Bases de la programmation en Python 3

Lycée Louis Marchal

2020/2021

NSI

Les variables

Définitions et syntaxe pour l'affectation

Une **variable** est un emplacement dans la mémoire vive (RAM) de l'ordinateur auquel on donne :

- ✓ un nom : qui doit commencer par une lettre et ne contenir que des lettres non accentuées, des caractères _ et des chiffres,
- ✓ une valeur : qui est modifiable,
- ✓ un **type** : entier, réel, etc..., que l'on peut obtenir en Python avec la fonction `type`.

L'**affectation** est une opération permettant de définir la valeur d'une variable
En pseudo code, on note, si, par exemple, le nom de la variable est `age` :

```
age ← 18
```

En Python, on note pour le même exemple : `age = 18`

On dit que "=" est l'**opérateur d'affectation** en Python

Note : Une variable est comme un tiroir dans lequel on range une valeur en mettant une étiquette sur le tiroir. Quand on écrit `a = 3`, en Python, on range 3 dans un tiroir qui s'appelle a. On dit qu'on affecte `3` à la variable `a`. En fait l'ordinateur va ranger 3 dans une case mémoire et il fait correspondre cette case mémoire au nom `a`.

 On écrit toujours le nom de la variable à gauche et sa valeur à droite.

On ne peut écrire : `18 = age`.

On peut modifier la valeur de la variable :

```
age = 19
```
ou

```
age = age + 1
```

 Dans ce type d'instructions on effectue d'abord les opérations à droite du signe égal avant d'affecter le résultat à la variable dont le nom est à gauche.

Les types

Définition

Le **type** d'une donnée est sa nature : un entier, un flottant (un réel), une chaîne de caractères, une liste d'objets ou bien d'autres.

Voici les types que nous allons rencontrer tout au long de l'année de première :

Nature	**Exemples**	**En anglais**	**Pour Python**
entier	`-3 ; 4 ; 0`	integer	`int`
réel ou flottant	`-1,5 ; 3,7; 1,0`	float	`float`
chaîne de caractères	`"informatique" ; "35"; "@!a3"`	string	`str`
listes d'objets	`[3,56,89] ; [ ] ; ["and","or","not"]`	list	`list`
dictionnaire	`{"nom":Dupont, "age" : 41}`	dictionary	`dict`
tuple ou n-uplet	`(3.5,4.1,6.7)`	tuple	`tuple`
booléens	`True; False`	boolean	`bool`

Pour connaître le type d'une donnée on utilise en Python la fonction : `type`

Exemple : `type (1.3)` va donner `float`

Pour écrire un **décimal** on écrira `1.3` et non 1,3 (qui serait un tuple)

Les opérations

Voici les principales opérations (outre +, -, *, /) utiles au lycée :

Nom de l'opération	Symbole	Exemples
Puissance	**	`3 ** 2` donne 9
Division entière ou division euclidienne	//	`25 // 8` donne 3 et `37 // 5` donne 7 c'est le quotient entier de la division de 25 par 8 ou de 37 par 5
Modulo	%	`25 % 8` donne 1 alors que `37 % 5` donne 2 c'est le reste de la division entière.

La racine carrée (square root en anglais, noté **sqrt**), le sinus et de nombreuses autres fonctions mathématiques se trouvent dans le module `math` de Python.

On écrira :

```
from math import sqrt

print(sqrt(36))
```

et 6 sera alors affiché.

Note : on prendra soin de toujours laisser un espace avant et après chaque symbole d'opération ainsi qu'avant et après le symbole d'affectation (=)

Utilisations :

Déterminer si un entier n est **multiple** d'un entier m, en particulier si n est **pair** ou impair.

Les fonctions

Définition

Une **fonction** est un ensemble d'instructions réalisant une certaine tâche.

Pour que cette tâche soit effectuée, il faut **appeler la fonction** dans le programme principal.

Syntaxe sur un exemple :

```
def f(x):
    return 3*x - 5
```

Vocabulaire associé :

`def` est le **mot clé** annonçant la définition d'une fonction

`f` est le nom de la fonction (ce nom peut être choisi de la même manière que celui d'une variable)

`x` est le **paramètre** de la fonction (c'est ce que l'utilisateur doit fournir quand il fait appel à la fonction). On dit aussi l'**argument** quand l'utilisateur le fournit, même si ces deux mots ont des sens un peu différents

les ":" sont absolument obligatoires. Ils finissent la première ligne et sont toujours suivis à la ligne suivante d'une **indentation**, c'est-à-dire d'un espacement correspondant à une tabulation.

`return` est un mot clé. Ce qui suit le `return` est le résultat de la tâche effectuée par la fonction. Ce résultat pourra être récupéré par l'utilisateur. Cette instruction `return` fait aussi quitter la fonction, même si d'autres instructions sont écrites ensuite dans le même bloc.

un **bloc** est une partie de code dont la première ligne finit par ":" et dont les autres instructions sont toutes indentées au même niveau.

Exemple :

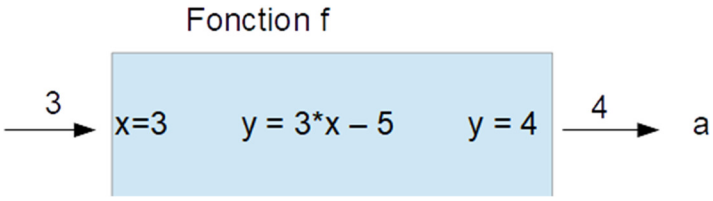
```
1. def f(x):
2.     y = 3 * x - 5
3.     return y
4. #  programme principal
5. a = f(3)  # ici on appelle la fonction f
6. print(a)
7. print(f(7)) #ici on appelle à nouveau la fonction f
```

Notes :

- le # permet de faire des **commentaires** pour rendre le code plus facile à lire et à comprendre. Ce qui est derrière le # n'est pas considéré comme du code.
- On place toujours les définitions de fonctions avant le programme principal
- Appeler une fonction, c'est un peu comme passer commande de la tâche qu'effectue cette fonction.

Quand on exécute ce programme, aucune instruction n'est effectuée jusqu'à la ligne 5

Voici ce qui se passe à la ligne 5 :



L'entier 3 qui arrive en entrée à la fonction f est affecté à la variable x.

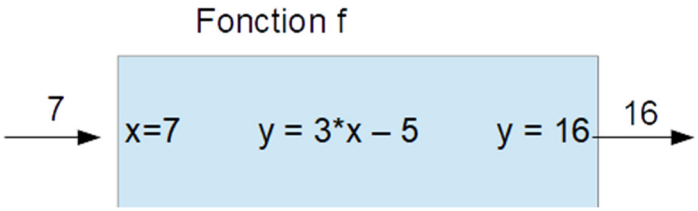
Le calcul 3*x - 5 est effectué et le résultat est affecté à la variable y, qui vaut alors 4.

Ce 4 est retourné (ou renvoyé) au programme principal, il correspond à f(3). Ce résultat est alors affecté à la variable a.

Cette méthode permet de réutiliser le résultat du calcul de f(3) dans le reste du programme. On peut alors, par exemple, l'afficher (ligne 6)

C'est donc à la ligne 5 que les tâches prévues par la fonction f sont donc exécutées.

A la ligne 7, il se passe quelque chose de similaire :



et 16 est alors affiché. L'inconvénient de cette méthode est que le résultat obtenu ne peut être réutilisé ensuite dans le programme. Les tâches prévues par la fonction f sont encore exécutées, mais avec un autre paramètre. C'est tout l'intérêt des fonctions !

Définir une fonction est un peu comme écrire une recette de cuisine : on fait la liste des ingrédients (paramètres d'entrée), puis on note les instructions à effectuer avec ces ingrédients pour arriver à un résultat final. Mais une fois qu'on a fini d'écrire la recette de cuisine, on n'a toujours rien cuisiné : il faut que quelqu'un réunisse les ingrédients et demande à ce que les instructions de la recette soient effectuées avec ceux-ci, pour obtenir le résultat final.

Une fonction peut avoir un, plusieurs paramètres (en entrée), ou aucun, elle peut retourner une valeur, plusieurs valeurs ou aucune.

Syntaxe générale d'une fonction:

```
1. def nom_fonction(parametre_1, parametre_2, ...) :
2.    """Aide de la fonction"""
3.    instruction1
4.    instruction2
5.    instruction3
.
.
?.     return valeur_1, valeur_2,...
```

S'il n'y a aucun paramètre, on ne met rien dans les parenthèses de la ligne 1, mais ces parenthèses restent obligatoires.

Si on ne retourne rien, le `return` n'est pas obligatoire.

Documentation

L'aide de la fonction s'appelle une **documentation** ou **docstring** en anglais et permet de savoir comment utiliser la fonction et ce qu'elle fait. Elle s'écrit entre des triples guillemets.

Il est recommandé d'y donner le type des variables d'entrée et de sortie, les conditions qui doivent être réunies sur ces variables d'entrée ou de sortie, ainsi qu'une phrase expliquant ce que fait la fonction (sauf si c'est évident, comme par exemple dans l'exemple ci-dessous). On pourra aussi ajouter d'autres explications.

Les types des variables d'entrée et de sortie seront symbolisés par quelque chose du genre :

int, float → float

qui signifierait que les deux paramètres d'entrée sont, dans l'ordre un int et un float et que la donnée retournée est un float.

Exemple :

```
from math import sqrt

def calcul_bizarre (x,y):
     """ float , float → float
     Condition : x doit être positif et y non nul
     Cette fonction effectue le calcul de racine(x)/y"""
     return sqrt(x)/y
```

Si, dans la console on tape `help(calcul_bizarre)` on verra s'afficher cette documentation.

Variables locales et variables globales

Les variables définies à l'intérieur d'une fonction sont des **variables locales**. Les paramètres passés à une fonction sont aussi des variables locales.

Les variables définies dans le programme principal sont des **variables globales**.

Les variables locales ne peuvent être utilisées en dehors de la fonction.

Les variables globales peuvent être utilisées dans une fonction, mais pas y être modifiées sans instruction supplémentaire.

Les instructions conditionnelles et les booléens

Syntaxe d'une instruction conditionnelle sur un exemple :

```
a = eval(input("entrez un entier"))
b = a%2  #reste de la division de a par 2
if a == 0:
    print (a, " est nul et pair")
elif b == 0:
    print(a," est pair")
else:
    print(a," est impair")
```

Notes :

- ✓ La première ligne du bloc "**`if`**" finit par ":" et les lignes suivantes sont indentées (avec une tabulation). On sort du bloc en arrêtant l'indentation.
- ✓ on peut mettre autant de **`elif`** que l'on veut. La signification de `elif` est "Sinon si". Il pourrait même n'y avoir aucun `elif`.
- ✓ **`else`** signifie "Sinon" et ce bloc regroupe tous les cas restants.

On ne met aucune condition derrière un `else`.
Il est possible de ne pas mettre de bloc avec `else`.

Définition

Les opérateurs `==`, `<`, `>`, `<=`, `>=` et `!=` (**différent**) sont des **opérateurs de comparaison**.

Ils peuvent être utilisés dans les instructions conditionnelles.

Attention à ne pas confondre "=" qui est l'opérateur d'affectation et "==" qui est un opérateur de comparaison, permettant de **tester si deux données sont égales**.

Définition

Les conditions qui apparaissent dans les structures conditionnelles peuvent prendre deux valeurs :

- ✓ vrai (**`True`** en Python) (Attention aux majuscules !)
- ✓ faux (**`False`** en Python)

Ce sont des **booléens** (type `bool` en Python).

Exemples : `a > 3`, `a == 3`, `a >= 3` sont des booléens

```
a = 1
print( a > 3)          # ceci affichera False
```

Les entrées

Syntaxe pour une saisie ou une entrée

L'instruction **`input`** permet à un utilisateur d'**entrer** ou de **saisir** des données nécessaires pour faire fonctionner un programme. Elle s'utilise d'une manière similaire à la suivante :

```
a = input("Entrez une donnée")
```

La chaîne de caractères `"Entrez une donnée"` s'affichera au moment où l'utilisateur devra entrer des données. Le programme sera arrêté jusqu'à ce que l'utilisateur ait entré une valeur. Cette chaîne de caractères peut être choisie librement.

La donnée entrée par l'utilisateur est alors affectée à la variable dont le nom est à gauche (ici la variable `a`).

Cette variable est alors toujours de type **`str`**, même si l'utilisateur a entré un entier par exemple.

Définition du transtypage et utilisations

Transtypage : changement de type pour une donnée

Pour transformer la chaîne de caractère "32" en un entier on utilisera l'instruction : **`int`**`("32")`.

A l'inverse, pour transformer l'entier 32 en une chaîne de caractères on utilisera :

`str``(32)`.

Notes

Si un programme doit recevoir un entier entré par un utilisateur on peut écrire :

- ✓
```
a = input("Entrez une donnée")
a = int(a)   #transforme le str a en un int
```

ou alors :

- ✓
```
a = int (input("Entrez une donnée"))
```
On combine alors l'instruction `input` et l'instruction `int`

A savoir

On pourra utiliser l'instruction `a =` **`eval`** `(input ("Entrez une donnée"))` pour laisser à Python le soin de fixer le type de la donnée. Il reconnaîtra ainsi un entier (`int`), un réel (`float`) ou une liste, un tuple ou même un dictionnaire par exemple. A éviter, toutefois, si on attend simplement une chaîne de caractères, auquel cas on utilisera l'instruction `input` toute seule.

Les sorties

A savoir

Pour **afficher** une donnée on utilisera la fonction **`print`**.

On peut afficher plusieurs objets les uns à la suite des autres en les séparant par des virgules.

Exemple :

```
nom = "Toto"

age = 3

print ("Mon neveu s'appelle ",nom, " et a ", age, " ans").
```

On remarquera que l'on affiche ainsi les valeurs des variables `nom` et `age`, alors que le reste s'affiche tel qu'il a été tapé.

Notes

- ✓ Les parenthèses sont indispensables.
- ✓ Après une instruction `print` le curseur se met, par défaut, au début de la ligne suivante.
- ✓ Si on écrit `print("lundi", end = ";")` , le curseur ne passe pas à la ligne mais un `";"` s'écrit après l'affichage. L'affichage suivant se fera après ce ";".
  On peut remplacer ce `";"` par n'importe quelle autre chaîne de caractères.

Les boucles bornées (boucles `for`)

Définition.

Une **boucle** est une structure permettant de répéter plusieurs fois l'exécution d'un bloc d'instructions. Il en existe deux types en Python : la boucle "Tant que" (boucle **`while`**) et la boucle "Pour" (boucle **`for`**). On utilise la boucle `for` quand on connaît le nombre de boucles à exécuter.

Syntaxe de la boucle `for` sur un exemple

On utilise souvent cette boucle de la manière suivante :

```
1. for i in range (1, 11):
2.   print ( "7 * ", i, " = " , 7 * i)
3. print ("fin de la table de multiplication de 7")
```

Notes

- ✓ Là encore on a un bloc dont la première ligne finit par ":" et dont les autres lignes sont indentées. Toutes les instructions à répéter doivent donc être indentées
- ✓ L'instruction de la ligne 3 ne sera pas répétée. Si on l'avait indentée elle aurait été répétée.
- ✓ la variable `i` s'appelle un **compteur**

Utilisation de range

- ✓ `range (10, 25)` représente les entiers de 10 à 24.

Faire très attention : le deuxième paramètre de range est toujours exclu de la liste des entiers du `range`.

- ✓ `range(24)` est équivalent à `range(0, 24)` qui contient les 24 entiers qui vont de 0 à 23 ! On pourra donc utiliser `range(24)` pour répéter 24 fois un bloc d'instructions.
- ✓ La syntaxe générale de `range` est :
  `range (valeur_debut incluse, valeur_fin_exclue, pas),`
  par exemple `range( 3, 27, 2)` représente les entiers : 3, 5, 7, 9, ……25.
  On a avancé avec un **pas** de 2, en s'arrêtant juste avant 27, car 27 est exclu.
  Si on n'indique pas le pas, il vaudra 1. 1 est le pas par défaut.

Programmes classiques : calcul d'une somme et calcul d'un minimum

Les boucles conditionnelles (boucles `while`)

Syntaxe de la boucle `while` ("Tant que" en français)

```
while condition :
     instructions indentées qui sont répétées
instructions qui ne sont pas dans répétées
```

Les conditions utilisées sont les mêmes que dans les instructions conditionnelles.

Complément :

L'instruction **`break`** permet de sortir de sortir d'un bloc `while` ou `for` sans exécuter les boucles restantes

Exemple classique avec un compteur :

```
1. from random import *
2. de = 0
3. compteur = 0
4. while de != 6 :
5.   de = randint(1,6)
6.   print("score : ", de)
7.   compteur = compteur + 1
8. print(compteur)
```

Ce programme compte combien de jets de dé sont nécessaires pour obtenir un 6.

Note : on utilisera souvent un tel **compteur** dans une boucle `while`.

Les listes ou tableaux

Définition

Une **liste** ou un **tableau** est une collection de données rangées entre crochets et séparées par des virgules. C'est un objet de type `list`.

Exemples :

liste1 = [3, 5, 7, 23, "a"]

liste2 = [ ] # c'est la **liste vide**.

On utilise des virgules pour séparer les termes, pas des point virgules.

Définition : index (ou indices)

Dans une liste appelée t (car c'est aussi un tableau) :

- ✓ Le premier terme de la liste est d'index 0, c'est `t[0]`
- ✓ Le deuxième élément de la liste est d'index 1, c'est `t[1]`
- ✓ Plus généralement, le terme d'index i est `t[i]`.

Exemples :

```
t = [ 2, 4, 8, 16, 32, 64]
print (t[0])      # affiche 2
print ( t[3])     # affiche 16
print (t[5])      # affiche 64
```

On remarquera que la longueur de la liste est 6, mais que l'index du dernier terme est 5, car on commence à l'index 0.

Index négatifs

Le dernier terme d'une liste `t` est `t[-1]`, l'avant dernier est `t[-2]` …..

Longueur d'une liste

La longueur d'une liste `t` est **`len`** `(t)`.

La fonction `len` s'applique aussi à la longueur d'une chaîne de caractères, d'un tuple, d'un dictionnaire ou de tout autre objet itérable (qui peut se parcourir avec une boucle `for`).

Le dernier terme d'une liste `t` est d'index `len(t) - 1`

Ajout d'un élément à la fin d'une liste

Pour ajouter 3 à la fin de la liste `t` on écrira : `t.append(3)`

Concaténation

La **concaténation** permet de "coller" deux listes l'une à l'autre : cela se fait en utilisant l'opérateur "+".

Exemple :

["mon","ma","mes"] + ["ton", "ta", "tes"] donne ["mon","ma","mes", "ton", "ta", "tes"]

Parcours d'une liste

On dit qu'on **parcourt** un objet, en programmation, lorsque l'on passe par chacun des termes de cet objet à l'aide d'une boucle (l'objet doit être **itérable,** par exemple, un entier n'est pas itérable).

On peut parcourir une liste `t` de deux manières.

- ✓ par ses index :

```
for i in range (len(t)):
    print ( t[i] )
```

Cette méthode permet de modifier les termes de la liste.

- ✓ par ses termes :

```
for terme in t :
    print(terme)
```

Avec cette méthode on ne peut modifier les termes de la liste.

Compléments :

Quand on crée une liste `t`, l'ordinateur associe le nom `t` à une case mémoire qui contient l'adresse du premier terme de la liste (cela s'appelle un pointeur en informatique).
Quand on modifie la liste `t`, on ne modifie pas cette adresse.
Si `t` est une variable globale dans un programme, on pourra donc modifier cette liste dans une fonction sans problème. Il n'est pas recommandé de la faire : il vaudra mieux passer la liste en paramètre et la récupérer en la retournant à la fin de la fonction.
Par ailleurs si on écrit :

```
tbis = t
```

et que l'on modifie `t` alors `tbis` sera aussi modifiée, car `t` et `tbis` pointent vers la même adresse en mémoire : il n'y a en fait qu'une seule liste qui a deux noms différents (deux alias).

Algorithmes de référence :

Fonction qui détermine s'il y a une **occurrence** d'un élément dans un tableau
Fonction qui compte le nombre d'occurrences d'un élément dans un tableau
Fonction qui détermine la **somme** des éléments d'un tableau
Fonction qui détermine le **minimum** ou le **maximum** des éléments d'un tableau

Les chaînes de caractères

Définition

Une **chaîne de caractères** est une suite de caractères placée entre des guillemets.
C'est un objet de type **`str`**.

Exemple :

```
c = " Dupont@monmail.fr "
```

Similarité avec les listes

`len (c)` est la **longueur** de la chaîne `c`
`c[0]` est le premier caractère de la chaîne, `c[1]` le second et `c[-1]` est le dernier
`"Hey " + "Toto"` est la **concaténation** de ces deux chaînes et donne `"Hey Toto"`
`c = ""` est la chaîne vide
On peut parcourir une chaîne par ses **index** : `for i in range (len(c))`
On peut aussi parcourir une chaîne par ses termes : `for caractere in c`

Différence essentielle avec les listes

Les chaînes ne sont pas modifiables. Cela signifie que l'on peut utiliser `c[0]`, `c[1]` etc…., mais qu'on ne peut leur affecter une certaine valeur.

On ne peut écrire `c[0] = "d"` par exemple.

Instruction join

Si on veut modifier facilement une chaîne, on la transforme en une liste, on la modifie, puis on transforme le résultat en une chaîne.

```
c = " Dupont@monmail.fr "

c = list( c)

#  c est alors une liste constituée de tous les caractères de c

c[0] = "d"

c = "".join(c)      # c vaut alors " dupont@monmail.fr "
```

Note :
si on écrit : `"-".join( ["a", "b", "c"]` on obtiendra la chaîne `"a-b-c"`

Instruction split (briser en anglais)

`"Toto est un bon fils".split()` donne la liste
`["Toto","est","un","bon","fils"]`
`"1-2-3-4".split ("-")` donne `["1","2","3","4"]`
Cette instruction fait l'inverse de ce que fait `join`.

Les imports

Il existe de nombreuses bibliothèques de fonctions pour Python. On peut les utiliser dans les programmes, à condition d'avoir importé ces fonctions.

Trois manières de faire des imports

- ✓ `from random import *` permet d'importer toutes les fonctions du module (ou de la bibliothèque) `random`. Elles peuvent alors être utilisées dans le programme.
- ✓ `from random import randrange` permet d'importer uniquement la fonction `randrange`
- ✓ `import random` permet d'importer le module `random`, mais pour utiliser `randrange` il faudra écrire `random.randrange`.
  Cette méthode a l'air plus compliquée mais elle permet d'éviter toute confusion de noms entre les fonctions du module `random` et celles du programme.

Notes sur le module random

Il comporte entre autres les fonctions :

- ✓ **`randrange`** `(1, 13)` permet de choisir un entier au hasard entre 1 et 12 (fonctionne comme `range`)
- ✓ **`randint`** `(1, 13)` permet de choisir un entier entre 1 et 13
- ✓ **`choice`** `(t)` permet de choisir un terme au hasard dans la liste

On utilisera aussi le module `math`, qui contient les fonction `abs, sin, cos, sqrt` et bien d'autres : taper `help("math)` dans une console pour en avoir le détail.

Autres modules utilisés : `turtle, pygame, pillow`.....

Tout fichier avec une extension ".py" est un module. Vous pouvez donc créer vos propres modules, puis les importer dans un autre fichier.

A savoir

Les imports s'écrivent au début du programme, avant les fonctions. Le programme principal se place à la fin.

Les erreurs

Il est important de comprendre les messages d'erreur que peut renvoyer l'interpréteur Python : on pourra ainsi plus facilement débuguer les programmes.

Voici les principaux d'entre eux, mais il y en a bien d'autres.

1) Erreurs d'indentation : `IndentationError`

- ✓ Si on a mal indenté le code (avec un espace de trop par exemple), l'interpréteur Python dira : `IndentationError : unexpected indent`
- ✓ S'il manque une indentation (après la première ligne d'un bloc se finissant par ":"),
  `IndentationError : expected an indented block`
- ✓ Si à la sortie d'un bloc, on n'a pas aligné correctement le code avec ce qui précède, `IndentationError: unindent does not match any outer indentation level` sera écrit.

2) Erreurs de syntaxe : `SyntaxError`

Définition :

> En informatique, la **syntaxe** d'un langage de programmation est l'ensemble des règles qui définissent l'arrangement correct des mots et symboles de ce langage.

Par exemple, c'est la syntaxe de Python qui impose de mettre ":" à la fin des premières lignes des blocs et de faire une indentation ensuite, ou d'écrire "`for i in range(4):`" et pas "`For i in range(4)`". Dès que la syntaxe de Python n'est pas respectée, l'interpréteur écrit `SyntaxError`.

`SyntaxError: invalid syntax` se produit par exemple si :

- ✓ on a oublié les ":" à la fin de la première ligne d'un bloc ;
- ✓ si on a confondu "=" et "==" ;
- ✓ si on a oublié un opérateur (si on écrit `2a` au lieu de `2*a` par exemple).

`SyntaxError : EOL while scanning …` ou `while parsing` se produit quand on a mal équilibré les symboles ouvrants et fermants (parenthèse, crochet, accolade, guillemet…manquant). Attention l'erreur est souvent indiquée pour la ligne d'après.

3) Erreurs de nom : `NameError`

L'erreur `NameError : name 'var' is not defined` apparaît si l'on utilise une variable (nommée ici `var`) que l'on n'a pas encore définie.

Ceci se produit notamment quand :

- ✓ on a commis une erreur en tapant le nom de la variable (par exemple on a écrit `talbe` au lieu de `table` qui était la variable que l'on voulait utiliser);
- ✓ on a oublié d'initialiser la valeur d'une variable avant une boucle

```
for i in range (1, 11):
     somme = somme + i;
```

La variable somme n'a pas été **initialisée** (on ne lui a pas donné de valeur initiale avant de l'utiliser).

- ✓ on a oublié un cas dans une série de conditions

```
age = int(input("Entrez votre âge"))
if age < 18:
     etat = "mineur "
elif age > 18:
     etat = "majeur "
print(etat).
```

On a oublié le cas où l'âge valait 18, si l'utilisateur entre 18 on aura une `NameError : name 'etat' is not defined`.

4) Erreurs de type : `TypeError.`

- Ces erreurs surviennent surtout si on essaie de faire une opération qui n'est pas permise avec les types de données utilisés.
  - ✓ Cela arrive souvent suite à l'utilisation de `input`, quand on n'a pas pris la précaution de transformer la donnée entrée de manière à ce qu'elle ait le type souhaité.

```
1. nb = input(" veuillez saisir un nombre : ")
2. nb = 1 + nb
> line 2, TypeError: unsupported operand type(s) for +:
'int' and 'str'
```

`nb` est un `str` et on ne peut faire l'opération "+" entre un `str` et un `int`.

  - ✓ Cela peut également se produire si on a écrit 1,42 au lieu de 1.42 pour un décimal.
- Elles se produisent également lors d'une mauvaise utilisation des parenthèses :

```
a = 3 (4 + 5)
> TypeError: 'int' object is not callable
```

Dans ce cas, l'interpréteur Python pense que vous essayez d'appeler (*call* en anglais) une fonction qui s'appellerait 3. Or 3 ne peut être une fonction, puisque les noms de fonctions ne peuvent commencer par un chiffre. Il dit donc que l'`int 3` ne peut être appelé. Il manque simplement l'opérateur "*" avant la parenthèse. On dit que Python ne comprend pas les **multiplications implicites** (qui n'ont pas été explicitement écrites avec l'opérateur "*").

- Elles apparaissent aussi lors d'appels de fonctions quand on ne met pas le même nombre de paramètres que dans la définition de la fonction.

```
1. def f(x):
2.     return 2*x
3. f(4, 10)
> line 3, TypeError: f() takes 1 positional arguments but 2 were given.
```

Il faut lire attentivement le texte des erreurs qui est souvent assez clair.

"argument" a une signification proche de "paramètre".

5) Erreurs d'indice : `IndexError`

Cette erreur survient lorsqu'on essaie d'utiliser un indice qui ne fait pas partie des indices autorisés pour une liste.

```
1. tab = [4, 2, 10]
2. print(tab [3])
> line 2, IndexError: list index out of range
```

Là encore le texte est explicite.

Ceci se produit souvent quand on oublie que l'indice maximal est `len(t) -1` pour une liste `t`

6) Erreurs liées à la division par zéro : `ZeroDivisionError`

On ne peut diviser par 0 et `35 % 0` (**modulo**) n'a aucun sens.

Le débogage et les tests

Débuguer un programme c'est chercher les erreurs dans ce programme.

Deux cas peuvent se présenter lorsqu'on exécute un programme.

1. Votre programme provoque une erreur. Il faut alors essayer de réparer ce bogue (*bug* en anglais) en commençant par analyser posément le texte de l'erreur qui explique quelle est l'erreur et à quelle ligne elle a été détectée (voir le chapitre "Erreurs").
2. Votre programme produit un résultat et fonctionne sans produire d'erreur. **Cela ne veut pas dire qu'il est correct**. Il est très important de comprendre cela.
   Si votre programme fournit un résultat faux, il y a aussi un bogue !

Il est souvent très difficile de prouver qu'un programme est correct (mais cela se fait), en revanche il est indispensable de réaliser un jeu de tests suffisant pour être persuadé que le programme est correct.

Il faut tester :

- ✓ tous les cas quand il y a des instructions conditionnelles ;
- ✓ avec des données pour lesquelles on a un moyen de vérification (une somme de trois termes au lieu de 100, deux boucles au lieu de mille, avec une erreur de 0.1 au lieu de 0.0001 etc…), quand c'est possible ;
- ✓ les cas limites ou extrêmes (même s'ils sont très improbables);
- ✓ le fonctionnement des boucles : est-ce que tout se passe comme attendu lors du premier passage dans une boucle, lors du deuxième et lors du dernier.
  
  On voit beaucoup d'erreurs avec l'utilisation de `range` (la dernière valeur est exclue, il ne faut pas l'oublier) ;
- ✓ chaque fonction écrite, chaque nouvelle partie de code, sans attendre d'avoir de nombreuses lignes à tester en même temps. Le code doit être écrit très progressivement même si les premières versions ne donnent qu'un résultat partiel ou intermédiaire.

En cas d'erreur on pourra :

- ✓ faire afficher des résultats intermédiaires ;
- ✓ utiliser le débogueur de l'éditeur où le code a été écrit ;
- ✓ faire tourner à "la main" le programme, ou la partie de programme, qui pose problème ;

Pour faciliter le débogage et la lecture d'un programme par d'autres (dont les autres membres d'un groupe de travail, le professeur) on prendra soin de :

- ✓ choisir des noms de variables cohérents et compréhensibles;
- ✓ rédiger une documentation pour chaque fonction;
- ✓ mettre des commentaires si la compréhension du code risque d'être, malgré tout, compliquée

Sommaire

Chapitre	Notion	Page	Vu	En voie d'acquisition	Acquis
Fonctions	Variables locales	8			
	Variables globales	8			
	Argument	8			
Instructions conditionnelles	**Instructions conditionnelles**	9			
	`if`	9			
	`elif`	9			
	`else`	9			
	Différent (!=)	9			
	Opérateurs de comparaison	9			
	`Comparaison`				
	Test d'égalité (==)	9			
	Booléen	9			
	`True`	9			
	`False`	9			
	`bool`	9			
	Egal et égal-égal	9			
Entrées	**Entrées**	10			
	Saisie				
	`input`	10			
	`str`	10			
	`int`	10			
	`eval`	10			
	Transtypage	10			
Sorties	**Sorties**	11			
	Affichage				
	`print`	11			

Chapitre	Notion	Page	Vu	En voie d'acquisition	Acquis
Sorties	`print` avec `end = ???`	11			
	`end`	11			
Boucles bornées (for)	**Boucles `for`**	12			
	`for`	12			
	`range`	12			
	Compteur	12			
	Pas	12			
	Somme	12			
	Minimum	12			
	Maximum	12			
Boucles conditionnelles (`while`)	**Boucles `while`**	13			
	`while`	13			
	`break`	13			
	Compteur	13			
Listes ou tableaux	**Listes**	14			
	Tableau	14			
	Liste vide	14			
	Index	14			
	Indice	14			
	`len`				
	Longueur	14			
	`append`	14			
	concaténation	15			
	Plus (+) avec une liste	15			

Chapitre	Notion	Page	Vu	En voie d'acquisition	Acquis
Listes ou tableaux	Parcours d'un objet	15			
	Objet itérable	15			
	`for terme in ma_liste`	15			
	Occurrences dans un tableau	15			
	Minimum d'un tableau	15			
	Maximum d'un tableau	15			
	Somme des éléments d'un tableau	15			
	Ajout à la fin d'une liste	15			
Chaînes de caractères	**Chaînes de caractères**	16			
	`str`	16			
	Longueur	16			
	`len`	16			
	Index	16			
	Indice	16			
	`for caractere in chaine`	16			
	Concaténation	16			
	Plus (+) avec une chaîne	16			
	`join`	16			
	`split`	16			
Imports	**Imports**	17			
	`random`	17			
	`randrange`	17			
	`randint`	17			
	`choice`	17			

Chapitre	Notion	Page	Vu	En voie d'acquisition	Acquis
P-uplets	**P-uplets**				
	Tuples				
	`tuple`				
	Mutable				
	Disperser un tuple				
	Echange de valeurs				
Dictionnaires	**Dictionnaires**				
	`dict`				
	`key`				
	Clé				
	`value`				
	Valeur				
	`item`				
	Longueur d'un dictionnaire				
	len avec un dictionnaire				
	`del`				
	Parcours d'un dictionnaire				
	`with open….`				
	`read`				
	`chr`				

Index

J	`join`	16
K	`key`	
L	`len` avec un tuple	
	`len` avec une liste	16
	`len` avec un dictionnaire	
	`list`	3
	Liste	20
	Liste vide	14
	Listes	14
	Longueur d'une liste	14
	Longueur d'une chaîne	16
	Longueur d'un dictionnaire	
M	Maximum	12
	Maximum d'un tableau	15
	Minimum	12
	Minimum d'un tableau	15
	Modulo (%)	4, 20
	Mot clé	5
	Multiple de ???	4
	Multiplication implicite	20
	Mutable	
N	`NameError`	19
O	Objet itérable	15
	Occurrences dans un tableau	15
	Opérateurs de comparaison	9
	Opérations	4
	Opérateur d'affectation	2

	Tri par sélection	
	`True`	9
	`tuple`	3
	Tuples	
	Type	3
	`TypeError`	19
U		
V	Valeur	
	`value`	
	Variables	2
	Variables globales	8
	Variables locales	8
W	`while`	13
	`with open….`	
X-Y		
Z	`ZeroDivisionError`	20