Leçon d'informatique : Variables et fonctions

S. Benlhajlahsen - PCSI$_1$

Sommaire

Introduction : Dans ce chapitre, nous allons introduire les notions suivantes :

- les **variables** ;
- les **expressions** ;
- les objets `int`, `float` et `str` ;
- documentation d'une fonction : `help`.

I Déclarations de variables

I.A Variables

Une variable a quatre paramètres associés :

- **nom** : utilisé pour désigner la variable et écrire des calculs symboliques avec ;
- **valeur** : quand le calcul sera mené, l'ordinateur pourra remplacer le nom par la valeur ;
- **adresse** : emplacement dans la mémoire de l'ordinateur où la valeur est stockée ;
- **type** : détaillé plus bas.

Une variable ne garde aucune trace de ses valeurs antérieures. Seule sa valeur actuelle est stockée à l'adresse indiquée.

I.B Noms de variables

Syntaxe Un nom de variable est une séquence de lettres (a . . . z , A . . . Z), de chiffres (0 . . . 9) et du caractère _ (*underscore*) [a].
Le nom doit **toujours commencer par une lettre**.

- La casse est significative [b].
- Il existe une liste de *mots-clés* réservés qui ne peuvent pas être employés comme noms de variables.

a. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que $, #, @, etc. sont interdits.
b. les caractères majuscules et minuscules sont distingués

Interdiction stricte Dans le cas de Python, il faut éviter mots-clés :

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		

Il y a, de plus, quelques usages qu'il est recommandé de respecter.

- Choisissez de préférence des noms aussi explicites que possible, de manière à exprimer clairement ce que la variable est sensée contenir. Par exemple si on a besoin d'un majorant, d'un minorant et d'une moyenne il vaut mieux éviter des les appeler `m1`, `m2` et `m3`. Il vaudrait mieux utiliser simplement `majorant`, `minorant` et `moyenne` [1].
- Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre).
- Les majuscules ou le symbole _ permettent de découper le nom : `premierTerme` ou `premier_terme`.

I.C Affectation ou assignation

Une variable définit une association entre son nom et une valeur stockée dans la mémoire. Si on exécute l'instruction `b=1`, la variable `b` est associée à la valeur 1. La valeur à calculer peut être donnée explicitement mais peut être aussi le résultat d'un calcul.

À retenir : Une expression est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur.

À retenir : La forme générale de l'affectation est donc `variable = expression`

Remarque : Cette instruction se fait en 3 étapes.

- L'expression à droite est évaluée, le résultat est stocké à une adresse `ad`.
- Un nouveau nom de variable est défini (même s'il existait déjà avant)
- `ad` est lié au nom de la variable.

Dans certains langages, il faut réserver la place en mémoire *avant* d'utiliser la variable (on initialise la variable) ; il faut pour cela spécifier le type de la variable.

Remarque La syntaxe d'affectation n'est pas commutative (voir figure 2).

```
1 >>> 1=b
2 File "<stdin>", line 1
3 SyntaxError: can't assign to literal
```

FIGURE 1 – Erreur de syntaxe.

Typage dynamique Dans Python les trois étapes se font en une seule instruction simplement notée `=`. C'est un typage dynamique qui s'oppose au type statique dans d'autres langages où le typage doit être déclaré à l'avance.

Exemple Par exemple, l'instruction `a = 2 + 3` (voir figure 2) :

1. calcule 2 + 3, ce qui donne l'entier 5,
2. réserve de la place en mémoire pour un entier,
3. y stoke l'entier 5,
4. associe le nom de variable `a` à cette adresse.

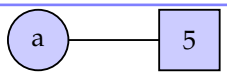


FIGURE 2

Permutation de variables On souhaite permuter les valeurs de deux variables `a` et `b` (voir figure 3). Le script de gauche (lignes 5 et 6) se comprend ainsi :

- on affecte la valeur de `b` (donc 1) à la variable `a`. A ce moment, `a` et `b` valent tous les deux 1. ;
- ensuite, on affecte la valeur de `a` à `b`.

1. Par contre `majorant_des_termes_de_la_suite` est sans doute un peu excessif.

Dans ce processus, on a perdu la valeur initiale de `a`. Une première solution est alors de stockée dans une variable temporaire `temp` la valeur de la variable `a` (voir figure 3). Une seconde solution est d'utiliser une affection parallèle (voir dernière ligne de la figure 4).

```
>>> a=3
>>> b=1
>>> print(a,b)
3 1
>>> a=b
>>> b=a
1 1
```

```
>>> a=3
>>> b=1
>>> print(a,b)
3 1
>>> temp=a
>>> a=b
>>> b=temp
>>> print(a,b)
1 3
```

FIGURE 3 – Permutation de valeurs. À gauche, méthode naïve et fausse. À droite, on crée une variable temporaire.

Affectations multiples ou parallèles. On considère l'exemple de la figure 4. On a tout d'abord affecter une valeur à plusieurs variables `a` et `b`. Ensuite, on a affecté en parallèle les valeurs 7 et 9 aux variables `x` et `y`. Enfin, en ligne 11, on a permuter les valeurs de `x` et `y`.

```
>>> a=b=3
>>> a
3
>>> b
3
>>> x, y = 7, 9
>>> x
7
>>> y
9
>>> x, y = y, x
```

FIGURE 4 – Affectations multiples

II Entiers Python

II.A opérations de base

Les entiers, en Python, sont des objets de type `int`. Les opérations sur les entiers sont, en Python,

- +, -, $*$ l'addition, la soustraction et la multiplication
- $**$ l'exponentiation `n**p` donne n^p, elle peut aussi s'écrire `pow(n, p)`,
- // et % donnent respectivement le quotient et le reste de la division euclidienne, on les obtient aussi par la fonction `divmod`,
- `abs` renvoie la valeur absolue.

Voir quelques exemples en figure 5.

II.B modification

Syntaxe Pour modifier une variable `a`, on peut simplement lui affecter une nouvelle expression. Si on veut lui ajouter 1, on peut utiliser les deux syntaxes de la figure 6.

```
>>> type(2)
<class 'int'>
>>> 2+3
5
>>> 3*14
42
>>> 3**5
243
>>> 18//7
2
>>> 18%4
2
>>> divmod(7,4)
(1, 3)
```

FIGURE 5

```
    >>> a = 3
    >>> a = a + 1
```

```
    >>> a = 3
    >>> a += 1
```

FIGURE 6 – Syntaxes de modification d'un entier où on lui ajoute 1.

Autre modifications

- pour **soustraire** 3, on peut utiliser les deux syntaxes `a = a-3` et `a -= 3`;
- pour **multiplier** par 10, on peut utiliser les deux syntaxes `a=a*10` et `a *= 10`;
- pour **diviser** par 2, on peut utiliser les deux syntaxes `a = a/2` et `a /= 2`;

III Flottants

Les valeurs approchant les nombres réels sont appelés **nombres flottants** pour rappeler que leur virgule « flotte »en fonction de la précision d'approximation.
En Python, ces valeurs ont le type `float`. On les représente avec **un point à la place de notre virgule**; on peut aussi utiliser un exposant `e` qui signifie puissance 10; `898547e-5` signifie `8.98547`.

Remarque Les flottants ont une précision limitée. Une conséquence est qu'un test de nullité d'un flottant ne donne que rarement le résultat `True` même si, mathématiquement, la variable devrait avoir la valeur nulle.

Les flottants supportent les mêmes opérations que les entiers avec la division en plus. Si on mélange entiers et flottants le résultat sera un flottant, lors d'une division de deux entiers le résultat est un flottant.
Il est possible de convertir une variable `float` en variable de type `int` avec la fonction `int()` qui réalise une troncature, ce n'est pas la partie entière. La fonction `float()` réalise l'inverse.

Remarque L'import du module `math` ajoute un grand nombre d'opérations mathématiques usuelles (voir figure 7)[a].

a. La liste est fourni par `help(math)`

IV Booléens

IV.A opérations de base

Le type `bool` est réservé aux variables pouvant prendre deux valeurs : `False` ou `True`, c'est-à-dire vrai ou faux.

- Une erreur classique est d'oublier la majuscule initiale.
- Les tests d'égalité, noté `==`, ou d'inégalité, noté `!=`, donnent un résultat booléen.

```
>>> import math
>>> type(3.2)
<class 'float'>
>>> 2 + 3.5
5.5
>>> 7/3
2.333333333
>>> int(4.3)
4
>>> int(-3.2)
-3
>>> float(5)
>>> 2.3**1.4
3.2093639532679714
>>> math.sin(math.pi/4)
0.7071067811865475
>>> math.log(2.7)
0.9932517730102834
```

FIGURE 7

- Les comparaisons d'entiers ou de flottants ont pour résultat des valeurs booléennes. Les opérateurs sont notés `<, >, <=, >=`
- On dispose aussi des opérateurs logiques usuels : `and, or, not`.

Voir la table 1 et les exemples de la figures 8.

a	b	a and b	a or b	not b
True	True	True	True	False
True	False	False	True	True
False	True	False	True	
False	False	False	False	

TABLE 1

```
>>> (5+7) == 12
True
>>> (5+7) != 13
True
>>> 6 > 8
False
>>> 6 <= 8 and 5 < 3
False
```

FIGURE 8

IV.B autres opérations

Test de parité d'un entier L'instruction `a%2==0` vérifie que le reste de la division euclidienne de `a` par 2 est nul. On teste si `a` est un entier pair.

Test d'imparité d'un entier L'instruction `a%2==1` vérifie que le reste de la division euclidienne de `a` par 2 vaut 1. On teste si `a` est un entier impair.

V Chaînes de caractère

Une chaîne de caractère[2] est une suite de caractères, de type `str`. On définit une chaîne en en écrivant les caractères entourés d'apostrophes simples, `nom = 'Jean Dupont'`, ou doubles, `nom = "John Wayne"`.

Remarque Les caractères sont codés en mémoire par des entiers. Les caractères usuels sont associées à un entier entre 0 et 255 selon un code ASCII.

Fonctions associées à la classe `str` Les fonctions de conversion sont accessibles en python :

- `ord` donne le code ASCII d'un caractère,
- `chr` donne le caractère associé à un entier.

On se reportera à la figure 9

```
>>> type("hello")
<class 'str'>
>>> a = "hello"
>>> ord('a')
97
>>> ord('\n')
10
>>> chr(125)
'}'
>>> chr(32)
' '
>>> chr(20)
'\x14'
```

FIGURE 9

Remarque hors-programme L'American Standard Code for Information Interchange (Code américain normalisé pour l'échange d'information), plus connu sous l'acronyme ASCII, est une norme informatique de codage de caractères apparue dans les années 1960. Dans la table de la figure 10, on a représenté les principaux caractères de ce système d'encodage.

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

FIGURE 10 – Table des caractères ASCII entre 32 et 127.

V.A Conversion

Python permet de convertir les nombres en chaînes de caractère à l'aide de la fonction `str` (voir figure 11). Si une chaîne `ch` représente un réel, on peut la convertir avec la fonction en flottant par `float(ch)` (voir figure 12). Si c'est en entier, on

2. *string* en anglais.

demandera `int(ch)`. Si une chaîne représente une expression, Python peut l'évaluer directement avec la fonction `eval` (voir figure 13).

```
>>> a = 1/33
>>> b = 257
>>> str(a)
'0.3333333333333333'
>>>str(b)
'257'
```

FIGURE 11

```
>>> float("3.14159")
3.14159
>>> int("254")
254
>>> float("254")
254.0
```

FIGURE 12

```
>>> x = 1
>>> eval("x + 1.2")
2.2
```

FIGURE 13

V.B Affichage

`print` permet d'afficher des chaînes de caractère à l'écran ou l'interprétation des valeurs d'expressions sous forme de chaînes de caractère (voir figure **??**).

Afficher plusieurs paramètres Si on envoie plusieurs paramètres à imprimer, séparés par une virgule, ils seront séparés par un espace. On peut changer ce séparateur (voir figure 14).

Utilisation de la documentation En cas de doute, on peut utiliser la documentation à l'aide de la fonction `help` (voir figure 15).

V.C opérations sur les chaînes de caractère

syntaxe L'opérateur `+` entre deux chaînes les met bout-à-bout. On parle de **concaténation** (voir figure 16).

syntaxe L'opérateur `*` permet la répétition de chaîne (voir figure 17).

syntaxe - longueur d'une chaîne La syntaxe `len(ch)` fournit le nombre de caractères dans une chaîne `ch` (voir figure 18).

Introduction : À la fin de ce cours, il faudra appréhender les notions suivantes :

- utilisation de fonctions prédéfinies,
- importation de modules de fonctions : `import`,

```
>>> x = 3
>>> y = 'fois'
>>> z = 5
>>> print(x, y, z)
3 fois 5
>>> print(x, y, z, sep=';')
3;fois;5
>>> print(x, y, z ,end='\n')
3
fois
5
```

FIGURE 14

```
>>> help(print)
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
    ...
    sep: string inserted between values, default a space.
    end: string appended after the last value, default a newline.
```

FIGURE 15

- déclaration d'une nouvelle fonction : `def`,
- définition du corps d'une fonction en blocs d'instructions indentées,
- renvoi de valeurs `return`,
- documentation d'une fonction : `help`,

VI Des fonctions prédéfinies

Par défaut, le langage propose quelques fonctions prédéfinies.

VI.A Les fonctions de base

Les fonctions utilisables dans python sont peu nombreuses, on en donne la liste en table 2.
On y reconnait :

1. des fonctions mathématiques : `abs`, `divmod`, `max`, `min`, `pow`;
2. les nombreuses fonctions de conversion : `bin`, `bool`, `chr`, `complex`, `float`, `hex`, `int`, `list`, `oct`, `ord`, `round`, `set`, `str`, `tuple`;
   - `bin`, `hex`, `oct` donnent les écritures des entiers en base 2, 16 et 8 sous forme de chaînes de caractères précédées respectivement de `'0b` `'0x` et `'0o'`

```
>>> bin(64)
'0b1000000'
```

```
>>> a = "bon"
>>> b = "jour"
>>> a+b
"bonjour"
```

FIGURE 16 – Concaténation de chaînes

```
>>> a = "bon"
>>> 3*a
"bonbonbon"
```

FIGURE 17 – Répétition de chaîne.

```
>>> a = "bon"
>>> len(a)
3
```

FIGURE 18 – Répétition de chaîne.

```
>>> hex(64)
'0x40'
>>> oct(64)
'0o100'
```

- `int` transforme en entier en enlevant la partie décimale tandis que `round` donne la valeur entière la plus proche,

3. des fonctions d'entrée-sortie[3] : `input, open, print`.

VI.B Des fonctions intégrées en module

Pour des usages spécifiques, de nouvelles fonctions qui ne sont pas disponibles à l'origine, peuvent être importées en faisant appel à des modules. Par exemple, le module `math` rajoute des fonctionnalités mathématiques (voir programme 19).

```
#importation des fonctions du module math
import math
a = math.sqrt(15)
c = math.cos(math.pi/4)
```

FIGURE 19 – Programme utilisant le module math

Un autre exemple est le module `random` qui permet d'effectuer des tirages aléatoires (voir programme 20).

Différentes manière d'importer Dans ces exemples on a importé le module simplement : pour utiliser les fonctions, il faut donc utiliser leur nom en le faisant précéder[a] par le nom du module (`math.cos` par exemple)[b]. On peut importer les fonctions explicitement depuis le module (colonne de gauche de la figure 21).
On peut même importer toutes les fonctions du module (colonne de droite de la figure 21). On voit qu'alors on ne préfixe plus par le nom du module.

a. en le préfixant
b. C'est une notation commune avec la programmation objet. La fonction cos peut être vue comme une méthode appartenant à la classe/objet math. On retiendra donc la syntaxe assez générale *class.methode*.

VII Définir ses propres fonctions

VII.A Une première fonction

Les fonctions constituent le principal élément structurant d'un programme. Afin de s'initier à la déclaration de ses propres fonctions, commençons par le calcul d'une fonction mathématique simple :

$$f : x \mapsto x^2 + 1$$

Ce qui donne en Python le programme de la figure 22).

3. La fonction `open` sera étudiée plus tard.

abs	all	any	ascii	bin	bool	breakpoint
bytearray	bytes	callable	chr	@classmethod	compile	complex
delattr	dict	dir	divmod	enumerate	eval	exec
filter	float	format	frozenset	getattr	globals	hasattr
hash	help	hex	id	input	int	isinstance
issubclass	iter	len	list	locals	map	max
memoryview	min	next	object	oct	open	ord
pow	print	property	range	repr	reversed	round
set	setattr	slice	sorted	@staticmethod	str	sum
super	tuple	type	vars	zip	__import__	

TABLE 2 – Fonctions de base de Python. On a mis en rouge celles qui seront les plus utilisées.

```
1 import random
2 a = random.randint(1, 6)
3 # Renvoie un entier tiré au hasard dans l'intervalle [1; 6],
4 # un peu comme au jeu de dés :)
5 print(a)
```

FIGURE 20 – Programme utilisant le module random.

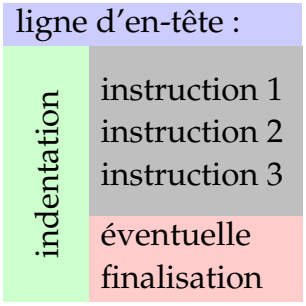
VII.B Indentation

Instruction composite

La définition[a] de la fonction est un exemple d' **instruction composite**. Comme sur le schéma ci-dessous, on constate que celle-ci se compose :

- d'une ligne d'en-tête commençant par un mot-clé et se terminant par deux points " :";
- un bloc-d'instruction **indenté**;
- une éventuelle finalisation.

a. ou déclaration



À retenir : La syntaxe de python est basée sur une **indentation** significative. Vous devez dans la suite y porter une grande attention.

Erreur de syntaxe Une erreur d'indentation produira une erreur et pourra arrêter l'exécution du programme (voir figure 23).

Conseil :

```
1 from math import sin, cos
2 print(cos(1) + sin(2))
```

```
1 from math import *
2 print(cos(1) + sin(2))
```

FIGURE 21 – Différentes façons d'importer.

```
def maFonction(x):
    return x**2+1
```

FIGURE 22 – Déclaration d'une fonction.

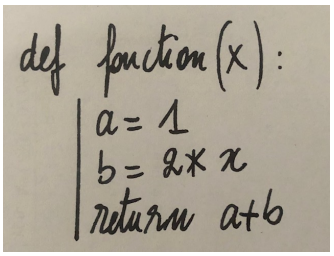
```
def maFonction(x):
return x**2+1
```

produira le message :

```
IndentationError: expected an indented block
```

FIGURE 23 – Erreur d'indentation

A l'écrit, il est conseillé de bien marquer l'indentation comme sur la photo ci-contre.



VII.C Appel de fonction

Le résultat de la fonction peut alors être évalué dans l'interpréteur (voir figure 24) ou par des appels dans l'éditeur, il conviendra d'utiliser la fonction `print` afin d'afficher le résultat (voir figure 25).

```
>>> maFonction(2)
5
>>> maFonction(5)
13.25
>>> a=2
>>> b=maFonction(a)
>>> b
5
```

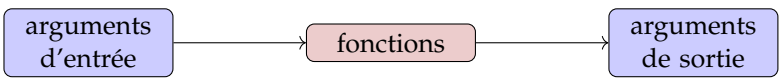
FIGURE 24 – Appel d'une fonction dans le shell.

On retiendra Une fonction est définie par :

1. une **introduction** déclarée par le mot-clef `def`,
2. un **corps de fonction** constitué d'un bloc d'**instructions indentées**,
3. une **finalisation** pouvant être initiée par le mot-clef `return`.

VII.D éléments de syntaxe

Vocabulaire Une fonction va agir sur des **arguments d'entrée** et éventuellement renvoyer des arguments de sortie.



```
a = maFonction(2)
print(a)
----------------------
# Réponse dans le shell
26
```

FIGURE 25 – Appel d'une fonction dans l'éditeur.

Syntaxe S'il y a plusieurs arguments d'entrée ou de sortie, il faut les séparer par des virgules.

Syntaxe Si la liste des arguments d'entrée est vide, il faut laisser les parenthèses après le nom de la fonction, pour ne pas confondre la fonction elle-même et sa valeur (voir figure 26).

```
def fonction():
    return("bonjour")
# dans le shell
>>> fonction()
'bonjour'
```

FIGURE 26 – Fonction sans argument.

Syntaxe Une fonction sans `return` ne renvoie rien (voir figure 26).

```
def fonction(x):
    print(2*x)
# dans le shell
>>> fonction(10)
20
>>> 2*fonction(10)
20
...
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

FIGURE 27 – Fonction ne renvoyant rien.

Remarque Bien qu'un résultat soit affiché par `print`, la fonction ne renvoie rien, c'est-à-dire qu'il n'y a pas d'arguments de sortie. Le résultat de la fonction n'est pas réutilisable dans un calcul ultérieur.

VII.E Variables locales et globales

Syntaxe Tout le code qui n'appartient pas à une fonction constitue le **programme principal**. Ensuite :

- Une variable née dans le programme principal s'appelle une variable **globale** ;
- Une variable née dans une fonction s'appelle une variable **locale**.

Exemple de variable locale Dans le script de la figure 28, `a` n'existe pas en dehors de la fonction.
Une variable locale meurt quand sa fonction se termine, de sorte qu'elle n'est pas accessible de l'extérieur. Si une fonction est appelée plusieurs fois, les variables locales sont créées et détruites à chaque fois.

Exemple de variable globale Dans le script de la figure 29, il y a deux variables `x`. La première ligne crée une variables globale `x` qui vaut 2 dans le programme principal. La variable `x` des lignes 2 et 3 est une variable locale. Dans la ligne

```
1  def f(x):
2      a = 2*x
3      return a
4  # dans le shell
5  >>> fonction(10)
6  20
7  >>> a
8  20
9  ...
10 NameError: name 'a' is not defined
```

FIGURE 28 – Exemple de variable locale.

3, on est dans le programme principal donc c'est la valeur global qui est appelée. Dans la ligne 4, lors de l'appel de la fonction, `x` vaut la valeur de l'argument de `f` soit 10.

```
1  x = 2
2  def f(x):
3      return 2*x
4  print(x)
5  print(f(10))
6  print(x)
7  #--------- affichage
8  2
9  20
10 2
```

FIGURE 29 – Exemple de variable globale.

VII.F La documentation d'une fonction pour l'usager

Afin de savoir quel est le rôle d'une fonction, on écrit une documentation de la fonction à l'intérieur de celle-ci : le docstring (voir figure 30).

```
1  def hypotenuse(a, b):
2      """Entrees : 2 nombres, les cotes d un triangle rectangle
3      Sortie : l hypotenuse du triangle """
4      c=sqrt(a**2 + b**2)
5      return c
```

FIGURE 30 – Fonction avec commentaire.

La documentation de la fonction s'écrit dans le corps de la fonction, juste après la première ligne de déclaration. La documentation commence par `"""` et se termine par `"""`. Cette documentation doit donc indiquer :

- la liste des paramètres attendus dans l'ordre,
- la ou les valeurs renvoyées s'il y en a.

Remarque Il est possible de faire appel à cette documentation dans une console en tapant `help(nom_fonction)` (voir figure 31).

```
>>> help(hypotenuse)
Help on function hypotenuse in module __main__:

hypotenuse(a, b)
    Entrees : 2 nombres, les cotes d un triangle rectangle
    Sortie : l hypotenuse du triangle
```

FIGURE 31 – Commentaire.

Remarque La documentation n'est pas obligatoire Python saura utiliser la fonction. Mais elle a pour rôle de permettre à un utilisateur de mettre en œuvre une fonction dont il ne connaît pas la structure interne, à ce titre elle est indispensable.

VII.G La documentation d'une fonction pour les programmeurs

Lorsque l'on écrit une fonction, il faut prévoir qu'elle sera relue, modifiée, corrigée. À ce moment, le lecteur devra comprendre les idées de l'auteur, c'est souvent très difficile. Il est donc recommandé de "commenter" la fonction en indiquant les étapes intermédiaires, les significations des variables, les astuces utilisées. Pour cela on peut écrire des phrases lisibles après le caractère `#`[4], tout ce qui suit sera ignoré par python mais sera lisible par celui ou celle qui lira le code. Voici, par exemple en figure 32, un code produit dans un TIPE, il n'y a malheureusement pas de docstring.

```
def premiersZeros(nb,n):
    h=0.1          # Pas pour la recherche
    epsilon=1e-10 # Précision pour les zéros,
                   # on peut la diminuer
    zeros=[]       # Liste pour recevoir les zéros
    a=h            # On commence à h, pas en 0,
                   # car jn(n,0)=0
    def f(x):      # On définit la fonction que l'on
        return jn(n,x) # utilise Bessel à l'ordre n
    for i in range(nb):  # On veut nb zéros
        while f(a)*f(a+h)>0: # On balaye à la recherche
            a =a+h           # d'un changement de signe
        z=dicho(f,a,a+h,epsilon) # On cherche la racine
        zeros.append(z)      # On l'ajoute à la liste
        a=a+h          # On part un cran plus loin
    return zeros
```

FIGURE 32 – Commentaires d'une fonction.

Plus la fonction est longue plus il sera nécessaire de la commenter, il arrivera fréquemment que les commentaires prennent plus de place que le code.

VII.H Exercice : le jeu des erreurs

On veut calculer la distance parcourue par un point matériel selon la loi horaire : $z = \frac{1}{2}g \cdot t^2$. Chacun des scripts des figures 33 à 38 sont réalisés dans l'éditeur puis compilé complètement ; il engendre une erreur ou des résultats surprenants. Prévoir les problèmes et les corriger.

Indications
Les messages d'erreur peuvent être instructifs.

1. NameError : global name 'g' is not defined
2. 20.0. La distance calculée est affichée. Mais elle n'est pas renvoyée, on ne peut rien en faire. Elle est de type **None**.
3. NameError : name 'chuteLibre' is not defined

4. croisillon ou *hash symbol* en anglais

```
def chuteLibre(t):
    z=0.5*g*t**2
    return z
print(chuteLibre(2))
```

FIGURE 33 – Script 1

```
def chuteLibre(t):
    g=10
    z=0.5*g*t**2
    print(z)
resultat=chuteLibre(2)
print(resultat)
```

FIGURE 34 – Script 2

On utilise la fonction avant sa définition.

4. 20.0
   NameError : name 'z' is not defined
   `z` n'existe plus en dehors de la fonction.
5. `5.0` : le temps est modifié dans la fonction, on a calculé `chuteLibre(1)`.
6. `IndentationError: unindent does not match any outer indentation level`. Une erreur parfois difficile à détecter

```
print(chuteLibre(2))

def chuteLibre(t):
    g=10
    z=0.5*g*t**2
    return z
```

FIGURE 35 – Script 3

```
def chuteLibre(t):
    g=10
    z=0.5*g*t**2
    return z
print(chuteLibre(2))
print(z)
```

FIGURE 36 – Script 4

```
def chuteLibre(t):
    g=10
    t=1
    z=0.5*g*t**2
    return z
    temps=2
print(chuteLibre(temps))
```

FIGURE 37 – Script 5

```
def chuteLibre(t):
    g=10
    z=0.5*g*t**2
return z
print(chuteLibre(2))
```

FIGURE 38 – Script 6