INTRODUCTION À PYTHON

Programmation

Mr Lambrechts

TABLE DES MATIÈRES

Historique des modifications

Date	Version	Modification
10-09-22	1.0	Création du document

LES VARIABLES

LES VARIABLES EN PROGRAMMATION

En programmation, les **variables** sont des **espaces de mémoire réservés pour stocker des valeurs**. Les variables sont très utiles car elles permettent de stocker et de manipuler des données tout au long de l'exécution d'un programme.

DÉCLARATION DE VARIABLES

En Python, **les variables sont déclarées en leur attribuant une valeur**. Par exemple, pour déclarer une variable nommée "age" avec une valeur de 25, nous pouvons utiliser le code suivant :

```
age = 25
```

Il est important de noter que Python est un langage de programmation à **typage dynamique**, ce qui signifie que **le type de données d'une variable est déterminé lors de l'exécution du programme**.

TYPES DE VARIABLES

En Python, il existe plusieurs types de variables de base :

- **Integer (int)** : pour stocker des **nombres entiers**, tels que 1, 2, 3, etc.
- **Float (float)** : pour stocker des **nombres à virgule flottante**, tels que 3.14, 2.718, etc.
- **Boolean (bool)** : pour stocker des **valeurs booléennes True (vrai) ou False (faux**).
- **String (str)** : pour stocker des **chaînes de caractères**, telles que "Bonjour" ou "Hello".

Voici des exemples de déclaration de variables pour chacun de ces types :

```
age = 25            # Integer
prix = 19.99        # Float
est_vrai = True     # Boolean
nom = "Jean"        # String
```

MANIPULATION DE VARIABLES

Les variables peuvent être manipulées de différentes manières en Python. Par exemple, nous pouvons leur **attribuer de nouvelles valeurs**, les **utiliser dans des opérations mathématiques**, les **concaténer** pour former des chaînes de caractères, etc.

Voici des exemples de manipulation de variables :

```
# Attribuer une nouvelle valeur à une variable
age = 26


# Utiliser des variables dans des opérations mathématiques
a = 10
b = 20
somme = a + b
difference = b - a
produit = a * b
quotient = b / a


# Concaténer des variables pour former une chaîne de caractères
prenom = "Jean"
nom = "Dupont"
nom_complet = prenom + " " + nom
```

UTILITÉ DES VARIABLES EN PROGRAMMATION

Dans la partie précédente, nous avons vu les bases des variables en programmation ainsi que leur déclaration et leur affectation de valeur. Dans cette partie, nous allons nous intéresser aux différentes utilisations et manipulations des variables en programmation.

OPÉRATIONS MATHÉMATIQUES

Les variables peuvent être utilisées pour effectuer des **opérations mathématiques simples** telles que l'addition, la soustraction, la multiplication et la division. Voici quelques exemples :

```
# Addition
```

```
a = 5
b = 7
c = a + b
print(c) # Output : 12

# Soustraction
a = 5
b = 7
c = b - a
print(c) # Output : 2

# Multiplication
a = 5
b = 7
c = a * b
print(c) # Output : 35

# Division
a = 5
b = 7
c = b / a
print(c) # Output : 1.4
```

Il est important de noter que **lorsqu'on effectue une division, le résultat est un nombre à virgule flottante**. Si l'on veut obtenir un résultat entier, on peut utiliser l'opérateur **`//`**.

LA DIVISION ENTIÈRE

Par exemple :

```
# Division entière
a = 5
b = 7
c = b // a
print(c) # Output : 1
```

L'EXPOSANT

En mathématiques, un exposant est un moyen d'indiquer la multiplication répétée d'un nombre par lui-même. En Python, l'opérateur pour effectuer des exposants est **.

Exemple :

```
a = 2
b = 3
c = a ** b
print(c)
```

Dans cet exemple, a est **élevé à la puissance** de b en utilisant l'opérateur **, ce qui signifie que `c` est égal à 2 à la puissance de 3, ou 8.

MODULO

Le modulo est un opérateur qui **renvoie le reste de la division entière de deux nombres**. En Python, l'opérateur pour effectuer un modulo est `%`.

Exemple :

```
a = 7
b = 3
c = a % b
print(c)
```

Dans cet exemple, a est divisé par b en utilisant l'opérateur %, ce qui signifie que c est égal au reste de la division entière de 7 par 3, ou 1

Le **modulo** est **souvent utilisé pour déterminer si un nombre est pair ou impair**. Si un nombre est divisible par 2, le résultat du modulo sera 0, ce qui signifie qu'il est pair. Sinon, le résultat du modulo sera 1, ce qui signifie qu'il est impair.

Exemple :

```
a = 5
if a % 2 == 0:
  print("Le nombre est pair.")
else:
  print("Le nombre est impair.")
```

Dans cet exemple, a est un nombre impair, donc le résultat du modulo sera 1. La condition if a % 2 == 0: vérifie si a est divisible par 2. Puisque a est impair et donc non divisible par 2, la condition else: est exécutée, ce qui signifie que le résultat est "Le nombre est impair

CONCATÉNATION DE CHAÎNES DE CARACTÈRES

Les variables peuvent également être utilisées pour concaténer des chaînes de caractères. **La concaténation est l'opération qui consiste à joindre deux chaînes de caractères pour en former une seule**. Voici un exemple :

```
# Concaténation de chaînes de caractères
nom = "John"
prenom = "Doe"
nom_complet = prenom + " " + nom
print(nom_complet) # Output : "John Doe"
```

Dans cet exemple, nous avons utilisé l'opérateur **+ pour concaténer les chaînes de caractères**.

COMPARAISON DE VARIABLES

Les variables peuvent également être utilisées pour comparer des valeurs. Les opérateurs de comparaison les plus courants sont == (égal à), != (différent de), > (plus grand que), < (plus petit que), >= (plus grand ou égal à) et <= (plus petit ou égal à).

Les **opérateurs de comparaison** permettent de **comparer deux valeurs** et **de retourner un booléen en fonction du résultat de la comparaison**. Les opérateurs de comparaison de base sont :

- **==** : **égal** à. Retourne True si les deux valeurs sont égales.
- **!=** : **différent** de. Retourne True si les deux valeurs sont différentes.
- **<** : **inférieur** à. Retourne True si la première valeur est inférieure à la deuxième.
- **>** : **supérieur** à. Retourne True si la première valeur est supérieure à la deuxième.
- **<= : inférieur ou égal** à. Retourne True si la première valeur est inférieure ou égale à la deuxième.
- **>=** : **supérieur ou égal** à. Retourne True si la première valeur est supérieure ou égale à la deuxième.

Voici un exemple :

```
# Comparaison de variables
a = 5
b = 7
print(a == b) # Output : False
print(a != b) # Output : True
print(a > b) # Output : False
print(a < b) # Output : True
print(a >= b) # Output : False
print(a <= b) # Output : True
```

UTILISATION DE VARIABLES DANS LES CONDITIONS

Les variables peuvent également être utilisées dans les conditions pour exécuter des blocs de code en fonction de certaines valeurs. Voici un exemple :

```
# Utilisation de variables dans les conditions
a = 5
b = 7
if a > b:
    print("a est plus grand que b")
elif a < b:
    print("a est plus petit que b")
else:
    print("a et b sont égaux")
```

TYPES DE VARIABLES EN PYTHON

En Python, il existe plusieurs types de variables, chacun étant utilisé pour stocker différents types de données. Les types de variables de base en Python sont les suivants :

LES NOMBRES

Les nombres sont utilisés pour stocker des **données numériques**. Il existe trois types de nombres en Python :

- **int (entier)** : utilisé pour stocker des nombres entiers, par exemple : 42
- **float (flottant)** : utilisé pour stocker des nombres à virgule flottante, par exemple : 3.14
- **complex (complexe)** : utilisé pour stocker des nombres complexes, par exemple : 1 + 2j

LES CHAÎNES DE CARACTÈRES

Les chaînes de caractères sont utilisées pour **stocker des séquences de caractères,** c'est-à-dire **du texte**. Elles sont délimitées par des guillemets simples ou doubles :

```
ma_chaine = "Hello, World!"
autre_chaine = 'Hello happiness'
```

On peut **concaténer** des chaînes de caractères avec l'opérateur + :

```
prenom = "John"
nom = "Doe"
nom_complet = prenom + " " + nom
# La variable nom_complet contient "John Doe"
```

La **concaténation** est l'opération qui consiste à **joindre deux chaînes de caractères pour en former une seule**.

On peut accéder aux **caractères individuels** d'une chaîne de caractères en utilisant **l'opérateur d'indexation [ ]**. Par exemple :

```
message = "Bonjour"
premier_caractere = message[0]
# La variable premier_caractere contient la lettre "B"
```

LES BOOLÉENS

Un **booléen** est une variable qui ne peut prendre que **deux valeurs** : `True` (vrai) ou `False` (faux). Les booléens sont très utiles dans les conditions, car ils permettent de vérifier si une expression est vraie ou fausse.

DÉCLARATION D'UN BOOLÉEN

Pour déclarer un booléen en Python, on utilise les mots-clés True et False :

```
a = True
b = False
```

OPÉRATEURS LOGIQUES

Les opérateurs logiques permettent de combiner plusieurs booléens entre eux. Les opérateurs logiques de base sont :

- **and** : l'opérateur ET. Retourne True **si les deux booléens sont vrais**.
- **or** : l'opérateur OU. Retourne True si **au moins l'un des deux booléens est vrai**.
- **not** : l'opérateur NON. **Inverse** le booléen.

Voici quelques exemples :

```
a = True
b = False

# ET logique
print(a and b)  # False

# OU logique
print(a or b)  # True

# NON logique
print(not a)  # False
```

LES LISTES

Les listes sont utilisées pour stocker des **séquences** d'éléments, qui peuvent être de différents types. Les listes en Python sont donc des **collections ordonnées d'éléments**, qui peuvent être de types différents. Les listes sont **définies entre crochets**, avec les éléments **séparés par des virgules**. Elles sont délimitées par des crochets :

```
ma_liste = [1, 2, "trois", True]
```

On peut **accéder aux éléments individuels** d'une liste en utilisant l'opérateur d'indexation **[ ]**. Par exemple :

```
nombres = [1, 2, 3, 4, 5]
premier_nombre = nombres[0]   # La variable premier_nombre contient le nombre 1
```

On peut **ajouter des éléments** à une liste avec la méthode **`append()`** :

```
nombres = [1, 2, 3]
nombres.append(4)   # La liste nombres contient maintenant [1, 2, 3, 4]
```

On peut également supprimer des éléments d'une liste avec la méthode **`remove()`** :

```
nombres = [1, 2, 3, 4]
nombres.remove(3)   # La liste nombres contient maintenant [1, 2, 4]
```

LES CONDITIONS

INTRODUCTION

Les **conditions** sont des **structures de contrôle** en Python qui **permettent d'exécuter un bloc d'instructions uniquement si une certaine condition est vraie**. Les conditions sont des **éléments clés** de la programmation, car elles permettent aux programmes de **prendre des décisions en fonction des données qu'ils traitent**. Dans ce cours, nous allons explorer les différentes utilisations possibles des conditions en Python, ainsi que des exemples concrets pour mieux comprendre leur fonctionnement.

LES EXPRESSIONS BOOLÉENNES

Les conditions en Python sont basées sur des **expressions booléennes, qui sont des expressions qui ont une valeur booléenne, c'est-à-dire True (vrai) ou False (faux).** Les expressions booléennes sont créées en utilisant des **opérateurs de comparaison, tels que == (égal à), != (différent de), < (inférieur à), > (supérieur à), <= (inférieur ou égal à), >= (supérieur ou égal à),** ainsi que des **opérateurs logiques, tels que and, or et not.**

UTILISATION DE L'INSTRUCTION IF

L'instruction `if` est la première structure de contrôle conditionnelle en Python. Elle permet d'exécuter un bloc d'instructions si une certaine condition est vraie. Par exemple, si nous avons une variable x qui contient un nombre entier, nous pouvons utiliser une instruction if pour afficher un message si x est supérieur à 10 :

```
x = 15
if x > 10:
    print("x est supérieur à 10")
```

Dans cet exemple, nous utilisons une instruction `if` pour vérifier si x est supérieur à 10. Si la condition est vraie, le bloc d'instructions indenté sous l'instruction if est exécuté, ce qui affiche le message "x est supérieur à 10".

UTILISATION DE L'INSTRUCTION IF...ELSE

L'instruction **`if...else`** est une structure de contrôle conditionnelle en Python qui permet d'exécuter un bloc d'instructions si une certaine condition est vraie, et **un autre bloc d'instructions si la condition est fausse**. Par exemple, si nous avons une variable x qui contient un nombre entier, nous pouvons utiliser une instruction if...else pour afficher un message différent en fonction de la valeur de x :

```
x = 5

if x > 10:
    print("x est supérieur à 10")
else:
    print("x est inférieur ou égal à 10")
```

Dans cet exemple, nous utilisons une instruction **`if...else`** pour vérifier si x est supérieur à 10. Si la condition est vraie, le premier bloc d'instructions indenté sous l'instruction if est exécuté, ce qui affiche le message "x est supérieur à 10". Sinon, le bloc d'instructions indenté sous l'instruction else est exécuté, ce qui affiche le message "x est inférieur ou égal à 10".

UTILISATION DE L'INSTRUCTION IF...ELIF...ELSE

L'instruction **`if...elif...else`** est une structure de contrôle conditionnelle en Python qui **permet de tester plusieurs conditions successives**. Elle est utilisée lorsque nous **avons plus de deux options et que chaque option doit être testée séparément**. Par exemple, si nous avons une variable x qui contient un nombre entier, nous pouvons utiliser une instruction if...elif...else pour afficher un message différent en fonction de la valeur de x :

```
x = 5

if x > 10:
    print("x est supérieur à 10")
elif x > 5:
    print("x est compris entre 6 et 10")
else:
    print("x est inférieur ou égal à 5")
```

Dans cet exemple, nous utilisons une instruction **`if...elif...else`** pour tester trois conditions successives. Si x est supérieur à 10, le premier bloc d'instructions indenté sous l'instruction if est exécuté, ce qui affiche le message "x est supérieur à 10". Si x est compris

entre 6 et 10, le bloc d'instructions indenté sous l'instruction elif est exécuté, ce qui affiche le message "x est compris entre 6 et 10". Sinon, le bloc d'instructions indenté sous l'instruction else est exécuté, ce qui affiche le message "x est inférieur ou égal à 5".

UTILISATION DES OPÉRATEURS LOGIQUES

Les **opérateurs logiques** (`and, or, not`) sont utilisés pour **combiner des expressions booléennes**. Par exemple, si nous avons deux variables x et y qui contiennent des nombres entiers, nous pouvons utiliser l'opérateur and pour vérifier si x est supérieur à 10 et y est inférieur à 5 :

```
x = 15
y = 3

if x > 10 and y < 5:
    print("x est supérieur à 10 et y est inférieur à 5")
else:
    print("au moins une des conditions n'est pas remplie")
```

Dans cet exemple, nous utilisons l'opérateur and pour combiner deux conditions : x > 10 **et** y < 5. **Si les deux conditions sont vrai**es, le bloc d'instructions indenté sous l'instruction if est exécuté, ce qui affiche le message "x est supérieur à 10 et y est inférieur à 5". Sinon, le bloc d'instructions indenté sous l'instruction else est exécuté, ce qui affiche le message "au moins une des conditions n'est pas remplie".

CONCLUSION

Les conditions en Python sont des structures de contrôle qui permettent aux programmes de prendre des décisions en fonction des données qu'ils traitent. Les expressions booléennes, les instructions if, if...else, if...elif...else, les opérateurs logiques et l'instruction ternaire sont des outils importants pour écrire des programmes conditionnels en Python.

BOUCLES EN PYTHON

Les boucles sont **utilisées pour répéter des actions plusieurs fois** dans un programme. Il existe deux types de boucles en Python : la boucle for et la boucle while.

BOUCLE FOR

La boucle for est utilisée pour **parcourir une séquence**, comme une liste, un tuple ou une chaîne de caractères. La syntaxe de la boucle for est la suivante :

```
for variable in sequence:
    instructions
```

La `variable` prendra la valeur de chaque élément de la séquence à chaque itération de la boucle. Les instructions à l'intérieur de la boucle seront exécutées **pour chaque élément de la séquence**.

Une **itération** correspond à **un passage** dans la boucle.

Exemple :

```
liste = [1, 2, 3, 4, 5]
for nombre in liste:
    print(nombre)
Résultat :
1
2
3
4
5
```

BOUCLE WHILE

La boucle `while` est utilisée pour **répéter une action tant qu'une condition est vraie**. La syntaxe de la boucle while est la suivante :

```
while condition:
    instructions
```

Les instructions à l'intérieur de la boucle seront **répétées tant que la condition est vraie**.

Exemple :

```
nombre = 1
while nombre <= 5:
    print(nombre)
    nombre += 1
Résultat :
1
2
3
4
5
```

CONCLUSION

Les boucles sont un outil **puissant** en Python pour **répéter des actions plusieurs fois** dans un programme. En utilisant des `boucles` for et `while` vous pouvez écrire des programmes efficaces et élégants qui résolvent une variété de problèmes.

LES BOUCLES FOR EN DÉTAIL

La boucle `for` est une structure de contrôle qui permet de parcourir des éléments dans une séquence ou une collection de données, comme une liste, un tuple, un dictionnaire ou une chaîne de caractères. Elle peut également être utilisée pour **exécuter des instructions un nombre spécifique de fois en utilisant la fonction** `range`**().** Dans ce cours, nous allons apprendre à utiliser la boucle for en Python, en explorant ses différentes utilisations et exemples concrets.

SYNTAXE DE LA BOUCLE FOR

La syntaxe de la boucle for en Python est la suivante :

```
for variable in sequence:
    instructions
```

- **variable** : c'est la variable qui prendra la valeur de chaque élément de la séquence à chaque itération de la boucle. Cette variable est utilisée pour accéder à **l'élément actuel** dans la séquence.
- **sequence** : c'est la séquence de données que nous voulons **parcourir**. Cela peut être une liste, un tuple, un dictionnaire ou une chaîne de caractères.
- **instructions** : ce sont les instructions qui seront exécutées pour chaque élément de la séquence.

UTILISATION DE LA BOUCLE FOR POUR PARCOURIR UNE SÉQUENCE

La boucle `for` est souvent utilisée pour parcourir une séquence et exécuter des instructions pour chaque élément de cette séquence. Par exemple, si nous avons une liste de nombres, nous pouvons utiliser une boucle for pour afficher chaque nombre de cette liste :

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print(number)
```

Dans cet exemple, nous avons une liste de nombres et nous utilisons la boucle for pour parcourir chaque nombre de la liste et l'afficher.

UTILISATION DE LA BOUCLE FOR POUR PARCOURIR UNE CHAÎNE DE CARACTÈRES

La boucle **`for`** peut également être utilisée pour parcourir une chaîne de caractères. Par exemple, si nous avons une chaîne de caractères, nous pouvons utiliser une boucle for pour **afficher chaque caractère** de cette chaîne :

```
message = "Hello World"
for letter in message:
    print(letter)
```

Dans cet exemple, nous avons une chaîne de caractères et nous utilisons la boucle for pour parcourir chaque caractère de la chaîne et l'afficher.

UTILISATION DE LA BOUCLE FOR AVEC LA FONCTION RANGE()

La boucle for peut également être utilisée pour **exécuter des instructions un nombre spécifique de fois en utilisant la fonction `range()`**.

La fonction **`range`**() **génère une séquence de nombres**, et nous pouvons utiliser une boucle for pour parcourir cette séquence et exécuter des instructions pour chaque nombre. Par exemple, si nous voulons **exécuter une instruction 5 fois**, nous pouvons utiliser la fonction range() avec une boucle for :

```
for i in range(5):
    print("Hello")
```

Dans cet exemple, nous utilisons la fonction range() avec une boucle for pour exécuter l'instruction "Hello" 5 fois.

LA FONCTION RANGE

La fonction **`range()`** en Python est utilisée pour **générer une séquence de nombres**, généralement utilisée pour l'itération à travers une boucle.

Nous avons déjà vu l'utilisation de base précédemment (`range(5)`).

La **syntaxe complète** de la fonction **`range()`** est la suivante:

```
range([start], stop[, step])
```

où

- **start** est le **premier** nombre de la séquence (par défaut 0)
- **stop** est le **dernier** nombre de la séquence (**non inclus**)
- et **step** est **l'incrément** entre les nombres (par défaut 1).

Voici quelques exemples pour mieux comprendre la fonction range():

```
# Affiche les nombres de 0 à 4
for i in range(5):
    print(i)

# Affiche les nombres de 2 à 6
for i in range(2, 7):
    print(i)

# Affiche les nombres pairs de 0 à 10
for i in range(0, 11, 2):
    print(i)

# Affiche les nombres impairs de 1 à 9
for i in range(1, 10, 2):
    print(i)
```

Dans le premier exemple, range(5) génère une séquence de nombres allant de 0 à 4 (inclus) avec un pas de 1. La boucle for itère à travers cette séquence et affiche chaque nombre.

Dans le deuxième exemple, range(2, 7) génère une séquence de nombres allant de 2 à 6 (inclus) avec un pas de 1. La boucle for itère à travers cette séquence et affiche chaque nombre.

Dans le troisième exemple, range(0, 11, 2) génère une séquence de nombres allant de 0 à 10 (inclus) avec un pas de 2. La boucle for itère à travers cette séquence et affiche chaque nombre pair.

Dans le quatrième exemple, range(1, 10, 2) génère une séquence de nombres allant de 1 à 9 (inclus) avec un pas de 2. La boucle for itère à travers cette séquence et affiche chaque nombre impair.

CONCLUSION

La boucle for est une structure de contrôle puissante en Python qui permet de parcourir des éléments dans une séquence ou une collection de données, ou d'exécuter des instructions un nombre spécifique de fois en utilisant la fonction range(). Avec les exemples et les utilisations présentés dans ce cours, vous avez maintenant les connaissances de base pour utiliser la boucle for dans vos projets en Python.

INTRODUCTION AUX FONCTIONS EN PYTHON

En programmation, une fonction est **un bloc de code réutilisable qui effectue une tâche spécifique**. Les fonctions permettent de **diviser le code en parties plus petites et plus gérables**, **facilitant ainsi la lecture**, la compréhension et la maintenance du code.

En Python, les fonctions sont définies à l'aide du mot clé **`def`**, suivi du nom de la fonction, de ses paramètres éventuels et de deux points (:). Le code de la fonction doit être indenté.

EXEMPLE D'UNE FONCTION SIMPLE EN PYTHON

Voici un exemple d'une fonction simple qui prend un paramètre nom et qui imprime un message de salutation :

```
def saluer():
    print("Bonjour !")
```

APPELER UNE FONCTION EN PYTHON

Une fois qu'une fonction est définie, **vous pouvez l'appeler** en **utilisant son nom et en fournissant les arguments requis entre parenthèses**. Par exemple, pour appeler la fonction saluer() que nous avons définie précédemment, vous pouvez faire :

```
saluer()
```

Cela imprime le message "Bonjour !".

VALEURS DE RETOUR DES FONCTIONS EN PYTHON

Les fonctions peuvent également **renvoyer des valeurs** en utilisant le mot clé **`return`**. Par exemple, voici une fonction qui calcule la somme de deux nombres et renvoie le résultat :

```
def additionner(a, b):
    somme = a + b
    return somme
```

Vous pouvez appeler cette fonction et stocker le résultat dans une variable comme ceci :

```
x = additionner(3, 4)
print(x) # imprime 7
```

ARGUMENTS DE FONCTIONS

Lorsque nous écrivons une fonction en Python, il est courant de **vouloir passer des informations ou des valeurs à la fonction**. C'est là que les arguments de fonction interviennent. Les arguments permettent à une fonction de **prendre des entrées spécifiques**, **de les utiliser dans la fonction** et de retourner une sortie en fonction de ces entrées.

Les fonctions peuvent donc **prendre des arguments**, qui sont **des valeurs que la fonction utilise** pour effectuer des calculs ou des actions. Les arguments sont **définis entre les parenthèses lors de la définition de la fonction**.

Il existe deux types d'arguments : les arguments **obligatoires** et les arguments **optionnels**.

ARGUMENTS OBLIGATOIRES

Les arguments **obligatoires** sont **nécessaires** pour que la fonction fonctionne correctement. Lorsque vous appelez une fonction qui a des arguments obligatoires, vous **devez** leur passer des valeurs correspondantes.

Voici un exemple de fonction avec des arguments obligatoires :

```
def addition(a, b):
    result = a + b
    print(result)


addition(2, 3)  # affiche 5
```

Dans cet exemple, la fonction addition prend **deux arguments obligatoires, a et b**. Lorsque nous appelons la fonction avec addition(2, 3), les valeurs 2 et 3 sont passées aux arguments a et b, respectivement.

ARGUMENTS OPTIONNELS

Les arguments optionnels ne sont **pas nécessaires** pour que la fonction fonctionne correctement**. Ils ont des valeurs par défaut**, qui sont utilisées si aucune valeur n'est passée pour cet argument lors de l'appel de la fonction.

Voici un exemple de fonction avec des arguments optionnels :

```
def multiplication(a, b=2):
    result = a * b
```

```
        print(result)

multiplication(3)  # affiche 6
multiplication(3, 4)  # affiche 12
```

Dans cet exemple, la fonction multiplication prend un argument obligatoire a et un argument optionnel b**. Si aucun argument n'est fourni pour b lors de l'appel de la fonction, la valeur par défaut de 2 sera utilisée**. Si un argument est fourni pour b, cette valeur sera utilisée à la place de la valeur par défaut.

CONCLUSION

Les fonctions sont des éléments clés de la programmation en Python, car elles permettent de **diviser le code** en parties plus petites et plus gérables, facilitant ainsi la lecture, la compréhension et la maintenance du code. Les fonctions peuvent prendre des **paramètres**, **renvoyer des valeurs** et avoir des paramètres optionnels avec des valeurs par défaut.

PORTÉE DES VARIABLES EN PYTHON

La **portée** des variables se réfère à la **région du programme où une variable est accessible**. En Python, la portée des variables est **déterminée par le bloc de code dans lequel elle est déclarée**. Il existe deux types de portées de variables en Python : globale et locale.

PORTÉE GLOBALE

Une variable **définie en dehors de toutes les fonctions** est considérée comme ayant une **portée globale**. Cela signifie que la variable **est accessible de partout dans le programme**.

Exemple :

```
x = 10
def my_function():
  print(x)

my_function()
print(x)
```

Dans cet exemple, la variable x est définie en dehors de la fonction my_function(), ce qui en fait une variable globale. La fonction my_function() **peut accéder à la variable x** et l'afficher en appelant simplement print(x). La même variable x peut également être utilisée en dehors de la fonction my_function() et est affichée en appelant simplement print(x).

PORTÉE LOCALE

Une variable déclarée **à l'intérieur d'une fonction** est considérée comme ayant une **portée locale**. Cela signifie que la variable n'est accessible que dans la fonction où elle est déclarée.

Exemple :

```
def my_function():
  y = 5
  print(y)

my_function()
```

Dans cet exemple, la variable y est déclarée à l'intérieur de la fonction my_function(), ce qui en fait une variable locale. **La variable** `y` **ne peut être utilisée qu'à l'intérieur de la fonction my_function() et n'est pas accessible en dehors de cette fonction**.

VARIABLES GLOBALES ET LOCALES AVEC LE MÊME NOM

Si une variable locale et une variable globale ont le même nom, **la variable locale prendra le dessus** et sera utilisée à la place de la variable globale dans la fonction où elle est déclarée.

Exemple :

```
x = 10
def my_function():
  x = 5
  print(x)

my_function()
print(x)
```

Dans cet exemple, la variable x est définie à l'extérieur de la fonction my_function(), mais elle est également redéfinie à l'intérieur de cette fonction. Lorsque my_function() est appelée, la variable locale x avec une valeur de 5 est imprimée. Lorsque la fonction est terminée, la variable globale x avec une valeur de 10 est imprimée.

UTILISATION DE LA DÉCLARATION "GLOBAL"

La déclaration `global` permet **de modifier la portée d'une variable à l'intérieur d'une fonction**, **en lui donnant une portée globale**.

Exemple :

```
x = 10
def my_function():
  global x
  x = 5
  print(x)

my_function()
```

```
print(x)
```

Dans cet exemple, la variable x est définie à l'extérieur de la fonction `my_function()`. La déclaration `global x` à l'intérieur de la fonction permet de modifier la portée de la variable `x` en lui donnant une portée globale. Lorsque `my_function()` est appelée, la variable `x` est définie sur 5, puis immédiatement affichée à l'écran. Le résultat affiché est 5. Ensuite, le programme sort de la fonction et affiche à nouveau le contenu de la variable x, qui vaut toujours 5.

AJOUTER DES FONCTIONNALITÉS À VOS PROGRAMMES

En Python, l'importation de modules est essentielle pour étendre les fonctionnalités de votre programme. Il existe plusieurs façons d'importer des modules, chacune ayant ses avantages et ses inconvénients.

IMPORTER DES MODULES

L'IMPORTATION SIMPLE

La méthode la plus simple pour **importer un module** est d'utiliser l'instruction `import`. Lorsque vous importez un module avec l'instruction `import`, Python recherche le module dans le `sys.path`, qui est une liste de répertoires où Python recherche les modules.

Voici un exemple d'importation simple :

```
import math
print(math.pi)
```

Dans cet exemple, nous importons le module `math` et affichons la valeur de `math.pi`.`math.pi` est une constante qui représente la valeur de π (3,141592653589793).

Remarquez qu'avec cette façon d'importer un module, vous devez **préfixer** l'utilisation de votre fonction ou de vos constantes par le nom du module (`math.pi, math.ceil()` …)

L'IMPORTATION AVEC ALIAS

Parfois, les noms des modules sont très longs ou difficiles à retenir. Dans ce cas, vous pouvez **importer un module sous un alias** en utilisant l'instruction `as`.

Voici un exemple :

```
import pandas as pd
data = pd.read_csv('data.csv')
```

Dans cet exemple, nous importons le module `pandas` et créons un alias `pd`. Nous pouvons maintenant utiliser `pd` pour faire référence à pandas dans notre code.

L'IMPORTATION SPÉCIFIQUE

Il est également possible d'**importer des fonctions spécifiques d'un module**. Cela peut être utile lorsque vous n'avez besoin que de **quelques fonctions** d'un module et que vous ne voulez pas importer tout le module.

Voici un exemple :

```
from math import pi
print(pi)
```

Dans cet exemple, nous importons uniquement la constante `pi` du module `math`.

L'IMPORTATION DE TOUS LES SYMBOLES

Lorsque vous utilisez l'instruction **`from module import *`**, vous importez tous les symboles du module dans l'espace de noms courant. Autrement dit, vous pouvez utiliser dans votre code toutes les fonctions et constantes du module sans devoir préfixer vos appels par le nom du module.

Cependant, cette méthode n'est généralement **pas recommandée** car elle peut rendre le code difficile à lire et à maintenir.

Voici un exemple :

```
from math import *
print(pi)
print(sqrt(16))
```

Dans cet exemple, nous importons tous les symboles du module `math`.

ARRONDIR DES VALEURS

Python offre plusieurs méthodes pour **arrondir des nombres**. Voici les plus courantes :

LA FONCTION ROUND()

La fonction **`round()`** arrondit un nombre à la **précision spécifiée**. Si la précision n'est pas spécifiée, le nombre est arrondi à **l'entier le plus proche** selon les règles de l'**arrondi classique**.

L'arrondi classique, également appelé **arrondi arithmétique** ou arrondi bancaire, est une méthode courante pour arrondir des nombres dans de nombreux domaines, tels que la

finance, la comptabilité et les sciences. Cette méthode consiste à arrondir un nombre à l'entier le plus proche en utilisant la règle suivante :

- Si la première décimale à droite du chiffre à arrondir est **inférieure à 5**, le chiffre est **arrondi vers le bas** (à l'entier immédiatement inférieur).

- Si la première décimale à droite du chiffre à arrondir est **supérieure ou égale à 5**, le chiffre est **arrondi vers le haut** (à l'entier immédiatement supérieur).

Par exemple, pour arrondir le nombre **3,14159** à deux décimales en utilisant l'arrondi classique, nous examinerions la troisième décimale :

La troisième décimale est 1, qui est inférieure à 5, donc nous arrondirions vers le bas.

Le résultat serait 3,14.

De même, pour arrondir le nombre **3,1459** à deux décimales en utilisant l'arrondi classique, nous examinerions également la troisième décimale :

La troisième décimale est 5, qui est supérieure ou égale à 5, donc nous arrondirions vers le haut.

Le résultat serait 3,15.

Voici un exemple :

```
x = 3.14159
print(round(x)) # Sortie : 3
print(round(x, 2)) # Sortie : 3.14
```

Dans cet exemple, nous arrondissons le nombre x à deux décimales en utilisant la fonction round().

LES FONCTIONS MATHÉMATIQUES

Python possède également plusieurs fonctions mathématiques pour arrondir des nombres : notamment `math.floor()` et **math.ceil()**. Comme vous devez le deviner, cette fonctions sont disponibles dans le module `math` de Python, qu'il faut **importer dans votre code** avant de pouvoir utiliser ces fonctions.

La fonction `math.floor()` retourne **le plus grand entier qui est inférieur ou égal** au nombre donné.

Voici un exemple :

```
import math
x = 3.14159
print(math.floor(x)) # Sortie : 3
```

La fonction `math.ceil()` retourne le **plus petit entier qui est supérieur ou égal** au nombre donné.

Voici un exemple :

```
import math
x = 3.14159
print(math.ceil(x)) # Sortie : 4
```

GÉNÉRER DES NOMBRE ALÉATOIRES

La génération de **nombres aléatoires** est souvent utilisée en informatique pour créer des simulations, des jeux et des analyses de données. Python dispose d'une bibliothèque intégrée, appelée `random`, qui permet de générer des nombres aléatoires.

Un **nombre aléatoire** est un nombre qui est **généré au hasard** sans qu'il y ait de modèle prévisible. En d'autres termes, il n'y a aucune méthode fiable pour prédire quelle sera la valeur d'un nombre aléatoire donné.

Les nombres aléatoires sont utilisés dans **de nombreux domaines**, notamment en mathématiques, en informatique, en science et en jeux de hasard. Les applications des nombres aléatoires incluent la simulation de systèmes complexes, la génération de **clés de cryptage**, la création de jeux équitables, l'analyse de données et la modélisation de phénomènes naturels.

LA FONCTION RANDINT()

La fonction `randint()` de la bibliothèque `random` permet de générer un nombre entier **aléatoire** dans une **plage spécifiée**. La plage est définie en utilisant **deux arguments**, qui représentent respectivement la borne inférieure et la borne supérieure de la plage.

Voici un exemple :

```
import random
nombre_entier_aleatoire = random.randint(0, 10)
print(nombre_entier_aleatoire)
```

Dans cet exemple, nous importons la bibliothèque **random** et utilisons la fonction `randint()` pour générer un nombre entier aléatoire entre **0 inclus et 10 inclus**. Le nombre entier aléatoire est stocké dans la variable nombre_entier_aleatoire et affiché à l'aide de la fonction print().

LA MÉTHODE CHOICE()

La méthode `choice()` de la bibliothèque `random` permet de **choisir un élément aléatoire dans une liste** donnée.

Voici un exemple :

```
import random
nombres = [1, 2, 3, 4, 5]
nombre_aleatoire = random.choice(nombres)
print(nombre_aleatoire)
```

Dans cet exemple, nous avons créé une liste de nombres et avons utilisé la méthode choice() pour choisir un nombre aléatoire dans cette liste. Le nombre aléatoire est stocké dans la variable nombre_aleatoire et affiché à l'aide de la fonction print().