



OPEN MINDS CLUB

FORMATION

---

# Langage Python

---



Sidahmed Bilal MALAOU

14<sup>es</sup> Février 2017 — 20 Mars 2017

# Sommaire

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>La programmation avec Python 3</b>                       | <b>1</b> |
| 1.1      | Introduction  | 1        |
| 1.1.1    | C'est quoi un programme??                                   | 1        |
| 1.1.2    | Comment sont créés ces programmes??                         | 2        |
| 1.1.3    | Les langages de programmation les plus connus               | 4        |
| 1.1.4    | Compilé vs Interprété                                       | 4        |
| 1.1.5    | Le langage Python   | 4        |
| 1.2      | Les logiciels nécessaires pour la programmation             | 5        |
| 1.2.1    | Installer Python et PyCharm sous Ubuntu Linux               | 6        |
| 1.2.2    | Installer Python et PyCharm sous Windows                    | 7        |
| 1.3      | Hello World   | 9        |
| 1.3.1    | Les commentaires  | 11       |
| 1.3.2    | Exercice 1  | 11       |
| 1.3.3    | Solution de l'exercice                                      | 12       |
| 1.4      | Les variables   | 13       |
| 1.4.1    | Créer et utiliser les variables avec python                 | 14       |
| 1.4.2    | Quel genre d'information on peut stocker dans nos variables | 15       |
| 1.4.3    | Afficher les variables                                      | 15       |
| 1.4.4    | Récupérer une saisie  | 16       |
| 1.4.5    | Comment nommer les variables                                | 17       |
| 1.4.6    | Exercice 2  | 18       |
| 1.5      | les opérateurs arithmétique                                 | 19       |
| 1.5.1    | Addition, soustraction, multiplication                      | 19       |
| 1.5.2    | Division entière et Division euclidienne                    | 19       |
| 1.5.3    | Modulo (reste de la division euclidienne)                   | 19       |
| 1.5.4    | La puissance  | 20       |
| 1.5.5    | Opérations entre variables                                  | 20       |
| 1.5.6    | Bonus   | 20       |
| 1.5.7    | Un petit problème!  | 21       |
| 1.5.8    | Exercice 3  | 21       |
| 1.6      | Les conditions  | 23       |
| 1.6.1    | Le if   | 23       |
| 1.6.2    | Le else   | 24       |
| 1.6.3    | Le elif   | 24       |
| 1.6.4    | Comparez les chaînes de caractères                          | 25       |
| 1.6.5    | Plusieurs conditions à la fois                              | 25       |
| 1.6.6    | Les booléens  | 26       |
| 1.6.7    | Exercice 4  | 27       |
| 1.6.8    | Exercice 4.1  | 27       |
| 1.7      | Les boucles   | 28       |
| 1.7.1    | La boucle while   | 28       |
| 1.7.2    | La boucle for   | 29       |
| 1.7.3    | Un bonus de l'utilisation de la boucle for                  | 30       |
| 1.7.4    | Exercice 5  | 31       |
| 1.8      | Découper le programme en fonctions                          | 32       |
| 1.8.1    | C'est quoi une fonction?                                    | 32       |

|          |   |           |
|----------|---|-----------|
| 1.8.2    | Fonction simple                                 | 32        |
| 1.8.3    | Fonction avec plusieurs paramètres              | 33        |
| 1.8.4    | Fonction qui ne renvoie rien                    | 34        |
| 1.8.5    | Fonction qui renvoie plusieurs valeurs          | 34        |
| 1.8.6    | Pourquoi découper notre programme??             | 35        |
| 1.8.7    | Exercice 6                                      | 35        |
| 1.8.8    | Solution  | 35        |
| 1.9      | Les modules                                     | 37        |
| 1.9.1    | Voir toutes les fonctions d'un module           | 37        |
| 1.9.2    | Le module random                                | 38        |
| 1.9.3    | Quelles sont les modules disponibles??          | 38        |
| 1.9.4    | Exercice 7                                      | 39        |
| 1.10     | TP  | 39        |
| 1.10.1   | La solution                                     | 39        |
| 1.10.2   | Extension du programme                          | 40        |
| 1.10.3   | La solution                                     | 40        |
| 1.11     | Le help   | 41        |
| <b>2</b> | <b>Les objets</b>                               | <b>44</b> |
| 2.1      | Les listes                                      | 44        |
| 2.1.1    | Autres exemples                                 | 44        |
| 2.1.2    | Accéder à un élément de la liste                | 45        |
| 2.1.3    | Parcourir une liste                             | 46        |
| 2.1.4    | Exercice 8                                      | 46        |
| 2.1.5    | Opérations sur les listes                       | 46        |
| 2.1.6    | Exercice 9                                      | 47        |
| 2.1.7    | Accéder au dernier élément de la liste          | 47        |
| 2.2      | Les string                                      | 48        |
| 2.2.1    | Accéder à une lettre                            | 48        |
| 2.2.2    | Des opérations sur les strings                  | 48        |
| 2.2.3    | Opérations avancées                             | 49        |
| 2.2.4    | Exercice 10                                     | 50        |
| 2.3      | Les dictionnaires                               | 50        |
| 2.3.1    | Modifier la valeur d'un élément du dictionnaire | 51        |
| 2.3.2    | Ajouter un élément au dictionnaire              | 51        |
| 2.3.3    | Supprimer un élément du dictionnaire            | 51        |
| 2.3.4    | Parcourir un dictionnaire                       | 52        |
| 2.3.5    | Vérifier si une clé existe dans le dictionnaire | 52        |
| 2.3.6    | Exercice 11                                     | 52        |
| 2.4      | Les fichiers                                    | 53        |
| 2.4.1    | Ouvrir un fichier                               | 53        |
| 2.4.2    | Lire un fichier                                 | 53        |

# 1 La programmation avec Python 3

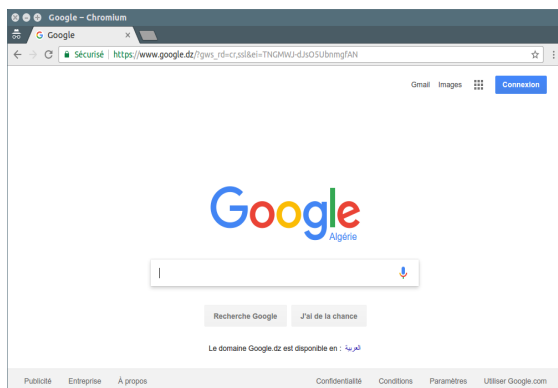
## 1.1 Introduction

Ceci est une petite introduction où on va parler des programmes et des langages de programmations.

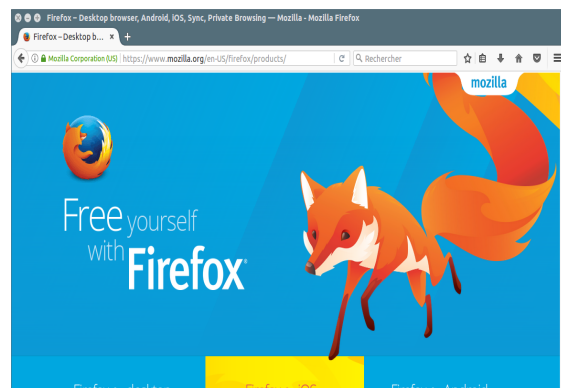
### 1.1.1 C'est quoi un programme??

Je vais commencer par donner une définition qui peut vous sembler un peu floue, mais cette description va s'éclaircir coûte que coûte au cours de notre formation.

Alors, un programme est un ensemble d'instructions exécuté par l'ordinateur pour qu'il puisse (l'ordinateur) accomplir une tâche bien précise. On a pour exemple des programmes : les navigateurs web (comme chromium, chrome, firefox), qui permettent de visiter les sites et de chatter en ligne; on a aussi les jeux vidéos, les logiciels de retouches de photos ... etc



(a) Google Chrome



(b) Mozilla Firefox



FIGURE 2 – Le jeu Assassin's Creed

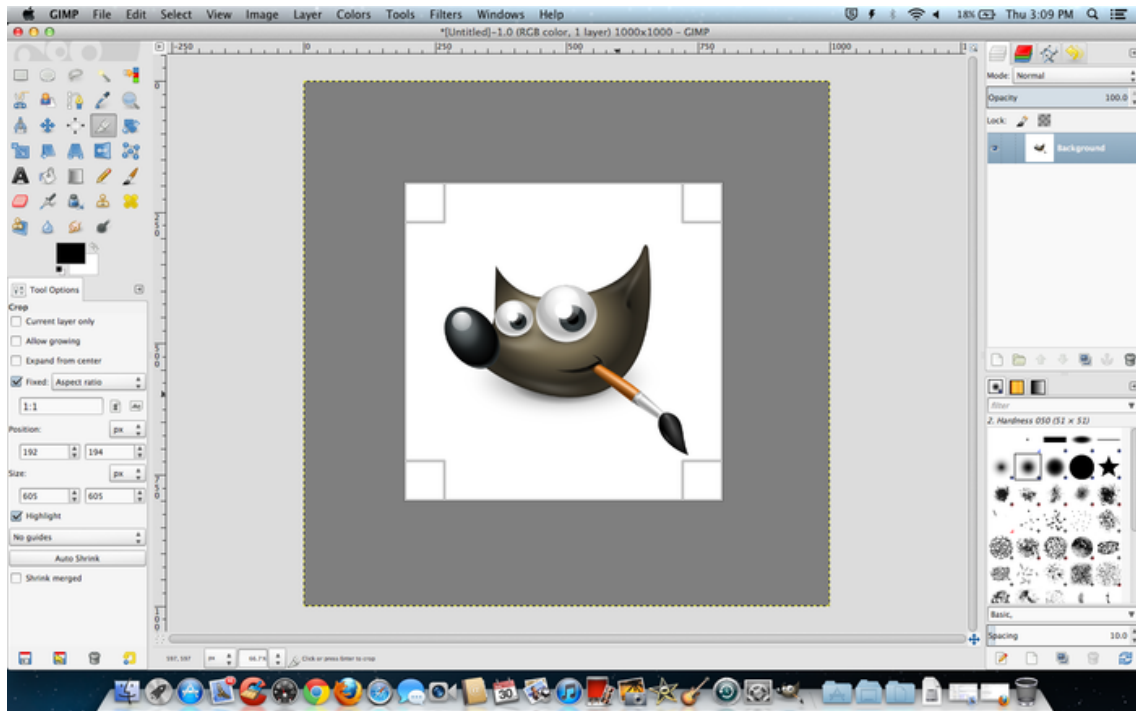


FIGURE 3 – Logiciels de retouches “Gimp”

### 1.1.2 Comment sont créés ces programmes??

L'ordinateur est une machine bête et disciplinée qui ne comprend qu'un seul langage, qui est le langage *binaire*. Ce langage est une succession de 1 et de 0 comme suit :

```
01101100001100110011000111101011101
```

Donc, normalement, pour créer des programmes, il fallait apprendre ce langage. Mais heureusement, les informaticiens ont eu la réflexion de créer d'autres langages intermédiaires (qu'on appelle langage de programmation) qui sont beaucoup plus faciles que le binaire. Ces langages ont tous le même but : permettre de créer des programmes plus facilement qu'en binaire.

Voici comment cela fonctionne :

- On écrit nos instructions en utilisant un langage de programmation.
- Les instructions sont traduites en binaire grâce à un programme de traduction.
- L'ordinateur peut alors lire le binaire et faire ce qu'on lui a demandé.

Voici une image qui illustre ce qu'on vient de dire :

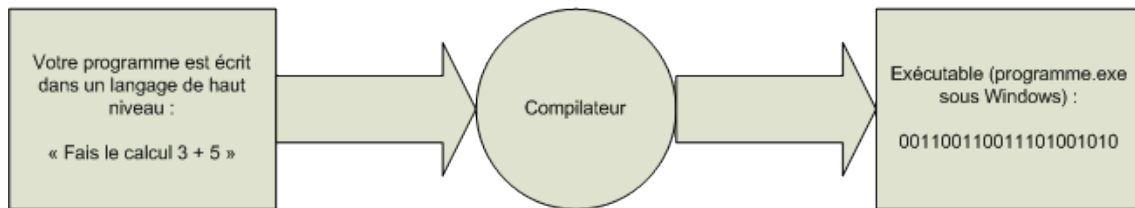


FIGURE 4 – Les étapes de création d'un programme



Une remarque très importante : les instructions que nous écrivons dans un langage de programmation sont appelé **Code Source**. Gardez bien ce terme à l'esprit, parce qu'on va beaucoup l'utiliser par la suite. Voici un exemple de code source python que nous écrirons plus tard :

```
#!/usr/bin/python3
import os
try:
    import psutil
except ImportError:
    import pip
    print("Installing missing packages (this will be done the first time only) :)")
    if os.getuid() == 0:
        pip.main(["install", "psutil"])
    else:
        pip.main(["install", "psutil", "--user"])
    print("Done.\nRerun the command now.")
    exit(0)
import argparse
from time import sleep, localtime

# This function will execute until the end of the copy
def copy(time, verbose):
    while True:
        bytes_read = psutil.disk_io_counters().read_bytes / (1024*1024)
```

FIGURE 5 – Coude source Python

### 1.1.3 Les langages de programmation les plus connus



(a) Logo C++



(b) Logo Java



(c) Logo Python

### 1.1.4 Compilé vs Interprété

Nous avons dit tout à l'heure que les instructions sont traduites en binaire pour que l'ordinateur puisse les comprendre et les exécuter. Mais ce qu'on a pas dit c'est qu'il y a deux façons de traduire les instructions : *Compilation* et *Interprétation*.

**Compilation :** Avec ce processus, toutes nos instructions sont traduites en binaires à la fois. Ce qui nous donne en résultat un gros fichier binaire qui peut être exécuté directement par l'ordinateur.

**Interprétation :** Avec cette technique, on ne traduit pas tous les instructions en binaires, mais on va plutôt traduire une instruction par instruction à chaque fois qu'on veut exécuter notre programme. Par exemple, si mon code source contient 3 instructions, quand je veux l'exécuter je traduit la première instruction en binaire, et je la passe à l'ordinateur pour qu'il l'exécute; ensuite je traduit la deuxième instructions, et je la passe à l'ordinateur pour qu'il l'exécute ... et ainsi de suite jusqu'à la traduction de toutes mes instructions.

Le programme qui fait l'interprétation est appelé l'*interpréteur*, et dans notre cas on va utiliser l'interpréteur *Python*.

### 1.1.5 Le langage Python

Python est un langage de programmation interprété, *Open Source*, facile à apprendre, et très pratique. On cite quelques avantages de ce langage :

- Python est facile à apprendre et son code est facile à lire.
- Il est multiplate-forme, il marche sous Linux, Windows, et Mac OS.
- C'est un langage orienté objet (on en parlera de ça dans les prochaines séances).

- Il a une bibliothèque tierce qui permet de faire de la programmation web d'une manière très flexible. Cette bibliothèque c'est *Django*.
- Il permet de développer très rapidement en utilisant peu de code, et de crée des applications très complexe avec facilité. Tous ça grâce à son style et à sa bibliothèque standard très complète.
- Il est très utilisé par les pentesters et les hackers.

Pour finir, il faut savoir que les géants de l'informatique comme Google, Yahoo, NASA préfèrent le langage python; et ce langage est intégré à la quasi totalité des distributions Linux (C'est-à-dire que vous pouvez l'utiliser dans ces distributions sans devoir l'installer).



Juste pour information, il en existe deux versions de python qui ne sont pas compatibles entres eux. Ces deux versions sont les 2.x et les 3.x . Nous on va utiliser la version 3.x qui est le future de python.

## 1.2 Les logiciels nécessaires pour la programmation

Pour pouvoir créer des programmes en python, il existe deux manières :

1. Soit on écrit nos instructions dans un fichier texte, et puis on appel l'interpréteur python pour qu'il exécute nos instructions et nous affiche le résultat. Cette méthode comporte de nombreux avantages, et elle est très flexible, mais elle est compliqué (surtout pour les débutants). Donc nous allons optez pour la deuxième méthode dans cette formation.
2. Cette méthode consiste à utiliser un programme appelé *IDE* (Integrated Development Environment), ou *EDI* en français (Environnement de Développement Intégré). Ce programme contient tous ce dont on a besoin pour créer nos programmes : une partie ou on peut écrire nos instructions, des boutons pour demander à python d'exécuter nos instructions, et finalement une partie ou le résultat de l'exécution est affiché.

L'IDE qu'on va utiliser est appelé *PyCharm* (Figure 7). C'est un IDE *Open Source* et très puissant.



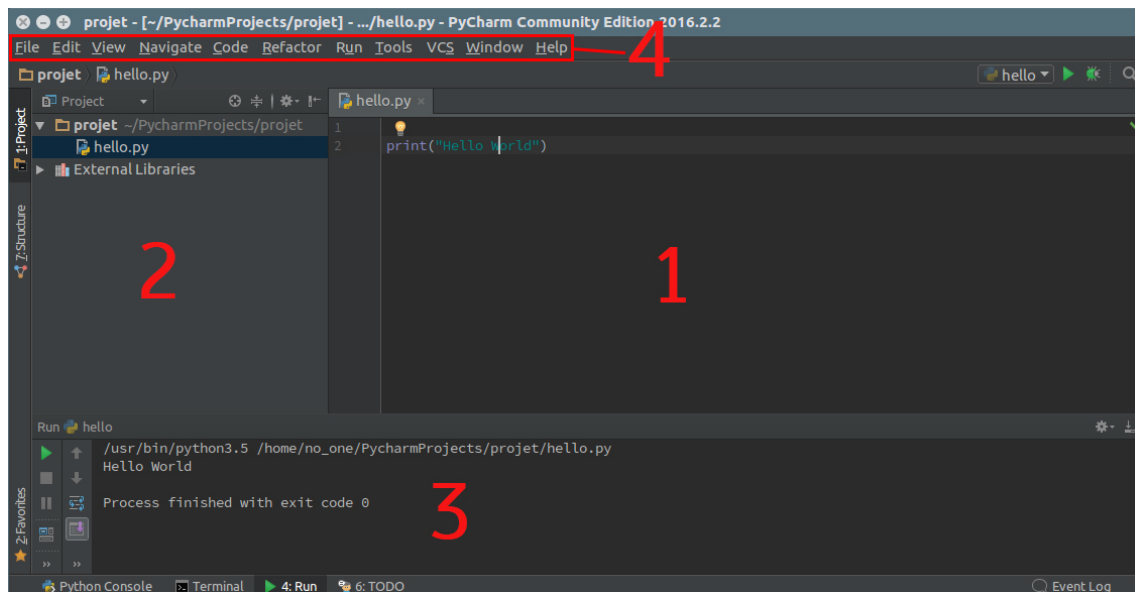


FIGURE 7 – Pycharm

PyCharm est composé essentiellement de 4 partie :

**La zone principale (1) :** c'est dans cette zone où on écrira notre code source (nos instructions).

**La liste des fichiers du projet (2) :** dans cette zone on peut voir tous les fichiers de notre projet. On voit bien que le projet présenté dans la [Figure 7](#) contient un seul fichier seulement.

**La zone d'affichage (3) :** c'est cette zone qui nous affichera le résultat d'exécution de notre programme, où les éventuelles erreurs s'il y en a.

**La barre d'outils (4) :** elle contient de nombreux boutons qui font différentes choses : exécuter le projet, configurer l'IDE ... etc, mais nous on utilisera que quelques boutons.

### 1.2.1 Installer Python et PyCharm sous Ubuntu Linux

Python est installé par défaut dans toutes les distributions Linux. Par contre, l'IDE PyCharm il faut l'installer. Pour cela, il suffit d'ouvrir un terminal (Ctrl+alt+t) et exécuter ces commandes :

```
sudo add-apt-repository ppa:ubuntu-desktop/ubuntu-make -y
sudo apt-get update
sudo apt-get install ubuntu-make -y
umake ide pycharm
```

Il vous demandera après de choisir un emplacement pour l'installation, tapez juste entrer sans rien changer.

```
Choose installation path: /home/no_one/.local/share/umake/ide/pycharm
Downloading and installing requirements
46% |#####|
```

FIGURE 8 – Installation PyCharm

Et voila, PyCharm est installé sur votre machine. Il suffit juste de le lancer en appuyant le bouton windows (Figure 9) et tapez `pycharm` (Figure 10) puis tapez entrer.

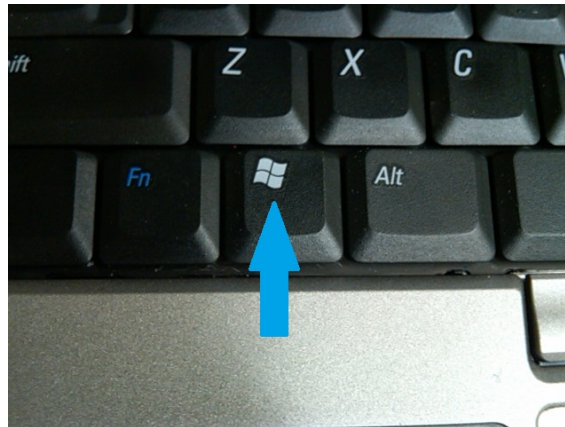


FIGURE 9 – Bouton windows

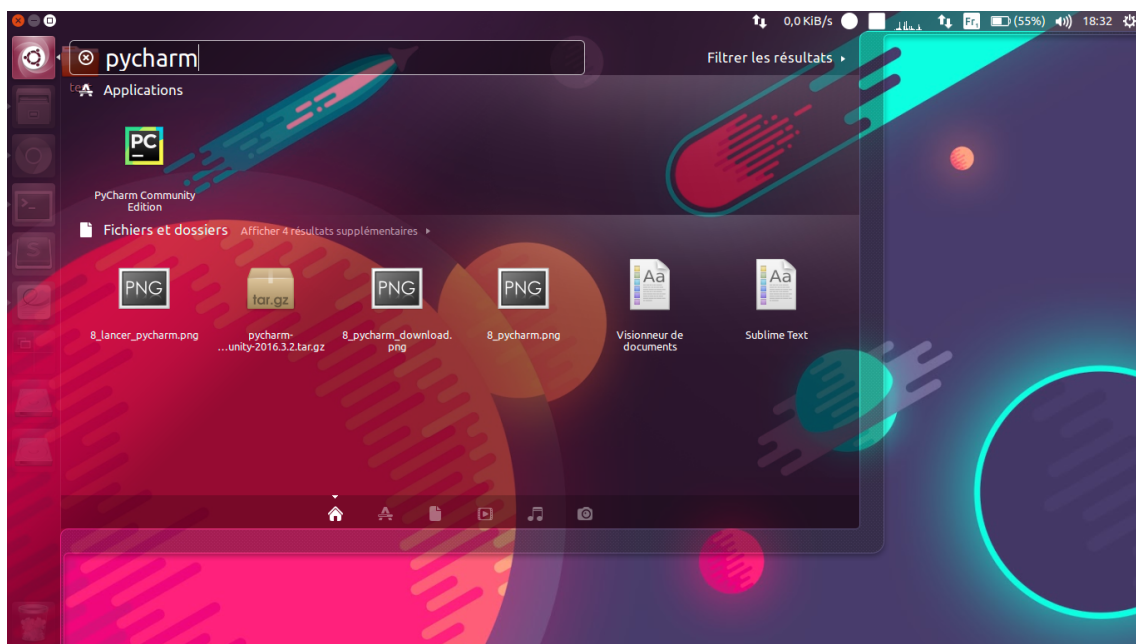


FIGURE 10 – Lancer PyCharm

### 1.2.2 Installer Python et PyCharm sous Windows

Python n'est pas installé par défaut sur Windows, donc il faut installer Python et l'IDE PyCharm.

Pour installer Python, visitez ce lien <https://www.python.org/downloads/> et téléchargez la dernière version de Python (au moment du déroulement de la formation, c'est la version 3.6). Installez là quand vous finissez le téléchargement.

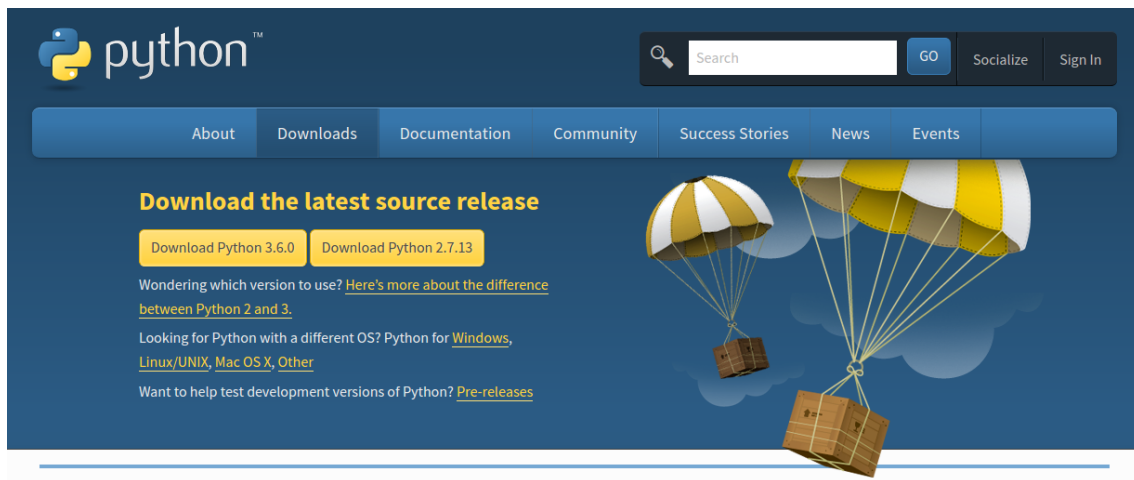


FIGURE 11 – Lien de téléchargement de Python

Pour PyCharm, télécharger la version `community` sur ce lien <https://www.jetbrains.com/pycharm/download/> (Figure 12) et installez là.

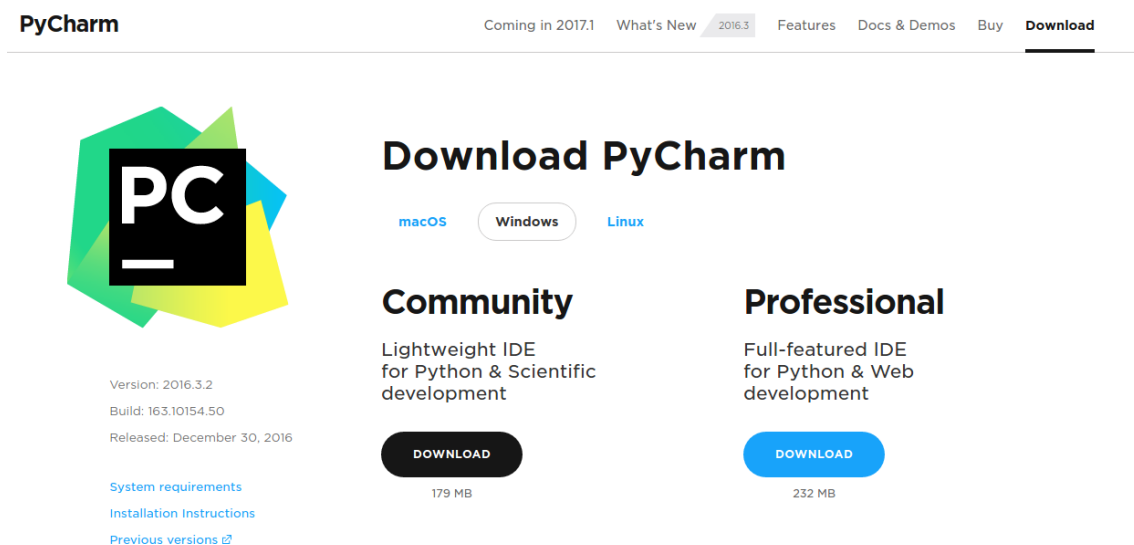


FIGURE 12 – Lien de de téléchargement PyCharm

## 1.3 Hello World

Nous allons écrire maintenant notre premier programme. Un programme petit et simple pour comprendre le fonctionnement de ce langage fascinant.

On commence par créer un nouveau projet en cliquant sur la barre d'outils sur `File -> New Project`. Ensuite on choisit où on veut placer notre projet, et la version de l'interpréteur qu'on veut utiliser (Figure 13). Choisissez n'importe quelle version 3.x disponible sur votre ordinateur (Pour moi, j'utilise la version 3.5).

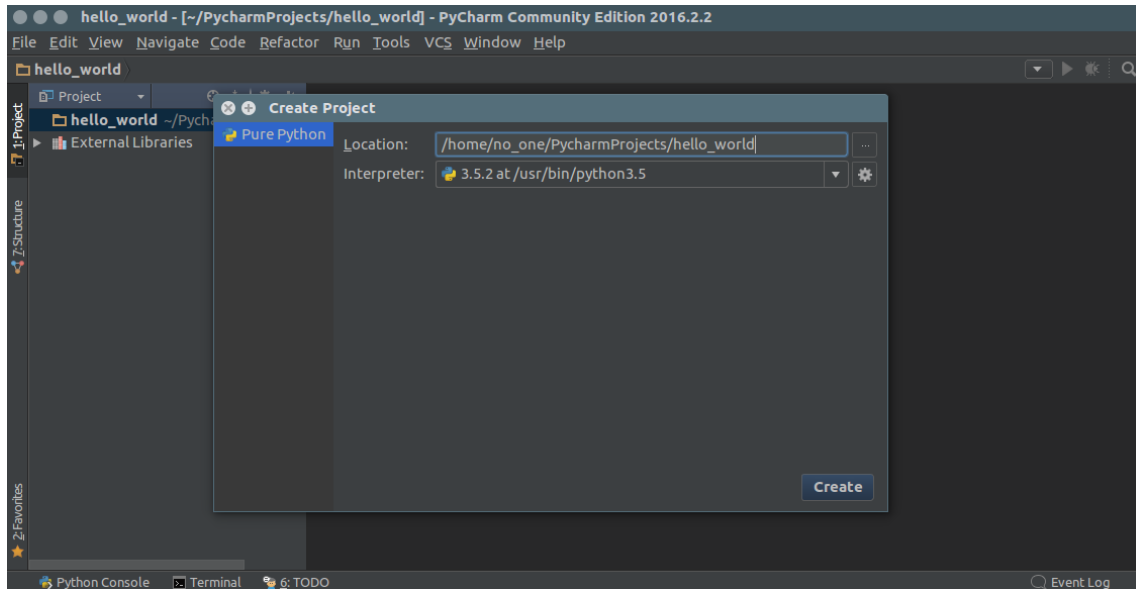


FIGURE 13 – Création d'un nouveau projet

On a créé notre projet, mais ce dernier est vide, il ne contient aucun fichier. Nous allons maintenant ajouter un fichier où nous allons écrire nos instructions (programmes). On fait un clic sur la barre d'outils sur `File -> New` et on choisit `Python File`. On lui donne un nom à notre fichier, et on clique sur `OK` (Figure 14).

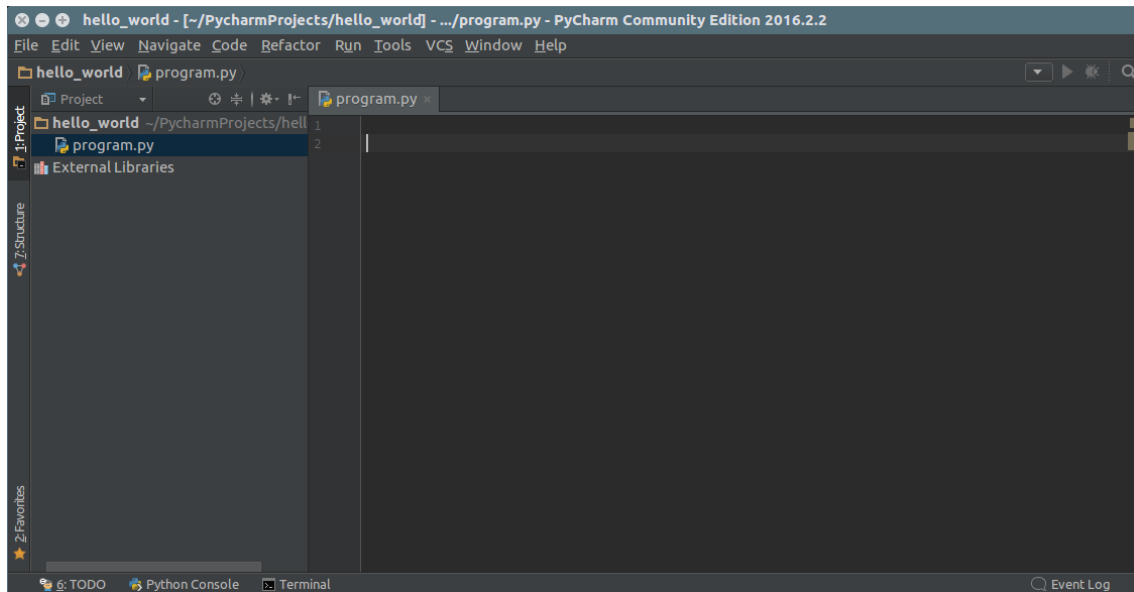


FIGURE 14 – Création d'un fichier pour écrire les instructions

Dorénavant, je vais seulement vous afficher les codes que vous devez écrire dans la zone principale (la zone 1, Figure 7).

On arrive maintenant à la programmation, voici votre premier programme :

```
1 | print("Hello World!")
```

Écrivez ce code source dans la zone principale (zone n° 1 Figure 7), et demandez à PyCharm d'exécuter notre programme en cliquant sur la barre d'outils sur **Run** -> **Run**, et on choisit le fichier à exécuter (on en a un seul dans notre cas). Voici le résultat de l'exécution :

---

```
/usr/bin/python3.5 /home/no_one/PycharmProjects/hello_world/program.py
Hello World!
```

```
Process finished with exit code 0
```

---

Alors, la ligne que nous avons écrit dans l'IDE `print("Hello World!")` est une instruction qui demande à l'ordinateur d'afficher `Hello World!` à l'écran. Nous pouvons écrire n'importe quoi à la place de `Hello World!` et l'ordinateur se chargera d'afficher le message que nous avons demandé.

Avant de continuer, nous allons détailler ce que PyCharm nous a affiché :

**/usr/bin/python3.5** c'est le chemin vers l'interpréteur python que PyCharm a utilisé pour exécuter notre programme. La majorité des programmes sous *Linux* sont installés dans le dossier `/usr/bin/`.

**/home/no\_one/PycharmProjects/hello\_world/program.py** ça c'est notre fichier (là où on est entrain d'écrire notre code source). Donc d'après cette ligne, on comprend que `python3.5` est entrain d'exécuter ce fichier.

**Hello World!** c'est le message que nous avons demandé à python d'affiché (on voit qu'il a bien obéi).

**Process finished with exit code 0** ceci est un message affiché par PyCharm pour nous indiquer que notre programme s'est terminé avec succès.

Donc, `print()` est l'instruction responsable d'afficher un message à l'écran. On lui donne entre les `"""` le message qu'on veut afficher. Essayez de Remplacer `Hello World!` par `Salut tout le monde` et ré-exécuté pour voir le nouveau résultat.

### 1.3.1 Les commentaires

Avant de terminer cette section avec un TP, nous allons parler des commentaires. Dans n'importe quel langage de programmation (y compris python), on a la possibilité d'écrire des commentaires.

Alors c'est quoi les commentaires? Ce sont des bouts de textes qu'on écrit dans notre code source et qui permettrons d'expliquer le fonctionnement de notre code source. Ça peut vous semblez bizarre et inutile, mais sachez que c'est très commun qu'on revient à notre programme après quelques semaines pour l'améliorer ou pour le modifier, et sans les commentaires on sera obligé de tout repenser du début. Ne vous faites pas avoir par le fait que c'est vous qui a crée ce programme, car même si vous comprenez tous sur votre programme maintenant, vous oubliera après un certain temps.

Les commentaires ne serviront pas seulement à comprendre son code seulement, mais c'est très utile aussi quand vous travaillez en groupe pour crée un programme. Quand vous utilisez les commentaires, vous ne serez pas obligé d'expliquer la partie que vous avez écrit à vos camarades.

J'insiste exprès sur le fait d'utiliser les commentaires, ils sont très très utiles et indispensables.

Alors, on sait ce que sont les commentaires, mais comment les utiliser avec python ? Voici un exemple sans plus tardé :

```
1 | # Cette instruction affiche Hello World! a l'ecran.  
2 | print("Hello World!")
```

Les commentaires commence par un `#`, tous ce qui suit ce caractère est ignoré par python. Donc on peut écrire autant de ligne de commentaire qu'on veut, il faut juste précédé chaque ligne avec un `#`.

### 1.3.2 Exercice 1

Bon, on a assez bavardé, vous allez écrire votre vrai premier programme crée par vous même. Voici l'exercice :

- Créez un nouveau projet et nommé le "exercice 1".

- Créez un fichier dans ce projet où nous allons mettre notre code source. nommé ce fichier "main".
- Débrouillez vous pour que le programme affiche :

---

Salut tout le monde.

On est dans une formation python, et on va cree des programmes super cool :D.

---

- N'oubliez pas d'utiliser les commentaires pour expliquer votre code.

### 1.3.3 Solution de l'exercice

```
1 | # Ces deux instructions affiche des messages a l'ecran.  
2 | print("Salut tout le monde.")  
3 | print("On est dans une formation python, et on va cree des programmes super co
```

## 1.4 Les variables

Les variables sont une notion fondamentale en informatique et il sont utilisé dans tous les langages de programmation sans exception. Il ont plusieurs définitions, mais je pense que celle qu'on va voir est la plus simple.

En faite, votre ordinateur est une armoire géante qui contient des millions et des millions de tiroirs, et chaque tiroir est utilisé pour stocker des informations. Donc quand vous voulez stocké une information (disons votre age) pour l'utilisé plus tard dans votre programme, vous demandez à votre ordinateur de vous allouer un tiroir. Ce tiroir vous serra allouer jusqu'à la fin de votre programme.

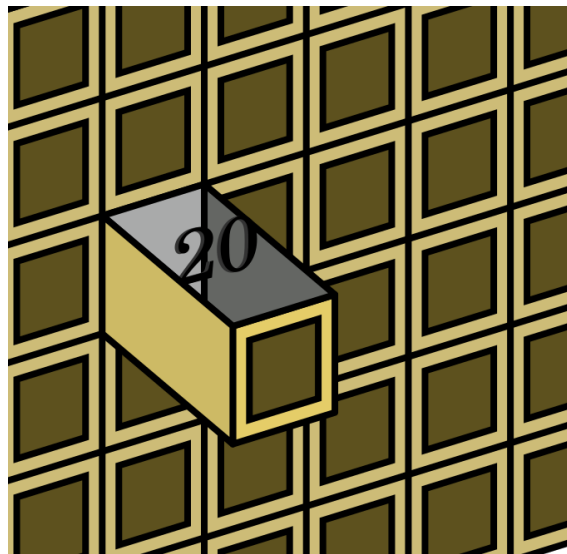


FIGURE 15 – Une tiroir qui stocke notre information

Ce tiroir est le votre, vous pouvez placez des information dedans, les consulter, les retirer.



Donc, où sont les variables dans tous ça??

Les variables sont des noms qu'on associé aux tiroirs que nous avons demandé. Donc, dans l'exemple précédent, nous avons utilisé un tiroir pour stocker un age, donc nous allons nommé ce tiroir `age`. Et à chaque fois qu'on veut consulter ou modifier le contenu de ce tiroir, (supposons vous avez oublié votre age) on utilise le mot `age`.



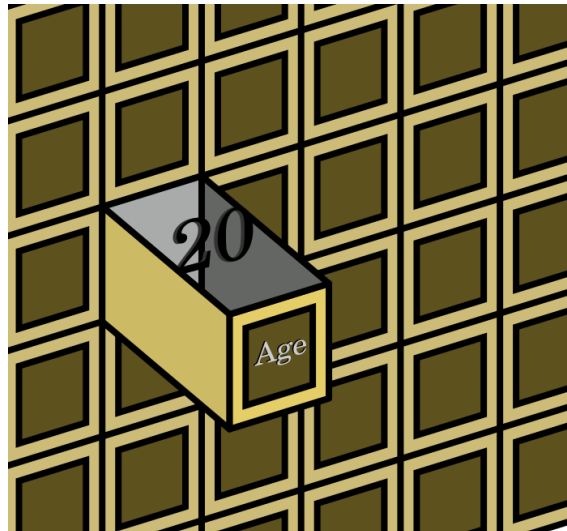


FIGURE 16 – Une tiroir qui stocke notre information, et qui est accessible par le nom `age`

### 1.4.1 Créer et utiliser les variables avec python

Pour créer une variable avec python, rien de plus simple. Il suffit d'écrire le nom de notre variable :

```
1 | age = 20
```

Avec cette instruction, nous avons maintenant un tiroir réservé à nous, ce tiroir est accessible en écrivant le mot `age` dans python.

Si on veut voir le contenu de notre tiroir (variable), on peut utiliser l'instruction qu'on a vu précédemment `print()` (celle qui permet d'afficher à l'écran) :

```
1 | age = 20
2 | # Cette instruction va afficher mon age.
3 | print(age)
```

On peut même modifier le contenu de notre variable à n'importe quel moment :

```
1 | age = 20
2 | # Cette instruction va afficher mon age.
3 | print(age)
4 |
5 | # Ces deux instructions vont me rendre plus vieux,
6 | # et afficher mon nouveau age.
7 | age = 30
8 | print(age)
```

Voici ce qui nous sera affiché par PyCharm après l'exécution de notre programme :

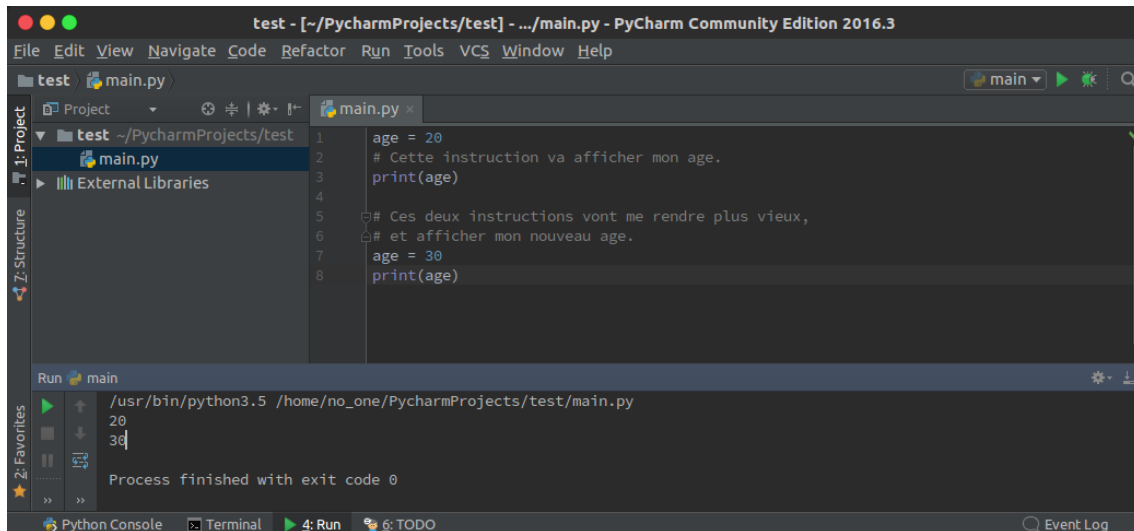


FIGURE 17 – Affichage de la variable age

## 1.4.2 Quel genre d'information on peut stocker dans nos variables

Il existe essentiellement 3 types d'information qu'on peut stocker dans les variables :

**Les entiers :** comme ce que nous avons stocké dans la variable `age` que nous avons utilisé jusque là. Les entiers sont appelé `int` en python.

**Les valeurs décimales :** des nombres à virgules, (comme : 3.453). Ces nombres sont appelé `float` en python. Exemple :

```
1 pi = 3.14
```

**Du texte :** on les appelle les chaînes de caractères ou `str` en python. On les définit on les entourant par `"` ou `'`. Exemple :

```

1 prenom = "Amine"
2 prenom2 = "Moustapha"
3 nom = 'Malaoui'

```

## 1.4.3 Afficher les variables

Vous savez déjà qu'on peut afficher le contenu d'une variable avec `print()`, mais ce que vous ne savez pas c'est que `print()` permet d'afficher plusieurs variables à la fois. Il suffit juste de les séparer par des `,` :

```

1 # Je cree mes variables
2 prenom = "Amine"
3 nom = "Malaoui"
4
5 print("Salut, je m'appelle", prenom, nom)

```

En exécutant notre programme, PyCharm nous affichera ça :

```
/usr/bin/python3.5 /home/no_one/PycharmProjects/test/main.py
Salut, je m'appelle Amine Malaoui
```

```
Process finished with exit code 0
```

#### 1.4.4 Récupérer une saisie

Ce que nous avons fait jusque là c'est de créer nos variables, et de les donner des valeurs directement dans le code source. Ce que nous allons apprendre maintenant c'est de demander de l'utilisateur de donner une valeur à notre variable au cours d'exécution.

Pour cela on va utiliser l'instruction de récupération de saisie, qui est `input()` :

```
1 # On demande a l'utilisateur de nous donner son prenom,
2 # et on stock son prenom apres dans la variable prenom.
3 prenom = input("Veuillez entrer votre prenom : ")
4
5 # On dit bonjour a l'utilisateur.
6 print("Bonjour", prenom)
```

Ce que fait ce code c'est qu'il affiche à l'utilisateur `Veuillez entrer votre prenom :`, et il lui donne la main après pour qu'il entre son prénom. Son prénom est maintenant stocker dans la variable `prenom`. Voilà ce que PyCharm nous affiche après l'exécution :

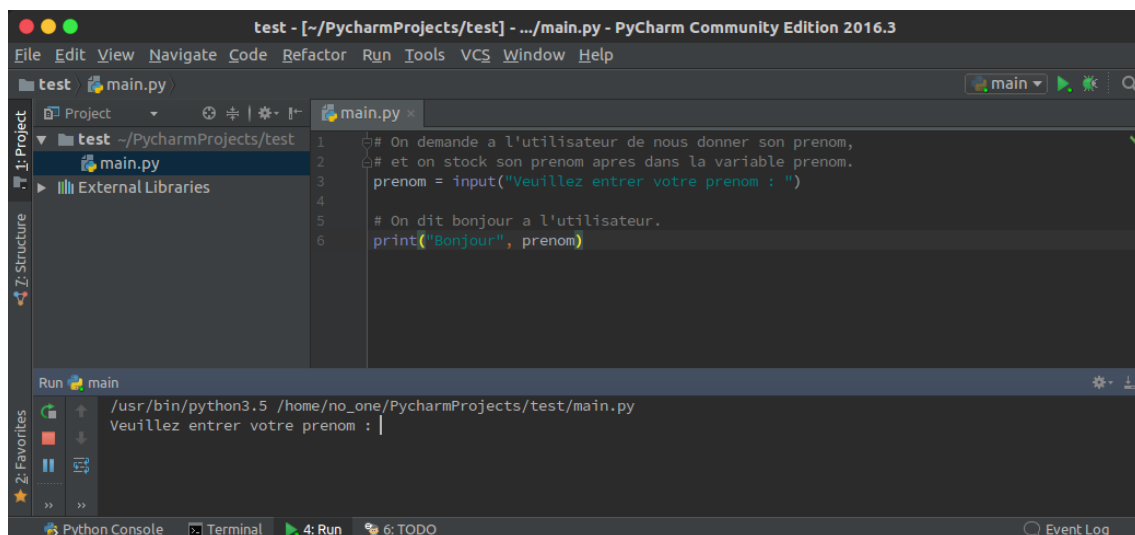


FIGURE 18 – PyCharm attend qu'on entre le prénom

Dès que l'utilisateur entre son nom et tape `Entrée`, PyCharm lui affiche `Bonjour` suivi du prénom de l'utilisateur :

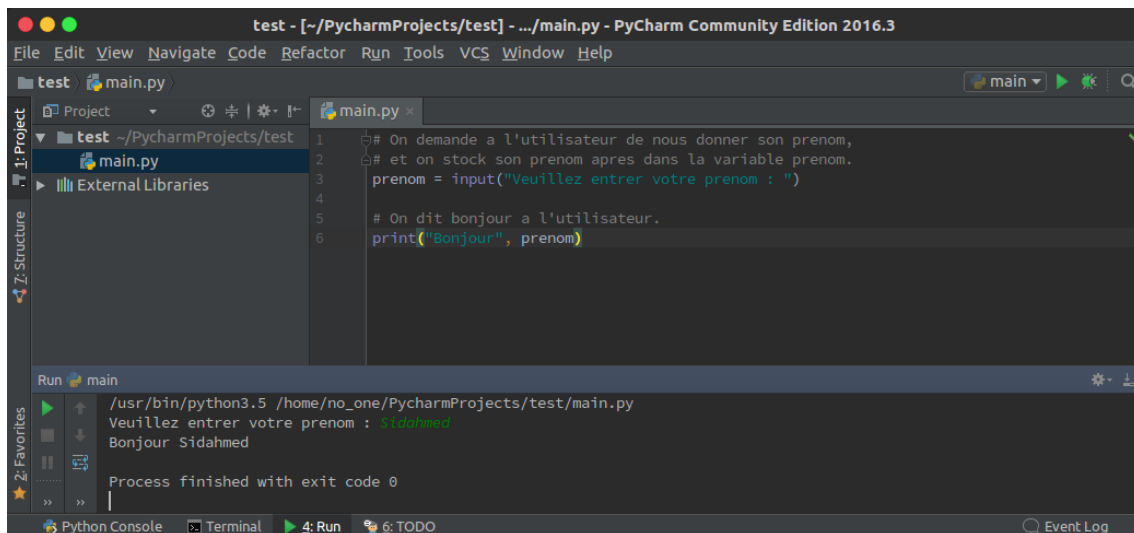


FIGURE 19 – PyCharm affiche Bonjour suivi du prénom

### 1.4.5 Comment nommé les variables

On va parler de la meilleur manière de nommé les variables avant de finir cette partie. vous avez juste quelques règles à suivre, et tout passera à merveille :

- Le nom d'une variable doit commencer par une lettre. Exemple :

```

1  # Faux
2  1prenom = 20
3
4  # Juste
5  prenom1 = 20
6
7  # Juste
8  prenom = 20

```

- Donner toujours des noms explicites à vos variables. C'est à dire en observant le nom de votre variable, il faut que vous soyez capable de comprendre son fonctionnement. Croyez moi, ça vous sera extrêmement utiles quand vous réexaminer votre code.

Par exemple, si on veut stocker l'âge d'un utilisateur :

```

1  # Faux, car c'est pas claire que x designe un age.
2  x = 20
3
4  # Juste
5  age = 20

```

- Si le nom de votre variable doit être composé de plusieurs mots, séparer les mots par un `_`. Exemple :

```

1  # Faux
2  notedemath = 15
3

```

```
4 | # Juste
5 | note_de_math = 15
```

- Et enfin, n'utilisez pas de caractères avec accents (comme é, è, à ...) car ça peut vous poser des problèmes plus tard.

Enfin, nous avons terminé de parler sur les variables. Nous allons faire un petit exercice avant de passer à la section suivante.

### 1.4.6 Exercice 2

Ne sautez pas ces exercices, c'est très important de pratiquer en informatique.

Voici l'énoncé de l'exercice :

- Créez un nouveau projet et nommé le "exercice 2", et créez un fichier "main.py" dans ce projet.
- Débrouillez vous pour que le programme demande le "prenom", "nom" et "age" de l'utilisateur. Puis affichez tous ça dans une phrase du genre :  
Tu t'appelles Yanis Khelloufi et tu as 22 ans..

## 1.5 les opérateurs arithmétique

Les ordinateurs sont des vrai machines à calcul, c'est ce qu'il savent faire le mieux. Donc dans cette partie nous allons apprendre à faire des calculs avec python.

Il y a dans python 5 symboles qui nous permettent de faire des opérations arithmétique :

- `+` fait l'addition.
- `-` fait la soustraction.
- `*` fait la multiplication.
- `/` fait la division entière.
- `//` fait la division euclidienne.
- `%` donne le reste de la division euclidienne (on appelle ça le *modulo*).
- `**` calcul la puissance.

### 1.5.1 Addition, soustraction, multiplication

Pour l'addition et la soustraction et la multiplication, rien de plus simple, on utilise seulement les symboles appropriés, et tout passe à merveille. Exemple :

```
1 x = 10 + 5
2 y = 21 - 7
3 z = 9 * 9
4
5 # Ceci affichera : 15 14 81
6 print(x, y, z)
```

### 1.5.2 Division entière et Division euclidienne

Pour voir la différence, un exemple vaut mieux qu'un long discours :

```
1 division_entiere = 10 / 3
2 division_euclidienne = 10 // 3
3
4 # Ceci affichera : 3.3333333333333335 3
5 print(division_entiere, division_euclidienne)
```

### 1.5.3 Modulo (reste de la division euclidienne)

Exemple :

```

1 | x = 10 % 3
2 |
3 | # Ceci nous affichera 1. Car le reste de la division de 10 par 3 donne 1.
4 | print(x)

```

### 1.5.4 La puissance

Rien de plus simple. Exemple :

```

1 | x = 5 ** 2
2 |
3 | # Ceci affichera 125
4 | print(x)

```

### 1.5.5 Opérations entre variables

Sachez bien que toutes les opérations que nous avons vu précédemment sont aussi applicables sur les variables. Donc on peut faire des calculs entres variables. Exemple :

```

1 | a = 10
2 | b = 15
3 | resultat = a + b
4 |
5 | # Ceci affichera : 10 + 15 = 25
6 | print(a, "+", b, "=", resultat)

```

### 1.5.6 Bonus

Dans la majorité des autres langages de programmation, pour permuter deux variables (échanger leurs contenus), il faut toute une série d'opérations.

Exemple : disons on a les variables `a` et `b`, pour permuter leurs valeurs dans les autres langages de programmation, il faut faire ça :

```

1 | # Il faut ces trois opérations pour permuter les valeurs
2 | # de a et b
3 | variable_temporaire = a
4 | a = b
5 | b = variable_temporaire

```

Alors en python, il suffit juste de faire :

```

1 | # cette instruction permutent les valeurs de a et b
2 | a, b = b, a

```

Vous pouvez créer les variables `a` et `b`, essayez ce code, et afficher ces deux variables pour voir le résultat.

### 1.5.7 Un petit problème!

Essayez ce code :

```
1 |
2 | date_de_naissance = input("Veuillez entrer votre date de naissance : ")
3 | age = 2017 - date_de_naissance
4 |
5 | print("Votre avez", age, "ans")
```

Malheureusement, PyCharm n'a pas pu exécuter notre programme, et il nous affiche une erreur :

---

```
/usr/bin/python3.5 /home/no_one/PycharmProjects/test/main.py
Veuillez entrer votre date de naissance : 1994
Traceback (most recent call last):
  File "/home/no_one/PycharmProjects/test/main.py", line 3, in <module>
    age = 2017 - date_de_naissance
TypeError: unsupported operand type(s) for -: 'int' and 'str'

Process finished with exit code 1
```

---

Ce qui s'est passé c'est que l'instruction `input()` récupère la saisie sous forme de texte. Donc ce qu'elle a stocké dans la variable `date_de_naissance` n'est pas le nombre `1994` mais plutôt le texte (chaînes de caractères) `"1994"`. Donc quand Python arrive à l'instruction de la soustraction, il essaye de soustraire un texte d'un entier, et là ça plante, car ce n'est pas possible.

La solution consiste à convertir notre variable `date_de_naissance` en un entier. C'est extrêmement simple, il suffit juste d'utiliser l'instruction `int()`. Voici à quoi rassemblerai notre nouveau code source :

```
1 | date_de_naissance = input("Veuillez entrer votre date de naissance : ")
2 | # On converti notre variable en un entier.
3 | date_de_naissance = int(date_de_naissance)
4 |
5 | age = 2017 - date_de_naissance
6 |
7 | print(age)
```

Et cette fois, le programme marche à merveille.

N'oubliez pas, dorénavant, quand vous utilisez `input()`, mettez en tête qu'elle récupère la saisie sous forme de texte. Pensez donc à convertir votre variable si vous voulez l'utiliser comme entier (utilisez `int()`) ou nombre à virgule (utilisez `float()`).

### 1.5.8 Exercice 3

Voici l'énoncé de l'exercice :



- Créez un nouveau projet et nommé le “exercice 3”, et créez un fichier “main.py” dans ce projet.
- Demandez l’age de l’utilisateur, et dite lui à quel année il aura 100 ans.

## 1.6 Les conditions

Les conditions sont des concepts fondamentaux de n'importe quel langage de programmation. Mais il sont un petit peu difficile à comprendre au début, mais vous verrez que dès que vous comprenez les conditions, vous serez déjà capable de créer un programme disons intelligent et qui réagit selon la situation.

Par exemple : vous pouvez demander l'âge de l'utilisateur, et puis lui afficher un message selon son âge. Si il a plus de 18 ans, vous lui affichez `Vous êtes majeur`, sinon vous lui affichez `Vous êtes mineur`.

Avant de commencer de parler des conditions, il y a un ensemble de symboles que vous devez apprendre par cœur. Ces symboles sont utilisés pour réaliser les conditions :

| Symbole            | Signification           |
|--------------------|-------------------------|
| <code>==</code>    | est égal à              |
| <code>&gt;</code>  | est supérieur à         |
| <code>&lt;</code>  | est inférieur à         |
| <code>&gt;=</code> | est supérieur ou égal à |
| <code>&lt;=</code> | est inférieur ou égal à |
| <code>!=</code>    | est différent de        |

TABLE 1 – Symboles de comparaison

### 1.6.1 Le if

Pour réaliser une condition minimal en python, il faut juste suivre un schéma simple, le voici :

```
1 if condition :
2     instruction1
3     instruction2
4     .
5     .
6     .
7     instructionN
```

Donc, la condition `condition` sera évalué par python, si elle est vrai, il exécute toutes les instructions qui se trouvent dans le bloc du `if`, et si elle est fausse, alors il va sauter toutes les instructions qui se trouvent dans le bloc du `if`.



Attention, il faut que les instructions soient décalé par rapport à notre `if` comme le montre l'exemple précédent. C'est comme ça que python comprend quelles sont les instructions qui appartiennent au `if` et quelles sont les instructions qui appartiennent au programme principale.



Juste pour titre informatif, ce décalage est appelé *Indentation*, et l'ensemble d'instruction qui se trouvent dans le `if` est appelé *Un bloc de code*.

On donne un exemple :

```
1 age = 20
2
3 # On test si l'age est superieur a 18 pour afficher un message.
4 if age > 18 :
5     print("Vous etes majeur")
6
7 print("Au revoir")
```

Si on exécute ce code, on voit que le message `Vous etes majeur` est affiché, et même le message `Au revoir` est affiché. Mais si on modifie l'âge pour lui donner une valeur inférieure à 18, alors seulement le message `Au revoir` est affiché, car cette instruction n'est pas inclut dans le bloc du `if`.

### 1.6.2 Le else

Pour enrichir encore plus notre code, on peut utilisé le `else`. Ça permet de désigner un bloc sinon, ce bloc sera exécuté si la condition évalué dans le `if` est fausse. Un exemple :

```
1 age = 20
2
3 # Si l'age est supérieur a 18 on affiche : Vous êtes majeur.
4 # Sinon (c'est a dire l'age inférieur ou égale a 18) alors
5 # on affiche : Vous êtes mineur.
6 if age > 18 :
7     print("Vous êtes majeur")
8 else :
9     print("Vous êtes mineur")
10
11 print("Au revoir")
```

Si on modifie l'âge pour qu'il soit inférieur ou égale à 18, on remarquera que cette fois c'est le message `Vous etes mineur` qui sera affiché. Le message `Au revoir` est toujours affiché, car il n'appartient ni au `if` ni au `else`.

### 1.6.3 Le elif

Le `elif` est une abréviation de `else if` qui veut dire *Sinon Si*. Il se situe entre le `if` et le `else`, et ça permet de tester une autre condition.

Ça peut vous semblez abstrait, mais un exemple va éclaircir le concept :

```

1 age = 20
2
3 # Si l'age est superieur a 70 alors on affiche : Salut papi
4 # Sinon si l'age est superieur a 40 alors on affiche : Salut monsieur
5 # Sinon si l'age est superieur a 18 alors on affiche : Salut jeune
6 # Sinon on affiche le message qui se trouve dans le else.
7 if age > 70 :
8     print("Salut papi")
9 elif age > 40 :
10    print("Salut monsieur")
11 elif age > 18 :
12    print("Salut jeune")
13 else :
14    print("Tu n'appartient à aucune catégorie, dégage -_-")
15
16 # Ce message est toujours affiché.
17 print("Au revoir")

```

En utilisant le `elif`, on pourra re-tester la valeur de la variable `age` une autre fois. Amusez vous à changer la valeur de la variable `age`, et vous verrez qu'à chaque fois l'un des 4 messages est affiché.



Faites très attention, un seule bloc d'instruction est exécuté parmi tous ces 4 blocs. Donc si on donne la valeur `70` à notre variable `age`, seul bloc de code du `if` sera exécuté, et tout les autres `elif` et le `else` seront ignoré.

### 1.6.4 Comparez les chaînes de caractères

Sachez que c'est testés qu'on vient de voir ne sont pas applicables seulement au nombres, on peut aussi les appliquer au chaînes de caractères. Exemples :

```

1 mot_de_passe = input("Veuillez entrer votre mot de passe: ")
2
3 if mot_de_passe == "MonSuperMotDePasse" :
4     print("Bienvenu")
5 else
6     print("Dégaaaaaaaaaage imposteur")

```

### 1.6.5 Plusieurs conditions à la fois

Jusque là, nous avons tester une seule condition dans le `if` ou le `elif`, par exemple "Si l'age est supérieur à 18". Mais comment faire si on veut tester "Si l'age est supérieur à 18 et inférieur à 40 au même temps"? Pour cela, il suffit d'utiliser les opérateurs logique. Ces opérateurs permettent de combiner plusieurs condition à la fois.

Il y a un ensemble de règles à apprendre pour réaliser cela :

| Symbole | Signification |
|---------|---------------|
| and     | ET            |
| or      | OU            |
| not     | NON           |

TABLE 2 – Les opérateurs logiques

Voici quelques exemples du fonctionnement de ces opérateurs :

```
1 | # Si l'age est supérieur à 18 et inférieur à 40 au même temps.
2 | if age >= 18 and age <= 40:
3 |     # Les instructions

1 | # Si on est Samedi ou Vendredi alors on entre dans le bloc du if.
2 | if jour == "Samedi" or jour == "Vendredi" :
3 |     # Pas d'études

1 | # Le 'not' est utilisé pour exprimé la négation d'une condition.
2 | # Ça se traduit en : si l'age n'est pas supérieur à 18.
3 | if not age > 18 :
4 |     # Les instructions
```

Sachez qu'on peut combinez les opérateurs logiques entres eux, mais pour cela vaut mieux utiliser les parenthèses pour exprimer la priorité. Exemple :

```
1 | if (jour == "Samedi" or jour == "Vendredi") and chaleur < 35 :
2 |     # Sortir de la maison
```

### 1.6.6 Les booléens

Il y a un autre type de variables dont je ne vous ai pas parlé avant pour ne pas vous en charger. Ce type de variable est appelé *Booléen* (`bool` en python) et il est très utilisé dans les conditions et les boucles.

Les booléens sont des variables qui peuvent contenir une de deux valeurs seulement : `True` (qui veut dire vrai) ou `False` (qui veut dire faux), et on peut testé directement si ces variables sont à vrai dans les conditions :

```
1 | heureux = True
2 |
3 | # Si la variable heureux est vrai, alors on entre dans le if,
4 | # Sinon on entre dans le else.
5 | if heureux :
6 |     print("Je suis heureux aujourd'hui :D")
7 | else :
8 |     print("Que personne ne croise mon regard")
```



Les opérateurs logiques sont applicable sur les booléens. Exemple :

```
1 # Ça se traduit en : Si la personne est heureuse et
2 # le jour et beau alors Sorte de la maison.
3 if heureux and beau_jour :
4     # Sortir de la maison
```

#### 1.6.7 Exercice 4

Voici l'énoncé de l'exercice :

- Créer un projet nommé le 'exercice 4' et créer un fichier 'main.py' dedans.
- Demandez un nombre de l'utilisateur, et dites lui si ce nombre est paire ou impaire.

#### 1.6.8 Exercice 4.1

Voici l'énoncé de l'exercice :

- Créen un projet nommée le 'exercice 4.1' et créer un fichier 'main.py' dedans.
- Demandez de l'utilisateur son nom et son mot de passe, et dite d'entrer si le mot de passe est juste. S'il a tort, alors dite le est-ce que l'erreur est dans le nom d'utilisateur ou le mot de passe.

## 1.7 Les boucles

Si vous avez compris les conditions, les boucles seront un jeu d'enfant. En gros, les boucles servent à répéter certaines instructions plusieurs fois, et on a deux boucles dans le langage python :

- La boucle `while` qui signifie *tant que*.
- La boucle `for` qui signifie *pour*.

### 1.7.1 La boucle while

La boucle `while` permet de répéter un bloc d'instructions *tant que* une condition est vraie. Exemple :

```
1 touche = ""
2
3 while touche != "q" :
4     touche = input("Tapez q pour sortir : ")
5
6 print("Au revoir")
```

Dans cet exemple, on demande à l'utilisateur d'entrer la lettre `q`. Si il reste sage et il fait ce que nous le avons demandé, alors on lui laisse sortir de la boucle, sinon on lui redemande à nouveau jusqu'à ce qu'il entre la touche `q`.



On peut utilisé les booléens avec la boucle `while`. Exemple :

```
1 heureux = True
2
3 while heureux :
4     # Continuez de sourire.
```

On donne un autre exemple de l'utilisation de la boucle `while`. Imaginez que votre prof vous demande d'écrire un programme qui affiche le message "Ceci est un message très utile" 10 fois, vous pouvez faire ça :

```
1 print("Ceci est un message très utile")
2 print("Ceci est un message très utile")
3 print("Ceci est un message très utile")
4 print("Ceci est un message très utile")
5 print("Ceci est un message très utile")
6 print("Ceci est un message très utile")
7 print("Ceci est un message très utile")
8 print("Ceci est un message très utile")
9 print("Ceci est un message très utile")
10 print("Ceci est un message très utile")
```

C'est vrai que ça accomplira ce que votre prof a demandé, mais ce n'est ni propre ni pratique. Et imaginez encore si cette fois elle vous demande d'afficher ce message 100 fois, ou 1000 fois ...on voit bien que cette technique n'est plus efficace.

Ce qu'on peut faire pour réaliser ce que le prof a demandé c'est d'utiliser la boucle `while` et un *compteur* :

```
1 # On initialise notre variable qui nous servira de compteur.
2 i = 0
3
4 # Tant que la variable compteur est inférieur à 10 on
5 # affiche un autre message.
6 while i < 10:
7     print("Ceci est un message très utile")
8     # On incrémente notre compteur pour savoir quand s'arrêter.
9     i = i + 1
```

Voici le résultat d'exécution :

---

```
Ceci est un message très utile
Ceci est un message très utile
Ceci est un message très utile
Ceci est un message très utile
Ceci est un message très utile
Ceci est un message très utile
Ceci est un message très utile
Ceci est un message très utile
Ceci est un message très utile
Ceci est un message très utile
Ceci est un message très utile
```

---



La variable `i` est souvent utilisé en informatique comme un compteur ou pour parcourir les tableaux (on verra ça plus tard).



En informatique, dans la majorité des cas, on commence nos compteur de `0` et pas de `1`. Ça a une relation avec le parcours des tableaux qu'on verra dans les prochaines sections. Donc si vous voulez utilisé un compteur, il faut de préférence commencer de `0`.

### 1.7.2 La boucle for

La boucle `for` est utilisé uniquement avec les compteurs. C'est une forme compacté de la boucle `while` précédente. Si on refait l'exemple précédent en utilisant la boucle `for` ça nous donne ça :

```
1 # Ça se traduit en : pour i allant de 0 à 10 (exclus).
```



```
2 | for i in range(0, 10):  
3 |     print("Ceci est un message très utile")
```

Dans cet exemple, `i` prend chacune des valeurs `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`. Et à chaque fois on affiche le message `Ceci est un message très utile`.

Pour voir que `i` prend vraiment toutes ces valeurs là, on peut afficher la variable `i` à chaque itération de la boucle :

```
1 | for i in range(0, 10) :  
2 |     print(i)
```

Ce qui nous donne ça :

---

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

---

On a vu la boucle `for`. Maintenant détaillons un peu comment l'utiliser :

```
1 | for i in range(depart, arrivee) :  
2 |     print(i)
```

La variable `i` parcourra toutes les valeurs en démarrant de la valeur `depart` (inclus) jusqu'à la valeur `arrivee` (exclus). Amusez-vous à changer la valeur de `depart` et `arrivee` pour voir quelles sont les valeurs prises par `i`.

### 1.7.3 Un bonus de l'utilisation de la boucle for

Dans certaines situations, on utilise la boucle `for` de cette manière :

```
1 | for i in range(depart, arrivee, pas) :  
2 |     print(i)
```

C'est comme l'utilisation précédente, mais là on a ajouté le `pas`. Ce `pas` est utilisé pour dire à la variable `i` combien elle doit sauter à chaque itération de la boucle. Essayez cet exemple et vous verrez :

```
1 | for i in range(0, 10, 2) :  
2 |     print(i)
```

L'exécution nous donne ça :

---

0  
2  
4  
6  
8

---

`i` démarre de `0`, mais à chaque itération de la boucle, elle fait un saut de `2`.

#### 1.7.4 Exercice 5

Voici l'énoncé :

- Créez un projet nommé 'exercice 5' et créez un fichier dedans nommé 'main.py'.
- Demandez à l'utilisateur d'entrer un nombre, et affichez lui tous les diviseurs de ce nombre. Exemple : Si l'utilisateur entre `10`, vous lui affichez `1, 2, 5, 10`.

## 1.8 Découper le programme en fonctions

Jusque là, nous avons écrit toutes nos instructions directement , l'une après l'autre. Maintenant, nous allons apprendre à découper nos programmes en petits bouts, et cela grâce aux fonctions.

### 1.8.1 C'est quoi une fonction ?

Une fonction est un ensemble d'instructions qu'on groupe ensemble. Cet ensemble d'instructions exécute des actions et renvoie un résultat.

Il vaut mieux que vous compreniez les fonctions avec les exemples, alors on va donner plusieurs exemples et on va les expliquer un à un.

### 1.8.2 Fonction simple

Voici l'exemple d'une fonction qui calcule le triple d'un nombre :

```
1 def triple(nombre) :  
2     resultat = nombre * 3  
3     return resultat
```

Et voici comment on va utiliser cette fonction dans notre programme :

```
1 # Instructions ici  
2 x = triple(7)  
3  
4 # Ceci nous affichera 21  
5 print(x)
```

Des explications s'impose ici :

- `def triple(nombre) :` : `def` est un mot clé utilisé par python, ce mot nous permet de créer les fonctions. `triple` est le nom de notre fonction, c'est ce mot qui nous permet d'utiliser notre fonction dans notre programme principale. `nombre` est une variable qui est créée à l'intérieur de la fonction, cette variable prend la valeur qu'on donne à notre fonction au moment de l'utiliser (dans cet exemple elle prend la valeur `7` à cause de l'instruction `x = triple(7)`); une fois la fonction est finie, la variable `nombre` est effacé.
- `resultat = nombre * 3` : ceci est une instruction qui est exécutée dans notre fonction. On peut mettre autant d'instruction qu'on veut dans la fonction, il faut juste ne pas oublier de les décaler (comme on a vu dans les conditions).
- `return resultat` : cette ligne renvoie la valeur de la variable `resultat` au programme principale.

- `x = triple(7)` : cette instruction appelle la fonction `triple()` en lui donnant la valeur `7`. Ensuite, la valeur renvoyée par la fonction `triple()` sera stockée dans la variable `x`.

Donc voici notre programme en tout :

```
1 def triple(nombre) :
2     resultat = nombre * 3
3     return resultat
4
5
6 # C'est d'ici que commence l'exécution de notre programme.
7 x = triple(7)
8
9 print(x)
```



N'oubliez pas de décaler les instructions dans la fonction, sinon python ne fera pas la différence entre les instructions qui sont dans la fonction et les instructions qui sont dans le programme principale.

### 1.8.3 Fonction avec plusieurs paramètres

On peut utiliser plusieurs variables entre les parenthèses de la fonction, et pas seulement une comme on a fait dans l'exemple précédent. Par exemple, on a ici une fonction qui calcule la somme de deux nombres :

```
1 def addition(nombre1, nombre2) :
2     resultat = nombre1 + nombre2
3     return resultat
```



Les variables qui sont créées entre les parenthèses de la fonction sont appelées *paramètres*. On les appelle comme ça dans le sens où se sont des paramètres qu'on passe à la fonction pour qu'elle fasse ces calculs.

Et voici notre programme qui utilise cette fonction :

```
1 def addition(nombre1, nombre2) :
2     resultat = nombre1 + nombre2
3     return resultat
4
5 # C'est ici que commence l'exécution de notre programme.
6 a = 10
7 b = 25
8
9 x = addition(a, b)
10
```

```
11 | print(x)
```

### 1.8.4 Fonction qui ne renvoie rien

On peut aussi créer une fonction qui ne renvoie rien. Exemple :

```
1 | def presentation(nom, prenom, age) :  
2 |     print("Bonjour.")  
3 |     print("Je m'appelle", nom, prenom, "et j'ai", age, "ans")
```

Et voici le programme qui l'utilise :

```
1 | def presentation(nom, prenom, age) :  
2 |     print("Bonjour.")  
3 |     print("Je m'appelle", nom, prenom, "et j'ai", age, "ans")  
4 |  
5 | # C'est ici que commence l'exécution de notre programme.  
6 | presentation("Yanis", "Khelloufi", 22)
```

### 1.8.5 Fonction qui renvoie plusieurs valeurs

On peut aussi créer une fonction qui renvoie plusieurs valeurs. Voici un exemple d'une fonction qui fait la division euclidienne, et renvoie le résultat de la division et le reste de la division :

```
1 | def division_euclidienne(divisee, diviseur) :  
2 |     resultat_de_division = divisee // diviseur  
3 |     reste_de_division = divisee % diviseur  
4 |  
5 |     return resultat_de_division, reste_de_division
```

Pour récupérer ce que la fonction a renvoyé on fait ça :

```
1 | # 'a' recevra la valeur de la variable 'resultat_de_division',  
2 | # et 'b' recevra la valeur de la variable 'reste_de_division'.  
3 | a, b = division_euclidienne(20, 3)
```

Voici le code complet de notre programme :

```
1 | def division_euclidienne(divisee, diviseur) :  
2 |     resultat_de_division = divisee // diviseur  
3 |     reste_de_division = divisee % diviseur  
4 |  
5 |     return resultat_de_division, reste_de_division  
6 |  
7 | # C'est ici que commence l'exécution de notre programme.  
8 | a, b = division_euclidienne(20, 3)  
9 |  
10 | # Ceci nous affichera : 6 2  
11 | print(a, b)
```

### 1.8.6 Pourquoi découper notre programme??

C'est vrai que les exemples que nous avons donnés jusque là ne semblent pas vraiment utiles. Mais sachez que ce sont que des exemples pour que vous compreniez comment créer les fonctions. Quand on découpe vraiment, on découpe en des fonctions qui sont vraiment utiles, exemple : une fonction qui nous dit si un nombre est premier ou pas.



On revient à notre question, pourquoi découper??

Et bien, pour faciliter la réalisation d'un programme et pour rendre notre code source plus lisible et plus organisé. Car les programmes que nous avons réalisés jusque là sont des programmes simples, qui ne dépassent pas une dizaine de lignes de code. Mais les vrais programmes sont beaucoup plus complexes que ça, et ils peuvent atteindre les dizaines de milliers de lignes de codes très facilement. Sans découper, vous ne pouvez pas créer ou maintenir ce genre de programme.

### 1.8.7 Exercice 6

Voici l'énoncé de l'exercice :

- Créez un projet nommé 'exercice 6' et créez un fichier dedans nommé 'main.py'.
- Créez une fonction qui reçoit un nombre en paramètre, et qui nous dit s'il est premier ou pas.



Cette fonction retourne `True` ou `False`



Un nombre premier est un nombre qui est divisible seulement par `1` et par lui-même. En d'autres termes, aucun autre nombre ne divise ce nombre

- Dans votre programme principale, demander un nombre de l'utilisateur, et dire lui si ce nombre est premier ou pas.

### 1.8.8 Solution

Voici la solution de l'exercice :

```
1 | # On crée notre fonction
2 | def premier(nombre):
```

```

3     for i in range(2, nombre):
4         if nombre % i == 0:
5             return False
6
7     return True
8
9
10
11 n = input("Entrez un nombre svp : ")
12 n = int(n)
13
14 # On utilise notre fonction pour savoir si 'n' est
15 # premier ou pas.
16 prem = premier(n)
17
18 # La variable 'prem' contiendra 'True' si
19 # le nombre n est premier.
20 if prem:
21     print("Le nombre", n, "est premier")
22 else:
23     print("Le nombre", n, "n'est pas premier")

```

## 1.9 Les modules

Un module est un ensemble de fonctions créées déjà (par les créateurs de python ou par d'autres programmeurs). Ces fonctions sont regroupées ensemble, et ils ont le même thème. Par exemple, on peut trouver les fonctions mathématiques (cosinus, sinus, racine ...) dans le module `math`.

Pour utiliser les fonctions mathématiques, il faut commencer par importer le module `math`. Pour cela, il suffit tout simplement d'utiliser le mot clé `import` :

```
1 | import math
```

Maintenant que le module `math` est importé, on peut utiliser les fonctions qui se trouvent à l'intérieur comme ça :

```
1 | math.nom_de_fonction()
```

Par exemple, pour utiliser la fonction qui calcule la racine d'un nombre, on l'appelle comme ça :

```
1 | # La variable 'a' contiendra la valeur 3 après cette instruction
2 | a = math.sqrt(9)
```

Voici le code complet de notre programme :

```
1 | import math
2 |
3 | a = math.sqrt(9)
4 | print(a)
```



Par convention, tous les modules utilisés doivent être importés au début du programme.

Le module `math` ne contient pas uniquement les fonctions mathématiques, mais il contient aussi des variables qui ont une relation avec les maths (comme la variable `pi`, et `e`) :

```
1 | print("pi =", math.pi)
2 | print("e  =", math.e)
```

### 1.9.1 Voir toutes les fonctions d'un module

Pour voir les fonctions d'un module, il suffit juste de taper le nom du module dans pycharm, puis taper un point, pycharm ensuite vous affichera dans une liste tous les fonctions de ce module ([Figure 20](#)) .



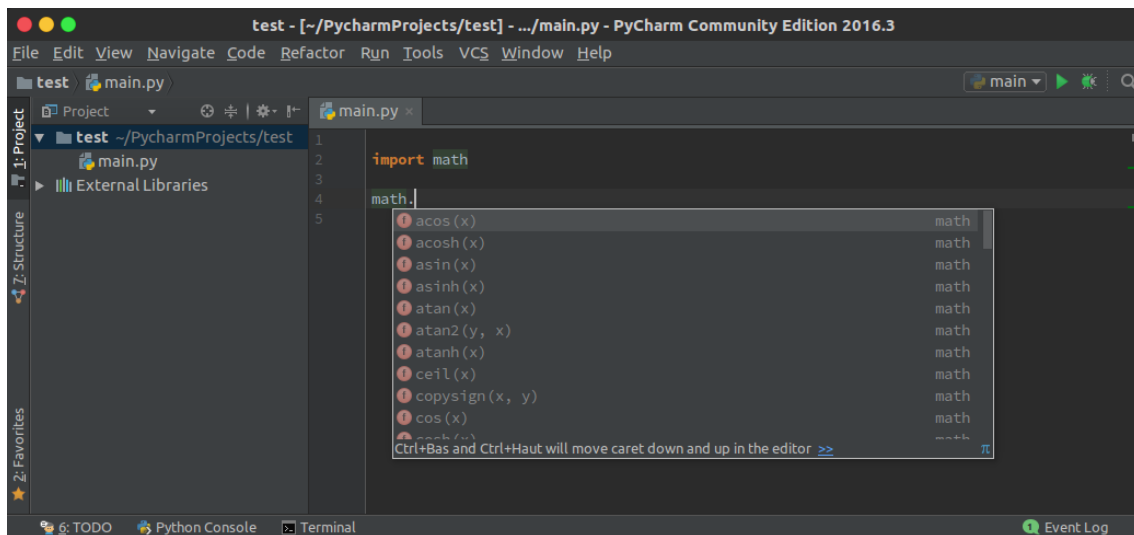


FIGURE 20 – La liste des fonctions du module math

Vous pouvez parcourir cette liste avec les touches `bas` et `haut` pour voir toutes les fonctions contenu par ce module.

### 1.9.2 Le module random

Parmi les modules de python, on trouve le module `random`. Ce module nous permet de générer des nombres aléatoires.

La fonction la plus intéressante de ce module est la fonction `randint()`. Cette fonction reçoit en paramètre deux entiers, et renvoie un nombre aléatoire entre ces deux entier. Voici un exemple d'utilisation :

```
1 import random
2
3 # la variable 'a' aura une valeur aléatoire compris entre
4 # 0 (inclut) et 100 (inclut)
5 a = random.randint(0, 100)
6
7 # la variable 'b' aura une valeur aléatoire compris entre
8 # -50 (inclut) et 300 (inclut)
9 b = random.randint(-50, 300)
10
11 print(a, b)
```

Vous pouvez vous amusez a re-exécuter ce code, et vous verrai qu'à chaque fois, les variables `a` et `b` prennent des valeurs différentes.

### 1.9.3 Quelles sont les modules disponibles??

Python contient par défaut des centaines de modules, et comme c'est un langage très populaire et *open source*, beaucoup de développeurs crée des modules très utiles et les partages avec la communauté.

Parmi les modules créés par les développeurs, on trouve :

- Des modules pour d'interagir avec les bases de données,
- des modules pour faire de la programmation graphique,
- des modules pour afficher des barres de progressions,
- des modules pour manipuler les documents xml,
- ... etc

On verra dans la fin de cette formation comment installer de nouveaux modules.

### 1.9.4 Exercice 7

Voici l'énoncé de l'exercice :

- Créez un projet nommé 'exercice 7' et créez un fichier dedans nommé 'main.py'.
- Chercher la fonction qui fait le factoriel dans le module `math`, et utiliser cette fonction pour calculer le factoriel de `1000`.

## 1.10 TP

Nous allons faire un TP qui regroupera tous les concepts que nous avons vu jusque là. Nous allons créer un mini jeu.

Notre jeu va générer un nombre aléatoire, puis il va demander à l'utilisateur de trouver ce nombre. À chaque fois que l'utilisateur essaye de trouver ce nombre aléatoire, notre programme doit l'informer si c'est juste ou pas; si ce n'est pas juste, notre programme doit dire à l'utilisateur si le nombre aléatoire est supérieur ou inférieur au nombre donné par l'utilisateur.

Voici un exemple du déroulement de notre programme :

---

```
J'ai générer un nombre aléatoire entre 0 et 100.  
Trouve le nombre : 45  
C'est plus  
Trouve le nombre : 90  
C'est moins  
Trouve le nombre : 70  
C'est moins  
Trouve le nombre : 67  
Bravo, vous avez trouvé le nombre que j'ai générer.
```

---

### 1.10.1 La solution

```

1 import random
2
3 nombre_aleatoire = random.randint(0, 100)
4 nombre_utilisateur = -1
5
6 print("J'ai générer un nombre aléatoire entre 0 et 100")
7
8 while nombre_utilisateur != nombre_aleatoire:
9     nombre_utilisateur = input("Trouve le nombre : ")
10    nombre_utilisateur = int(nombre_utilisateur)
11    if nombre_aleatoire > nombre_utilisateur:
12        print("C'est plus")
13    elif nombre_aleatoire < nombre_utilisateur:
14        print("C'est moins")
15    elif nombre_aleatoire == nombre_utilisateur:
16        print("Bravo, vous avez trouvez le nombre que j'ai générer.")

```

### 1.10.2 Extension du programme

Maintenant que votre jeu est fonctionnel, mettez les instructions de votre programmes dans une fonction appelé `jouer`, et débrouiller vous pour que le jeux se répète si l'utilisateur a en vie de répéter.

Voici un exemple du déroulement :

---

```

J'ai générer un nombre aléatoire entre 0 et 100.
Trouve le nombre : 45
C'est plus
Trouve le nombre : 90
C'est moins
Trouve le nombre : 70
C'est moins
Trouve le nombre : 67
Bravo, vous avez trouvez le nombre que j'ai générer.

```

Est-ce que vous voulez rejouer (o/n) : o

```

J'ai générer un nombre aléatoire entre 0 et 100.
Trouve le nombre : 55
C'est moins
.
.
.

```

---

### 1.10.3 La solution

```

1 import random
2
3

```

```

4 def jouer():
5
6     nombre_aleatoire = random.randint(0, 100)
7     nombre_utilisateur = -1
8
9     print("J'ai générer un nombre aléatoire entre 0 et 100")
10
11     while nombre_utilisateur != nombre_aleatoire:
12         nombre_utilisateur = input("Trouve le nombre : ")
13         nombre_utilisateur = int(nombre_utilisateur)
14         if nombre_aleatoire > nombre_utilisateur:
15             print("C'est plus")
16         elif nombre_aleatoire < nombre_utilisateur:
17             print("C'est moins")
18         elif nombre_aleatoire == nombre_utilisateur:
19             print("Bravo, vous avez trouvez le nombre que j'ai générer.")
20
21
22 # C'est à cette instruction que commence notre programme.
23 rejouer = ""
24 while rejouer != "n":
25     jouer()
26     rejouer = input("Est-ce que vous voulez rejouer (o/n)")

```

## 1.11 Le help

Nous allons voir dans cette partie, comment faire pour obtenir de l'aide sur le fonctionnement d'une fonction ou d'un module.

Disons par exemple que vous avez entendu parler que la fonction `print()` est utilisé pour l'affichage, mais le problème c'est que vous ne savez pas ce qu'elle fait exactement, ni comment l'utiliser. C'est là où le `help()` vient au secours.

Le `help()` (qui signifie aide) est une fonction de python qui vous permet d'obtenir l'aide sur une fonction ou un module. On lui donne en paramètre le nom de la fonction ou le module, et elle nous affiche qu'est-ce que fait ce dernier et comment l'utiliser.

Continuons dans l'exemple de la fonction `print()`, écrivez et exécutez ce code dans PyCharm :

```
1 | help(print)
```

Ce qui vous affichera :

---

Help on built-in function print in module builtins:

```

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

```

Prints the values to a stream, or to sys.stdout by default.  
Optional keyword arguments:

```
file: a file-like object (stream); defaults to the current sys.stdout.
sep:  string inserted between values, default a space.
end:  string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

---

On voit que la fonction `print()` peut être utilisé avec beaucoup d'option. Nous allons pas voir comment elle fonctionne exactement aujourd'hui, car on a pas assez d'information. Mais nous allons le voir plus tard si on aura le temps de terminer la formation.

Continuons avec un autre exemple. Disons qu'on veut comprendre le module `math`, il suffit aussi d'appeler le `help()` comme on a fait avec la fonction `print()` :

```
1 import math
2
3 help(math)
```

Voici une partie du résultat qui apparaît :

---

Help on built-in module math:

NAME

math

DESCRIPTION

This module is always available. It provides access to the mathematical functions defined by the C standard.

FUNCTIONS

acos(...)  
acos(x)

Return the arc cosine (measured in radians) of x.

acosh(...)  
acosh(x)

Return the inverse hyperbolic cosine of x.

---

L'affichage est généralement divisé en 3 parties. La première partie est `NAME` qui est le nom du module; la deuxième est `DESCRIPTION` qui est une bref description du contenu du module; et la troisième partie est `FUNCTIONS` qui est l'ensemble de fonction contenu dans ce module. Vous remarquez aussi que chaque fonction est expliqué.

Maintenant, supposons qu'on veut seulement connaître ce que fait la fonction `acos()`, et non pas tout le module `math`, comment fait on?? Il suffit juste d'appeler le `help()` sur la fonction `acos()` seulement :

```
1 import math
2
```

3 | `help(math.acos)`

Ce qui nous affichera seulement l'aide de la fonction `acos()` du module `math` :

---

Help on built-in function acos in module math:

```
acos(...)  
    acos(x)
```

```
    Return the arc cosine (measured in radians) of x.
```

---

Le `help()` est une fonctionnalité très importante, et elle vous sera très utile quand vous programmez.

## 2 Les objets



Mais c'est quoi un objet déjà??

Il n'y a pas une définition simple sur la notion d'objet, mais on peut le voir de cette manière : l'objet est comme une variable, mais il est beaucoup plus puissant et beaucoup plus pratique.

Nous allons essayer de voir les différents types d'objets qui existent en python, et nous allons voir comment ils fonctionnent, et comment les utiliser. Ainsi nous allons comprendre de mieux en mieux la notion d'objet.

### 2.1 Les listes

Le premier objet que nous allons apprendre dans ce chapitre sont les listes.

Une liste est un type objet qui peut contenir des objets ou des variables.

Disons qu'on veut créer une liste des noms des gens qui participe à la formation, nous allons procéder de cette manière :

```
1 | participants = ["Amine", "Khaled", "Imene"]
```

Maintenant, on a un objet `participants` de type liste, qui contient les étudiants qui participe à la formation.

Pour afficher notre objet de type liste, on utilise tout simplement `print()` :

```
1 | print(participants)
```

Et voici le résultat obtenu :

---

```
['Amine', 'Khaled', 'Imene']
```

---

On aurait pu mettre autant de personnes qu'on veut dans la liste,

#### 2.1.1 Autres exemples

Nous allons créer quelques listes pour que vous compreniez.

Une liste d'entiers :

```
1 | nombres = [10, 345, -46, 315]
2 |
3 | print(nombres)
```

Une liste nombres décimaux :

```

1 | decimaux = [1.134, 100.3]
2 |
3 | print(nombres)

```

Et voici le dernier exemple, une liste qui contient des variables et des objets de différents types :

```

1 | liste_heterogene = ["je suis une phrase", 13, 25, "salut", 22.5]
2 |
3 | print(liste_heterogene)

```

## 2.1.2 Accéder à un élément de la liste

Nous avons vu comment créer une liste et comment l'afficher en entier.



Maintenant si on veut afficher juste un élément de la liste, comment va-t-on faire ?

Pour afficher un élément d'une liste, on utilise les `[]`. Par exemple, si on veut afficher le premier élément de notre liste des noms, on fait comme ça :

```

1 | participants = ["Amine", "Khaled", "Imene"]
2 |
3 | print(participants[0])

```

Ceci nous affichera :

---

Amine

---



Dans python, comme dans tous les langages de programmation qui se respectent, la numérotation des éléments commence de `0` et non pas de `1`. Donc le premier élément de la liste est situé à la position `0`, le 2ème élément à la position `1` ...

Donc le dernier élément de notre liste se situe à la position `2`, et on peut l'afficher de cette manière :

```

1 | print(participants[2])

```

Sachez qu'on peut aussi récupérer un élément de la liste et le stocker dans une variable en utilisant le `[]`. Exemple :

```

1 | participants = ["Amine", "Khaled", "Imene"]
2 |
3 | un_element = participants[2]
4 |
5 | # Ceci nous affichera Imene

```



```
6 | print(un_element)
```

### 2.1.3 Parcourir une liste

Pour parcourir une liste, on utilise la boucle `for` :

```
1 | participants = ["Amine", "Khaled", "Imene"]
2 |
3 | for elm in participants:
4 |     print("Nom :", elm)
```

À chaque itération de la boucle, `elm` prend le prochain élément de notre liste `participants`. Ainsi, le résultat de l'exécution c'est :

---

```
Nom : Amine
Nom : Khaled
Nom : Imene
```

---

### 2.1.4 Exercice 8

Crée une liste qui contient ces éléments : `"Moustapha"`, `128`, `42.1`, `"Eau"`, `"Tomate"` et parcourez cette liste avec la boucle `for`.

### 2.1.5 Opérations sur les listes

Nous allons voir maintenant la puissance des objets dont je vous ai parlé tout à l'heure.

Chaque type d'objet a un ensemble d'opérations propre à lui, et qu'on peut utiliser directement. La syntaxe pour utiliser une opération est le suivant :

```
1 | notre_objet.nom_operation()
```

Revenons à notre exemple. Disons que nous avons créé notre liste :

```
1 | participants = ["Amine", "Khaled", "Imene"]
```

Puis au milieu de notre programme, on veut ajouter un autre membre à notre liste de participants, comment allons nous faire?? Et bien, il suffit tout simplement d'utiliser l'opération qui permet d'ajouter un nouveau élément à notre liste :

```
1 | participants.append("Hmidouche")
```

Ainsi, nous avons ajouté un autre membre à notre liste. Si vous affichez notre liste maintenant vous allez voir qu'elle contient un nouveau membre. Voici le code de notre programme :

```

1 participants = ["Amine", "Khaled", "Imene"]
2 # Ceci nous affichera ["Amine", "Khaled", "Imene"]
3 print(participants)
4
5 participants.append("Hmidouche")
6 # Ceci nous affichera ["Amine", "Khaled", "Imene", "Hmidouche"]
7 print(participants)

```

Donnant un autre exemple d'une autre opération. Disons que maintenant, Amine s'est désinscrit de la formation, et on doit le supprimer de la liste. Pour cela, il suffit d'utiliser l'opération qui permet de supprimer un élément de la liste :

```

1 participants.remove("Amine")
2 # Ceci nous affichera ["Khaled", "Imene", "Hmidouche"]
3 print(participants)

```

Pour chercher une opération dans un objet, on utilise la même technique qu'avec les modules. Il suffit juste d'écrire le nom de notre objet, suivi par un `.`, Py-Charm alors nous affichera toutes les opérations qu'on peut utiliser sur cet objet.

### 2.1.6 Exercice 9

Crée la liste `participants` comme on a fait tout à l'heure, et chercher l'opération qui permet de trier cette liste. Quand vous la trouvez, triez la liste et affichez là.

### 2.1.7 Accéder au dernier élément de la liste

Pour accéder au dernier élément de notre liste, on peut utiliser la méthode standard : Celle de récupérer la taille de notre liste, puis accéder au dernier élément en utilisant cette taille :

```

1 participants = ["Amine", "Khaled", "Imene"]
2 # On récupère la taille de notre liste dans la variable 'taille'
3 taille = len(participants)
4
5 # On affiche le dernier élément de notre liste
6 print(participants[taille - 1])

```

Ou on peut utiliser la méthode la plus appropriée de python, qui est :

```

1 participants = ["Amine", "Khaled", "Imene"]
2
3 # On affiche le dernier élément de notre liste
4 print(participants[-1])

```

Avec python, si on utilise des positions négatives, on accède aux éléments depuis la fin. Donc pour accéder au dernier élément on utilise `participants[-1]`, pour accéder à l'avant dernier élément, on utilise `participants[-2]` ...

## 2.2 Les string

Nous avons parlé sur les objets de types liste dans la parties précédentes, maintenant on va parler sur les objets de type `string`.

Pour rappel, les objets de type `string` contiennent du texte. Exemple :

```
1 phrase = "Salut tout le monde"
```



Nous avons dit au début de la formation qu'on peut déclarer une variable qui contient du texte en utilisant les `" "`. Mais sachez que ceux-ci ne sont pas des variables, mais plutôt des objets. J'ai dit que ce sont des variables pour vous aidez à comprendre seulement.

### 2.2.1 Accéder à une lettre

Vous pouvez voir les strings comme une succession de lettres. Pour accéder aux lettres de notre `string`, on utilise les `[]` comme on a fait avec les listes :

```
1 phrase = "Salut tout le monde"
2 # On récupère la première lettre
3 lettre1 = phrase[0]
4 # On récupère la 5ème lettre
5 lettre2 = phrase[5]
6 # On récupère la dernière lettre
7 lettre3 = phrase[-1]
8
9 print(lettre1, lettre2, lettre3)
```

### 2.2.2 Des opérations sur les strings

Comme avec les liste, les objets de type `string` ont plusieurs opérations qu'on peut utiliser.

Parmi ces opérations, on trouve l'opération qui vérifie si notre `string` contient que des lettres :

```
1 phrase = "Salut tout le monde"
2 if phrase.isalpha():
3     print("Notre string contient que des lettres.")
4 else :
5     print("Non, il ne contient pas que des lettres.")
```

Il y a une autre opération qui vérifie si notre objet contient que des chiffres :

```

1 | telephone = "0550102030"
2 | if telephone.isdigit():
3 |     print("Notre string contient que des chiffres.")
4 | else :
5 |     print("Non, il ne contient pas que des chiffres.")

```

On trouve plusieurs opérations de ce genre qui font la vérification. Je vous donne deux autres si vous voulez les essayer à la maison :

**isupper()** : Renvoie True si notre string contient que des majuscules.

**islower()** : Renvoie True si notre string contient que des minuscules.

### 2.2.3 Opérations avancées

Il existe une opération qui permet de chercher du texte dans notre string. C'est l'opération `find()` :

```

1 | phrase = "Salut tout le monde"
2 | position = phrase.find("tout")
3 |
4 | # Ceci nous affichera 6
5 | print("Le mot 'tout' est au position", position)

```

L'opération `find` (celle qu'on vient de voir), nous renvoie la position du texte dans notre objet string. Si elle ne trouve pas le texte, alors elle nous renvoie la valeur `-1`.

Allons voir une autre opérations très intéressantes. Disons qu'on veut diviser notre string en mot! Il suffit tout simplement d'utiliser l'opération `split()` comme ça :

```

1 | phrase = "Salut tout le monde"
2 | # On demande que notre phrase soit diviser
3 | # par rapport au espaces.
4 | les_mots = phrase.split(" ")
5 |
6 | # Ceci nous affichera : ["Salut", "tout", "le", "monde"]
7 | print(les_mots)

```

L'opération `split()` divise notre string selon le séparateur qu'on lui a donnée en paramètre, et elle nous renvoie une liste contenant les éléments qu'elle a obtenu.

Si on veut à nouveau joindre les éléments, il suffit d'utiliser l'opération `join()` :

```

1 | les_mots = ["Salut", "tout", "le", "monde"]
2 |
3 | # On demande que notre liste soit regroupé dans un string
4 | # et que entre chaque éléments de la liste en met un espace
5 | # ainsi on obtiendra a nouveau notre phrase.
6 | une_autre_phrase = " ".join(les_mots)

```

```

7 |
8 | # Ceci nous affichera : Salut tout le monde
9 | print(une_autre_phrase)

```

### 2.2.4 Exercice 10

Vous avez le string `"fromage,gâteau,tarte,pizza"`, diviser cette string en une liste selon le `,`; puis joigne cette liste en utilisant le `-` pour à la fin obtenir `"fromage-gâteau-tarte-pizza"`.

## 2.3 Les dictionnaires

Avant d'expliquer c'est quoi un dictionnaire, nous allons imaginer un programme que nous voulons créer.

Disons que nous voulons créer un programme où nous allons stocker les étudiants d'une classe, avec chacun sa moyenne, et on veut que ce programme soit capable d'ajouter de nouveaux étudiants, et de nous montrer la moyenne d'un étudiant si on veut. Comment on va faire? C'est dans ce genre de situation que les dictionnaires sont très utiles.

Voici comment créer un dictionnaire avec python.

```

1 | # Un dictionnaire vide
2 | moyenne = {}
3 |
4 | # Un dictionnaire initialisé
5 | moyenne = {"Khaled": 12, "Ahmed": 10, "Mustapha": 17}

```



C'est quoi un dictionnaire?

Les dictionnaires sont des objets qui contiennent d'autres objets (comme les listes), mais la différence avec les listes, c'est que l'accès aux éléments du dictionnaires se fait par clé, et non par indice.

Revenons au dictionnaire `moyenne` que nous venons de créer. Pour accéder à la moyenne de Khaled, il suffit de faire ça :

```

1 | # Création du dictionnaire
2 | moyenne = {"Khaled": 12, "Ahmed": 10, "Mustapha": 17}
3 |
4 | # Ceci nous affichera 12
5 | print(moyenne["Khaled"])

```

Dans l'exemple précédent, nous avons directement affiché la moyenne de Khaled, mais on aurait pu la stocker dans une variable, et faire ce qu'on veut avec.

```

1 moyenne_khaled = moyenne["Khaled"]
2
3 # Ceci nous affichera 12
4 print(moyenne_khaled)

```

Pour afficher un dictionnaire en entier, il suffit juste de passer le dictionnaire à `print` :

```

1 moyenne = {"Khaled": 12, "Ahmed": 10, "Mustapha": 17}
2
3 # On affiche notre dictionnaire
4 print(moyenne)

```

### 2.3.1 Modifier la valeur d'un élément du dictionnaire

Disons que nous avons fait une erreur en calculant la moyenne de Khaled, pour modifier sa moyenne après la création du dictionnaire, il suffit de faire ça :

```

1 moyenne = {"Khaled": 12, "Ahmed": 10, "Mustapha": 17}
2
3 # On modifie la moyenne de Khaled
4 moyenne["Khaled"] = 13
5
6 # Ceci nous affichera 13
7 print(moyenne["Khaled"])

```

### 2.3.2 Ajouter un élément au dictionnaire

Disons maintenant que nous avons calculer la moyenne d'un autre étudiant, et on veut l'ajouter à notre dictionnaire de moyenne, il suffit de faire ça pour ajouter ce nouveau étudiant avec sa moyenne :

```

1 moyenne = {"Khaled": 12, "Ahmed": 10, "Mustapha": 17}
2
3 # On ajoute un nouveau étudiant à notre dictionnaire.
4 moyenne["Imene"] = 15
5
6 # On affiche notre dictionnaire
7 print(moyenne)

```

### 2.3.3 Supprimer un élément du dictionnaire

Maintenant imaginons qu'on veut supprimer l'un des étudiants de notre dictionnaire, il suffit de faire ça :

```

1 moyenne = {"Khaled": 12, "Ahmed": 10, "Mustapha": 17, "Imene":
2     15}

```

```

3 | # On supprime un élément de notre dictionnaire.
4 | del moyenne["Mustapha"]
5 |
6 | # On affiche notre dictionnaire
7 | print(moyenne)

```

### 2.3.4 Parcourir un dictionnaire

Pour parcourir tous les éléments d'un dictionnaire, on utilise la boucle `for` :

```

1 | moyenne = {"Khaled": 12, "Ahmed": 10, "Mustapha": 17, "Imene":
    15}
2 |
3 | for key in moyenne:
4 |     print(key, "==>", moyenne[key])

```

À chaque itération de la boucle, `key` prend une clé du dictionnaire, et on utilise cette clé pour afficher la valeur correspondante.

### 2.3.5 Vérifier si une clé existe dans le dictionnaire

Disons qu'à un certain moment on veut vérifier si on a déjà ajouter l'étudiant Ahmed à notre dictionnaire, il suffit d'utiliser le mot clé `in` dans une condition :

```

1 | moyenne = {"Khaled": 12, "Ahmed": 10, "Mustapha": 17, "Imene":
    15}
2 |
3 | if "Ahmed" in moyenne:
4 |     print("Ahmed existe")
5 | else:
6 |     print("Ahmed n'existe pas")

```

### 2.3.6 Exercice 11

Réaliser les tâches suivantes dans un programme.

- Crée un dictionnaire de joueurs et de leurs score. Le dictionnaire contient :
  - Mohamed, score 500
  - Ahmed, score 20
  - Fawzi, score 0
- Ajouter un autre joueur nommé Hamid avec le score 100.
- Supprimer le joueur Fawzi du dictionnaire.
- Parcourir le dictionnaire et afficher les scores de tous les joueurs.

## 2.4 Les fichiers



Pourquoi utiliser les fichiers?

Imaginons que nous voulons créer un jeu vidéo, et dans ce jeu, on veut sauvegarder le score des meilleurs joueurs. Avec les variables c'est impossible, car dès que notre programme s'arrête, tous les variables disparaissent.

Mais avec les fichiers, c'est différent; car le contenu des fichiers reste même après l'arrêt du programme.

### 2.4.1 Ouvrir un fichier

Avant de pouvoir lire ou écrire dans un fichier, il faut d'abord l'ouvrir en utilisant la fonction `open`. Voici comment on utilise cette fonction :

```
1 fichier = open("chemin du fichier à ouvrir", "mode d'ouverture  
    du fichier")
```

La fonction `open` reçoit deux paramètres :

- Le premier paramètre c'est le chemin du fichier à ouvrir. On peut la donner soit un chemin absolu (Exemple : `C:/dossier1/dossier2/fichier.txt`) soit un chemin relative (relative par rapport d'où se trouve le programme).
- Le deuxième paramètre c'est le mode d'ouverture du fichier (lecture, écriture, ajout à la fin ...).

La fonction `open` renvoi un objet qui nous permet de manipuler le fichier ouvert.

### 2.4.2 Lire un fichier

Nous allons voir ici comment lire le contenu d'un fichier. Commencer par créer un fichier nommée `test.txt` dans le même dossier que votre programme. Mettez dans ce fichier :

---

```
Ceci est un fichier texte.  
On teste la lecture depuis un fichier avec python.
```

---

Nous allons maintenant ouvrir ce fichier et le lire avec python :

```
1 # Le "r" (read) dans le deuxième paramètre indique à la fonction  
    open qu'on  
2 # veut ouvrir le fichier en lecture.  
3 fichier = open("test.txt", "r")  
4
```



```

5 | # Ceci nous permet de lire tous le contenu du fichier et le
   |     stocker dans texte.
6 | texte = fichier.read()
7 |
8 | # On manipule le contenu du fichier comme on veut.
9 | print(texte)
10 |
11 | # Il faut fermer le fichier après avoir fini avec.
12 | fichier.close()

```

`read()` permet de lire tous le fichier comme un texte. Si par contre vous voulez récupérer le contenu dans une liste, et dont chaque élément de la liste représente une ligne, il faut utiliser `readlines()` :

```

1 | fichier = open("test.txt", "r")
2 |
3 | # Ceci nous permet de lire tous le contenu du fichier et le
   |     stocker dans texte.
4 | lignes = fichier.readlines()
5 |
6 | # On manipule le contenu du fichier comme on veut.
7 | print(lignes)
8 |
9 | # Il faut fermer le fichier après avoir fini avec.
10 | fichier.close()

```