La **programmation**, appelée aussi **codage**, est l'écriture de **programmes informatiques**, c'est-à-dire la description d'un algorithme dans un langage de programmation compréhensible par une machine et par un humain.

Un **langage de programmation** est un moyen de communication avec l'ordinateur mais aussi entre programmeurs ; les programmes étant d'ordinaire écrits, lus et modifiés par des équipes de programmeurs.

Il existe des centaines de langages de programmation[1], par exemple C, C++, JavaScript, Java, PHP, etc. Le langage utilisé dans ce chapitre est Python.

Python[2] est un langage informatique inventé par Guido Van Rossum. La première version publique date de 1991. Il est multiplateforme (Linux, MacOS, Windows, android, iOS), libre et gratuit, mis à jour régulièrement (version actuelle 3.11[3]).

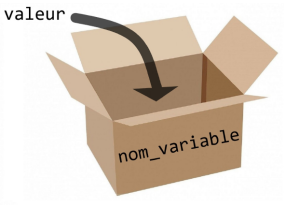
1. Variables et affectation

1.1. Variables

L'informatique désigne le traitement automatique de l'information[4]. Dans les programmes informatiques, l'information est représentée par des données.

Cours

Les programmes informatiques manipulent des données en utilisant des **variables**. Une variable permet d'associer un nom à une valeur.



Une variable informatique peut se concevoir comme une sorte de “boite” étiquetée avec un nom, dans laquelle un programme enregistre une valeur pour la consulter ou la modifier pendant son exécution[5].

En Python, comme dans la plupart des langages informatiques, le nom d'une variable :

- s'écrit en lettres minuscules (« `a` » à « `z` ») et majuscules (« `A` » à « `Z` ») et peut contenir des chiffres (« `0` » à « `9` ») et le caractère blanc souligné (« `_` ») ;
- ne doit pas comporter d'espace, de signes d'opération « `+` », « `-` », « `*` » ou « `/` », ni de caractères spéciaux comme des signes de ponctuation « `'` », « `"` », « `,` », « `.` », « `:` », « `@` », etc. ;
- ne doit pas commencer par un chiffre ;
- ne doit pas être un mot réservé de Python, par exemple « `for` », « `if` », « `print` », etc. ; et
- est sensible à la casse, ce qui signifie que les variables « `TesT` », « `test` » ou « `TEST` » sont différentes.

En pratique cela permet d'éviter les noms de variable réduits à une lettre et d'utiliser des noms qui ont un sens !

PEP 8

La PEP 8 [6] donne un grand nombre de recommandations de style pour écrire du code Python agréable à lire et recommande en particulier de nommer les variables par des mots en minuscule séparés par des blancs soulignés (`_`) [7], par exemple d'appeler une variable `somme_des_nombres` plutôt que `s` dans un programme qui additionne des nombres.

1.2. Types de variable

Cours

Les variables peuvent être de **types** différents en fonction des données qu'elles représentent.

Les principaux types de variable sont :

- les nombres entiers (type `int`) ;
- les nombres décimaux, appelés « flottants » (type `float`) qui s'écrivent toujours avec un point (⚠️ le séparateur décimal est un point, **pas une virgule**), par exemple `5.0` .

  Noter que `.5` et `5.` permettent d'écrire rapidement les flottants `0.5` et `5.0` et que `2e5` ou `2E5` (pour 2×10^5) permettent d'écrire le nombre flottant `200000.0` ;
- les booléens prenant seulement les valeurs `True` ou `False` (type `bool`) ;
- les textes ou chaines des caractères (type `str`) écrits entre une paire de guillemets (`"`) ou d'apostrophes (`'`) ;
- d'autres types dits "construits" comme les p_uplets, tableaux, dictionnaires[8], etc.

1.3. Affectation

Cours

L'**affectation** consiste à donner une valeur à une variable. En Python, comme dans la plupart des langages informatiques, l'**affectation d'une valeur à une variable** est représentée par le signe « `=` ».[9]

Par exemple, saisir les commandes suivantes dans la console Python permet d'affecter les valeurs 3 (type `int`), 3.0 (type `float`) et "3" (type `str`) à des variables nommées respectivement `a`, `b` et `c` :

La console Python, ou interpréteur Python, est un moyen rapide d'exécuter des commandes. Il suffit de taper une instruction en réponse à l'invite `>>>` puis d'appuyer sur la touche « Entrée » pour lancer son exécution.

PEP 8

Mettre des espaces autour d'un égal (`=`).

```
>>> a = 3
>>> b = 3.0
>>> c = "3"
```

En Python, c'est l'affectation qui définit le type d'une variable[10].

⚠️ C'est bien la valeur qui se trouve à droite du signe « `=` » qui est affectée à la variable à gauche, et pas dans l'autre sens.

```
>>> 3 = a
  File "<interactive input>", line 1
SyntaxError: can't assign to literal
```

PEP 8

Mettre un espace après une virgule (`,`) , mais pas avant.

Il est aussi possible d'affecter des valeurs à plusieurs variables en même temps en une seule ligne.

```
>>> a, b = 3, 4
>>> a
3
>>> b
4
```

et d'affecter la valeur d'une variable à une autre variable, par exemple :

Quand on tape le nom d'une variable dans la console, elle affiche sa valeur.

```
>>> a = 3
>>> b = a
>>> b
3
```

⚠️ Il n'est pas possible d'utiliser une variable avant de lui avoir affecté une valeur.

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

Exercice corrigé

On affecte les valeurs 5 et 6 (de type `int`) à deux variables nommées respectivement `a` et `b` :

```
>>> a = 5
>>> b = 6
```

Compléter ces instructions pour échanger les valeurs de `a` et de `b` (sans utiliser les chiffres 5, 6 ou tout autre chiffre).

Réponse

1. Voyons d'abord la solution qui ne fonctionne pas :

```
>>> a = b
>>> b = a
>>> a
6
>>> b
6
```

Cette solution ne fonctionne pas car la valeur intiale de `a`, c'est-à-dire 5, est perdue quand on écrit `a = b`, on dit qu'elle est "écrasée" par la valeur de `b`. Ensuite quand on écrit `b = a` on affecte la nouvelle valeur de `a`, c'est-à-dire 6 au lieu de 5, à `b`.

2. Une première solution consiste donc à utiliser une autre variable pour conserver la valeur intiale de `a` temporairement, appelons la `temp` :

```
>>> temp = a
```

Et ensuite de faire l'échange :

```
>>> a = b
>>> b = temp
>>> a
6
>>> b
5
```

Notons que la dernière instruction affecte la valeur de `temp`, c'est-à-dire 5, à `b`, et pas la valeur de `a` puisqu'elle vaut 6 à ce moment là.

3. Une deuxième solution plus élégante consiste à utiliser l'affectation de plusieurs variables sur une seule ligne :

```
>>> a, b = b, a
>>> a
6
>>> b
5
```

4. Pour information, il existe une troisième solution un peu plus compliquée, qui n'utilise ni variable temporaire, ni affectation de plusieurs variables en une seule ligne :

```
>>> a = a + b
>>> b = a - b
>>> a = a - b
>>> a
6
>>> b
5
```

2. Opérations, comparaisons, expression

2.1. Opérateurs arithmétiques sur les nombres

Les opérations arithmétiques usuelles sont effectuées sur des nombres de types `int` ou `float` :

opérateur	notation
addition	`a + b`
soustraction	`a - b`
multiplication	`a * b`
puissance	`a**b`
divisions décimale	`a / b`

```
>>> a = 5
>>> b = 2
```

PEP 8

Entourer les opérateurs mathématiques (`+`, `-`, `/`, `*`) d'un espace avant et d'un espace après.

```
>>> a + b
7
>>> a / b
2.5
>>> a**b
25
```

À noter :

> Si `a` et `b` sont deux variables toutes les deux de type `int` alors le résultat d'une opération entre les deux est de type `int`, sauf pour la division qui est toujours de type `float` même si le résultat est un entier :
>
> ```
> >>> 10 / 5
> 2.0
> ```
>
> et si l'un de `a` ou de `b` est de type `float` alors le résultat est toujours de type `float`.
>
> La racine carrée d'un nombre peut s'obtenir avec : `a**0.5` [11].
>
> L'ordre des priorités mathématiques est respecté.

Il est possible d'affecter une valeur à une variable qui dépend de son ancienne valeur, par exemple l'augmenter d'une quantité donnée (on dit **incrémenter**)[12].

```
>>> a = 3
>>> a = a + 1
>>> a
4
```

PEP 8

Dans ce cas particuliers, on peut omettre les espaces autour de la multiplication (`*`) pour montrer la priorité sur l'addition et améliorer la lisibilité de la formule.

```
>>> a = 2*a + 1
>>> a
9
```

[12]

Des raccourcis d'écriture existent pour aller plus vite (mais attention aux erreurs en les utilisant !).

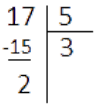
- `a += 1` signifie `a = a + 1` ;
- `a += b` signifie `a = a + b` ; et
- `a *= 2` signifie `a = a * 2`.

2.1.1. Division entière (ou division euclidienne)

L'opérateur de division entière `//` et l'opération modulo `%` utilisés avec des entiers (de type `int`) donnent respectivement le quotient et le reste d'une division euclidienne : si `a` et `b` sont des entiers tels que $a = b \times q + r$, alors `a // b` donne q et `a % b` donne r[13].

opérateur	notation
quotient	`a // b`
reste	`a % b`

Par exemple, le quotient et le reste de la division entière de 17 par 5 sont 3 et 2 respectivement (car $17 = 3 \times 5 + 2$) :




```
>>> a = 17
>>> b = 5
>>> a // b
3
>>> a % b
2
```

L'opérateur modulo, `%`, qui donne le reste d'une division entière, est très utile pour déterminer si un nombre est divisible par un autre nombre, dans ce cas le reste est égal à zéro :

```
>>> 10 % 5
0
>>> 10 % 3
1
```

10 est divisible par 5 mais pas par 3.

2.2. Opérateurs sur les chaines de caractères

Les textes ou chaines des caractères, de type `str` (abréviation de *string*), sont définis entre une paire de guillemets (`"`) ou d'apostrophes (`'`)[14].

```
>>> chaine1 = 'Hello '
>>> chaine2 = "world"
```

Pour les chaînes de caractères, deux opérations sont possibles, l'addition et la multiplication[15] :

- L'opérateur d'addition « `+` » **concatène** (assemble) deux chaînes de caractères.

  ```
  >>> chaine1 + chaine2
  'Hello world'
  ```

  [16]

- L'opérateur de multiplication « `*` » entre un nombre entier et une chaîne de caractères **duplique** (répète) plusieurs fois une chaîne de caractères.

  ```
  >>> chaine1 * 3
  'Hello Hello Hello '
  ```

La fonction `len()` donne le nombre de caractère d'une chaine (y compris les espaces et les signes de ponctuation).

```
>>> ch = 'Hello world'
>>> len(ch)
11
```

Chaque caractère d'une chaine de caractères `ch` a une position qui va de `0` à `len(ch) - 1`.

- `ch[0]` permet d'accéder au premier caractère en position `0` de la chaine `ch`,
- `ch[1]` au second caractère en position `1`,
- ...
- `ch[i]` au caractère en `i` $^{\text{ième}}$ position,
- ...
- `ch[len(ch) - 1]` au dernier caractère.

⚠️ Les positions sont comptées en commençant à la position `0`, le premier caractère est `ch[0]` et non pas `ch[1]` !

```
>>> ch[6]
'w'
```

- De même, en partant de la fin, `ch[-1]` permet d'accéder au dernier caractère, `ch[-2]` à l'avant dernier, etc.

  ```
  >>> ch[-1]
  'd'
  ```

PEP 8

Pas d'espace autour d'un deux-points (`:`).

- Enfin `ch[i:j]` permet d'obtenir la sous-chaîne de tous les caractères entre les positions `i` (**inclus**) et `j` (**exclus**), appelée une tranche.

  ```
  >>> ch[2:5]
  'llo'
  ```

Les mots-clés `in` et `not in` permettent de vérifier l'appartenance, ou pas, d'une sous-chaine dans une chaine :

```
>>> "py" in "python"
True
>>> "Py" not in "python"
True
```

Il existe de nombreuses méthodes[17] pour traiter les chaines de caractères, quelques exemples :

fonction	description	exemple

`.index('c')`	trouve l'index du premier caractère `"c"` dans une chaîne.	`>>> chaine = 'aaabbbccc'`
		`>>> chaine.index('b')`
		`3`

fonction	description	exemple
`.find('sc')`	cherche la position d'une sous-chaîne `sc` dans la chaîne.	`>>> chaine.find('bc')` `5`
`.count('sc')`	compte le nombre de sous-chaînes `sc` dans la chaîne.	`>>> chaine.count('bc')` `1`
`.lower('sc')`	onvertit une chaîne en minuscules.	`>>> 'ABCdef'.lower()` `'abcdef'`
`.upper('sc')`	onvertit une chaîne en majuscules.	`>>> 'ABCdef'.upper()` `'ABCDEF'`
`.replace('old', 'new')`	remplace tous les caractères `old` par `new` dans la chaîne.	`>>> 'aaabbbccc'.replace('c', 'e')` `'aaabbbeee'`

2.3. Opérateurs de comparaison

Les opérations de comparaison usuelless permettent de comparer des valeurs de même type entre elles. Le résultat est toujours un booléen (de type `bool`) égal à `True` ou `False`[18].

PEP 8

Entourer les opérateurs de comparaison (`==`, `!=`, `>=`, etc.) d'un espace avant et d'un espace après.

opérateur	notation
=	`a == b`
≠	`a != b`
<	`a < b`
≤	`a <= b`
>	`a > b`
≥	`a >= b`

[19]

⚠️ Une erreur courante consiste à confondre l'opérateur de comparaison `==` pour vérifier si deux valeurs sont égales avec l'affectation qui utilise le signe `=` !

```
>>> a, b, c = 5, 5, 6
>>> a == b
True
>>> a == c
False
```

Il est possible de combiner les comparaisons, par exemple pour vérifier si `a` est compris entre 2 et 6 :

```
>>> 2 <= a < 6
True
```

entre 7 et 8 :

```
>>> 7 < a < 8
False
```

mais ce n'est pas recommandé car c'est en fait une combinaison de plusieurs comparaisons, ce qui peut donner des hérésies mathématiques :

```
>>> 4 < a > 2
True
```

Les chaines de caractères, quant à elles, sont comparées en ordre lexicographique, c'est-à-dire caractère par caractère comme l'ordre des mots dans un dictionnaire : on commence par comparer le premier caractère de chaque chaîne, puis en cas d'égalité le deuxième de chaque, et ainsi de suite jusqu'à trouver un caractère qui est

différent de l'autre[20].

```
>>> 'aa'>'ab'
False
>>> "python" == "python"
True
>>> "python" != "PYTHON"
True
```

⚠️ Attention aux majuscules (elles sont "avant" toutes les minuscules) :

```
>>> "java" < "python"
True
>>> "java" > "Python"
True
```

et aux nombres écrits dans des chaînes de caractères :

```
>>> "10" < "2"
True
```

Les nombres de type `int` ou `float` peuvent être comparés entre eux même s'ils sont de types différents :

```
>>> 7 == 7.0
True
>>> 0.0 < 1
True
```

Mais pas les nombres avec les chaines de caractères :

```
>>> 7 == "7"
False
>>> 7 < '8'
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

⚠️ Attention aux égalités entre nombres de type `float` qui ne sont pas toujours encodés de façon exacte[21] :

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

2.4. Opérateurs logiques (ou booléens)

Les opérations logiques peuvent être effectuées sur des booléens (type `bool`). Le résultat est un booléen égal à `True` ou `False`.

opérateur	notation	description	priorité
Négation de `a`	`not a`	`True` si `a` est `False`, `False` sinon	1
`a` et `b` (conjonction)	`a and b`	`True` si `a` et `b` sont `True` tous les deux, `False` sinon	2
`a` ou `b` (disjonction)	`a or b`	`True` si `a` ou `b` (ou les deux) est `True`, `False` sinon	3

(`a` et `b` sont des booléens).

Comme pour les opérations mathématiques, les opérations logiques suivent des règles de priorité :

1. Négation (`not`),
2. Conjonction (`and`),
3. Disjonction (`or`).

`a or not b and c` est équivalent à `a or ((not b) and c)` mais en pratique les parenthèses sont plus lisibles.

2.5. Expressions

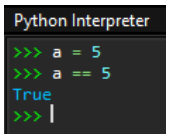
Cours

Une **expression** (ne pas confondre avec une instruction) est un calcul d'opérations et de comparaisons qui donne une valeur.

Exemples :

- `2*a + 5` est une expression, elle a une valeur (qui dépend de la valeur de `a`).
- `a == 5` est une expression booléene, elle vaut `True` ou `False`.
- `a = 5` n'est **pas** une expression, c'est une affectation de la valeur 5 à la variable `a`.

À noter:



> Quand une affectation est saisie dans la console Python, par exemple `>>> a = 5`, rien n'est affiché par l'interpréteur car ce n'est pas une expression.
>
> Quand une expression est saisie dans la console Python, par exemple `>>> a == 5`, elle est évaluée par l'interpréteur et le résultat est affiché en dessous.

Puisqu'elle a une valeur, une expression peut être affectée à une variable : `b = a**2` est une affectation de la valeur de l'expression `a**2` (le carré de `a`) à la variable `b`.

Exercice corrigé

La valeur d'une variable `annee` de type `int` est donnée, par exemple `>>> annee = 2023`.

Ecrire dans l'interpréteur une expression booléenne, qui vaut `True` si `annee` est une année bissextile ou `False` sinon.

« Depuis l'ajustement du calendrier grégorien, l'année sera bissextile (elle aura 366 jours) seulement si elle respecte l'un des deux critères suivants :

1. C1 : l'année est divisible par 4 sans être divisible par 100 (cas des années qui ne sont pas des multiples de 100) ;
2. C2 : l'année est divisible par 400 (cas des années multiples de 100).

Si une année ne respecte ni le critère C1 ni le critère C2, l'année n'est pas bissextile ». Source: https://fr.wikipedia.org/wiki/Année_bissextile.

Réponse

Avant d'écrire cette expression on peut se poser quelques questions :

- Comment savoir si un nombre est divisible par un autre ? Il suffit de vérifier si le reste de la division entière est égal à zéro ou pas. Par exemple 2023 n'est pas divisible par 4 car le reste de la division entière de 2023 par 4 est 3 :

  ```
  >>> annee = 2023
  >>> annee % 4
  3
  ```

  Par contre 2024 est divisible par 4 car le reste de la division entière de 2024 par 4 est bien 0 :

  ```
  >>> annee = 2024
  >>> annee % 4
  0
  ```

- On peut traduire directement en Python chaque condition C1 et C2 :

  C1 : l'année est divisible par 4 sans être divisible par 100 (cas des années qui ne sont pas des multiples de 100) ;

  `>>> annee % 4 == 0 and annee % 100 != 0`

  C2 : l'année est divisible par 400 (cas des années multiples de 100).

  `>>> annee % 400 == 0`

- ⚠️ la dernière clause indique qu'une année **n'est pas** bissextile si les conditions C1 **et** C2 sont toutes les deux **fausses**. Il faut donc comprendre qu'une année **est** bissextile si l'une des conditions C1 **ou** C2 est **vraie** (ou les deux en même temps).

Traduit en Python, on obtient l'expression suivante que l'on peut tester dans la console.

> On pourrait se passer des parenthèses et utiliser les règles de priorités des opérateurs booléens : `annee % 4 == 0 and annee % 100 != 0 or annee % 400 == 0`, mais en pratique ce n'est pas recommandé.

```
>>> annee = 2023
>>> (annee % 4 == 0 and annee % 100 != 0) or annee % 400 == 0
False
```

3. Instructions

Cours

Une **instruction** (ne pas confondre avec une expression) est une commande qui doit être effectuée par un programme.

Une **séquence** est une suite d'instructions.

Par exemple :

- `a = 2` est une instruction qui affecte la valeur `2` à la variable `a`.
- `print('Hello world')` est une instruction qui affiche la chaine `'Hello world'` dans la console.
- `a == 2` n'est pas une instruction, c'est une expression qui compare la valeur de `a` à la valeur `2`.

3.1. `type()`

La fonction `type()` permet de connaitre le type d'une variable.[22]

PEP 8

Pas d'espace avant et à l'intérieur des parenthèses d'une fonction.

```
>>> x = 2
>>> type(x)
<class 'int'>
>>> y = 2.0
>>> type(y)
<class 'float'>
>>> z = '2'
>>> type(z)
<class 'str'>
```

3.2. Conversion de type

Les fonctions suivantes permettent de convertir une variable d'un type à un autre :

fonction	description	exemple

`int()`	Convertit une chaine de caractères ou un flottant en entier.	`>>> int(2.8)`
		`2`
		`>>> int('2')`
		`2`

`float()`	Convertit une chaine de caractères ou un entier en flottant.	`>> float(5)`
		`5.0`
		`>>> float('5.5')`
		`5.5`

`str()`	Convertit un entier ou un flottant en une chaine de caractères.	`>>> str(5.5)` `'5.5'`

Observons dans la console comment une variable de type `float` qui a une valeur entière est affiché avec un point :

```
>>> a = 5
>>> a
5
>>> float(a)
5.0
```

3.3. Instructions d'entrée et sortie

Cours

Une instruction d'**entrée** permet à un programme de lire une valeur saisie au clavier par l'utilisateur. Une instruction de **sortie** affiche un message sur l'écran de l'utilisateur.

En Python, la fonction `input()` permet d'écrire une instruction d'entrée qui affecte la valeur saisie par l'utilisateur à une variable.

```
>>> saisie = input('Saisir un message')
>>> saisie
'abc'
```

La valeur renvoyée par `input()` est toujours du type `str` :

```
>>> nombre_entier = input('Entrez un nombre entier')
>>> nombre_entier
'25'
```

Ici la valeur affectée à `nombre_entier` est une chaine de caractères : `'25'`. Pour obtenir un nombre, de type `int` ou `float`, afin de faire des calculs par la suite par exemple, il faut la convertir :

```
>>> nombre_entier = int(input('Entrez un nombre entier'))
>>> nombre_entier
25
```

> Si l'utilisateur ne saisit pas un nombre entier, cette instruction génère un message d'erreur.

Une instruction de sortie s'écrit en utilisant `print()` pour afficher à l'écran des chaines de caractère et/ou des variables, séparés par des virgules.

PEP 8

Un espace après une virgule (`,`), mais pas avant.

```
>>> print('Hello')
Hello
>>> message='world'
>>> print('Hello', message)
Hello world
>>> nombre = 5
>>> print(nombre)
5
>>> print('le nombre est', nombre)
le nombre est 5
>>> a = 5
>>> b = 6
>>> print('la somme de', a, 'et de', b, 'est', a + b)
la somme de 5 et de 6 est 11
```

Par défaut, `print()` provoque un retour à la ligne après chaque affichage. Pour changer ce comportement il faut préciser la fin de l'affichage en ajoutant un paramètre `end=` suivi d'une chaine de caractères, par exemple un espace `end=' '` ou même une chaine vide `end=''`.

```
>>> print('Hello', end=' ')
Hello >>>
```

Python 3.6 a introduit les chaine de caractères f-strings (*formatted string*) qui s'écrivent avec `f` devant et permettent d'y insérer des variables, ou même des expressions, entre accolades.

```
>>> prenom = 'Paul'
>>> annee_naissance = 2010
>>> print(f'Bonjour {prenom}, vous avez {2023 - annee_naissance} ans')
Votre nom est un Paul et vous avez 13 ans
```

3.4. Premier programme

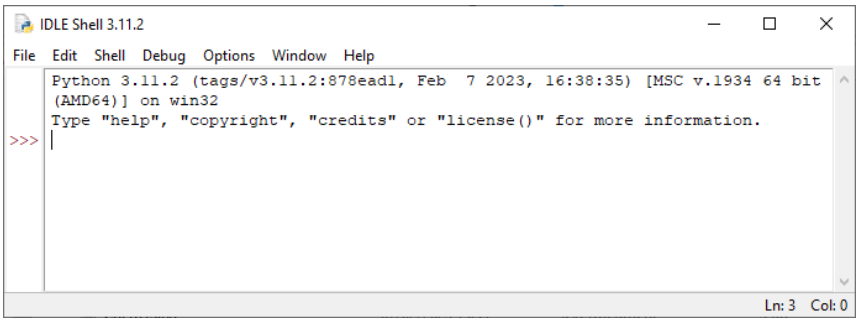
Pour permettre à l'utilisateur d'entrer son prénom et sa date de naissance et d'affecter ses réponses aux variables `prenom` et `annee_naissance`, il faut à chaque fois écrire dans la console les instructions suivantes :

```
>>> prenom = input('Entrez votre prénom : ')
Entrez votre prénom : Paul
>>> annee_naissance = int(input('Entrez votre date de naissance : '))
Entrez votre date de naissance : 2010
>>> print(f'Bonjour {prenom}, vous avez {2023 - annee_naissance} ans')
Bonjour Paul, vous avez 13 ans
>>>
```

Cette séquence montre les limites de la console, qui répond à des commandes de façon interactive, mais ne permet pas d'écrire un programme élaboré !

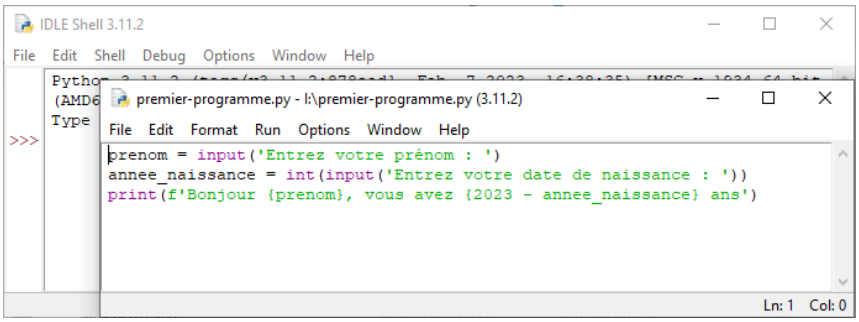
Ouvrons IDLE (/python/Lib/idlelib/idle.bat) pour écrire un premier programme, l'interpréteur de commande avec l'invite Python `>>>` apparaît :

Python propose par défaut un **IDE** (pour *Integrated Development Environment*) appelé IDLE. Il existe de nombreux IDE, certains dédiés à Python comme PyScripter, Thonny, etc. et d'autres généralistes comme VS Codium, VS Code, etc. acceptant plusieurs langages informatiques.

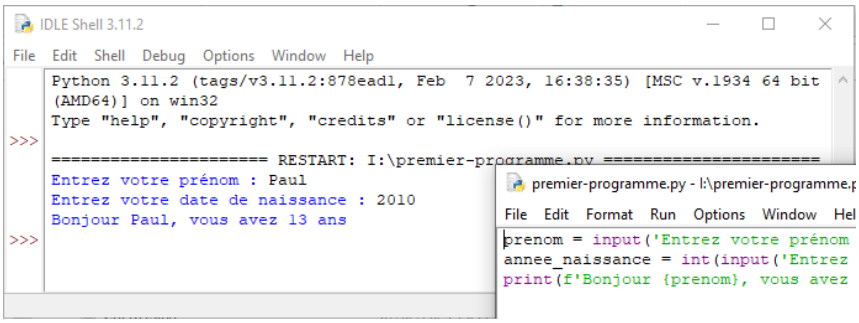


Ouvrons un nouveau fichier avec le menu `File/New` pour entrer le programme Python suivant :

```
prenom = input('Entrez votre prénom : ')
annee_naissance = int(input('Entrez votre date de naissance : '))
print(f'Bonjour {prenom}, vous avez {2023 - annee_naissance} ans')
```



Enregistrons le programme dans nos fichiers avec le menu `File/Save As` puis `Run/Run Module` pour exécuter le programme. Le résultat est affiché dans la console :



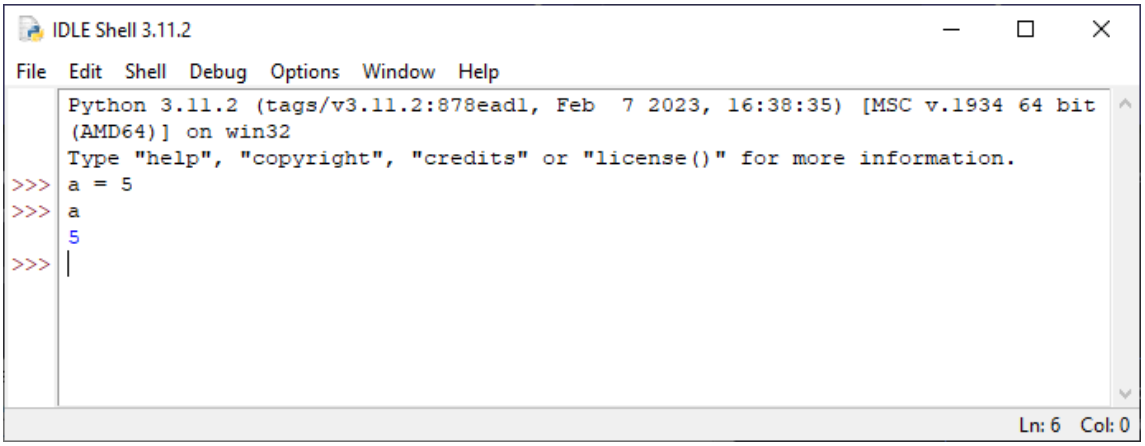
Nous avons écrit notre premier programme informatique !

Notons au passage une différence importante entre l'affichage d'une variable depuis la console et depuis un programme :

Depuis la console

Il suffit de saisir le nom de la variable à l'invite de commande pour afficher sa valeur :

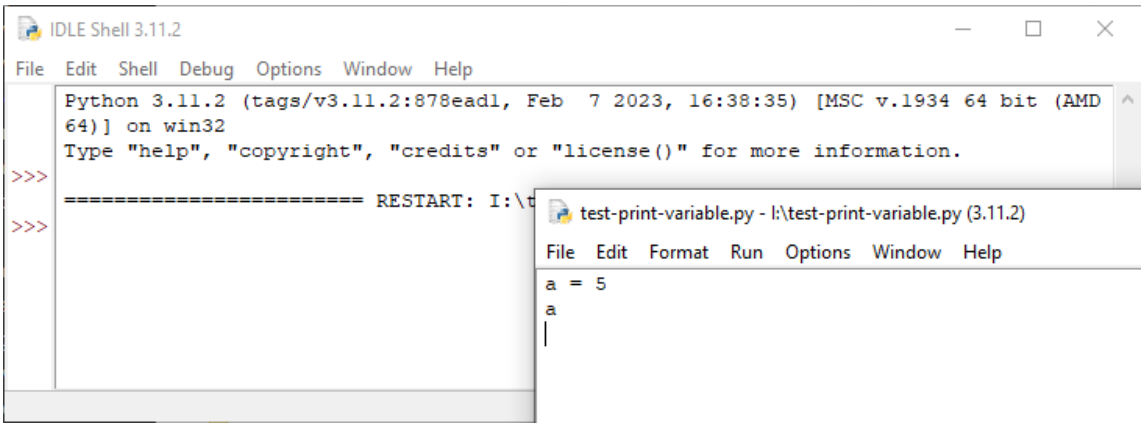
```
>>> a = 5
>>> a
5
```



Depuis un programme

Le programme suivant n'affiche rien dans la console :

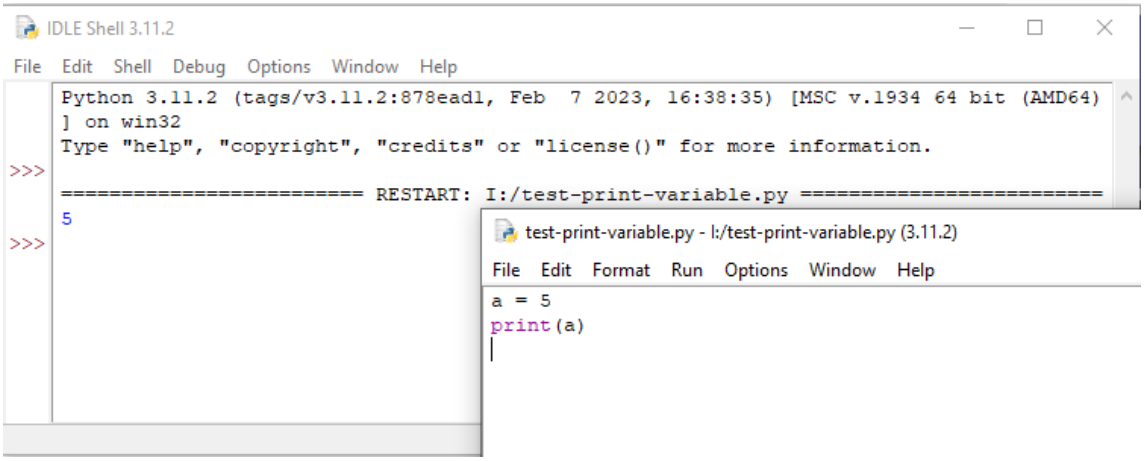
```
1  a = 5
2  a
```



Depuis un programme avec print()

Il faut utiliser l'instruction `print()` dans un programme pour afficher la valeur d'une variable dans la console.

```
1  a = 5
2  print(a)
```



Exercice corrigé

Pour passer d'un pixel couleur codé RGB (mélange des trois couleurs rouge, vert, bleu) à un pixel en nuance de gris, on utilise la formule suivante qui donne le niveau de gris : $G = 0,11 \times R + 0,83 \times V + 0,06 \times B$ où R, V et B sont les niveaux de rouge, vert et bleu.

Ecrire le programme qui demande en entrée les 3 couleurs d'un pixel et affiche en sortie la nuance de gris.

Réponse

Quelques questions à se poser avant d'écrire le programme demandé :

- Quelles sont les informations à saisir par l'utilisateur ? Les trois niveaux de couleurs R, V et B.
- Où stocker ces informations ? Dans trois variables de type `int` nommées par exemple `R`, `V` et `B` comme dans la formule.
- Que doit calculer le programme ? Le niveau de gris calculé en utilisant la formule et stocké dans une variable, nommée par exemple `G`, de type `int`.
- Que doit faire ensuite le programme ? Le programme doit afficher le niveau de gris.

Traduit en Python, le programme s'écrit simplement :

Noter la présence de commentaires dans le code, commençant par le signe `#`, ils sont ignorés par l'interpréteur Python.

Essayer le programme sans faire la conversion des variables `R`, `V` et `B` en `int` et constater l'erreur produite.

```
# Demande les 3 couleurs R, V et B de type int
R = int(input('Rouge:'))
V = int(input('Vert:'))
B = int(input('Bleu:'))
# Calcule le niveau de gris G, de type int
G = int(0.11 * R + 0.83 * V + 0.06 * B)
# Affiche le niveau de gris
print(f'Le niveau de Gris est {G}')
```

4. Constructions élémentaires

Cours

En Python, l'**indentation**, ou décalage vers la droite du début de ligne, délimite les séquences d'instructions et facilite la lisibilité en permettant d'identifier des **blocs**. La ligne précédant une indentation se termine toujours par le signe deux-points.

PEP 8

Préférer les espaces aux tabulations.

L'indentation est normalement réalisée par quatre caractères « espace ». Python accèpte aussi une tabulation, voire un seul ou un autre nombre d'espaces, mais dans tous les cas il faut être consistant à travers tout un programme.

4.1. Instructions conditionnelles

Cours

Une instruction conditionnelle exécute, ou pas, une séquence d'instructions suivant la valeur d'une condition (une expression booléenne qui prend la valeur `True` ou `False`).

```
if condition:
    instructions
```

Par exemple, ce programme détermine le stade de la vie d'une personne selon son age.

PEP 8

Pas d'espace avant le deux-points (`:`).

```
age = int(input("Quel age avez-vous ?"))
if age >= 18:
    stade = "adulte"
    print(f"Vous êtes un {stade}")
    print(f"Vous avez {age} ans")
```

> Ne pas oublier les deux-points « `:` » après la condition et l'indentation sur les lignes suivantes.

L'indentation détermine la séquence à exécuter (ou pas), dans ce cas les instructions qui suivent aux lignes 3, 4 et 5. Quand la condition (l'expression booléenne) `age >= 18` n'est pas vérifiée, aucune des trois instructions n'est exécutée.

Il est possible de ne pas indenter l'instruction en ligne 5 `print(f"Vous avez {age} ans")`, dans ce cas elle ne ferait plus partie de l'instruction conditionnelle et serait alors exécutée dans tous les cas.

```
1  age = int(input("Quel age avez-vous ?"))
2  if age >= 18:
3      stade = "adulte"
4      print(f"Vous êtes {stade}")
5  print(f"Vous avez {age} ans")
```

Par contre, si l'instruction en ligne 4 `print(f"Vous êtes {stade}")` n'était pas indentée, la variable `stade` ne serait pas définie quand la condition n'est pas vérifiée et dans ce cas il y aura un message erreur à la ligne 4.

La structure `if-else` permet de gérer le cas où la condition est fausse :

Cours

```
if condition:
    instructions
else :
    instructions_sinon
```

> L'instuction `else` n'a pas de condition, elle est toujours suivie des deux-points « `:` » .

```
1  age = int(input("Quel age avez-vous ?"))
2  if age >= 18:
3      stade = "adulte"
4  else:
5      stade = "enfant"
6  print(f"Vous êtes {stade}")
7  print(f"Vous avez {age} ans")
```

Dans ce cas, la variable `stade` est toujours définie, et l'instruction en ligne 6 `print(f"Vous êtes {stade}")` peut ne pas être indentée.

La structure `if-elif-else` permet de remplacer des instructions conditionnelles imbriquées pour gérer plusieurs cas distincts :

Cours

```
if condition_1:
    instructions_si_1
elif condition_2:
    instructions_si_2
elif condition_3:
    instructions_si_3
…
else :
    instructions_sinon
```

Ces deux programmes font exactement la même chose, mais le second est plsu lisible :

Instructions conditionnelles imbriquées

```
1  if age >= 18:
2      stade = "adulte"    # au dessus de 18
3  else:
4      if age >= 12:
5          stade = "ado"    # entre 12 et 18
6      else:
7          if age >= 2:
8              stade = "enfant"    # entre 2 et 12
9          else :
10             stade = "bébé"    # moins de 2
11
12 print(f"Vous êtes {stade}, vous avez {age} ans")
```

Chaque fois qu'une condition `if` n'est pas vérifiée, le programme exécute toute la séquence `else` correspondante. Il "descend" ainsi de suite dans les conditions imbriquées jusqu'à ce qu'une condition soit vérifiée, et à ce stade il sort de tous les blocs conditionnels et reprend à l'instruction qui suit tous les blocs conditionnels imbriqués, ici à ligne 12 `print(f"Vous êtes {stade}, vous avez {age} ans")`.

Par exemple, si on affecte la valeur `10` à la variable `age`, la première condition en ligne 1 `if age >= 18:` est fausse, le programme exécute donc toute la partie indentée après le premier `else:` correspondant en ligne 3. Il passe à la seconde condition en ligne 4 `if age >= 12:` qui est encore fausse, il "passe" donc à la séquence indentée après le `else:` correspondant en ligne 6. La troisième condition en ligne 7 `if age >= 2:` est cette fois vraie, il exécute la ligne 8 `stade = "enfant"` et sort de toutes les instructions conditionnelles pour reprendre à la ligne 12 et afficher le message `Vous êtes enfant, vous avez 10 ans`.

Noter qu'il faut bien prendre soin à l'indentation et que ce programme n'est pas très lisible !

Instructions conditionnelles en utilisant la structure `if-elif-else`

```
1
2
3
4
5
6
7
8
9
10
```

```python
if age >= 18:
    stade = "adulte"    # au dessus de 18
elif age >= 12:
    stade = "ado"    # entre 12 et 18
elif age >= 2:
    stade = "enfant"    # entre 2 et 12
else :
    stade = "bébé"    # moins de 2

print(f"Vous êtes {stade}, vous avez {age} ans")
```

Dès que la première conditions `if` ou une condition `elif` est vérifiée, le programme ne teste plus les conditions `elif` suivantes ni le `else` final, mais reprend directement à l'instruction qui suit le bloc conditionnel, ici `print(f"Vous êtes {stade}, vous avez {age} ans")`.

Par exemple, si on affecte la valeur `15` à la variable `age`, la première condition en ligne 1 `if age >= 18:` est fausse, le programme passe donc à la conditions suivante `elif age >= 12:…`, celle-ci est vérifiée, il exécute donc la ligne 4 `stade = "ado"` et n'a pas besoin de vérifier la condition suivante en ligne 5 `elif age >=2 :…` ni d'exécuter le `else:…`. Dès que la condition `elif age >= 12:…` a été vérifiée, il sort de toute l'instruction conditionnelle pour reprendre à la ligne 10 et afficher le message `Vous êtes ado, vous avez 15 ans`.

Rappel

Eviter les conditions d'égalité avec les nombres de type float. Par exemple :

```python
if 2.3 - 0.3  == 2:
    print('Python sait bien calculer avec les float')
else:
    print('Python ne sait pas bien calculer avec les float')
```

PEP 8

Eviter de comparer des variables booléennes à `True` ou `False` avec `==` ou avec `is`.

On écrit :

```python
cond = True
if cond:
    print("la condition est vraie"
```

mais pas `if cond == True:` ni `if cond is True:`

Exercice corrigé

Écrire un programme qui demande une année et affiche si elle est bissextile ou pas :

1. en utilisant des conditions imbriquées.
2. en utilisant une structure `if-elif-else`.

« Depuis l'ajustement du calendrier grégorien, l'année sera bissextile (elle aura 366 jours) seulement si elle respecte l'un des deux critères suivants :

1. C1 : l'année est divisible par 4 sans être divisible par 100 (cas des années qui ne sont pas des multiples de 100) ;
2. C2 : l'année est divisible par 400 (cas des années multiples de 100).

Si une année ne respecte ni le critère C1 ni le critère C2, l'année n'est pas bissextile ».

Source: https://fr.wikipedia.org/wiki/Année_bissextile.

Réponse 1.en utilisant des conditions imbriquées

Analysons la définition donnée par Wikipedia sous la forme d'un arbre :

```
graph LR
A[annee%4 == 0] --> |True| B;
A -->|False| C{pas bissextile};
B[annee%100 != 0] --> |True| D{bissextile};
B --> |False| E;
E[annee%400 == 0] --> |True| F{bissextile};
E --> |False| G{pas bissextile};
```

Traduit en Python, on obtient le programme suivant.

```
annee = int(input('annee: ') )
if annee % 4 == 0:                      # si annee est divisible par 4 ...
    if annee % 100 != 0:                    # ... mais pas par 100,
        print(annee, "est bissextile")          # alors elle est bissextile
    else:                                   # ... et par 100, alors
        if annee % 400 == 0:                    # soit elle est divisible par 400 ...
            print(annee, "est bissextile")          # ...donc elle est bissextile
        else:                                   # soit elle n'est pas divisible par 400
            print(annee, "n'est pas bissextile")    # ...donc elle n'est pas bissextile
else:                                   # sinon, elle n'est divisible par 4 ...
    print(annee, "n'est pas bissextile")   # ...donc elle n'est pas bissextile
```

Le programme n'est pas facile à lire. On note ici l'importance de l'indentation !

Réponse 2.en utilisant une structure `if-elif-else`

Chaque condition `if`, `elif`, `else` vérifiée termine la structure conditionnelle. Il faut donc tester les divisibilités en partant du plus grand nombre, `400`, puis `100`, puis `4` (car une fois qu'on a testé la divisibilité par `4`, on ne peut plus tester la divisibilité par `100` ou `400`). Modifions l'arbre précédent:

```
graph LR
A[annee%400 == 0] --> |True| C{bissextile};
A -->|False| B;
B[annee%100 == 0] -->  |True| E;
B -->|False| D{pas bissextile};
E[annee%4 == 0] --> |True| F{bissextile};
E --> |False| G{pas bissextile};
```

Traduit en Python, on obtient le programme suivant.

```
annee = int(input('annee: ') )
if annee % 400 == 0:        # si annee est divisible par 400
    print(annee, "est bissextile")
elif annee % 100 == 0:        # sinon, si annee est divisible par 100 (et pas par 400 car déjà testé)
    print(annee, "n'est pas est bissextile")
elif annee % 4 == 0:          # sinon, si annee est divisible par 4 (et pas par 100 et 400 car déjà testés)
        print(annee, "est bissextile")
else:                         # sinon (l'annee n'est pas divisible par 4, 100 et 400 car déjà testés)
    print(annee, "n'est pas est bissextile")
```

Le programme est beaucoup plus lisible.

Note

il est bien sûr aussi possible d'utiliser l'expression trouvée dans l'exercice corrigé précédent :

```
annee = int(input('annee: ') )
if (annee % 4 == 0 and not annee % 100 == 0) or annee % 400 == 0:
    print(annee, 'est bissextile')
else:
    print(annee, "n'est pas bissextile")
```

mais ce n'est pas très lisible.

4.2. Boucles non bornées

Cours

Une boucle non bornée (ou boucle conditionnelle) permet de répéter une instruction ou une séquence d'instructions, tant qu'une condition (une expression booléenne) est vraie.

```
while condition:
    instructions
```

La structure est similaire à l'instruction conditionnelle. La condition qui suit le mot `while` est une expression de type booléen qui prend la valeur `True` ou `False`. Le bloc d'instructions qui suit est exécuté tant que la condition est vraie. Ce bloc d'instruction doit impérativement modifier la valeur de la condition afin qu'elle finisse par ne plus être vérifiée, sinon la boucle est sans fin et le programme ne se terminera jamais !

Exemple :

Rappel

`i += 2` est une abréviation de `i = i + 2`

```
>>> i = 0
>>> while i <= 10:
...     print(i)
...     i += 2
...
0
2
4
6
8
10
```

Il faut toujours impérativement **vérifier que la condition ne sera plus vérifiée** après un nombre fini de passage, sinon le programme ne s'arrête jamais, le **programme boucle** ou diverge. Ici, c'est bien le cas grâce à l'instruction `i += 2`, `i` finira bien par être plus grand que 10.

Exemple de programme qui boucle (erreur d'indentation dans l'instruction `i += 2`):

```
1  i = 0
2  while i <= 10:
3      print(i)
4  i += 2
```

⚠ Comme pour les instructions conditionnelles, il faut faire particulièrement attention aux boucles avec les nombres de type `float`. La boucle suivante qui semble écrite correctement ne finira jamais :

```
i = 1
while i != 2:
    i += 0.1
    print(i)
```

Testons la même boucle écrite correctement :

```
i = 1
while i < 2:
    i += 0.1
    print(i)
```

Elle affiche dans la console :

```
1.1
1.2000000000000002
1.3000000000000003
1.4000000000000004
1.5000000000000004
1.6000000000000005
1.7000000000000006
1.8000000000000007
1.9000000000000008
2.000000000000001
```

`i` ne prend donc jamais la valeur `2`.

4.3. Boucles bornées

Cours

Une boucle bornée (ou boucle non conditionnelle) permet de répéter `n` fois, `n` étant un nombre entier connu, une instruction ou une séquence d'instructions.

```
for i in range(n):
    instructions
```

`i` est appelé l'**indice de boucle ou compteur de boucle**, il prend les `n` valeurs entières comprises **entre 0 et n - 1**.

⚠ **`i` ne prend pas la valeur `n`.**

Il est aussi possible d'utiliser :

- `range(d, f)` qui énumère les `f-d` nombres entiers compris entre `d` et `f-1`. [23]
- `range(d, f, p)` qui énumère les nombres entiers compris entre `d` et `f-1` avec un pas de `p` : `d`, `d+p`, `d+2p`, etc. [24]

La boucle bornée ci-dessus est très similaire à la boucle non bornée suivante :

```
i = 0
while i < n :
    instructions
    i += 1
```

⚠️mais attention, comparons ces deux programmes :

Programme 1

```
for i in range(5):
    print(i, end=" ")
```

Programme 2

```
i = 0
while i < 5:
    print(i, end=" "))
    i = i + 1
```

Les deux programmes semblent afficher la même chose, tous les chiffres de 0 à 4 : `0 1 2 3 4`

Pourtant ils sont différents. Ajoutons une instruction `print(i)` à la fin les deux programmes.

Programme 1

```
for i in range(5):
    print(i, end=" ")
print(i)
```

affiche : `0 1 2 3 4 4`

La valeur finale de `i` est `4`

Programme 2

```
i = 0
while i < 5:
    print(i, end=" ")
    i = i + 1
print(i)
```

affiche : `0 1 2 3 4 5` La valeur finale de `i` est `5` afin que la condition ne soit plus valide.

Dans le programme 1, la valeur finale de `i` est `4`, alors que dans le programme 2 c'est `5` afin que la condition ne soit plus valide.

Autre différence, quand on modifie l'indice de boucle dans une boucle `for`, il reprend la valeur suivante à la prochaine répétition.

Programme 1

```
for i in range(5):
    print(i, end=" ")
    i = i + 2
    print(i, end=" ")
```

affiche : `0 2 1 3 2 4 3 5 4 6`

La valeur de `i` est modifiée à l'intérieur de la boucle mais reprend la valeur suivante à la prochaine répétition.

Programme 2

```
i = 0
while i < 5:
    print(i, end=" ")
    i = i + 2
    print(i, end=" ")
```

affiche : `0 2 2 4 4 6`

La valeur de `i` est modifiée à l'intérieur de la boucle.

A chaque passage dans une boucle `for`, l'indice de boucle repart de sa valeur au dernier passage dans la boucle, même si cette valeur a changé dans la boucle. En pratique, il n'est pas recommendé de changer sa valeur dans la boucle.

La boucle `for` possède d'autres possibilités très utiles, par exemple elle permet d'énumérer chaque caractère d'une chaine de caractères. Le programme ci-dessous affiche chaque lettre d'une variable message l'une après l'autre.

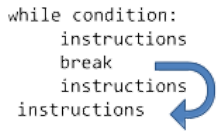
```
message = 'Hello world'
for c in message:
    print(c)
```

4.4. Instructions break et continue

Il existe deux instructions qui permettent de modifier l'exécution des boucles `while` et `for`, il s'agit de `break` et de `continue`.

Cours

L'instruction `break` permet de sortir de la boucle courante et de passer à l'instruction suivante.

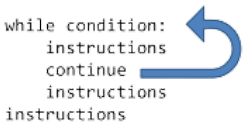


Par exemple, voici un programme qui redemande un mot de passe jusqu'à obtenir le bon :

```
while True:
    mdp = input('mot de passe')
    if mdp == '123456':
        break
print('trouvé')
```

Cours

L'instruction `continue` permet de sauter les instructions qui restent jusqu'à la fin de la boucle et de reprendre à la prochaine itération de boucle.



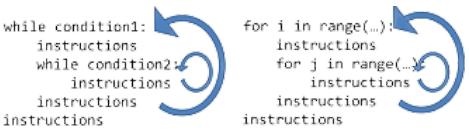
Imaginons par exemple un programme qui affiche la valeur de $1/(n-7)$ pour tous les entiers n compris entre 1 et 10. Il est évident que quand n prend la valeur 7 il y aura une erreur. Grâce à l'instruction `continue`, il est possible de traiter cette valeur à part puis de continuer la boucle.

```
for n in range(10):
    if n == 7:
        continue
    print(1 / (7 - n))
```

4.5. Boucles imbriquées

Cours

Il est possible d'imbriquer des boucles. A chaque passage dans la première boucle (la boucle externe), la seconde boucle (la boucle interne) est effectuée entièrement.



⚠️ Attention à l'indentation pour déterminer à quelle boucle appartiennent les instructions.

Par exemple le programme suivant affiche toutes les heures, minutes et seconde de la journée (de 0h0min0s à 23h59min59s) :

```
for heure in range(24):
    for minute in range(60):
        for seconde in range(60):
            print(f"{heure}h{minute}min{seconde}s")
```

Exercice corrigé

Exercice corrigé : Écrire un programme en Python qui affiche tous les nombres premiers inférieurs à 100.

Rappel : un nombre est premier s'il n'a que deux diviseurs, 1 et lui-même.

Réponse

On utilise deux boucles for imbriquées :

- La première boucle parcourt tous les nombres `n` allant de 2 (0 et 1 ne sont pas premiers) à 100 pour vérifier s'ils sont premiers ou pas.

On peut se contenter de chercher un diviseur `div` seulement entre 2 et $\sqrt{n}$ avec une boucle `while div**2 <= n:` ou `while div <= n**0.5:`.

- Pour chaque nombre `n`, la deuxième boucle vérifie s'il est divisible par un nombre `div` compris entre 2 et `n - 1`. Si `n` est divisible par `div`, alors il n'est pas premier et on affecte la valeur `False` à la variable `est_premier`. Dans ce cas, inutile de chercher d'autres diviseurs, on peut sortie de la boucle avec une instruction `break`.

Ensuite, à la fin de la deuxième boucle, on vérifie si la variable `est_premier` est `True` et dans ce cas, cela signifie que le `n` est premier et il est affiché à l'écran.

nombres_premiers.py

```
for n in range(2, 101):
    est_premier = True
    for div in range(2, n):   #on cherche un diviseur compris entre 2 et n - 1
        if n % div == 0:
            est_premier = False  # on a trouvé un diviseur de n, il n'est pas premier
            break  # on peut sortir de la boucle interne ici, inutile de continuer
    if est_premier:         # on préfère à : if premier == True
        print(n, end="-")
```

5. Appel de fonctions

Nous avons déjà utilisé des fonctions comme `print()` ou `len()` qui sont des fonctions prédéfinies par Python. Un programme utilise beaucoup de ces fonctions Python, mais il est aussi souvent très utile de créer nos propres fonctions, ce qui présente de nombreux avantages :

Noter ici la différence avec une fonction mathématique.

- Modularité : Les fonctions permettent de découper un programme en petites parties indépendantes, ce qui facilite la lisibilité et la résolution de problèmes complexes.
- Réutilisabilité : Une fois qu'une fonction est définie, elle peut être appelée plusieurs fois sans avoir à réécrire le même bloc de code à chaque fois.
- Testabilité : Les fonctions sont des séquences isolées de code qui peuvent être testées individuellement.

Cours

Une fonction est définie (ou « déclarée ») par :

- le mot réservé `def` (pour *define*),
- son nom,
- zéro, un ou plusieurs paramètres écrits entre parenthèses (les parenthèses sont obligatoires même quand il n'y a pas de paramètres) et séparés par des virgules,
- deux-points `:`,
- une séquence d'instructions indentées (le « corps » de la fonction).

```
def nom_dela_fonction(param1, param2, ...):
    instructions
```

Comme pour les noms de variables, le nom d'une fonction :

- s'écrit en lettres minuscules (« `a` » à « `z` ») et majuscules (« `A` » à « `Z` ») et peut contenir des chiffres (« `0` » à « `9` ») et le caractère blanc souligné (« `_` ») ;
- ne doit pas comporter d'espace, de signes d'opération « `+` », « `-` », « `*` » ou « `/` », ni de caractères spéciaux comme des signes de ponctuation « `'` », « `"` », « `,` », « `.` », « `:` », « `@` », etc. ;
- ne doit pas commencer par un chiffre ;
- ne doit pas être un mot réservé de Python, par exemple « `for` », « `if` », « `print` », etc. ; et
- est sensible à la casse, ce qui signifie que les fonctions « `TesT` », « `test` » ou « `TEST` » sont différentes.

De la même façon que dans les constructions élémentaires vues précédemment (`if-else`, `while`, `for`), c'est l'indentation qui suit les deux-points qui détermine le bloc d'instructions qui forment la fonction.

Lorsqu'une fonction est définie dans un programme, elle ne s'exécute pas automatiquement. Et ceci même si la fonction comporte une erreur, l'interpréteur Python ne s'en aperçoit pas.

Programme 1

La fonction `bonjour` n'est pas appelée, ce programme ne fait rien.

```
def bonjour():
    print('hello')
```

Programme 2

La fonction `bonjour` n'est pas appelée, ce programme ne fait rien, même s'il y une erreur, 🐛 il manque des apostrophes ou des guillemets autour de `'hello'`.

```
def bonjour():
    print(hello)
```

Définir une fonction consiste simplement à décrire son comportement et à lui donner un nom. Pour exécuter la fonction, il faut l'**appeler** depuis un programme ou depuis la console Python en écrivant son nom suivi des parenthèse.

> ⚠️ Quand la fonction n'a pas de paramètres, il faut quand même mettre les parenthèses pour l'appeller.

Depuis la console

```
def bonjour():
    print('hello')
```

Ici le programme définit une fonction mais ne l'appelle pas. Elle peut être appelée depuis la console :

```
>>> bonjour()
hello
```

Depuis un programme

```
def bonjour():
    print('hello')

bonjour()
```

Ici le programme définit une fonction et l'appelle immédiatement. Quand le programme est exécuté, il affiche dans la console :

```
hello
```

Il faut définir une fonction **avant** de l'appeler. Ces deux programmes affichent un message d'erreur :

Programme 1

```
bonjour()

def bonjour():
    print('hello')
```

La fonction `bonjour` est appelée avant d'être définie, le programme affiche un message d'erreur :

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

Programme 2

```
def main():
    bonjour()

if __name__ == '__main__':
    main()

```

```
7  def bonjour():
8      print('hello')
```

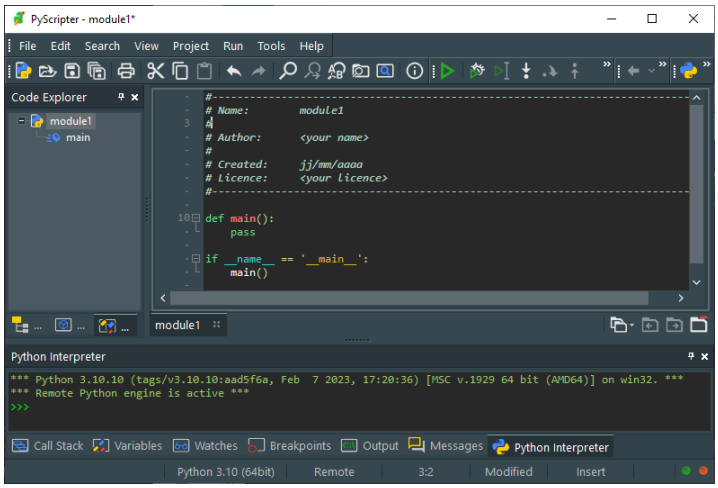
La fonction `bonjour` est appelée avant d'être définie, le programme affiche un message d'erreur :

```
Traceback (most recent call last):
  File "<module1>", line 5, in <module>
  File "<module1>", line 2, in main
NameError: name 'bonjour' is not defined
```

5.1. La fonction `main()`

PyScripter, comme d'autres IDE (*Integrated Development Environment*), génère automatiquement une fonction appelée `main` avec le code suivant :

```
1  def main():
2      pass
3
4  if __name__ == '__main__':
5      main()
```



En Python, comme dans la plupart des langages de programmation, il y a une fonction principale, appelée souvent `main()`. Elle sert de point de départ de l'exécution d'un programme.

L'interpréteur Python exécute tout programme linéairement de haut en bas, donc il n'est pas indispensable de définir cette fonction `main()` dans chaque programme, mais il est recommandé de le faire dans un long programme découpés en plusieurs fonctions afin de mieux comprendre son fonctionnement.

5.2. Paramètres et arguments

Cours

Même si dans la pratique les deux termes sont souvent confondus par abus de langage, il faut faire la différence entre :

- Les **paramètres** (ou paramètres formels) d'une fonction sont des noms de variables écrits entre parenthèses après le nom de la fonction qui sont utilisées par la fonction.
- Les **arguments** (ou paramètres réels) sont les valeurs qui sont données aux paramètres lorsque la fonction est appelée.

On appelle une fonction en écrivant son nom suivi des arguments entre parenthèses.

Prenons en exemple une fonction simple :

```
1  def bonjour(prenom1, prenom2):
2      print('hello', prenom1, 'and', prenom2)
3
4  bonjour('Tom', 'Lea')
```

La fonction `bonjour` est définie en ligne 1 par « `def bonjour(prenom1, prenom2):` » avec deux **paramètres** `prenom1` et `prenom2`. Elle est ensuite appelée à la ligne 4, `bonjour('Tom', 'Lea')`, en lui passant les **arguments** `'Tom'` et `'Lea'`, ce sont les valeurs que prennent les deux paramètres `prenom1` et `prenom2` pendant l'exécution de la fonction.

`prenom1` prend la valeur du premier argument quand on appelle cette fonction `bonjour` et `prenom2` la valeur du deuxième. `prenom1` et `prenom2` sont appelés des **paramètres positionnels** (en anglais *positional arguments*). Il est **obligatoire** de leur donner une valeur quand on appelle une fonction. Par défaut, les paramètres prennent les valeurs des arguments **dans l'ordre de leurs positions respectives**, dans l'exemple ci-dessus `prenom1` prend la valeur `'Tom'` et `prenom2` la valeur `'Lea'`, comme indiqué par leur position.

Néanmoins il est possible de changer l'ordre des arguments en précisant le nom du paramètre auquel chacun correspond. Par exemple, ces deux appels de fonctions sont identiques :

```
>>> bonjour('Tom', 'Lea')
hello Tom and Lea
>>> bonjour(prenom2 = 'Lea', prenom1 = 'Tom')
hello Tom and Lea
```

Dans tous les cas, il faut appeler une fonction avec **suffisament d'arguments pour tous ses paramètres positionnels**, sinon la fonction ne peut pas s'éxécuter et affiche un message d'erreur 🐛 :

```
>>> bonjour('Tom')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: bonjour() missing 1 required positional argument: 'prenom2'
```

En plus des paramètres positionnels qui sont obligatoires, il existe des paramètres qui sont facultatifs ayant une valeur d'argument par défaut s'il ne sont pas renseignés, c'est-à-dire la valeur que prendra un paramètre si la fonction est appelée sans argument correspondant.

PEP 8

Pas d'espace autour du égal (`=`) dans le cas des arguments par mot-clé (à la différence de l'affectation où ils sont recommandés).

```
1  def bonjour(prenom1, prenom2='Lisa'):
2      print('hello', prenom1, 'and', prenom2)
3
4  --- Exemple d'appel dans l'interpreteur-----------------
5  >>> bonjour('Tom')
6  hello Tom and Lisa
```

Ici, lorsque la fonction est définie à la ligne 1 par « `def bonjour(prenom1, prenom2='Lisa'):` », la valeur de `prenom2` est `'Lisa'` par défaut, c'est la valeur qui est utilisée par la fonction quand elle est appelée sans argument correspondant. `prenom2` est appelé un **paramètre par mot-clé** (en anglais *keyword argument*). Le passage d'un tel argument lors de l'appel de la fonction est **facultatif**.[25]

Comme les paramètres positionnels, il est possible de changer l'ordre des arguments en précisant le nom du paramètre auquel chacun correspond.

Prenons l'exemple d'une fonction avec un paramètre positionnel (obligatoire) et deux paramètres (facultatifs) :

```
def bonjour(prenom1, prenom2='Lisa', prenom3='Zoe'):
    print('hello', prenom1, ',', prenom2, 'and', prenom3)
```

et comparons plusieurs appels de la fonction :

Appel 1

La fonction est appelée avec trois arguments sans mot-clé, ils sont pris dans l'ordre.

```
>>> bonjour("Tom", "Lea", "Jean")
hello Tom , Lea and Jean
```

Appel 2

La fonction est appelée sans arguments alors qu'elle a un paramètre positionnel obligatoire, il y a une erreur : bug:.

```
>>> bonjour()
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: bonjour() missing 1 required positional argument: 'prenom1'
```

Appel 3

La fonction est appelée avec deux arguments sans mot-clé, ils sont pris dans l'ordre. Le troisième paramètre utilise la valeur par défaut.

```
>>> bonjour("Tom", "Lea")
hello Tom , Lea and Zoe
```

Appel 4

La fonction est appelée avec deux arguments, le premier est positionnel, le second correspondant au mot-clé du troisième paramètre. Le deuxième paramètre utilise la valeur par défaut.

```
>>> bonjour("Tom", prenom3="Lea")
hello Tom , Lisa and Lea
```

Appel 5

La fonction est appelée avec les deux arguments par mot-clé, mais il manque l'argument postionnel obligatoire, il y a une erreur : bug:

```
>>> bonjour(prenom2="Jean", prenom3="Lea")
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: bonjour() missing 1 required positional argument: 'prenom1'
```

Appel 6

La fonction est appelée avec deux arguments, le premier corresponant au mot-clé du troisième paramètre et le second correspond au paramètre positionnel. Il y a une

erreur car les paramètres positionnels doivent être placés avant.

```
>>> bonjour(prenom3="Lea", "Tom")
File "<interactive input>", line 1
bonjour(prenom3="Lea", "Tom")
                       ^
SyntaxError: positional argument follows keyword argument
```

Appel 7

La fonction est appelée avec deux arguments, le premier corresponant au mot-clé du troisième paramètre et le second correspond au paramètre positionnel identifié par son mot-clé. Le deuxième paramètre utilise la valeur par défaut.

```
>>> bonjour(prenom3="Lea", prenom1="Tom")
hello Tom , Lisa and Lea
```

À noter :

> Si une fonction est définie avec des paramètres positionnels et des paramètres par mot-clé, les paramètres positionnels doivent toujours être placés avant les paramètres par mot-clé : Ecrire « `def bonjour (prenom1='Tim', prenom2):` » est incorrect.

5.3. L'instruction `return`

Prenons l'exemple d'une fonction très pratique, `prix` qui permet d'afficher un prix en ajoutant la TVA. Cette fonction a deux paramètres, `prix_ht` le prix hors taxe d'un bien et `tva` le taux de TVA exprimé en pourcent et qui vaut `20` par défaut :

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    print(prix_ttc)
```

Comment afficher le prix d'un article de 100 euros avec 5% de TVA ? C'est très simple, il suffit de l'appeler :

```
>>> prix(100, 5)
105.0
```

Mais comment afficher le prix total de plusieurs articles avec des taux de tva différents ? Par exemple un panier contenant un article de 100 euros à 5% de TVA et un autre article de 50 euros à 20% de TVA ?

```
>>> prix(100, 5) + prix(50)
105.0
60.0
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

Impossible de faire la somme des prix des deux articles ! Cette fonction montre très rapidement ses limites.

Plutôt que d'afficher le prix calculé, il est plus judicieux de le **renvoyer**.

Il n'y a pas de parenthèse à l'instuction `return`.

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    return prix_ttc
```

et d'afficher les prix qui nous intéressent :

```
>>> prix(100, 5)
105.0
>>> prix(100, 5) + prix(50)
165
```

Voyons plus en détail la différence entre les deux fonctions avec `print()` et `return`.

Fonction avec `print()`

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    print(prix_ttc)
```

Fonction avec `return`

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    return prix_ttc
```

```
>>> prix(100, 5)
105.0
```

Alors quelle est la différence ? Elle apparaît immédiatement si on appelle la fonction depuis le programme avec `prix(100, 5)` :

Fonction avec `print()`

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    print(prix_ttc)

prix(100, 5)
```

Le programme affiche `105`.

Fonction avec `return`

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    return prix_ttc

prix(100, 5)
```

Le programme n'affiche rien.

Et si on essaye d'appeller la fonction depuis le programme avec `print(prix(100, 5))` :

Fonction avec `print()`

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    print(prix_ttc)

print(prix(100, 5))
```

Le programme affiche `105` quand `print(prix_ttc)` s'exécute puis `None` quand `print(prix(100, 5))` s'exécute.

Fonction avec `return`

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    return prix_ttc

print(prix(100, 5))
```

Le programme affiche `105` quand `print(prix(100, 5))` s'exécute.

- Avec `print()` la première fonction `prix` **affiche** le résultat calculé dans la console mais ce résultat n'est plus utilisable dans la suite du programme, il est perdu ;
- par contre, avec `return` la seconde fonction **renvoie** le résultat calculé qui peut être utilisé par exemple pour faire d'autres opérations, pour l'affecter à une variable, ou encore comme argument d'une autre fonction, voire même pour être tout simplement affiché comme par exemple `print(prix(100, 5))`.

Appelons `prix(100, 5)` et essayons d'affecter la valeur renvoyée par ces deux fonctions à une variable :

Fonction avec `print()`

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    print(prix_ttc)

p = prix(100, 5)
```

Dans ce cas la variable `p` a la valeur `None`, 🦎 ce n'est probablement pas ce qui était attendu !

Fonction avec `return`

```
def prix(prix_ht, tva=20):
    prix_ttc = prix_ht * (1 + tva/100)
    return prix_ttc
```

```
p = prix(100, 5)
```

Dans ce cas la variable `p` a bien la valeur `105` comme attendu.

Dans le doute, de façon générale, il faut éviter d'afficher un résultat avec `print()` dans une fonction autre que la fonction `main()` et préférer renvoyer le résultat avec `return`.

Un autre point important à noter est qu'une fonction se termine immédiatement dès qu'une instruction `return` est exécutée.

Par exemple dans la fonction `plus_petit(a, b)` suivante[26], qui renvoie le plus petit de deux nombres `a` et `b` :

```
1  def plus_petit(a, b):
2      if a < b:
3          return a
4      else:
5          return b
```

le `else` en ligne 4 est inutile. On peut simplement écrire :

```
1  def plus_petit(a, b):
2      if a < b:
3          return a
4      return b
```

En effet, si `a` est plus petit que `b`, la fonction se termine à la ligne 3 et le dernier `return b` ne sera jamais exécuté.

Pour finir, Une fonction peut aussi renvoyer plusieurs valeurs en même temps, séparées par des virgules, par exemple la fonction `carre_cube(x)` suivante renvoie le carré le cube d'un nombre `x` :

```
def carre_cube(x):
   return x**2, x**3

print(carre_cube(5))
```

affiche `(25, 125)`.

Cours

> Le verbe "renvoyer" est préféré à "retourner" (anglicisme pour *return*).

Une fonction peut **renvoyer** une ou plusieurs valeurs avec l'instruction `return`.

La fonction se termine immédiatement dès qu'une instruction `return` est exécutée. Les instructions suivantes sont ignorées.

À noter :

> S'il n'a pas d'instruction `return` dans une fonction, elle renvoie `None`[27].

Exercice corrigé

Écrire une fonction `est_premier(nombre)` qui renvoie `True` si `nombre` est un nomber premier et `False` sinon.

Rappel : un nombre est premier s'il n'a que deux diviseurs, 1 et lui-même.

Aide

Le fait qu'une fonction se termine immédiatement après une instruction `return` est bien utile dans ce cas.
Pour vérifier si `nombre` est premier, il suffit de tester tous les entiers entre `2` et `n-1` les uns après les autres pour trouver un diviseur autre que `1` et `nombre`. Dès qu'un diviseur est trouvé, inutile de continuer, le nombre n'est pas premier et dans ce cas l'instruction `return False` termine la fonction. Si aucun diviseur n'est trouvé après les avoir tous testés, la fonction se termine en renvoyant `True`.

Réponse

Avec une boucle `for` en testant les entiers allant de 2 à `nombre` (exclus)

```
def est_premier(nombre):
    # Cherche un diviseur entre 2 et nombre-1
    for div in range(2, nombre):
        if nombre % div == 0:
            return False    # div est un diviseur, nombre n'est pas premier, la fonction se termine et renvoie False
    return True   # si aucun diviseur n'a été trouvé alors le nombre est premier, la fonction renvoie True
```

Avec une boucle `while` en testant les entiers allent de 2 à la racine carrée du nombre

```
def est_premier(nombre):
    div = 2
    # Cherche un diviseur entre 2 et la racine carré de nombre
    while div**2 <= nombre:
        if nombre % div == 0:
            return False   # div est un diviseur, nombre n'est pas premier, la fonction se termine et renvoie False
        div = div + 1    # essayons le suivant
    return True   # si aucun diviseur n'a été trouvé alors le nombre est premier, la fonction renvoie True
```

Appelons la fonction `estpremier` avec les arguments 13 et 21 :

Appel `estpremier(13)`

`div` prend les valeurs `2`, `3`, etc. et aucune de ces valeurs n'est un diviseur de `13`, l'instruction conditionnelle `nombre % div == 0` n'est jamais vérifiée, la boucle se termine et la dernière instruction `return True` est exécutée, la fonction se termine.

```
>>> estpremier(13)
True
```

Appel `estpremier(21)`

`div` prend la valeur `2`, ce n'est pas un diviseur de `21` (`21 % 2` est égal à 1), la boucle continue. `div` prend la valeur `3`, c'est pas un diviseur de `21` (`21 % 3` est égal à 0), l'instruction conditionnelle `nombre % div == 0` est vérifiée, donc l'instruction `return False` est exécutée et la fonction se termine, la dernière instruction `return True` n'est jamais exécutée.

```
>>> estpremier(21)
False
```

5.4. Fonction lambda

Cours

En Python, les fonctions lambda sont des fonctions extrêmement courtes, limitées à une seule expression, sans utiliser le mot-clé `def`.

```
nom_de_fonction = lambda param1, param2,…: expression
```

Prenons par exemple une fonction qui ajoute deux valeurs :

```
>>> somme = lambda x, y: x + y
>>> somme(3, 5)
8
```

Ici la fonction lambda est définie par l'expression `lambda x, y: x + y` qui comporte :

- le mot réservé `lambda`,
- suivi de deux paramètres `x` et `y` placés avant les deux-points,
- deux-points `:`,
- l'expression de la valeur renvoyée `x + y`, placée après les deux-points.

Le signe `=` affecte cette fonction à une variable, ici `somme`, c'est le nom de cette fonction.

L'instruction `somme(3, 5)` permet ensuite d'appeler la fonction avec deux arguments `3` et `5`.

Exercice corrigé

Écrire la fonction cube qui renvoie le cube d'un nombre sous formes classique et lambda.

Réponse

```
def cube(y):
   return y**3
```

et

```
cube = lambda y: y**3
```

Réduite à une seule expression, les fonctions lambda permettent d'utiliser une instruction conditionnelle écrite sous une forme un peu différente que vue précedemment :

```
>>> entre_10_et_20 = lambda x: True if (x > 10 and x < 20) else False
>>> entre_10_et_20(5)
False
```

5.5. Portée de variables

Cours

La **portée** d'une variable désigne les endroits du programme où cette variable **existe et peut être utilisée**. En Python, la portée d'une variable commence dès sa première affectation.

Exemple : Les programmes suivants affichent un message d'erreur

Programme 1

```
print(a)
a = 1
```

Ce programme essaie d'afficher la variable `a` avant qu'elle ne soit définie, il affiche un message d'erreur :

```
>>>
Traceback (most recent call last):
File "<module1>", line 1, in <module>
NameError: name 'a' is not defined
```

Programme 2

```
a = a + 1
print(a)
```

Ce programme essaie d'affecter à la variable `a` une valeur calculée en utilisant `a` avant qu'elle ne soit définie, il affiche un message d'erreur :

```
>>>
Traceback (most recent call last):
File "<module1>", line 1, in <module>
NameError: name 'a' is not defined
```

5.5.1. Variables locales

Cours

Une variable définie à l'intérieur d'une fonction est appelée **variable locale**. Elle ne peut être utilisée que localement c'est-à-dire qu'à l'intérieur de la fonction qui l'a définie.

Tenter d'utiliser une variable locale depuis l'extérieur de la fonction qui l'a définie provoquera une erreur.

Exemple :

```
def affiche_a():
    a = 1
    print(f'valeur de a dans affiche_a : {a}')

affiche_a()
print(f'valeur de a dans le programme : {a}')
```

La variable `a` elle est locale à `affiche_a`, le programme suivant affiche un message d'erreur :

```
>>>
Traceback (most recent call last):
  File "<module1>", line 6, in <module>
NameError: name 'a' is not defined
```

5.5.2. Paramètres passés par valeur

Dans les exemples précédents (`est_premier(13`), etc.), les arguments utilisés en appelant les fonctions étaient des valeurs.

Les arguments utilisés dans l'appel d'une fonction peuvent aussi être des variables ou même des expressions. Les trois appels de fonctions suivants font le même chose :

```
>>> est_premier(13)
True
>>> a = 13
>>> est_premier(a)
True
>>> nombre = 6
>>> est_premier(2*nombre + 1)
True
```

Quand un argument de fonction est une variable (ou une expression contenant une variable), par exemple dans le cas de `est_premier(a)`, c'est la valeur de cette variable (ou de cette expression) qui est passée au paramètre correspondant de la fonction. On dit que le paramètre est « passé par valeur ». Des modifications éventuelles de ce paramètre dans la fonction ne modifient pas la valeur de la variable qui a servit d'argument à la fonction. Et c'est le cas même quand le nom de la variable est identique au nom du paramètre de la fonction, c'est seulement sa valeur qui est passée à la fonction.

Cours

Une fonction ne peut pas modifier la valeur d'une variable passée en paramètre en dehors de son exécution. **Les paramètres sont passés par valeur**.

Exemple :

```
def ajoute_1(a):
    a = a + 1
    print(f'valeur de a dans ajoute_1 : {a}')

a = 1
ajoute_1(a)
print(f'valeur de a dans le programme : {a}')
```

La valeur de a est modifiée en `2` à l'intérieur de la fonction `ajoute_1` pendant son exécution, mais pas dans le programme où elle garde sa valeur initiale de `1`.

```
>>>
valeur de a dans ajoute_1 : 2
valeur de a dans le programme : 1
```

5.5.3. Variables globales

Sauf exception il est préférable d'utiliser uniquement des variables locales pour faciliter la compréhension des programmes et réduire l'utilisation de mémoire inutile, mais dans certains cas leur portée n'est plus suffisante.

Cours

Une variable définie en dehors de toute fonction est appelée **variable globale**. Elle est utilisable à travers l'ensemble du programme.

Elle peut être affichée par une fonction :

```
def affiche_a():
    print(f'valeur de a dans affiche_a : {a}')

a = 1
affiche_a()
```

Mais ne peut pas être modifiée.

```
def affiche_a():
    a += 1
    print(f'valeur de a dans affiche_a : {a}')

a = 1
affiche_a()
```

Ce programme affiche un message d'erreur 🐛 :

```
>>>
Traceback (most recent call last):
  File "<module1>", line 6, in <module>
  File "<module1>", line 2, in affiche_a
UnboundLocalError: local variable 'a' referenced before assignment
```

On peut néanmoins essayer de lui assigner une nouvelle valeur :

```
def affiche_a():
    a = 2
    print(f'valeur de a dans affiche_a : {a}')

a = 1
affiche_a()
print(f'valeur de a dans le programme : {a}')
```

Mais dans ce cas, Python part du principe que `a` est locale à la fonction, et non plus une variable globale. L'instruction `a = 2` a créé un nouvelle variable locale à la fonction, la variable globale n'a pas changé :

```
>>>
valeur de a dans affiche_a : 2
valeur de a dans le programme : 1
```

Dans certaines situations, il serait utile de pouvoir modifier la valeur d'une variable globale dans une fonction et que cette nouvelle valeur soit gardée dans le reste du programme. Pour cela, il faut utiliser le mot clef `global` devant le nom d'une variable globale utilisée localement afin d'indiquer qu'il faut modifier la valeur de la variable globale et non pas créer une variable locale de même nom :

```
def affiche_a():
    global a
    a = 2
    print(f'valeur de a dans affiche_a : {a}')

a = 1
affiche_a()
print(f'valeur de a dans le programme : {a}')
```

La variable `a` a été modifiée dans la fonction et que sa nouvelle valeur est gardée dans le reste du programme :

```
>>>
valeur de a dans affiche_a : 2
valeur de a dans le programme : 2
```

Cours

Une variable globale est accessible uniquement en lecture à l'intérieur des fonctions du programme. Pour la modifier il faut utiliser le mot-clé `global`.

5.6. Fonction récursive

Une fonction peut être appelée n'importe où dans un programme (après sa définition), y compris par elle-même.

Cours

Une fonction récursive est une fonction qui peut s'appeler elle-même au cours de son exécution.

Prenons pour exemple une fonction qui renvoie le produit de tous les nombres entiers entre 1 et n. Ce produit est appelé factorielle de n et noté $n!$.

$$n! = 1 \times 2 \times 3 \times 4 \times \ldots \times (n-1) \times n$$

Une simple boucle `for` permet de multiplier tous les entiers allant de `1` à `n` :

```
def factorielle(n):
    fact = 1
    for i in range(1, n + 1):
        fact = fact * i
    return fact
```

Mais il est aussi possible de remarquer que $n! = (n-1)! \times n$ et que $1! = 1$, ce qui permet d'écrire un programme récursif suivant :

```
def factorielle_recursive(n):
    if n == 1:
        return 1
```

```
    else:
        return  factorielle_recursive(n-1) * n         # le else est facultatif
```

⚠️ Il est impératif de prévoir une condition d'arrêt à la récursivité, sinon le programme ne s'arrête jamais ! On doit toujours tester en premier la condition d'arrêt, et ensuite, si la condition n'est pas vérifiée, lancer un appel récursif.

Exercice corrigé

Écrire une fonction récursive `compte_a_rebours(n)` qui affiche les nombres entiers à rebours allant de `n` à `0`.

Réponse

```
def compte_a_rebours(n):
    if n < 0:
        pass
    else:
        print(n)
        compte_a_rebours(n-1)
```

ou plus simplement :

```
def compte_a_rebours(n):
    print(n)
    if n > 0:
        compte_a_rebours(n-1)
```

5.7. Modules et bibliothèques

Nous avons vu dans l'exercice précédent comment écrire un programme qui affiche la décomposition d'un nombre en facteurs premiers en utilisant la fonction `est_premier`.

```
def est_premier(nombre):
    for div in range(2, nombre):
        if nombre % div == 0:
            return False
    return True

def main():
    nombre = int(input('Entrez un nombre '))
    premier = 2 # on commence par le plus petit nombre premier : 2
    while nombre > 1:
        if nombre % premier == 0:      # si premier divise nombre
            print(premier, end=" ")                # alors on l'affiche
            nombre = nombre // premier     # et on recommence après avoir divisé nombre par premier
        else:                          # sinon, premier n'est pas un diviseur
            premier += 1                 # on cherche le nombre premier suivant
            while not(est_premier(premier)):
                premier += 1


if __name__ == '__main__':
    main()
```

Rappel

La fonction `est_premier` doit être est écrite au début du programme, elle doit être définie **avant** d'être appelée.

Ici le programme est écrit dans le même fichier Python que la fonction `est_premier`. Mais que se passerait-il si un programme écrit dans un autre fichier Python appelle cette même fonction ?

Ecrivons un nouveau programme Python qui appelle `est_premier` :

```
n = int(input('Entrez un nombre'))
print(est_premier(n))
```

Il affiche une erreur 🐛 :

```
Traceback (most recent call last):
  File "<module2>", line 2, in <module>
NameError: name 'est_premier' is not defined
```

Alors comment faire pour appeler `est_premier` depuis ce nouveau programme sans la réécrire ? Il faut créer un module.

Enregistrons la fonction `est_premier` dans un fichier qu'on appelle « mesfonctions.py ». ⚠️ Le fichier « mesfonctions.py » doit être **enregistré dans le même répertoire** que le nouveau programme. Nous avons créé le module `mesfonctions`.

Testons à nouveau le programme principal. Toujours la même erreur. Le module `mesfonctions` doit d'abord être importé pour utiliser les fonctions qu'il contient.

Cours

Un **module** est un programme Python regroupant des fonctions et des constantes (des variables dont la valeur ne change pas[28]) ayant un rapport entre elles.

Un module doit être **importé** dans un programme avant de pouvoir utiliser ses fonctions et ses constantes.

Une **bibliothèque** (ou *package* en anglais) regroupe des fonctions, des constantes et des **modules**.

5.8. `import`

Pour importer un module dans un programme, il faut utiliser l'instruction `import` en début de programme :

```
import mesfonctions
```

Pour appeler une fonction ou utiliser une constante d'un module il faut écrire le nom du module suivi d'un point « . » puis du nom de la fonction ou de la constante.

```
1  import mesfonctions
2
3  n = int(input('Entrez un nombre '))
4  print(mesfonctions.est_premier(n))
```

⚠️ Prendre soin d'enregistrer ce programme dans le même répertoire que le fichier « mesfonctions.py ».

Il est aussi possible donner un alias à un module pour le renommer avec un nom plus simple à écrire, par exemple pour utiliser `mf` au lieu de `mesfonctions` :

```
1  import  mesfonctions  as mf
2
3  n = int(input('Entrez un nombre '))
4  print(mf.est_premier(n))
```

Cours

Sans alias

Pour importer un module `nom_module`, il faut écrire en début de programme l'instruction :

`import nom_module`

puis pour utiliser une fonction `nom_fonction()` de ce module :

`nom_module.nom_fonction()`

Avec alias

Pour importer un module `nom_module` en lui donnant un alias `nom_alias`, il faut écrire en début de programme l'instruction :

`import nom_module as nom_alias`

puis pour utiliser une fonction `nom_fonction()` de ce module :

`nom_alias_module.nom_fonction()`

La fonction `help()` permet de savoir ce que contient un module :

```
>>> help(mesfonctions)
Help on module mesfonctions:
…
```

5.9. `from … import …`

Il existe une autre méthode pour importer des fonctions ou constantes depuis un module. Admettons que le module `mesfonctions` contienne des dizaines de fonctions, mais que nous ayons uniquement besoin dans notre programme de la fonction `est_premier`, dans ce cas il est préférable d'importer uniquement cette fonction plutôt que tout le module en utilisant l'instruction `` `from mesfonctions import est_premier ``.

```
1  from  mesfonctions import est_premier
2
3  n = int(input('Entrez un nombre '))
4  print(est_premier(n))
```

À noter :

> Ici on ne met pas le préfixe « `mesfonctions.` » devant le nom de la fonction `est_premier`.

On peut aussi donner un alias à une fonction : `from mesfonctions import est_premier as estprems` et utiliser ensuite la fonction `estprems()`.

Cours

Sans alias

Pour importer une fonction `nom_fonction()` depuis le module `nom_module`, il faut écrire en début de programme l'instruction :

`from nom_module import nom_fonction`

puis pour utiliser cette fonction :

`nom_fonction()`

Avec alias

Pour importer une fonction `nom_fonction()` en lui donnant un alias `nom_alias` depuis le module `nom_module`, il faut écrire en début de programme l'instruction :

`from nom_module import nom_fonction as nom_alias`

puis pour utiliser cette fonction :

`nom_alias()`

Il est aussi possible d'importer plusieurs fonctions d'un même module séparées par des virgules :

```
from mesfonctions import est_premier, une_autre_fonction
```

voire même toutes les fonctions d'un module en tapant « `*` » à la place du nom de la fonction à importer.

```
from mesfonctions import *
```

Mais cette dernière utilisation est vivement déconseillée, hormis dans des cas très particuliers par exemple des programmes très courts, car il peut y avoir des conflits entre des fonctions qui ont le même nom. Pour s'en convaincre, imaginons un programme écrit en utilisant l'instruction `pow(1, 2, 3)` qui fonctionnerait parfaitement jusqu'à ce qu'une modification necessitant le module `math` ajoute l'instruction `from math import *` en début de programme et génère une erreur inattendue [29] là où il n'y en avait pas.

Python offre des centaines de modules avec des milliers de fonctions déjà programmées. Il y a différents types de modules :

- ceux que l'on peut faire soi-même (comme `mesfonctions`).
- ceux qui sont inclus dans la bibliothèque standard de Python comme `random` ou `math`,
- ceux que l'on peut rajouter en les installant séparemment comme `numpy` ou `matplotlib`.

5.10. De l'utilité de la fonction 'main()'

On a vu auparavant la définition de la fonction `main()` contenant le programme principal, suivi du bout de code suivant :

```
if __name__ == '__main__':
    main()
```

Cette instruction conditionnelle vérifie si une variable appelée `__name__` est égale à `'__main__'` et dans ce cas exécute la fonction `main()`.

L'interpréteur Python définit la variable `__name__` selon la manière dont le code est exécuté :

- directement en tant que script, dans ce cas Python affecte `'__main__'` à `__name__`, l'instruction conditionnelle est vérifiée et la fonction `main()` est appelée ; ou alors
- en important le code dans un autre script et dans ce cas la fonction `main()` n'est pas appelée.

En bref, la variable `__name__` détermine si le fichier est exécuté directement ou s'il a été importé.[30]

5.11. Le module math

Le module `math` permet d'avoir accès aux fonctions mathématiques, par exemple les fonctions cosinus (`cos`), sinus (`sin`), racine carrée (`sqrt`), le nombre π (`pi`), la partie entière (`floor`)[31], etc.

```
>>> import math
>>> math.cos(math.pi)            # cosinus d'un angle en radian
-1.0
```

```
>>> math.sqrt(25)           # racine carrée
5.0
```

5.12. Le module random

Le module `random` permet d'utiliser des fonctions générant des nombres aléatoires. Deux fonctions très utiles sont :

- `random()` qui renvoie un nombre aléatoire entre 0 et 1, et
- `randint(a, b)` qui renvoie au hasard un nombre entier compris entre `a` et `b` **inclus**.

```
>>> from random import random, randint
>>> random()
0.34461947461259612
>>> randint(1, 2)
2
```

5.13. Le module pyplot

La bibliothèque `mathplotlib` contient le module `pyplot` (utilisé pour tracer des courbes). Pour importer ce module, l'utilisation d'un alias est particulièrement utile dans ce cas.

```
import matplolib.pyplot as plt
```

`pyplot` permet d'afficher des données sous de multiples formes. Par exemple pour afficher un point de coordonnées (2, 4) sous forme d'une croix verte dans un repère :

```
plt.plot(2, 4, 'g+')
plt.show()
```

5.14. D'autres modules

Il y en a beaucoup d'autres, tant dans la nature (https://github.com/search?q=python+module) que dans la bibliothèque standard (http://docs.python.org/3/py-modindex.html),

module	Description
`numpy`	Permet de faire du calcul scientifique.
`time`	Fonctions permettant de travailler avec le temps.
`turtle`	Fonctions de dessin.
`doctest`	Execute des tests ecrits dans la docstring d'une fonction.

La fonction `dir` permet d'explorer le contenu d'un module :

```
>>> import math
>>> dir(math)
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
…
```

En plus de la documentation en ligne, la fonction `help` donne les spécifications d'une fonction.

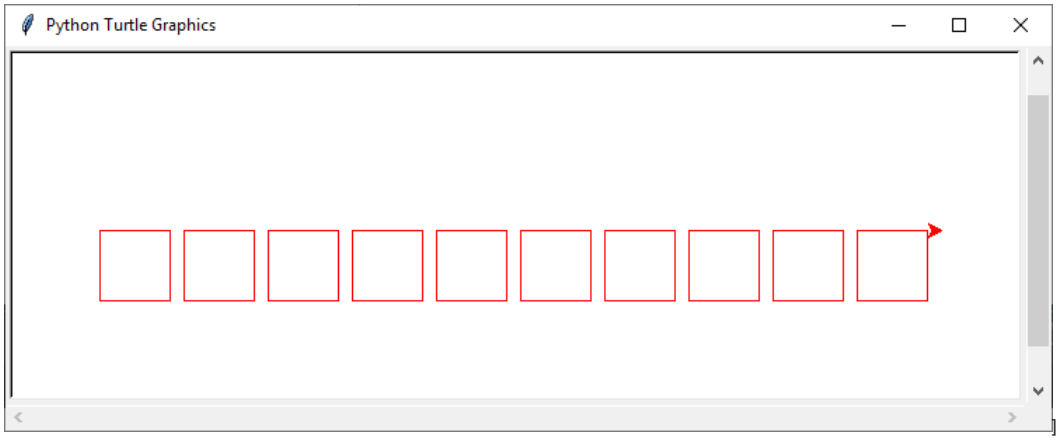
```
>>> help(math.cos)
Help on built-in function cos in module math:

cos(...)
    cos(x)

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

Exercice corrigé

Réaliser le dessins suivant (dix carrés rouges de 50 pixels de coté), à l'aide du module turtle. [



Réponse

```
import turtle

def carre(taille, couleur):
    turtle.pendown()
    turtle.color(couleur)
    for _ in range(4):
        turtle.forward(taille)
        turtle.right(90)
    turtle.penup()

turtle.penup()
turtle.goto(-300,0)
for c in range(10):
    carre(50,'red')
    turtle.forward(60)
```

6. Mise au point des programmes et gestion des bugs

Comment s'assurer qu'un programme fasse ce qu'il est censé faire ? Qu'il ne contient pas de bugs ? Ces questions que chacun se pose quand il écrit un programme peuvent devenir extrêmement cruciales et compliquées quand certains programmes informatiques contiennent des millions de lignes de code, voire des milliards (Google)[32] ou avoir des défauts de fonctionnements aux conséquences désastreuses (avionique, nucléaire, médical, etc.). Des solutions existent pour essayer de limiter ces effets néfastes.

L'utilisation combinée de **spécifications**, d'**assertions**, de **documentations des programmes** et de **jeux de tests** permettent de limiter (mais pas de garantir ! [33]) la présence de bugs dans les programmes.

6.1. Bugs (ou bogues) et exceptions

Il existe de nombreuses causes qui peuvent être à l'origine de bugs dans un programme : oubli d'un cas[34], typo, dépassement de capacité mémoire[35], mauvaise communication avec les utilisateurs ou entre programmeurs, etc.

Cours

Un bug (ou bogue) est une erreur dans un programme à l'origine d'un dysfonctionnement.

Un bug peut conduire à un résultat qui n'est pas celui attendu, par exemple si `est_premier(5)` renvoyait False, voire même dans certains cas à une exception (mais ce n'est pas toujours le cas).

Cours

Une exception est une erreur qui se produit pendant l'exécution du programme. Lorsqu'une exception se produit (on dit que l'exception est 'levée'), l'exécution normale du programme est interrompue et l'exception est traitée.

```
num1, num2 = 7, 0
print(num1/num2)
>>>
ZeroDivisionError : division by zero
```

Une bonne façon de gérer les exceptions est de comprendre les différents types d'erreurs qui surviennent et pourquoi elles se produisent. Soyons attentifs aux messages d'erreur que nous affiche l'interpréteur, ils sont d'une grande utilité[36]. En voici certains parmi les plus courants :

- `SyntaxError` : Une ligne de code non valide empêche le programme de s'exécuter.

```
>>> print("Hello World)
  File "<interactive input>", line 1
    print("Hello World)
          ^
SyntaxError: incomplete input
```

- `IndentationError` : Une mauvaise indentation ne permet pas de définir les blocs de code correctement[37].

```
for i in range(10):
print(i)
IndentationError: expected an indented block after 'for' statement on line 1
```

- `TypeError` : Une opération utilise des types de données incompatibles.

```
>>> "abc" + 2
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

- `ValueError` : Une fonction est appelée avec une valeur d'argument non autorisée.

```
>>> int('abc')
ValueError: invalid literal for int() with base 10: 'abc'
```

Pour corriger les bugs et exceptions inévitables lorsqu'on écrit un programme, le débogueur est un outil très utile.

Cours

Le débogueur permet d'effectuer l'exécution ligne par ligne en observant l'évolution du programme et les valeurs des variables.

Pour utiliser le débogueur de PyScripter :

1. Créer un point d'arrêt sur une ligne (clic sur le numéro de la ligne), ou plusieurs.
2. Lancer le débogage () ce qui exécute le script jusqu'au point d'arrêt
3. Exécuter le script pas à pas tout en inspectant l'évolution des variables dans les onglets Variables ou Watches (surveillances). Pour ajouter une variable à surveiller, cliquer droit dans la fenêtre Watches et ajouter un nom de la variable ou une expression.

6.2. Commentaires, noms de variables et de fonctions

Il est difficile de dire ce que fait cette fonction au premier coup d'oeil :

```
1  def f(x):
2      a = 0
3      for i in range(1, x):
4          if x % i == 0:
5              a = a + i
6      return x == a
```

PEP 8

Ecrire les noms tout en minuscule avec des mots séparés par des blancs soulignés (_) par exemple `nom_de variable` (*snake case*) plutôt que `NomDeVariable` (*camel case*)

C'est plus déjà plus lisible avec des noms de fonction et variable qui ont un sens plutôt que réduits à une lettre.

```
1  def parfait(nombre):
2      somme_diviseurs = 0
3      for i in range(1, nombre):
4          if nombre % i == 0:
5              somme_diviseurs = somme_diviseurs + i
6      return nombre == somme_diviseurs
```

et encore plus lisible avec des commentaires :

```
1  def parfait(nombre):
2      somme_diviseurs = 0
3      # Iterer sur tous les entiers i compris entre 1 et nombre - 1
```

```
    for i in range(1, nombre):
        # Si i est un diviseur de nombre on l'ajoute à somme_diviseur
        if nombre % i == 0:
            somme_diviseurs = somme_diviseurs + i
    # Si nombre est egal à la somme de ses diviseurs, c'est un nombre parfait
    return nombre == somme_diviseurs
```

Cours

PEP 8

Limiter la longueur des lignes à 79 ou 80 caractères.

Écrire un beau code Python implique d'adopter certaines conventions et bonnes pratiques pour le rendre clair, lisible, maintenable et compréhensible pour vous-même et pour les autres développeurs[38] :

1. Choisir des noms de variables et de fonctions significatifs et éviter les noms génériques comme "a", "b", "x", etc. qui ne donnent pas d'indication sur leur contenu. Préférer des noms descriptifs comme "somme_diviseurs" plutôt que "sd".
2. Commenter le code pour expliquer les parties importantes, les décisions de conception, les algorithmes, etc. Les commentaires doivent être clairs, concis et utiles. N'ajoutez pas de commentaires évidents qui ne font que répéter le code.

```
# Commentaire sur un bloc
instruction          # Commentaire sur une instruction particuliere
```

3. Eviter les répétition de code et diviser les programmes en fonctions logiques, plus modulaires, plus faciles à comprendre et à déboguer.

6.3. Spécifications de fonctions

Cours

La **spécification** (ou **prototype**) d'une fonction est un mode d'emploi à l'attention des utilisateurs d'une fonction expliquant clairement :

- ce que fait la fonction,
- les paramètres qu'elle accepte,
- les valeurs qu'elle renvoie.

En python, la spécification est résumée dans la « **docstring** », un commentaire au début du corps de la fonction entre **tripe guillemets** (ou **triple apostrophes**) :

Par convention, les `"""` de fin sont seuls sur la dernière ligne.

```
def nom_dela_fonction (parametres):
""" spécification de la fonction écrit entre triple guillemets comprenant :
- ce que fait la fonction,
- les paramètres qu'elle accepte,
- les valeurs qu'elle renvoie.
"""
```

Si l'idée générale est toujours la même, aucun format n'est imposé même si certaines conventions sont données dans la PEP 257. En pratique, il existe différentes habitudes d'écrire les *docstrings* de fonctions et il est important de rester consistant à travers un même programme pour améliorer la lisibilité du code.

Par exemple, la fonction précédente `parfait(nombre)` pourrait se présenter sous la forme :

```
def parfait(nombre):
    """ (int) -> bool
    Renvoie True si nombre est parfait, False sinon
    """

    somme_diviseurs = 0
    # Itérer sur tous les entiers i compris entre 1 et nombre - 1
    for i in range(1, nombre):
        # Si i est un diviseur de nombre on l'ajoute à somme_diviseur
        if nombre % i == 0:
            somme_diviseurs = somme_diviseurs + i
    # Si nombre est egal à la somme de ses diviseurs, c'est un nombre parfait
    return nombre == somme_diviseurs
```

ou encore :

```
def parfait(nombre):
    """ Renvoie True si nombre est parfait, False sinon
    Parameters:
        nombre (int): un nombre entier.
```

```
    Returns:
        bool: True si nombre est parfait, False sinon.
    """
```

ou plus simplement sur une seule ligne (dans ce cas les `"""` sont écrits sur la même ligne):

```
def parfait(nombre):
    """ Renvoie True si nombre (int) est parfait, False sinon """
```

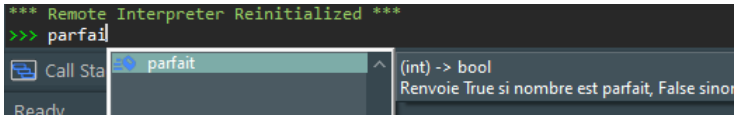
La fonction `help` affiche la *docstring* d'une fonction :

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

est_premier(nombre)
    (int) -> bool
    Renvoie True si un nombre est parfait, False sinon
```

⚠️ Ne pas confondre la spécification encadrée par `"""` avec les commentaires qui commencent par `#`. D'ailleurs il est possible d'ajouter des commentaires commençant par `#` dans une *docstring* qui ne seront pas affichés par `help`. La spécification sera lue par le programmeur qui utilise la fonction, les commentaires par celui qui lira et modifiera le code de la fonction.

Remarquer l'affichage dans la console ou dans la zone de programme PyScriter la spécification qui s'affiche après avoir saisi le nom de la fonction, par exemple `parfait(`.



6.4. Préconditions, postconditions

Testons la fonction `parfait(nombre)` avec un exemple simple :

```
>>> parfait(13)
False
```

Que se passe t'il maintenant si un argument qui n'est pas un entier est passé à la fonction `parfait` ?

```
>>> parfait(13.0)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "<module2>", line 8, in parfait
TypeError: 'float' object cannot be interpreted as an integer
```

`nombre` doit impératif être de type entier. C'est une **précondition** de la fonction. Il est souvent recommandé d'indiquer les préconditions dans la *docstring* de la fonction pour limiter les risques d'erreur.

```
def parfait(nombre):
    """ (int) -> bool
    Precondition : nombre est de type int et positif
    Fonction qui renvoie True si nombre est parfait, False sinon
    """
```

Cours

Les **préconditions** sont des conditions qui doivent être vraies avant l'exécution d'une fonction pour garantir que celle-ci fonctionne correctement. Si une précondition échoue, cela signifie que l'appel de la fonction n'était pas correct.

Les **postconditions** sont des conditions qui doivent être vraies après l'exécution d'une fonction. Elles permettent de vérifier si la fonction s'est exécutée correctement et a donné les résultats attendus.

Par exemple : - une précondition à une fonction effectuant une division est de s'assurer que le dénominateur est non nul,

- une postcondition à une fonction renvoyant la valeur absolue d'un nombre est de vérifier que la fonction renvoie une valeur positive.

Les **preconditions** et **postconditions** peuvent être indiquées dans la *docstring* ou vérifiées par des assertions.

6.5. Variant et invariant de boucle

Cours

Un **variant de boucle** permet de s'assurer qu'une boucle se terminera.

Mais il ne vérifie pas qu'un algorithme fournit la réponse attendue.

Prenons un exemple. Une fonction de division euclidienne de deux entiers positifs `n` par `d` peut s'écrire de la manière suivante :

```
1  def division(n, d) :
2      q, r = 0, n
3      while r >= d:
4          q = q + 1
5          r = r - d
```

A noter que `d > 0` est une précondition qui doit être vérifiée au début de la fonction. Si ce n'est pas cas et que `d` ≤ 0 alors la boucle ne se terminera jamais !

Ici le variant de boucle est `r`. A chaque passage dans la boucle il diminue de `d` (d est positif) donc la condition `r >= d` finira par ne plus être vérifiée, la boucle se terminera.

Cours

Un invariant de boucle est une propriété ou une expression :

– qui est vraie avant d'entrer dans la boucle ;

– qui reste vraie après chaque itération de boucle ;

– et qui, conjointement à la condition d'arrêt, permet de montrer que le résultat attendu est bien le résultat calculé.

Ici l'invariant de boucle est la propriété: `n == q * d + r`. Prenons en exemple `n = 13` et `d = 3` et observons les états successifs du programme au début de chaque instruction. Au début de la ligne 2, les valeurs de `q` et `r` ne sont pas spécifiées, donc la condition `r >= d` ne peut être évaluée, la prochaine instruction à exécuter est la ligne 3 :

```
(ligne)   q       r       r >= d     (ligne suivante)     n == q * d + r
=========================================================================
  2       -       -         -              3                  -
```

Au début de la ligne 3, `q` et `r` ont pris les valeurs `0` et `13`, la condition `r >= d` est vérifiée, le programme entre dans la boucle et la prochaine instruction à exécuter est la ligne 4. Complétons la table.

```
(ligne)   q       r       r >= d     (ligne suivante)     n == q * d + r
=========================================================================
  2       -       -         -              3                  -
  3       0       13       True            4              VRAI (entrée dans la boucle)
```

Au début de la ligne 4, les valeurs de `q` et `r` sont inchangées, la condition `r >= d` reste donc vérifiée, l'instruction suivante est 5. Complétons ainsi la table jusqu'à la fin du programme :

```
(ligne)   q       r       r >= d     (ligne suivante)     n == q * d + r
=========================================================================
  2       -       -         -              3                  -
  3       0       13       True            4              VRAI (entrée dans la boucle)
  4       0       13       True            5              VRAI
  5       1       13       True            2              FAUX
  3       1       10       True            4              VRAI (retour dans la boucle)
  4       1       10       True            5              VRAI
  5       2       10       True            3              FAUX
  3       2        7       True            4              VRAI (retour dans la boucle)
  4       2        7       True            5              VRAI
  5       3        7       True            3              FAUX
  3       3        4       True            4              VRAI (retour dans la boucle)
  4       3        4       True            5              VRAI
  5       4        4       True            3              FAUX
  3       4        1       False     sortie de boucle     VRAI
```

il n'y a pas unicité de variant ni d'invariant de boucle.

Observons que la propriété `n == q * d + r` reste vraie à chaque retour dans la boucle, même si elle n'est pas toujours vraie au milieu de la boucle. Elle est aussi vraie en sortie de boucle et permet de s'assurer que le résultat calculé est celui attendu.

Exercice corrigé

On considère la fonction `palindrome` suivante :

```
1  def palindrome(mot):
2      """ Renvoie True si mot est un palindrome, False sinon """
3      i = 0
4      j = len(mot) - 1
5      while i <= j:
6      if mot[i] == mot[j]:
7          i = i + 1
8          j = j - 1
9      else:
10         return False
11     return True
```

1. Décrire l'évolution des valeurs des variables le fonctionnement de l'algorithme précédent pour le mot `"radar"`.
2. Montrer que `j - i` est un variant de boucle. En déduire que la fonction `palindrome` se termine.
3. Montrer que `i + j == len(mot) - 1` est un invariant de boucle.

Réponse 1

La table suivante montre les états successifs du programme avec le mot `"radar"`.

```
(ligne)   i     j     i <= j   mot[i]   mot[j]   j - i     i + j    (ligne suivante)
================================================================================
  4       0     -       -        r         -       -         -       (5)
  5       0     4     True       r         r       4         4       (6) entrée dans la boucle
  6       0     4     True       r         r       4         4       (7) mot[i] == mot[j]
  7       0     4     True       r         r       4         4       (8)
  8       1     4     True       a         r       3         5       (5)
  5       1     3     True       a         a       2         4       (6) entrée dans la boucle
  6       1     3     True       a         a       2         4       (7) mot[i] == mot[j]
  7       1     3     True       a         a       2         4       (8)
  8       2     3     True       d         a       1         5       (5)
  5       2     2     True       d         d       0         4       (6) entrée dans la boucle
  6       2     2     True       d         d       0         4       (7) mot[i] == mot[j]
  7       2     2     True       d         d       0         4       (8)
  8       3     2     False      a         d      -1         5       (5)
  5       3     1     False      a         d      -2         4       (11) sortie de la boucle
```

Réponse 2

A chaque itération `j - i` diminue de 2 , c'est un **variant de boucle** qui finira par devenir négatif, autrement dit la condition `i <= j` deviendra fausse et la boucle s'arrêtera (à moins qu'elle se termine plus tôt si mot n'est pas un palindrome), donc **le programme se terminera**.

Il est aussi possible de le démontrer formellement. Supposons que l'on rentre dans la boucle à la ligne 5 avec `i` et `j` ayant des valeurs appelées x et y. `j - i` est égal à $y - x$. Après les lignes 7 et 8, `i` devient égal à $x + 1$ et `j` à $y - 1$, donc `j - i` devient bien égal à $(y - 1) - (x + 1) = y - x - 2$. `j - i` a bien diminué de 2.

Réponse 3

De la même façon, il est possible de démontrer que `i + j == len(mot) - 1` est un invariant de boucle :

- Au début du programme, avant de rentrer dans la boucle, `i` est égal à 0 et j est égal à `len(mot)- 1` donc `i + j` est bien égal à `len(mot) - 1`.
- Lorsqu'on rentre dans la boucle à la ligne 5 avec `i` et `j` ayant des valeurs x et y telles que $x + y$ est égal à `len(mot) - 1` , après les lignes 7 et 8, `i` devient égal à $x + 1$ et `j` à $y - 1$, donc `i + j` est toujours égal à $(x + 1) + (y - 1) = x + y$ c'est-à-dire à `len(mot) - 1`.

6.6. Assertions

Des assertions permettent de tester les préconditions, postconditions et les invariants de boucles. Leur non-respect alerte sur une erreur de programmation.

Cours

Une **Assertion** vérifie qu'une expression est *vraie* et arrête le programme sinon .

```
assert <condition>
```

Un message peut être affiché quand une assertion est **fausse** avant d'arrêter le programme :

```
assert <condition>, 'message '
```

Reprenons la fonction `est_premier(nombre)` vue précédemment. Le paramètre `nombre` doit être de type entier et positif. Ce sont des préconditions. Ajoutons les assertions correspondantes au début de la fonction.

```
def est_premier(nombre):
    """ (int) -> bool
    Precondition : nombre est de type int et positif
    Renvoie True si nombre est premier, False sinon
    """
    assert type(nombre) == int
    assert nombre >= 0, 'nombre doit être positif'
    # Cherche un diviseur entre 2 et nombre-1
    for d in range(2, nombre):
        if nombre%d == 0:   # d divise nombre donc nombre n'est pas premier
            return False
    # Pas diviseur entre 2 et n-1, donc nombre est premier
    return True
```

et testons le résultat :

```
>>> est_premier('5')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "….", line 16, in est_premier
    assert(type(nombre) == int)
AssertionError

>>> est_premier(-1)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "….", line 7, in est_premier
    assert nombre >= 0, 'nombre doit être positif'
AssertionError: nombre doit être positif
```

`assert` est souvent utilisé en phase de test seulement ou en **programmation défensive**[39].

L'instruction `try...…except...` (hors programme) permet de gérer efficament les erreurs prévisibles d'utilisateur, lors d'une saisie par exemple :

```
while True:
    try:
        n = input("Entrez un nombre entier ")
        n = int(n)
        break
    except ValueError:
        print(n, "n'est pas un entier, essayer à nouveau ...")
print(n, "est bien un nombre entier")
```

6.7. Jeux de tests

Les spécifications et les vérifications des pré et postconditions d'un programme ne garantissent pas l'absence de bugs. Avant de pouvoir utiliser un programme, il est important d'effectuer un jeu de tests pour déceler d'éventuelles erreurs.

Cours

Un jeu de test permet de trouver d'éventuelles erreurs. Le succès d'un jeu de tests ne garantit pas qu'il n'y ait pas d'erreur.

La **qualité** et le **nombre** de tests sont importants.

6.7.1. La qualité des tests

Cours

Les tests doivent porter sur des valeurs d'arguments "normales" mais aussi des valeurs "spéciales" ou "extrêmes" du programme.

Par exemple, que se passe-t-il quand les valeurs `0` ou `1` sont passées en argument à la fonction `est_premier` ?

```
>>> est_premier(0)
True
>>> est_premier(1)
True
```

Mais `0` et `1` ne sont pas des nombres premiers ! Il faut donc corriger la fonction en ajoutant ces cas qui avaient été oubliés.

```
def est_premier(nombre):
    """ (int) -> bool
```

```
    Precondition : nombre est de type int et positif
    Renvoie True si nombre est premier, False sinon
    """
    assert type(nombre) == int
    assert nombre >= 0, 'nombre doit être positif'
    # 0 et 1 ne sont pas premiers
    if (nombre == 0) or (nombre == 1):
        return False
    # Cherche un diviseur entre 2 et nombre-1
    for d in range(2, nombre):
        if nombre%d == 0:   # d divise nombre donc nombre n'est pas premier
            return False
    # Pas diviseur entre 2 et n-1, donc nombre est premier
    return True
```

6.7.2. Le nombre de tests

Cours

Un **programme de test** permet d'effectuer **un grand nombre de tests automatiquement**.

Vérifions par des assertions la fonction `est_premier` pour tous les multiples de 2 allant de 4 à 100.

```
def test_est_premier():
    """Jeu de tests de est_premier() pour tous les multiples de 2 entre 4 et 100 """
    for i in range(2, 51):
        assert not est_premier(2 * i)
    return True
```

Il est aussi possible d'écrire un programme de tests en utilisant la célèbre formule d'Euler : $n^2 + n + 41$ qui produit de nombreux nombres premiers, notamment pour tous les nombres n allant de 0 à 39.

```
def test2_est_premier():
    """Jeu de tests de est_premier() par la formule d'Euler 2**2+n+41"""
    for i in range(40):
        assert est_premier(i**2 + i + 41)
    return True
```

6.7.3. Le module doctest

La fonction `tesmod()` du module `doctest` permet d'effectuer automatiquement un jeu de tests défini dans la docstring d'une fonction. Chaque test à effectuer est indiqué dans la *docstring* sur une ligne commençant par `>>>` pour simuler la console et le résultat attendu dans la ligne suivante.

Par exemple :

```
import doctest

def est_premier(nombre):
    """ (int) -> bool
    Precondition : nombre est de type int et positif
    Renvoie True si nombre est premier, False sinon
    >>> est_premier(3)
    True
    >>> est_premier(4)
    False
    """
…

doctest.testmod()
```

---

1. https://fr.wikipedia.org/wiki/Liste_de_langages_de_programmation. ↩
2. Nommé en hommage à la série britannique *Monty Python Flying Circus*. ↩
3. en 2023. ↩
4. Le terme « informatique » résulte de l'association du terme « information » au suffixe « -ique » signifiant « qui est propre à ». ↩
5. La notion de variable en informatique diffère des mathématiques. En mathématique une variable apparait dans l'expression symbolique d'une fonction $f(x) = 2x + 3$, ou dans une équation $2x + 3 = 5x - 3$ pour désigner une inconnue qu'il faut trouver, ou encore dans une formule comme $(a + b)^2 = a^2 + 2ab + b^2$ pour indiquer que l'égalité est vraie pour toutes les valeurs de a et b. ↩
6. Une PEP (pour *Python Enhancement Proposal*) est un document fournissant des informations à la communauté Python, ou décrivant une nouvelle fonctionnalité. En particulier la PEP 8 décrit les conventions de style de code agréable à lire. ↩
7. Le style qui consiste à nommer les variables par des mots écritsen minuscule séparés par des blancs soulignés, par exemple `somme_des_nombres`, est appelé « *snake case* » en opposition au style qui consiste à écrire les mots attachés en commençant par des majuscules, par exemple `SommeDesNombres`, appelé « *camel case* ». ↩
8. Les p_uplet, tableaux, dictionnaires sont étudiés dans un autre chapitre du programme de 1ère. ↩

9. En algorithmique, l'affectation est symbolisée par une flèche allant de la valeur (à droite) vers le nom de la variable (à gauche), par exemple $a \leftarrow 3$ pour affecter la valeur 3 à la variable a. ↩
10. Python est un langage de typage dynamique, ce n'est pas le cas de nombreux langages comme le C ou le C++ qui forcent à définir le type d'une variable et à le conserver au cours de la vie de la variable, ils sont de typage statique. Exemple d'affectation en C :

```c
int a;
a = 3;
```

↩
11. En mathématique $\sqrt{a} = a^{\frac{1}{2}}$. ↩
12. Noter dans cet exemple la différence entre variable informatique et mathématique, et la signification du signe « = ». En mathématique $a = 2 * a + 1$ est une équation dont l'inconnue est a (elle peut être facilement résolue pour trouver la solution $a = -1$). En informatique, c'est l'affection du résultat de `2*a + 1` à la variable `a` qui prend une nouvelle valeur (même si $a \neq -1$). ↩↩
13. Vrai pour des entiers positifs. Attention aux surprises avec des nombres relatifs ! Les résultats sont différents entre langages/systèmes informatiques. En Python on peut tester `7 // -5` et `-17 // 5` qui donnent tous les deux `-4` mais `17 % -5` donne `-3` alors que `-17 % 5` donne `3`. ↩
14. Pouvoir utiliser les apostrophes ou les guillemets offre un énorme avantage : les guillemets permettent d'écrire une chaîne qui contient des apostrophes et vis-versa, par exemple `"J'aime Python"` ou `'Il dit "hello".'`. ↩
15. Attention : les opérateurs `+` et `*` se comportent différemment s'il s'agit d'entiers ou de chaînes de caractères : `2 + 2` est une addition alors que `'2' + '2'` est une concaténation, `2 * 3` est une multiplication alors que `'2' * 3` est une duplication. ↩
16. « Hello world » (traduit littéralement en français par « Bonjour le monde ») sont les mots traditionnellement écrits par un programme informatique simple dont le but est de faire la démonstration rapide de son exécution sans erreur. Source : https://fr.wikipedia.org/wiki/Hello_world ↩
17. Une méthode est un type de fonction particulier propre aux langages orientés objet. Remarquer la construction `nom_variable.nom_methode()` dans ces cas différente de `nom_fonction(nom_variable)` par exemple `len('abc')`. ↩
18. `True` et `False` (et `None`) sont les rares mots en Python qui s'écrivent avec une majuscule. `TRUE` ou `true` ne sont pas acceptés. ↩
19. Préférer `is` et `is not` à `==` et `!=` pour comparer à `None`, par exemple `a is not None` plutôt que `a != None`. ↩
20. Les comparaisons entre chaînes de caractère se font en comparant le point de code Unicode de chaque caractère. Il est donné par la fonction `ord()` (la fonction `chr()` fait 'inverse'). Par exemple, `ord('A')` vaut `65` et `ord('a')` vaut `97` donc `'A' < 'a'` est vrai. ↩
21. Les nombres de type `float` sont encodés par des fractions binaires qui "approchent" leur valeur le plus précisément possible sans être toujours parfaitement exactes. Par exemple le nombre $0,1$ est représenté par la valeur `0.1000000000000000055511151231257827021181583404541015625` en Python (`format(0.1,'.55f')` permet d'afficher toutes les décimales). Une particularité de Python est de ne pas limiter l'encodage des `int`, par exemple comparer `>>> 2*1000` avec `>>> 2.**1000` dans la console. ↩
22. Nous n'abordons pas la notion de classe ici. ↩
23. `range(d, f)` montre l'avantage d'exclure la borne supérieure, il y `f-d` nombres compris entre `d` (inclus) et `f` (exclus), comme l'a expliqué Edsger W. Dijkstra dans une note de 1982 ↩
24. L'instruction `range()` fonctionne sur le modèle `range([début,] fin [, pas])`. Les arguments entre crochets sont optionnels. ↩
25. Nous avons déjà utilisé une fonction avec un paramètre facultatif par mot-clé avec `end=''` dans `print('hello', end='')`. ↩
26. La fonction `min()` existe dans Python. ↩
27. Une fonction qui renvoie `None` (ou qui ne renvoie rien dans d'autres langages) est appelée une procédure. ↩
28. Par exemple la valeur de π dans le module `math`. ↩
29. La fonction standard Python `pow()` prend trois paramètres alors que la fonction `pow()` du module `math` n'en a que deux ! L'import de `from math import *` a importé la seconde fonction probablement à l'insu du programmeur. ↩
30. On peut facilement se convaincre de l'utilité de la fonction `main()` en écrivant le programme qui affiche la décomposition d'un nombre en facteurs premiers sans `main()` dans le fichier « mesfonctions.py » :

```python
def est_premier(nombre):
    for div in range(2, nombre):
        if nombre % div == 0:
            return False
    return True

nombre = int(input('Entrez un nombre '))
premier = 2 # on commence par le plus petit nombre premier : 2
while nombre > 1:
        if nombre % premier == 0:      # si premier divise nombre
            print(premier, end=" ")               # alors on l'affiche
            nombre = nombre // premier     # et on recommence après avoir divisé nombre par premier
        else:                          # sinon, premier n'est pas un diviseur
            premier += 1                 # on cherche le nombre premier suivant
            while not(est_premier(premier)):
                premier += 1
```

Lorsque pour utiliser la fonction `est_premier` le module est importé dans un autre programme avec `import mesfonctions`, toute la suite du programme de la décomposition d'un nombre en facteurs premiers est exécuté automatiquement. ↩
31. Nous avons déjà vu la fonction `int()` qui semble similaire à `math.floor()`, mais attention aux différences entre les deux.

Avec un `float` positif

```
>>> math.floor(12.5)
12.0
>>> int(12.5)
12
```

Avec un `float` négatif

```
>>> math.floor(-12.5)
-13.0
>>> int(-12.5)
-12
```

↩

32. https://www.informationisbeautiful.net/visualizations/million-lines-of-code/ ↩
33. Dans la pratique il n'est pas possible de tester un logiciel dans toutes les conditions qu'il pourrait rencontrer lors de son utilisation et donc pas possible de contrer la totalité des bugs : un logiciel comme Microsoft Word compte 850 commandes et 1 600 fonctions, ce qui fait un total de plus de 500 millions de conditions à tester. ↩
34. En 1996, l'USS Yorktown teste le programme Navy's Smart Ship. Un membre d'équipage rentre un zéro comme valeur lors de manœuvres. Source : https://en.wikipedia.org/wiki/USS_Yorktown_(CG-48) ↩
35. Premier vol d'Ariane 5 en 1996 : Le code utilisé était celui d'Ariane 4, mais les valeurs d'accélération de la fusée dépassent les valeurs maximales prévues ! Source: https://fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5 ↩
36. RTFM est, en anglais, le sigle de la phrase *Read the fucking manual*, injonction signifiant que la réponse à une question sur le fonctionnement d'un appareil est à chercher dans son mode d'emploi. ↩
37. Contrairement à d'autres langages comme Java, C ou C++, qui utilisent des accolades pour séparer les blocs de code, Python utilise l'indentation pour définir la hiérarchie et la structure des blocs de code. ↩
38. Comme le disait Guido van Rossum: "*Code is read much more often than it is written.*" ↩
39. La programmation défensive est un mode de programmation qui vise à créer des programmes et des applications robustes face aux erreurs et aux entrées de données inattendues. ↩