Programmation avec Python

MASTER SITBD

Pr. Lotfi ELAACHAK
Département Génie Informatique

2023 — 2024

Table des matières

Chapitre 1

Les bases de Python

1.1 Introduction

Python est un langage de programmation généraliste, dynamique, de haut niveau et interprété. Il prend en charge l'approche de programmation orientée objet pour développer des applications. Il est simple et facile à apprendre et fournit de nombreuses structures de données de haut niveau.

Python est un langage de script facile à apprendre mais puissant et polyvalent, ce qui le rend attrayant pour le développement d'applications.

La syntaxe et le typage dynamique de Python avec sa nature interprétée en font un langage idéal pour les scripts et le développement rapide d'applications.

Python prend en charge plusieurs modèles de programmation, y compris les styles de programmation orientés objet, impératifs et fonctionnels ou procéduraux.

Python n'est pas destiné à fonctionner dans un domaine particulier, tel que la programmation Web. C'est pourquoi il est connu sous le nom de langage de programmation polyvalent car il peut être utilisé avec le Web, l'entreprise, la CAO 3D, etc.

Nous n'avons pas besoin d'utiliser des types de données pour déclarer la variable car elle est typée dynamiquement afin que nous puissions écrire a=10 pour affecter une valeur entière dans une variable entière.

Python accélère le développement et le débogage car aucune étape de compilation n'est incluse dans le développement Python et le cycle édition-test-débogage est très rapide.

1.1.1 Pourquoi apprendre Python ?

Python fournit de nombreuses fonctionnalités utiles au programmeur. Ces caractéristiques en font la langue la plus populaire et la plus largement utilisée. Nous avons énuméré ci-dessous quelques fonctionnalités essentielles de Python.

- Facile à utiliser et à apprendre
- Langage expressif
- Langue interprétée
- Langage orienté objet
- Langage Open Source
- Extensible
- Large gamme de bibliothèques et de frameworks

1.1.2 Python First Program

Contrairement aux autres langages de programmation, Python offre la possibilité d'exécuter le code en utilisant quelques lignes. Par exemple - Supposons que nous voulions imprimer le programme "Hello World" en Java ; il faudra trois lignes pour l'imprimer.

Hello word

```
print("Hello word")
```

1.2 Variable et type de données

Python ne nous oblige pas à déclarer une variable avant de l'utiliser dans l'application. Il nous permet de créer une variable au moment voulu.

Nous n'avons pas besoin de déclarer explicitement la variable en Python. Lorsque nous affectons une valeur à la variable, cette variable est déclarée automatiquement.

L'opérateur égal (=) est utilisé pour affecter une valeur à une variable.

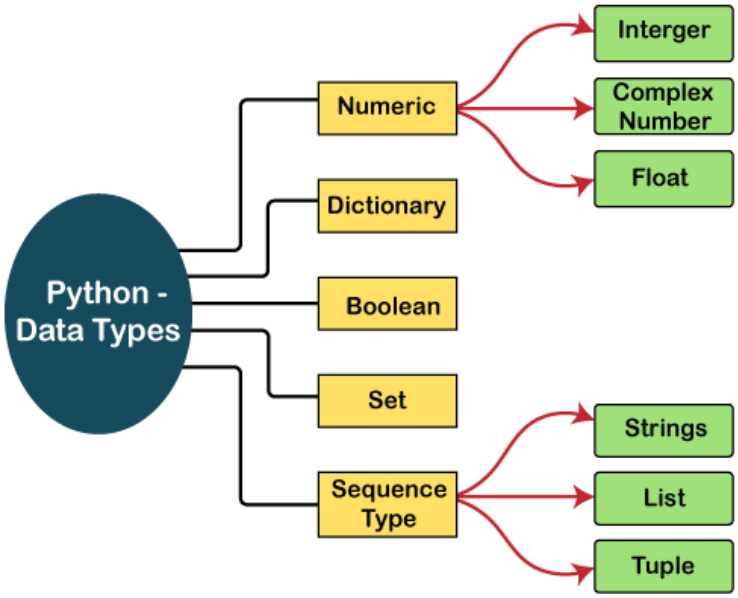
Variables et affectation

```
name = "Devansh"
age = 20
marks = 80.50

print(name)
print(age)
print(marks)
```

Les variables peuvent contenir des valeurs, et chaque valeur a un type de données. Python est un langage à typage dynamique ; par conséquent, nous n'avons pas besoin de définir le type de la variable lors de sa déclaration. L'interpréteur lie implicitement la valeur à son type.

Type de données

```
a=10
b="Hi Python"
c = 10.5
print(type(a))
print(type(b))
print(type(c))
```



1.3 Instructions Python If-else

Type de données

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even")
else :
    print("Number is odd")
```

elif

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

Operateur logique

```
 marks = int(input("Enter the marks? "))
if marks > 85 and marks <= 100:
   print("Congrats ! you scored grade A ...")
elif marks > 60 and marks <= 85:
   print("You scored grade B + ...")
elif marks > 40 and marks <= 60:
```

```
   print("You scored grade B ...")
elif (marks > 30 and marks <= 40):
   print("You scored grade C ...")
else:
   print("Sorry you are fail ?")
```

1.4 Instructions iteratives

1.4.1 La boucle for

Boucle for

```
for i in range(10):
    print(i,end = ' ')

n = int(input("Enter the number "))
for i in range(1,11):
    c = n*i
    print(n,"*",i,"=",c)

n = int(input("Enter the number "))
for i in range(2,n,2):
    print(i)
```

1.4.2 La boucle while

Boucle while

```
i=1
While(i<=10):
    print(i)
    i=i+1
```

Boucle while et else

```
 i=1
while(i<=5):
    print(i)
    i=i+1
    if(i==3):
        break
else:
    print("The while loop exhausted")
```

1.5 Fonction Python

Les fonctions sont l'aspect le plus important d'une application. Une fonction peut être définie comme le bloc organisé de code réutilisable, qui peut être appelé chaque fois que nécessaire.

Python nous permet de diviser un grand programme en blocs de construction de base appelés fonction. La fonction contient l'ensemble d'instructions de programmation entouré de . Une fonction peut être appelée plusieurs fois pour permettre la réutilisation et la modularité du programme Python.

Les fonctions Python présentent les avantages suivants :

- En utilisant des fonctions, nous pouvons éviter de réécrire la même logique/code encore et encore dans un programme.
- Nous pouvons appeler des fonctions Python plusieurs fois dans un programme et n'importe où dans un programme.
- Nous pouvons facilement suivre un grand programme Python lorsqu'il est divisé en plusieurs fonctions.
- La réutilisabilité est la principale réalisation des fonctions Python.

Function

```
#function definition
def hello_world():
    print("hello world")
# function calling
hello_world()
```

Function Avec param

```
#Python function to calculate the sum of two variables
#defining the function
def sum (a,b):
    return a+b;

#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))

#printing the sum of a and b
print("Sum = ",sum(a,b))
```

1.5.1 Appel par référence en Python

En Python, appeler par référence signifie passer la valeur réelle comme argument dans la fonction. Toutes les fonctions sont appelées par référence, c'est-à-dire que toutes les modifications apportées à la référence à l'intérieur de la fonction reviennent à la valeur d'origine référencée par la référence.

Passage par reference

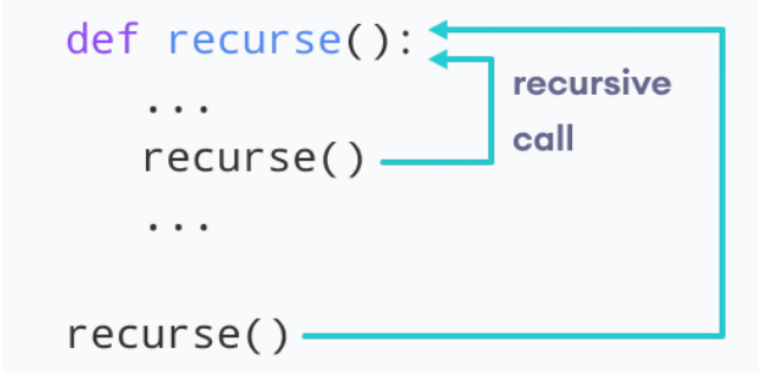
```python
#defining the function
def change_list(list1):
    list1.append(20)
    list1.append(30)
    print("list inside function = ",list1)

#defining the list
list1 = [10,30,40,50]

#calling the function
change_list(list1)
print("list outside function = ",list1)
```

Passage d'un objet immutable

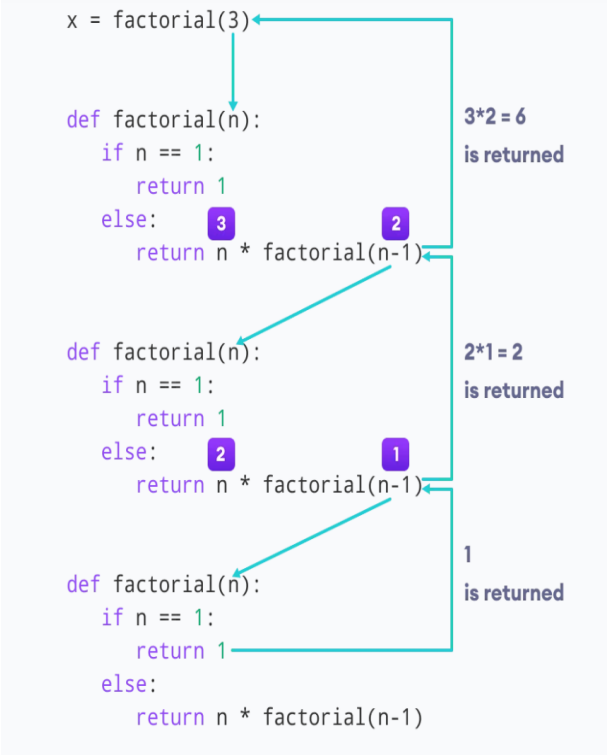
```python
#defining the function
def change_string (str):
    str = str + " Hows you "
    print("printing the string inside function :",str)

string1 = "Hi I am there"

#calling the function
change_string(string1)

print("printing the string outside function :",string1)
```

1.5.2 La récursivité

La récursivité est le processus de définition de quelque chose en termes de lui-même.
En Python, nous savons qu'une fonction peut appeler d'autres fonctions. Il est même possible que la fonction s'appelle elle-même. Ces types de construction sont appelés fonctions récursives.
L'image suivante montre le fonctionnement d'une fonction récursive appelée recurse.



```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```



1.6 Tableaux Python

Un tableau est défini comme une collection d'éléments stockés dans des emplacements de mémoire contigus. C'est un conteneur qui peut contenir un nombre fixe d'articles, et ces articles doivent être du même type. Un tableau est populaire dans la plupart des langages de programmation comme C/C++, JavaScript, etc.

Le tableau est une idée de stockage de plusieurs éléments du même type ensemble et il est plus facile de calculer la position de chaque élément en ajoutant simplement un décalage à la valeur de base.

Le tableau peut être géré en Python par un module nommé array. C'est utile lorsque nous devons manipuler uniquement des valeurs de données spécifiques. Voici les termes pour comprendre le concept d'un tableau :

— Élément - Chaque élément stocké dans un tableau est appelé un élément.

— Index - L'emplacement d'un élément dans un tableau a un index numérique, qui est utilisé pour identifier la position de l'élément.

Array

```
from array import *
arrayName = array(typecode, [initializers])

import array as arr
a = arr.array('i', [2, 4, 6, 8])
print("First element:", a[0])
print("Second element:", a[1])
print("Last element:", a[-1])

# changing first element
numbers[0] = 0
print(numbers)    # Output: array('i', [0, 2, 3, 5, 7, 10])

# changing 3rd to 5th element
numbers[2:5] = arr.array('i', [4, 6, 8])
print(numbers)    # Output: array('i', [0, 2, 4, 6, 8, 10])

del number[2]                           # removing third element
print(number)                           # Output: array('i', [1, 2, 3, 4])

a=arr.array('d',[1.1 , 2.1 ,3.1,2.6,7.8])
b=arr.array('d',[3.7,8.6])
c=arr.array('d')
c=a+b
print("Array c = ",c)
```

1.7 Les chaînes de caractères Python

La chaîne Python est la collection des caractères entourés de guillemets simples, doubles ou triples. L'ordinateur ne comprend pas les caractères ; en interne, il stocke le caractère manipulé comme la combinaison des 0 et des 1.

Chaque caractère est codé dans le caractère ASCII ou Unicode. On peut donc dire que les chaînes Python sont aussi appelées collection de caractères Unicode.

String

```
#Using single quotes
str1 = 'Hello Python'
print(str1)
#Using double quotes
str2 = "Hello Python"
```

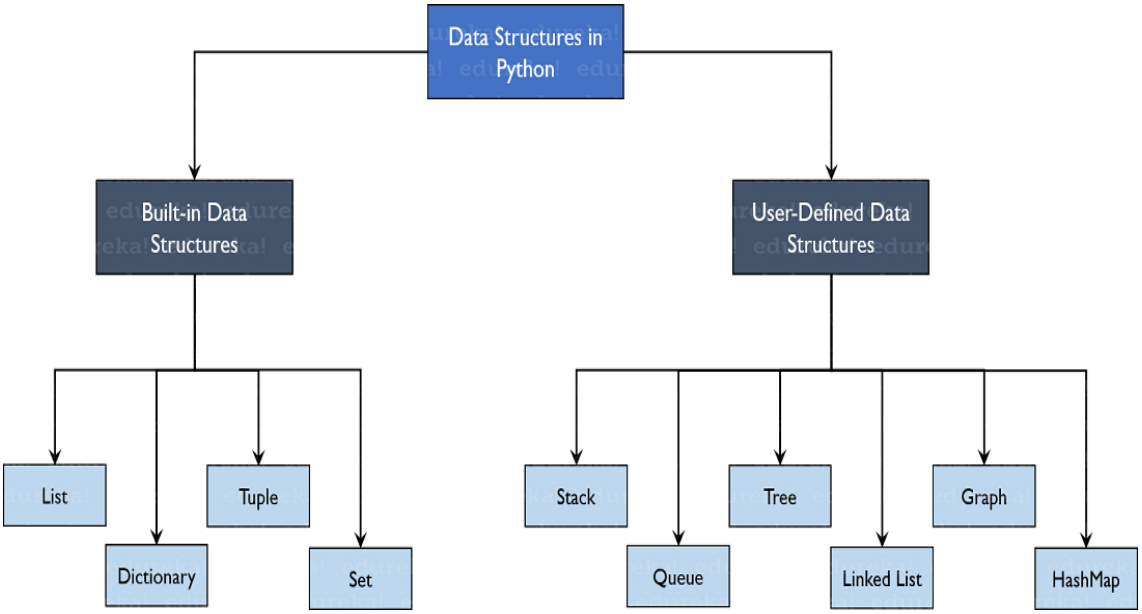
```
print(str2)


str = "HELLO"
print(str[0])
print(str[1])
print(str[2])
print(str[3])
print(str[4])
# It returns the IndexError because 6th index doesn't exist
print(str[6])

# Given String
str = "JAVATPOINT"
# Start 0th index to end
print(str[0:])
# Starts 1th index to 4th index
print(str[1:5])
# Starts 2nd index to 3rd index
print(str[2:4])
# Starts 0th to 2nd index
print(str[:3])
#Starts 4th to 6th index
print(str[4:7])

str = "Hello"
str1 = " world"
print(str*3) # prints HelloHelloHello
print(str+str1)# prints Hello world
print(str[4]) # prints o
print(str[2:4]); # prints ll
print('w' in str) # prints false as w is not present in str
print('wo' not in str1) # prints false as wo is present in str1.
print(r'C://python37') # prints C://python37 as it is written
print("The string str : %s"%(str)) # prints The string str : Hello
```

Chapitre 2

Structures de données intégrées

L'organisation, la gestion et le stockage des données sont importants car cela permet un accès plus facile et des modifications efficaces. Les structures de données vous permettent d'organiser vos données de manière à vous permettre de stocker des collections de données, de les relier et d'effectuer des opérations sur elles en conséquence.

Python prend implicitement en charge les structures de données qui vous permettent de stocker et d'accéder aux données. Ces structures sont appelées List, Dictionary, Tuple et Set.



2.1 Python List

Les listes sont utilisées pour stocker des données de différents types de données de manière séquentielle. Des adresses sont attribuées à chaque élément de la liste, appelée Index. La valeur de l'index commence à 0 et se poursuit jusqu'au dernier élément appelé index positif. Il existe également une indexation négative qui commence à partir de -1 vous permettant d'accéder aux éléments du dernier au premier. Comprenons maintenant mieux les listes à l'aide d'un exemple de programme.

List

```
my_list = [] #create empty list
print(my_list)
my_list = [1, 2, 3, 'example', 3.132] #creating list with data
print(my_list)


my_list = [1, 2, 3]
print(my_list)
my_list.append([555, 12]) #add as a single element
```

```
print(my_list)
my_list.extend([234, 'more_example']) #add as different elements
print(my_list)
my_list.insert(1, 'insert_example') #add element i
print(my_list)


my_list = [1, 2, 3, 'example', 3.132, 10, 30]
del my_list[5] #delete element at index 5
print(my_list)
my_list.remove('example') #remove element with value
print(my_list)
a = my_list.pop(1) #pop element from list
print('Popped Element: ', a, ' List remaining: ', my_list)
my_list.clear() #empty the list
print(my_list)


my_list = [1, 2, 3, 'example', 3.132, 10, 30]
for element in my_list: #access elements one by one
    print(element)

print(my_list) #access all elements
print(my_list[3]) #access index 3 element
print(my_list[0:2]) #access elements from 0 to 1 and exclude 2
print(my_list[::-1]) #access elements in reverse


my_list = [1, 2, 3, 10, 30, 10]
print(len(my_list)) #find length of list
print(my_list.index(10)) #find index of element that occurs first
print(my_list.count(10)) #find count of the element
print(sorted(my_list)) #print sorted list but not change original
my_list.sort(reverse=True) #sort original list
print(my_list)


list1 = [12, 14, 16, 18, 20]
# repetition operator *
l = list1 * 2
print(l)

# concatenation of two lists
# declaring the lists
list1 = [12, 14, 16, 18, 20]
list2 = [9, 10, 32, 54, 86]
```

```
# concatenation operator +
l = list1 + list2
print(l)
```

2.2 Python Set

Les Sets sont une collection d'éléments non ordonnés qui sont uniques. Cela signifie que même si les données sont répétées plus d'une fois, elles ne seront entrées dans l'ensemble qu'une seule fois. Cela ressemble aux ensembles que vous avez appris en arithmétique. Les opérations sont également les mêmes qu'avec les ensembles arithmétiques. Un exemple de programme vous aiderait à mieux comprendre.

Set

```
my_set = {1, 2, 3, 4, 5, 5, 5} #create set
print(my_set)

my_set = {1, 2, 3}
my_set.add(4) #add element to set
print(my_set)

my_set = {1, 2, 3, 4}
my_set_2 = {3, 4, 5, 6}
print(my_set.union(my_set_2), '----------', my_set | my_set_2)
print(my_set.intersection(my_set_2), '----------', my_set & my_set_2)
print(my_set.difference(my_set_2), '----------', my_set - my_set_2)
print(my_set.symmetric_difference(my_set_2), '----------', my_set ^ my_set_2)
my_set.clear()
print(my_set)
```

2.3 Python Tuple

Les tuples sont les mêmes que les listes, à l'exception du fait que les données une fois entrées dans le tuple ne peuvent pas être modifiées quoi qu'il arrive. La seule exception est lorsque les données à l'intérieur du tuple sont modifiables, alors seulement les données du tuple peuvent être modifiées. L'exemple de programme vous aidera à mieux comprendre.

- Les tuples sont un type de données immuable, ce qui signifie que leurs éléments ne peuvent pas être modifiés après leur création.
- Chaque élément d'un tuple a un ordre spécifique qui ne changera jamais car les tuples sont des séquences ordonnées.

Tuple

```
my_tuple = (1, 2, 3) #create tuple
print(my_tuple)

my_tuple2 = (1, 2, 3, 'edureka') #access elements
for x in my_tuple2:
    print(x)
print(my_tuple2)
print(my_tuple2[0])
print(my_tuple2[:])
print(my_tuple2[3][4])

my_tuple = (1, 2, 3)
my_tuple = my_tuple + (4, 5, 6) #add elements
print(my_tuple)

my_tuple = (1, 2, 3, ['hindi', 'python'])
my_tuple[3][0] = 'english'
print(my_tuple)
print(my_tuple.count(2))
print(my_tuple.index(['english', 'python']))
```

2.4 Python Dictionary

Les dictionnaires sont utilisés pour stocker les paires clé-valeur. Pour mieux comprendre, pensez à un annuaire téléphonique où des centaines et des milliers de noms et leurs numéros correspondants ont été ajoutés. Maintenant, les valeurs constantes ici sont le nom et les numéros de téléphone qui sont appelés comme touches. Et les différents noms et numéros de téléphone sont les valeurs qui ont été transmises aux touches. Si vous accédez aux valeurs des touches, vous obtiendrez tous les noms et numéros de téléphone. C'est donc ce qu'est une paire clé-valeur. Et en Python, cette structure est stockée à l'aide de dictionnaires. Laissez-nous comprendre cela mieux avec un exemple de programme.

Dictionary

```
my_dict = {} #empty dictionary
print(my_dict)
my_dict = {1: 'Python', 2: 'Java'} #dictionary with elements
print(my_dict)

my_dict = {'First': 'Python', 'Second': 'Java'}
print(my_dict)
my_dict['Second'] = 'C++' #changing element
print(my_dict)
my_dict['Third'] = 'Ruby' #adding key-value pair
print(my_dict)
```

```
my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}
a = my_dict.pop('Third') #pop element
print('Value:', a)
print('Dictionary:', my_dict)
b = my_dict.popitem() #pop the key-value pair
print('Key, value pair:', b)
print('Dictionary', my_dict)
my_dict.clear() #empty dictionary
print('n', my_dict)

my_dict = {'First': 'Python', 'Second': 'Java'}
print(my_dict['First']) #access elements using keys
print(my_dict.get('Second'))

my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}
print(my_dict.keys()) #get keys
print(my_dict.values()) #get values
print(my_dict.items()) #get key-value pairs
print(my_dict.get('First'))
```

Chapitre 3

Programmation fonctionnelle en Python

La programmation fonctionnelle est un paradigme de programmation dans lequel nous essayons de tout lier dans le style des fonctions mathématiques pures. C'est un type déclaratif de style de programmation. Son objectif principal est « que résoudre » par opposition à un style impératif où l'objectif principal est « comment résoudre ». Il utilise des expressions au lieu d'instructions. Une expression est évaluée pour produire une valeur alors qu'une instruction est exécutée pour affecter des variables.

3.1 Concepts de la programmation fonctionnelle

Tout langage de programmation fonctionnel est censé suivre ces concepts.

— Fonctions pures : ces fonctions ont deux propriétés principales. Premièrement, ils produisent toujours la même sortie pour les mêmes arguments indépendamment de toute autre chose. Deuxièmement, ils n'ont pas d'effets secondaires, c'est-à-dire qu'ils modifient n'importe quel argument ou variable globale ou produisent quelque chose.

— Récursivité : Il n'y a pas de boucle « for » ou « while » dans les langages fonctionnels. L'itération dans les langages fonctionnels est implémentée par récursivité.

— Les fonctions sont de première classe et peuvent être d'ordre supérieur : les fonctions de première classe sont traitées comme des variables de première classe. Les variables de première classe peuvent être transmises aux fonctions en tant que paramètre, peuvent être renvoyées à partir de fonctions ou stockées dans des structures de données.

— Les variables sont immuables : En programmation fonctionnelle, nous ne pouvons pas modifier une variable après son initialisation. Nous pouvons créer de nouvelles variables, mais nous ne pouvons pas modifier les variables existantes.

3.2 Fonctions pures

Comme discuté ci-dessus, les fonctions pures ont deux propriétés.

— Il produit toujours la même sortie pour les mêmes arguments. Par exemple, 3+7 sera toujours 10 quoi qu'il arrive.
— Il ne change pas ou ne modifie pas la variable d'entrée.

La deuxième propriété est également connue sous le nom d'immutabilité. Le seul résultat de la fonction pure est la valeur qu'elle renvoie. Ils sont déterministes. Les programmes réalisés à l'aide de la programmation fonctionnelle sont faciles à déboguer car les fonctions pures n'ont pas d'effets secondaires ni d'E/S cachées. Les fonctions pures facilitent également l'écriture d'applications parallèles/concurrentes. Lorsque le code est écrit dans ce style, un compilateur intelligent peut faire beaucoup de choses - il peut paralléliser les instructions, attendre d'évaluer les résultats en cas de besoin et mémoriser les résultats puisque les résultats ne changent jamais tant que l'entrée

ne change pas.

Pure function

```
def pure_func(List):

    New_List = []

    for i in List:
        New_List.append(i**2)

    return New_List

# Driver's code
Original_List = [1, 2, 3, 4]
Modified_List = pure_func(Original_List)

print("Original List:", Original_List)
print("Modified List:", Modified_List)
```

3.3 Récursivité

Au cours de la programmation fonctionnelle, il n'y a pas de concept de boucle for ou de boucle while, à la place la récursivité est utilisée. La récursivité est un processus dans lequel une fonction s'appelle directement ou indirectement. Dans le programme récursif, la solution du cas de base est fournie et la solution du plus gros problème est exprimée en termes de petits problèmes. Une question peut se poser quel est le cas de base ? Le cas de base peut être considéré comme une condition qui indique au compilateur ou à l'interpréteur de quitter la fonction.

Récursivité

```
def Sum(L, i, n, count):

    # Base case
    if n <= i:
        return count

    count += L[i]

    # Going into the recursion
    count = Sum(L, i + 1, n, count)

    return count

# Driver's code
L = [1, 2, 3, 4, 5]
count = 0
n = len(L)
```

```
print(Sum(L, 0, n, count))
```

3.4 Les fonctions sont de première classe et peuvent être d'ordre supérieur

Les objets de première classe sont traités uniformément. Ils peuvent être stockés dans des structures de données, transmis en tant qu'arguments ou utilisés dans des structures de contrôle. On dit qu'un langage de programmation prend en charge les fonctions de première classe s'il traite les fonctions comme des objets de première classe.

Propriétés des fonctions de première classe :

- Une fonction est une instance du type Object.
- On peut stocker la fonction dans une variable.
- On peut passer la fonction en tant que paramètre à une autre fonction.
- On peut renvoyer la fonction à partir d'une fonction.
- On peut les stocker dans des structures de données telles que des tables de hachage, des listes, ...

First order function

```
 # Python program to demonstrate
# higher order functions


def shout(text):
    return text.upper()

def whisper(text):
    return text.lower()

def greet(func):
    # storing the function in a variable
    greeting = func("Hi, I am created by a function passed as an argument.")
    print(greeting)

greet(shout)
greet(whisper)
```

3.4.1 Fonctions d'ordre supérieur intégrées

Pour faciliter le traitement des objets itérables tels que les listes et les itérateurs, Python a implémenté certaines fonctions d'ordre supérieur couramment utilisées. Ces fonctions renvoient un itérateur peu encombrant. Certaines des fonctions intégrées d'ordre supérieur sont :

Map()

La fonction map(fonction , itérables) renvoie une liste des résultats après avoir appliqué la fonction donnée à chaque élément d'un itérable donné (liste, tuple, etc.)

— fonction - Il s'agit d'une fonction dans laquelle une map transmet chaque élément de l'itérable.
— itérables - Il s'agit d'une séquence, d'une collection ou d'un objet itérateur qui doit être mappé.

map

```
# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
results = map(addition, numbers)

# Does not Prints the value
print(results)

# For Printing value
for result in results:
    print(result, end = " ")
```

filter()

La méthode filter(fonction , itérables) filtre la séquence donnée à l'aide d'une fonction qui teste chaque élément de la séquence pour être vrai ou non.

— fonction - Il s'agit d'une fonction dans laquelle une map transmet chaque élément de l'itérable.
— itérables - Il s'agit d'une séquence, d'une collection ou d'un objet itérateur qui doit être mappé.

filter

```
 # function that filters vowels
def fun(variable):

    letters = ['a', 'e', 'i', 'o', 'u']

    if (variable in letters):
        return True
    else:
        return False


# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
```

```
# using filter function
filtered = filter(fun, sequence)

print('The filtered letters are:')

for s in filtered:
    print(s)
```

reduce()

reduce() est une fonction intégrée qui applique une fonction donnée aux éléments d'un itérable, les réduisant à une valeur unique.

- L'argument de la fonction est une fonction qui prend deux arguments et renvoie une seule valeur. Le premier argument est la valeur accumulée, et le second argument est la valeur actuelle de la table itérative.
- L'argument itérable est la séquence de valeurs à réduire.
- L'argument facultatif initialisateur est utilisé pour fournir une valeur initiale au résultat accumulé. Si aucun initialisateur n'est spécifié, le premier élément de l'itérable est utilisé comme valeur initiale.

reduce

```
 from functools import reduce

# Function that returns the sum of two numbers
def add(a, b):
    return a + b

# Our Iterable
num_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# add function is passed as the first argument, and num_list is passed as the
   second argument
sum = reduce(add, num_list)
print(f"Sum of the integers of num_list : {sum}")
```

3.4.2 Les fonctions Lambda

Fonctions lambda : en Python, fonction anonyme signifie qu'une fonction est sans nom. Comme nous le savons déjà, le mot-clé def est utilisé pour définir les fonctions normales et le mot-clé lambda est utilisé pour créer des fonctions anonymes.
La syntaxe de la fonction Lambda est donnée : arguments lambda : expression.

lambda

```
cube = lambda x: x*x*x
print(cube(7))

 fct = lambda x, y : (x * y)
print(fct(4, 5))
```

lambda et filter

```
 # This code used to filter the odd numbers from the given list
list_ = [35, 12, 69, 55, 75, 14, 73]
odd_list = list(filter( lambda num: (num % 2 != 0) , list_ ))
print('The list of odd number is:',odd_list)
```

lambda et map

```
 #Code to calculate the square of each number of a list using the map() function
numbers_list = [2, 4, 5, 1, 3, 7, 8, 9, 10]
squared_list = list(map( lambda num: num ** 2 , numbers_list ))
print( 'Square of each number in the given list:' ,squared_list )
```

lambda et list comprehension

```
#Code to calculate square of each number of lists using list comprehension
squares = [lambda num = num: num ** 2 for num in range(0, 11)]
for square in squares:
       print('The square value of all numbers from 0 to 10:',square(), end = " ")
```

lambda et if else

```
# Code to use lambda function with if-else
Minimum = lambda x, y : x if (x < y) else y
print('The greater number is:', Minimum( 35, 74 ))
```

lambda et Multiple Statements

```
# Code to print the third largest number of the given list using the lambda
    function
my_List = [ [3, 5, 8, 6], [23, 54, 12, 87], [1, 2, 4, 12, 5] ]
# sorting every sublist of the above list
sort_List = lambda num : ( sorted(n) for n in num )
# Getting the third largest number of the sublist
third_Largest = lambda num, func : [ l[ len(l) - 2] for l in func(num)]
result = third_Largest( my_List, sort_List)
print('The third largest number from every sub list is:', result )
```

3.5 Immutabilité

L'immuabilité dans un paradigme de programmation fonctionnelle peut être utilisée pour le débogage car elle générera une erreur lorsque la variable est modifiée et non lorsque la valeur est modifiée. Python prend également en charge certains types de données immuables tels que string, tuple, numérique, etc.

Immutabilité

```
# String data types
immutable = "TestTest"

# changing the values will
# raise an error
immutable[1] = 'K'
```

3.6 Function Decorator

Voici quelques informations importantes sur les fonctions en Python qui sont utiles pour comprendre les fonctions décoratrices.

— En Python, nous pouvons définir une fonction à l'intérieur d'une autre fonction.
— En Python, une fonction peut être passée en paramètre à une autre fonction (une fonction peut également renvoyer une autre fonction).

netsed function

```
def messageWithWelcome(str):

    # Nested function
    def addWelcome():
        return "Welcome to "

    # Return concatenation of addWelcome()
    # and str.
    return  addWelcome() + str

# To get site name to which welcome is added
def site(site_name):
    return site_name

print messageWithWelcome(site("GeeksforGeeks"))
```

Les décorateurs sont un outil très puissant et utile en Python, car ils permettent aux programmeurs de modifier le comportement d'une fonction ou d'une classe. Les décorateurs nous permettent d'envelopper une autre fonction afin d'étendre le comportement de la fonction enveloppée, sans la modifier de manière permanente. Mais avant de plonger dans les décorateurs,

comprenons quelques concepts qui nous seront utiles dans l'apprentissage des décorateurs.

le code ci-dessus peut être réécrit comme suit. Nous utilisons @func_name pour spécifier un décorateur à appliquer à une autre fonction.

decorator function

```
# Adds a welcome message to the string
# returned by fun(). Takes fun() as
# parameter and returns welcome().
def decorate_message(fun):

    # Nested function
    def addWelcome(site_name):
        return "Welcome to " + fun(site_name)

    # Decorator returns a function
    return addWelcome

@decorate_message
def site(site_name):
    return site_name;

# Driver code

# This call is equivalent to call to
# decorate_message() with function
# site("GeeksforGeeks") as parameter
print site("GeeksforGeeks")
```

decorator function

```
# A Python example to demonstrate that
# decorators can be useful attach data

# A decorator function to attach
# data to func
def attach_data(func):
        func.data = 3
        return func

@attach_data
def add (x, y):
        return x + y

# Driver code

# This call is equivalent to attach_data()
```

```
# with add() as parameter
print(add(2, 3))

print(add.data)
```

Chapitre 4

Le Paradigm de la Programmation orientée objet

4.1 Introduction

La programmation orientée objet (POO) est le terme utilisé pour décrire une approche de programmation basée sur des objets et des classes. Le paradigme orienté objet nous permet d'organiser le logiciel comme une collection d'objets constitués à la fois de données et de comportements. Cela contraste avec la pratique de programmation fonctionnelle conventionnelle qui ne relie que vaguement les données et le comportement.

Depuis les années 1980, le mot « objet » est apparu en relation avec les langages de programmation, presque tous les langages développés depuis 1990 ayant des fonctionnalités orientées objet. Certains langages ont même été équipés de fonctionnalités orientées objet. Il est largement admis que la programmation orientée objet est le moyen le plus important et le plus puissant de créer des logiciels.

L'approche de programmation orientée objet encourage :

Modularisation : où l'application peut être décomposée en modules.

Réutilisation du logiciel : où une application peut être composée de modules existants et nouveaux.

Un langage de programmation orienté objet prend généralement en charge cinq fonctionnalités principales :

- Classes
- Objets
- Encapsulation
- Héritage
- Polymorphisme

4.2 Une classe orientée objet

Si on pense à un objet du monde réel, tel qu'une télévision , il aura plusieurs caractéristiques et propriétés :

- On n'a pas besoin d'ouvrir le boîtier pour l'utiliser.
- On a quelques commandes pour l'utiliser (télécommande).
- On peut encore comprendre le concept d'un téléviseur, même s'il est connecté à un lecteur DVD.
- Il est complet lorsque nous l'achetons, avec toutes les exigences externes bien documentées.
- Le téléviseur ne plantera pas !

Une classe doit :

- Fournir une interface bien définie - telle que la télécommande du téléviseur.
- Représenter un concept clair - tel que le concept d'une télévision.
- Complet et bien documenté - le téléviseur doit avoir une prise et un manuel qui documente toutes les fonctionnalités.
- Le code doit être robuste - il ne doit pas planter, comme la télévision.

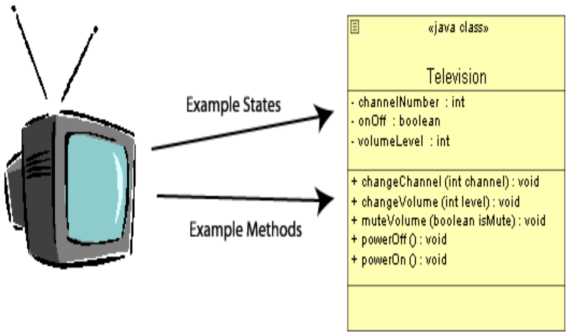
Avec un langage de programmation procédural (comme C), nous aurions les composants du téléviseur dispersés partout et nous serions responsables de les faire fonctionner correctement - il n'y aurait aucun cas entourant les composants électroniques.

Les humains utilisent tout le temps des descriptions basées sur les classes - qu'est-ce qu'un moteur ?

Les classes nous permettent de représenter des structures complexes dans un langage de programmation. Ils ont deux composants :

- Les états - (données/attributs) sont les valeurs de l'objet.
- Méthodes - (ou comportement) sont les façons dont l'objet peut interagir avec ses données, les actions.

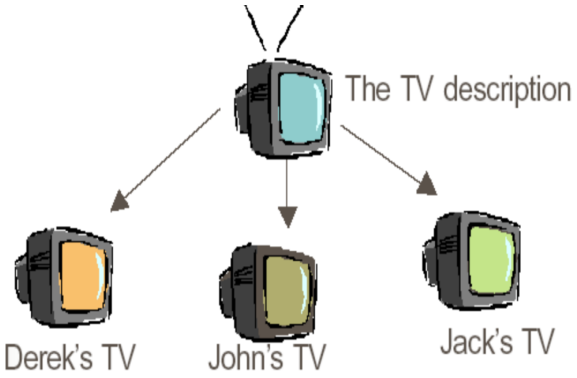
La notation utilisée dans la Figure suivante sur le côté droit est une représentation en langage de modélisation unifié (UML) de la classe Télévision pour la modélisation et la programmation orientées objet.



4.3 Un Objet

Un objet est une instance d'une classe. Vous pourriez considérer une classe comme la description d'un concept et un objet comme la réalisation de cette description pour créer une entité distincte indépendante. Par exemple, dans le cas de la télévision, la classe est l'ensemble de plans (ou de plans) pour une télévision générique, tandis qu'un objet de télévision est la réalisation de ces plans dans une télévision physique du monde réel. Il y aurait donc un ensemble de plans (la classe), mais il pourrait y avoir des milliers de téléviseurs du monde réel (objets).

Les objets peuvent être concrets (un objet du monde réel, un fichier sur un ordinateur) ou peuvent être conceptuels (comme une structure de base de données) chacun avec sa propre identité individuelle. La figure suivante montre un exemple où la description de la classe Télévision est réalisée en plusieurs objets télévision. Ces objets doivent avoir leur propre identité et sont indépendants les uns des autres. Par exemple, si le canal est modifié sur un téléviseur, il ne changera pas sur les autres téléviseurs.



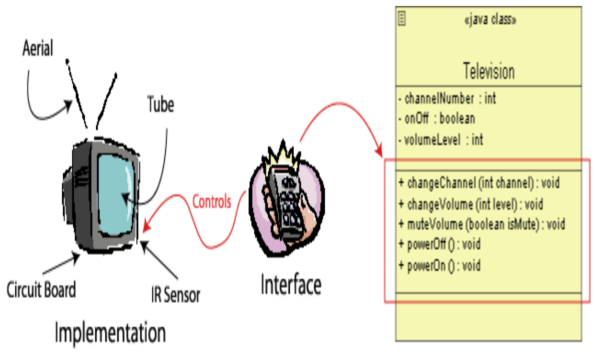
4.4 L'encapsulation

Le paradigme orienté objet encourage l'encapsulation. L'encapsulation est utilisée pour masquer la mécanique de l'objet, permettant de masquer la mise en œuvre réelle de l'objet, de sorte qu'on n'a pas besoin de comprendre comment l'objet fonctionne. Tout ce que nous devons comprendre, c'est l'interface qui nous est fournie.

On peut penser à cela dans le cas de la classe Télévision, où la fonctionnalité de la télévision nous est cachée, mais on nous fournit une télécommande ou un ensemble de commandes pour interagir avec la télévision, offrant un haut niveau d'abstraction. Ainsi, comme dans la figure suivante, il n'est pas nécessaire de comprendre comment le signal est décodé de l'antenne et converti en une image à afficher sur l'écran avant de pouvoir utiliser le téléviseur.

Il existe un sous-ensemble de fonctionnalités que l'utilisateur est autorisé à appeler, appelé l'interface. Dans le cas du téléviseur, il s'agirait de la fonctionnalité que nous pourrions utiliser via la télécommande ou les boutons situés à l'avant du téléviseur.

L'implémentation complète d'une classe est la somme de l'interface publique et de l'implémentation privée.



L'encapsulation est le terme utilisé pour décrire la manière dont l'interface est séparée de l'implémentation. Vous pouvez considérer l'encapsulation comme un "masquage de données", permettant à certaines parties d'un objet d'être visibles, tandis que d'autres parties restent cachées. Cela présente des avantages à la fois pour l'utilisateur et le programmeur.

Pour l'utilisateur (qui pourrait être un autre programmeur) :

— L'utilisateur n'a qu'à comprendre l'interface.

— L'utilisateur n'a pas besoin de comprendre comment l'implémentation fonctionne ou a été créée.

Pour le programmeur :

— Le programmeur peut modifier l'implémentation, mais n'a pas besoin d'en informer l'utilisateur.

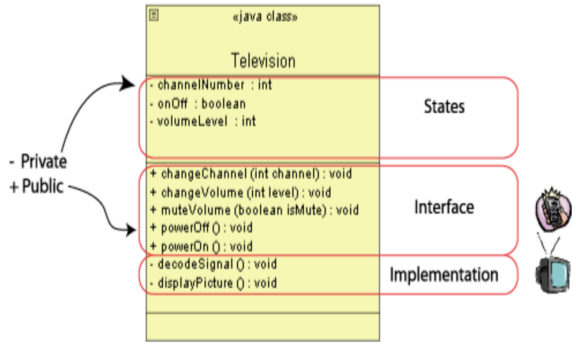
Ainsi, à condition que le programmeur ne modifie en aucune manière l'interface, l'utilisateur ne sera pas au courant de tout changement, à l'exception peut-être d'un changement mineur dans la fonctionnalité réelle de l'application.

On peut identifier un niveau de « masquage » de méthodes ou d'états particuliers au sein d'une classe en utilisant les mots-clés public, private et protected :

— méthodes publiques - décrivent l'interface.

— méthodes privées - décrivent la mise en œuvre.

La Figure suivante montre l'encapsulation en ce qui concerne la classe Télévision. Selon la notation UML, les méthodes privées sont désignées par un signe moins et les méthodes publiques sont désignées par un signe plus. Les méthodes privées seraient des méthodes écrites qui font partie du fonctionnement interne de la télévision, mais n'ont pas besoin d'être comprises par l'utilisateur. Par exemple, l'utilisateur aurait besoin d'appeler la méthode powerOn() mais la méthode privée displayPicture() serait également appelée, mais en interne si nécessaire, pas directement par l'utilisateur. Cette méthode n'est donc pas ajoutée à l'interface, mais masquée en interne dans l'implémentation en utilisant le mot clé private.



4.5 Encapsulation dans Python

4.5.1 Membres privés

Les membres privés et sécurisés de Python sont accessibles depuis l'extérieur de la classe grâce à la manipulation des noms de Python.

Private fields

```
class Base1:
   def __init__(self):
         self.p = "Javatpoint"
         self.__q = "Javatpoint"

# Creating a derived class
class Derived1(Base1):
```

```
      def __init__(self):

# Calling constructor of
# Base class
     Base1.__init__(self)
     print("We will call the private member of base class: ")
     print(self.__q)


# Driver code
obj_1 = Base1()
print(obj_1.p)
```

4.5.2 Membres protégés

La variable protected est accessible depuis la classe et dans les classes dérivées (elle peut également être modifiée dans les classes dérivées), mais il est d'usage de ne pas y accéder en dehors du corps de la classe.

Protected fields

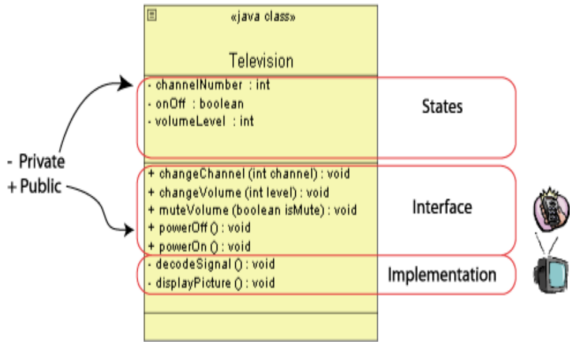
```
 class Base1:
    def __init__(self):

        # the protected member
        self._p = 78

# here, we will create the derived class
class Derived1(Base):
    def __init__(self):

# now, we will call the constructor of Base class
        Base1.__init__(self)
        print ("We will call the protected member of base class: ",
            self._p)

# Now, we will be modifing the protected variable:
        self._p = 433
        print ("we will call the modified protected member outside the class: ",
            self._p)


obj_1 = Derived1()

obj_2 = Base1()
```

```
# here, we will call the protected member
# this can be accessed but it should not be done because of convention
print ("Access the protected member of obj_1: ", obj_1._p)

# here, we will access the protected variable outside
print ("Access the protected member of obj_2: ", obj_2._p)
```

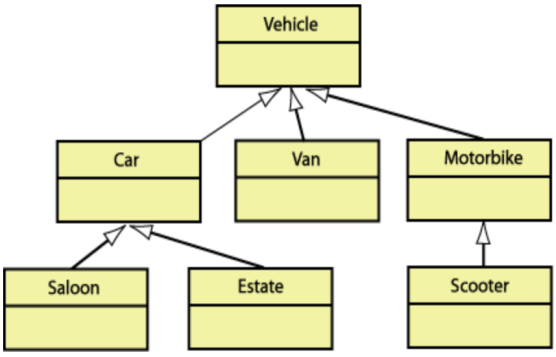
4.6 L'héritage

Si on a plusieurs descriptions avec des points communs entre ces descriptions, nous pouvons regrouper les descriptions et leurs points communs en utilisant l'héritage pour fournir une représentation compacte de ces descriptions. L'approche de programmation orientée objet nous permet de regrouper les points communs et de créer des classes qui peuvent décrire leurs différences par rapport aux autres classes.

Les humains utilisent ce concept pour catégoriser les objets et les descriptions. Par exemple, vous avez peut-être répondu à la question « Qu'est-ce qu'un canard ? » par « un oiseau qui nage », ou encore plus précisément « un oiseau qui nage, avec des pattes palmées et un bec au lieu d'un bec ». Nous pourrions donc dire qu'un canard est un oiseau qui nage, nous pourrions donc le décrire comme dans la figure suivante . Cette figure illustre la relation d'héritage entre un canard et un oiseau. En effet on peut dire qu'un Canard est un type particulier d'Oiseau.



Un autre exemple plus complexe :



4.7 Le polymorphisme

Lorsqu'une classe hérite d'une autre classe, elle hérite à la fois des états et des méthodes de cette classe, donc dans le cas de la classe Car héritant de la classe Vehicle, la classe Car hérite des méthodes de la classe Vehicle, telles que engineStart(), gearChange( ), lightsOn() etc. La classe Car héritera également des états de la classe Vehicle, tels que isEngineOn, isLightsOn, numberWheels etc.

Le polymorphisme signifie "formes multiples". En POO, ces formes multiples font référence à plusieurs formes de la même méthode, où exactement le même nom de méthode peut être utilisé dans différentes classes, ou le même nom de méthode peut être utilisé dans la même classe avec des paramètres légèrement différents. Il existe deux formes de polymorphisme, le surdefinition et la surcharge.

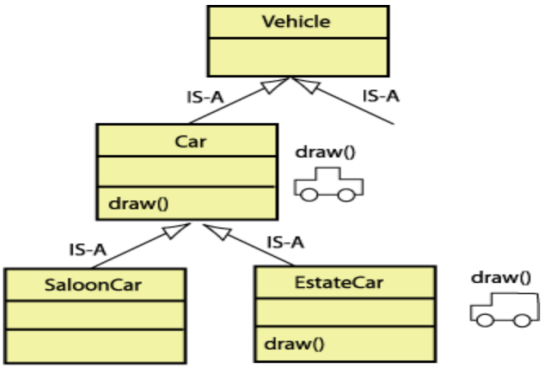
On pourrait définir le polymorphisme comme la propriété permettant à un programme de réagir de manière différenciée à l'envoi d'un même message (invocation de méthode) en fonction des objets qui reçoivent ce message.

- Il s'agit donc d'une aptitude d'adaptation dynamique du comportement selon les objets en présence.
- Avec l'encapsulation et l'héritage, le polymorphisme est une des propriétés essentielles de la programmation orientée objet.

4.7.1 Overriding : Surdefinition

Comme indiqué, une classe dérivée hérite de ses méthodes de la classe de base. Il peut être nécessaire de redéfinir une méthode héritée pour fournir un comportement spécifique à une classe dérivée - et ainsi modifier l'implémentation. Ainsi, la substitution est le terme utilisé pour décrire la situation où le même nom de méthode est appelé sur deux objets différents et chaque objet répond différemment.

Le Overriding permet à différents types d'objets partageant un comportement commun d'être utilisés dans du code qui ne requiert que ce comportement commun.



Considérons l'exemple précédent du diagramme de classe de véhicule. Dans ce cas, Car hérite de Véhicule et à partir de cette classe Car, il existe d'autres classes dérivées SaloonCar et EstateCar. Si une méthode draw() est ajoutée à la classe Car, cela est nécessaire pour dessiner une image d'un véhicule générique. Cette méthode ne dessinera pas de manière adéquate un break ou d'autres classes d'enfants. Over-Riding nous permet d'écrire une méthode draw() spécialisée pour la classe EstateCar - Il n'est pas nécessaire d'écrire une nouvelle méthode draw() pour la classe SaloonCar car la classe Car fournit une méthode draw() suffisamment appropriée. Tout ce que nous avons à

faire est d'écrire une nouvelle méthode draw() dans la classe EstateCar avec exactement le même nom de méthode. Ainsi, l'Overriding permet :

— Une API plus simple où nous pouvons appeler des méthodes du même nom, même si ces méthodes ont des fonctionnalités légèrement différentes.
— Un meilleur niveau d'abstraction, dans la mesure où les mécanismes d'implémentation restent cachés.

4.7.2 Overloading : SurCharge

La surcharge est la deuxième forme de polymorphisme. Le même nom de méthode peut être utilisé, mais le nombre de paramètres ou les types de paramètres peuvent différer, ce qui permet au compilateur de choisir la bonne méthode. Par example :

add (int x, int y)
add (String x, String y)

sont deux méthodes différentes qui ont le même nom et le même nombre de paramètres. Cependant, lorsque nous passons deux objets String au lieu de deux variables int, nous attendons des fonctionnalités différentes. Lorsque nous ajoutons deux valeurs int, nous attendons un résultat - par exemple 6 + 7 = 13. Cependant, si nous passons deux objets String, nous attendons un résultat de "6" + "7" = "67". En d'autres termes, les chaînes doivent être concaténées.

Le nombre d'arguments peut également déterminer quelle méthode doit être exécutée.

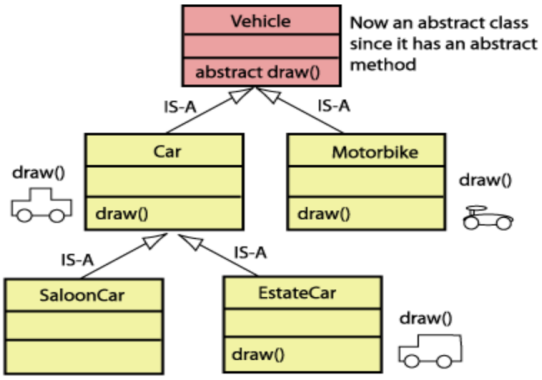
channel()
channel(int x)

fournira des fonctionnalités différentes où la première méthode peut simplement afficher le numéro de canal actuel, mais la deuxième méthode définira le numéro de canal sur le numéro passé.

4.8 Classe abstraite

Une classe abstraite est une classe incomplète, en ce sens qu'elle décrit un ensemble d'opérations, mais auquel il manque l'implémentation réelle de ces opérations. Cours abstraits :

— Ne peut pas être instancié.
— onc, ne peut être utilisé que par héritage.



Chapitre 5

La programmation POO : Python

5.1 Introduction

Comme d'autres langages de programmation à usage général, Python est également un langage orienté objet depuis ses débuts. Il nous permet de développer des applications en utilisant une approche orientée objet. En Python, nous pouvons facilement créer et utiliser des classes et des objets.

Un paradigme orienté objet consiste à concevoir le programme en utilisant des classes et des objets. L'objet est lié à des entités de mots réels telles que livre, maison, crayon, etc. Le concept oops se concentre sur l'écriture du code réutilisable. C'est une technique très répandue pour résoudre le problème en créant des objets.

Les principes majeurs du système de programmation orienté objet sont donnés ci-dessous.

- Classe
- Objet
- Héritage
- Polymorphisme
- Encapsulation

5.2 Python Classe et Objets

5.2.1 Class

La classe peut être définie comme une collection d'objets. C'est une entité logique qui a des attributs et des méthodes spécifiques. Par exemple : si vous avez une classe d'employés, elle doit contenir un attribut et une méthode, c'est-à-dire un identifiant de messagerie, un nom, un âge, un salaire, etc.

Class

```
class car:
    def __init__(self,modelname, year):
        self.modelname = modelname
        self.year = year
    def display(self):
        print(self.modelname,self.year)
```

5.2.2 Objets

L'objet est une entité qui a un état et un comportement. Il peut s'agir de n'importe quel objet du monde réel comme la souris, le clavier, la chaise, la table, le stylo, etc.

Tout en Python est un objet, et presque tout a des attributs et des méthodes. Toutes les fonctions ont un attribut intégré , qui renvoie la docstring définie dans le code source de la fonction.

Class

```
class car:
    def __init__(self,modelname, year):
        self.modelname = modelname
        self.year = year
    def display(self):
        print(self.modelname,self.year)

c1 = car("Toyota", 2016)
c1.display()
```

5.2.3 Constructeur Python

Un constructeur est un type spécial de méthode (fonction) qui est utilisé pour initialiser les membres d'instance de la classe.

En C++ ou Java, le constructeur a le même nom que sa classe, mais il traite le constructeur différemment en Python. Il est utilisé pour créer un objet.

Les constructeurs peuvent être de deux types.
- Constructeur paramétré
- Constructeur non paramétré

La définition du constructeur est exécutée lorsque nous créons l'objet de cette classe. Les constructeurs vérifient également qu'il y a suffisamment de ressources pour que l'objet effectue une tâche de démarrage.

Constructeur parametre

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))


emp1 = Employee("John", 101)
emp2 = Employee("David", 102)

# accessing display() method to print employee 1 information

emp1.display()

# accessing display() method to print employee 2 information
```

```
emp2.display()
```

Constructeur non parametre

```
class Employee:
class Student:
    # Constructor - non parameterized
    def __init__(self):
        print("This is non parametrized constructor")
    def show(self,name):
        print("Hello",name)
student = Student()
student.show("John")
```

5.3 Magic Methods

Les méthodes Dunder ou Magic en Python sont les méthodes ayant deux préfixes et suffixes de soulignement dans le nom de la méthode. Dunder signifie ici "Double Under (Underscores)". Ces méthodes sont couramment utilisées pour la surcharge des opérateurs.

List of Python Magic Methods : dir(int)

Destructeur

```
# Python program to demonstrate
# __del__


class Example:

    # Initializing
    def __init__(self):
        print("Example Instance.")

    # Calling destructor
    def __del__(self):
        print("Destructor called, Example deleted.")

obj = Example()
del obj
```

toString

```
class GFG:

        def __init__(self, f_name, m_name, l_name):
                self.f_name = f_name
```

```
                self.m_name = m_name
                self.l_name = l_name

        def __repr__(self):
                return f'GFG("{self.f_name}","{self.m_name}","{self.l_name}")'

gfg = GFG("Geeks", "For", "Geeks")
print(repr(gfg))
```

5.4 Python héritage

L'héritage est un aspect important du paradigme orienté objet. L'héritage permet la réutilisation du code au programme car nous pouvons utiliser une classe existante pour créer une nouvelle classe au lieu de la créer à partir de zéro.

5.4.1 Héritage simple

héritage simple

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

5.4.2 Héritage multiple

héritage simple

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
```

```
print(d.Divide(10,20))
```

5.4.3 La méthode issubclass(sub,sup)

La méthode issubclass(sub, sup) est utilisée pour vérifier les relations entre les classes spécifiées. Elle renvoie true si la première classe est la sous-classe de la deuxième classe et false dans le cas contraire.

issubclass

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

5.4.4 La méthode isinstance (obj, class)

La méthode isinstance() est utilisée pour vérifier la relation entre les objets et les classes. Il renvoie vrai si le premier paramètre, c'est-à-dire obj est l'instance du deuxième paramètre, c'est-à-dire la classe.

isinstance

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(isinstance(d,Derived))
```

5.4.5 Redéfinition de la méthode

Nous pouvons fournir une implémentation spécifique de la méthode de la classe parente dans notre classe enfant. Lorsque la méthode de la classe parente est définie dans la classe enfant avec une implémentation spécifique, le concept est appelé substitution de méthode. Nous pouvons

avoir besoin d'effectuer une substitution de méthode dans le scénario où la définition différente d'une méthode de classe parent est nécessaire dans la classe enfant.

Redéfinition

```
class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank()
b2 = SBI()
b3 = ICICI()
print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("ICICI Rate of interest:",b3.getroi());
```

5.4.6 La surcharge des opérateurs

La surcharge des opérateurs en Python permet d'étendre leur signification au-delà de leur signification opérationnelle prédéfinie. Par exemple, nous utilisons l'opérateur "+" pour additionner deux entiers, joindre deux chaînes de caractères ou fusionner deux listes. Cela est possible car l'opérateur "+" est surchargé par les classes "int" et "str". L'utilisateur peut remarquer que le même opérateur ou la même fonction intégré(e) a un comportement différent pour des objets de classes différentes. Ce processus est connu sous le nom de surcharge d'opérateur.

\+ Overload

```
class complex_1:
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

    # Now, we will add the two objects
    def __add__(self, U):
        return self.X + U.X, self.Y + U.Y

Object_1 = complex_1(23, 12)
Object_2 = complex_1(21, 22)
Object_3 = Object_1 + Object_2
print (Object_3)
```

Les opérateurs

```
+       __add__(self, other)
-       __sub__(self, other)
*       __mul__(self, other)
/       __truediv__(self, other)
//      __floordiv__(self, other)
\%      __mod__(self, other)
**      __pow__(self, other)
>>      __rshift__(self, other)
<<      __lshift__(self, other)
&       __and__(self, other)
|       __or__(self, other)
^       __xor__(self, other)
<       __LT__(SELF, OTHER)
>       __GT__(SELF, OTHER)
<=      __LE__(SELF, OTHER)
>=      __GE__(SELF, OTHER)
==      __EQ__(SELF, OTHER)
!=      __NE__(SELF, OTHER)
-=      __ISUB__(SELF, OTHER)
+=      __IADD__(SELF, OTHER)
*=      __IMUL__(SELF, OTHER)
/=      __IDIV__(SELF, OTHER)
//=     __IFLOORDIV__(SELF, OTHER)
%=      __IMOD__(SELF, OTHER)
**=     __IPOW__(SELF, OTHER)
>>=     __IRSHIFT__(SELF, OTHER)
<<=     __ILSHIFT__(SELF, OTHER)
&=      __IAND__(SELF, OTHER)
|=      __IOR__(SELF, OTHER)
^=      __IXOR__(SELF, OTHER)
-       __NEG__(SELF, OTHER)
+       __POS__(SELF, OTHER)
~       __INVERT__(SELF, OTHER)
```

5.4.7 Abstraction

L'abstraction est utilisée pour masquer la fonctionnalité interne de la fonction aux utilisateurs. Les utilisateurs n'interagissent qu'avec l'implémentation de base de la fonction, mais le fonctionnement interne est caché. L'utilisateur est familier avec "ce que fait la fonction" mais il ne sait pas "comment il le fait".

En termes simples, nous utilisons tous le smartphone et connaissons très bien ses fonctions telles que l'appareil photo, l'enregistreur vocal, la numérotation des appels, etc., mais nous ne savons

pas comment ces opérations se déroulent en arrière-plan. Prenons un autre exemple - Lorsque nous utilisons la télécommande du téléviseur pour augmenter le volume. Nous ne savons pas comment appuyer sur une touche augmente le volume du téléviseur. On sait seulement appuyer sur le bouton "+" pour augmenter le volume.

Abstraction

```
# Python program demonstrate
# abstract base class work
from abc import ABC, abstractmethod
class Car(ABC):
    def mileage(self):
        pass

class Tesla(Car):
    def mileage(self):
        print("The mileage is 30kmph")
class Suzuki(Car):
    def mileage(self):
        print("The mileage is 25kmph ")
class Duster(Car):
     def mileage(self):
          print("The mileage is 24kmph ")

class Renault(Car):
    def mileage(self):
            print("The mileage is 27kmph ")

# Driver code
t= Tesla ()
t.mileage()

r = Renault()
r.mileage()

s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
```

Points à retenir

- Une classe abstraite peut contenir à la fois la méthode normale et la méthode abstraite.
- Une classe abstraite ne peut pas être instanciée.

L'abstraction est essentielle pour cacher la fonctionnalité de base aux utilisateurs. Nous avons couvert tous les concepts de base de l'abstraction en Python.

5.5 Try Except

Une exception peut être définie comme une condition inhabituelle dans un programme entraînant l'interruption du déroulement du programme.

Chaque fois qu'une exception se produit, le programme arrête l'exécution et le code suivant n'est donc pas exécuté. Par conséquent, une exception concerne les erreurs d'exécution qui ne peuvent pas être gérées par le script Python. Une exception est un objet Python qui représente une erreur.

Python fournit un moyen de gérer l'exception afin que le code puisse être exécuté sans aucune interruption. Si nous ne gérons pas l'exception, l'interpréteur n'exécute pas tout le code qui existe après l'exception.

Python a de nombreuses exceptions intégrées qui permettent à notre programme de s'exécuter sans interruption et de donner la sortie.

- ZeroDivisionError : se produit lorsqu'un nombre est divisé par zéro.
- NameError : Cela se produit lorsqu'un nom n'est pas trouvé. Il peut être local ou global.
- IndentationError : si une indentation incorrecte est donnée. erreur d'espace
- IOError : Cela se produit lorsque l'opération Input Output échoue.
- EOFError : Cela se produit lorsque la fin du fichier est atteinte, et pourtant des opérations sont en cours d'exécution.

5.5.1 Le problème sans gérer les exceptions

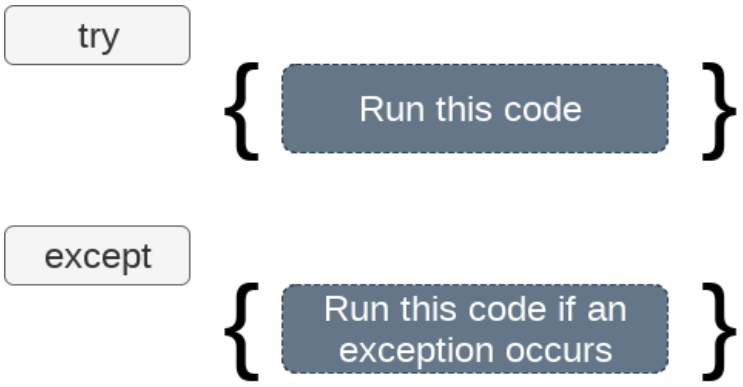
No Exception

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b
print("a/b = %d" %c)

#other code:
print("Hi I am other part of the program")
```

Le programme ci-dessus est syntaxiquement correct, mais il passe par l'erreur en raison d'une entrée inhabituelle. Ce type de programmation peut ne pas être adapté ou recommandé pour les projets car ces projets nécessitent une exécution ininterrompue. C'est pourquoi une gestion des exceptions joue un rôle essentiel dans la gestion de ces exceptions inattendues. Nous pouvons gérer ces exceptions de la manière suivante.

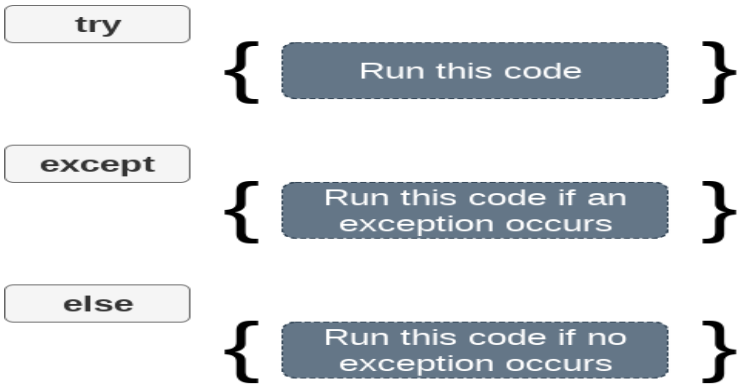
5.5.2 Gestion des exceptions en python

Si le programme Python contient du code suspect susceptible de lever l'exception, nous devons placer ce code dans le bloc try. Le bloc try doit être suivi de l'instruction except, qui contient un bloc de code qui sera exécuté s'il y a une exception dans le bloc try.



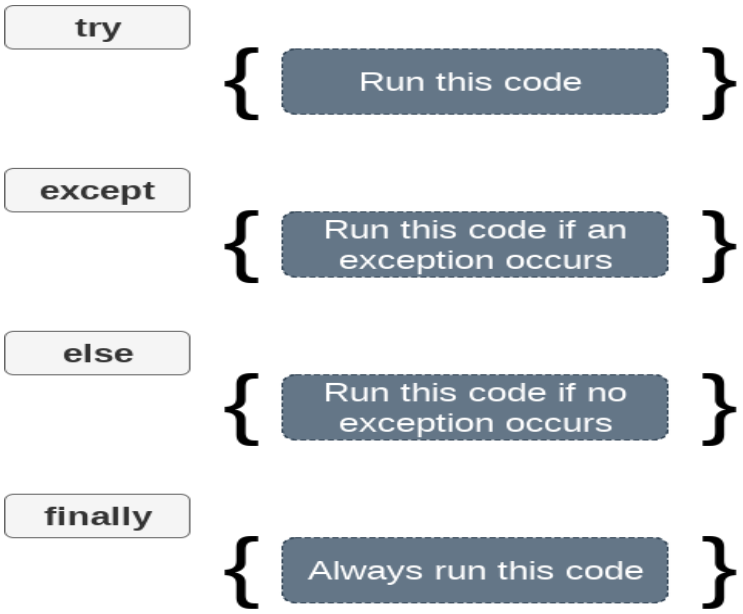
Exception

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
except:
    print("Can't divide with zero")
```



Exception

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
# Using Exception with except statement. If we print(Exception) it will return
    exception class
except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block")
```



Exception

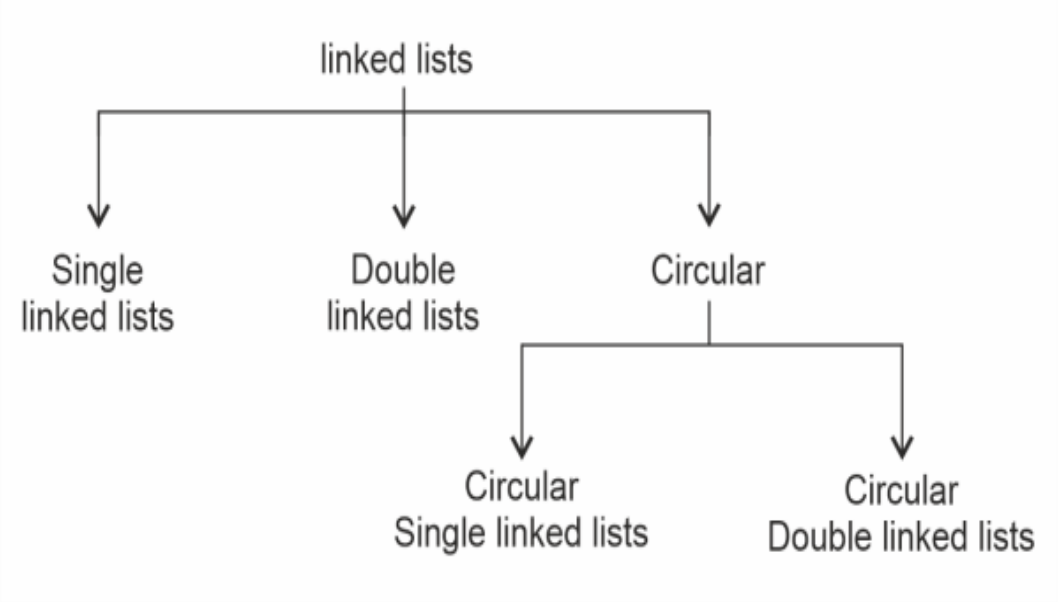
```
try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```

Chapitre 6

Structures de données Personnalisées

6.1 Listes chaînées Linked List

La liste chaînée, comme son nom l'indique, est liée. Chaque nœud de la liste liée se compose de deux segments : le champ de données contenant les données/valeurs et le champ suivant contenant la référence au nœud suivant, ce qui permet de les relier entre eux. Il s'agit d'une structure de données linéaire, mais les éléments ne sont pas stockés dans des emplacements de mémoire contigus.



Linked List

```
#Creating and displaying a linked list
class node:
    def __init__(self, data):
        self.data = data
        self.next = None


class LL:
    def __init__(self):
        self.head = None
    #represent a class's objects as a string.
    def __repr__(self):
        node = self.head
        nodes = []
        while node is not None:
            nodes.append(node.data)
```

```
            node = node.next
        nodes.append("None")
        return " -> ".join(map(str, nodes))
    #insert a node at the beginning
    def insertatbeg(self, newdata):
       newnode = node(newdata)
       newnode.next = self.head
       self.head = newnode

    #insert a node at the ending
    def insertatend(self, newdata):
        newnode = node(newdata)
        if(self.head is None):
            self.head = newnode
            return
        i = self.head
        while(i.next != None):
            i = i.next
        i.next = newnode
    #insert a node after the specified node
    def insertafteranode(self, givennodedata, newdata):
        i = self.head
        newnode = node(newdata)
        while(i.data != givennodedata):
            givennode = i.next
            i = i.next
        newnode.next = givennode.next
        givennode.next = newnode
    #insert a node before the specified node
    def insertbeforeanode(self, givennodedata, newdata):
        i = self.head
        newnode = node(newdata)
        while(i.next.data != givennodedata):
            givennode = i.next.next
            i = i.next
        i.next = newnode
        newnode.next = givennode

    #traversing the LinkedList
    def traversal(self):
        temp = self.head
        while(temp != None):
            print(temp.data)
            temp = temp.next

Llist = LL()
```

```
Llist.head = node(10)
node2 = node(20)
node3 = node(30)
Llist. head.next = node2
node2.next = node3
print("Displaying the linked list: ", Llist)

#Traversing call
print("Traversing from node to node:")
Llist.traversal()
Llist.insertatbeg(5)
print("After inserting 5 at the beginning:", Llist)
Llist.insertatend(40)
print("After inserting 40 at the end:", Llist)
Llist.insertafteranode(10, 15)
print("After inserting a node after 15:", Llist)
Llist.insertbeforeanode(30, 25)
print("After inserting a node before 30:", Llist)
```

6.2 Piles Stack

La pile est une structure de données linéaire. Elle est mise en œuvre selon le principe de l'abréviation "LIFO" : Dernier entré, premier sorti. Cela signifie que le dernier élément inséré dans une pile sera le premier à être supprimé. Une pile n'a qu'une seule ouverture, ce qui signifie que pour insérer ou supprimer des éléments, il faut utiliser la même extrémité. Lorsque nous insérons des éléments dans une pile, nous les plaçons les uns au-dessus des autres - de nouveaux éléments sur l'élément existant. Après avoir inséré tous les éléments, si nous voulons supprimer des éléments de la pile, le dernier élément inséré sera le premier à sortir.

1. Insérer un élément dans la pile : push
2. Suppression d'un élément de la pile : pop
3. La fin/l'ouverture de la pile : top of the stack

Stack Using List

```
#Implementation of stacks using lists
stack = []
#pushing elements
stack.append(1)
stack.append(2)
stack.append(3)
#Printing the stack
print("The elements of the stack")
for i in stack:
    print(i)
#popping elements
print("The first element to come out:", stack.pop())
```

```
print("The second element to come out:", stack.pop())
print("Final stack:", stack)
```

Stack Using Linked List

```
#Stack implementation using a Linked list
class node:
    def __init__(self, data):
        self.data = data
        self.next = None
class Stack:
    def __init__(self):
        self.head = node("head")
        self.size = 0
    def size(self):
        #size of the stack
        return self.size
    def top(self):
        #Top of the stack
        if(self.size==0):
            raise Exception("Empty stack")
        return self.head.next.data
    def push(self, data):
        Node = node(data)
        Node.next = self.head.next
        self. head.next = Node
        self.size += 1
    def pop(self):
        if(self.size==True):
            raise Exception("Empty stack")
        temp = self.head.next
        self. head.next = self.head.next.next
        self.size -= 1
        return temp.data
    def __repr__(self):
        #Representation
        Node = self.head
        nodes = []
        while node is not None:
            nodes.append(Node.data)
            Node = Node.next
        nodes.append("None")
        return " ".join(map(str, nodes))
mystack = Stack()
print("Stack without pushing any elements:", mystack)
for i in range(10, 15):
    mystack.push(i)
```

```
print("Stack after pushing elements:\n", mystack)
print("After pop 3 times:")
for i in range(0, 3):
    print(mystack.pop())
print("Stack:\n", mystack)
print("Element at the top of the stack:")
print(mystack.top())
print("Pop till stack becomes empty:")
for i in range(0, 3):
    print(mystack.pop())
```

6.3 Files Queue

Une file d'attente est une structure de données linéaire comme une pile, mais le principe de mise en œuvre de la file d'attente est FIFO (First in, first out). Cela signifie que le premier élément inséré dans la file d'attente sera le premier élément à en sortir.

— Une file d'attente comporte deux extrémités : l'extrémité avant et l'extrémité arrière.
— Les éléments sont insérés à partir de l'extrémité avant et supprimés à partir de l'extrémité arrière.

Queue Using List

```
#Implementation of queue using lists
print("Using lists:")
Queue1 = []
print("Queue: ", Queue1)
print("Inserting elements:")
for i in range(1, 6):
    Queue1.append(i)
print("Queue:", Queue1)
print("Deleting two elements:")
for i in range(0, 2):
    Queue1.pop(0)
print("Final queue:", Queue1)
```

Queue Using Linked List

```
# A single node of a singly linked list
class Node:
  # constructor
  def __init__(self, data = None, next = None):
    self.data = data
    self.next = next
# A Linked List class with a single head node
class LinkedList:
  def __init__(self):
    self.head = None
  # insert at the tail of the linked list
```

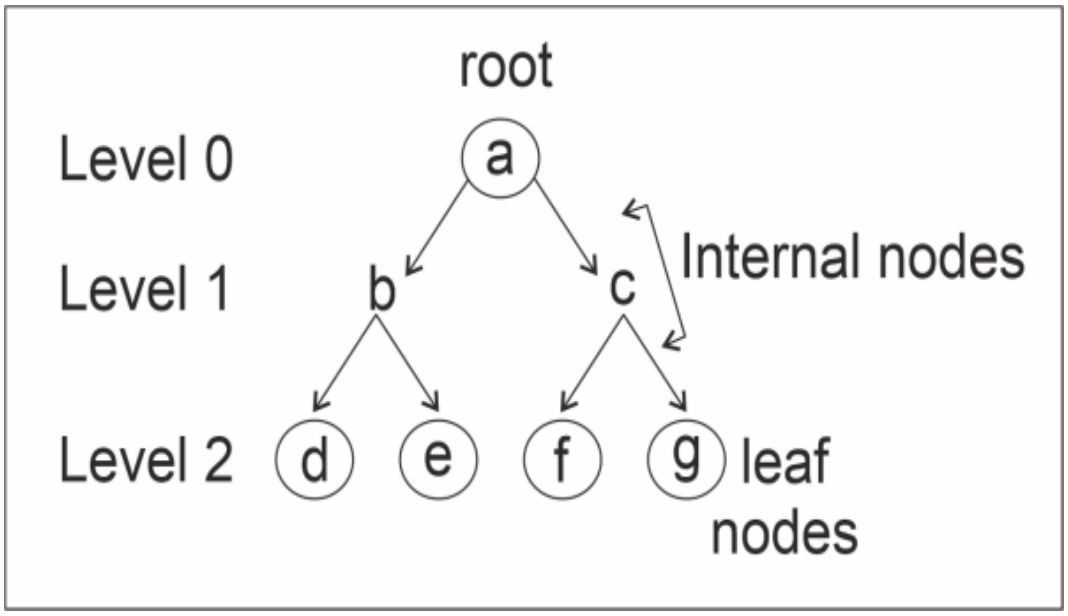
```
  def insert(self, data):
    newNode = Node(data)
    if(self.head):
      current = self.head
      while(current.next):
        current = current.next
      current.next = newNode
    else:
      self.head = newNode
  # remove an element from the head of the linked list
  def remove(self):
    temp = self.head
    if (temp is not None):
      element = temp.data
      self.head = temp.next
      temp = None
      return element
  # dispay an element from the tail of the linked list
  def peek(self):
    if(self.head):
      current = self.head
      while(current.next):
        current = current.next
      return current.data
  # print method for the linked list
  def printLL(self):
    current = self.head
    while(current):
      print(current.data)
      current = current.next

# A queue implemented using a linked list
Queue = LinkedList()
# Enqueue 3
Queue.insert(3)
# Display element at the tail
print("Element at the tail:", Queue.peek())
# Enqueue 4
Queue.insert(4)
# Display element at the tail
print("Element at the tail:", Queue.peek())
# Enqueue 5
Queue.insert(5)
# Display element at the tail
print("Element at the tail:", Queue.peek())
# Dequeue elements
```

```
print("Element dequeued:", Queue.remove())
print("Element dequeued:", Queue.remove())
print("Element dequeued:", Queue.remove())
```

6.4 Arbres Binaires / Binary Tree

Un arbre est une représentation hiérarchique de nœuds. Les arbres généalogiques sont des exemples d'arbres en temps réel. Chaque nœud ne peut avoir que deux enfants. Le nœud le plus élevé dans la hiérarchie ou le nœud le plus haut est appelé "nœud racine".

- Chaque nœud peut avoir un sous-arbre gauche et un sous-arbre droit.
- Par conséquent, un nœud dans un arbre binaire comporte trois segments : des données, une référence à l'enfant de gauche et une référence à l'enfant de droite.
- Les nœuds ayant la hiérarchie la plus basse sans aucun enfant sont appelés nœuds feuilles.



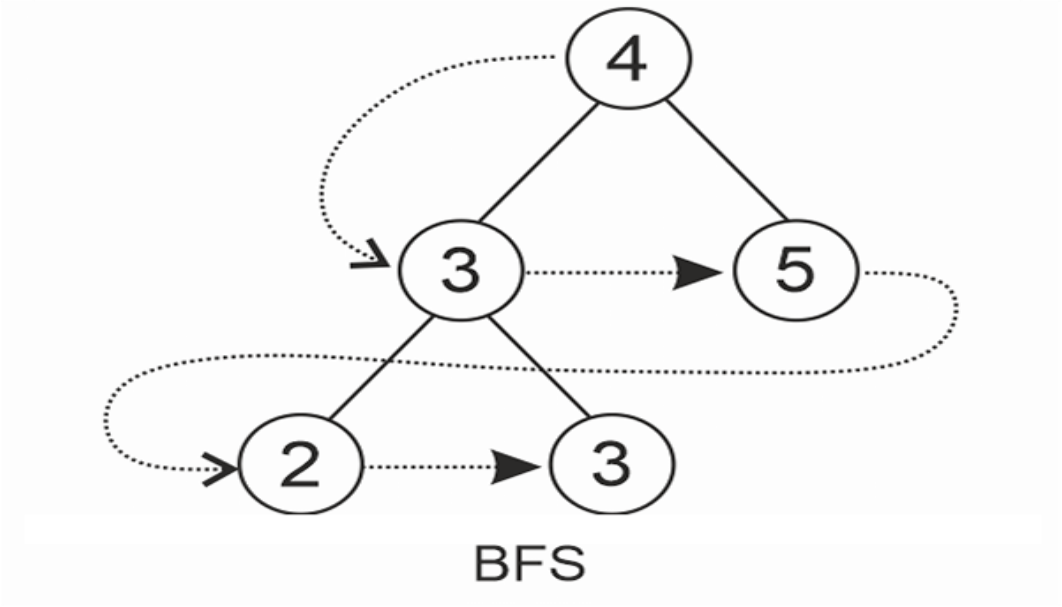
Un arbre peut être parcouru selon deux méthodes :

- DFS : par profondeur
- BFS : par largeur (ou niveau)

La méthode de parcours DFS comporte en outre trois types de parcours :

- Traversée pré-ordre : La racine est visitée en premier, puis le sous-arbre de gauche, suivi du sous-arbre de droite.
- Traversée post-ordre : Le sous-arbre de gauche est visité en premier, puis le sous-arbre de droite, suivi du nœud racine.
- Parcours dans l'ordre : Le sous-arbre de gauche est visité en premier, puis le nœud racine, suivi du sous-arbre de droite.

La traversée BFS consiste à visiter l'arbre par niveau.



BT Traversals

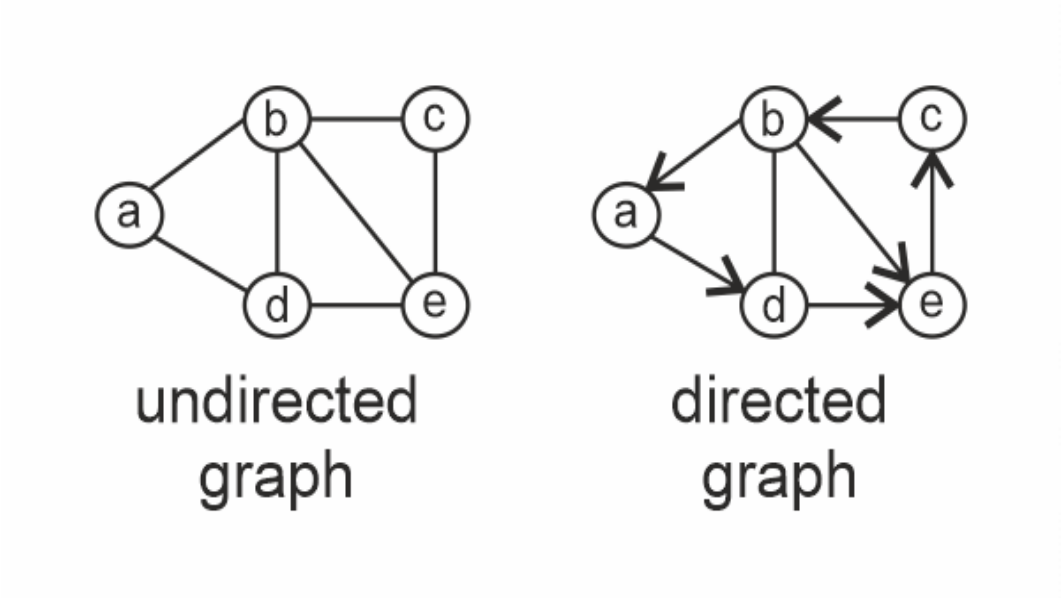
```
class TreeNode:
    def __init__(self, value):
        self.left = None
        self.right = None
        self.value = value
def Inorder(root):
      if(root):
          Inorder(root.left)
          print(root.value, end = " ")
          Inorder(root.right)
def Preorder(root):
    if(root):
        print(root.value, end = " ")
        Preorder(root.left)
        Preorder(root.right)
def Postorder(root):
    if(root):
        Postorder(root.left)
        Postorder(root.right)
        print(root.value, end = " ")
def BFS(root):
    if root is not None:
        Q = []
        Q.append(root)
        while(len(Q) > 0):
            print(Q[0].value, end = " ")
            temp = Q.pop(0)
            if temp.left is not None:
                Q.append(temp.left)
            if temp.right is not None:
```

```
                Q.append(temp.right)
root = TreeNode(4)
root.left = TreeNode(3)
root.right = TreeNode(5)
root.left.left = TreeNode(2)
root.left.right = TreeNode(3)
print("Preorder traversal:")
Preorder(root)
print("\nPostorder traversal:")
Postorder(root)
print("\nInorder traversal:")
Inorder(root)
print("\nBFS traversal:?)
BFS(root)
```

6.5 Graphs

en abrégé, G = (V, E). Ici, V représente les sommets/vertexes et E les arêtes/edges. Un graphe est une structure de données non linéaire. Il se compose de nœuds/vertices reliés par des arêtes. Les sommets et les arêtes doivent être un ensemble fini. Une arête peut être représentée par (u, v) étant donné que u et v sont les deux sommets que l'arête relie.

Un graphe peut être orienté ou non orienté. Dans un graphe non orienté, E = (u, v) et E = (v, u) sont identiques, tandis que dans un graphe orienté, ils ne sont pas identiques aux éléments orientés. Par conséquent, les arêtes sont représentées comme des paires ordonnées.



1. Les arêtes d'un graphique peuvent avoir des coûts ou des poids.
2. Les réseaux en temps réel sont représentés par des graphes.
3. Un graphe peut être implémenté à l'aide de :
   Matrice d'incidence

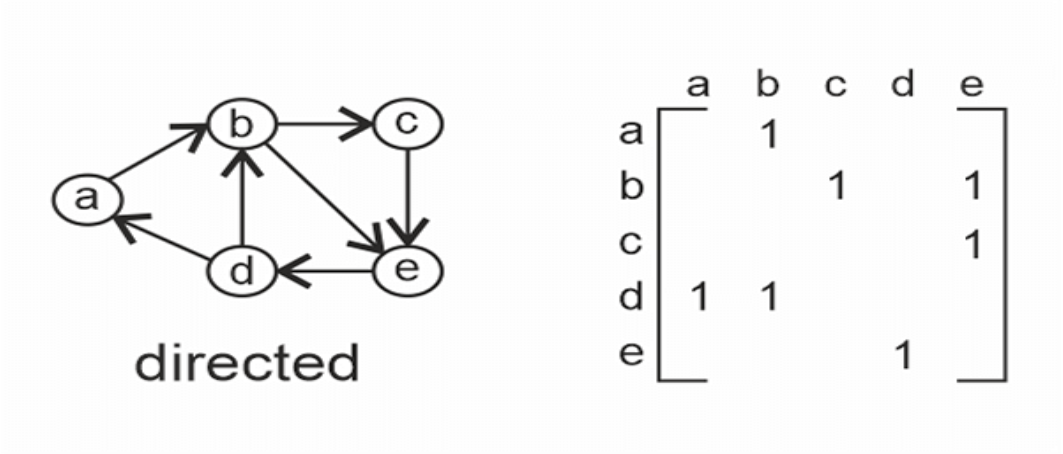
Liste d'incidence
Matrice d'adjacence
Liste d'adjacence

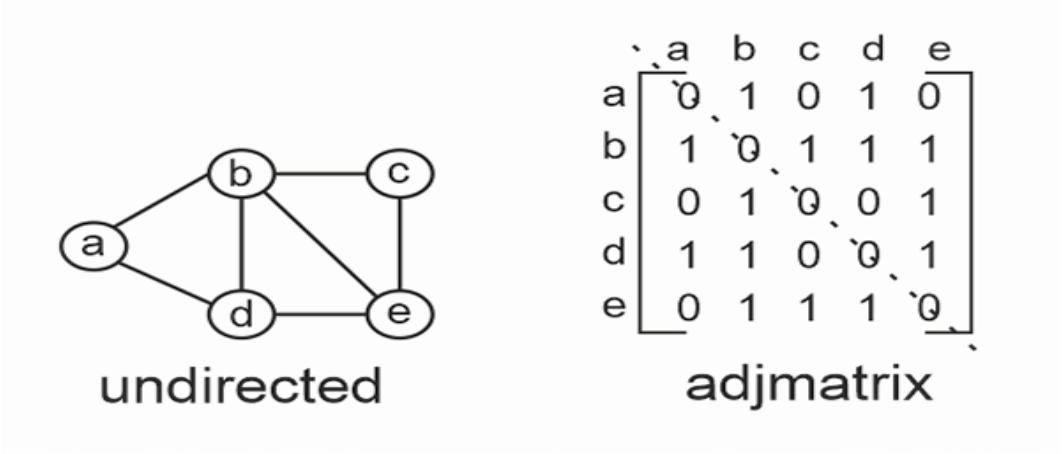
4. C'est le programmeur qui choisit la manière d'implémenter le graphe en fonction des besoins du scénario.
5. Un graphe peut être constitué de cycles.
6. Pour la traversée des graphes, les techniques BFS et DFS sont utilisées comme pour les arbres, mais pour éviter de visiter le même sommet encore et encore dans le cas des cycles, nous devons maintenir un tableau des sommets visités pour ne pas les visiter à nouveau.

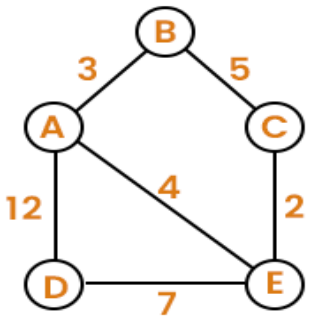
6.5.1 Matrice d'adjacence

Une matrice d'adjacence est un tableau 2D (V X V) où V représente les sommets du graphe. Dans la matrice adj[u][v], s'il existe une arête entre u et v, adj[u][v] = 1, sinon 0 est attribué.

— Dans un graphe non orienté, s'il existe une arête entre u et v, adj[u][v] = 1 et adj[v][u] = 1 car il n'y a pas de direction. Par conséquent, la matrice d'adjacence d'un graphe non orienté est toujours symétrique.
— Dans un graphe orienté, adj[u][v] n'est pas équivalent à adj[v][u].
— Si les arêtes ont des poids ou des coûts donnés, à la place de 1, on donne le poids/coût attribué dans la matrice.
— L'inconvénient de cette représentation est qu'elle prend plus de place - O(V2)







	A	B	C	D	E
A	0	3	0	12	4
B	3	0	5	0	0
C	0	5	0	0	2
D	12	0	0	0	7
E	4	0	2	7	0

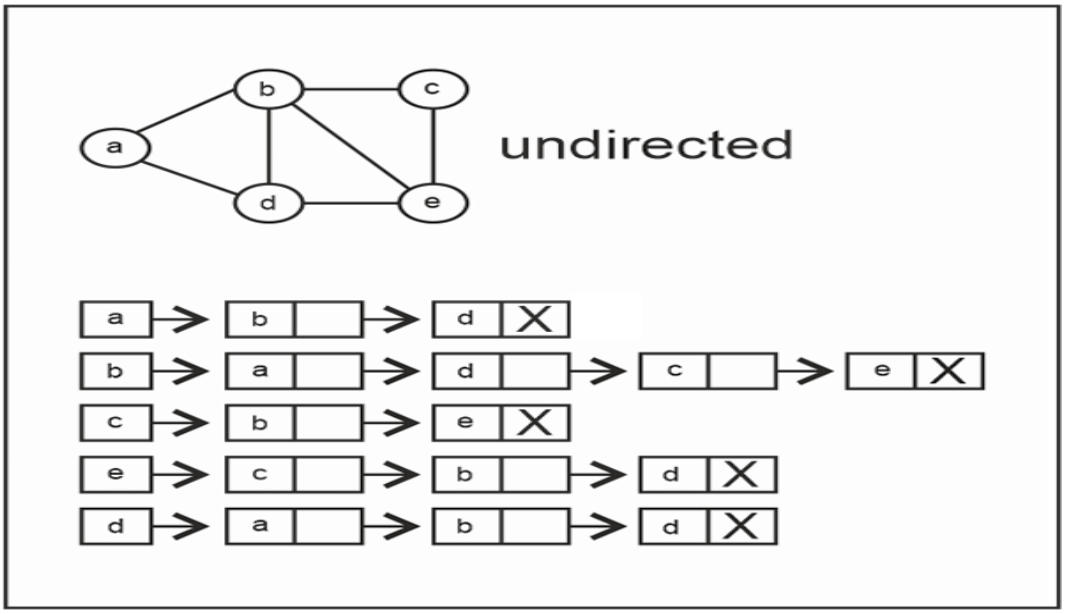
Graph Adj Matrix

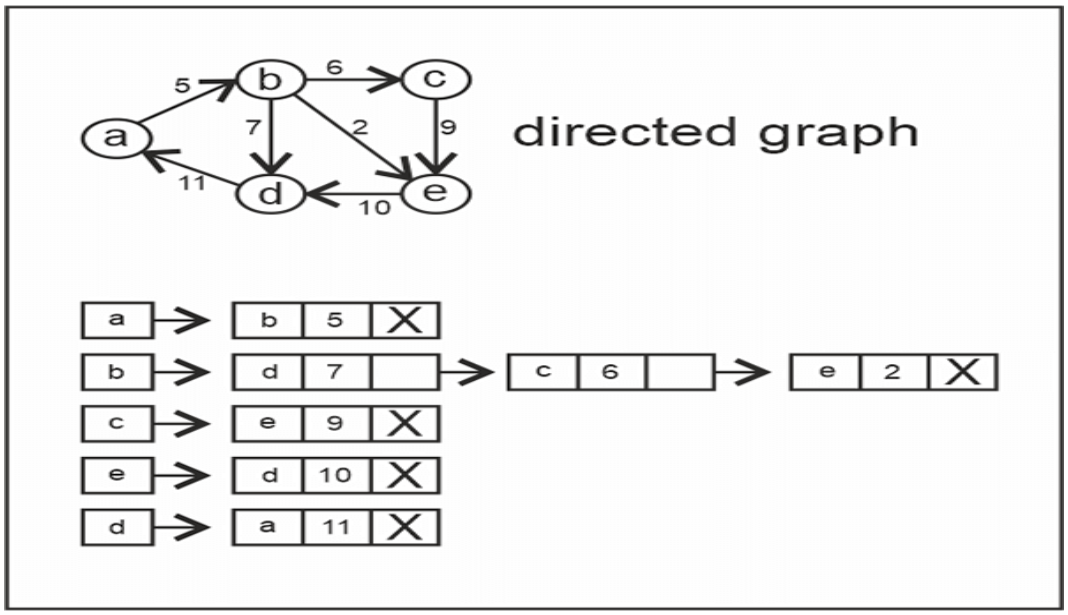
```
class matrix:
    def __init__(self, no_of_V):
        self.no_of_V = no_of_V
        self.mat = [[0]*no_of_V for i in range(0, no_of_V)]
        self.vertices = {}
    def set_v(self, no, name):
        self.vertices[name] = no
    def set_edge(self, to, by, cost):
        to = self.vertices[to]
        by = self.vertices[by]
        self.mat[to][by] = cost
        self.mat[by][to] = cost # Avoid the line if the graph is directed
    def get_mat(self):
        return self.mat
graph = matrix(5)
graph.set_v(0, 'A')
graph.set_v(1, 'B')
graph.set_v(2, 'C')
graph.set_v(3, 'D')
graph.set_v(4, 'E')
graph.set_edge('A', 'B', 3)
graph.set_edge('B', 'C', 5)
graph.set_edge('C', 'E', 2)
graph.set_edge('E', 'D', 7)
graph.set_edge('A', 'D', 12)
graph.set_edge('A', 'E', 4)
adj_mat = graph.get_mat()
for i in range(5):
    for j in range(5):
        print(adj_mat[i][j], end = " ")
    print()
```

6.5.2 Liste d'adjacen

Pour mettre en œuvre une liste d'adjacence, nous utilisons un tableau/une liste de listes liées pour représenter les sommets et les arêtes du graphe. Le nombre de listes chaînées utilisées dans la représentation est égal au nombre de sommets du graphe.

- Un tableau de longueur = nombre de sommets est créé, et pour chaque sommet, nous créons une liste chaînée avec tous les sommets adjacents, et ces listes chaînées sont disposées dans le tableau.
- Dans le cas d'un graphe orienté, tous les nœuds/vertices vers lesquels on peut se rendre à partir du nœud dans le tableau sont liés dans la liste chaînée.
- Pour simplifier, une liste d'adjacence est un tableau de listes liées contenant les nœuds adjacents au premier nœud.





Graph Adj List

```
class node:
    def __init__(self, vertex):
        self.vertex = vertex
        self.next = None
```

```
class adjlist:
    def __init__(self, no_of_V):
        self.no_of_V = no_of_V
        self.graph = [None]*self.no_of_V

    def edge(self, by, to):
        vertex = node(to)
        vertex.next = self.graph[by]
        self.graph[by] = vertex

        #Include the next three lines only if the graph is undirected
        vertex = node(by)
        vertex.next = self.graph[to]
        self.graph[to] = vertex

    def display(self):
        for i in range(self.no_of_V):
            print(str(i) + ":", end = "")
            temp = self.graph[i]
            while temp:
                print(" -> {}".format(temp.vertex), end= "")
                temp = temp.next
            print("\n")

graph = adjlist(5)
graph.edge(0, 1)
graph.edge(0, 2)
graph.edge(1, 3)
graph.edge(1, 2)
graph.edge(1, 4)
graph.edge(3, 4)
graph.edge(2, 4)
graph.display()
```

Chapitre 7

Python Multiprocessing

Le multiprocessus est la capacité du système à exécuter un ou plusieurs processus en parallèle. En d'autres termes, le multiprocessus utilise deux ou plusieurs unités centrales au sein d'un même système informatique. Cette méthode permet également de répartir les tâches entre plusieurs processus.

Les unités de traitement partagent la mémoire principale et les périphériques pour traiter les programmes simultanément. Multiprocessing L'application se divise en plusieurs parties et fonctionne de manière indépendante. Chaque processus est attribué au processeur par le système d'exploitation.

Python fournit un paquetage intégré appelé multiprocessus qui prend en charge l'échange de processus. Avant de travailler avec le multiprocessus, nous devons connaître l'objet processus.

7.1 Pourquoi le multiprocessus

Le multiprocessus est essentiel pour effectuer les multiples tâches au sein du système informatique. Supposons un ordinateur sans multiprocesseur ou à processeur unique. Nous assignons plusieurs processus à ce système en même temps.

Il devra alors interrompre la tâche précédente et passer à une autre pour que tous les processus se poursuivent. C'est aussi simple qu'un chef travaillant seul dans sa cuisine. Il doit effectuer plusieurs tâches pour préparer les aliments : couper, nettoyer, cuire, pétrir la pâte, cuire au four, etc.

C'est pourquoi le multitraitement est essentiel pour effectuer plusieurs tâches en même temps sans interruption. Il facilite également le suivi de toutes les tâches. C'est la raison pour laquelle le concept de multiprocessing est apparu.

Le multitraitement peut être représenté comme un ordinateur doté de plus d'un processeur central. Un processeur multicœur fait référence à un seul composant informatique avec deux unités indépendantes ou plus. Dans le multiprocessus, le processeur central peut assigner plusieurs tâches à la fois, chaque tâche ayant son propre processeur.

Multi Processor

```
from multiprocessing import Process
def disp():
    print ('Hello !! Welcome to Python Tutorial')

if __name__ == '__main__':
    p = Process(target=disp)
    p.start()
    p.join()
```

Dans le code ci-dessus, nous avons importé la classe Process et créé l'objet Process dans la fonction disp(). Nous avons ensuite démarré le processus à l'aide de la méthode start() et terminé le processus à l'aide de la méthode join(). Nous pouvons également passer les arguments dans la fonction déclarée en utilisant les mots-clés args.

Multi Processes

```
# Python multiprocessing example
# importing the multiprocessing module

import multiprocessing
def cube(n):
   # This function will print the cube of the given number
   print("The Cube is: {}".format(n * n * n))

def square(n):
    # This function will print the square of the given number
   print("The Square is: {}".format(n * n))

if __name__ == "__main__":
   # creating two processes
   process1 = multiprocessing.Process(target= square, args=(5, ))
   process2 = multiprocessing.Process(target= cube, args=(5, ))

   # Here we start the process 1
   process1.start()
   # Here we start process 2
   process2.start()

   # The join() method is used to wait for process 1 to complete
   process1.join()
   # It is used to wait for process 1 to complete
   process2.join()

   # Print if both processes are completed
   print("Both processes are finished")
```

Nbr de CPUs

Multi Processes (CPUs)

```
import multiprocessing
print("The number of CPU currently working in system : ", multiprocessing.cpu_count
    ())
```

7.2 Multiprocessus à l'aide de la classe Queue

Le multiprocessus Python est précisément le même que la structure de données queue, qui repose sur le concept "premier entré-premier sorti". La file d'attente stocke généralement les objets Python et joue un rôle essentiel dans le partage des données entre les processus.

Les files d'attente sont passées en paramètre dans la fonction cible du processus pour permettre à ce dernier de consommer des données. La file d'attente fournit la fonction put() pour insérer les données et la fonction get() pour récupérer les données des files d'attente. Comprenons l'exemple suivant.

Multi Processes (Queue)

```
from multiprocessing import Queue

fruits = ['Apple', 'Orange', 'Guava', 'Papaya', 'Banana']
count = 1
# creating a queue object
queue = Queue()
print('pushing items to the queue:')
for fr in fruits:
    print('item no: ', count, ' ', fr)
    queue.put(fr)
    count += 1

print('\npopping items from the queue:')
count = 0
print(queue)

while not queue.empty():
    print('item no: ', count, ' ', queue.get())
    count += 1
```

7.3 Classe de Lock multiprocesseur

La classe Lock pour le multiprocessus est utilisée pour acquérir un verrou sur le processus afin de permettre à l'autre processus d'exécuter un code similaire jusqu'à ce que le verrou soit libéré. La classe Lock effectue principalement deux tâches. La première consiste à acquérir un verrou à l'aide de la fonction acquire() et la seconde à libérer le verrou à l'aide de la fonction release().

Multi Task Processors

```
from multiprocessing import Lock, Process, Queue, current_process
import time
import queue
```

```
def jobTodo(tasks_to_perform, complete_tasks):
    while True:
        try:

            # The try block to catch task from the queue.
            # The get_nowait() function is used to
            # raise queue.Empty exception if the queue is empty.

            task = tasks_to_perform.get_nowait()

        except queue.Empty:

            break
        else:

                # if no exception has been raised, the else block will execute
                # add the task completion


            print(task)
            complete_tasks.put(task + ' is done by ' + current_process().name)
            time.sleep(.5)
    return True


def main():
    total_task = 8
    total_number_of_processes = 3
    tasks_to_perform = Queue()
    complete_tasks = Queue()
    number_of_processes = []

    for i in range(total_task):
        tasks_to_perform.put("Task no " + str(i))

    # defining number of processes
    for w in range(total_number_of_processes):
        p = Process(target=jobTodo, args=(tasks_to_perform, complete_tasks))
        number_of_processes.append(p)
        p.start()

    # completing process
    for p in number_of_processes:
        p.join()

    # print the output
```

```
    while not complete_tasks.empty():
        print(complete_tasks.get())

    return True


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

7.4 Multiprocessing Pool

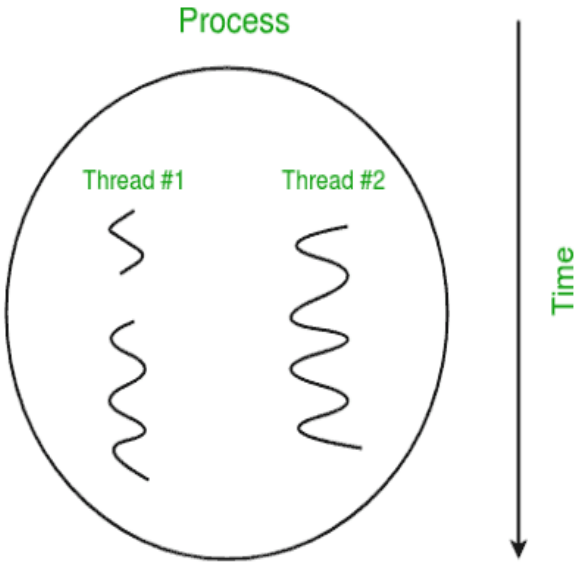
Le pool multiprocessus de Python est essentiel pour l'exécution parallèle d'une fonction sur plusieurs valeurs d'entrée. Il est également utilisé pour distribuer les données d'entrée entre les processus (parallélisme des données). Prenons l'exemple suivant d'un pool multiprocessus.

Multi Task Parellel

```
from multiprocessing import Pool
import time

w = (["V", 5], ["X", 2], ["Y", 1], ["Z", 3])

def work_log(data_for_work):
    print(" Process name is %s waiting time is %s seconds" % (data_for_work[0],
    data_for_work[1]))
    time.sleep(int(data_for_work[1]))
    print(" Process %s Executed." % data_for_work[0])


def handler():
    p = Pool(2)
    p.map(work_log, w)

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

7.5 Threads

Un thread est une entité au sein d'un processus dont l'exécution peut être programmée. Il s'agit également de la plus petite unité de traitement pouvant être exécutée dans un système d'exploitation. En d'autres termes, un thread est une séquence d'instructions au sein d'un programme qui peut être exécutée indépendamment d'autres codes. Pour simplifier, on peut supposer qu'un thread est simplement un sous-ensemble d'un processus.



Threads

```
# Python program to illustrate the concept
# of threading
# importing the threading module
import threading


def print_cube(num):
        # function to print cube of given num
        print("Cube: {}" .format(num * num * num))


def print_square(num):
        # function to print square of given num
        print("Square: {}" .format(num * num))


if __name__ =="__main__":
        # creating thread
        t1 = threading.Thread(target=print_square, args=(10,))
        t2 = threading.Thread(target=print_cube, args=(10,))

        # starting thread 1
        t1.start()
        # starting thread 2
        t2.start()

        # wait until thread 1 is completely executed
        t1.join()
        # wait until thread 2 is completely executed
        t2.join()
```

```
        # both threads completely executed
        print("Done!")
```

Threads wiyh id process

```
# Python program to illustrate the concept
# of threading
import threading
import os

def task1():
        print("Task 1 assigned to thread: {}".format(threading.current_thread().
    name))
        print("ID of process running task 1: {}".format(os.getpid()))

def task2():
        print("Task 2 assigned to thread: {}".format(threading.current_thread().
    name))
        print("ID of process running task 2: {}".format(os.getpid()))

if __name__ == "__main__":

        # print ID of current process
        print("ID of process running main program: {}".format(os.getpid()))

        # print name of main thread
        print("Main thread name: {}".format(threading.current_thread().name))

        # creating threads
        t1 = threading.Thread(target=task1, name='t1')
        t2 = threading.Thread(target=task2, name='t2')

        # starting threads
        t1.start()
        t2.start()

        # wait until all threads finish
        t1.join()
        t2.join()
```

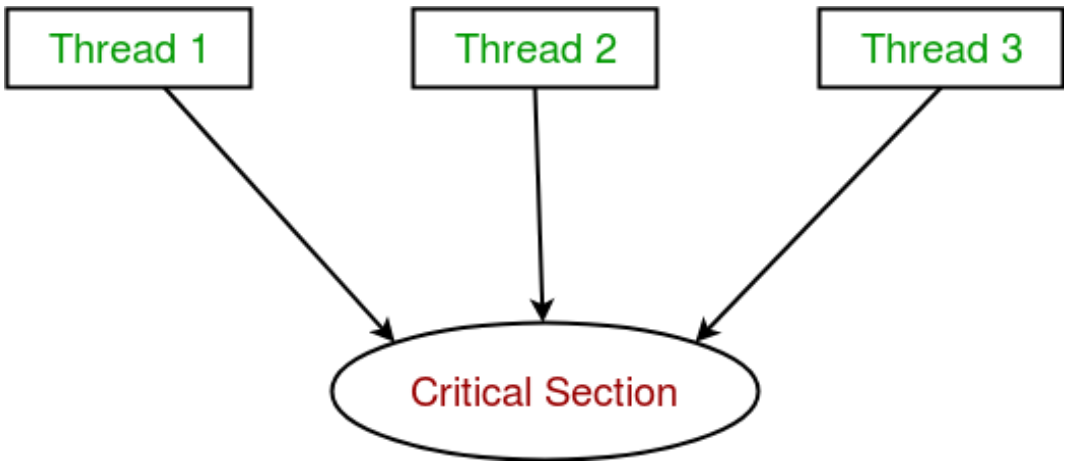
MultiThreads

```
import _thread
import time
# Define a function for the thread
def thread_task( threadName, delay):
   for count in range(1, 6):
```

```
      time.sleep(delay)
      print ("Thread name: {} Count: {}".format ( threadName, count ))

# Create two threads as follows
try:
   _thread.start_new_thread( thread_task, ("Thread-1", 2, ) )
   _thread.start_new_thread( thread_task, ("Thread-2", 4, ) )
except:
   print ("Error: unable to start thread")

while True:
   pass
```

7.5.1 synchronisation des threads

La synchronisation des threads est définie comme un mécanisme qui garantit que deux ou plusieurs threads concurrents n'exécutent pas simultanément un segment particulier du programme, connu sous le nom de section critique.



Les accès simultanés à des ressources partagées peuvent conduire à des conditions de course.

Une condition de course se produit lorsque deux threads ou plus peuvent accéder à des données partagées et qu'ils essaient de les modifier en même temps. Par conséquent, les valeurs des variables peuvent être imprévisibles et varier en fonction du moment où les processus changent de contexte.

Race Problem

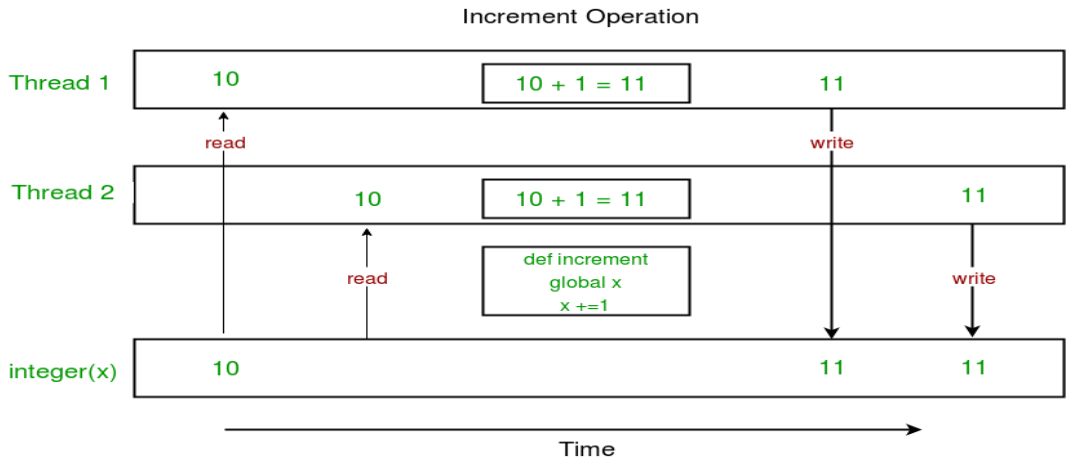
```
import threading

# global variable x
x = 0

def increment():
        """
```

```
        function to increment global variable x
        """
        global x
        x += 1

def thread_task():
        """
        task for thread
        calls increment function 100000 times.
        """
        for _ in range(100000):
                increment()

def main_task():
        global x
        # setting global variable x as 0
        x = 0

        # creating threads
        t1 = threading.Thread(target=thread_task)
        t2 = threading.Thread(target=thread_task)

        # start threads
        t1.start()
        t2.start()

        # wait until threads finish their job
        t1.join()
        t2.join()

if __name__ == "__main__":
        for i in range(10):
                main_task()
                print("Iteration {0}: x = {1}".format(i,x))
```

— Deux threads t1 et t2 sont créés dans la fonction main_task et la variable globale x est fixée à 0.
— Chaque thread a une fonction cible thread_task dans laquelle la fonction increment est appelée 100000 fois.
— La fonction increment incrémentera la variable globale x de 1 à chaque appel.

La valeur finale attendue de x est de 200 000, mais ce que nous obtenons après 10 itérations de la fonction main_task est différent.

Cela est dû à l'accès concurrent des threads à la variable partagée x. Cette imprévisibilité de la valeur de x n'est rien d'autre qu'une condition de course.

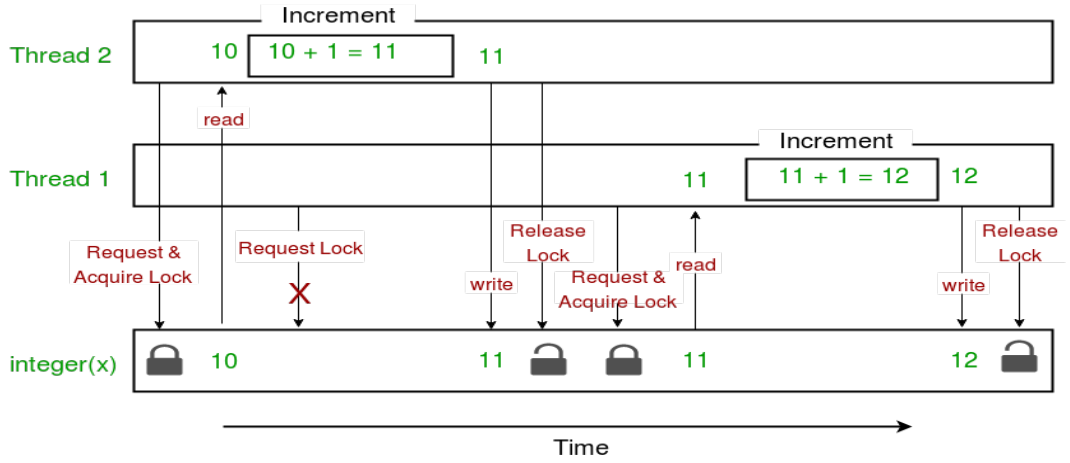


Remarquez que la valeur attendue de x dans le diagramme ci-dessus est 12, mais qu'en raison d'une condition de course, elle s'avère être 11 !

Nous avons donc besoin d'un outil permettant une synchronisation correcte entre plusieurs threads.

Utilisation des Locks : Le module de threading fournit une classe Lock pour gérer les conditions de course. Le Lock est implémenté à l'aide d'un objet sémaphore fourni par le système d'exploitation.

Un sémaphore est un objet de synchronisation qui contrôle l'accès de plusieurs processus/filières à une ressource commune dans un environnement de programmation parallèle.



Race Solution : Lock

```
import threading

# global variable x
x = 0

def increment():
        """
        function to increment global variable x
        """
        global x
```

```
        x += 1

def thread_task(lock):
        """
        task for thread
        calls increment function 100000 times.
        """
        for _ in range(100000):
                lock.acquire()
                increment()
                lock.release()

def main_task():
        global x
        # setting global variable x as 0
        x = 0

        # creating a lock
        lock = threading.Lock()

        # creating threads
        t1 = threading.Thread(target=thread_task, args=(lock,))
        t2 = threading.Thread(target=thread_task, args=(lock,))

        # start threads
        t1.start()
        t2.start()

        # wait until threads finish their job
        t1.join()
        t2.join()

if __name__ == "__main__":
        for i in range(10):
                main_task()
                print("Iteration {0}: x = {1}".format(i,x))
```

Chapitre 8

Python et Base de données (Mysql)

Python peut être utilisé dans les applications de base de données.L'une des bases de données les plus populaires est MySQL.
Python a besoin d'un pilote MySQL pour accéder à la base de données MySQL.

python -m pip install mysql-connector-python

Pour tester si l'installation a réussi, ou si vous avez déjà installé "MySQL Connector", créez une page Python avec le contenu suivant :

import mysql.connector

8.1 Créer une connexion

Commencez par créer une connexion à la base de données. Utilisez le nom d'utilisateur et le mot de passe de votre base de données MySQL :

Mysql Connection

```
import mysql.connector
mydb = mysql.connector.connect( host="localhost", user="yourusername", password="
    yourpassword")
print(mydb)
```

8.2 Création d'une base de données

Pour créer une base de données dans MySQL, utilisez l'instruction "CREATE DATABASE" :

BDD Creation

```
import mysql.connector
mydb = mysql.connector.connect( host="localhost", user="yourusername", password="
    yourpassword")
mycursor = mydb.cursor()
mycursor.execute("CREATE DATABASE mydatabase")
```

Si le code ci-dessus a été exécuté sans erreur, vous avez créé avec succès une base de données. Vérifier si la base de données existe :

Show BDDs

```
mycursor = mydb.cursor()
mycursor.execute("SHOW DATABASES")
for x in mycursor:
        print(x)
```

Ou on peut essayer d'accéder à la base de données lors de la connexion :

Test cnx BDD

```
import mysql.connector
mydb = mysql.connector.connect( host="localhost", user="yourusername", password="
    yourpassword",database="mydatabase")
```

8.3 Création d'un tableau

Create Table

```
import mysql.connector
mydb = mysql.connector.connect( host="localhost", user="yourusername", password="
    yourpassword", database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255))"
    )
```

8.4 Insérer dans le tableau

Pour remplir une table dans MySQL, utilisez l'instruction "INSERT INTO".

Mysql Insert

```
mycursor = mydb.cursor()
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)" val = ("John", "
    Highway 21")
mycursor.execute(sql, val)
mydb.commit()
print(mycursor.rowcount, "record inserted.")
```

Insérer plusieurs lignes :

Mysql Insert Set

```
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)" val = [
 ('Peter', 'Lowstreet 4'),
 ('Amy', 'Apple st 652'),
 ('Hannah', 'Mountain 21'),
 ('Michael', 'Valley 345'),
 ('Sandy', 'Ocean blvd 2'),
 ('Betty', 'Green Grass 1'),
 ('Richard', 'Sky st 331'),
 ('Susan', 'One way 98'),
 ('Vicky', 'Yellow Garden 2'),
 ('Ben', 'Park Lane 38'),
 ('William', 'Central st 954'),
```

```
 ('Chuck', 'Main Road 989'),
 ('Viola', 'Sideway 1633')
]
mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, "was inserted.")
```

Obtenir l'ID inséré :

Mysql Insert Return id

```
import mysql.connector
mydb = mysql.connector.connect( host="localhost", user="yourusername", password="
    yourpassword",
database="mydatabase")
mycursor = mydb.cursor()
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)" val = ("Michelle", "
    Blue Village")
mycursor.execute(sql, val)
mydb.commit()
print("1 record inserted, ID:", mycursor.lastrowid)
```

8.5 Sélectionner à partir d'un tableau

Pour effectuer une sélection dans une table de MySQL, utilisez l'instruction "SELECT" :

Select et Cursor

```
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchall()
for x in myresult:
        print(x)
```

Sélectionner avec un filtre :

elect et Cursor with param

```
mycursor = mydb.cursor()
sql = "SELECT * FROM customers WHERE address ='Park Lane 38'"
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
        print(x)
```

Caractères génériques On peut également sélectionner les enregistrements qui commencent, incluent ou se terminent par une lettre ou une expression donnée. Utilisez le % pour représenter les caractères génériques :

elect et Cursor Like

```
mycursor = mydb.cursor()
```

```
sql = "SELECT * FROM customers WHERE address LIKE '%way%'"
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
        print(x)
```

Empêcher l'injection SQL

Lorsque des valeurs de requête sont fournies par l'utilisateur, vous devez échapper les valeurs. Cela permet d'éviter les injections SQL, qui est une technique de piratage Web courante pour détruire ou abuser de votre base de données. Le module mysql.connector a des méthodes pour échapper les valeurs de requête :

Select et Cursor Prepared

```
mycursor = mydb.cursor()
sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )
mycursor.execute(sql, adr)
myresult = mycursor.fetchall()
for x in myresult:
        print(x)
```

8.6 Supprimer depuis d'un tableau

On peut supprimer des enregistrements d'une table existante à l'aide de l'instruction "DELETE FROM" :

Delete

```
mycursor = mydb.cursor()
sql = "DELETE FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )
mycursor.execute(sql, adr)
mydb.commit()
print(mycursor.rowcount, "record(s) deleted")
```

8.7 Tableau de mise à jour

On peut mettre à jour des enregistrements existants dans une table à l'aide de l'instruction "UPDATE" :

Update

```
mycursor = mydb.cursor()
sql = "UPDATE customers SET address = %s WHERE address = %s"
val = ("Valley 345", "Canyon 123") mycursor.execute(sql, val)
mydb.commit()
print(mycursor.rowcount, "record(s) affected")
```

8.8 Gestion des Transactions

Les transactions assurent la cohérence des données de la base. Nous devons nous assurer que plusieurs applications ne peuvent pas modifier les enregistrements lors de l'exécution des opérations sur la base de données. Les transactions ont les propriétés suivantes.

- Atomicité : Soit la transaction se termine, soit il ne se passe rien. Si une transaction contient 4 requêtes, toutes ces requêtes doivent être exécutées ou aucune d'entre elles ne doit être exécutée.
- Cohérence : La base de données doit être cohérente avant le début de la transaction et elle doit également être cohérente après la fin de la transaction.
- Isolement : Les résultats intermédiaires d'une transaction ne sont pas visibles en dehors de la transaction en cours.
- Durabilité : Une fois qu'une transaction a été validée, ses effets sont persistants, même après une défaillance du système.

Méthode Python commit() : Python fournit la méthode commit() qui permet de s'assurer que les modifications apportées à la base de données sont toujours prises en compte.

Méthode Python rollback() : La méthode rollback() est utilisée pour annuler les modifications apportées à la base de données. Cette méthode est utile dans le sens où, si une erreur se produit pendant les opérations sur la base de données, nous pouvons annuler cette transaction pour maintenir la cohérence de la base de données.

Close de la connexion : Nous devons fermer la connexion à la base de données une fois que nous avons effectué toutes les opérations concernant la base de données. Python fournit la méthode close(). La syntaxe pour utiliser la méthode close() est donnée ci-dessous.

Transaction

```
import mysql.connector

#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google
    ",database = "PythonDB")

#creating the cursor object
cur = myconn.cursor()

try:
    cur.execute("delete from Employee where Dept_id = 201")
    myconn.commit()
    print("Deleted !")
except:
    print("Can't delete !")
    myconn.rollback()

myconn.close()
```

Chapitre 9

Flask et Rest API

Flask est un framework d'application Web écrit en Python. Armin Ronacher, qui dirige un groupe international de passionnés de Python nommé Pocco, le développe. Flask est basé sur la boîte à outils Werkzeug WSGI et le moteur de modèles Jinja2. Les deux sont des projets Pocco.

Qu'est-ce qu'un Web Framework

Web Application Framework ou simplement Web Framework représente une collection de bibliothèques et de modules qui permettent à un développeur d'applications Web d'écrire des applications sans avoir à se soucier des détails de bas niveau tels que les protocoles, la gestion des threads, etc.

Qu'est-ce que Flask

Flask est un framework d'application Web écrit en Python. Il est développé par Armin Ronacher, qui dirige un groupe international de passionnés de Python nommé Pocco. Flask est basé sur la boîte à outils Werkzeug WSGI et le moteur de modèles Jinja2. Les deux sont des projets Pocco.

WSGI

L'interface de passerelle de serveur Web (WSGI) a été adoptée comme norme pour le développement d'applications Web Python. WSGI est une spécification pour une interface universelle entre le serveur Web et les applications Web.

Werkzeug

Il s'agit d'une boîte à outils WSGI, qui implémente des requêtes, des objets de réponse et d'autres fonctions utilitaires. Cela permet de créer un framework Web par-dessus. Le framework Flask utilise Werkzeug comme l'une de ses bases.

Jinja2

Jinja2 est un moteur de template populaire pour Python. Un système de modèles Web combine un modèle avec une certaine source de données pour afficher des pages Web dynamiques. Flask est souvent appelé un micro framework. Il vise à garder le noyau d'une application simple mais extensible. Flask n'a pas de couche d'abstraction intégrée pour la gestion de la base de données, ni de support de validation. Au lieu de cela, Flask prend en charge les extensions pour ajouter de telles fonctionnalités à l'application. Certaines des extensions Flask populaires sont abordées plus loin dans le didacticiel.

9.1 Flask – Application

FLask

```
from flask import Flask
```

```
app = Flask(__name__)
@app.route('/')
def hello_world():
   return 'Hello World'
if __name__ == '__main__':
   app.run()
```

L'importation du module flask dans le projet est obligatoire. Un objet de la classe Flask est notre application WSGI. Le constructeur Flask prend le nom du module actuel (___name___) comme argument.
La fonction route() de la classe Flask est un décorateur, qui indique à l'application quelle URL doit appeler la fonction associée.

app.route(rule, options)

- Le paramètre de rule représente la liaison d'URL avec la fonction.
- Les options sont une liste de paramètres à transmettre à l'objet Rule sous-jacent.

la méthode run() de la classe Flask exécute l'application sur le serveur de développement local.

app.run(host, port, debug, options)

Parameters & Description

- Host : Nom d'hôte sur lequel écouter. La valeur par défaut est 127.0.0.1 (localhost). Définir sur "0.0.0.0" pour que le serveur soit disponible en externe.
- Port par defaut 5000
- Debug : La valeur par défaut est false. Si défini sur true, fournit des informations de débogage
- Options : À transmettre au serveur Werkzeug sous-jacent.

9.2 Flask – Variable Rules

Il est possible de construire une URL dynamiquement, en ajoutant des parties variables au paramètre rule. Cette partie variable est marquée comme <variable-name>. Il est passé en tant qu'argument mot-clé à la fonction à laquelle la règle est associée. Dans l'exemple suivant, le paramètre de règle du décorateur route() contient une partie variable <name> attachée à l'URL "/hello". Par conséquent, si le http ://localhost :5000/hello/TutorialsPoint est entré comme URL dans le navigateur, 'TutorialPoint' sera fourni à la fonction hello() comme argument.

FLask / Variables

```
from flask import Flask
app = Flask(__name__)
@app.route('/hello/<name>')
def hello_name(name):
   return 'Hello %s!' % name
if __name__ == '__main__':
   app.run(debug = True)
```

9.3 Flask – HTTP methods

Le protocole HTTP est le fondement de la communication de données sur le World Wide Web. Différentes méthodes de récupération de données à partir d'une URL spécifiée sont définies dans ce protocole.
La liste suivante résume les différentes méthodes http.

- GET : Envoie les données sous forme non cryptée au serveur. Méthode la plus courante.
- HEAD : Identique à GET, mais sans corps de réponse.
- POST : Utilisé pour envoyer des données de formulaire HTML au serveur. Les données reçues par la méthode POST ne sont pas mises en cache par le serveur.
- PUT : Remplace toutes les représentations actuelles de la ressource cible par le contenu téléchargé.
- DELETE : Supprime toutes les représentations actuelles de la ressource cible donnée par une URL.

9.4 Flask-WTF

WTF signifie WT Forms et a pour but de fournir une interface utilisateur interactive à l'utilisateur. Le WTF est un module intégré de flask qui offre une alternative pour la conception de formulaires dans les applications web de flask.

Le WTF est utile pour les raisons suivantes.

- Les éléments du formulaire sont envoyés avec l'objet de la demande du côté client au côté serveur. Le script côté serveur doit recréer les éléments du formulaire car il n'y a pas de correspondance directe entre les éléments du formulaire côté client et les variables à utiliser côté serveur.
- Il n'y a aucun moyen de rendre les données du formulaire HTML en temps réel.

WT Forms est une bibliothèque flexible de rendu et de validation de formulaires utilisée pour fournir l'interface utilisateur.

Class Form

```
from flask_wtf import Form
from wtforms import TextField, IntegerField, TextAreaField, SubmitField, RadioField
    , SelectField
from wtforms import validators, ValidationError

class ContactForm(Form):
   name = TextField("Candidate Name ",[validators.Required("Please enter your name.
   ")])
   Gender = RadioField('Gender', choices = [('M','Male'),('F','Female')])
   Address = TextAreaField("Address")

   email = TextField("Email",[validators.Required("Please enter your email address.
   "),
   validators.Email("Please enter your email address.")])

   Age = IntegerField("Age")
```

```
   language = SelectField('Programming Languages', choices = [('java', 'Java'),('py
    ', 'Python')])

   submit = SubmitField("Submit")
```

Bakend flask

```
from flask import Flask, render_template, request, flash
from forms import ContactForm
app = Flask(__name__)
app.secret_key = 'development key'

@app.route('/contact', methods = ['GET', 'POST'])
def contact():
   form = ContactForm()
   if form.validate() == False:
      flash('All fields are required.')
   return render_template('contact.html', formform = form)



@app.route('/success',methods = ['GET','POST'])
def success():
   return render_template("success.html")

if __name__ == '__main__':
   app.run(debug = True)
```

Form HTML

```
<!doctype html>
<html>
   <body>
      <h2 style = "text-align: center;">Registration Form</h2>

      {% for message in form.name.errors %}
         <div>{{ message }}</div>
      {% endfor %}

      {% for message in form.email.errors %}
         <div>{{ message }}</div>
      {% endfor %}

      <form action = "http://localhost:5000/success" method = "POST">

            {{ form.hidden_tag() }}

            <div style = "font-size:18px;" font-weight:bold; margin-left:150px;>
```

```
                {{ form.name.label }}<br>
                {{ form.name }}
                <br>
                {{ form.Gender.label }} {{ form.Gender }}
                {{ form.Address.label }}<br>
                {{ form.Address }}
                <br>
                {{ form.email.label }}<br>
                {{ form.email }}
                <br>
                {{ form.Age.label }}<br>
                {{ form.Age }}
                <br>
                {{ form.language.label}}<br><br>
                {{ form.language }}
                <br><br>
                {{ form.submit }}
            </div>

        </fieldset>
    </form>
  </body>
</html>
```

Result HTML

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
<h1>Form posted successfully</h1>
</body>
</html>
```

Chapitre 10

Analyse de données

L'analyse des données est la technique de collecte, de transformation et d'organisation des données qui permet de faire des prévisions et de prendre des décisions éclairées fondées sur les données. Elle permet également de trouver des solutions possibles à un problème commercial. L'analyse des données comporte six étapes. Elles sont les suivantes :

— Demander ou spécifier les exigences en matière de données
— Préparer ou collecter les données
— Nettoyer et traiter
— Analyser
— Partager
— Agir ou rapporter

10.1 Analyse de données numériques avec NumPy

NumPy est un paquetage de traitement de tableaux en Python qui fournit un objet de tableau multidimensionnel à haute performance et des outils pour travailler avec ces tableaux. Il s'agit du paquetage fondamental pour le calcul scientifique avec Python.

Numpy

```
import numpy as np

# Defining both the matrices
a = np.array([5, 72, 13, 100])
b = np.array([2, 5, 10, 30])

# Performing addition using arithmetic operator
add_ans = a+b
print(add_ans)

# Performing addition using numpy function
add_ans = np.add(a, b)
print(add_ans)

# The same functions and operations can be used for
# multiple matrices
c = np.array([1, 2, 3, 4])
add_ans = a+b+c
print(add_ans)

add_ans = np.add(a, b, c)
print(add_ans)
```

```
sub_ans = a-b
print(sub_ans)

# Performing subtraction using numpy function
sub_ans = np.subtract(a, b)
print(sub_ans)

 Performing multiplication using arithmetic
# operator
mul_ans = a*b
print(mul_ans)

# Performing multiplication using numpy function
mul_ans = np.multiply(a, b)
print(mul_ans)

# Performing division using arithmetic operators
div_ans = a/b
print(div_ans)

# Performing division using numpy functions
div_ans = np.divide(a, b)
print(div_ans)
```

10.1.1 Broadcasting avec les tableaux NumPy

Le terme broadcasting fait référence à la manière dont numpy traite les tableaux de dimensions différentes pendant les opérations arithmétiques qui conduisent à certaines contraintes, le plus petit tableau est diffusé sur le plus grand tableau de manière à ce qu'ils aient des formes compatibles.

Brodcasting Naive Way

```
macros = array([
  [0.8, 2.9, 3.9],
  [52.4, 23.6, 36.5],
  [55.2, 31.7, 23.9],
  [14.4, 11, 4.9]
   ])

# Create a new array filled with zeros,
# of the same shape as macros.
result = zeros_like(macros)

cal_per_macro = array([3, 3, 8])

# Now multiply each row of macros by
# cal_per_macro. In Numpy, `*` is
# element-wise multiplication between two arrays.
 for i in range(macros.shape[0]):
     result[i, :] = macros[i, :] * cal_per_macro

result
```

Brodcasting

```
import numpy as np
a = np.array([17, 11, 19]) # 1x3 Dimension array
print(a)
b = 3
print(b)

# Broadcasting happened because of
# miss match in array Dimension.
c = a + b
print(c)

A = np.array([[11, 22, 33], [10, 20, 30]])
print(A)

b = 4
print(b)

C = A + b
print(C)
```

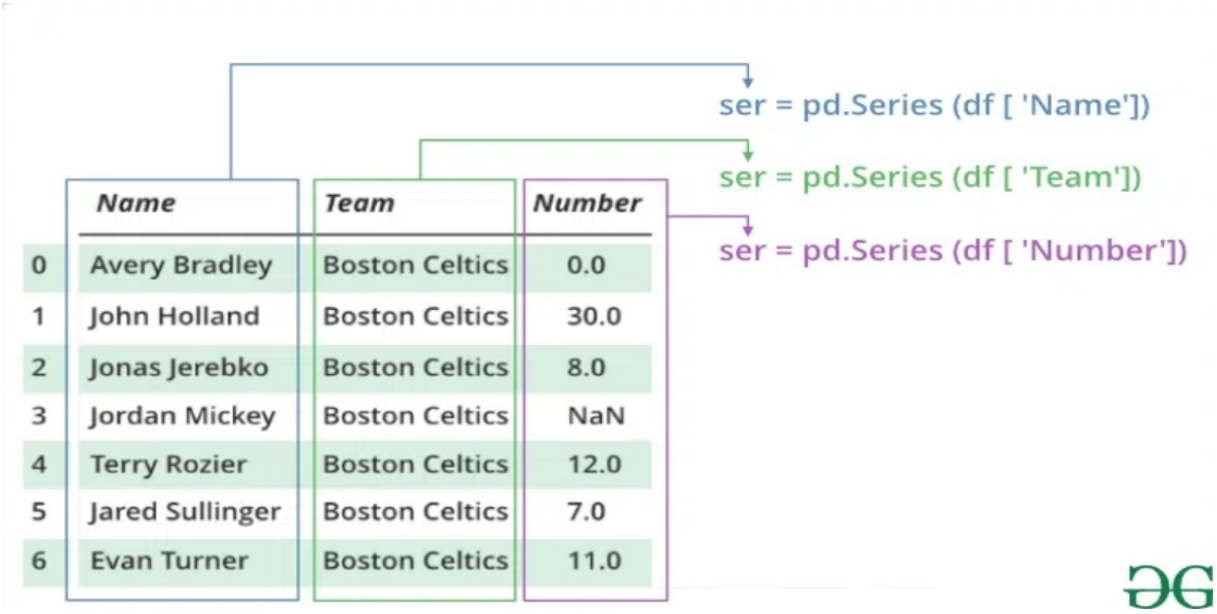
10.2 Analyser des données à l'aide de Pandas

Python Pandas est utilisé pour les données relationnelles ou étiquetées et fournit diverses structures de données pour manipuler de telles données et séries temporelles. Cette bibliothèque est construite au-dessus de la bibliothèque NumPy.

Pandas propose généralement deux structures de données pour manipuler les données :
- Série
- DataFram

10.2.1 La série Pandas

La série Pandas est un tableau unidimensionnel étiqueté capable de contenir des données de n'importe quel type (entier, chaîne, flottant, objets python, etc.). Les étiquettes des axes sont collectivement appelées index. La série Pandas n'est rien d'autre qu'une colonne dans une feuille Excel. Les étiquettes n'ont pas besoin d'être uniques, mais doivent être de type hachable. L'objet prend en charge l'indexation basée sur les nombres entiers et les étiquettes et fournit une série de méthodes pour effectuer des opérations impliquant l'index.



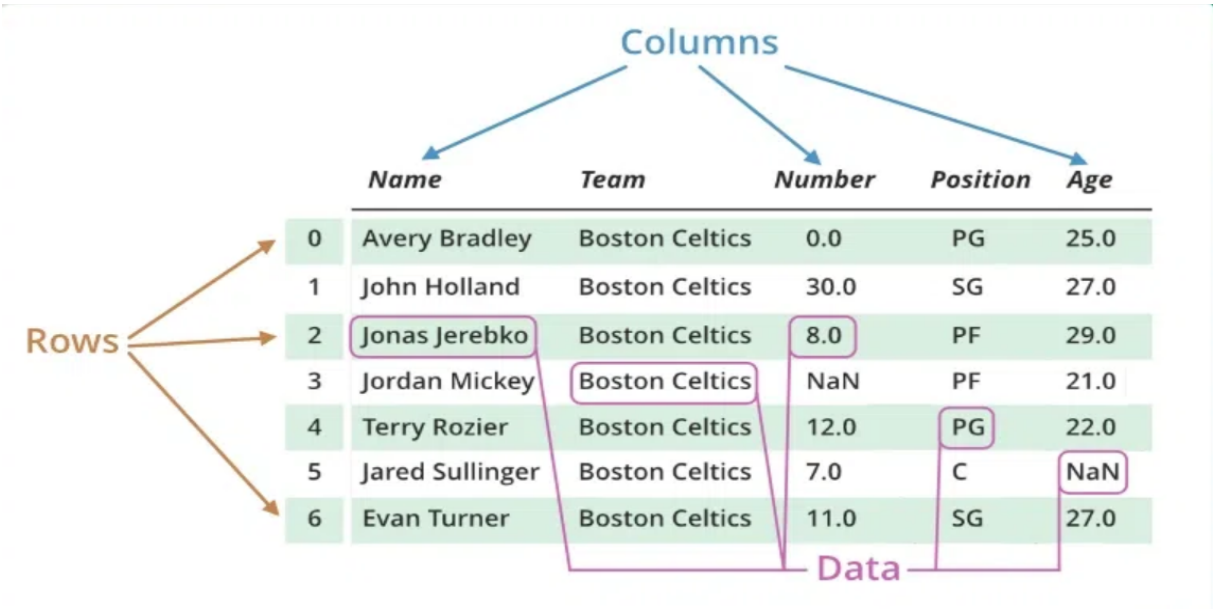
Pandas Serie

```
import pandas as pd
import numpy as np


# Creating empty series
ser = pd.Series()

print(ser)

# simple array
data = np.array(['g', 'e', 'e', 'k', 's'])
```

```
ser = pd.Series(data)
print(ser)
```

10.2.2 La DataFrame

Le DataFrame de Pandas est une structure de données tabulaires bidimensionnelle à taille variable, potentiellement hétérogène, avec des axes étiquetés (lignes et colonnes). Un cadre de données est une structure de données bidimensionnelle, c'est-à-dire que les données sont alignées de manière tabulaire en lignes et en colonnes. Le DataFrame de Pandas se compose de trois éléments principaux : les données, les lignes et les colonnes.



Pandas DataFrame

```
import pandas as pd

# Calling DataFrame constructor
df = pd.DataFrame()
print(df)

# list of strings
lst = ['Geeks', 'For', 'Geeks', 'is',
                        'portal', 'for', 'Geeks']

# Calling DataFrame constructor on list
df = pd.DataFrame(lst)
df
```

Création d'un DataFrame à partir d'un fichier CSV

Pandas CSV

```
import pandas as pd
```

```
# Reading the CSV file
df = pd.read_csv("Iris.csv")

# Printing top 5 rows
df.head()
```

10.2.3 Filtrage d'un DataFrame

La fonction Pandas dataframe.filter() est utilisée pour subdiviser les lignes ou les colonnes d'un tableau de données en fonction des étiquettes de l'index spécifié. Notez que cette routine ne filtre pas un dataframe sur son contenu. Le filtre est appliqué aux étiquettes de l'index.

Pandas Filter

```
import pandas as pd

# Reading the CSV file
df = pd.read_csv("Iris.csv")

# applying filter function
df.filter(["Species", "SepalLengthCm", "SepalLengthCm"]).head()
```

10.2.4 Tri des données d'un DataFrame

La fonction sort_values() de Pandas est utilisée pour trier la base de données. Pandas sort_values() peut trier la base de données par ordre croissant ou décroissant.

Pandas Sort

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("Iris.csv")

# sorting data frame by name
data.sort_values("Species", axis = 0, ascending = True,
                                 inplace = True, na_position ='last')

# display
data
```

10.2.5 Pandas GroupBy

Groupby est un concept assez simple. Nous pouvons créer un regroupement de catégories et appliquer une fonction à ces catégories. Dans les projets réels de science des données, vous aurez à traiter de grandes quantités de données et à essayer des choses à plusieurs reprises, c'est

pourquoi, pour des raisons d'efficacité, nous utilisons le concept de Groupby. Le regroupement se réfère principalement à un processus impliquant une ou plusieurs des étapes suivantes :

- Splitting : Il s'agit d'un processus qui consiste à diviser les données en groupes en appliquant certaines conditions aux ensembles de données.
- Applying : Il s'agit d'un processus au cours duquel nous appliquons une fonction à chaque groupe indépendamment.
- Combining : Il s'agit d'un processus au cours duquel nous combinons différents ensembles de données après avoir appliqué la fonction groupby et les résultats dans une structure de données.

Pandas GroupBy

```
# importing pandas module
import pandas as pd

# Define a dictionary containing employee data
data1 = {'Name': ['Jai', 'Anuj', 'Jai', 'Princi',
                                'Gaurav', 'Anuj', 'Princi', 'Abhi'],
                'Age': [27, 24, 22, 32,
                                33, 36, 27, 32],
                'Address': ['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj',
                                        'Jaunpur', 'Kanpur', 'Allahabad', 'Aligarh'
    ],
                'Qualification': ['Msc', 'MA', 'MCA', 'Phd',
                                                'B.Tech', 'B.com', 'Msc', 'MA']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data1)

print("Original Dataframe")
display(df)

# applying groupby() function to
# group the data on Name value.
gk = df.groupby('Name')

# Let's print the first entries
# in all the groups formed.
print("After Creating Groups")
gk.first()
```

Pandas GroupBy 2

```
import pandas as pd
technologies   = ({
    'Courses':["Spark","PySpark","Hadoop","Python","Pandas","Hadoop","Spark","
    Python","NA"],
    'Fee' :[22000,25000,23000,24000,26000,25000,25000,22000,1500],
```

```
    'Duration':['30days','50days','55days','40days','60days','35days','30days','50
   days','40days'],
    'Discount':[1000,2300,1000,1200,2500,None,1400,1600,0]
          })
df = pd.DataFrame(technologies)
print(df)

# Use groupby() to compute the sum
df2 =df.groupby(['Courses']).sum()
print(df2)
```

10.2.6 Pandas Aggregation

L'agrégation est un processus par lequel nous calculons une statistique récapitulative pour chaque groupe. La fonction aggregated renvoie une seule valeur agrégée pour chaque groupe. Après avoir divisé les données en groupes à l'aide de la fonction groupby, plusieurs opérations d'agrégation peuvent être effectuées sur les données groupées.

Pandas Aggregation

```
# importing pandas module
import pandas as pd

# importing numpy as np
import numpy as np

# Define a dictionary containing employee data
data1 = {'Name': ['Jai', 'Anuj', 'Jai', 'Princi',
                  'Gaurav', 'Anuj', 'Princi', 'Abhi'],
         'Age': [27, 24, 22, 32,
                 33, 36, 27, 32],
         'Address': ['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj',
                     'Jaunpur', 'Kanpur', 'Allahabad', 'Aligarh'],
         'Qualification': ['Msc', 'MA', 'MCA', 'Phd',
                           'B.Tech', 'B.com', 'Msc', 'MA']}


# Convert the dictionary into DataFrame
df = pd.DataFrame(data1)

# performing aggregation using
# aggregate method

grp1 = df.groupby('Name')

print(grp1.aggregate(np.sum))
```

10.2.7 Concatenating DataFrame

Pour concaténer le cadre de données, nous utilisons la fonction concat() qui aide à concaténer le cadre de données. Cette fonction effectue toutes les opérations lourdes de concaténation avec un axe d'objets Pandas tout en exécutant une logique d'ensemble optionnelle (union ou intersection) des index (le cas échéant) sur les autres axes.

Pandas Concatenation

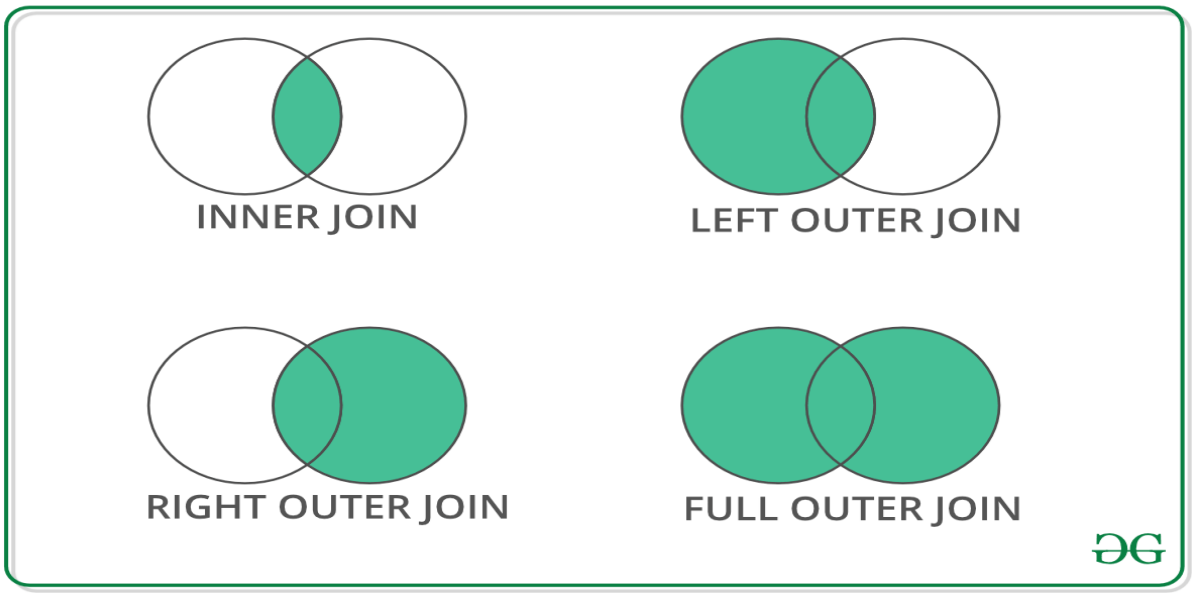
```
# importing pandas module
import pandas as pd

# Define a dictionary containing employee data
data1 = {'key': ['K0', 'K1', 'K2', 'K3'],
         'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
        'Age':[27, 24, 22, 32],}

# Define a dictionary containing employee data
data2 = {'key': ['K0', 'K1', 'K2', 'K3'],
         'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
        'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data1)

# Convert the dictionary into DataFrame
df1 = pd.DataFrame(data2)


display(df, df1)

# combining series and dataframe
res = pd.concat([df, df1], axis=1)

res
```

10.2.8 Merging DataFrame

Lorsque nous devons combiner de très grands DataFrames, les jointures constituent un moyen puissant d'effectuer ces opérations rapidement. Les jointures ne peuvent être effectuées que sur deux DataFrames à la fois, appelées tables de gauche et de droite. La clé est la colonne commune sur laquelle les deux cadres de données seront joints. C'est une bonne pratique d'utiliser des clés qui ont des valeurs uniques dans toute la colonne afin d'éviter la duplication involontaire des valeurs des lignes. Pandas fournit une fonction unique, merge(), comme point d'entrée pour

toutes les opérations de jointure de base de données standard entre les objets DataFrame.

Il existe quatre façons de gérer la jointure (inner, left, right, and outer), en fonction des lignes qui doivent conserver leurs données.



Pandas Merge

```
# importing pandas module
import pandas as pd

# Define a dictionary containing employee data
data1 = {'key': ['K0', 'K1', 'K2', 'K3'],
                'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
                'Age':[27, 24, 22, 32],}

# Define a dictionary containing employee data
data2 = {'key': ['K0', 'K1', 'K2', 'K3'],
                'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
                'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data1)

# Convert the dictionary into DataFrame
df1 = pd.DataFrame(data2)


display(df, df1)

# using .merge() function
res = pd.merge(df, df1, on='key')
```

```
res
```

10.2.9 Joining DataFrame

Pour joindre les dataframes, nous utilisons la fonction .join(). Cette fonction est utilisée pour combiner les colonnes de deux DataFrames potentiellement indexées différemment en un seul DataFrame de résultat.

Pandas Merge

```
# importing pandas module
import pandas as pd

# Define a dictionary containing employee data
data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
                'Age':[27, 24, 22, 32]}

# Define a dictionary containing employee data
data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
                'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data1,index=['K0', 'K1', 'K2', 'K3'])

# Convert the dictionary into DataFrame
df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])


display(df, df1)

# joining dataframe
res = df.join(df1)

res
```

10.3 Visualization with Matplotlib

La visualisation des données est la présentation des données sous forme d'images. Elle est extrêmement importante pour l'analyse des données, principalement en raison du fantastique écosystème de packages Python centrés sur les données. Elle aide à comprendre les données, quelle que soit leur complexité, leur importance en résumant et en présentant une grande quantité de données dans un format simple et facile à comprendre, et permet de communiquer des informations de manière claire et efficace.

10.3.1 Pyplot

Pyplot est un module de Matplotlib qui fournit une interface de type MATLAB. Pyplot fournit des fonctions qui interagissent avec la figure, c'est-à-dire qu'il crée une figure, décore le tracé avec des étiquettes et crée une zone de traçage dans une figure.

Pyplot plot

```
# Python program to show pyplot module
import matplotlib.pyplot as plt


plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.axis([0, 6, 0, 20])
plt.show()
```

Bar chart

Un diagramme à barres est un graphique qui représente la catégorie de données par des barres rectangulaires dont la longueur et la hauteur sont proportionnelles aux valeurs qu'elles représentent. Les diagrammes en bâtons peuvent être tracés horizontalement ou verticalement. Un diagramme à barres décrit les comparaisons entre les catégories discrètes. Il peut être créé à l'aide de la méthode bar().

Pyplot Bar chart

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv("Iris.csv")

# This will plot a simple bar chart
plt.bar(df['Species'], df['SepalLengthCm'])

# Title to the plot
plt.title("Iris Dataset")

# Adding the legends
plt.legend(["bar"])
plt.show()
```

Histogrammes

Un histogramme est essentiellement utilisé pour représenter des données sous forme de groupes. Il s'agit d'un type de diagramme à barres où l'axe X représente les intervalles de valeurs et l'axe Y donne des informations sur la fréquence. Pour créer un histogramme, la première étape consiste à créer un intervalle de valeurs, puis à répartir l'ensemble des valeurs en une série d'intervalles et à compter les valeurs qui tombent dans chacun de ces intervalles. Les cellules sont clairement identifiées comme des intervalles consécutifs de variables qui ne se chevauchent pas. La fonction hist() est utilisée pour calculer et créer un histogramme de x.

Pyplot Histogram

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv("Iris.csv")

plt.hist(df["SepalLengthCm"])

# Title to the plot
plt.title("Histogram")

# Adding the legends
plt.legend(["SepalLengthCm"])
plt.show()
```

Scatter Plot

Les diagrammes de dispersion sont utilisés pour observer la relation entre les variables et utilisent des points pour représenter la relation entre elles. La méthode scatter() de la bibliothèque matplotlib est utilisée pour dessiner un diagramme de dispersion.

Pyplot Scatter

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv("Iris.csv")

plt.scatter(df["Species"], df["SepalLengthCm"])

# Title to the plot
plt.title("Scatter Plot")

# Adding the legends
plt.legend(["SepalLengthCm"])
plt.show()
```

Box Plot

Un diagramme en boîte, une corrélation, également connu sous le nom de diagramme en boîte et de diagramme à moustaches. Il s'agit d'une très bonne représentation visuelle lorsqu'il s'agit de mesurer la distribution des données. Il représente clairement les valeurs médianes, les valeurs aberrantes et les quartiles. La compréhension de la distribution des données est un autre facteur important qui permet de mieux construire les modèles. Si les données présentent des valeurs aberrantes, le diagramme en boîte est un moyen recommandé de les identifier et de prendre les mesures nécessaires. Le diagramme en boîte et moustaches montre comment les données sont réparties. Cinq éléments d'information sont généralement inclus dans le graphique.

— Le minimum est indiqué à l'extrême gauche du graphique, à l'extrémité de la "moustache" gauche.
— Le premier quartile, Q1, se trouve à l'extrême gauche de la boîte (moustache de gauche).
— La médiane est représentée par une ligne au centre de la boîte.
— Troisième quartile, Q3, à l'extrême droite de la boîte (moustache droite)
— Le maximum se trouve à l'extrême droite de la boîte.

Pyplot Box plot

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv("Iris.csv")

plt.boxplot(df["SepalWidthCm"])

# Title to the plot
plt.title("Box Plot")

# Adding the legends
plt.legend(["SepalWidthCm"])
plt.show()
```

Correlation Heatmap

Une carte thermique en 2D est un outil de visualisation des données qui permet de représenter l'ampleur du phénomène sous forme de couleurs. Une carte thermique de corrélation est une carte thermique qui montre une matrice de corrélation en 2D entre deux dimensions discrètes, en utilisant des cellules colorées pour représenter les données à partir d'une échelle généralement monochromatique. Les valeurs de la première dimension apparaissent comme les lignes du tableau, tandis que la seconde dimension est une colonne. La couleur de la cellule est proportionnelle au nombre de mesures qui correspondent à la valeur de la dimension. Les cartes thermiques de corrélation sont donc idéales pour l'analyse des données, car elles rendent les modèles facilement lisibles et mettent en évidence les différences et les variations dans les mêmes données. Une carte thermique de corrélation, comme une carte thermique classique, est accompagnée d'une barre de couleur qui rend les données facilement lisibles et compréhensibles.

Pyplot Box plot

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv("Iris.csv")

plt.imshow(df.corr() , cmap = 'autumn' , interpolation = 'nearest' )

plt.title("Heat Map")
plt.show()
```

10.3.2 Seaborn

Seaborn est une bibliothèque de visualisation étonnante pour les graphiques statistiques en Python. Elle est construite au-dessus de la bibliothèque matplotlib et est également étroitement intégrée dans les structures de données de pandas.

Seabron

```
# Importing libraries
import numpy as np
import seaborn as sns


# Selecting style as white,
# dark, whitegrid, darkgrid
# or ticks
sns.set( style = "white" )

# Generate a random univariate
# dataset
rs = np.random.RandomState( 10 )
d = rs.normal( size = 50 )

# Plot a simple histogram and kde
# with binsize determined automatically
sns.distplot(d, kde = True, color = "g")
```

pour plus de details : https://seaborn.pydata.org/tutorial.html

10.4 Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) se réfère à la méthode d'étude et d'exploration d'ensembles d'enregistrements afin d'appréhender leurs caractéristiques prédominantes, de découvrir des modèles, de localiser les valeurs aberrantes et d'identifier les relations entre les variables. L'EDA est normalement réalisée comme une étape préliminaire avant d'entreprendre des analyses statistiques ou des modélisations plus formelles.

Les principaux objectifs de l'AED

1. Nettoyage des données : L'EDA consiste à examiner les informations à la recherche d'erreurs, de valeurs manquantes et d'incohérences. Elle comprend des techniques telles que l'imputation des enregistrements, la gestion des statistiques manquantes et l'identification et l'élimination des valeurs aberrantes.
2. Statistiques descriptives : L'EDA utilise des enregistrements précis pour reconnaître la tendance principale, la variabilité et la distribution des variables. Des mesures telles que la suggestion, la médiane, le mode, l'écart préférentiel, l'étendue et les percentiles sont généralement utilisées.

3. Visualisation des données : L'EDA utilise des techniques visuelles pour représenter les statistiques sous forme de graphiques. Les visualisations consistant en des histogrammes, des diagrammes en boîte, des diagrammes de dispersion, des diagrammes linéaires, des cartes thermiques et des diagrammes à barres permettent d'identifier les styles, les tendances et les relations au sein des faits.
4. Ingénierie des caractéristiques : L'EDA permet d'explorer diverses variables et leurs ajustements afin de créer de nouvelles fonctions ou d'obtenir des informations significatives. L'ingénierie des caractéristiques peut comprendre la mise à l'échelle, la normalisation, le regroupement, l'encodage de variables expresses et la création d'interactions ou de variables dérivées.
5. Corrélation et relations : L'EDA permet de découvrir les relations et les dépendances entre les variables. Des techniques telles que l'analyse des corrélations, les diagrammes de dispersion et les tableaux de passage donnent un aperçu de la puissance et de la direction des relations entre les variables.
6. Segmentation des données : L'EDA peut consister à diviser l'information en segments significatifs sur la base de normes ou de caractéristiques précises. Cette segmentation permet de mieux comprendre les sous-groupes uniques à l'intérieur de l'information et peut donner lieu à une analyse plus ciblée.
7. Génération d'hypothèses : L'EDA aide à générer des hypothèses ou des questions d'études basées totalement sur l'exploration préliminaire des données. Elle facilite la création d'une source d'inspiration pour l'évaluation et la construction de modèles.
8. Évaluation de la qualité des données : L'EDA permet d'évaluer la qualité et la fiabilité des informations. Elle implique la vérification de l'intégrité, de la cohérence et de l'exactitude des enregistrements afin de s'assurer que les données sont exactes.

10.4.1 Etude de cas 1

EDA 1

```
import pandas as pd
import numpy as np
# read datasdet using pandas
df = pd.read_csv('employees.csv')
df.head()

df.shape

df.describe()

df.info()

# convert "Start Date" column to datetime data type
df['Start Date'] = pd.to_datetime(df['Start Date'])

df.nunique()
```

Traitement des valeurs manquantes

Vous devez tous vous demander pourquoi un ensemble de données contient des valeurs manquantes. Cela peut se produire lorsqu'aucune information n'est fournie pour un ou plusieurs éléments ou pour une unité entière. Par exemple, supposons que différents utilisateurs interrogés choisissent de ne pas communiquer leurs revenus, et que certains utilisateurs choisissent de ne pas communiquer leur adresse ; de cette manière, de nombreux ensembles de données sont manquants. Les données manquantes constituent un très gros problème dans les scénarios de la vie réelle. Les données manquantes sont également appelées valeurs NA (Not Available) dans pandas. Il existe plusieurs fonctions utiles pour détecter, supprimer et remplacer les valeurs nulles dans Pandas DataFrame :

- isnull()
- notnull()
- dropna()
- fillna()
- replace()
- interpolate() https://www.geeksforgeeks.org/python-pandas-dataframe-interpolate/

EDA 1

```
import pandas as pd
import numpy as np
# read datasdet using pandas
df = pd.read_csv('employees.csv')
df.head()

df.shape

df.describe()

df.info()

# convert "Start Date" column to datetime data type
df['Start Date'] = pd.to_datetime(df['Start Date'])

df.nunique()

df.isnull().sum()

df["Gender"].fillna("Unkown", inplace = True)

df.isnull().sum()

mode = df['Senior Management'].mode().values[0]
df['Senior Management']= df['Senior Management'].replace(np.nan, mode)

df.isnull().sum()

df = df.dropna(axis = 0, how ='any')
```

```
print(df.isnull().sum())
df.shape
```

Data Encoding

Certains modèles, comme la régression linéaire, ne fonctionnent pas avec les données catégorielles. Dans ce cas, nous devons essayer d'encoder les données catégorielles dans la colonne numérique. Nous pouvons utiliser différentes méthodes d'encodage, comme l'encodage par étiquette ou l'encodage à chaud. pandas et sklearn fournissent différentes fonctions d'encodage. Dans notre cas, nous utiliserons la fonction LabelEncoding de sklearn pour encoder la colonne du sexe.

EDA 1

```
import pandas as pd
import numpy as np
# read datasdet using pandas
df = pd.read_csv('employees.csv')
df.head()

df.shape

df.describe()

df.info()

# convert "Start Date" column to datetime data type
df['Start Date'] = pd.to_datetime(df['Start Date'])

df.nunique()

df.isnull().sum()

df["Gender"].fillna("Unkown", inplace = True)

df.isnull().sum()

mode = df['Senior Management'].mode().values[0]
df['Senior Management']= df['Senior Management'].replace(np.nan, mode)

df.isnull().sum()

df = df.dropna(axis = 0, how ='any')

print(df.isnull().sum())
df.shape
```

```
from sklearn.preprocessing import LabelEncoder
# create an instance of LabelEncoder
le = LabelEncoder()

# fit and transform the "Senior Management"
# column with LabelEncoder
df['Gender'] = le.fit_transform\
                    (df['Gender'])

# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

  # histogram plot
sns.histplot(x='Salary', data=df, )
plt.show()


 # box plot Bivaraite analysis
sns.boxplot( x="Salary", y='Team', data=df, )
plt.show()

# scatter plot bivaraite analysis
sns.scatterplot( x="Salary", y='Team', data=df,
                hue='Gender', size='Bonus %')

# Placing Legend outside the Figure
plt.legend(bbox_to_anchor=(1, 1), loc=2)

plt.show()
#multivariate analysis

sns.pairplot(df, hue='Gender', height=2)
```

Traitement des valeurs aberrantes (Outliers)

Une valeur aberrante est un élément/objet de données qui s'écarte de manière significative du reste des objets (dits normaux). Elles peuvent être dues à des erreurs de mesure ou d'exécution. L'analyse pour la détection des valeurs aberrantes est appelée extraction de valeurs aberrantes. Il existe de nombreuses façons de détecter les valeurs aberrantes, et le processus de suppression de ces valeurs aberrantes de la dataframe est identique à la suppression d'un élément de données de la dataframe du panda.

EDA Outliers

```
# importing packages
import seaborn as sns
```

```
import matplotlib.pyplot as plt

# Load the dataset
df = pd.read_csv('Iris.csv')

sns.boxplot(x='SepalWidthCm', data=df)

 IQR
Q1 = np.percentile(df['SepalWidthCm'], 25,
                   interpolation = 'midpoint')

Q3 = np.percentile(df['SepalWidthCm'], 75,
                   interpolation = 'midpoint')
IQR = Q3 - Q1

print("Old Shape: ", df.shape)

# Upper bound
upper = np.where(df['SepalWidthCm'] >= (Q3+1.5*IQR))

# Lower bound
lower = np.where(df['SepalWidthCm'] <= (Q1-1.5*IQR))

# Removing the Outliers
df.drop(upper[0], inplace = True)
df.drop(lower[0], inplace = True)

print("New Shape: ", df.shape)

sns.boxplot(x='SepalWidthCm', data=df)
```

10.4.2 Etude de cas 2

https://www.geeksforgeeks.org/exploratory-data-analysis-on-iris-dataset/

Bibliographie

https ://openclassrooms.com/fr/courses/4302126-decouvrez-la-programmation-orientee-objet-avec-python
https ://www.javatpoint.com/python-oops-concepts
https ://www.edureka.co/blog/data-structures-in-python/#linkedlist
https ://www.geeksforgeeks.org/functional-programming-in-python/
https ://www.javatpoint.com/user-defined-data-structures-in-python
https ://www.educative.io/answers/how-to-implement-a-queue-using-a-linked-list
https ://www.geeksforgeeks.org/what-is-exploratory-data-analysis/