Initiation à la programmation Python

Boite à outils pour résoudre les exercices de maths

2de

ellipses

Démarrer avec Python

1. Le choix de l'IDE

Il existe de nombreux IDE pour *Python*, nous faisons le choix de la simplicité et nous proposons de travailler avec logiciel *Thonny.* C'est un excellent éditeur léger, gratuit, multiplateforme (Linux, PC, Mac) en langue française. Développé par l'Université de Tartu en Estonie, il est téléchargeable à l'adresse suivante : https://thonny.org

L'interface minimale propose une zone pour les scripts en haut et la console en bas. Nous conseillons de rajouter la zone de variables et l'assistant à partir du menu « Affichage ».

2. La console

Dans un premier temps, nous allons nous intéresser à la console, reconnaissable par son invite : >>>

C'est la zone principale de dialogue entrée/sortie de *Python*. Nous pouvons l'utiliser un peu comme une calculatrice. Il suffit d'écrire un calcul ou une instruction et de valider avec la touche entrée. Le résultat s'affiche sur la ou les lignes suivantes. Parfois, on obtient des messages d'erreur !

Essayons d'effectuer une addition et une soustraction :

```
>>> 2+3
5
>>> 7-9
-2
```

Pour effectuer une multiplication, nous utiliserons l'astérisque : *

```
>>> 5*7
35
```

Pour effectuer une division décimale, nous utiliserons la barre oblique : /

```
>>> 24/5
4.8
```

Remarquons, que la partie entière et la partie décimale d'un nombre sont séparées par un point (à l'anglo-saxonne) et non par une virgule. C'est une source d'erreur, il faudra y faire attention. Le respect des symboles et de la syntaxe est crucial pour *Python* comme pour beaucoup de langages de programmation.

Pour effectuer une division euclidienne, nous utiliserons deux barres obliques : // pour le quotient et pourcent : % pour le reste.

```
>>> 26//4 ; 26%4
6
2
```

Nous observons que des calculs et plus généralement des instructions, peuvent être séparées sur une même ligne par un point-virgule.

En langage *Python*, il n'est pas rare d'avoir à sa disposition plusieurs outils pour obtenir un même résultat. Les opérations précédentes sont aussi obtenues par :

```
>>> divmod(26, 4)
(6, 2)
```

Attention, même si tous ces derniers calculs, peuvent être effectués avec des nombres négatifs, les résultats ne sont pas conformes à la définition de la division euclidienne car le reste doit être un entier positif.

Pour effectuer une puissance, nous utiliserons deux astérisques : **

```
>>> 2**3
8
```

Un résultat identique est obtenu avec :

```
>>> pow(2, 3)
8
```

L'usage des parenthèses est classique, mais le symbole de la multiplication est obligatoire, sous peine d'obtenir un message d'erreur écrit en rouge :

```
>>> 2(8+3)
<pyshell>:1: SyntaxWarning: 'int' object is not callable; perhaps you missed a
comma?
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'int' object is not callable
```

Nous devrons écrire :

```
>>> 2*(8+3)
22
```

Une remarque importante : *Python* est un outil très puissant avec les nombres entiers. Il est capable d'effectuer à une vitesse assez déconcertante, des calculs avec des nombres composés de centaines ou de milliers de chiffres. Il est par exemple possible d'obtenir très rapidement les 3304 chiffres qui composent le résultat de 2025**999. Par contre, avec les décimaux, il n'est pas nécessaire d'aller très loin pour trouver certaines limites :

```
>>> 0.1+0.1+0.1
0.30000000000000004
```

Étonnant, non ? Certes, l'erreur n'est que de l'ordre de 10^{-17}, mais cela fait un peu désordre. Nous verrons plus tard comment contourner cette difficulté.

3. Les variables

Pour s'attaquer à des opérations un plus ambitieuses, on se retrouve vite dans le besoin de garder des nombres en mémoire pour pouvoir les réutiliser ou les modifier plus tard. Pour cela, on utilise des variables. Ici, une variable est simplement une partie de la mémoire de l'ordinateur que l'on nomme afin d'y stocker une information (un nombre, un mot, …). Cependant, il y a des contraintes :

Il est d'usage d'utiliser seulement des lettres minuscules, même si l'utilisation de majuscules ne crée pas d'erreur. Le nom de la variable peut contenir des chiffres, mais pas en initiale. Un seul caractère spécial est utilisable « *l'underscore » :* _.

Par exemple :

```
>>> a = 12
>>> cheval_5 = 48.2
>>> le_nombre_de_chats = 55
```

sont des noms valides. Contre-exemple :

```
>>> 1a = 12
>>> cheval#5 = 48.2
>>> le nombre de chats = 55
```

ne sont pas des noms valides et génèrent une erreur.

Le choix d'un nom pour une variable est important. Il aide à la compréhension et facilite la relecture d'un script.

```
nb_chats = 8
```

Il est possible d'utiliser les lettres accentuées :

```
>>> nb_éléphants = 38
```

Python est sensible à la casse, concrètement, N et n sont deux variables différentes. Différencier deux variables juste par l'utilisation de minuscules ou de majuscules n'est pas recommandé car c'est source de confusion.

```
>>> univers = 12
>>> Univers = 15
```

Il existe une trentaine de mots réservés en *Python*. Il est impossible de les utiliser pour nommer une variable. Ceux sont des noms suivants :

« and, as, assert, break, class, continue, def, del, elif, else, except, False, Finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield ».

Concrètement, cela veut dire qu'il est impossible de nommer une variable « and » ou « break »...

```
>>> and = 12
  File "<pyshell>", line 1
    and = 12
        ^
SyntaxError: invalid syntax
```

Mais de manière assez surprenante on peut tout à fait nommer une variable « print » ou « input » qui sont des instructions du langage. Nous allons les découvrir très prochainement. Dans ce cas, on perd malencontreusement l'usage premier de cette instruction, ce qui n'est pas forcément très intéressant.

```
>>> print = 12
>>> print(12)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'int' object is not callable
```

Dans le même ordre d'idée, on définira plus tard des fonctions et il faudra faire attention, car une variable ne peut pas avoir le même nom qu'une fonction.

Il est possible de définir plusieurs variables sur une même ligne avec un seul signe égal. À gauche du signe égal, on énumère le nom des variables séparées par une virgule, à droite, on énumère les valeurs séparées aussi par des virgules :

```
>>> a, b, c, d = 12, 55, 6.9, 94.1
```

Cette possibilité est très pratique notamment pour échanger le contenu de deux variables sans avoir recours à une troisième variable « tampon » :

```
>>> a, b = 8, 12
>>> a, b = b, a
>>> a
12
>>> b
8
```

Maintenant, définissons deux variables :

```
>>> a, b = 11, 12.5
```

Ces deux nombres sont de différentes natures, a est un nombre entier, b est un nombre décimal. En *Python*, nous dirons que les variables sont de types différents. Il y a la variable a de type « int » et la variable b de type « float ».

```
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
```

Nous pouvons effectuer des opérations et stocker le résultat dans une nouvelle variable :

```
>>> c = a+b
>>> type(c)
<class 'float'>
>>> d = a*b
>>> type(d)
<class 'float'>
```

Une opération entre deux « int » donne un « int » et une opération entre deux « float » donne un « float ». Une opération entre un « int » et un « float » donne un « float ».

```
>>> a, b = 10, 12.5
>>> a*b
125.0
>>> type(125.0)
<class 'float'>
```

Une erreur assez classique :

```
>>> longueur, largeur = 5, 3
>>> périmètre = 2(longueur+largeur)
<pyshell>:1: SyntaxWarning: 'int' object is not callable; perhaps you missed a
comma?
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'int' object is not callable
```

Qu'est-ce qui ne va pas ? D'où vient cette erreur…

Tout simplement il manque le * la multiplication devant la parenthèse ouvrante. Dans le même ordre d'idée, pour l'expression $3x^2-8x+3$, il faudra écrire :

```
>>> 3*x**2-8*x+3
```

Pour afficher du texte, soyons civilisés, essayons :

```
>>> Bonjour
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'Bonjour' is not defined
```

Nous obtenons un message d'erreur. Pour l'interpréteur *Python*, le mot « bonjour » est vu comme le nom d'une variable (ou d'une fonction) inconnue, il ne comprend pas notre marque de civilité et nous le fait savoir.

Tapons maintenant :

```
>>> print("bonjour")
bonjour
```

Un bonjour s'affiche en retour de notre commande, Il serait naïf de croire qu'il nous a répondu, il ne fait qu'écrire sans aucune analyse ce que nous lui avons demandé d'écrire. Le mot « print » désigne une fonction de *Python*, qui sert à afficher un texte ou le contenu d'une variable. Les fonctions de *Python 3* comportent quasi-systématiquement des parenthèses : « print(...) », « input(...) », « len(...) » etc. Le texte que nous voulons afficher est entouré au choix par des guillemets ou des apostrophes.

```
>>> "Salut les amis"
'Salut les amis'
```

Attention de ne pas confondre du texte et des noms d'une variable. Pour comprendre la différence, entrons ceci :

```
>>> bonjour = 2
>>> print(bonjour)
2
>>> print("bonjour")
bonjour
>>> salut = "bonjour"
>>> print(salut)
bonjour
```

Petite analyse des lignes précédentes :

- À la première ligne, nous définissons la variable nommée bonjour et nous lui affectons la valeur 2.
- À la deuxième ligne, nous donnons l'ordre à *Python* d'afficher le mot (la chaine de caractère) bonjour et il exécute cet ordre à la ligne suivante.
- À l'avant-dernière ligne, nous définissons la variable nommée salut et nous lui affectons le mot bonjour.
- À la dernière ligne, nous demandons l'affichage du contenu de la variable nommée salut.

Nous pouvons donc aussi garder en mémoire des mots, (des chaînes de caractères), il s'agit alors du type « string » (ou « str »). Il existe de nombreux autres types de variable, nous en verrons d'autres par la suite.

```
>>> animal_1 = "Chat"
>>> type(animal_1)
<class 'str'>
>>> animal_2 = "Chien"
>>> animal_1
'Chat'
```

Poursuivons dans le même ordre d'idées :

```
>>> a = "salut"
>>> print("a")
a
>>> print(a)
salut
>>> b = " les amis"
>>> c = a+b
>>> print(c)
salut les amis
```

Nous observons que l'addition de deux chaines de caractère a pour effet de mettre bout à bout deux chaînes de caractères (cela s'appelle la concaténation).

Qu'en est-il de la multiplication, entrons ceci :

```
>>> a = 3
>>> print(2*a)
6
>>> print(2*"a")
aa
```

Petite analyse des lignes précédentes :

- Nous affectons la valeur 3 à la variable nommée a.
- Nous demandons l'affichage de deux fois a valeur de a et nous obtenons 6.
- Nous demandons l'affichage de deux fois la chaine de caractère composée d'une unique lettre a et nous obtenons aa.

À l'affichage, nous pouvons mélanger du texte et des nombres :

```
>>> print("1+1 est égal à", 1+1)
1+1 est égal à 2
```

La virgule sert à séparer les éléments à afficher. Il existe dans les versions récentes de *Python* une syntaxe très intéressante, en utilisant la lettre f et des accolades :

```
>>> a = 7
>>> print(f"Le carré de {a} est {a**2}.")
Le carré de 7 est 49.
```

On peut même formater l'écriture, pour par exemple aligner les nombres :

```
>>> print(f"{3.21:6.1f}") ; print(f"{12.58:6.1f}")
   3.2
  12.6
```

Parlons rapidement d'un autre type de variable qui nous sera utile notamment pour faire des tests : le « booléen ». C'est un type de données qui ne peut prendre que deux valeurs : « True » ou « False », c'est-à-dire vrai ou faux (1 ou 0).

```
>>> a = True
>>> a
True
>>> not a
False
>>> type(a)
<class 'bool'>
```

Nous y reviendrons plus tard.

La console est très pratique tester des instructions, pour effectuer des petites opérations, des vérifications, mais si nous quittons l'IDE tout est perdu. Pour prendre un peu plus d'envergure, il est temps d'écrire des scripts.

4. L'éditeur

L'éditeur est la zone en haut à gauche sur *Thonny*. Dans cet éditeur, nous allons pouvoir écrire des programmes plus ou moins longs, les enregistrer, puis ensuite les exécuter. Le résultat s'affichera dans la console, parfois dans une fenêtre supplémentaire. Il y a des différences de fonctionnement entre le mode éditeur et le mode interpréteur.

À la manière d'un traitement de textes, nous allons créer un fichier vierge, le nommer et l'enregistrer.

Regardons la barre d'outils de *Thonny* :

La première icône est un raccourci pour Fichier / Nouveau (c'est le dessin d'une feuille vierge). Elle permet de créer une nouvelle page de script vide.

La deuxième icône est un raccourci pour Fichier / Ouvrir (c'est le dessin d'un classeur ouvert). Elle permet d'ouvrir un fichier déjà créé en navigant si besoin dans l'arborescence de son disque dur.

La troisième icône est un raccourci pour Fichier / Enregistrer (c'est le dessin symbolique d'une disquette moyen de sauvegarde d'un autre temps), elle permet d'enregistrer son script.

La quatrième icône est un raccourci pour Exécuter / Exécuter le script courant (le « run » C'est un triangle blanc sur fond vert, qui semble dire en avant la musique !). Elle permet de lancer le script.

Les cinq icônes suivantes (un petit scarabée et des flèches) serviront plus tard dans des opérations de recherche de bugs.

La dernière icône est un raccourci pour Exécuter / Arrêter / redémarrer l'interpréteur (c'est un panneau Stop). Elle permet, tel un arrêt d'urgence, de sortir des boucles infernales et de reprendre la main pour modifier le script.

Pour commencer, il nous faut donc cliquer sur la première icône afin de créer une zone de script vierge. Par défaut, l'onglet créé s'appelle <sans nom>. Tapons, 2+3 sur cette page blanche et enregistrons le script (troisième icône). Lançons l'exécution du script en utilisant le bouton « run », on obtient dans la console :

```
>>> %Run essai.py
```

Mais ensuite, rien, aucun affichage. Contrairement à la console, dans l'éditeur, l'instruction « print() » est indispensable pour espérer une sortie dans la console. Corrigeons donc :

```
print(2+3)
```

Remarquons que nous sommes bien dans l'éditeur, il n'y a pas les trois chevrons de la console. Relançons le script avec « run ». Nous obtenons en retour dans la console :

```
>>> %Run essai.py
5
```

Cette fois 5 apparaît dans la console. Voilà notre premier script est écrit et il fonctionne !

Remarquons que lancer le script d'un clic sur le bouton « run » est équivalent à valider « %Run essai.py » dans la console.

En programmation, il est important dès le début de prendre de bonnes habitudes et l'une d'entre-elles consiste à commenter son script. Pour cela nous

allons utiliser le symbole #. Lorsque ce symbole apparaît dans un script, la fin de la ligne est transparente, elle est ignorée par *Python*. Par exemple :

```
a, b = 12, 7 # a et b sont deux variables
c = a + b # c est la somme de a et b
print(F"la somme de {a} et {b} est {c}") # Affichage du résultat
# Attention, dans la suite de ce programme a sera peut-être négatif.
```

Les commentaires permettent de s'approprier le script écrit par quelqu'un d'autre plus rapidement et aussi souvent de reprendre plus facilement ses propres scripts.

Pour écrire un commentaire sur plusieurs lignes, il est possible de placer un # au début de chacune des lignes ou alors nous pouvons utiliser trois guillemets avant et trois guillemets après. (Les guillemets peuvent être remplacés par des apostrophes). Par exemple :

```
"""
Ceci est un commentaire sur plusieurs lignes.
Ce script permet de répondre à l'exercice 8 de la page 32.
"""
```

Enfin, remarquons que la coloration syntaxique est une fonctionnalité très utile de *Thonny* (et de beaucoup d'autres IDE) qui permet d'afficher le code avec des couleurs aidant à sa compréhension. Selon que l'on écrit un commentaire, une instruction, une variable ou autre, une couleur différente sera utilisée.

Il va être intéressant d'instaurer un dialogue entre l'opérateur (l'utilisateur du script) et l'ordinateur. Parfois l'opérateur, qui utilise le programme, n'est pas le programmeur qui a écrit le script, Il sera alors important que les questions soient précises et les réponses claires, les commentaires pourront aider dans le dialogue.

Une première approche du dialogue opérateur/machine peut-être envisagée avec une logique de questions/réponses. Nous verrons plus tard une deuxième approche consistant à passer des paramètres à une fonction. Comme souvent *Python* permet des approches différentes pour obtenir des résultats similaires.

Pour demander à l'opérateur de saisir une valeur, nous allons utiliser l'instruction « input() », dans les parenthèses, on écrit le message qui s'affiche. Par exemple :

```
a = input("Saisir a : ")
print(f"a vaut {a}.")
```

Si on lance avec « run », on obtient dans la console :

```
>>> %Run essai.py
Saisir a : 12
a vaut 12.
```

Un dialogue c'est instauré entre l'utilisateur et l'ordinateur : Le programme a écrit : « Saisir a : », l'utilisateur (nous !) avons écrit 12 et validé. Le programme

a écrit : « a vaut 12. ». Ce n'est pas forcément très utile, mais ce n'est qu'un début.

En fait, ce petit script cache un piège puisque la variable saisie n'est pas considérée comme un nombre par *Python* mais comme une chaine de caractères. La mauvaise surprise apparaît avec :

```
a = input("Saisir a : ")
print(f"Le double de a vaut {2*a}.")
```

En effet, la console nous renvoie :

```
>>> %Run essai.py
Saisir a : 12
Le double de a vaut 1212
```

C'est fâcheux, ce script ne donne pas le résultat espéré ! Pour résoudre le problème, le programme doit convertir la chaine de caractères saisie en nombre entier ou flottant. Ce choix est fait par le programmeur au moment de l'écriture du programme. Pour cela, nous pouvons utiliser les fonctions « int() » ou « float() » :

Si a est un entier :

```
a = int(input("Saisir a : "))
```

Si a est un nombre décimal :

```
a = float(input("Saisir a : "))
```

Une remarque utile, si l'on ne sait pas de quel type (« int », « float ») sera le nombre, nous pourrons utiliser la fonction « eval() », ainsi *Python* s'adaptera à notre réponse :

```
a = eval(input("Saisir a : "))
print(type(a))
```

Cela nous donne :

```
>>> %Run essai.py
Saisir a : 12.6
<class 'float'>
>>> %Run essai.py
Saisir a : 1/3
<class 'float'>
>>> %Run essai.py
Saisir a : 16
<class 'int'>
```

5. Quelques défis

Défi 1 : Écrire un programme qui pose la question à l'utilisateur : « En quelle année êtes-vous né ? » et qui renvoie « Votre âge est : ... ».

Défi 2 : Écrire un programme qui demande 2 notes à l'utilisateur et qui renvoie : « La moyenne est : ... ».

Défi 3 : Écrire un programme qui demande à l'utilisateur la longueur du côté d'un carré et qui renvoie : « Le périmètre est : ... » et « L'aire est : ... ».

Défi 4 : Écrire un programme qui demande à l'utilisateur le taux de TVA et le prix HT (hors taxe) et qui renvoie : « Le prix TTC est : ».

Les corrections se trouvent pages 39 à 48.

6. Les tests

Nous allons ici parler de test qui permettent à un programme de s'adapter à plusieurs situations. Le mot-clé pour les tests est « if » (en français : si).

Dans la console de *Thonny*, nous allons observer les réponses de *Python* aux propositions suivantes :

```
>>> 6 > 2
True
>>> 5 < 5
False
>>> 8 >= 8
True
>>> a = -3
>>> a > 0
False
```

Python vérifie chacune des inégalités. Quand elle est vraie, il répond « True », quand elle est fausse, il répond « False », c'est un « booléen ». Observons maintenant d'autres propositions :

```
>>> 4 = 2
  File "<pyshell>", line 1
    4 = 2
    ^
SyntaxError: cannot assign to literal

>>> 4 == 2
False
>>> 4 == 4
True
>>> d = 5
>>> d == 8
False
>>> d == 5
True
```

Petite analyse des lignes précédentes :

- l'instruction 4 = 2 veut dire place le nombre 2 dans une variable nommée 4, c'est impossible d'où le message d'erreur.
- L'instruction 4 == 2 veut dire regarde si le nombre 4 est égal au nombre 2, *Python* répond logiquement que non.

- On affecte ensuite la valeur 5 à la variable d puis on demande à *Python* de regarder si d est égale à 8 (ce qui est faux) puis si b est égale à 5, ce qui est vrai.

En conclusion : le == sert à vérifier si deux choses sont égales. Attention : ne pas confondre pas le = (pour les affectations) et le == (pour les tests).

La non-égalité s'exprime avec les symboles !=. Par exemple :

```
>>> a = -3
>>> a == 2
False
>>> a != 2
True
```

Petite analyse des lignes précédentes :

- on donne la valeur -3 à la variable a ;
- on demande à *Python* si la valeur de a est égale à 2, c'est faux ;
- on demande à *Python* si la valeur de a est différente de 2, c'est vrai ;
- le symbole != veut dire « différent » ou « non égal à ». Ce qu'en mathématiques, on écrirait symbole ≠.

Dans un programme écrit en *Python*, nous pouvons rencontrer plusieurs structures de test, qui permettent de faire évoluer le programme.

6.1 Le test « Si … alors ... »

Dans le mode éditeur, entrons ce programme :

```
temp = float(input("Quelle température fait-il ? "))
if temp <= 0:
    print("A cette température, l'eau gèle.")
```

Testons-le avec plusieurs valeurs. Modifions-le exactement ainsi puis testons à nouveau :

```
temp = float(input("Quelle température fait-il ? "))
if temp <= 0:
    print("A cette température, l'eau gèle.")
print("Faites attention au verglas sur la route.")
```

Nous remarquons vite que le programme ne fonctionne pas bien : il affiche toujours « Faites attention au verglas sur la route. » quelle que soit la température, ce n'est pas acceptable.

Cela est dû au fait que l'instruction « print("Faites attention au verglas sur la route.") » n'a pas été décalée vers la droite comme l'instruction « print("A cette température, l'eau gèle.") ». Elle est alors extérieure au test et s'exécute toujours. Si on veut qu'elle ne s'exécute que si « temp<0 », nous devons faire ainsi :

```
temp = float(input("Quelle température fait-il ? "))
if temp <= 0:
    print("A cette température, l'eau gèle.")
    print("Faites attention au verglas sur la route.")
```

Ce décalage vers la droite avec une tabulation (ou quatre espaces) s'appelle une **indentation**. C'est un élément essentiel de la programmation dans le langage *Python*. Il est primordial de respecter l'indentation !

Remarquons, qu'avec l'IDE *Thonny*, quand nous validons une ligne finissant par « : » l'indentation de la ligne suivante est automatique.

6.2 Le test « Si … alors … sinon ... »

On a parfois besoin de traiter le cas où la condition n'est pas vérifiée. On utilise alors « if … else … » (else veut dire sinon). Par exemple :

```
temp = float(input("Quelle température fait-il ? "))
if temp <= 0:
    print("A cette température, l'eau gèle.")
    print("Faites attention au verglas sur la route.")
else:
    print("Aucun risque de verglas sur la route.")
```

6.3 Le test « Si … alors, autre si ... alors … sinon ... »

Pour un test à plusieurs conditions on utilise « elif » qui est la contraction de else if). Par exemple :

```
moy = float(input("Quelle est votre moyenne en maths ? "))
print("Cette moyenne est ", end="")
if moy <= 8:
    print("faible, il ne faut pas se décourager.")
elif moy < 10:
    print("trop juste, il faut faire un effort.")
elif moy < 13:
    print("convenable, il faut continuer ainsi.")
else :
    print("bonne, bravo !")
```

Une petite remarque : le « end="" » à la fin du « print() » évite le retour à la ligne et permet d'écrire la remarque issue du test à la suite sur la même ligne.

6.4 L'écriture compacte

Lorsque une seule instruction résulte du choix binaire opéré par un test, *Python* permet une écriture assez compacte sur une seule ligne, sans indentation :

```
age = int(input("Quel est votre age ? "))
print("Vous êtes", "mineur." if age < 18 else "majeur.")
```

Sans cette écriture compacte, nous aurions dû écrire :

```
age = int(input("Quel est votre age ? "))
print("Vous êtes", end =" ")
if age < 18 :
    print("mineur.")
else:
    print("majeur.")
```

6.5 Les tests imbriqués

Il est parfois utile d'imbriquer plusieurs tests. Voici un exemple :

```
note_1 = int(input("Quelle est la note 1 : "))
note_2 = int(input("Quelle est la note 2 : "))
if note_1 < 10:
    if note_2 < 10 :
        print("Les deux notes sont inférieures à 10.")
    else:
        print("Une seule note est supérieure à 10.")
else:
    if note_2 < 10 :
        print("Une seule note est supérieure à 10.")
    else:
        print("Les deux notes sont supérieures ou égales à 10.")
```

Attention à l'indentation dans ces cas là !

6.6 Les booléens

Il peut parfois être utile de créer une variable de type « booléen » pour stocker le résultat d'un test et l'utiliser dans un calcul.

```
>>> a = 15
>>> test = a > 10
>>> test
True
```

En fait dans un calcul, « True » vaut 1 et « False » vaut 0. Ainsi :

```
>>> a = 8
>>> 2*(a < 12)+3*(a >= 7)
5
```

Prenons un autre exemple : Imaginons que nous désirons écrire un programme qui nous demande de choisir un nombre entier et qui nous attribue deux points si notre choix est un multiple de 2, trois points si c'est un multiple de 3, (donc cinq points si c'est un multiple de 6) et zéro point sinon. Voici un script possible :

```
nombre = int(input("Quel nombre entier choisissez-vous ? "))
score = (nombre%2==0)*2+(nombre%3==0)*3
print(f"Vous gagnez {score} points")
```

7. Quelques défis

Défi 5 : Écrire un programme qui pose la question à l'utilisateur : « Êtes-vous un humain ou un robot ? » et qui renvoie « Bonjour » ou « 426F6E6A6F7572 » selon le cas.

Défi 6 : Écrire un programme qui demande un nombre entier à l'utilisateur. L'ordinateur affiche ensuite le message « Ce nombre est pair. » ou « Ce nombre est impair. » selon le cas.

Défi 7 : Reprendre le programme du **défi 2** et rajouter une appréciation :

- Si la moyenne est supérieure à 15 : « C'est très bien ! »
- Si la moyenne est entre 10 et 15 : « C'est bien ! ».
- Sinon : « Il ne faut pas se décourager. ».

Défi 8 : Écrire un programme qui demande les longueurs des trois côtés d'un triangle et qui utilise la réciproque du théorème de Pythagore pour dire si le triangle est rectangle ou pas.

Défi 9 : Écrire un programme qui demande trois longueurs AB, AC, BC et qui utilise l'inégalité triangulaire pour dire s'il est possible de tracer le triangle ABC.

Défi 10 : Écrire un programme qui demande un nombre entier à l'utilisateur et qui dit si ce nombre est divisible par 2, par 3 et donc par 6.

Les corrections se trouvent pages 39 à 48.

8. Les boucles

Les boucles sont très utiles quand on veut répéter plusieurs fois des instructions. Nous allons voir deux types de boucles que l'on utilise dans des contextes précis, même s'il est possible de déborder de ce cadre.

8.1 Les boucles « Pour »

En général, quand on sait combien de fois doit avoir lieu la répétition, on utilise une boucle for.

Souvent pour fabriquer un compteur, on utilise : "Pour … allant de … à …

```
for i in range(10):
    print(i)
```

Attention, observons bien quelles sont les bornes du compteur !

La fonction « range(a, b) » renvoie la séquence de nombres entiers a, a+1, a+2, …, b-2, et b-1.

Avec la fonction « range(a, b, k) », k est utilisé comme valeur de pas : le premier élément de la séquence est a. Chaque élément successif dans la séquence va incrémenter la valeur de l'élément qui le précède par k. b est la limite. Le dernier élément de la séquence doit être inférieur strictement à b.

Par exemples :

```
for v in range(3, 9, 2):
    print(v)
for v in range(13, 2, -3):
    print(v)
```

8.2 Les boucles « Tant Que »

La syntaxe de la boucle « Tant Que » est :

```
while condition:
    instruction_dans_boucle1
    instruction_dans_boucle2
    …
instruction_hors_boucle1
instruction_hors_boucle2
…
```

Tant que la condition est vérifiée, on effectue les instructions dans la boucle 1, 2 ..., (celles qui sont indentés, c'est-à-dire qui ont été décalées de 4 espaces). Quand la condition devient fausse, le programme sort de la boucle et continue avec les instructions hors boucle 1, 2 ...

Par exemple, le script suivant affiche le message de 14 caractères « Bonjour à tous » tant que le nombre total de caractères affichés est inférieur à 100.

```
i = 14
while i <= 100:
    i = i + 14
    print("Bonjour à tous")
```

On peut remarquer qu'une boucle compteur (« For ») aurait pu être utilisée ici, si on avait calculé préalablement le nombre de boucles nécessaires.

```
maxi = 100//14
for i in range(maxi):
    print("Bonjour à tous")
```

Un problème apparaît parfois : la boucle ne s'arrête jamais (on parle de boucle infinie).

```
i = 1
while i > 0:
    print("Bonjour")
    i = i+1
```

On doit alors interrompre l'exécution dans la commande Shell. Il faut donc faire attention que la condition finisse par être réalisée.

Par prudence, on peut prévoir un « break », mais c'est un peu lourd !

```
i = 1
while i > 0:
    print("Bonjour")
    i = i+1
    if i > 100:
        break
```

Envisageons le script suivant : nous voulons afficher tous les entiers de 1 à 100.

```
i = 1
while i <= 100:
    print(i)
    i += 1
```

Il faut remarquer que : « i += 1 » est identique à « i = i+1 ».

L'affichage de ce script demande une centaine de ligne dans la console, voici une petite amélioration de l'affichage, qui sera parfois utile :

```
i = 1
maxi = 100
while i <= maxi:
    print(i, end = ', ' if i != maxi else '.')
    i += 1
```

De manière générale, la boucle « Pour » est plus compacte, mais la boucle « Tant Que » est plus universelle car on n'a pas besoin de savoir au départ combien de répétitions la machine va faire.

9. Quelques défis

Défi 11 : Écrire un programme qui affiche les carrés de tous les entiers de 1 à 100.

Défi 12 : Écrire un programme qui affiche une table de multiplication au choix.

Défi 13 : Écrire un programme qui demande à l'aide d'une boucle de saisir 9 notes puis calcule la moyenne des 9 notes et ensuite affiche un message de félicitations si la moyenne est strictement supérieure à 15.

Défi 14 : Reprendre le **Défi 5** de sorte qu'il repose la question tant que la réponse est différente de h ou r.

Défi 15 : Écrire un programme qui demande un nombre entier n et il renvoie la somme 1+2+3+...+n.

Défi 16 : Écrire un programme qui donne la plus grande puissance de 2 inférieure à un nombre donné.

Défi 17 : Reprendre le programme **Défi 12** pour un nombre quelconque de notes.

Les corrections se trouvent pages 39 à 48.

10. Les fonctions

10.1 Les fonctions « lambda »

Pour définir une fonction mathématique afin de par exemple, calculer des images, il est pratique d'utiliser l'instruction « lambda ».

Par exemple : Pour calculer l'image de 4, puis l'image de -38,5 par la fonction x^2>4.

```
>>> f = lambda x: x**2-3*x+2
>>> f(4)
6
>>> f(-38.5)
1599.75
```

Remarquons que l'on peut utiliser les fonctions lambda indifféremment en mode console ou en mode éditeur.

10.2 Les fonctions « def … return »

Lorsqu'une tâche doit être réalisée plusieurs fois par un programme avec seulement des paramètres différents, on peut l'isoler au sein d'une fonction.

La syntaxe *Python* pour la définition d'une fonction est la suivante :

```
def nom_fonction(liste de paramètres):
    bloc d'instructions
    ...
    return résultats
```

Les paramètres sont les éléments que l'on donne à la fonction, après un traitement, cette fonction nous retourne des résultats.

Remarquons que l'on peut choisir n'importe quel nom pour la fonction que l'on crée, à l'exception des mots-clés réservés du langage, et à la condition de n'utiliser aucun caractère spécial (« l'underscore » _ est permis). Comme c'est le cas pour les noms de variables, on utilise par convention des minuscules, notamment au début du nom.

10.3 Fonction sans paramètre et sans sortie

On peut définir une fonction dont l'objectif est d'écrire « Bonjour !!! »

```
def bonjour():
    print("Bonjour !!!")
```

On appelle la fonction par : « bonjour() » dans la console ou dans la suite du script. Pas besoin de « return » car la fonction affiche à l'écran, mais ne renvoie pas de valeur.

10.4 Fonction avec un paramètre et sans sortie

```
def bonjour(nom):
    print(f"Bienvenue à toi {nom}.")

bonjour("André")
```

On appelle la fonction par : « bonjour("Marcel") ». "Marcel" est un paramètre.

Il pourrait y avoir plusieurs paramètres (nom1, nom2).

```
def bonjour(nom_1, nom_2):
    print(f"Bienvenue à vous {nom_1} et {nom_2}.")

bonjour("André", "Marcel")
```

10.5 Fonction sans paramètre et avec une sortie

Avec le module random que l'on verra plus en détail, plus tard, on peut créer une fonction qui simule un dé et renvoie une valeur aléatoire entre 1 et 6 :

```
def face_de_dé():
    from random import randint
    return randint(1, 6)

a = face_de_dé()
```

On appelle la fonction par :

```
>>> face_de_dé()
```

dans la console ou bien « a = face_de_dé() » dans le script.

Il est important de comprendre que rien n'est affiché, mais une valeur est renvoyée pour la suite du programme.

10.6 Fonction avec un ou plusieurs paramètres et avec une sortie

Un paramètre et une sortie :

```
def plus_1(a):
    return(a+1)
```

On appelle la fonction par : >>> plus_1(12) ou bien a = plus_1(12).

Deux paramètres et une sortie :

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

On appelle la fonction par :

```
>>> somme(12, 14)
```

ou bien « s = somme(12, 14) » dans le script.

Deux paramètres et deux sorties :

```
def somme_produit(a, b):
    return(a+b, a*b)
```

On appelle la fonction par :

```
>>> somme_produit(12, 14)
```

ou bien « s, p = somme_produit(12, 14) ».

10.7 Variables locales, variables globales

Voici quelques considérations, certes un peu difficiles, mais qui éviteront des confusions sur la gestion par *Python* des variables à l'intérieur et à l'extérieur d'une fonction.

Lorsqu'une fonction est appelée, *Python* réserve pour elle (dans la mémoire de l'ordinateur) un espace de noms. Cet espace de noms local à la fonction est à distinguer de l'espace de noms global où se trouvait les variables du programme principal. Dans l'espace de noms local, nous aurons des variables qui ne sont accessibles qu'au sein de la fonction.

À chaque fois que nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des variables locales à la fonction. Une variable locale peut avoir le même nom qu'une variable de l'espace de noms global mais elle reste néanmoins indépendante.

Les contenus des variables locales sont stockés dans l'espace de noms local qui est inaccessible depuis l'extérieur de la fonction.

Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.

11. Quelques défis

Défi 18 : Écrire une fonction qui reçoit deux notes en paramètres et qui affiche la valeur de la moyenne.

Défi 19 : Écrire une fonction qui reçoit en paramètre la longueur du côté d'un carré, qui calcule et qui affiche son périmètre et son aire.

Défi 20 : Écrire une fonction qui reçoit en paramètre un nombre entier et qui affiche ses diviseurs.

Défi 21 : Écrire une fonction qui reçoit en paramètre un montant hors taxe, et qui renvoie le montant toutes taxes comprises pour un taux de TVA de 20 %.

Défi 22 : Écrire une fonction qui reçoit en paramètre un nombre entier et qui vérifie si 12n-18 > 90.

Les corrections se trouvent pages 39 à 48.

12. Des modules supplémentaires

12.1 Les modules

Qu'est-ce qu'un module en *Python* ? Il s'agit d'une sorte de bibliothèque (un regroupement de fonctions prédéfinies) qui une fois importée permet d'accéder à de nouvelles fonctions.

Il en existe beaucoup. On peut citer :

- le module « turtle » qui permet de réaliser des dessins géométriques ;
- les modules « numpy » ou « scipy » qui permettent de faire du calcul scientifique ;
- le module « matplotlib » qui permet de faire des graphiques en tout genre ;
- le module « random » qui permet de simuler le hasard ;
- le module « folium » utilisé en cartographie ;
- le module « pillow » ou PIL utilisé pour les images et les photographies ;
- le module « networkx » pour les graphes...

Pour pouvoir utiliser un module, il faut l'importer pour le mettre en mémoire.

```
>>> import turtle
```

On trouve aussi :

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

12.2 Installation d'un module supplémentaire

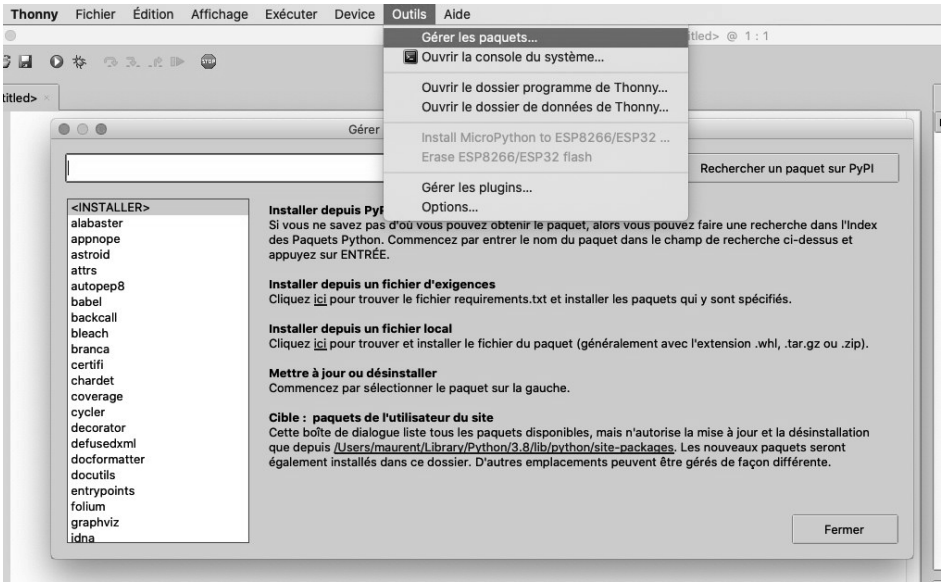
Certains modules sont déjà présents dans la distribution dès l'installation « math », « random », « turtle »..., d'autres doivent être installés pour pouvoir être utilisés, par exemple « matplotlib », la bibliothèque permettant de tracer des graphiques.

Sur *Thonny*, il faut suivre :

Outils → Gérer les paquets

Rechercher par son nom un module.

L'installer, fermer la fenêtre de dialogue.



Attention, Il ne faut pas confondre, l'installation d'un module et son importation en début de script. L'installation ne se fait qu'une fois (éventuellement, il peut y avoir de temps en temps des mises à jour).

Par convention en début de script on procède aux importations des modules que l'on va utiliser dans le programme.

Par exemple, si nous souhaitons calculer cos(pi). Ni cos(), ni pi ne sont connus nativement par *Python* :

```
>>> cos(pi)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'cos' is not defined
```

Nous pouvons donc voir avec le module math, dont c'est la vocation de nous apporter des fonctions mathématiques. Ce module contient une valeur approchée de pi et la fonction cosinus, mais nous aurions pu utiliser d'autres modules comme « numpy » ou « scipy » ou bien « pylab »...

Première possibilité : importer complètement le module, mais séparément les nouvelles fonctions dues à ce module des autres fonctions :

```
>>> import math
```

Ainsi, nous appellerons cosinus et la valeur de pi par :

```
>>> math.cos(math.pi)
-1.0
```

Ici, on doit à chaque fois préciser le nom du module avant la fonction. C'est une précaution bien utile si par exemple le même nom de fonction existe dans deux modules différents.

Pour importer complètement le module, mais séparément des autres fonctions avec un pseudo :

```
import math as ma
>>> ma.cos(ma.pi)
-1.0
```

Cela permet parfois d'alléger le code surtout si le nom du module est long et compliqué.

Pour importer seulement certaines de ces fonctions du module :

```
>>> from math import cos, pi
>>> cos(pi)
-1.0
```

C'est une écriture plus rapide, pour quelques fonctions, si la fonction existait, elle sera écrasée par l'importation.

Pour importer complètement le module, sans séparation des autres fonctions :

```
from math import *
>>> cos(pi)
-1.0
```

Cette méthode est plus légère à coder, mais elle est parfois déconseillée car elle peut-être source d'erreurs.

Par exemple :

```
>>> pow(2, 3)
8
>>> from math import *
>>> pow(2, 3)
8.0
```

Petite analyse des lignes précédentes :

- « pow() » est une fonction connue par *Python*, elle permet ici de calculer le cube de 2.
- Si nous importons tout le module math sans précaution « pow() » est écrasée et remplacée par celle présente dans le module.
- Cette nouvelle fonction renvoie systématiquement un float, même si le résultat est entier. Rien de très méchant, mais cela peut-être parfois source d'erreurs...

12.3 Les modules « math » et « numpy »

Pour pouvoir l'utiliser, il faut l'importer :

```
>>> import math
```

ou éventuellement :

```
>>> from math import *
```

C'est un module qui permet d'avoir accès aux fonctions mathématiques comme le cosinus (cos), le sinus (sin), la racine carrée (sqrt), le nombre π et bien d'autres...

```
>>> from math import *
>>> pi
3.141592653589793
>>> cos(pi)
-1.0
>>> sqrt(25)
5.0
>>> fabs(-65)
65.0
>>> gcd(14, 21)
7
>>> ceil(-7.6)
-7
```

Le module « numpy » est beaucoup plus complet et puissant, il est souvent utilisé pour les vecteurs et plus généralement les matrices. Pour tracer des graphiques, nous utiliserons une fonction qui fabrique une liste de nombres régulièrement répartis entre 2 bornes.

```
>>> from numpy import linspace
>>> x = linspace(-10, 10, 100)
```

12.4 Le module « random »

Le module « random » est un module qui regroupe des fonctions permettant de simuler le hasard. Nous avons déjà croisé des modules (comme le module « turtle ») et comme tous les modules, pour pouvoir l'utiliser, il faut l'importer pour le charger en mémoire. Ainsi, dès que nous voudrons utiliser les fonctions qui suivent pour simuler le hasard, nous mettrons en en-tête la commande :

```
import random
```

ou bien :

```
from random import*
```

Une fonction des plus utiles : « randint(a, b) » : donne un entier choisi au hasard entre a et b compris. Très pratique pour simuler un dé.

```
from random import randint
n = randint(1, 6)
print(n)
```

Pour obtenir un flottant au hasard dans l'intervalle [0 ; 1[on utilise « random() » :

```
from random import random
x = random()
print(x)
```

Pour obtenir un un flottant au hasard dans l'intervalle [a ; b[on utilise « uniform(a,b) » :

```
from random import uniform
x = uniform(12, 16)
print(x)
```

12.5 Le module « matplotlib »

Comme tous les modules, il faut le charger et plus précisément c'est un sous-module qui va nous intéresser : « pyplot ».

Pour cela, nous n'allons pas charger toutes les fonctions comme d'habitude mais l'importer sous un nom plus court à utiliser. On utilisera donc :

```
import matplotlib.pyplot as plt
```

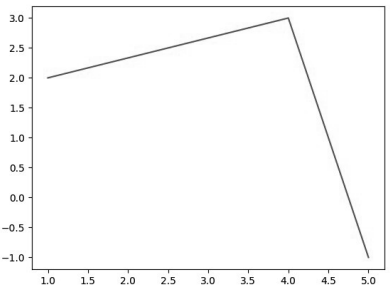
Ce qui signifie que pour utiliser une fonction de ce module comme « show() » par exemple, on devra écrire « plt.show() ».

plt.show() : pour afficher le graphique. Toutes les autres fonctions servent à préparer le graphique mais si on ne demande pas de l'afficher, rien ne se passera.

plt.plot(liste_x,liste_y) : où liste_x est une liste de nombres [, , ...,] et liste_y une liste de nombres [, , ...,] avec le même nombre d'éléments. Alors plt.plot(liste_x, liste_y) placera les points de coordonnées (,), (,), ..., (,) et les reliera de proche en proche par un segment.

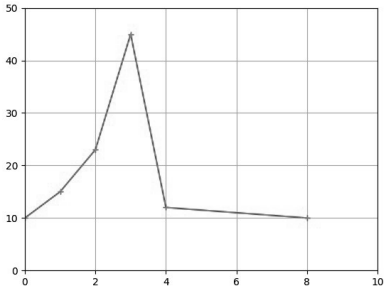
Voici un exemple où on relie les points (1 ; 2), (4 ; 3) et (5 ; -1) :

```
import matplotlib.pyplot as plt
plt.plot([1, 4, 5], [2, 3, -1])
plt.show()
```



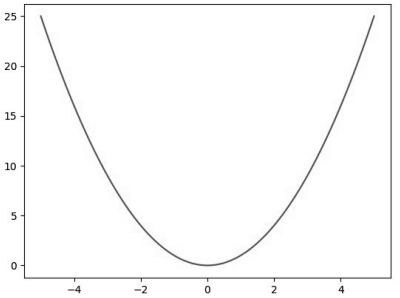
Avec deux listes de valeurs :

```
import matplotlib.pyplot as plt
temps = [0, 1, 2, 3, 4, 8]
vitesse = [10, 15, 23, 45, 12, 10]
plt.axis([0, 10, 0, 50])
plt.grid()
plt.plot(temps, vitesse, 'r+-')
plt.show()
```



Voici un exemple pour tracer la fonction carrée :

```
import matplotlib.pyplot as plt
from numpy import*
x = linspace(-5, 5, 100)
y = x**2
plt.plot(x, y)
plt.show()
```



Nous pourrons modifier la fonction, les bornes et le nombre de points (par exemple 10) utilisés pour bien comprendre le fonctionnement. D'autres fonctions peuvent être utiles :

Pour régler les bornes d'affichage sur les abscisses :

```
plt.xlim(x_min, x_max)
```

Pour régler les bornes d'affichage sur les ordonnées :

```
plt.ylim(y_min, y_max)
```

Affiche un quadrillage en plus sur notre repère :

```
plt.grid()
```

Pour nommer les axes :

```
plt.xlabel("Nom de l'axe x")
plt.ylabel("Nom de l'axe y")
```

Pour mettre un titre au graphique :

```
plt.title("Le titre")
```

Pour ajouter une légende aux courbes :

```
plt.plot(temps, vitesse, 'r+-', 'Vitesse')
```

Pour nommer la courbe :

```
plt.legend()
```

Pour tracer des vecteurs, on peut dessiner des flèches en tapant :

```
plt.quiver(x0, y0, deltax, deltay, angles='xy')
```

La commande précédente dessinera une flèche avec les caractéristiques suivantes :

- « x0 » et « y0 » sont les coordonnées du point de départ ;
- « deltax » et « deltay » sont les composantes du vecteur selon l'abscisse et l'ordonnée ;
- « angles='xy' » permet d'obtenir une flèche qui a une direction cohérente avec l'échelle des axes. Donc c'est un paramètre à utiliser systématiquement en l'écrivant toujours tel quel.

Si à l'affichage, la taille de la flèche est trop faible ou trop importante, il faudra rajouter deux paramètres :

```
plt.quiver(x0, y0, deltax, deltay, angles='xy', scale=1, scale_units='xy')
```

Pour agrandir la flèche choisissons une valeur de « scale » inférieure à un. Pour rétrécir la flèche, choisissons une valeur de « scale » supérieure à 1.

Pour faire des animations :

```
import matplotlib.pyplot as plt
from math import *
for i in range(61):
    plt.cla()
    plt.axis([-5, 5, -5, 5])
    plt.quiver(0, 0, 4*cos((pi/2)-i*pi/30), 4*sin((pi/2)-i*pi/30), angles='xy',
scale=1, scale_units='xy')
    plt.pause(0.01)
plt.show()
```

12.6 Le module « turtle »

Le module « turtle » de *Python* offre aux utilisateurs la possibilité de créer des dessins assez simples. Les commandes ou fonctions à utiliser sont relativement simples à maîtriser. En utilisant le module « turtle », nous pouvons contrôler une tortue qui dessine des segments à l'écran. Ce module trouve son inspiration dans le célèbre langage « Logo », développé par Seymour Papert dans les années 60. Pour pouvoir utiliser les commandes de « turtle », il faut dans un premier les importer, c'est-à-dire télécharger les fichiers nécessaires à l'exécution. Pour cela, il faut en premier lieu saisir la commande suivante :

```
from turtle import*
```

Soit dans la console si nous voulons essayer les différentes fonctionnalités, ou bien en début de script si nous voulons programmer une figure précise.

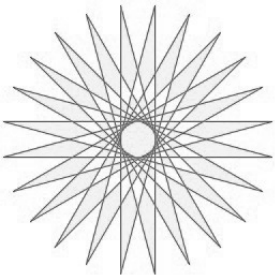
Ensuite, analyser chacune de ces lignes et expliquer ce que fait chacune des commandes suivantes :

```
>>> forward(120)
>>> left(90)
```

```
>>> color("red")
>>> forward(80)
>>> reset()
```

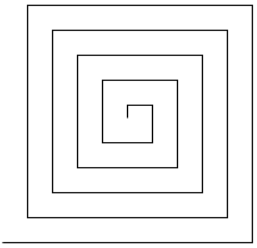
Le script suivant permet de tracer un soleil :

```
from turtle import *

color('red', 'yellow')
begin_fill()

for i in range(24):
    forward(250)
    left(165)
hideturtle()
end_fill()
done()
```



Le script suivant permet de tracer une spirale carrée :

```
from turtle import*

n = 20
u = 10
k = n
for k in range(n):
    longueur = n-k
    forward(longueur*u)
    left(90)
hideturtle()
```



Les fonctions principales de turtle sont :

Nom de la fonction	Abrégé	Description
from turtle import *		Importer les commandes du module Turtle
down()		Descendre le stylo pour dessiner
up()		Relever le stylo pour ne pas dessiner
forward(x)	fd()	Avancer le trait de distance x
backward(x)	bk()	Reculer le trait de distance x
left(α)	lt()	Tourner sur la gauche de α degré
right(α)	rt()	Tourner sur la droite de α degré
goto(x,y)		Aller aux coordonnées x et y du dessin
reset()		Réinitialiser le dessin, tout effacer
fill(‘ ‘)		Remplir un contour d’une couleur
write(‘ ‘)		Écrire un texte à la position du curseur
circle(x)		Tracer un cercle de rayon x