


Mise en route

I Introduction

I. 1 Notions de base

Python est un langage de programmation. Il a été créé en 1991 par Guido van Rossum un informaticien hollandais qui se décrit au sein de la communauté de Python comme un « Dictateur bienveillant à vie »¹

Python est grandement utilisé dans le monde scientifique et c'est un des langages informatiques les plus populaires. Voici un tableau qui récapitule les langages les plus utilisés dans le monde.

Mar 2023	Mar 2022	Change	Programming Language	Ratings	Change
1	1		 Python	14.83%	+0.57%
2	2		 C	14.73%	+1.67%
3	3		 Java	13.56%	+2.37%
4	4		 C++	13.29%	+4.64%
5	5		 C#	7.17%	+1.25%
6	6		 Visual Basic	4.75%	-1.01%
7	7		 JavaScript	2.17%	+0.09%
8	10	▲	 SQL	1.95%	+0.11%
9	8	▼	 PHP	1.61%	-0.30%
10	13	▲	 Go	1.24%	+0.26%

Ce langage peut-être utilisé pour des choses très différentes :

- Des mathématiques, c'est une calculatrice très très poussée...
- Gérer des bases de données, lire et modifier des fichiers.
- Manipuler des images, créer des algorithmes de reconnaissance d'images.
- Faire du « big data », manipuler beaucoup d'informations et les trier.
- Créer des sites web (c'est le cas par exemple d'Instagram ou de Netflix !)

Cette année on se concentrera sur les trois premiers points.

I. 2 Utiliser Python

Python est un langage de programmation, on écrit des phrases en Python et l'ordinateur doit ensuite comprendre ce qui est dit. Pour cela il faut un **interpréteur**, un logiciel qui permet de passer de la syntaxe Python au langage de l'ordinateur (C, assembleur et au final des 0 et des 1).

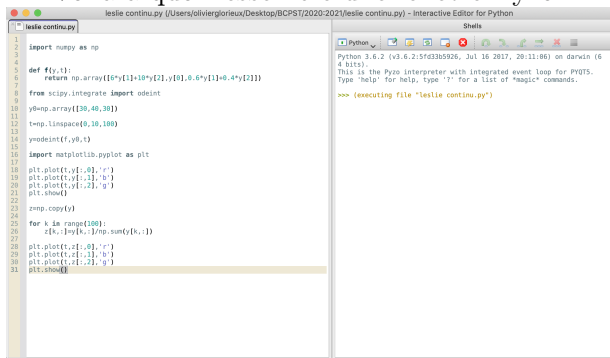
Il existe beaucoup d'interpréteur Python. Celui qui vous est conseillé cette année est le logiciel **Pyzo**. Vous pouvez le télécharger <https://pyzo.org/start.html>. Si vous voulez progresser en Python durant l'année je vous conseille vivement de l'installer sur un ordinateur qui vous est accessible.

Si vous ne pouvez pas l'installer sur un ordinateur, il vous est possible de faire des programmes Python directement sur internet grâce à Capytale. Pour cela, il faut :

1. aller sur votre espace ENT ;
2. aller dans « mes applis » ;
3. ouvrir « Capytale 2 » ;
4. cliquer sur « Accédez à vos activités » ;
5. cliquer sur « Créer une nouvelle activité » ;
6. sélectionner « Script-Console ».

1. Benevolent Dictator for Life

Voilà à quoi ressemble une fenêtre Pyzo :



La partie de gauche est appelée **éditeur** ce qui y est écrit est en général appelé **script**. La partie de droite est appelée l'interpréteur ou le **Shell**, c'est là où s'affiche ce qu'on a demandé à l'ordinateur, c'est aussi là où s'affiche les erreurs dans la syntaxe...

Le symbole `>>>` indique l'endroit où il faut écrire des instructions dans le shell.

I. 3 Opérations de base

Les fonctions suivantes permettent d'utiliser python comme une calculatrice

Opération mathématique	Instruction Python
Addition	<code>+</code>
Soustraction	<code>-</code>
Multiplication	<code>*</code>
Puissance	<code>**</code> (et non <code>^</code> !)
Division	<code>/</code>
Quotient de la division entière	<code>//</code>
Reste de la division entière	<code>%</code>

Que se passe-t-il dans l'ordre des opérations? En règle général, l'ordre mathématiques prime... Si on n'est pas sûr, et que l'on fait plusieurs opérations à la suite, il vaut mieux mettre des parenthèses pour éviter les erreurs d'ordres.

Si on veut écrire $((3+4) \times 2)^3$ mieux vaut écrire

```
1 >>> ((3+4)*2)**3
2 2744
```

Pour éviter de le confondre avec $(3 + 4) \times 2^3$

```
1 >>> (3+4)*(2**3)
2 56
```

I. 4 Opération plus avancée

Pour pouvoir effectuer des opérations plus complexes, on peut importer des modules. Pour commencer le module `math` contient une grande partie des fonctions et constantes que l'on utilise couramment en maths.

On peut les importer avec la syntaxe `« from math import ... »` :

Sur les lignes 2 à 4, l'interpréteur nous informe que `pi` n'est pas défini. À la ligne 5, on charge le nombre π ce qui nous permet de l'utiliser. On fait ensuite de même avec la fonction `cos`.

```
1 >>> pi
2 Traceback (most recent call last):
3   File "<console>", line 1, in <module>
4 NameError: name 'pi' is not defined
5 >>> from math import pi
6 >>> pi
7 3.141592653589793
8 >>> from math import cos
9 >>> cos(pi)
10 -1.0
```

II Variables Python

II. 1 Affectation

Afin de conserver des valeurs en mémoire, on utilise des **variable**. Les noms des variables peuvent être très simple : `x` ou `y`, ou beaucoup plus long `Ma_variable_preferee`, le mieux est de lui donner un nom qui reflète ce que c'est : `age` , `volume_total`, `lancer_de_des...` selon le contexte de ce que l'on cherche à faire.

Voilà les règles à suivre pour donner un nom aux variables :

- Un nom de variable doit commencer par une lettre.
- Un nom de variable ne peut contenir que des caractères alpha-numériques (c'est-à-dire des lettres ou des nombres) et des soulignés `_`
- Les noms de variables sont sensibles à la casse : les majuscules et les minuscules. Les variables `Age`, `age`, et `AGE` seront trois variables différentes.

Pour donner une valeur à une variable, on dira que l'on lui **affecte** une valeur, il suffit d'utiliser le symbole `"="`.

On peut définir plusieurs variables en même temps en les écrivant les unes à la suite des autres espacées par des virgules.

Quand une valeur existe déjà, le symbole `"="` écrase l'ancienne variable et donne la nouvelle valeur à cette variable. C'est le cas de la variable `x` dans l'exemple ci-contre. On lui donne la valeur 2 à la ligne 5, puis la valeur 5 à la ligne 8. Donc lorsqu'on l'affiche à la ligne 9 elle vaut 5.

```
1 >>> x
2 Traceback (most recent call last):
3   File "<console>", line 1, in <module>
4 NameError: name 'x' is not defined
5 >>> x = 2
6 >>> x
7 2
8 >>> x, y, z = 5, 6, 7
9 >>> x
10 5
11 >>> y, z
12 (6, 7)
```

Attention ! Il ne faut pas voir le symbole `"="` comme une égalité, en Python contrairement au math, il n'est pas symétrique, c'est-à-dire que `'x=y'` ne revient pas au même que `'y=x'`. Dans le premier cas on affecte à la variable `x` la valeur de `y` et dans le second on affecte à la variable `y` la valeur de `x`...

À méditer.

Enfin, il est possible d'affecter autre chose que des nombres à des variables.

On peut par exemple lui donner pour valeur des **chaines de caractères**, c'est-à-dire une suite de symboles entre guillemet.

```
1 >>> x=2
2 >>> y=3
3 >>> x=y
4 >>> x
5 3
6 >>> y
7 3
```

```
1 >>> x=2
2 >>> y=3
3 >>> y=x
4 >>> x
5 2
6 >>> y
7 2
```

```
1 >>> x = "Bonjour !"
2 >>> y = "Au-revoir"
3 >>> x, y
4 ('Bonjour !', 'Au-revoir')
```

II. 2 Opérations sur les variables

Nous nous concentrerons pour cette partie sur les opérations sur les nombres. Nous verrons plus tard les opérations sur les chaines de caractères.

Les opérations que nous avons vu plus haut peuvent directement être appliquée sur des variables.

```
1 >>> a=2
2 >>> b=3*a+4
3 >>> b
4 10
```

Par ailleurs, les variables ne sont pas réactualisées lors des opérations. Dans l'exemple suivant modifier `b` ne modifie pas `c`.

```
1 >>> a=2
2 >>> b=3
3 >>> c=a+b
4 >>> b=4
5 >>> c
6 5
```

III Fonctions

Le problème dans le fait de n'écrire que des instructions dans le shell est qu'il n'est pas facile de modifier des choses que l'on a déjà écrites. Pour cela, on écrit plutôt dans l'éditeur (la fenêtre de gauche).

Cependant, lorsque l'on exécute un des scripts que nous avons écrit précédemment, rien ne s'affiche. Si l'on souhaite que quelque chose s'affiche, il faut utiliser la commande `print`.

On peut par exemple taper l'expression suivante dans l'éditeur pour obtenir le résultat de `2**3+2+1`

```
1 x = 2
2 print(x**3+x+1)
```

L'inconvénient de cette méthode est qu'on ne peut pas récupérer de manière automatique la valeur de `x**3+x+1`. l'interpréteur l'affiche mais il n'est pas gardé en mémoire.

Nous allons donc utiliser les fonctions pour automatiser des opérations.

III. 1 Syntaxe

Une fonction reçoit toujours


- 0, 1 ou plusieurs paramètres d'entrée, qui correspondent aux données;
- 0, 1 ou plusieurs paramètres de sortie, qui correspondent aux différents résultats fournis par la fonction.

```
def ma_fonction(x1, x2, ..., xn) :
    instructions1
    instructions2
    ...
    instructionsq
    return(y1, y2, ..., yp)
```

Explications

La syntaxe ci-dessus permet de créer une fonction

- dont le nom est `ma_fonction`
- qui prend en variables d'entrée `x1, x2, ..., xn`
- et qui donne en sortie `y1,y2, ...,yp`.

Remarques. •  Le nom (ici `ma_fonction`) que l'on donne à la fonction doit suivre les mêmes règles que les noms de variables.

- La première ligne correspond à la définition de la fonction : elle définit le nom et les paramètres d'entrée.

Enfin la suite des lignes constitue le corps /coeur de la fonction. Elles contiennent les instructions effectuées, ainsi qu'éventuellement une ou plusieurs lignes `return` qui renvoient le ou les résultats de la fonction.

- Il ne faut pas oublier les ":" et pour arrêter la définition de la fonction, il arrêter l'indentation.
- Au moment de la lecture, lorsque Python rencontre un `return`, il renvoie ce que vous lui demandez et il s'arrête.

La fonction `f` ci-contre prend en argument un `x` et renvoie la valeur $\sqrt{x^2 + 6}$. Pour calculer la racine carrée nous allons importer la fonction `sqrt` de la bibliothèque `math`

```
1 from math import sqrt
2
3 def f(x) :
4     return sqrt(x**2+6)
```

Exercice 1. Écrire une fonction `Energie_cinetique` qui prend en argument la masse m d'une particule et sa vitesse v et qui renvoie l'énergie cinétique de cette particule $\frac{1}{2}mv^2$.

Réponse :

III. 2 Appel d'une fonction

Lors de la définition d'une fonction, Python ne fait aucun calcul. Il "apprend" seulement à faire les calculs, afin de pouvoir effectuer les opérations sur des valeurs fixées ensuite. Ainsi, pour utiliser les fonctions définies, il faut les exécuter. Pour cela, on appelle la fonction avec des valeurs données :

```
ma_fonction(val1, val2, ..., valn)
```

Explications

En exécutant cette instruction, Python

- appelle la fonction `ma_fonction` créée auparavant,
- crée les variables `x1`, `x2`, ..., `xn` en leur donnant les valeurs `val1`, `val2`, ..., `valn`,
- exécute les instructions pour calculer `y1,y2,..., yp`,
- renvoie les valeurs de `y1,y2,..., yp`.

Remarque.  L'ordre des paramètres d'entrée est important. Par exemple :

`Energie_cinetique(1,2) ≠ Energie_cinetique(2,1)`.

III. 3 Erreurs classiques

Oubli des deux points :

```
1 # Erreur 1 : oubli des deux points en fin de ligne
2 def f(x)
3     return x**2-3*x
4
5 SyntaxError: invalid syntax
```

Oubli de l'indentation :

```
1 # Erreur 2 : non respect de l'indentation
2 def f(x):
3 return x**2-3*x
4
5 File "<stdin>", line 2
6     return x**2-3*x
7 IndentationError:
```

Oubli du mot-clef return et confusion print/return :

```

1 # Erreur 3 : oubli du mot-cle return
2 def f(x):
3     x**2-3*x
4 # dans ce cas, saisir f(2),f(3),f(4) dans le
5 # shell donne (None, None, None)

```

Que s'est-il passé dans la dernière tentative de définition de la fonction `f` ? Tout simplement, au cours de l'exécution de la fonction, l'expression

$$x**2-3*x$$

est calculée, mais l'interpréteur, n'ayant pas reçu l'instruction de renvoyer le résultat, le garde pour lui : il reste muet et se contente de renvoyer comme valeur l'objet `None`.

- On pourrait être tenté de remplacer l'instruction `return` par la fonction `print` que nous avons déjà utilisée :

```

1 def f(x):
2     print(x**2-3*x)

```

Dans ce cas, dans le shell, on obtient :

```

1 >>> f(2),f(3),f(4) # on obtient un affichage
2 -2
3 0
4 4
5 >>> f(2)+f(3)+f(4) # mais on ajoute du vide !!!
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 TypeError: unsupported operand type(s) for +: 'NoneType'
9 and 'NoneType'

```

En faisant ainsi, l'interpréteur affiche en effet la valeur de l'expression

$$x**2-3*x$$

à chaque fois que la fonction est appelée, mais ensuite, il ne renvoie aucun objet réutilisable ultérieurement.

IV Structures conditionnelles

Dans cette section, on présente les différentes commandes permettant de programmer sous Python des instructions conditionnelles, c'est-à-dire des instructions qui ne sont effectuées que si une certaine condition est remplie. On s'intéresse ainsi à des opérations du type :

Si un test est vrai alors faire :
 la suite d'instructions 1
Sinon faire :
 la suite d'instructions 2

IV. 1 Écriture de tests

IV. 1. a Opérateurs de comparaison et tests élémentaires

Lors d'un test, on évalue si une affirmation est vraie ou fausse. On utilise donc le type `bool` défini au chapitre précédent, qui peut prendre uniquement deux valeurs : « `True` » (vrai) ou « `False` » (faux). Pour construire des tests, on compare généralement des variables numériques : on teste par exemple s'il y a égalité ou non entre deux variables numériques, si l'une est supérieure à une autre, etc ...

On rappelle les 6 opérateurs de comparaison qui permettent d'effectuer des tests :

Relation mathématique	Instruction Python
Égal	<code>==</code>
Différent	<code>!=</code>
Inférieur strict	<code><</code>
Supérieur strict	<code>></code>
Inférieur ou égal	<code><=</code>
Supérieur ou égal	<code>>=</code>

```

1 >>> x=3
2 >>> x>2
3 True
4 >>> x<=3
5 True
6 >>> x==4
7 False
8 >>> x!=0
9 True

```

Remarque. ⚠ Ne pas confondre le symbole = qui permet de définir une variable avec le symbole == qui teste l'égalité.

Dans l'exemple précédent, à la ligne 1, on affecte à x la valeur 3 et à la ligne 6 on test si x est égal à 4 (ce qui est faux).

IV. 1. b Opérateurs logiques

Nous venons de donner la définition de tests élémentaires. Il est parfois pratique d'utiliser des tests plus élaborés. On combine alors les opérateurs de comparaison avec les trois opérateurs de logique suivants :

Relation logique	Instruction Python
Et	<code>and</code>
Ou	<code>or</code>
Négation	<code>not</code>

```

1 >>> x = 1.5
2 >>> (x > 1) and (x < 2)
3 True
4 >>> (x > 4) or (x <= 3)
5 True
6 >>> not x != 0
7 False

```

Remarques. • Il est vivement conseillé d'écrire chaque test élémentaire entre parenthèses.

- Le dernier test peut-être réécrit `x == 0`.
- ⚠ Mieux vaut éviter les doubles inégalités !
Ainsi pour le test $0 \leq x \leq 1$, il vaut mieux écrire : `(0<=x) and (x<=1)`

IV. 2 Les schémas conditionnels

Un schéma est dit conditionnel si, suivant le résultat d'un test, l'ordinateur doit effectuer telle suite d'instructions ou telle autre. Il existe essentiellement quatre variantes à connaître.

IV. 2. a Version courte : Si ... Alors ...

```

SYNTAXE :
if test :
    instruction 1
    instruction 2
    instruction 3

```

Explications

Les instructions ne sont exécutées que si le test est vrai : lorsque Python arrive à `if`, il évalue le test :

- ★ Si le test est vérifié, c'est-à-dire si le booléen a pour valeur True alors les instructions sont effectués.
- ★ Sinon, il ne se passe rien ! Pyzo ne regarde que les lignes après l'indentation finie.

Remarque. • Ne pas oublier les « : », ils indiquent que le test est terminé.

- Ne pas oublier l'indentation, c'est-à-dire les espaces avant les instructions. C'est cela qui indique que l'on est toujours dans le bloc `if`. Lorsque l'on veut en sortir, il suffit de ne plus indenter.

Dans l'exemple ci-contre, on définit une fonction qui prend en argument un entier n et qui renvoie la chaîne de caractère "Ce nombre est pair" si ce nombre est pair. À la ligne 2, on test si le reste de la division euclidienne de n par 2 est égal à 0 pour déterminer si le nombre est pair.

```

1 def pair(n) :
2     if n % 2 == 0 :
3         return "Ce nombre est pair"

```

Exercice 2. Écrire une fonction qui prend en argument un nombre `n`, et renvoie la chaîne de caractère « gagné! » si ce nombre est égal à 0.

Réponse :

IV. 2. b Version classique : Si ... Alors ... Sinon ...

SYNTAXE :

```
if test:
    instructions 1
else:
    instructions 2
```

Explications

Lorsque Python arrive à `if`, il évalue le test :

- ★ Si le test est vrai : l'instruction 1 est effectuée
- ★ Sinon, le test est alors faux et l'instruction 2 est effectuée

Remarque. • On peut remarquer que dans tous les cas, seul l'un des deux blocs d'instructions est réalisé.

- PAS DE TEST après le `else`.
- L'indentation du `else` doit être la même que l'indentation du `if`.
- Le `else` doit être suivi de « : ».

Exercice 3. Le billet d'avion pour faire Paris-Londres est à 100 euros. Il y a en ce moment une réduction de 50% pour les moins de 25 ans.

Écrire une fonction `reduction` qui prend en argument l'âge `a` de l'utilisateur et qui renvoie le prix de son billet.

Réponse :

Il est parfois nécessaire d'effectuer plusieurs tests les uns après les autres.

SYNTAXE :

```
if test 1:  
    instructions 1  
elif test 2:  
    instructions 2  
else:  
    instructions 3
```

Explications

- ★ Python examine d'abord le test 1 :
- ★ s'il est vrai, alors l'instruction 1 est effectuée.
- ★ s'il est faux, Python examine le test 2 :
s'il est vrai, l'instruction 2 est effectuée
- ★ si aucun des deux tests n'est vrai, Python exécute la série d'instructions 3 situées après le **else**.

Remarque. Python n'examine un test que si les tests précédents sont faux. Bien vérifier que les tests ne s'intersectent pas.

Exemple : test 1 : $a \geq 0$, test 2 : $a = 0$. Python n'examinera jamais le test 2 puisque le test 1 est vrai avant.

Exercice 4. Une place plein tarif de billet de train est à 100 euros. Si le voyageur a moins de 4 ans son billet est gratuit. S'il a entre 5 et 25 ans et qu'il possède une carte SNCF, il bénéficie d'une réduction de 60%. S'il a plus de 60 ans et qu'il possède une carte SNCF, il bénéficie d'une réduction de 50%. Sinon, il paye le plein tarif.

Écrire un programme qui demande l'âge de l'utilisateur et affiche le prix de son billet.

Réponse :

IV. 2. d Version à tests emboîtés

Il s'agit ici de faire des tests emboîtés : à l'intérieur d'un test, on fait un autre test.

SYNTAXE :

```
if test 1:
    if test 1.1:
        instructions 1.1
    else:
        instructions 1.2
else:
    if test 2.1:
        instructions 2.1
    else:
        instructions 2.2
```

Explications

Python examine le test 1 :

- ★ s'il est vrai : alors on vérifie le test1.1
- ★ s'il est faux : alors on vérifie le test2.1

Remarque. • Bien faire attention à l'indentation pour séparer les différents blocs.

- On peut bien évidemment combiner comme on le souhaite tous les types d'instructions conditionnelles vus avant.

Exercice 5. Écrire une fonction `affine` qui prend en argument deux nombre `a` et `b` et renvoie les solutions de $ax + b = 0$.

Réponse :

V Les boucles

Souvent, dans un programme, certains groupes d'instructions doivent être exécutés plusieurs fois, on dit qu'on les itère. On utilise alors, afin de ne pas avoir à répéter la même instruction 10, 20, 100... fois, la notion de boucle.

Il existe deux grandes catégories de boucles :

- les boucles “for” : on connaît à l'avance le nombre d'itérations à effectuer.
Exemple : Calculer la somme des entiers de 1 à 1000
- les boucles “while” : on ne connaît pas à l'avance le nombre d'itérations à effectuer. Les itérations s'arrêtent lorsqu'une certaine condition est satisfaite.
Exemple : générer des nombres aléatoires entre 1 et 6 jusqu'à tomber sur 6.

V. 1 Boucles FOR

V. 1. a Instruction range

Cette instruction décrit une liste d'entiers avec 3 principales syntaxes.

- `range(n)` : une séquence avec tous les entiers de 0 à n-1 ⚠
- `range(d, n)` : une séquence avec tous les entiers de d à n-1 ⚠
- `range(d, n, p)` : une séquence avec tous les entiers de d à n-p par pas de p.
⚠ attention aux p négatifs, c'est parfois assez contre-intuitif.

Exemple. Que renvoient les instructions suivantes :

- `range(5)` :, `range(-1)` :, `range(0)` :
- `range(3,8)` :, `range(-5,-1)` :, `range(4,1)` :
- `range(0,8,2)` :, `range(5,-1,-1)` :, `range(1,10,-3)` :

V. 1. b Boucle for

SYNTAXE :

```
for k in sequence:  
    instructions_dependant_de_k
```

Explications

- **sequence** : correspond par exemple à une liste d'entiers fournie par la fonction `range`, mais peut être aussi une chaîne de caractères, ou d'autres choses plus compliquées ...
- Toujours mettre des “ : ” après la séquence !
- Penser à indenter les instructions de la boucle. Pour sortir de la boucle, il suffit de ne plus indenter.

Pour calculer la somme des entiers de 1 à n . On introduit S qui va jouer le rôle de la somme à la ligne 1 en l'initialisant à 0. À la ligne 2, on fait une boucle for pour que k parcourt l'entier de 1 à n . Enfin à la ligne 3, on ajoute à chaque itération k à la valeur de S . La dernière ligne permet d'afficher la valeur de S .

```
1 S = 0  
2 for k in range(1, n+1) :  
3     S = S+k  
4 print(S)
```

L'exemple précédent est symptomatique de ce qu'il faudra souvent faire en général pour calculer les termes d'une suite, d'une somme ou d'un produit. Pour cela, il faut :

1. introduire une variable qui jouera le rôle du terme que l'on veut calculer en l'initialisant à la bonne valeur ;
2. déterminer la valeur de début et la valeur de fin à mettre dans le `range` ;
3. effectuer dans la boucle l'opération qui permet de passer d'un terme à son successeur.

Exercice 6. Écrire un script qui calcule $n!$.

Réponse :

Exercice 7. Écrire un programme qui calcule le terme u_{10} des suites définies par :

- $\begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N}, u_{n+1} = \frac{1}{2}u_n + 1 \end{cases}$
- $\begin{cases} v_0 = 2 \\ \forall n \in \mathbb{N}, v_{n+1} = \sqrt{1 + v_n^2} \end{cases}$
- $\begin{cases} w_1 = 2 \\ \forall n \in \mathbb{N}^*, w_{n+1} = \sqrt{n}(\sin(w_n))^2 \end{cases}$
- $\begin{cases} t_1 = 2 \\ \forall n \in \mathbb{N}^*, t_{n+1} = \ln\left(t_n + \frac{1}{n}\right) \end{cases}$

Chaque fonction utilisée dans la définition de ces suites sont dans la bibliothèque `math`. Attention pour la fonction `ln`, il faut utiliser la fonction `log` de la bibliothèque `math`.

Réponse :

V. 2 Boucles WHILE

V. 2. a Syntaxe

```
SYNTAXE :  
while test:  
    instructions
```

Explications

Python commence par évaluer le test.

- ★ S'il n'est pas vérifié, alors on sort de la boucle.
- ★ S'il est vérifié, Python effectue les instructions puis refait le test.
 - S'il n'est plus vérifié, alors on sort de la boucle.
 - S'il est toujours vérifié, Python effectue les instructions puis refait le test.

Ainsi de suite : tant que le test est vérifié, Python effectue les instructions.

Remarque. • Toujours mettre des « : » après la séquence !

- Penser à indenter les instructions de la boucle. Pour sortir de la boucle, il suffit de ne plus indenter.

Le programme ci-contre permet d'afficher la plus petite puissance de 2 supérieur ou égale à 1000000. On commence par initialiser à 1 la variable qui variable P qui va jouer le rôle de la puissance de 2. Les lignes 2 et 3 signifie que tant que P est strictement inférieur à 1000000, il faut multiplier P par 2. La dernière ligne permet d'afficher la valeur de P

```
1 P = 1  
2 while P < 1000000 :  
3     P = 2*P  
4 print(P)
```

V. 2. b Compteur

Il n'y a pas de compteur de boucle dans une boucle while. On aura souvent intérêt à en introduire un afin de savoir combien de fois les instructions ont été réalisées. Pour cela on crée une variable n que l'on initialise à 0, et que l'on incrémente de 1 à chaque fois que l'on effectue la boucle.

Exercice 8. Réécrire le programme précédent en ajoutant un compteur pour qu'il affiche la plus petite valeur $n \in \mathbb{N}$ telle que $2^n \geq 1000000$.

Réponse :

V. 2. c Test d'exécution et test d'arrêt

Attention de ne pas confondre :


- le test d'exécution, à mettre dans la boucle while, qui se lit "*Tant que le test est vrai, les instructions sont exécutées*"
- le test d'arrêt donné dans l'énoncé, qui se lit "*Effectuer les instructions jusqu'à obtenir une condition*".

Exercice 9. Écrire un script qui affiche le plus petit n tel que la somme $s_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$ soit supérieure stricte à 15.

Le test d'arrêt est :

Le test d'exécution est :

Réponse :

Remarque.  Une erreur fréquente est d'avoir une condition toujours satisfaite, auquel cas les itérations ne s'arrêteront jamais. Les instructions après le `while` doivent être telles qu'à un moment donné, le test deviendra faux. Sinon, pour stopper l'exécution d'un programme en cours, il suffit d'appuyer sur les touches `Ctrl+I`.

Exemple de boucle infinie :

```
1 i=0
2 while i==0 :
3     print("AU SECOURS")
```