward`, la tortue reculera du nombre de pixel en valeur absolue (toujours dans la direction indiquée par la tête).
- La fonction `left(angle)` faire pivoter la tortue de l'angle **en degré** indiqué en argument dans le sens trigonométrique. Par exemple, `left(60)` fera pivoter la tortue de 60° dans le sens trigonométrique.
  De même, en indiquant un nombre négatif en argument à la fonction `left`, la tortue pivotera de ce nombre de degré en valeur absolue mais dans le sens horaire. Remarque, on peut obtenir le même effet grâce à la fonction... ... `right` !
- La fonction `circle(rayon,extent=None)` permet de tracer un cercle, voici ses 2 arguments :
  - Le premier argument de `circle` est le rayon en pixel du cercle à tracer à partir de la tête de la tortue ; si le rayon est positif, le tracé se fait dans le sens trigonométrique, s'il est négatif, dans le sens horaire. Ainsi, la commande `circle(100)` tracera dans le sens trigonométrique, un cercle de rayon 100.
  - Le deuxième argument nommé `extent` est facultatif et prend pour valeur par défaut `None` (qui est équivalent à fournir l'argument 360). Cette argument permet d'indiquer la portion de cercle à tracer à partir de la tête de la tortue : la commande `circle(100,60)` tracera un arc de cercle de rayon 100 de l'angle 0° à partir de la tête, jusqu'à un angle de 60°.

Réinitialiser la "tortue" :

Pour réinitialiser la fenêtre de dessin et replacer la tortue dans sa position initiale, il suffit d'employer la commande qui nous a permis d'ouvrir la fenêtre de dessin :

```
>>>reset()
```

Changer la vitesse de la tortue :

Petite astuce concernant notre pauvre tortue qui est très lente par nature ! On peut lui donner des vitamines afin qu'elle puisse tracer plus vite nos dessins (car il deviendront de plus en plus long à tracer sinon) :

La fonction `speed` permet de modifier la vitesse de tracé de la tortue. L'argument de la fonction `speed` est un entier allant de 0 à 10.

- 1 correspond à la vitesse minimale possible pour la tortue ;
- 6 correspond à la vitesse normale de la tortue (la vitesse par défaut) ;
- 10 correspond à la vitesse presque maximale de la tortue ; et
- 0 correspond à de la salade radioactive : la vitesse de la tortue est cette fois-ci, vraiment maximale !

On peut bien-sûr moduler la vitesse entre ces valeurs selon nos convenances.

Voici un exemple de changement de vitesse :

```
1 >>>reset()
2 >>>speed(1) # la tortue est très lente
3 >>>forward(300)
4 >>>speed(0) # la tortue est extrêmement rapide
5 >>>forward(300)
```

ATTENTION ! La vitesse est réinitialisée après chaque appel de `reset()`. N'oubliez pas de spécifier de nouveau la vitesse qui vous convient après une réinitialisation de la fenêtre de dessin !

c. Options avancées

Le module `turtle` fait partie de la bibliothèque standard de Python : on trouvera la liste exhaustive des options, fonctions etc... du module sur la documentation officielle. Mais voici une sélection d'outils de personnalisation intéréssants :

Arrêter le tracé / le reprendre // Pour que la tortue stoppe puis reprenne son tracé lors de ses déplacemements, on utilise les méthodes `penup()` et `pendown()`

Exemple :

```
forward(100) #la tortue se déplace et trace
penup() #on lève le stylo
forward(100) #la tortue se déplace et NE trace PAS
pendown() #on pose le stylo
forward(100) #la tortue se déplace et trace de nouveau
```

d. Ouvrir / fermer une fenêtre de dessin

Pour ouvrir une fenêtre vide et modifier sa taille, on utilisera la classe `Screen` et ses méthodes :

```
S=Screen()
S.setup(500,500) #largeur,hauteur de la fenêtre en pixels
```

Pour fermer propremement une fenêtre de dessin, on utilise la méthode `bye()`

```
S.bye()
```

Plusieurs tortues

Pour créer une nouvelle tortue, on emploie l'instruction `Turtle()` et pour la manipuler, on affecte cette tortue à une variable qui permettra de l'identifier lors des instructions de mouvement ou de remise à zéro :

```
t1 = Turtle()
t1.forward(100)
t1.reset()
t1.speed(1)
t1.penup()
t1.forward(50)
```

Puis on peut introduire autant de tortues qu'on le souhaite dans notre fenêtre en affectant des `Turtle()` à d'autres variables :

```
t2 = Turtle()
t2.forward(100)
t1.forward(-20)
t2.reset()
```

Attention : les instructions `forward(100)`, `reset()`, `speed()`, etc... sans indiquer de tortue (comme nous l'avions fait jusqu'à présent en fait !) ne concernent que la tortue qui s'affiche lors de l'ouverture d'une fenêtre avec un `reset()` simple.

Moralité :
- Quand on ne manipule qu'une seule tortue, on peut faire comme nous l'avions fait depuis le début : démarrer avec un simple `reset()` puis manipuler notre unique tortue ;
- Quand on veut manipuler plusieurs tortues, on ouvre une fenêtre vide avec `S=Screen()` puis on crée nos tortues une par une en les affectant à des variables.

On peut également cacher nos tortues si elles deviennent "génantes" puis les faire réapparaître :

```
t3 = Turtle()
t3.hideturtle() #on fait disparaître la tortue
t3.forward(100) #mais elle continue de tracer !
t3.showturtle() #elle réapparait !
```

Couleur et forme de la tortue
Maintenant que l'on a plusieurs tortues, on veut les différencier : on va donc les personnaliser !

Couleur

```
t=Turtle()
t.color('red') #la tortue et le tracé sont rouges
t.forward(100)
t.color(0.2,0.5,0.6) #couleurs RVB
```

Leur forme : les différentes valeurs sont "arrow", "turtle", "circle", "square", "triangle" ou "classic". Même si les noms sont représentatifs, il faut tester pour se rendre compte !

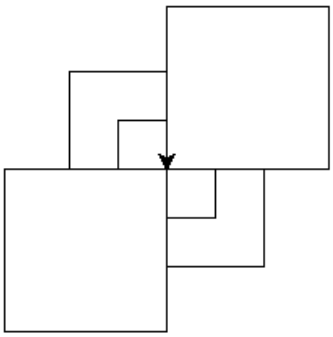
Forme

```
t.shape('turtle') #forme de tortue (enfin !)
```

e. Exercices basiques

Exercice 21.

1. Programmer une fonction `carre` d'argument un entier `n` qui permette de tracer un carré de côté `n` pixels et qui replace la tortue dans sa position précédente (avant le début du tracé du carré) à la fin du tracé. ATTENTION : pour cette fonction, on ne veut pas réinitialiser la fenêtre de dessin avant le tracé du carré, on n'utilisera pas la fonction `reset()` DANS la fonction.
2. Reproduire le dessin suivant en utilisant plusieurs fois la fonction `carre` :



Correction.

1.
```
def carre(n):
    """ Trace un carré de côté de longueur n pixels et replace la tortue
        en position précédente """
    forward(n)
    left(90)
    forward(n)
    left(90)
    forward(n)
    left(90)
    forward(n)
    left(90)
```

```
reset()
carre(100)
left(90)
carre(30)
carre(60)
left(90)
carre(100)
left(90)
carre(60)
carre(30)
```

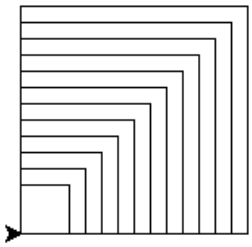
2.

Désormais, utilisons ce que nous avons appris à propos des structures itératives (les boucles `for` et `while`).

Exercice 22.

Programmer une fonction `carresemboites` qui prend pour arguments un entier `n` et un entier `nb` plus grand que 1 qui permette de tracer un carré initial de côté `n` puis `nb-1` carrés autour de celui-ci en augmentant à chaque nouveau carré tracé de 10 pixels la longueur du côté. On pourra bien-sûr se servir de la fonction `carre` déjà programmée !

Pour mieux comprendre, voici un exemple de ce que doit produire `carresemboites(30,12)` :

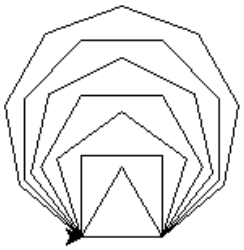


Correction.

```
def carresemboites(n,nb):
    for k in range(nb):
        carre(n+10*k)
```

Exercice 23.

1. Définir une fonction `polygoneregulier(n,L)` qui trace un polygone régulier à `n` côtés de longueur `L` et qui replace la tortue dans sa position précédant le tracé du polygone.
2. Définir une fonction `polygonesemboites(n,L)` qui trace les polygones réguliers à `k` côtés de longueur `L` pour `k` variant de 3 à `n` emboîtés les uns dans les autres. Voici ce que doit produire `polygonesemboites(10,50)` :



3. Tester `polygoneregulier(150,5)`. Que dire de la forme que l'on semble obtenir sur le dessin ?

Correction.

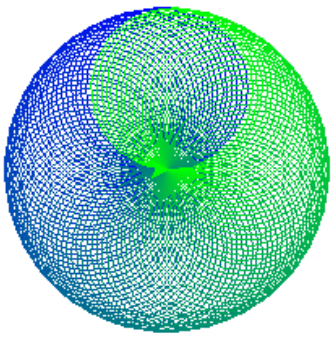
1.
```
def polygoneregulier(n,L):
    for i in range(n):
        forward(L)
        left(360/n)
```

2.
```
def polygonesemboites(n,L):
    for k in range(3,n):
        polygoneregulier(k,L)
```

3. Waouh, ça ressemble à un cercle !

Exercice 24.

Reproduire la figure ci-dessous (en n'oubliant pas d'augmenter la vitesse : `speed(0)` sinon... ce sera long !)



L'image contient 100 cercles de rayon 50 pixels. Dans un premier temps, on se contentera de tracer ce dessin en noir.

Pour changer la couleur d'un tracé, on utilise la fonction `color(couleur)` où `couleur` est une chaîne de caractère désignant une couleur (en anglais). Par exemple, `color('blue')` indique que la couleur des futurs tracés sera bleue.
On peut également fournir un tuple correspondant à une couleur RVB sachant que `1.0,0,0` correspond au rouge, `0,1.0,0` correspond au vert et `0,0,1.0` correspond au bleu. en modifiant les trois nombres entre 0.0 et 1.0, on obtient des mélanges de ces trois couleurs (de manière plus ou moins claires). Ainsi, `color(1.0,0.55,0.0)` correspondra à une couleur orangée.

Correction.

```
def supercercle(n):
    for i in range(n):
        color(0.0,1.0*(i/(n-1)),1.0*(n-1-i)/(n-1))
        circle(50)
        left(360/n)
```

3. T.P. n°1 : `random` et `turtle`

TP n°1 - Mise en pratique : les modules `random` et `turtle`