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. Dans ce chapitre on étudie 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

Cours

Un programme peut manipuler des informations en utilisant des **variables**. Une variable associe un nom à la valeur.

On peut concevoir une variable un peu comme une “boite” qui est identifiée par un nom et contient les informations utilisées par un programme informatique. Le contenu de cette “boite” peut évolue pendant l'exécution du programme[4].

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 souligné (`_`) ;
- ne doit pas comporter d'espace, de signes d'opération `+` , `-` , `*` ou `/` , ni de caractères spéciaux comme `','` ou `@` ;
- ne doit pas débuter 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 [5] 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 « `_` » . Par exemple on utilise `somme_des_nombres` plutôt que `s` dans un programme qui additionne des nombres, ce style est appelé « *snake case* »[6].

1.2. Types de variable

Cours

Les variables peuvent être de **types** différents en fonction des informations qu'elles contiennent.

On trouve par exemple :

- des nombres entiers (type `int`) ;
- des nombres décimaux, appelés « flottants » (type `float`) (⚠ le séparateur décimal est un point, PAS une virgule), noter qu'on peut par exemple écrire `5.` ou `.5` pour 5.0 ou 0.5 et `2e5` ou `2E5` pour 2×10^5 ;
- des booléens prenant seulement les valeurs `True` ou `False` (type `bool`) ;
- des textes ou chaines des caractères (type `str`) écrits entre une paire de guillemets (`"`) ou d'apostrophes (`'`) ;
- des p_uplet, tableaux, dictionnaires, etc.

1.3. Affectation

Cours

L'**affectation** consiste à stocker une valeur dans une variable. En Python, comme dans la plupart des langages informatiques, l'**affectation d'une valeur à une variable** est représentée par le signe « `=` ».[7]

Par exemple pour affecter les valeurs 3 (type `int`), 3.3 (type `float`) et "trois" (type `str`) à des variables nommées `a`, `b` et `c` dans la console Python:

```
>>> a = 3
>>> b = 3.3
>>> c = "trois"
```

PEP-8

Mettre des espaces autour du signe « `=` ».

⚠️ On ne peut pas écrire :

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

On peut aussi affecter la valeur d'une variable à une autre variable, par exemple :

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

et affecter les valeurs de plusieurs variables en une seule ligne :

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

PEP-8

Mettre un espace après les virgules (mais pas avant).

En Python, c'est l'affectation qui définit le type de la variable[8].

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

Les variables `a`, `b` et `c` sont de types `int`, `float` et `str` respectivement.

⚠️ On ne peut pas utiliser une variable avant de l'avoir définie.

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

2. Opérations, comparaisons, expression

2.1. Opérateurs arithmétiques sur les nombres

Les opérations arithmétiques usuelles sont effectuées sur des nombres et/ou des variables 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 (`+`, `-`, `/`, `*`) d'un espace avant et d'un espace après.

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

> Le résultat d'une opération est de type `float` si `a` ou `b` est de type `float`, sinon de type `int`, sauf pour la division où le résultat est toujours `float`.
>
> Pour obtenir la racine carrée d'un nombre on peut utiliser `a ** 0.5`.
>
> L'ordre des priorités mathématiques est respecté.

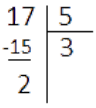
2.1.1. Division entière (ou division euclidienne)

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[9].

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 ($17 = 2 \times 5 + 2$)




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

On peut affecter une valeur à une variable qui dépend de son ancienne valeur, par exemple l'augmenter d'une quantité donnée (on dit **incrémenter**)[10].

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

PEP-8

Pas d'espace autour du signe `*` pour montrer la priorité sur l'addition.

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

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`.

2.2. 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 (`'`)

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

Pour les chaînes de caractères, deux opérations sont possibles, l'addition et la multiplication [11]:

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

```
>>> chaine1 + chaine2
'hello world'
```

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

```
>>> chaine1 * 3
'hellohellohello'
```

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

```
>>> ch = 'hello world !'
>>> len(ch)
13
```

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

- On peut accéder au premier caractère avec `ch[0]`, au second avec `ch[1]`, … au caractère en iieme position par `ch[i]`. ⚠️ Le premier caractère est `ch[0]`, et non pas `ch[1]` !

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

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

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

PEP-8

Pas d'espace autour du signe « `:` ».

- On peut enfin accéder à la sous-chaîne de tous les caractères entre `i` et `j - i`, appelée une tranche, avec `ch[i:j]`.

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

On peut aussi vérifier l'appartenance, ou pas, d'une chaine dans une autre avec `in` et `not in` :

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

Il existe de nombreuses méthodes[12] pour traiter les chaines de caractères, en voilà quelques exemples :

fonction	description	exemple

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

`.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'`

fonction	description	exemple
`.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 mathématiques de comparaison peuvent être effectuées sur des valeurs. Le résultat est toujours un booléen (de type `bool`) égal à `True` ou `False` [13].

PEP-8

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

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

[14]

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

On peut aussi combiner les comparaisons. Pour vérifier si `a` est compris entre 2 et 6 ou entre 7 et 8 on peut écrire :

```
>>> 2 <= a < 6
True
>>> 7 < a < 8
False
```

Attention, c'est en fait une combinaison de plusieurs comparaisons, ce qui peut donner des hérésies mathématiques :

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

On peut aussi comparer les chaines de caractères par ordre lexicographique, c'est-à-dire que l'on commence par comparer le premier caractère de chacune des deux chaînes, puis en cas d'égalité on s'intéresse au second, et ainsi de suite comme quand on cherche un mot dans un dictionnaire. Attention aux majuscules et aux nombres écrits dans des chaînes [15]:

```
>>> 'aa'>'ab'
False
>>> "python" == "python"
True
>>> "python" != "PYTHON"
True
>>> "java" < "python"
True
>>> "java" > "Python"
True
>>> "10" < "2"
True
```

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

```
>>> 17 == 17.0
True
>>> 0.0 < 1
True
```

Mais pas les nombres avec les chaines de caractères :

```
>>> 17 == "17"
False
```

⚠️ Attention aux égalités entre nombres de type `float` qui ne sont pas toujours encodés de façon exacte [16] :

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

2.4. Opérateurs logiques (ou booléens)

Les opérations logiques peuvent être effectuées sur des booléens. 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, on met des parenthèse.

2.5. Expressions

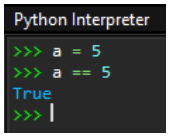
Cours

Une **expression** (ne pas confondre avec une instruction) est le résultat d'un calcul d'opérations ou de comparaisons sur des variables, des nombres, etc.

Exemples :

- `2*a + 5` est une expression qui prend la valeur 2 fois la valeur de `a` plus 5.
- `a = 5` n'est PAS une expression, c'est une affection de la valeur 5 à la variable `a`.
- `a == 5` est une expression qui prend la valeur `True` si `a` est égal à 5, ou la valeur `False` sinon.

On observe dans la console Python quand on entre une expression, elle est évaluée par l'interpréteur et le résultat est affiché. Quand on entre une affectation, rien n'est affiché.



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

On donne une variable `annee` de type `int`. Par exemple `annee = 2023`. Ecrire l'expression booléenne (qui vaut True ou False) qui indique si `annee` est une année bissextile ou pas. « 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 ? On vérifie que le reste de la division entière est égal à zéro. 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 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 C1 **ou** C2 (ou les deux) est **vraie**.

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 la machine avant de passer à la suivante.

Une **séquence** est une suite d'**instructions exécutées** dans l'ordre où elles sont écrites.

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.

3.1. type()

On peut écrire une instruction pour connaitre le type d'une variable avec la fonction `type()` [17].

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'>
```

Rappel

En Python, la valeur `2` (type `int`) est égale à `2.0` (type `float`) et mais est différente de `'2'` (type `str`).

3.2. Conversion de type

On peut convertir une variable d'un type à un autre dans une instruction en utilisant:

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'`

> Observer dans la console comment un flottant qui prend une valeur entière (5) est affiché avec un point (5.0) :

```
>>> 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 des valeurs saisies au clavier par l'utilisateur. Une instruction de **sortie** affiche des messages à l'écran.

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

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

La valeur renvoyée par `input()` est toujours du type chaine de caractères (type `str`). Pour obtenir un entier ou un flottant, il faut la convertir.

```
>>> 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 entier, il faut la convertir avec `int()`.

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

> Si l'utilisateur ne saisit pas un nombre entier, cela 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 le caractère « `,` » 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 l'affichage. Pour éviter cela on utilise une autre chaine de caractères à accoler à la fin de l'affichage avec `end=`, par exemple un espace ou même rien.

```
>>> 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.

```
>>> nom = 'Paul'
>>> age = 23
>>> print(f'Bonjour {nom}, vous avez {age} ans')
Votre nom est un Paul et vous avez 23 ans
```

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

Avant d'écrire le programme on peut se poser quelques questions :

- 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` qu'on peut nommer par exemple `R`, `V` et `B` comme dans la formule.
- Que doit calculer le programme ? Le niveau de gris en utilisant la formule. On peut mettre le résultat dans une variable `G` de type entier.
- Que doit faire ensuite le programme ? Le programme doit afficher le niveau de gris.

Traduit en Python, on obtient le programme suivant. Noter la présence de commentaires dans le code, commençant par le signe `#`, ils sont ignorés par l'interpréteur Python.

Dans un premier temps on n'utilise pas `def main():` généré par défaut par certains IDE.

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('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.

En Python, on utilise 4 espaces (ou Tab) pour faire une indentation (on pourrait aussi utiliser un seul ou plusieurs espaces mais il faut être consistant).

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 caractère « `:` ».

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

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

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.

On peut choisir de ne pas indenter l'instruction en ligne 5 `print(f"Vous avez {age} ans")`, elle ne fera plus partie de l'instruction conditionnelle et sera 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 on n'indentait pas l'instruction en ligne 4 `print(f"Vous êtes {stade}")`, la variable `stade` ne serait pas définie quand la condition n'est pas vérifiée et dans ce cas on aurait un message erreur.

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, on peut ne plus indenter l'instruction en ligne 6 `print(f"Vous êtes {stade}")`.

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 :

programme 1	programme 2

programme 1	programme 2

`if age >= 18:`
`  stade = "adulte"`
`else:`
`  if age >= 12:`
`  stade = "ado"`
`  else:`
`  if age >= 2:`
`  stade = "enfant"`
`  else :`
`  stade = "bébé"`
`print(f"Vous êtes {stade}, vous avez {age} ans")`
	`if age >= 18:`
	`stade = "adulte"`
	`elif age >= 12:`
	`  stade = "ado"`
	`elif age >= 2:`
	`  stade = "enfant"`
	`else :`
	`  stade = "bébé"`
	`print(f"Vous êtes {stade}, vous avez {age} ans")`

Dès qu'une condition est vérifiée, le programme ne teste pas les conditions `elif` suivantes ni `else` mais saute directement à la fin du bloc conditionnel, ici `print(f"Vous êtes {stade}, vous avez {age} ans")`. Par exemple, si on affecte la valeur `15` à la variable `age`, le programme exécute le bloc `elif age >= 12:…` mais pas les blocs suivants `elif age >=2 :…` ni `else:…`.

Rappel

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

```
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 :

```
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 --> |True| 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 % 400 == 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

On pourrait aussi utiliser l'expression trouvé 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 l'expression à tester par la condition afin qu'elle finisse par ne plus être vérifiée, sinon la boucle est sans fin (non bornée).

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, on parle de **programme qui boucle** ou de divergence. Ici, c'est bien le cas grâce à l'instruction `i += 2`.

Exemple de programme qui boucle (erreur d'indentation dans l'instruction `i += 1`):

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

Comme pour les instructions conditionnelles, attention de faire particulièrement attention aux boucles avec les nombres de type `float`. Testons par exemple :

```
>>> while i < 2:
...     i += 0.1
...     print(i)
...
1.1
1.2000000000000002
1.3000000000000003
1.4000000000000004
1.5000000000000004
1.6000000000000005
1.7000000000000006
1.8000000000000007
1.9000000000000008
2.000000000000001
```

On voit que `i` ne prend jamais la valeur `2`. La boucle suivante qui semble équivalente ne finira donc jamais !

```
i = 1
while i != 2:
    i += 0.1
```

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**.

On peut aussi utiliser :

- `range(d, f)` qui énumère les `f-d` nombres entiers compris entre `d` et `f-1`. [18]
- `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. [19]

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	programme 2

`for i in range(5):`
`  print(i)`
	`i = 0`
	`while i < 5:`
	`  print(i)`
	`  i += 1`

Ils semblent donner le même résultat : afficher les chiffres de 0 à 4.

Pourtant ils sont différents. Ajoutons une instruction `print(i)` à la fin les deux programmes. Dans le premier cas, la valeur finale de `i` est `4`, dans le second cas c'est `5` afin que la condition ne soit plus valide.

Autre différence, on peut modifier l'indice de boucle, mais il reprend sa valeur à la répétition suivante. En pratique, on évite de le faire.

programme 1	programme 2

`for i in range(5):`
`  print(i)`
`  i = i + 2`
`  print(i)`
	`i = 0`
	`while i < 5:`
	`  print(i)`
	`  i = i + 2`
	`  print(i)`

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

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.

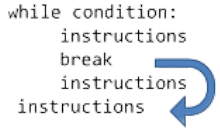
```
message = 'bonjour'
for c in message:
    print(c)
```

4.4. Instructions break et continue

Il existe deux instructions qui permettent de modifier l'exécution des boucles, 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.

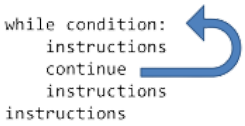


Par exemple, voici un programme qui redemande un mot de passe jusqu'à obtenir le bon :

```
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.



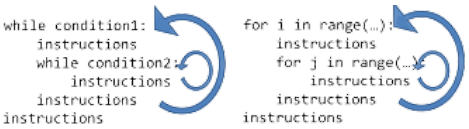
Imaginons par exemple que l'on veuille imprimer pour tous les entiers n compris entre 1 et 10 la valeur de $1/(n-7)$. Il est évident que quand n prend la valeur 7 il y aura une erreur. Grâce à l'instruction `continue` , il est possible de traiter cette valeur à part puis de continuer la boucle.

```
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.

On peut par exemple afficher toutes les heures et minutes de la journée (de 0h0min à 23h59 min) :

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

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:` .

- 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.

```
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 depuis le début de cette leçon, par exemple `print()` ou `len()` sont des fonctions prédéfinies par Python. Quand on écrit un programme on utilise beaucoup de fonctions existantes, mais très souvent on veut aussi créer nos propres fonctions.

Noter ici la différene avec une fonction mathématique.

Une fonction permet d'isoler une séquence d'instructions pour pouvoir l'utiliser à n'importe quel moment et autant de fois que souhaité sans la réécrire. L'utilisation de fonctions facilite aussi la lisibilité de longs programmes .

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
```

De la même façon que pour 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.

Quand on définit une fonction, elle ne s'exécute pas. Et ceci même si la fonction contient une erreur, l'interpréteur Python ne s'en aperçoit pas. Les deux programmes suivants ne font strictement rien :

Programme 1

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

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

Programme 2

```
1  def bonjour():
2      prit('hello')
```

La fonction `bonjour` n'est pas appelée, ce programme ne fait rien, même s'il y une erreur (`prit` au lieu de `print`).

Pour exécuter la fonction, il faut l'**appeler** dans un programme ou dans l'interpréteur Python en écrivant son nom suivi des parenthèse.

⚠️ Quand la fonction n'a pas de paramètres il faut quand même mettre les parenthèses quand on l'appelle.

```
1  def bonjour():
2      print('hello')
3
4  >>> bonjour()
5  hello
```

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

Programme 1

```
1  bonjour()
2
3  def bonjour():
4      print('hello')
```

La fonction `bonjour` est appelée avant d'être définie.

Programme 2

```
1  def main():
2      bonjour()
3
4  if __name__ == '__main__':
5      main()
6
```

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

La fonction `bonjour` est appelée avant d'être définie.

5.1. La fonction main()

On a déjà vu une fonction appelée `main` générée automatiquement par certains éditeurs suivie par le code suivant :

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

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

L'interpréteur Python exécute tout programme linéairement à partir du haut donc il n'est pas indispensable de définir cette fonction `main()` dans chaque programme, mais il est recommandé de le faire pour indiquer le point de départ pour l'exécution afin de mieux comprendre le fonctionnement du programme.

5.2. Paramètres et arguments

Cours

Même si dans la pratique on confond souvent les deux termes par abus de langage :

- 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.

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

Ici on appelle la fonction `bonjour` à la ligne 4 en lui passant les **arguments** `'Tom'` et `'Lisa'`. Ce sont les valeurs que prennent les deux **paramètres** `prenom1` et `prenom2` pendant l'exécution de la fonction.

🪱 Il faut appeler une fonction avec le même nombre d'arguments qu'elle a de paramètres sinon on obtient 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'
```

Lorsqu'on définit la fonction avec « `def bonjour(prenom1, prenom2):` », `prenom1` prend la valeur du premier argument quand on appelle la fonction et `prenom2` la valeur du deuxième. `prenom1` et `prenom2` sont appelés des **arguments positionnels** (en anglais *positional arguments*). Il est **obligatoire** de leur donner une valeur quand on appelle une fonction et **dans le bon ordre**. Dans l'exemple ci-dessus, `prenom1` prend la valeur `'Tom'` et `prenom2` la valeur `'Lisa'`, comme indiqué par leur position.

Pour avoir des paramètres facultatifs, il faut leur affecter une valeur par défaut.

PEP-8

Pas d'espace autour du signe = pour les arguments par mot-clé.

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

Ici, lorsqu'on définit la fonction avec « `def bonjour(prenom1, prenom2='Tim'):` », la valeur de `prenom2` est `'Tim'` par défaut, c'est la valeur qui est utilisée par la fonction si on ne la précise pas quand on l'appelle. `prenom2` **est appelé un** argument par mot-clé **(en anglais *keyword argument*). Le passage d'un tel argument lors de l'appel de la fonction est** facultatif**. On peut donner la valeur des paramètres par leur mot clé dans n'importe quel ordre.

```
def bonjour(prenom1='Tom', prenom2='Tim'):
    print('hello', prenom1, 'and', prenom2)
```

Exemple d'appel 1

```
>>> bonjour("Paul", "Pierre")
hello Paul and Pierre
```

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

Exemple d'appel 2

```
>>> bonjour(prenom2="Jack")
hello Tom and Jack
```

La fonction est appelée avec la valeur de `prenom2` qui est donnée, `prenom1` prend sa valeur par défaut.

Exemple d'appel 3

```
>>> bonjour(prenom2="Jack", prenom1="Joe")
hello Joe and Jack
```

La fonction est appelée avec deux arguments donnés par leur mot-clé, l'ordre n'a pas d'importance.

Si une fonction a un mélange d'arguments positionnels et par mot-clé, les arguments positionnels doivent toujours être placés avant les arguments par mot-clé : Ecrire « `def bonjour (prenom1='Tim', prenom2):` » est incorrect.

5.3. L'instruction Return

Cours

On dit préfère le verbe "renvoyer" à "retourner" qui est un anglicisme pour *return*.

Une fonction peut **renvoyer** une ou plusieurs valeurs (nombres, textes, booléens, etc..) avec l'instruction return.

S'il n'a pas d'instruction `return` dans une fonction, elle renvoie `None`[20] (on parle alors de procédure) .

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

Voici par exemple une fonction qui vérifie si un nombre est premier ou pas. On teste tous les diviseurs potentiels les uns après les autres en vérifiant si le reste de la division entière est égal à zéro. Dès qu'on trouve un diviseur, inutile de continer, le nombre n'est pas premier et dans ce cas l'instruction `return False` termine la fonction. Si on ne trouve aucun diviseur après les avoir tous testés, la fonction se termine à la dernière ligne avec l'instruction `return True`.

Avec une boucle for jusqu'à nombre - 1

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

Appelons la fonction avec les nombres 10 et 13

Avec le nombre 10

```
>>> estpremier(10)
False
```

L'instruction conditionnelle est vérifiée dès le premier passage dans la boucle, donc l'instruction `return False` est immédiatement exécutée et la fonction se termine là, la dernière instruction `return True` n'est jamais exécutée.

Avec le nombre 13

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

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

⚠️ Attention à ne pas confondre `print()` et `return`. Comparons ces deux fonctions :

Fonction 1

```
def ajoute_1(nombre):
    print(nombre + 1)
```

Fonction 2

```
def ajoute_1(nombre):
    return nombre + 1
```

Quand on appelle l'un ou l'autre programme dans la console, on obtient le même résultat :

```
>>> ajoute_1(5)
6
```

Alors quelle est la différence ?

- Avec `print()`, la première fonction `ajoute_1()` 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 la seconde fonction le résultat calculé et renvoyé par la fonction peut être utilisé, par exemple pour l'affecter à une variable ou comme argument d'autres fonctions, voire même pour être affiché avec `print()` si on le souhaite.

Dans le doute, de façon générale, on évite d'afficher un résultat avec `print()` dans une fonction autre que la fonction `main()` et on préfère utiliser `return`.

Une fonction peut aussi renvoyer plusieurs valeurs en même temps, séparées par des virgules, par exemple:

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

Exercice corrigé

Écrire un programme qui affiche la décomposition d'un nombre en facteurs premiers en utilisant la fonction `est_premier()` donnée.

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

Réponse

Pour décomposer un nombre en facteurs premiers on commence par cherche son plus petit diviseur qui est un nombre premier (un "facteur premier") et on divise ce nombre par ce diviseur, puis on fait la même chose pour le quotient obtenu, puis sur le deuxième quotient, etc. tant que le quotient est plus grand que 1.

```
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)                 # 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
```

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 un exemple :

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

Comme avec une variable, on utilise le signe `=` pour affecter à la fonction (`ma_somme`) sa définition, avec d'abord le mot réservé `lambda` suivi de la liste des paramètres (zéro, un ou plusieurs), séparés par des virgules (ici deux paramètres `x` et `y`), ensuite figure un nouveau signe deux points « `:` » et l'expression de la fonction lambda.

Les fonctions lambda n'ont qu'une seule expression et c'est le résultat de cette expression qui est renvoyé par la fonction.

Dans notre exemple, `ma_somme` renvoie la valeur de l'expression `x + y` .

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
```

On peut utiliser une instruction conditionnelle par exemple :

```
>>> 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 lèvent une erreur

Programme 1

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

Ce programme essaie d'afficher la variable `a` avant qu'elle ne soit définie.

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.

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

Exemple : Le programme suivant lève une erreur. La variable a n'existe pas dans la fonction main(), elle est locale à affiche_a ().

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

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

5.5.2. Paramètres passés par valeur

Dans les exemples précédents, les arguments utilisés en appelant les fonctions étaient à chaque fois des valeurs (est_premier(10), est_premier(13). Cela n'est nullement obligatoire. Les arguments utilisés dans l'appel d'une fonction peuvent aussi être des variables ou même des expressions.

```
>>> est_premier(23)         >>> a = 23              >>> nombre = 22
True                    >>> est_premier(a)          >>> est_premier(nombre + 1)
                    True                    True
```

Quand une variable est passée en argument à la fonction, par exemple dans le cas de est_premier(a), sa valeur est copiée dans une variable locale à la fonction. C'est cette variable locale qui est utilisée pour faire les calculs dans la fonction appelée. Les modifications de cette variable locale à l'intérieur de la fonction ne modifient pas la variable qui a été passée en paramètre. Et c'est le cas même quand le nom de la variable passée en paramètre 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. On dit que **les paramètres sont passés par valeur**.

Exemple :

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

def main():
    a = 1
    ajoute_1 (a)
    print(f'valeur de a dans main :{a}')
>>>
valeur de a dans ajoute_1 : 2
valeur de a dans main : 1
```

`ajoute_1 (a)` change la valeur de `a` en `2` pendant son exécution, mais la valeur de `a` n'a pas changée dans `main()`.

5.5.3. Variables globales

Sauf exception on préfère 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 :

```
a = 1
def fonction():
    print(a)
fonction()
```

Mais ne peut pas être modifiée. Le programme suivant lève une erreur :

```
a = 1
def fonction():
    a += 1
    print(a)
fonction()
```

On peut néanmoins essayer de lui assigner une nouvelle valeur :

```
a = 1
def fonction():
    a = 2
    print(a)
fonction()
```

Mais dans ce cas, Python part du principe que `a` est locale à la fonction, et non plus une variable globale :

```
a = 1
def fonction():
    a = 2
    print(a)
fonction()
print(a)
```

L'instruction `a = 2` a créé un nouvelle variable locale à la fonction, la variable globale n'a pas changé.

Dans certaines situations, il serait utile de pouvoir modifier la valeur d'une variable globale depuis une fonction, notamment dans le cas où une fonction se sert d'une variable globale et la manipule. Pour faire cela, il faut utiliser le mot clef `global` devant le nom d'une variable globale utilisée localement afin d'indiquer à Python qu'on souhaite bien modifier son contenu de la variable globale et non pas créer une variable locale de même nom :

```
a = 1
def fonction():
    global a
    a = 2
    print(a)
fonction()
print(a)
```

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.

Exemple : Programmer la fonction factorielle $n! = 1 \times 2 \times 3 \times 4 \times \ldots \times (n-1) \times n$

Programme standard. On multiplie tous les entiers de 1 à n

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

Programme récursif :. n ! = (n-1)! * n et 1 !=1

```
def factorielle_recursive(n):
    if n == 1:
        return 1
    else:
        return n * factorielle_recursive(n-1)          # 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 compte à rebours 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

On a créé une fonction `est_premier` qui peut être appelée par d'autres programmes ou fonctions qui se trouvent dans le même fichier. Ecrivons un nouveau programme Python qui appelle est_premier:

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

ce programme génère une erreur : `NameError: name 'est_premier' is not defined`

Comment faire pour appeler `est_premier` depuis ce nouveau programme sans tout 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.

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

Cours

Un **module** est un ensemble de fonctions (et des variables) ayant un rapport entre elles que l'on a enfermé dans un fichier. On doit **importer** ce module dans un programme et utiliser ses fonctions. Une **bibliothèque** (ou *package* en anglais) est un ensemble de fonctions, variables, et de **modules** dans un même fichier.

5.8. `import`

Pour importer un module dans un programme, on utilise l'instruction `import`. On écrit en début de programme :

```
import mesfonctions
```

A la différence des fonctions que l'on utilise habituellement dans un programme, comme `print()` ou `range()`, pour appeler une fonction du module il faut taper le nom du module suivi d'un point « . » puis du nom de la fonction. Par exemple on écrit `mesfonctions.est_premier()`

```
1  import mesfonctions
2  def main():
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 ».

On peut aussi donner un alias à un module :

```
1  import  mesfonctions  as mf
```

et utiliser ensuite `mf.est_premier()`.

Cours

Sans alias

On importe un module nom_module en début de programme avec l'instruction :

`import nom_module`

puis on utilise une fonction de ce module nom_fonction() en écrivant :

`nom_module.nom_fonction()`

Avec alias

On importe un module nom_module en début de programme en lui donnant un alias avec l'instruction :

`import nom_module as nom_alias`

puis on utilise une fonction de ce module nom_fonction() en écrivant :

`nom_alias.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 d'importation qui ne fonctionne pas tout à fait de la même façon. 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 on préfèrera n'importer que cette fonction au lieu d'importer tout le module.

```
from  mesfonctions  import est_premier

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

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

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

On importe une fonction `nom_fonction()` depuis le module `nom_module` en début de programme avec l'instruction :

`from nom_module import nom_fonction`

puis on utilise cette fonction en écrivant :

`nom_fonction()`

Avec alias

On importe une fonction `nom_fonction()` en lui donnant un alias depuis le module `nom_module` en début de programme avec l'instruction :

`from nom_module import nom_fonction as nom_alias`

puis on utilise cette fonction en écrivant :

`nom_alias()`

On peut aussi appeler plusieurs fonctions d'un même module séparée 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 utilisation est vivement déconseillée[21] hormis dans des cas très particuliers par exemple des programmes très courts.

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. 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.[22]

5.11. Le module math

Le module `math` permet d'avoir accès aux fonctions mathématiques comme le cosinus (`cos`), le sinus (`sin`), la racine carrée (`sqrt`), le nombre π (`pi`), la partie entière (`floor`)[23], et bien d'autres...

```
>>> 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. Les deux plus utiles courantes 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 on utilise :

```
x, y = 2, 4
plt.plot(x, y,'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

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, on a déjà vu la fonction help qui donne les spécifications d'une fonction.

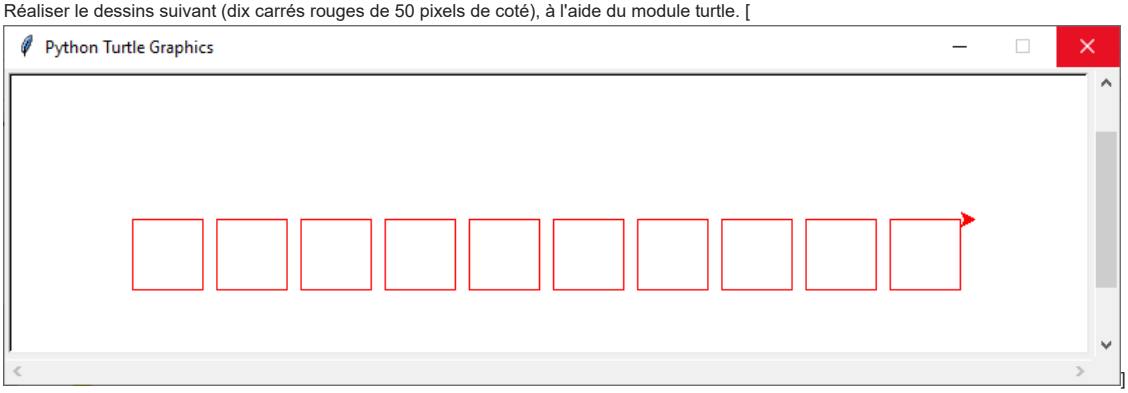
```
>>> help(math.cos)
Help on built-in function cos in module math:

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 bug ? Ces questions peuvent devenir extrêmement cruciales et compliquées quand certains programmes informatiques contiennent des millions de lignes de code, voire des milliards (Google)[21] 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 ! [22]) 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 [23], typo, dépassement de capacité mémoire[24], 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 se comprendre les différents types d'erreurs qui surviennent et pourquoi elles se produisent. Soyez attentif aux messages d'erreur de l'interpréteur, ils sont d'une grande utilité . Voici ceux que l'on rencontre le plus souvent :

- 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` : 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.

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

- `TypeError` : vous essayez d'effectuer une opération en utilisant un type de données incompatible :

```
>>> "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 un argument d'une value non autorisée :

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

- etc.

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,

Que fait cette fonction?

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

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

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*)

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

et encore plus lisible avec des commentaires :

```
def parfait(nombre):
    somme_diviseurs = 0
    # 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

Les commentaires sont souvent indispensables afin de comprendre et modifier un programme[26] mais sont ignorés par l'interpréteur Python.

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 contenant précisément les paramètres d'appel à fournir, leur type et les valeurs renvoyées.

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):
"""commentaires de spécifications
entre triple guillemets
"""
```

Si l'idée générale est toujours la même (spécifier les paramètres, ce que fait la fonction, ce qu'elle renvoie), et que certaines conventions sont données dans la PEP 257, on rencontre différentes habitudes d'écrire une docstring Python. Par exemple, la fonction parfait(nombre) pourra 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 :

```
    """ 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 """
```

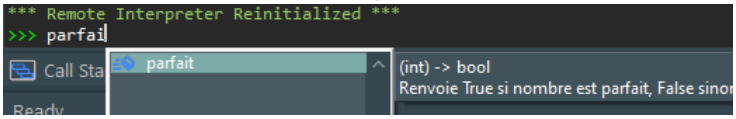
La fonction `help` affiche la docstring d'une fonction, par exemple on peut aussi saisir `help(print)` dans la console.

```
>>> 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 `#` . On peut d'ailleurs ajouter des commentaires indiqués par `#` dans la docstring qui ne seront pas affichés par `help` . La spécification sera lue par l'utilisateur de la fonction, les commentaires par le programmeur qui lit/écrit le programme.

Remarquer aussi dans la console ou en programmant quand on tape `parfait(` la spécification qui s'affiche.



6.4. Préconditions, postconditions

On a auparavant testé la fonction `parfait(nombre)` avec un exemple simple :

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

Que se passe t'il maintenant si on passe un argument qui n'est pas un entier 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. On peut indiquer les préconditions dans la docstring de la fonction.

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

De la même façon, on peut préciser des **postconditions** quand il y en a.

Cours

Une **précondition** doit être vraie avant le début d'un calcul / d'une fonction afin de garantir que son exécution soit possible sans erreur.

Exemple : avant une division, s'assurer que le dénominateur est non nul.

Cours

Une **postcondition** doit être vraie à la fin d'un calcul / d'une fonction afin de garantir que son résultat soit correct.

Une postcondition non satisfaite correspond à une erreur lors du calcul, généralement due à une erreur de programmation. Par exemple, vérifier à la fin d'une fonction renvoyant la valeur absolue d'un nombre que le résultat est supérieur ou égal à 0.

6.5. Variant et invariant de boucle

Cours

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

…mais pas qu'un algorithme fournit la réponse attendue

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
```

`d > 0` est une précondition à la fonction. Si ce n'est pas cas (`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. On peut ainsi compléter la table jusqu'à la fin du programme. On obtient :

```
(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.

On peut observer 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**.

On peut aussi 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 on peut 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

On peut tester les préconditions, postconditions et les invariants par des assertions, car leur non-respect est due à une erreur de programmation.

Cours

Assertion : vérifie qu'une expression est **vraie*, sinon stoppe le programme.

```
assert <condition>
```

On peut aussi afficher des informations quand l'assertion est **fausse** puis stopper le programme :

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

Reprenons la fonction `est_premier(nombre)` vue précédemment. Le parmaètre `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
```

On obtient:

```
>>> 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**[27].

Si on veut gérer les erreurs prévisibles d'utilisateur (lors d'une saisie par exemple), on peut utiliser `try...…except...` (hors programme). 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

On teste sur des valeurs d'arguments normales mais aussi des valeurs 'spéciales' ou 'extrêmes' du programme.

Par exemple, que se passe-t-il si on passe `0` et `1` comme 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**.

On peut vérifier 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
```

On peut aussi écrire un programme de tests en utilisant la célèbre formule d'Euler : n² + 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 on écrira :

```
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. 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. ↩
5. 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. ↩
6. En opposition au style appelé « *camel case* » qui consiste à écrire les mots attachés en commençant par des majuscules, par exemple `SommeDesNombres`. ↩
7. En algorithmique, l'affectation est symbolisée par une flèche allant de la valeur (à droite) vers la variable (à gauche), par exemple $a \leftarrow 3$ pour affecter la valeur 3 à la variable a. ↩
8. On dit que Python est un langage de typage dynamique. Ce n'est pas le cas de nombreux langages comme le C ou le C++ qui sont de typage statique. Exemple de programme en C :

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

↩

9. 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`. ↩
10. 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 (on peut la résoudre facilement et trouver la solution $a = -1$). En informatique, c'est une affectation qui remplace le contenu de la variable `a` part une nouvelle valeur égale à `2*a + 1`, (même si $a \neq -1$). ↩
11. 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. ↩
12. 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')`. ↩
13. `True` et `False` (et `None`) sont les rares mots en Python qui s'écrivent avec une majuscule. `TRUE` ou `true` ne sont pas acceptés. ↩
14. On préfèrera `is` et `is not` à `==` et `!=` pour comparer à `None`. On écrit `a is not None` plutôt que `a != None`. ↩
15. 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. ↩
16. Les `floats` sont encodés par une fraction binaire de numérateur sur 53 bits et de dénominateur une puissance de 2. Dans le cas de `0.1`, la fraction binaire est $3602879701896397/2^{55}$. Pour afficher toutes le décimales on peut faire: `format(.1,'.55f')`. Une particularité de Python est de ne pas limiter l'encodage des `int`, on peut par exemple comparer `2*1000` avec `2.**1000`. ↩
17. Nous n'abordons pas ici la notion de classe ici. ↩
18. `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 ↩
19. L'instruction `range()` fonctionne sur le modèle `range([début,] fin [, pas])`. Les arguments entre crochets sont optionnels. ↩
20. Donc une fonction renvoie toujours quelque chose. ↩
21. https://www.informationisbeautiful.net/visualizations/million-lines-of-code/ ↩ ↩
22. 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. ↩ ↩
23. 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) ↩ ↩
24. 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 ↩
25. 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. ↩
26. Comme le disait Guido van Rossum: "*Code is read much more often than it is written.*"

```
# Commentaire sur un bloc

instruction           # Commentaire sur une instruction particuliere
```

↩

27. La programmation défensive est un mode de programmation qui utilise des assertions pour vérifier les préconditions sont effectivement bien satisfaites. ↩