Initiation à la programmation en Python

Pierre Pompidor

19 septembre 2019

Table des matières

Intéressantes remarques liminaires :

Le contexte d'utilisation de mise en œuvre des scripts (des programmes) Python, donnés en exemple dans ce support, est le système d'exploitation Linux installé par défaut dans les salles de TP de la Faculté des Sciences de l'Université de Montpellier. Cela-dit, tous les scripts peuvent être exécutés sous Windows ou MacOS hormis indication contraire.

Ce support de cours ne doit être considéré que comme une introduction au langage de programmation Python. Notamment les arcanes de la programmation par objets en Python y sont très peu abordées, et en conséquence la nature "objets" de certaines structures de données (chaînes de caractères, listes, tuples...) ne sera pas explorée (ce qui n'empêche nullement de programmer gracieusement en Python).

1 Introduction à Python

1.1 Introduction à Python

Python [1] est un langage de programmation ubiquiste pour :
— le prototypage d'applications
— l'extension d'applications
— le calcul scientifique
— le scripting système
— la fouille de données (data mining)
— ...

Python est un langage mature (première version en 1991), multiplateformes (mais quasiment tous les langages "modernes" le sont), **modulaire**, **efficace** et **pédagogique**.

Voici le lien sur la documentation officielle de **Python 3** : `https://www.python.org/`

1.2 Python est un langage interprété

L'**interpréteur** python **python3** traduit, une par une, chaque instruction du programme écrit en Python (le programme *source*) en langage binaire exécutable par le processeur. Il n'y a donc pas (par défaut) d'exécutable stocké sur votre système de fichiers (contrairement par exemple à un programme écrit avec le langage C).

Un programme source **interprété**, (et non compilé par un *compilateur*), est généralement appelé un **script** (cette appellation renvoie aussi à l'idée que le programme est relativement court - personne ne s'attendant à ce qu'un script fasse plusieurs milliers de lignes - mais cette association d'idée est très subjective).

Il est possible de programmer :
— interactivement sous l'interpréteur (python3) (si vous y entrez, saisissez `exit()` pour en sortir) ;
— en créant donc des scripts (conventionnellement d'extension **.py**) que, dans l'écosystème des salles de TP de la Faculté des Sciences, nous exécuterons via un terminal (à l'"ancienne" donc).

Toute notre pratique passera par la création de scripts, notamment pour garder une trace pérenne de nos activités reptiliennes.

1.3 Python est modulaire

La **modularité** est une notion fondamentale en informatique en permettant le réemploi de codes déjà développés. Il est donc possible d'utiliser en Python des modules préexistants (beaucoup sont installés par défaut), et d'en écrire de nouveaux.

Un **module** Python :
— est donc un fichier contenant du code python ;
— permet de factoriser du code entre différentes applications ;
— permet de ne charger en mémoire que ce que l'on a besoin.

L'utilisation d'un module passe par le mot réservé `import` : `import <nomModule>`

Il est aussi possible de n'importer que certaines ressources d'un module (fonction, classe, ...) :

```
from math import sqrt
```

L'installation d'un module "public" sous Linux s'exécute avec avec la commande (à saisir dans un terminal) **pip3** : `pip3 install <nomModule>`

L'installation de `pip3` sous Linux/Ubuntu est réalisé par la commande suivante :

```
sudo apt install python-pip3
```

Par exemple, dans l'univers Python des mathématiques, un module très populaire est `numpy`.

1. créé par Guido van Rossum en 1989

La **bibliothèque** standard de Python contient non seulement des modules écrits en Python mais aussi des modules natifs écrits en C.

1.4 Un premier exemple de script python

Voici le traditionnel script *bonjour.py* (une variante du *"Hello world !"*) :

```
prenom = input("Quel est ton joli prénom ? ")
print("Bonjour", prenom)
```

Exécution du script : `python3 bonjour.py`

et le script saluera le prénom que vous aurez saisi au clavier !

Comme vous l'aurez deviné (et tous ces points seront détaillés par la suite) :
— `prenom` est une variable qui mémorise une valeur (et aurait pu s'appeler différemment) ;
— `input()` suspend l'exécution du script en attendant une saisie au clavier ;
— `print()` provoque un affichage sur le terminal (ou invite de commandes DOS) :
Il y d'autres syntaxes possibles pour utiliser print(), avec par exemple la spécification de formatages qui sera évoquée au paragraphe suivant.

Il est également possible sous Unix/Linux d'exécuter directement le script Python sans spécifier le nom de l'interpréteur Python dans le terminal.

Il faut pour cela faire apparaître la ligne `#!/usr/bin/env python3` comme première ligne du script :

```
#!/usr/bin/env python3
prenom = input("Quel est ton joli prénom ? ")
print("Bonjour", prenom)
```

Il ne faut alors pas oublier d'ajouter le droit d'exécution au script (ce point est expliqué lors du paragraphe suivant) : `chmod +x bonjour.py`

Et le script peut être exécuté ainsi : `./bonjour.py`

Le `./` en préfixe du nom du script indique à l'interpréteur de commandes que le script est situé dans le répertoire courant (sinon il faut modifier la variable système `PATH` qui liste les répertoires dans lequel l'interpréteur de commandes recherche les exécutables).

1.4.1 L'utilisation de la fonction print()

Différentes syntaxes de la fonction *print()* permettent de mettre en exergue les variables à intégrer dans la chaîne de caractères spécifiée comme premier paramètre (à l'instar de ce qui est proposé dans le langage C).

En voici différents exemples (*personne* et *age* étant deux variables contenant respectivement une chaîne de caractères et un entier) :

```
print("Je suis {} et j'ai {} ans", prenom, age)
print("Je suis {}, j'ai {} ans".format(prenom, age))
print("Je suis %s et j'ai %2d ans"%(prenom, age))
```

Le dernier exemple spécifie par exemple que l'âge est affiché sur deux caractères (s'il n'est formé que d'un seul chiffre, une espace sera ajoutée avant celui-ci).

Ce support de cours ne détaillera pas ce point bien éclairé au bout de ce lien :
`https://www.python-course.eu/python3_formatted_output.php`.

Par ailleurs des paramètres particuliers correspondant à des instructions de contrôle peuvent être donnés à *print()*. En voici deux exemples :

```
print("Après cela je ne veux pas passer à la ligne", end="")
print(mois, jour, annee, sep="/")
```

1.5 Contexte de développement sous Linux

1.5.1 Notion de chemin et déplacement dans un répertoire

Avant toute chose, précisons la notion de **chemin** sous Unix/Linux.

Un chemin désigne l'emplacement d'un dossier (ou répertoire) dans le système de fichiers (SF).

Un chemin peut être **absolu**, c'est à dire exprimé à partir de la racine du SF (cette racine étant le répertoire de plus haut niveau), ou **relatif**, c'est à dire exprimé dans le contexte du répertoire courant.

Le répertoire racine étant nommé `/`, un chemin absolu commencera par /.

Voici par exemple le chemin qui sur ma machine permet d'accéder au répertoire des principales applications systèmes : `/usr/bin`

Le répertoire `bin` est contenu dans le répertoire `usr`, lui-même contenu dans le répertoire racine.

Par exemple si je veux me déplacer dans ce répertoire, je peux exécuter la commande suivante : `cd /usr/bin` La commande *cd* est l'acronyme de *"change directory"*.

Dans ce chemin, le second / est juste un caractère de séparation syntaxique entre noms de dossiers.

Un chemin relatif est lui exprimé à partir du nom d'un sous-dossier ou des caractères `..` pour remonter au dossier parent.

Par exemple si je suis localisé dans `/usr` et je veux me déplacer dans `bin` ; je peux exécuter la commande suivante : `cd bin`

Autre exemple : je suis localisé dans `/usr/bin` et je veux me déplacer dans `/` via un chemin relatif ; je peux exécuter la commande suivante : `cd ../..`

Enfin, il a quelques raccourcis pour exprimer des chemins particuliers :
Le tilde `~` désigne votre répertoire d'accueil (celui dans lequel vous êtes positionné lors de votre connexion) ;
Le point `.` désigne le répertoire courant ;
et comme nous l'avons vu, des deux points `..` désignent le répertoire parent.

1.5.2 Quelques commandes Unix/Linux essentielles

Dans ce qui suit, `chemin/`, `chemin1/` et `chemin2/` sont facultatifs.

Emplacement dans le système de fichiers : `pwd`

Listing du contenu d'un dossier :
```
ls
ls -l
ls <chemin>
```
L'option `l` permet de connaître les informations détaillées.

Création d'un dossier (avec la commande *mkdir* comme *"make directory"*) : `mkdir <chemin/repertoire>`

Copie (avec la commande *cp* comme *"copy"*) :
```
cp fichier copieFichier
cp chemin1/fichier chemin2
cp chemin1/fichier chemin2/copieFichier
```

Renommage : `mv chemin1/fichier chemin2/nouveauFichier`

Déplacement d'un fichier (avec la commande *mv* comme *"move"*) `mv chemin1/fichier chemin2`

Gestion des droits :
```
chmod 755 <nomDuScript>
chmod +x <nomDuScript>
```
`rwxr-xr--` signifie "tous les droits pour le propriétaire, les droits de lecture et d'exécution/accès pour les membres du groupe, et seulement le droit de lecture pour les autres".

En effet, les droits associés à un fichier, ou à un répertoire, sont exprimés par 9 caractères répartis en trois groupes :

— les trois groupes désignent trois populations : le propriétaire, le groupe, les autres ;
— pour chaque population un droit est activé si la lettre correspondante est précisée : r (read), w (write), x (execute ou access), ou dénié si un tiret apparaît à la place de la sympathique lettre.

1.5.3 Comment développer en Python

Pour développer en Python, la première possibilité est d'utiliser un éditeur de texte "générique".

Parmi tous ceux qui existent sous Unix/Linux (*vi*, *nano*, *atom*, *sublime text*, ...), la "tradition" au sein du département informatique est d'utiliser **emacs** : `emacs <nom du script> &`

Cela-dit, `emacs` demande un certain temps d'appropriation si vous n'êtes pas familier avec les raccourcis clavier de l'interpréteur de commandes Unix/Linux.

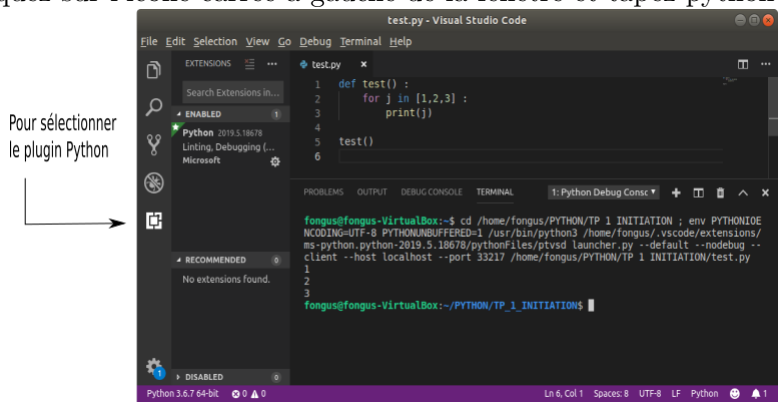
Ainsi si vous préférez les raccourcis clavier de Microsoft, et il faut le reconnaître un peu plus d'ergonomie, une bonne solution est **Visual Studio Code** (qui est libre et déjà installé) :

`code &`

Pour installer **Visual Studio Code** sur Linux/Ubuntu (`https://code.visualstudio.com/download`), voici la commande à utiliser :

`sudo apt install ./<nom de la dernière version>.deb`

Une fois Visual Studio Code installé, le plugin Python vous permettra de bénéficier d'aides syntaxiques : cliquez sur l'icône carrée à gauche de la fenêtre et tapez python dans la zone de recherche.



Vous pourrez lancer l'exécution d'un programme sous Visual Studio Code avec *Control-F5*.

La seconde possibilité est d'uiliser un environnement de développement spécifique à Python.

Voici les trois environnements de développement les plus populaires :

— **pyCharm** : `https://www.jetbrains.com/pycharm/`
— **pyzo** : `https://pyzo/`
— **spyder** : `https://www.spyder-ide.org/`

Enfin, il existe des environnements de développement en ligne tels que :

— jupyter : `https://jupyter.org/`
— cocalc : `https://cocalc.com/`

Quel que soit l'éditeur ou l'environnement de programmation que vous allez utiliser, il est probable (et souhaitable) que celui-ci encode les caractères que vous allez saisir dans le système d'encodage international UTF8 (encodage "multilingue" variable des caractères sur un ou plusieurs octets).

Nous vous recommandons ainsi de rajouter dans vos scripts la ligne `# -*- coding: utf-8 -*-` (dans l'exemple ci-dessous après le préambule destiné aux scripts sous Unix/Linux), qui permet à l'interpréteur de manipuler des données formatées dans cet encodage (y compris des noms de variables comprenant des caractères accentués, même si cela n'est pas trop recommandé) :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

prénom = input("Quel est ton joli prénom ? ")
print("Bonjour", prénom)
```

1.6 Programmes, variables scalaires, chaînes, listes et tuples

Un **programme** est un ensemble d'opérations destinées à être exécuté par un ordinateur (ou de manière plus spécifique par un ou plusieurs microprocesseur(s)). Un programme définit donc une suite d'actions (en tout cas dans le paradigme de la programmation impérative).

Un programme a pour but de manipuler des **données** (hormis ceux d'intérêt très limité).

Ces données sont mémorisées en mémoire vive pour pouvoir être manipulées (et ceci dans la plupart des cas même si elles sont récupérées sur un disque dur ou via une connexion réseau).

Une donnée allouée en mémoire peut être une **constante** si une fois mémorisée elle ne peut être modifiée, ou une **variable** dans le cas contraire.

Il est à remarquer que dans la plupart des langages, le nom de la donnée (par exemple un programme manipule une donnée nommée "login" qui contient l'identifiant d'un utilisateur) est associé à une adresse en mémoire.

Dans la quasi-totalité des langages les données sont typées par un **type** qui définit la nature de l'information qui est stockée. Ce type peut être "de base" (booléen (vrai ou faux), entier, réel/flottant, caractère...), ou renvoyer à une structure plus complexe associant différentes données, voire même associant les traitements qui peuvent être appliqués à ces données (on parlera alors d'**objets**).

Python est un langage de programmation par objets, mais cette notion complexe ne sera abordée que dans le dernier chapitre.

Python permet de créer :

- des variables scalaires qui ne contiennent qu'une seule valeur (notamment numériques) ;
- des chaînes de caractères (qui sont quelquefois considérées comme des variables scalaires) ;
- des variables composites qui regroupent plusieurs valeurs :
  - les **listes** (ou tableaux dynamiques) : les éléments d'une liste sont indicés par un entier ;
  - les **tuples** : les éléments d'un tuple ne peuvent être directement modifiés (avec les listes, les tuples sont regroupés sous l'agréable dénomination de *séquences*) [2] ;
  - les **dictionnaires** : les éléments d'un dictionnaire sont indexés par une clef qui est le résultat d'une fonction de hashage (un bref aperçu des dictionnaires sera donné un peu plus loin).
- et donc des **objets** qui dans le paradigme de la **programmation par objets** regroupent non seulement plusieurs variables, mais aussi spécifient les traitements qui peuvent leurs être appliqués.

*D*ans ce support de cours, les listes sont présentées avant les tuples bien que les élèves du secondaire semblent devoir s'initier aux tuples (qui ont très peu d'intérêt en programmation) avant les listes ; et la programmation par objets (voir le chapitre consacré à ce point) ne sera qu'esquissée bien que tout soit objet en Python (y compris les types de bases).

Pour connaître le type d'une variable en Python, vous pouvez utiliser la fonction `type()` :

```
i = 1
print(type(i)) # affiche <class 'int'>
```

Python vous explique d'une manière un peu absconse que la variable i est un objet de la classe int, et donc un entier.

2. Si l'élément d'un tuple est une variable composite modifiable (liste, objet, ...), le contenu de cette variable peut être néanmoins modifié.

1.6.1 Les variables scalaires

Voici quelques exemples de création de variables scalaires :

```
i = 1
f = 1.0  # float
booleen = False  # la valeur True serait plus positive
```

`f` est typé comme étant un *flottant* (ce point est détaillé dans l'enseignement consacré à la représentation des données) et nous remarquons au passage que tout ce qui suit un dièse, jusqu'à la fin de ligne, est un commentaire.

Le typage est dynamique :

— le type de la variable n'est pas précisé lors de sa création ;
— mais le transtypage (conversion d'une valeur d'un ceratin type vers un autre) n'est pas automatique comme le montrent les exemples donnés dans le paragraphe suivant.

1.6.2 Les chaînes de caractères

Les chaînes de caractères en Python sont définies en utilisant les simples ou doubles quote(s) pour circonscrire zéro, un, ou plusieurs caractères. *(Je vous rappelle que les chaînes de caractères sont des objets et que ce concept est introduit en fin de ce support de cours.)*

```
chaine = "Bonjour"
habileTraduction = 'Hello'
chaineVide = ""
autreExempleDeVacuite = ''
```

La manipulation d'une chaîne pour la considérer comme une valeur numérique, ou pour y insérer une valeur numérique, nécessite des fonctions de transtypage :

```
i = int("123")
chaine = "2fast"+str(4)+"U"
```

L'opérateur `+` permet ici de créer une chaîne de caractères à partir de plusieurs "morceaux" (il incarne bien sûr aussi l'opérateur arithmétique si ses opérandes sont numériques).

C'est ainsi que les deux dernières lignes de code suivant sont équivalentes :

```
nom = "toi"
print("Bonjour", nom)
print("Bonjour "+nom)
```

car la fonction `print()` insère automatiquement une espace entre chacun de ses paramètres.

Il est à remarquer avec délice que **la mémoire nécessaire au stockage de l'information est automatiquement allouée** ce qui permet d'éviter le principal risque de problème (bug) du développement informatique.

Pour illustrer ceci, voici une erreur fréquente en C :

```
char* chaine = "Bonjour"; // il y a déjà ici un gros risque
strcat(chaine, " toi");   // et là on assiste à la création d'une bombe à retardement !
```

En effet dans ce fragment de code C, le développeur a oublié de définir le nombre d'octets nécessaires à la mémorisation de la chaine "Bonjour toi" ce qui crée un code instable qui peut dysfonctionner au bout d'un nombre indéterminé d'exécutions !

En revanche il n'y a pas de souci en python, la mémoire étant automatique allouée (et plus ou moins désallouée aussi) :

```
chaine = "Bonjour"
chaine += " toi"  # chaine = chaine + " toi"
```

Ici l'opérateur `+` permet de concaténer, c'est à dire de "coller", une chaîne de caractères les unes au bout d'une autre.

Voici quelques exemples de création de chaînes de caractères qui illustrent la mise en œuvre des deux caractères d'encadrements (simples ou doubles quotes) d'une chaîne :

```
chaine1 = "Bonjour"
chaine2 = 'Hello'
chaine3 = "Aujourd'hui"
```

Le nombre de caractères d'une chaîne est accessible par la fonction `len()` (la création de fonctions sera présentée plus avant dans ce support de cours) :

```
print(len(chaine3))  # Affiche : 11
```

1.6.3 Les listes

Les **listes** sont en Python des structures de données qui ordonnent des éléments accessibles par un indice (un entier de valeur 0 pour le premier élément de la liste).

Avec les tuples qui sont présentés dans le paragraphe suivant, elles sont aussi appelées des *séquences*.

Les listes sont des tableaux dynamiques (des éléments peuvent être ajoutés ou supprimés à n'importe quel moment), et le type, et a fortiori, la taille mémoire de chaque élément, est indépendante de celle des autres éléments :

Les listes sont donc (comme les tuples, comme tout...) des objets (dans le sens de la programmation par objets), mais cette nature ne sera pas développée dans ce support de cours hormis l'exhibition d'une syntaxe particulière qui apparaîtra quand certains traitements sont appliqués à une liste.

```
menu = ["Attention", 1, 2, 3, "partez"]
print(menu[4])  # affiche : partez
```

Un indice négatif est utilisé pour "remonter" la liste à partir du dernier élément de celle-ci (d'indice -1) :

```
menu = ["Attention", 1, 2, 3, "partez"]
print(menu[-2])  # affiche : 3
```

La syntaxe des listes est définie par les points suivants :
— les crochets encadrent les éléments d'initialisation de la liste ;
— les éléments sont séparés par des virgules ;
— les éléments sont indicés par un entier (de valeur 0 pour le premier élément) ;
— un élément d'une liste peut être de n'importe quelle valeur.

Une liste peut-être initialisée à vide :

```
listeVide = []
```

Différents traitements comme l'ajout et la suppression d'éléments peuvent être appliqués à une liste :

```
liste = [1, 2, 3]
print(len(liste))  # affiche 3
liste.append("partez !")  # ajoute un élément à la fin de la liste
del liste[2]  # supprime le troisième élément
liste.remove(1)  # supprime le premier élément de valeur 1
```

Vous remarquerez, sans doute avec quelque dégoût, une certaine variabilité dans l'application des différents traitements sur une liste, et cela peut être en effet un peu déroutant. Sans rentrer dans le détail, une liste est fondamentalement un objet et peut être manipulée comme tel dans certains cas :

— `append()` et `remove()` sont des fonctions appelées **méthodes** appliquées sur l'**objet liste** via l'opérateur point `.` ;
— `len()` est une fonction générique qui est utilisée via une syntaxe classique ;
— `del` est une pseudo-fonction (c'est un cas d'utilisation rare, et vous pouvez ignorer superbement cette syntaxe).

La très populaire fonction `range()` permet de générer un tuple d'entiers (non modifiable), ainsi `range(10)` génère la séquence d'entiers de 0 à 9.

Il est possible de manipuler une tranche de listes (un *slice*) :

```
liste = [1, 2, 3, "partez", "!"]
print(liste[1:4]) # correspond à l'intervalle [1,4[
                  # affiche [2, 3, "partez"]
```

Un slice peut être très utile pour dupliquer une liste :

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste[:]
```

Pour essayez de comprendre pourquoi `nouvelleListe = liste` ne fonctionnerait pas, affichez avec la fonction *id()* l'adresse mémoire de *nouvelleListe* et *liste*.

Dans des cas très particuliers (références récursives) une copie de listes en profondeur peut être nécessaire :

```
nouvelleListe = copy.deepcopy(liste)
```

Notons une liste particulière : celle des paramètres du script (les paramètres du script sont les chaînes de caractères qui sont données sur la ligne de commande après le nom du script). Pour l'utiliser, il faut importer le module `sys` et préfixer le nom de cette liste avec le nom du module : `sys.argv`. Pour les puristes, le nom du module fait ici office d'*espace de noms*) ce qui permet d'éviter un télescopage entre deux entités de même nom faisant partie de deux modules différents.

Soit le script *bonjour.py* :

```
import sys

print(sys.argv[0])
print("Bonjour", sys.argv[1])
```

Son exécution `./bonjour.py Pierre` affiche

```
./bonjour.py
Bonjour Pierre
```

Il est à remarquer que le contenu d'un élément d'une liste est arbitraire : cela peut être par exemple une autre liste (ou d'autres structures de données) :

```
liste = [1, 2, 3, ["partez", "maintenant"]]
print(liste[3][1])  # affiche : maintenant
```

Pour vérifier si une valeur existe dans une liste, vous utiliserez l'opérateur `in` (et j'en profite pour faire un premier teasing sur les tests) :

```
if nom in listeEleves :
    print("Ah celui-là, je le connais bien")
```

Enfin une liste peut être initialisée via une syntaxe particulière nommée *liste en compréhension* [3], faisant intervenir la boucle `for`. En voici un exemple :

```
liste = [i*2 for i in range(10)]
for e in liste :
    print(e)
```

Enfin pour finir sur les listes, faites attention : une chaîne de caractères n'est pas une liste de caractères.

Pour modifier une chaine de caractères (autrement que pour lui concaténer une chaîne à sa suite), le mieux est de la transformer en liste avec `list()` (en fait `list()` est l'appel au constructeur de la classe qui crée les listes, chaque liste étant un objet).

3. en référence à la théorie des ensembles que je ne maîtrise absolument pas

Découvrez donc avec satisfaction ce code qui utilise, outre `list()`, la méthode `join()` pour créer uen chaîne de caractères à partir d'une liste.

```
chaine = "sybillin"
listeDeCaracteres = list(chaine)
listeDeCaracteres[1] = 'i'
listeDeCaracteres[3] = 'y'
print(listeDeCaracteres)
chaine = "".join(listeDeCaracteres)
print(chaine)
```

join() est une méthode sur une chaîne de caractères : nous utilisons donc une chaîne de caractères vide pour l'appliquer.

1.6.4 Les tuples

Un tuple renvoie à la notion mathématiques de **n-uplets**.
En Python, les **tuples** peuvent être considérés comme des listes non modifiables :
— ils sont encadrés par des parenthèses ;
— mais l'accès aux éléments se fait toujours via les crochets.

Voici un exemple de tuple :

```
unTuple = (1, 2, 3)
print(unTuple[0])  # affiche 1
unTuple[0] = "un"  # et bien sûr cela provoque une erreur !
```

Leur usage est restreint, et n'est motivé que si :
— des données doivent être protégées de toute altération ;
— de la mémoire doit être économisée (un tuple minimisant la mémoire).
Je n'aime pas les tuples.

1.7 Bloc d'instructions et structure conditionnelle

Passons maintenant aux choses sérieuses, abordons les structures qui permettent d'enchaîner des instructions et d'imposer notre volonté à la machine.

Un **bloc d'instructions** permet d'assujettir une série d'instructions :
— à un test (une structure conditionnelle) ;
— à une boucle (une structure itérative) ;
— à une fonction.

Voici un schéma de bloc d'instruction assujetti à une stucture conditionnelle.
La structure conditionnelle est initiée par le mot réservé `if`.
Une structure conditonnelle évalue une expression conditionnelle et si celle-ci est évaluée à vrai, exécute les instructions correspondantes comme le montre le schéma de programmation suivant .

```
if <expression conditionnelle> :
   instruction 1
   ...
   instruction n
```

La syntaxe d'un bloc d'instructions est régie par deux règles et une recommandation :
— le bloc d'instructions est initié par un **double point :** ;
— les instructions sont indentées (c'est-à-dire décalées) soit avec des espaces, soit avec des tabulations, à la même position horizontale
mais surtout ne mélangez pas des espaces avec des tabulations ;

— le bon goût recommande de ne programmer qu'une seule instruction par ligne (et dans ce cas il est inutile de mettre un point-virgule à la fin de la ligne ce qui est habituel dans beaucoup de langages de programmation) : si pour une raison obscure, vous voulez programmer plusieurs instructions sur la même ligne, le point-virgule servira alors de séparateur.

Voici un exemple de mise en œuvre d'une structure conditionnelle :

```
import sys
if len(sys.argv) != 2 :
   print("Le script doit avoir un paramètre")
   exit()
```

Dans le cas où le script est appelé sans paramètre, la liste `sys.argv` ne contient qu'un seul élément dont la valeur est le nom du script : l'expression conditionnelle est alors invalidée, et l'instruction `exit()` termine l'exécution du script.

L'alternative est introduite par le mot réservé `else` :

```
if <expression conditionnelle> :
   instruction 1
   ...
else :
   instruction 1 du bloc alternatif
   ...
```

Les opérateurs de l'expression conditionnelle sont :
— les opérateurs logiques : **and** et **or**
— les opérateurs de comparaison classiques (`>`, `<`, `>=`, `<=`) mais attention à la syntaxe de ceux d'égalité et d'inégalité : `==` et `!=`

Il est à noter qu'une expression renvoie faux si son évaluation est égale à 0 ou "".

Il n'est donc pas nécessaire, (et c'est même plutôt une faute de goût), d'encadrer l'expression conditionnelle par des parenthèses (comme dans la grande majorité des langages de programmation), mais celles-ci peuvent bien être sûr être utilisées au sein de l'expression pour définir des priorités dans l'évalution de certains termes de l'expression conditionnelle. Par exemple :

```
if (i > 10 and j < 10) or (i < 10 and j > 10) :
   print(i, "et", j, "ne se comprennent pas")
```

1.8 Les structures itératives

Pour répéter l'exécution d'une série d'instructions, une structure itérative (de petit nom "boucle") doit être mise en place.

En Python, nous avons le choix entre :
— une boucle "passe-partout", gérée généralement avec un indice de boucle : la boucle *while*
— une boucle spécifique pour itérer sur les éléments d'une séquence : la boucle *for* (qui permettra aussi de récupérer les clefs d'un dictionnaire, ou d'accéder à chaque ligne d'un fichier, ce que nous verrons plus loin.)

1.8.1 La boucle while

Commençons par examiner la mise en œuvre d'une boucle `while` dont le but est d'afficher tous les éléments d'une liste :

```
liste = [1, 2, 3, "partez", "!"]
i = 0
while i < len(liste) :
      print(liste[i])
      i += 1
```

La variable `i` est initialisée à 0, puis est incrémentée de 1 à la fin de chaque tour de boucle. Quand sa valeur atteint le nombre d'éléments de la liste, l'expression conditionnelle qui contrôle l'exécution de la boucle renvoie "faux", et la boucle s'arrête.

De cet exemple, nous pouvons généraliser le schéma de programmation usuel d'une boucle `while` :

```
<initialisation éventuelle d'un indice de boucle>
while <expression conditionnelle> :
      instructions du bloc
      <(in|de)crement éventuel d'un indice du boucle>
```

Nous pouvons remarquer que l'incrémentation utilisant l'opérateur unaire `++` n'existe pas en Python (on ne peut donc écrire dans l'exemple précédent `i++` comme nous pourrions le faire dans d'autres langages de programmation).

Une incrémentation/décrémentation peut avoir un pas supérieur à 1 : `i -= 10`

1.8.2 La boucle for

Découvrons le même traitement que précedemment mis en œuvre avec la boucle `for` :

```
liste = [1, 2, 3, "partez", "!"]
for valeur in liste :
    print(valeur)
```

Comme vous le voyez, il n'y a plus d'indice de boucle, et c'est magnifique !

Dans cet exemple, la variable `valeur` est déclarée dans l'entête de boucle, et est instanciée, itération après itération, avec la **valeur** de chaque élément de la liste.

Le schéma de programmation d'une boucle `for` est le suivant :

```
for <variable locale> in <iterable> :  # l'itérable peut être un tuple, une liste, ...
    instructions du bloc
```

Cette boucle peut être également utilisée pour itérer sur les clefs d'un dictionnaire ou les lignes d'un fichier.

1.9 Exemple synthétiques

1.9.1 Salutation de tous les paramètres donnés au script

Illustrons quelques points importants que nous avons vus (utilisation d'un module, liste (et slice), structures conditionnelle et itérative) dans un script qui teste son nombre de paramètres, et dans le cas où il en a au moins un, les salue :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import sys

if len(sys.argv) > 1 :
    for parametre in sys.argv[1:] :   # utilisation d'un slice pour éluder le nom du script
        print("Bonjour "+parametre)
else :
    print("J'aimerais tellement être poli !")
```

N'oubliez pas que la première ligne est destinée à l'interpréteur de commandes Linux. Sous Windows elle ne posera pas de problème (car elle sera considérée comme un commentaire), mais sera superflue.

1.9.2 Un tri par permutations (le tri à bulles)

Analysons maintenant une fonction qui ordonne de manière croissante les éléments d'une liste (à plat), via la mise en œuvre d'un tri par permutations (un *bubble sort*). Cette fonction présente deux structures de boucles emboîtées et une structure conditionnelle.

La complexité (très relative) de cette fonction nous pousse à exhiber dans un premier temps un **algorithme** (c'est le seul que nous verrons dans ce support), avant de présenter une possibilité d'implémentation en Python.

```
Fonction triParPermutations(liste)
   nouveauCycle <- vrai
   TantQue nouveauCycle est vrai
        nouveauCycle <- faux
        i <- 0
        TantQue i < nombreElementsDeLaliste - 1
            Si liste[i] > liste[i+1]
               echanger liste[i] avec liste[i+1]
               nouveauCycle <- vrai
            FinSi
            i <- i + 1
        FinTantQue
   FinTantQue
   renvoyer liste
FinFonction
```

```
def triParPermutations(liste) :
    l = len(liste)
    nouveauCycle = True
    while nouveauCycle :
        nouveauCycle = False
        i = 0
        while i < l-1 :
            if liste[i] > liste[i+1] :
                temp = liste[i]
                liste[i] = liste[i+1]
                liste[i+1] = temp
                nouveauCycle = True
            i += 1
    return liste
```

Comment pourrait-on utiliser cette fonction de tri via un module ?

2 Exercices : module, factorielle et génération de nombres premiers

2.1 Prise en main de l'environnement de travail

Ouvrez un terminal et :
— créez un dossier (par exemple "PROGPYTHON") ;
— déplacez-vous-y ;
— et prenez en main votre (futur) éditeur préféré (par exemple `emacs` ou `code`).

2.2 Création d'un module

Créez un module *tris.py* de tel sorte que le code suivant puisse fonctionner :

```
import tris

liste = [3, 2, 1, 5, 9, 1, -3, 6]
print(tris.bubbleSort(liste))
```

2.3 Factorielle en itératif

Ecrivez le script Python *fact.py* - voyez la paresse des informaticiens - qui calcule la factorielle d'un nombre entier positif.

Ce script doit être écrit :

— en mettant en œuvre une structure itérative (une autre version utilisant une fonction récursive sera écrite ultérieurement) ;

— en utilisant le moins de variables possibles.

Attention : une valeur saisie au clavier, ou donnée en paramètre au script, est considérée par Python comme étant de type "chaîne de caractères". Pour l'utiliser vous devrez donc la transtyper un entier.

2.4 Génération de nombres premiers sans utilisation de ceux déjà déterminés

Ecrivez le script Python *np.py* (ah là là) qui :

— soit génère les nombres premiers inférieurs à la valeur donnée au script ;

— soit génère les n premiers nombres premiers, n étant égal à la valeur donnée au script.

(Fou/folle d'entousiasme, vous pouvez aussi développer les deux versions).

Attention : la génération des nombres premiers ne doit pas se servir des nombres premiers déjà trouvés, et rappelez-vous le remarque faite précédemment au sujet de la saisie au clavier ou du passage de paramètre...

2.5 Génération de nombres premiers avec utilisation de ceux déjà déterminés

Ecrivez enfin une nouvelle version du script Python de génération de nombres premiers (*np2.py*), en réutilisant les nombres premiers déjà déterminés : pour cela implémenter une liste avec comme seule valeur initale 2.

Easayez de comparer les temps d'exécution de *np.py* et *np2.py* avec la commande Unix/linux suivante :

```
time <nom du script>
```

3 Dictionnaires et fonctions

3.1 Les dictionnaires

Si vous avez aimé les listes, une émotion excessive peut vous étreindre au contact des dictionnaires.

Commençons joyeusement par un exemple de mise en œuvre de **dictionnaire** :

```
nbJoursMois = {'janvier':31, 'fevrier':28.25, 'mars':31, 'avril':30}
print(nbJoursMois['fevrier'])  # Affiche : 28.25
```

Comme vous l'avez compris les dictionnaires permettent de rajouter de la **sémantique** aux éléments qui sont regroupés (car les éléments sont **nommés** par une **clef**), et donc confèrent une meilleure lisibilité des programmes : `print(nbJoursMois['fevrier'])` est quand même plus lisible que `print(nbJoursMois[1])`.

Mais, et c'est là le pic émotionnel, les dictionnaires permettent aussi d'accéder directement, et donc **beaucoup plus rapidement**, à l'emplacement de l'élément en mémoire grâce à une **fonction de hachage/hashage** qui renvoie une adresse à partir des caractères composant la clef) .

Dans d'autres langages, les dictionnaires sont aussi nommés tableaux ou tables de hachage (ou de hashage).

Un dictionnaire peut-être initialisé à vide :

```
dicoVide = {}
```

L'accès aux éléments d'un dictionnaire (par exemple pour les afficher) se fait généralement via la liste des clefs grâce à une syntaxe condensée :

```
nbJoursMois = {'janvier':31, 'fevrier':28.25, 'mars':31, 'avril':30}

for clef in nbJoursMois :
    print(clef, ":", nbJoursMois[clef])
```

La méthode `keys()` renvoie un tuple contenant les clefs du dictionnaire, et nous pourrions expliciter notre code ainsi :

```
nomsMois = nbJoursMois.keys()
for nomMois in nomsMois :
    print(nomMois, ":", nbJoursMois[clef])
```

Il est aussi possible d'utiliser la méthode `items()` qui renvoie une liste de tuples associant clef et valeur :

```
for (clef, valeur) in nbJoursMois.items() :
    print(clef, ":", valeur)
```

Dans ce monde merveilleux, il faut quand même garder une certaine stabilité émotionnelle, car l'ordre d'affichage des éléments :

— peut ne pas correspondre à l'ordre de création de ces éléments ;
— mais il est possible d'utiliser des dictionnaires ordonnés (non présentés dans ce support de cours).

Pour vérifier si une clef existe dans un dictionnaire, utilisez l'opérateur `in` :

```
if nom in dictionnaireEleves :
    print("Ah celui-là, je le connais très très bien")
```

Et à l'instar de ce que nous avons vu avec les listes, il est possible de créer des *dictionnaires en compréhension* [4]. En voici un exemple de mon cru (et donc incongru) :

```
lettres = "abcdefghijklmnopqrstuvwxyz"
dico = {lettres[i-1]:i for i in range(1, 27)}
for (k, v) in dico.items() :
    print(k, v)
```

Le dictionnaire aura pour clef chaque lettre de l'alphabet associée en valeur à son ordre (1 sera par exemple en valeur de `dico['a']`).

3.2 Les fonctions

Une **fonction** est un bloc d'instructions paramétré, dont voici un exemple trivial :

```
def somme(a, b) :
    return a+b

print(somme(1,2))
```

Dans cet exemple trivial, la fonction renvoie une valeur numérique, mais une fonction peut renvoyer également une variable composite (liste ...).

Une fonction permet de factoriser du code, et :

— est initiée par le mot réservé `def` :
— renvoie (généralement) un résultat avec le mot réservé `return`.

Des fonctions d'usages particuliers ont des noms particuliers :

— si une fonction ne renvoie pas de résultat, elle peut être nommée *procédure* ;
— une fonction peut elle-même se rappeler, c'est une *fonction récursive*.

Enfin quand une fonction est appliquée sur une variable via l'opérateur point, elle est appelée une **méthode**. Nous en avons vus déjà de nombreux exemples comme l'ajout d'un élément dans une liste avec `append()` :

```
chosesAFaire.append("allez les gars, un dernier calcul...")
```

Cette terminologie est utilisée dans le contexte de la programmation par objets qui est introduite dans le chapitre de même nom.

3.2.1 Fonctions récursives

Voici un exemple de fonction récursive qui affiche sur le terminal la décrémentation d'un entier (ici 10) ; lé décompte final affichant "FIN !" :

4. en référence à la théorie des ensembles toujours aussi peu maîtrisée

```
def decompte(n) :
    print(n)
    if n == 0 :
       print("FIN !")
       return
    decompte(n-1)

decompte(10)
```

L'examen de ce programme devrait susciter une question existentielle : comment une variable (ici n) peut-elle exister plusieurs fois en même temps ?

Vous remarquerez que :

— quand le paramètre `n` vaut par exemple `5`, il y 6 contextes d'exécution du bloc d'instructions de la fonction, donc 5 sont en suspens. Cela implique que :
  — le paramètre `n` a différentes valeurs simultanées ;
  — ces valeurs sont sauvegardées dans une zone mémoire particulière (la *pile*).

— le `return` ne renvoie aucun résultat, et ne sert qu'à arrêter l'exécution du contexte d'exécution courant de la fonction.

3.2.2 Fonctions à nombre de paramètres variables

Puthon permet de créer des fonctions possédant un nombre variable de paramètres (par exemple `print()` est une fonction qui peut recevoir un nombre arbitraire de paramètres).

Dans cet exemple, les paramètres supplémentaires sont mémorisés dans une liste :

```
def foo(theBoss, *args) :
    print("Ô bonjour", theBoss)
    for arg in args :
        print("Bonjour", arg)

foo("Pierre", "Paul", "Jacques")
```

Ainsi la valeur "Pierre" est stockée dans `theBoss`, et les valeurs `Paul` et `Jacques` dans la liste `args`.

Le symbole `*` indique que les paramètres surnuméraires sont regroupés dans une liste.

Le nom `args` est conventionnel, cette variable peut être nommée différemment (par exemple `etLesAutres`).

3.2.3 Fonctions à paramètres clefs/valeurs

Des paramètres peuvent être aussi gérés sous formes de clefs/valeurs pour peupler un dictionnaire :

```
def foo(**kwargs) :
    for (k, v) in kwargs.items() :
        print(k, v)

foo(merveille1="Pyramides de Kheops",
    merveille2="Jardins suspendus de Babylone")
```

Ici c'est le symbole `**` qui indique que les paramètres sont regroupés dans un dictionnaire (et le nom `kwargs` est conventionnel).

3.2.4 Des fonctions qui n'en sont pas : les générateurs

Voici brutalement (nous ne plaisantons plus) un exemple de générateur qui décompte à partir de 10 :

```
def creeGenerateur() :
    for i in range(10, 0, -1) :
        yield i

generateur = creeGenerateur()
print(next(generateur))  # affiche : 10
print(next(generateur))  # affiche : 9
```

Un générateur mis en œuvre grâce au mot-clef **yield** :

— crée un flux de données (une sorte de *réservoir de données* ;
— qui peut être sollicité **au coup par coup** par **next()**.

4 Exercices : mise en pratique des dictionnaires et des fonctions

4.1 Un chiffrement très très naïf

Ecrivez un script Python qui :
— crée un **dictionnaire** qui associe à quelques lettres une lettre de remplacement (par exemple 'a' correspond à 'y', et 'e' à 'z') (cette initailisation sera donnée en "dur" dans le programme) ;
— complète automatiquement ce dictionnaire pour ajouter les correspondances inverses (dans notre cas 'y' correspond à 'a', et 'z' à 'e')
(on pourrait essayer d'implémenter le chiffre de César..., mais on ne le fera pas) ;
— et affiche la chaîne (donnée en paramètre ou saisie au clavier) ainsi chiffrée.

Plusieurs problèmes devraient survenir qui seront sans doute résolus par l'utilisation de `list()`.

4.2 Factorielle en récursif

Ecrivez une nouvelle version du script Python de calcul de factoriel (*fact2.py*) qui calcule la factorielle d'un nombre entier positif en utilisant une **fonction récursive**.

5 Fichiers, manipulation de données textuelles, gestion des exceptions

5.1 Les fichiers

Python permet de lire le contenu de **fichiers** (si ces contenus sont lisibles), d'ajouter du contenu à des fichiers existants, ou d'en créer de nouveaux.

5.1.1 Lecture d'un fichier

Examinons tout d'abord le schéma de programmation classique de lecture du contenu d'un fichier "texte" (document, programme...) :

```
fd = open(<chemin/nomFichier>, "r")
for ligne in fd :
    print(ligne, end="")
fd.close()
```

La fonction `open()` verifie si le fichier est présent dans le répertoire indiqué (ou sinon dans le répertoire dans lequel s'exécute le script), et si l'utlisateur du script a les droits nécessaires à sa manipulation (ici une lecture).

Si ces conditions sont réunies, cette fonction retourne un **descripteur de fichier** qui peut être considéré comme un "pointeur" localisant le fichier dans la mémoire de masse (en fait la nature d'un descripteur de fichier est une chose si compliquée que je vous recommande de questionner à son sujet un enseignant mal-aimé), et ce descripteur de fichier peut alors être directement utilisé dans la boucle for (c'est magique).

La boucle `for` qui met en œuvre un descripteur de fichier value alors la variable qu'elle déclare (ici `ligne`) par chaque ligne du fichier, le retour à la ligne compris : c'est pour cela que le paramètre de contrôle `end=""` est ajouté au `print()` (il supprime le retour à la ligne provoqué par celui-ci.

La fonction `close()` indique au système d'exploitation que le fichier n'est plus utilisé et qu'il peut être "fermé" (dans cet exemple son emploi est de pure forme, les fichiers étant automatiquement fermés à la fin d'un script).

Un fichier peut être ouvert dans différents **modes**.
Voici les trois principaux modes :
— r : lecture ;
— w : écriture (si le fichier pré-existe, il est écrasé) ;
— a : écriture en ajout (dans un fichier existant).

Le schéma de programmation précédent n'est pas un modèle d'efficacité car il peut y avoir autant de lectures en mémoire de masse que de lignes dans le fichier.

Pour améliorer cela, un transfert complet du fichier en mémoire centrale (mais attention à la taille de votre mémoire vive), peut être effectué avec la méthode `readlines()` qui crée une liste contenant toutes les lignes du fichier (en optimisant donc le nombre d'accès à la mémoire de masse mais en consommant de la mémoire centrale) :

```
fd = open("fichier", "r")
for ligne in fd.readlines() :
    print(ligne, end="")
fd.close()
```

Un fichier peut aussi être lu, ligne par ligne, avec les méthodes `readline()` ou `next()` :

```
fd = open("fichier", "r")
print(fd.readline())
print(next(fd))
```

Attention : *next()* émet une erreur à la fin du fichier.

5.1.2 Ecriture dans un fichier

L'écriture dans un fichier "texte" - nous n'aborderons pas le domaine des fichiers binaires - est effectuée avec les méthodes `write()` (qui écrit une seule ligne), ou `writelines()` (qui écrit les lignes contenues dans la liste qui lui est passée en paramètre).

En voici un exemple :

```
fd = open("fichier", "w")
fd.write("Premiere ligne")
lignes = ["Seconde ligne", "Troisieme ligne"]
fd.writelines(lignes)
```

5.1.3 Eventuels problèmes d'encodage

Il est possible que vous rencontriez des problèmes d'encodage si par exemple vous ouvrez un fichier encodé en latin1 (chaque caractère étant mémorisé sur un seul octet en ASCII étendu), alors que vous êtes dans un environnement UTF8 (dans lequel les caractères peuvent être mémorisés sur un ou plusieurs octets).

Dans ce cas, précisez l'encodage lors de l'ouverture du fichier :

```
fd = open("fichier", "r", "iso-8859-15")
```

et réencodez le texte si nécessaire :

```
ligneUTF = ligne.encode("utf-8")
```

5.2 Manipulation (recherche, extraction) de données textuelles

La manipulation (restreinte à des recherches et des extractions) de données textuelles - dans les exemples qui suivent, cette manipulation concerne des chaînes de caractères - peut être effectuée de différentes manières, dont nous présentons ici deux possibilités :

— l'utilisation de la fonction `split()` qui découpe une chaîne de caractères par rapport à un séparateur ;
— la mise en œuvre d'expressions régulières (qui sera juste abordée) .

La fonction `split()` appliquée à une chaîne de caractères (suivant la syntaxe de la programmation objets) crée une liste qui contient toutes les sous-chaînes qui la composent.

Voici un exemple qui éclaire ce mécanisme (le séparateur étant ici le point) :

```
chaine = "coursPython.tar.gz"
sousChaines = chaine.split(".")
if len(chaines) > 1 :
   print("Derniere extension =", sousChaines[-1])
```

et donc `sousChaines[0]` vaut `coursPython`, sousChaines[1] vaut `tar` et sousChaines[2] vaut `gz`.

Et pour information (le sujet étant insondable), voici un petit aperçu de la mise en œuvre d'**expressions régulières** (le module `re` doit être importé).

Une expression régulière (une expression soumises à des règles) est un motif qui permet d'identifier, et le cas échéant d'extraire, une ou plusieurs sous-chaînes incluses dans une chaîne de caractères. Des méta-caractères permettent de définir la nature du motif et définissent une vraie grammaire.

Voici un premier exemple qui vérifie avec la fonction `re.search()` si dans une chaîne `chaine` se trouve le mot *Madame*, *Mademoiselle*, *Mondamoiseau* ou *Monsieur* :

```
import re
...
if re.search("(Madame|Mademoiselle|Mondamoiseau|Monsieur)", chaine) :
   print("Ligne avec civilite")
```

Si la variable `chaine` a par exemple pour valeur `Madame Jessica Nette`, le test est vérifié.

Les parenthèses définissent ici les choix possibles entre plusieurs valeurs qui sont séparés par des pipes (`|`). Comme nous le verrons juste après, les parenthèses peuvent être aussi utilisées pour une extraction d'informations.

Et voici l'extraction de sous-chaînes via la même fonction `re.search()` :
L'expression régulière ci-dessous utilise les symboles suivants :
— `\d` : un chiffre, et avec le répétiteur `+`, un nombre entier ;
— `.` : un caractère quelconque (lettre, chiffre, signe...) ;
— `\.` : le caractère point (l'antislash étant le métacaractère de déspécialisation).

```
# ligneDeLog vaut par exemple "127.0.0.1 - - [03/Jun/2019] ..."
resultat = re.search("(\d+\.\d+\.\d+\.\d+) ", ligneDeLog)
if resultat :
   print("Adresse IP : ", resultat.group(1))
```

Dans cet exemple un peu plus compliqué, le motif spécifie quatre nombres composés de un à une infinité de chiffres (`\d+` séparés par des points (de manière très approximative, une adresse IP).

Si ce motif est retrouvé en valeur de `ligneDeLog`, il est stocké dans la variable `resultat` (un objet) grâce à l'usage des parenthèses.

La variable `resultat` contient en fait une adresse qui désigne un bloc mémoire dans lequel sont mémorisées les informations extraites par l'application de l'expression régulière (si la recherche a échoué, `resultat` vaut `None` (adresse mémoire vide correspondant à 0) et le test suivant sera faux).

L'accès à l'information qui a été extraite, est mise en œuvre par la méthode `group()`, 1 signifiant que nous voulons manipuler la première chaine de caractères qui a été extraite (ce qui est parfait vu qu'une seule a été extraite) [5].

5.3 Gestion des exceptions

L'histoire de l'informatique étant jalonnée par la survenue de dysfonctionnements coûteux, (mais heureusement pour l'honneur des informaticiens que Tchernobyl était dû à une consommation ludique d'alcool), il est recommandé de mettre en place une **gestion d'exceptions** pour garder la main sur un programme en cas de la survenue de problèmes a priori exceptionnels mais envisageables.

Ainsi les zones "critiques" peuvent être protégées par l'exécution de codes alternatifs.

La gestion des exceptions repose sur trois clauses - des mots réservés qui introduisent des blocs d'instructions - principales :
— `try` : bloc d'instructions "protégé" ;
— `except` : bloc d'instructions à exécuter en cas d'erreur ;
— `else` : bloc d'instructions positionné après les clauses `exec`.

Pour illustrer cela, prenons le cas de l'exploitation d'une ressource qui devrait exister, mais qui n'existe pas, (ou qui ne peut pas être exploitée avec les droits que le programme possède).

```
try:
    fd = open("JeNExistePas", "r")

except OSError as err:
    print("OS error: {}".format(err))
    print("la ressource demandée n'existe pas !")

else :
    print("Ok tout va bien, et le nombre de lignes du fichier est", len(fd.readlines()))
    fd.close()
```

5. resultat.group(0) mémorise la chaîne de caractères recouverte par l'expression régulière, soit subtilement ici l'adresse IP suivie d'une espace.

Voici un autre exemple mettant en œuvre un déclenchement suicidaire d'une exception sur une erreur de typage :

```
try:
    valeur = int("abc")

except ValueError as err:
    print("Value Error: {}".format(err))
    print("il est toujours instructif de contempler une erreur")
```

6 Exercice : petit quizz sur les capitales

En utilisant un fichier publié par le gouvernement français et accessible à ces adresses :
— sur le Moodle du cours : `https://moodle.umontpellier.fr/course/view.php?id=11006`
— `https://advanse.lirmm.fr/~pompidor/DIU/capitales.csv`

écrivez un script Python qui :
— demande à l'utilisateur - via une saisie au clavier ou un paramètre - le nombre de questions à poser ;
— ouvre et analyse ("*parse*") le fichier *capitales.csv* pour créer :
  — soit deux listes contenant respectivement (au même indice) les noms de pays et leurs capitables
  — soit un dictionnaire avec comme clef le nom d'un pays et en valeur sa capitale
— affiche aléatoirement (en utilisant le module `random` et sans jamais reposer deux fois la même question) un nom de pays, et interroge l'utilisateur sur le nom de sa capitale, puis :
  — en cas de succès, incrémente un compteur
  — en cas d'échec, mortifie l'utilisateur en affichant la bonne réponse
— affiche en fin d'exécution le compteur (et un petit mot d'encouragement aux nuls en géographie).

Les informations données entre parenthèses pouvant être écartées, le cas du "Congo" sera ignoré :

```
...
Congo (le),Brazzaville
Congo (la République démocratique du),Kinshasa
...
```

Voici le début du fichier *capitales.csv* :

```
Afghanistan (l'),Kaboul
Afrique du Sud (l'),Prétoria
Albanie (l'),Tirana
Algérie (l'),Alger
Allemagne (l'),Berlin
Andorre (l'),Andorre-la-Vieille
...
```

Voici la superstructure du script *quizz.py* (ici le nombre de questions est reçu en paramètre mais pourrait bien sûr être saisi au clavier) :

```
#!/usr/bin/env python3
# -*- coding : utf-8 -*-
import sys, os, random

os.system("clear") # Pour une exécution sous Unix/Linux exclusivement)

if len(sys.argv) > 1 and sys.argv[1].isdigit() :

   ...

else :
   print("Mauvais usage du script : un nombre de questions doit être donné en paramètre !")
```

7 Quelques éléments de programmation système

7.1 Gestion d'un exécutable

7.1.1 Capture du résultat d'un exécutable

L'accès **au résultat** d'un exécutable est possible grâce à la fonction `popen()` du module `os`.

En voici un exemple d'utilisation :

```
import os
ll = os.popen("ls -l")
for ligne in ll :
    print(ligne)
```

`os.popen()` renvoie une liste et donc attention à votre mémoire...

7.1.2 Exécution d'un exécutable

L'exécution directe d'un exécutable est possible grâce à la fonction `system()` du module `os`.

En voici un exemple d'utilisation :

```
import os
os.system("ls -l")
```

7.2 Exploration d'une arborescence de dossier

L'exploration d'une arborescence de dossier s'effectue via une fonction récursive dont voici le squelette type :

```
import os
def parcours(repertoire) :
    print("Je suis dans ", repertoire)
    liste = os.listdir(repertoire)
    for fichier in liste :
        if os.path.isdir(...) :  # traitement d'un dossier
           ...
        if os.path.isfile(...) :  # traitement d'un fichier régulier
           ...

parcours(...)  # appel de l'exploration récursive
```

— `os.listdir()` : liste du répertoire
— `os.path.isdir()` : teste si un fichier est un dossier
— `os.path.isfile()` : teste si un fichier est un fichier régulier

8 Introduction à la programmation par objets

Ce chapitre n'ayant pour but que de donner un léger éclairage sur la programmation par objets, il n'aborde pas les concepts d'héritage ou d'interfaces.

La **programmation par objets** (ou programmation objets) est un paradigme de programmation qui permet de structurer un programme informatique, notamment si celui-ci manipule de nombreux concepts. Imaginons par exemple que nous voulions développer un logiciel commercial, les concepts associés vont se compter par dizaines (client, produit, fournisseur, marque, stock, facture, paiement...), et nécessitent que les traitements et leurs cibles soient bien identifiés.

La programmation objets en Python met en œuvre, comme dans la très grande majorité des langages objets, des **classes** qui représentent ces concepts, et qui vont produire des **objets**.

8.1 Les classes

Une **classe** définit une **spécification d'un concept** en réunissant :
— les informations qui définissent le concept ;

— et les traitements pouvant être appliqués sur ces informations (dans la terminologie objet ces fonctions sont nommées des **méthodes** (ou des opérations)).

Ce qui est important de comprendre est que cette classe ne produira une entité réellement manipulable, (se traduisant par des données allouées en mémoire), qu'au moment où une instance particulière de ce concept, appelée **objet**, sera créée : une classe est instanciable en un `objet` (en fait plusieurs ;)).

Illustrons primesautièrement ce qui vient d'être dit.

La saison s'y prêtant un peu, nous décidons de créer un script en Python qui nous permettra au retour de nos erratiques recherches, de noter tous les champignons récoltés, et leurs effets digestifs, (ce script devrait ensuite nous permettre de créer un fichier dans lequel nos belles cueillettes seront inscrites dans le marbre, mais laissons cela de côté pour se concentrer sur l'utilisation d'une classe).

Définissons donc une classe implémentant le concept de Champignon (réduit au nom commun de son espèce, au nom scientifique de son genre, sa comestibilité et la date de sa rencontre) :

```
# -*- coding: utf-8 -*-

class Champignon :
    def __init__(self, genre, espece, comestibilite, date) :
        self.genre = espece
        self.espece = espece
        self.comestibilite = comestibilite
        self.date = date

    def quiSuisJe(self) :
        print("Je suis un/une", self.espece, "("+self.genre+") :", self.comestibilite, "et trouvé(e) le",
```

Au niveau opératoire, si ce script est exécuté tel quel : il ne se passe rien !

En effet, la classe spécifie de manière abstraite qu'un champignon rencontré à une certaine date, fait partie d'une espèce attachée à un genre, et qu'il présente une certaine comestibilité. Et pour l'instant ce script n'a créé aucun champignon particulier à partir de cette matrice de création.

Au niveau syntaxique, la classe est définie par le mot réservé `class` et possède deux méthodes :

— son **constructeur** nommé `__init__` sera utilisé au moment de la création d'un champignon particulier :
`<nom de l'objet de type Champignon> = Champignon(...)`

— et `quiSuisJe()` qui permettra d'afficher toutes les informations relatives à un champignon une fois celui-ci créé.

Le paramètre `self`, qui apparaît en première position dans les méthodes de la classe, représente l'objet qui utilisera cette classe pour exister. Il contiendra l'adresse mémoire de la zone mémoire allouée à l'objet (en informatique, on parle de *référence*). *Dans d'autres langages, la référence à l'objet courant est appelé **this**, et n'apparaît pas dans les paramètres des méthodes.*

8.2 Les objets

Une classe est la matrice de création d'un objet.

Imaginons maintenant que je rajoute les lignes suivantes après la définition de la classe, et que je ré-exécute le script :

```
c1 = Champignon("morille blonde", "morchella", "excellente à condition d'être bien cuite", "16/06/2019")
c2 = Champignon("bolet méditerranéen", "Suillaceae", "délicatement laxatif", "16/06/2019")
c1.quiSuisJe()
c2.quiSuisJe()
```

Deux zones mémoires sont allouées en mémoire dans lesquelles les données (les attributs) correspondantes aux deux objets sont mémorisées.

Dans un objet, un attribut est accessible par l'opérateur point, ainsi nous pourrions afficher l'espèce de `c1` avec :

```
print(c1.espece)
```

La méthode `quiSuisJe()` permet d'afficher ces données.

9 Exercice : exemple de modélisation de titres musicaux (30 minutes)

Nous voulons créer des albums de musique composés de titres musicaux.

Création de la classe Titre et des titres :

Votre script Python doit définir une classe *Titre* de telle sorte que ces lignes de codes puissent fonctionner :

```
lEncreDeTesYeux = Titre("L'encre de tes yeux", "Cabrel", 187)
laDameDeHauteSavoie = Titre("La dame de Haute Savoie", "Cabrel", 182)
jePenseEncoreAToi = Titre("Je pense encore à toi", "Cabrel", 190)
```

Le troisième paramètre correspond à la durée du titre en secondes.

Création de la classe Album et des albums :

Votre script Python doit définir une classe *Album* de telle sorte que ces lignes de codes puissent fonctionner (les lignes en commentaires correspondent à l'affichage des appels de méthodes) :

```
fragile = Album(lEncreDeTesYeux, laDameDeHauteSavoie, jePenseEncoreAToi)

fragile.nbTitres();
# 3
fragile.duree();
# 579 secondes
```

10 Bref aperçu de l'interfaçage graphique

Plusieurs possibilités existent pour créer une interface graphique en Python.

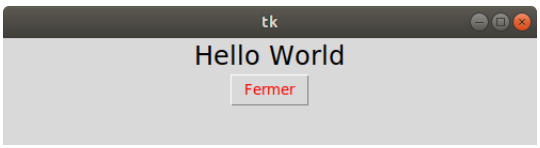
Pour qu'une interface s'incrive dans une nouvelle fenêtre locale, le module `tkinter` peut être utilisé (c'est une vieille bibliothèque graphique historiquement liée au langage *TCL* mais qui a été ensuite associée à de nombreux langages).

En voici un exemple d'illustration :

```
from tkinter import *
import tkinter.font as tkFont

fenetre = Tk()
fenetre.geometry("500x100")
myFont = tkFont.Font(size=18)

label = Label(fenetre, text="Hello World", font=myFont)
label.pack()
bouton=Button(fenetre, text="Fermer", command=fenetre.quit, fg="red")
bouton.pack()

fenetre.mainloop()
```

Ce code ne sera pas commenté en détail, (il est en fait assez intuitif), hormis sur les deux points suivants :
— la méthode `pack()` permet de faire apparaître un élément dans la fenêtre à laquelle il a été préalablement associé ;
— la méthode `mainloop()` permet de créer une boucle infinie qui fera perdurer la fenêtre, tant que le bouton ne sera pas sélectionné.

et voici le résultat de son exécution :



Outre l'utilisation du module `tkinter` :
— le module `matplotlib` propose de nombreuses fonctionnalités pour produire des graphiques, tracés de courbes... ;
— il est possible de créer des scripts "serveur" en Python dont les résultats sont médiatisés dans un navigateur via la mise en œuvre de différentes technologies du web côté client (HTML, CSS, JavaScript).

Index