A Comprehensive Guide on **PYTHON**

By Panuganti Sai Likhith

TABLE OF CONTENTS…

Sequences
- Strings
- Lists
- Tuples

Dictionaries

Miscellaneous
- Count and Index
- Split and Join

Iterations
- For Loop
- Nested For Loop
- While Loop

def() Function

Problems on def() function

File Handling
- Renaming a file
- Opening a file
- Reading data from a file
- Writing data to a file

Introduction to .csv format

Introduction to Object Oriented Programming
- Class and objects
- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

Introduction to Searching
- Sequential Search
- binary search

Introduction to Sorting
- Insertion sort
- Selection sort
- Bubble sort

Linked lists
- Single linked list
- Double linked list
- Circular linked list

Introduction to Stacks
- Implementation of Stacks using Arrays
- Implementation of Stacks using Linked Lists

Introduction to Queues
- Implementation of Queue using Arrays
- Implementation of Queue using Linked Lists

Priority Queue

Double-Ended Queues

Introduction to Trees

Tree traversals

Binary Search Tree (BST)

Introduction to NumPy
- Installing NumPy
- NumPy Arrays
- Array Creation
- Array Attributes
- Array Indexing
- Array Slicing
- Array Reshaping
- Array Operations
- Broadcasting
- Aggregation Functions
- Array Manipulation
- Universal Functions (ufunc)
- File Input/Output

The material here includes a variety of Python basic scripts that I developed myself and gathered from different sources.

PYTHON

Python is a high-level, interpreted programming language that was first released in 1991 by Guido van Rossum. It is an open-source, general-purpose language that is designed to be easy to read, write and maintain.
Python's syntax is simple and easy to learn, making it an excellent language for beginners. Its code is often described as being very readable and easy to understand. Python supports a variety of programming paradigms, including object-oriented, functional, and procedural styles. It also has a large standard library that provides a wide range of tools and functionality, making it easy to accomplish complex tasks.

Python is an interpreted language, which means that it does not need to be compiled before it can be run. Instead, it is interpreted at runtime by an interpreter, which executes the code line by line. This makes it easy to write and test code quickly without having to wait for compilation time. Additionally, Python is a cross-platform language, which means that it can run on a wide range of operating systems, including Windows, macOS, and Linux.

Python is a popular programming language for a variety of reasons:

- Easy to Learn: Python has a simple syntax and easy-to-understand structure that makes it a great language for beginners. It's also an interpreted language, which means that you can run code as soon as you write it, without having to compile it first.
- Versatility: Python can be used for a wide range of applications, including web development, data analysis, artificial intelligence, and scientific computing. Its versatility makes it a popular choice for developers in many different fields.
- Large Community: Python has a large and active community of developers who contribute to its development and support. This means that there are many libraries and frameworks available for Python, making it easier to build complex applications.
- Open-Source: Python is an open-source language, which means that it's free to use and can be modified by anyone. This makes it accessible to a wider range of developers and encourages collaboration.
- High Demand: Python is in high demand in the job market, particularly for data science and machine learning roles. Learning Python can help you build a valuable skillset and increase your career opportunities.

Some common use cases for Python include:

Web Development, Data Science, Scientific Computing, Automation, Artificial Intelligence, and many more

Getting Started

In Python, comments are used to explain or clarify the code. A comment starts with the hash symbol (#) and continues until the end of the line. Here's an example:

In [1]:
```
# This is a comment
print("Hello, World!")
```

```
Hello, World!
```

The first line starts with a hashtag or a pound sign (#). This is a comment. Comments are used to explain what is happening in the code, without actually executing it. In this case, the comment doesn't affect the execution of the code at all.

The second line is the actual code that is executed. It uses the built-in print() function to output the string "Hello, World!" to the console.

When you run this code, you should see the text "Hello, World!" printed to the console or terminal.

In [2]:
```
print("Welcome to Python")              #Prints the data
print('Sai Likhith')
```

```
Welcome to Python
Sai Likhith
```

This code consists of two lines of code.

The first line of code uses the print() function to output the string "Welcome to Python" to the console. The string is enclosed in double-quotes.

The second line of code uses the print() function to output the string "Sai Likhith" to the console. The string is enclosed in single-quotes.

In [3]:
```
print(2+20)
print(40-29)
```

```
22
11
```

The first line of code uses the print() function to output the result of the expression 2+20 to the console. The expression is evaluated, and the result, 22, is printed.

The second line of code uses the print() function to output the result of the expression 40-29 to the console. The expression is evaluated, and the result, 11, is printed.

In Python, you can use mathematical operators such as +, -, *, and / to perform arithmetic operations on numeric data. The + operator is used to add two numbers, and the - operator is used to subtract one number from another.

In [4]:
```
print('happy',end='')
print('days')
```

```
happydays
```

The first line of code uses the print() function to output the string "happy" to the console, with the optional end argument set to an empty string. The end argument specifies what should be printed after the string. By default, it is a newline character (\n), which moves the cursor to the next line after printing. Setting it to an empty string means that nothing will be printed after the string.

The second line of code uses the print() function to output the string "days" to the console. Since the end argument was not specified, the default value of \n is used, which means that the cursor moves to the next line after printing "days".

Note that there is no space between "happy" and "days", because we set the end argument of the first print() statement to an empty string. If we had set it to a space character instead, like this:

In [5]:
```
print('happy', end=' ')
print('days')
```

```
happy days
```

Data Types

In Python, there are several built-in data types that are used to represent different types of information. The most commonly used data types are:

1. Numbers: This includes integers, floating-point numbers, and complex numbers.

   int: used to represent whole numbers

   float: used to represent floating-point numbers with decimal places

   complex: used to represent numbers with real and imaginary parts

1. Strings: This is used to represent a sequence of characters, enclosed in single or double quotes.

   str: used to represent a string of characters

2. Booleans: This is used to represent True or False values.

   bool: used to represent boolean values

3. Lists: This is used to represent an ordered collection of elements, enclosed in square brackets.

   list: used to represent a list of elements

4. Tuples: This is used to represent an ordered collection of elements, enclosed in parentheses.

   tuple: used to represent a tuple of elements

5. Sets: This is used to represent an unordered collection of unique elements, enclosed in curly braces.

   set: used to represent a set of elements

6. Dictionaries: This is used to represent a collection of key-value pairs, enclosed in curly braces.

   dict: used to represent a dictionary of key-value pairs

7. NoneType: This is a special data type used to represent the absence of a value.

   None: used to represent the absence of a value

In [1]:
```
print(type('Hello'))
print(type("hello"))
print(type(2))
print(type(3.2))
print(type('12'))
print(type(True))
```

```
<class 'str'>
<class 'str'>
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

In [2]:

```
print(type([1,2,3]))
print(type((1,2,3)))
print(type({1,2,3}))
print(type({1:'hello',2:'hi',3:'python'}))
```

```
<class 'list'>
<class 'tuple'>
<class 'set'>
<class 'dict'>
```

Type conversion functions

Type conversion functions, also known as typecasting or type conversion, are functions in programming languages that allow you to convert a value of one data type to another data type.

int() The int function truncates all values after the decimal and prints the integer value.

In [1]:
```
print(3.14, int(3.14))
print(3.9999, int(3.9999))         # This doesn't round to the closest int!
print(3.0, int(3.0))
print(-3.999, int(-3.999))         # Note that the result is closer to zero
print("2345", int("2345"))         # parse a string to produce an int
print(17, int(17))                 # int even works on integers


#print(int("123Likhith"))
# This will raise a ValueError because the string "123Likhith"
#contains non-numeric characters, which cannot be converted to an integer.
```

```
3.14 3
3.9999 3
3.0 3
-3.999 -3
2345 2345
17 17
```

float()

In [2]:
```
print(float("1234.5"))
print(type(float("102345.89")))
```

```
1234.5
<class 'float'>
```

str()

In [3]:
```
print(str(12))
print(str(123.45))
print(type(str(123.455678)))
```

```
12
123.45
<class 'str'>
```

bool()

In [4]:
```
# convert an integer to a boolean
int_num = 0
bool_val = bool(int_num)
print(bool_val)  # output: False

# convert a non-empty string to a boolean
non_empty_str = "hello"
bool_val = bool(non_empty_str)
print(bool_val)  # output: True
```

```
False
True
```

list()

In [5]:
```python
# convert a tuple to a list
my_tuple = (1, 2, 3)
my_list = list(my_tuple)
print(my_list)  # output: [1, 2, 3]

# convert a string to a list of characters
my_str = "hello"
my_list = list(my_str)
print(my_list)  # output: ['h', 'e', 'l', 'l', 'o']
```

```
[1, 2, 3]
['h', 'e', 'l', 'l', 'o']
```

tuple()

In [6]:
```python
# convert a list to a tuple
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
print(my_tuple)  # output: (1, 2, 3)

# convert a string to a tuple of characters
my_str = "hello"
my_tuple = tuple(my_str)
print(my_tuple)  # output: ('h', 'e', 'l', 'l', 'o')
```

```
(1, 2, 3)
('h', 'e', 'l', 'l', 'o')
```

set()

In [7]:
```python
# convert a list to a set
my_list = [1, 2, 3, 2]
my_set = set(my_list)
print(my_set)  # output: {1, 2, 3}

# convert a string to a set of characters
my_str = "hello"
my_set = set(my_str)
print(my_set)  # output: {'h', 'e', 'l', 'o'}
```

```
{1, 2, 3}
{'o', 'h', 'l', 'e'}
```

Variables

In Python, a variable is a name given to a value stored in memory. It's a way to store and access data in a program. To create a variable in Python, you can simply assign a value to a name using the assignment operator =.

1. Starts with a letter
2. Can only contain letter, underscores and Numbers
3. But they cannot start with a digit. Also, Python is case sensitive, so x and X are two different variables.

```
76trombones = "big parade"
more$ = 1000000
class = "Computer Science 101"
```

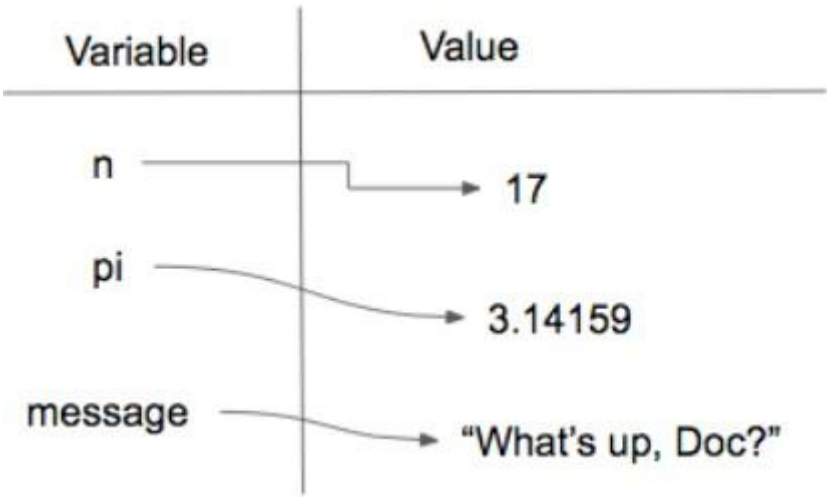
Python reserved keywords cannot be used as Variables

Python Keywords			
False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

In [3]:

```
n2=10
print(n2)
```

```
10
```



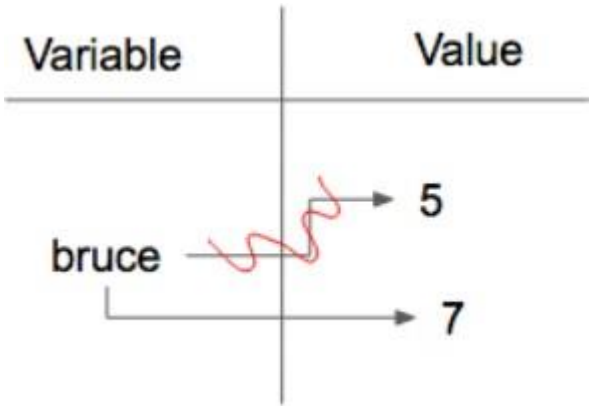
In [4]:
```
message = "What's up, Doc?"
n = 17
pi = 3.14159

print(message)
print(n)
print(pi)
```

```
What's up, Doc?
17
3.14159
```

In [5]:
```
x = 5
y = 2.5
z = "hello"

print(x + y)
print(z + " " + str(x))
```

```
7.5
hello 5
```

In programming, the latest value stored in a variable is simply the value that was most recently assigned to the variable. When a variable is updated with a new value, the previous value is overwritten, and the variable now holds the new value.



In [6]:
```
day = "Thursday"
day = 32.5
day = 19
print(day)
```

```
19
```

In [7]:
```
x = 5
print(x)

x = x + 1
print(x)

x = "Hello, world!"
print(x)
```

```
5
6
Hello, world!
```

In [8]:
```
x = 15
y = x
x = 22
print(x,y)
```

```
22 15
```

Operators

Python has several operators, including arithmetic operators, comparison operators, and logical operators. Here are some examples:

In [9]:
```
# Arithmetic operators
x = 5 + 3
y = 7 - 2
z = 3 * 4
a = 10 / 2

# Comparison operators
b = 5 > 3
c = 7 <= 2
d = 3 == 3
e = 10 != 2
```

Arithmetic operators

In Python, arithmetic operators are used to perform mathematical operations on numerical data types. Here are the basic arithmetic operators:

```
+ Addition operator: adds two values

- Subtraction operator: subtracts one value from another

* Multiplication operator: multiplies two values

/ Division operator: divides one value by another and returns a
floating-point number

// Integer division operator: divides one value by another and
returns the quotient as an integer

% Modulus operator: returns the remainder of division

** Exponentiation operator: raises one value to the power of
another
```

In [10]:
```
# Addition operator
a = 10
b = 20
c = a + b
print(c)  # Output: 30

# Subtraction operator
a = 10
b = 20
c = b - a
print(c)  # Output: 10

# Multiplication operator
a = 5
b = 6
c = a * b
```

```
print(c)  # Output: 30

# Division operator
a = 10
b = 3
c = a / b
print(c)  # Output: 3.3333333333333335

# Integer division operator
a = 10
b = 3
c = a // b
print(c)  # Output: 3

# Modulus operator
a = 10
b = 3
c = a % b
print(c)  # Output: 1

# Exponentiation operator
a = 2
b = 3
c = a ** b
print(c)  # Output: 8
```

```
30
10
30
3.3333333333333335
3
1
8
```

Note that the division operator / returns a floating-point number, even if the operands are integers. If you want to perform integer division and get the quotient as an integer, you can use the integer division operator //. The modulus operator % returns the remainder of division. The exponentiation operator ** raises one value to the power of another.

Logical Operators

In Python, logical operators are used to combine or negate Boolean expressions. There are three logical operators in Python:

1. and: This operator returns True if both expressions on the left and right side of the operator are True.
2. or: This operator returns True if at least one of the expressions on the left and right side of the operator is True.
3. not: This operator returns the opposite of the Boolean expression it is applied to. If the expression is True, not returns False, and if the expression is False, not returns True.

In [11]:
```
x = 5
y = 6
z = 7
print(x < y and y < z)
```

```
True
```

In [12]:
```
x = 5
y = 6
z = 7
print(x > y or y < z)
```

```
True
```

In [13]:
```
x = 5
print(not x == 5)
```

```
False
```

In [14]:
```
x = 5
y = 10
z = 15

# Using the and operator
if x < y and y < z:
    print("x is less than y and y is less than z")

# Using the or operator
if x > y or y < z:
    print("At least one of the conditions is true")

# Using the not operator
if not x == y:
    print("x is not equal to y")
```

```
x is less than y and y is less than z
At least one of the conditions is true
x is not equal to y
```

in and not Operators

The in operator and the not in operator are used in Python to check if a value is a member of a sequence, such as a string, list, or tuple.

The in operator returns True if the value is in the sequence, and False otherwise. The not in operator returns True if the value is not in the sequence, and False otherwise.

In [15]:
```
print(1 in [1,2,3])
10 in [1,2,3]
```

Out[15]:
```
True
False
```

In [16]:
```
print('p' in 'apple')
print('i' in 'apple')
print('ap' in 'apple')
print('pa' in 'apple')
print('' in 'a')
print(' ' in 'apple')
```

```
True
False
True
False
True
False
```

In [17]:
```
print('x' not in 'apple')
```

```
True
```

In [18]:
```
print('wow' not in ['gee wiz', 'gosh golly', 'wow', 'amazing'])
```

```
False
```

In [19]:
```
# Using the in operator with a string
my_string = "Hello, World!"
if "World" in my_string:
    print("World is in the string")  # This will print

# Using the not in operator with a list
my_list = [1, 2, 3, 4, 5]
if 6 not in my_list:
    print("6 is not in the list")  # This will print
```

```
World is in the string
6 is not in the list
```

Precedence of Operators

Operator precedence is the order in which operators are evaluated in an expression. In Python, operator precedence follows the standard mathematical conventions. Operators with higher precedence are evaluated first. If operators have the same precedence, they are evaluated from left to right.

Here is a table of the operator precedence in Python, from highest to lowest:

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x, -x`	Positive, negative
`*, /, //, %`	Multiplication, division, floor division, modulus
`+, -`	Addition, subtraction
`==, !=, <, <=, >, >=, is, is not, in, not in`	Comparison, identity, and membership tests
`not`	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR

For example, in the expression 2 + 3 *4, the multiplication ()* operator has a higher precedence than the addition (+) operator, so 3 * 4 is evaluated first, resulting in 12. Then the addition operation is performed on 2 and 12, resulting in 14.

If you want to change the order of evaluation, you can use parentheses (()) to group expressions. For example, in the expression (2 + 3) * 4, the parentheses force the addition to be performed before the multiplication, resulting in 20.

Boolean Expressions

In Python, Boolean expressions are used to evaluate whether a certain condition is true or false. These expressions are usually used in control flow statements such as if, while, and for loops, as well as in boolean operations.

Here are some common boolean operators used in Python:

== : Equal to

!= : Not equal to

< : Less than

<= : Less than or equal to

*>: Greater than

*>= : Greater than or equal to

In [20]:
```
print(type(True))
print(type(False))
print(type('False'))
```

```
<class 'bool'>
<class 'bool'>
<class 'str'>
```

In [21]:
```
print(9==0)
print(9!=0)
print(6<9)
print(4<=4)
print(88>9)
print(4>=94)
```

```
False
True
True
True
True
False
```

Statements and Expressions

A statement is an instruction that the Python interpreter can execute. You have only seen the assignment statement so far. Some other kinds of statements that you'll see in future chapters are while statements, for statements, if statements, and import statements. (There are other kinds too!)

An expression is a combination of literals, variable names, operators, and calls to functions. Expressions need to be evaluated. The result of evaluating an expression is a value or object.

In [22]:
```
print(1 + 1 + (2 * 3))
```

8

If you ask Python to print an expression, the interpreter evaluates the expression and displays the result.

In [23]:
```
print(len("Python"))
```

6

In this example len is a built-in Python function that returns the number of characters in a string.

The evaluation of an expression produces a value, which is why expressions can appear on the right hand side of assignment statements. A literal all by itself is a simple expression, and so is a variable.

Hard-Coding

Hard-coding in Python refers to the practice of explicitly defining values, constants, and other data directly in the source code, rather than reading them from external sources such as files, databases or user input.

In [25]:
```
pi = 3.14
radius = 5
area = pi * (radius ** 2)
print("The area of the circle is:", area)
```

```
The area of the circle is: 78.5
```

pi = 3.14 radius = 5 area = pi * (radius ** 2) print("The area of the circle is:", area)

User Input

In Python, to take input from the user, you can use the input() function. The input() function waits for the user to type something and press the Enter key.

In [26]:
```
name = input("What is your name? ")
print("Hello",name)
```

```
What is your name? Likhith
Hello Likhith
```

In the above example, the input() function will display the message "What is your name?" to the user, and then wait for the user to enter their name. Whatever the user types will be stored in the variable name. The print() function will then display the message "Hello, [name]!", where [name] is replaced by the value of the name variable.

Note that the input() function returns a string, so if you want to use the input as a number, you will need to convert it using the appropriate data type, such as int() or float().

In [27]:
```
n = input("Please enter your age: ")  #All input from users is read in as a string

print(type(n))
```

```
Please enter your age: 21
<class 'str'>
```

In [28]:
```
str_seconds = input("Enter the no. of seconds to convert : ")
total_secs = int(str_seconds)

hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes =  secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining  % 60

print("Hrs=", hours, "mins=", minutes, "secs=", secs_finally_remaining)
```

```
Enter the no. of seconds to convert : 7900
Hrs= 2 mins= 11 secs= 40
```

Conditional Execution

Conditional execution allows us to say, "Run this piece of code if something is true, run this piece of code if something else is true, or run this piece of code if something is false". So, let's jump right into conditional execution. So, we do conditional execution with if and else statements.

Conditional execution in Python allows you to run specific code only if a certain condition is met. The most common way to do this is with an if statement. This is sometimes referred to as binary selection since there are two possible paths of execution.

In [29]:
```
x = 5
if x > 0:
    print("x is positive")
```

```
x is positive
```

In [30]:
```
x = 10
if x < 0:
    print("The negative number ",  x, " is not valid here.")
print("This is always printed")
```

```
This is always printed
```

```
if BOOLEAN EXPRESSION:
    STATEMENTS_1        # executed if condition evaluates to True
else:
    STATEMENTS_2        # executed if condition evaluates to False
```

In [31]:
```
x = 15

if x % 2 == 0:
    print(x, "is even")
else:
    print(x, "is odd")
```

```
15 is odd
```

Boolean expressions conditions: In Python, any non-zero value is considered True and any zero value is considered False. This means you can use boolean expressions in conditional statements to test whether a value is truthy or falsy. For example:

In [32]:
```
x = 10
if x:
    print("x is truthy")
else:
    print("x is falsy")
```

```
x is truthy
```

In Python, conditional execution allows you to control the flow of your program based on certain conditions. This is typically achieved using the if, elif (short for "else if"), and else statements.

Here is an example of how elif conditional execution works in Python:

In [33]:
```
x = 10
y = 5

if x > y:
    print("x is greater than y")
elif x < y:
    print("y is greater than x")
else:
    print("x and y are equal")
```

```
x is greater than y
```

In this example, the program first checks whether x is greater than y. If it is, the program prints "x is greater than y". If it is not, the program moves on to the elif statement, which checks whether x is less than y. If it is, the program prints "y is greater than x". If neither condition is true, the program moves on to the else statement and prints "x and y are equal".

In [34]:
```
x = 10
y = 5
z = 7

if x > y and x > z:
    print("x is the largest")
elif y > x and y > z:
    print("y is the largest")
else:
    print("z is the largest")
```

```
x is the largest
```

In this example, the program checks whether x is greater than both y and z. If it is, the program prints "x is the largest". If not, the program moves on to the next condition and checks whether y is greater than both x and z. If it is, the program prints "y is the largest". If neither condition is true, the program prints "z is the largest".

Nested conditional statements: You can use if statements inside other if statements to create more complex conditions. For example:

In [35]:
```
x = 10
y = 5
z = 7

if x > y:
    if x > z:
        print("x is the largest")
    else:
        print("z is the largest")
elif y > z:
    print("y is the largest")
else:
    print("z is the largest")
```

```
x is the largest
```

In [36]:
```
x = 10
y = 10

if x < y:
```

```
    print("x is less than y")
else:
    if x > y:
        print("x is greater than y")
    else:
        print("x and y must be equal")
```

```
x and y must be equal
```

Chained conditions in Python are multiple conditions that are evaluated in sequence until one of them is found to be true. Chained conditions are commonly used when you want to test multiple conditions in order, and perform different actions based on which condition is true.

In Python, you can chain conditions using the and and or operators. Here's an example of how you can use chained conditions in Python:

In [37]:
```
x = 5
y = 10

if x > 0 and y > 0:
    print("Both x and y are positive.")
elif x > 0 or y > 0:
    print("At least one of x or y is positive.")
else:
    print("Neither x nor y is positive.")
```

```
Both x and y are positive.
```

In this example, the program first checks whether both x and y are greater than 0 using the and operator. If both conditions are true, the program prints "Both x and y are positive." If the first condition is not true, the program moves on to the next condition using the elif statement, which checks whether either x or y is greater than 0 using the or operator. If either condition is true, the program prints "At least one of x or y is positive." If neither of these conditions is true, the program moves on to the else statement and prints "Neither x nor y is positive."

You can chain together as many conditions as you need, but be careful not to make the conditions too complex, as this can make your code hard to read and understand. It's often a good idea to break complex conditions into simpler ones, and to use parentheses to group related conditions together.

Make a Flow Chart before You Write Your Code

Before writing your conditionals, it can be helpful to make your own flowchart that will plot out the flow of each condition. By writing out the flow, you can better determine how complex the set of conditionals will be as well as check to see if any condition is not taken care of before you begin writing it out.

Sequences

A sequence is an ordered collection, meaning that it's a collection of items and it has an order. Meaning that it has the first item, a second item, a third item and so on. It also has a length.

In Python, a sequence is an ordered collection of elements, where each element is identified by an index. There are three basic types of sequences in Python:

1. Strings
2. Lists
3. Tuples

Strings

A string is an immutable sequence of characters. Strings are created with quotes, either single ' or double ".

All sequences share some common characteristics:

- They can be indexed and sliced.
- They can be iterated over with loops.
- They can be concatenated with the + operator.
- They can be repeated with the * operator.

Multi line string using '''. .. '''

String is same using "...." or '...'

In [38]:
```
s='Hello World'
print(s)
print(type(s))

print('\n')

s='12445'
print(s)
print(type(s))
```

```
Hello World
<class 'str'>


12445
<class 'str'>
```

In [39]:
```
multi_line='''Hello !
World
How is it going ?'''
print(multi_line)
```

```
Hello !
World
How is it going ?
```

In [40]:
```
print(s)     #gives as 'Hello World' as we already defined above
```

```
12445
```

why "....." and '....' , and where we can use

In [41]:
```
print('This is Likhith's Colab NoteBook')
```

```
  File "<ipython-input-41-88093d7e1946>", line 1
    print('This is Likhith's Colab NoteBook')
                                             ^
SyntaxError: unterminated string literal (detected at line 1)
```

In [42]:
```
print("This is Likhith's Colab NoteBook")
```

```
This is Likhith's Colab NoteBook
```

In [43]:
```
print('She said 'HI'')
```

```
  File "<ipython-input-43-ef940407e644>", line 1
    print('She said 'HI'')
            ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

In [44]:
```
print('She said "HI"')
```

```
She said "HI"
```

Concatenation: You can concatenate two strings using the '+' operator:

In [45]:
```
str1 = "Hello"
str2 = "world"
str3 = str1 + " " + str2
print(str3)
```

```
Hello world
```

Indexing: You can access individual characters of a string using their index:

In [46]:
```
my_string = "Hello, world!"
print(my_string[0])
print(my_string[4])
print(my_string[7])
```

```
H
o
w
```

Slicing: You can extract a portion of a string using slicing. The syntax is string[start:stop:step], where start is the index of the first character to include, stop is the index of the first character to exclude, and step is the step size:

Note that Python indexes strings starting from 0.

In [47]:
```
my_string = "Hello, world!"
print(my_string[0:5])
print(my_string[7:])
print(my_string[::2])
```

```
Hello
world!
Hlo ol!
```

Length: You can find the length of a string using the len() function:

In [48]:
```
string='Python'
print(len(string))  # len Gives length of the string
# If 6 is the length of the string, the 0 to 5 is the index ranging
```

```
6
```

String methods: Python provides many built-in string methods that you can use to manipulate strings. Here are a few examples:

- upper() and lower(): Convert a string to all uppercase or all lowercase.

In [49]:
```
my_string = "Hello, world!"
print(my_string.upper()) # converts given string into Upper case
print(my_string.lower()) # converts given string into Lower case
```

```
HELLO, WORLD!
hello, world!
```

- strip(): Remove whitespace from the beginning and end of a string.

In [50]:
```
my_string = "   Hello, world!   "
print(my_string.strip()) # Output: "Hello, world!"
```

```
Hello, world!
```

- replace(): Replace all occurrences of a substring with another substring.

In [51]:
```
my_string = "Hello, world!"
print(my_string.replace("world", "Python")) # Output: "Hello, Python!"
```

```
Hello, Python!
```

String formatting: String formatting allows you to insert values into a string in a flexible way. There are several ways to format strings in Python, but the most commonly used method is using the % operator:

In [52]:
```
name = "Alice"
age = 25
print("My name is %s and I am %d years old." % (name, age))
# Output: "My name is Alice and I am 25 years old."

#Using format function
print("My name is {} and I am {} years old.".format(name, age))
# Output: "My name is Alice and I am 25 years old."
```

```
My name is Alice and I am 25 years old.
My name is Alice and I am 25 years old.
```

String comparison: In Python, you can compare two strings using the <, >, <=, >=, ==, and != operators. Strings are compared based on their lexicographical order (i.e., their order in the dictionary). Here's an example:

In [53]:
```
str1 = "Python"
str2 = "Hello"
print(str1 < str2) # Output: False
```

```
False
```

Method	Parameters	Description
upper	none	Returns a string in all uppercase
lower	none	Returns a string in all lowercase
count	item	Returns the number of occurrences of item
index	item	Returns the leftmost index where the substring item is found and causes a runtime error if item is not found
strip	none	Returns a string with the leading and trailing whitespace removed
replace	old, new	Replaces all occurrences of old substring with new
format	substitutions	Involved! See String Format Method, below

In [54]:
```
s = input("Enter some text")
ac = ""
for c in s:
    ac = ac + c + "-" + c + "-"

print(ac)
```

```
Enter some textPython
P-P-y-y-t-t-h-h-o-o-n-n-
```

Lists

Lists are a fundamental data structure in Python, and they are used to store a collection of items in an ordered sequence, where each value is identified by an index. Lists are mutable, you can change them, you can reassign the contents at any position to be a different value.

The list is a sequence data type which is used to store the collection of data, a list is a collection of things, enclosed in [ ] and separated by commas.

A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

Creating a list: You can create a list by enclosing a comma-separated sequence of items inside square brackets [].

In [55]:
```
my_list = [1, 2, 3, 4, 5]
```

Accessing list items: You can access individual items in a list using their index, which starts at 0.

In [56]:
```
print(my_list[0])  # Output: 1
print(my_list[2])  # Output: 3
```

```
1
3
```

Slicing a list: You can extract a portion of a list by specifying a range of indexes. The resulting list will contain all the items from the starting index up to but not including the ending index.

In [57]:
```
print(my_list[1:4])  # Output: [2, 3, 4]
```

```
[2, 3, 4]
```

Modifying list items: You can modify the value of an item in a list by accessing it and assigning a new value.

In [58]:
```
my_list[2] = 6
print(my_list)  # Output: [1, 2, 6, 4, 5]
```

```
[1, 2, 6, 4, 5]
```

In [63]:
```
my_list[4] = 'replaced'
print(my_list)  # Output: [1, 2, 6, 4,'replaced']
```

```
[1, 2, 99, 3, 'replaced', 5]
```

Adding items to a list: You can add new items to a list using the append() method, which adds the item to the end of the list.

In [60]:
```
my_list.append(9)
print(my_list)
```

```
[1, 2, 6, 4, 'replaced', 9]
```

Inserting an item into a list: You can insert a new item into a list at a specified index using the insert() method.

In [61]:
```
my_list = [1, 2, 3, 4, 5]
my_list.insert(2, 99)
print(my_list)  # Output: [1, 2, 99, 3, 4, 5]
```

```
[1, 2, 99, 3, 4, 5]
```

Removing items from a list: You can remove items from a list using the remove() method, which removes the first occurrence of the specified item.

In [64]:
```
my_list.remove('replaced')
print(my_list)  # Output: [1, 2, 4, 5, 6]
```

```
[1, 2, 99, 3, 5]
```

List Element Deletion: The del statement removes an element from a list by using its position.

In [65]:
```
a = ['one', 'two', 'three']
del a[1]
print(a)

alist = ['a', 'b', 'c', 'd', 'e', 'f']
del alist[1:5]
print(alist)
```

```
['one', 'three']
['a', 'f']
```

Finding the length of a list: You can find the number of items in a list using the len() function.

In [66]:
```
print(len(my_list))  # Output: 5
```

```
5
```

Concatenating lists: You can concatenate two or more lists using the + operator or the extend() method.

In [67]:
```
my_list = [1, 2, 3]
new_list = [4, 5, 6]
concatenated_list = my_list + new_list
print(concatenated_list)
```

```
[1, 2, 3, 4, 5, 6]
```

Copying lists: You can create a copy of a list using the copy() method or by using a slice [:].

In [68]:
```
my_list = [1, 2, 3]
copy_list = my_list.copy()
print(copy_list)
```

```
[1, 2, 3]
```

Iterating over a list: You can iterate over the items in a list using a for loop.

In [69]:
```
my_list = [1, 2, 3, 4, 5]
for item in my_list:
```

```
    print(item)
```

```
1
2
3
4
5
```

Sorting a list: You can sort the items in a list using the sort() method or the sorted() function.

In [70]:
```
my_list = [3, 1, 4, 2, 5]
my_list.sort()  # sorts the list in place
print(my_list)
```

```
[1, 2, 3, 4, 5]
```

Reversing a list: You can reverse the order of items in a list using the reverse() method.

In [71]:
```
my_list = [1, 2, 3, 4, 5]
my_list.reverse()
print(my_list)  # Output: [5, 4, 3, 2, 1]
```

```
[5, 4, 3, 2, 1]
```

Nested lists: A nested list is a list that contains other lists as its elements. You can access the elements of a nested list by using multiple indices.

In [72]:
```
my_list = [[1, 2], [3, 4], [5, 6]]
print(my_list[0][1])  # Output: 2
print(my_list[1][0])  # Output: 3
```

```
2
3
```

Finding the index of an item in a list: You can find the index of a specified item in a list using the index() method.

In [73]:
```
my_list = ['apple', 'banana', 'orange']
print(my_list.index('banana'))  # Output: 1
```

```
1
```

Removing duplicates from a list: You can remove duplicates from a list by converting it to a set and then back to a list.

In [74]:
```
my_list = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
unique_list = list(set(my_list))
print(unique_list)  # Output: [1, 2, 3, 4]
```

```
[1, 2, 3, 4]
```

pop() mutates the list by deteting or removing the last element from the list

In [75]:
```
my_list.pop()
my_list
```

Out[75]:
```
[1, 2, 2, 3, 3, 3, 4, 4, 4]
```

List comprehension: List comprehension is a concise way to create lists in Python. It allows you to create a new list by iterating over an existing list and applying some transformation

or filtering.

In [76]:
```
my_list = [1, 2, 3, 4, 5]
squared_list = [item**2 for item in my_list]
even_list = [item for item in my_list if item % 2 == 0]
```

Multidimensional lists: A multidimensional list is a list that contains other lists as its elements, and those lists may also contain other lists as their elements, and so on. You can access the elements of a multidimensional list by using multiple indices.

In [77]:
```
my_list = [[1, 2], [3, [4, 5]], [6, 7, 8]]
print(my_list[1][1][0])  # Output: 4
```

```
4
```

The above example accesses the second item in the second list in the main list, which is itself a list containing two items. It then accesses the first item in that inner list.

Method	Parameters	Result	Description
append	item	mutator	Adds a new item to the end of a list
insert	position, item	mutator	Inserts a new item at the position given
pop	none	hybrid	Removes and returns the last item
pop	position	hybrid	Removes and returns the item at position
sort	none	mutator	Modifies a list to be sorted
reverse	none	mutator	Modifies a list to be in reverse order
index	item	return idx	Returns the position of first occurrence of item
count	item	return ct	Returns the number of occurrences of item
remove	item	mutator	Removes the first occurrence of item

In [78]:
```
mylist = []
mylist.append(5)
mylist.append(27)
mylist.append(3)
mylist.append(12)
print(mylist)

mylist.insert(1, 12)
print(mylist)
print(mylist.count(12))

print(mylist.index(3))
print(mylist.count(5))

mylist.reverse()
print(mylist)

mylist.sort()
print(mylist)

mylist.remove(5)
```

```
print(mylist)

lastitem = mylist.pop()
print(lastitem)
print(mylist)
```

```
[5, 27, 3, 12]
[5, 12, 27, 3, 12]
2
3
1
[12, 3, 27, 12, 5]
[3, 5, 12, 12, 27]
[3, 12, 12, 27]
27
[3, 12, 12]
```

In [79]:

```
# Accumulating

nums = [3, 5, 8]
accum = []
for w in nums:
    x = w**2
    accum.append(x)
print(accum)
```

```
[9, 25, 64]
```

Tuples

A tuple, like a list, is a sequence of items of any type. The printed representation of a tuple is a comma-separated sequence of values, enclosed in parentheses. In other words, the representation is just like lists, except with parentheses () instead of square brackets [].

A tuple is similar to a list in many ways, but it is immutable, which means that once it is created, its elements cannot be modified.

To create a tuple, you can use parentheses () or the tuple() function. Here's an example:

In [80]:
```
my_tuple = (1, 2, 3, "four")
print(my_tuple)
```

```
(1, 2, 3, 'four')
```

In [81]:
```
t = (5,)
print(type(t))

x = (5)
print(type(x))
```

```
<class 'tuple'>
<class 'int'>
```

Tuples are useful when you need to group related values together, but you want to ensure that they cannot be modified. They are also slightly faster than lists, and they use less memory.

You can access the elements of a tuple using indexing, just like you would with a list. For example:

In [82]:
```
print(my_tuple[0])    # prints 1
print(my_tuple[3])    # prints "four"
```

```
1
four
```

You can also use slicing to access a subset of the elements in a tuple:

In [83]:
```
print(my_tuple[1:3])    # prints (2, 3)
```

```
(2, 3)
```

Finally, you can use the len() function to get the length of a tuple:

In [84]:
```
print(len(my_tuple))    # prints 4
```

```
4
```

Concatenating tuples: You can concatenate two tuples using the + operator. For example:

In [85]:
```
tuple1 = (1, 2, 3)
tuple2 = ("four", "five", "six")
concatenated_tuple = tuple1 + tuple2
```

Tuple methods: Tuples have a few built-in methods that you can use to manipulate them. For example:

- count(): This method returns the number of times a given element appears in a tuple.

In [86]:
```
my_tuple = (1, 2, 2, 3, 3, 3)
print(my_tuple.count(2))    # prints 2
print(my_tuple.count(3))    # prints 3
```

```
2
3
```

- index(): This method returns the index of the first occurrence of a given element in a tuple.

In [87]:
```
my_tuple = (1, 2, 2, 3, 3, 3)
print(my_tuple.index(2))    # prints 1
print(my_tuple.index(3))    # prints 3
```

```
1
3
```

Tuple comprehension: Tuples don't have a built-in comprehension syntax like lists and dictionaries do, but you can use a generator expression to create a tuple. For example:

In [88]:
```
my_tuple = tuple(x**2 for x in range(5))
#This creates a tuple that contains the squares of the numbers 0 through 4.
print(my_tuple)
```

```
(0, 1, 4, 9, 16)
```

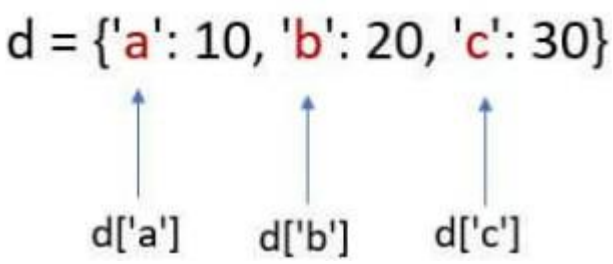
Dictionaries

In Python, a dictionary is a built-in data structure that is used to store and retrieve data in akey-value format. It is an unordered collection of elements, where each element is represented as a key-value pair.

The keys of a dictionary must be unique, immutable (cannot be changed), and hashable (canbe used as a key in a dictionary). The values can be of any data type, including integers, floats, strings, lists, tuples, and even other dictionaries.

Dictionaries can be created using curly braces {} or the dict() constructor. To add a key-valuepair to a dictionary, simply use square bracket notation dict[key] = value. To retrieve a valuefrom a dictionary, use the key as the index dict[key].

Dictionaries can also be updated, deleted, and searched using various built-in methods. Forexample, the update() method can be used to add or update key-value pairs in a dictionary,while the pop() method can be used to delete a key-value pair. The keys() method returns alist of all the keys in the dictionary, while the values() method returns a list of all the values.



Creating a Dictionary: In Python, dictionaries are created using curly braces {} or the dict()constructor. Here's an example:

In [1]:
```
# Create a dictionary using curly braces
my_dict = {'apple': 1, 'banana': 2, 'orange': 3}

# Create a dictionary using dict()
my_dict = dict(apple=1, banana=2, orange=3)
```

Adding or Updating Key-Value Pairs: To add or update a key-value pair in a dictionary,simply use square bracket notation dict[key] = value. Here's an example:

In [2]:
```
# Add a new key-value pair
my_dict['grape'] = 4

# Update a key-value pair
my_dict['banana'] = 5
```

Deleting a Key-Value Pair: To delete a key-value pair from a dictionary, use the delstatement. Here's an example:

In [3]:
```
# Delete a key-value pair
del my_dict['orange']
```

Accessing Values: To access a value in a dictionary, use the key as the index dict[key]. Here's an example:

In [4]:
```
# Access a value
print(my_dict['banana'])  # Output: 5
```

```
5
```

Getting Keys or Values: To get a list of all the keys or values in a dictionary, use the keys() or values() method. Here's an example:

In [5]:
```
# Get a list of keys
print(my_dict.keys())  # Output: ['apple', 'banana', 'grape']

# Get a list of values
print(my_dict.values())  # Output: [1, 5, 4]
```

```
dict_keys(['apple', 'banana', 'grape'])
dict_values([1, 5, 4])
```

Looping Through a Dictionary: To loop through a dictionary, you can use a for loop. There are three different methods you can use: keys(), values(), or items(). Here's an example:

In [6]:
```
# Loop through keys
for key in my_dict.keys():
    print(key)

# Loop through values
for value in my_dict.values():
    print(value)

# Loop through key-value pairs
for key, value in my_dict.items():
    print(key, value)
```

```
apple
banana
grape
1
5
4
apple 1
banana 5
grape 4
```

Updating a Dictionary: To update a dictionary with another dictionary, use the update()method. Here's an example:

In [7]:
```
# Update a dictionary
other_dict = {'pear': 6, 'peach': 7}
my_dict.update(other_dict)
```

Clearing a Dictionary: To remove all items from a dictionary, use the clear() method. Here'san example:

In [8]:
```
# Clear a dictionary
my_dict.clear()
```

Checking for the Existence of a Key: To check if a key exists in a dictionary, use the inkeyword. Here's an example:

In [9]:
```
# Check if a key exists
if 'apple' in my_dict:
    print('Apple is in the dictionary')
```

Getting a Default Value: To get a default value if a key doesn't exist in a dictionary, use theget() method. Here's an example:

In [10]:
```
# Get a default value
print(my_dict.get('pear', 0))  # Output: 0
```

```
0
```

Copying a Dictionary: To create a copy of a dictionary, use the copy() method. Here's anexample:

In [11]:
```
# Copy a dictionary
new_dict = my_dict.copy()
```

Merging Two Dictionaries: you can merge two dictionaries in Python using the update()method. Here's an example:

In [12]:
```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict1.update(dict2)
print(dict1)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

In this example, dict1 and dict2 are two dictionaries that we want to merge. We use the update() method of the dict class to merge dict2 into dict1. The update() method adds all the key-value pairs from dict2 to dict1. If there are any common keys, the values from dict2 overwrite the values in dict1.

You can also use the double asterisk (**) operator to merge two dictionaries. Here's an example:

In [13]:
```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = {**dict1, **dict2}
print(dict3)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

In this example, we use the double asterisk **() operator to merge dict1 and dict2 into dict3. The** operator unpacks the key-value pairs of the dictionaries and creates a new dictionary with all the key-value pairs.

miscellaneous

Count and Index

In [89]:
```
a = "I have had an apple on my desk before!"
print(a.count("e"))
print(a.count("ha"))
```

```
5
2
```

In [90]:
```
qu = "wow, welcome week! Were you wanting to go?"
ty = qu.count("we")
```

In [91]:
```
z = ['atoms', 4, 'neutron', 6, 'proton', 4, 'electron', 4, 'electron', 'atoms']
print(z.count("4"))
print(z.count(4))
print(z.count("a"))
print(z.count("electron"))
```

```
0
3
0
2
```

In [92]:
```
music = "Pull out your music and dancing can begin"
bio = ["Metatarsal", "Metatarsal", "Fibula", [], "Tibia", "Tibia", 43, "Femur", "O

print(music.index("m"))
print(music.index("your"))

print(bio.index("Metatarsal"))
print(bio.index([]))
print(bio.index(43))
```

```
14
9
0
3
6
```

Split and Join

Splitting and joining are two other useful methods on strings in lists. So, split takes a string and turns it into a list of substrings of that string.

In [93]:
```
song = "The rain in Spain..."
wds = song.split()
print(wds)
```

```
['The', 'rain', 'in', 'Spain...']
```

An optional argument called a delimiter can be used to specify which characters to use as word boundaries.

In [94]:
```
#The following example uses the string ai as the delimiter:
song = "The rain in Spain..."
wds = song.split('ai')
print(wds)
```

```
['The r', 'n in Sp', 'n...']
```

The inverse of the split method is join. You choose a desired separator string, (often called the glue) and join the list with the glue between each of the elements.

In [95]:
```
wds = ["red", "blue", "green"]
glue = ';'
s = glue.join(wds)
print(s)
print(wds)

print("***".join(wds))
print("".join(wds))
```

```
red;blue;green
['red', 'blue', 'green']
red***blue***green
redbluegreen
```

Iterations

Iteration runs a piece of code for every item in a sequence, be it a string, list or a tuple.

i.e.,Iteration in Python refers to the process of repeatedly executing a block of code, typically with a loop, until a certain condition is met. There are several types of loops in Python that allow you to iterate over different data types, including lists, tuples, dictionaries, and strings.

1. for loop: A for loop allows you to iterate over a sequence of elements, such as a list or a tuple.
2. while loop: A while loop allows you to iterate until a certain condition is met.
3. range() function: The range() function is a built-in function in Python that generates a sequence of numbers. You can use this function in conjunction with a for loop to iterate a specific number of times.
4. range() function: The range() function is a built-in function in Python that generates a sequence of numbers. You can use this function in conjunction with a for loop to iterate a specific number of times.

For Loop

In Python, you can use a for loop to repeat a block of code a specific number of times.

In [96]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

In this example, the range(5) function returns an iterable object that produces the numbers 0 to 4. The loop variable i takes on each of these values in turn, and the print() function is called with each value of i.

You can also use the range() function to loop through a sequence of numbers with a specific step size. For example, to print the even numbers between 0 and 10, you can use the following for loop:

In [97]:
```
for i in range(0, 11, 2):
    print(i)
```

```
0
2
4
6
8
10
```

This will print the numbers 0, 2, 4, 6, 8, and 10. The range(0, 11, 2) function call specifies that we want to start at 0, end at 11 (exclusive), and step by 2.

In addition to looping through sequences of numbers, you can also loop through sequences of other types, such as lists or strings. For example, to loop through a list of names and print each name, you can use the following for loop:

In [98]:
```
names = ["Alice", "Bob", "Charlie"]
for name in names:
    print(name)
```

```
Alice
Bob
Charlie
```

This will print the names "Alice", "Bob", and "Charlie" on separate lines. The loop variable name takes on each value in the names list during each iteration of the loop.

In [99]:
```
for achar in "Go Spot Go":
    print(achar)
```

```
G
o

S
p
o
t

G
o
```

In [100…
```
print("This outside line will execute first\n")

for i in range(3):
    print(i)
    print("This inside line will execute three times")
    print("This inside line will also execute three times\n")

print("\nNow we are outside of the for loop!")
```

```
This outside line will execute first

0
This inside line will execute three times
This inside line will also execute three times

1
This inside line will execute three times
This inside line will also execute three times

2
This inside line will execute three times
This inside line will also execute three times


Now we are outside of the for loop!
```

With a for loop, the loop variable is bound, on each iteration, to the next item in a sequence. Sometimes, it is natural to think about iterating through the positions, or indexes of a sequence, rather than through the items themselves.

For example, consider the list ['apple', 'pear', 'apricot', 'cherry', 'peach']. ‘apple’ is at position 0, ‘pear’ at position 1, and ‘peach’ at position 4.

Thus, we can iterate through the indexes by generating a sequence of them, using the range function.

In [101…
```
fruits = ['apple', 'pear', 'apricot', 'cherry', 'peach']
for n in range(5):
    print(n, fruits[n])
```

```
0 apple
1 pear
2 apricot
3 cherry
4 peach
```

In [102…
```
s = "python"
for idx in range(len(s)):
   print(s[idx % 2])
```

```
p
y
p
y
p
y
```

In [103…
```
n=int(input('Enter the Number of elements into the List : '))
l=[]
for i in range(0,n):
  l.append(int(input('enter the elements :')))
print(l)
```

```
Enter the Number of elements into the List : 5
enter the elements :1
enter the elements :2
enter the elements :3
enter the elements :4
enter the elements :5
[1, 2, 3, 4, 5]
```

In [104…
```
s=['Damon','Stefan','Elena','Klaus','Elija']
for i in s:
  print('All the Best',i)
```

```
All the Best Damon
All the Best Stefan
All the Best Elena
All the Best Klaus
All the Best Elija
```

In [105…
```
table=int(input('Enter the Table number : '))
for i in range(1,11):
  print(table,'x',i,'=',table*i)
```

```
Enter the Table number : 12
12 x 1 = 12
12 x 2 = 24
12 x 3 = 36
12 x 4 = 48
12 x 5 = 60
12 x 6 = 72
12 x 7 = 84
12 x 8 = 96
12 x 9 = 108
12 x 10 = 120
```

the way of programmer

1. Use a descriptive variable name for the loop variable: Instead of using a generic name like i or x, use a name that describes the meaning of the variable. For example, if you're iterating over a list of names, you might use name as the loop variable.

2. Use the range() function when iterating over a sequence of numbers: If you need to iterate over a sequence of numbers, use the range() function to generate the sequence. For example, if you want to iterate over the numbers 0 to 9, you can use range(10).

3. Avoid modifying the sequence you're iterating over: When iterating over a list or other sequence, it's generally best to avoid modifying the sequence within the loop. If you need to modify the sequence, consider creating a copy of the sequence first.

4. Use list comprehension when possible: If you need to create a new list based on the elements of an existing list, consider using a list comprehension instead of a for loop. List comprehensions can often be more concise and easier to read.

5. Printing Intermediate results

6. Keeping track of your Iterator Variable and your iterable

Problems

Q: Write a program that prints the first 10 even numbers.

In [106…
```
for i in range(2, 21, 2):
    print(i)
```

```
2
4
6
8
10
12
14
16
18
20
```

Q: Write a program that calculates the sum of the numbers in a list.

In [107…
```
numbers = [1, 2, 3, 4, 5]
sum = 0

for number in numbers:
```

```
    sum += number

print("The sum of the numbers is:", sum)
```

```
The sum of the numbers is: 15
```

Q: Write one for loop to print out each character of the string my_str on a separate line.

In [108…

```
my_str = "MICHIGAN"
for i in my_str:
    print(i)
```

```
M
I
C
H
I
G
A
N
```

Nested for Loop

In Python, a nested for loop is a loop within another loop. The outer loop executes its code block once for each iteration of the inner loop. Here's the basic syntax of a nested for loop:

```
for outer_variable in outer_iterable:
    for inner_variable in inner_iterable:
        # code to be repeated
```

The outer_variable and inner_variable are variables that take on each value in the outer_iterable and inner_iterable objects, respectively, during each iteration of the loop. The outer_iterable and inner_iterable can be any iterable objects, such as lists, tuples, or ranges. The code within the nested loop block is executed once for each combination of values from the outer and inner iterables.

In [109…

```
for i in range(1, 6):
    for j in range(1, 6):
        print(i * j)
```

```
1
2
3
4
5
2
4
6
8
10
3
6
9
12
15
4
8
12
16
20
5
10
15
20
25
```

In [110…

```
for i in range(1, 6):
    for j in range(1, 6):
        print(i, j)
```

```
1 1
1 2
1 3
1 4
1 5
2 1
2 2
2 3
2 4
2 5
3 1
3 2
3 3
3 4
3 5
4 1
4 2
4 3
4 4
4 5
5 1
5 2
5 3
5 4
5 5
```

Problems

Q: Write a program that creates a multiplication table for the numbers 1 to 10.

In [111…

```
for i in range(1, 11):
    for j in range(1, 11):
        print(i * j, end="\t")
    print()
```

```
1       2       3       4       5       6       7       8       9       10
2       4       6       8       10      12      14      16      18      20
3       6       9       12      15      18      21      24      27      30
4       8       12      16      20      24      28      32      36      40
5       10      15      20      25      30      35      40      45      50
6       12      18      24      30      36      42      48      54      60
7       14      21      28      35      42      49      56      63      70
8       16      24      32      40      48      56      64      72      80
9       18      27      36      45      54      63      72      81      90
10      20      30      40      50      60      70      80      90      100
```

This program uses a nested for loop to create a multiplication table. The outer loop iterates through the numbers 1 to 10, while the inner loop iterates through the same range for each outer loop iteration. It then prints the product of the current i and j values, separated by a tab character (\t). The end parameter of the print() function is set to an empty string ("") to prevent a newline from being printed after each row. Finally, a separate print() function call is made after each row to print a newline character (\n).

While loop

In Python, a while loop is a control flow statement that allows you to repeatedly execute a block of code as long as a specified condition is true.

In [112…

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

```
0
1
2
3
4
5
6
7
8
9
```

In this example, the loop starts with i set to 0. The condition i < 10 is true, so the code inside the loop is executed, which prints the value of i and then increments it by 1 using the += operator. The loop continues to execute as long as i < 10 remains true, until i reaches 10 and the condition becomes false.

In [113…

```
n = int(input('Enter to What extent Numbers you Want : '))
a=1
while a <= n :
  print(a,end=' ')
  a = a+1
```

```
Enter to What extent Numbers you Want : 15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

In [114…

```
n = int(input('Enter to What extent Even Numbers you Want : '))
a=0
while a <= n :
  print(a)
  a = a+2
```

```
Enter to What extent Even Numbers you Want : 20
0
2
4
6
8
10
12
14
16
18
20
```

break statement inside a while loop to immediately exit the loop, even if the condition is still true. Here's an example:

In [115…

```
i = 0
while True:
    if i >= 10:
        break
    print(i)
    i += 1
```

```
0
1
2
3
4
5
6
7
8
9
```

In this example, the loop starts with i set to 0, and the condition is set to True, which means the loop will continue indefinitely until the break statement is encountered. Inside the loop, there's an if statement that checks whether i is greater than or equal to 10. If it is, then the break statement is executed, and the loop immediately exits. Otherwise, the code inside the loop is executed, which prints the value of i and increments it by 1 using the += operator.

You can also use the **continue statement** inside a while loop to skip the current iteration and move on to the next one. Here's an example:

In [116…

```
i = 0
while i < 10:
    i += 1
    if i % 2 == 0:
        continue
    print(i)
```

```
1
3
5
7
9
```

In this example, the loop starts with i set to 0. Inside the loop, i is incremented by 1 using the += operator, and then there's an if statement that checks whether i is even (i.e., whether it's divisible by 2 using the modulus operator %). If it is, then the continue statement is executed, and the loop skips the remaining code and moves on to the next iteration. Otherwise, the code inside the loop is executed, which prints the value of i. The loop continues to execute until i reaches 10, at which point the condition becomes false and the loop exits.

Function (def)

In Python, a function is a block of reusable code that performs a specific task. It helps in organizing and simplifying the code by breaking it down into smaller, manageable chunks.

Here is the general syntax of defining a function in Python:

```
def function_name(parameters):
    """
    Docstring: A brief description of what the function does.
    """
    # code block
    return output
```

- The keyword 'def' is used to define a function in Python.
- 'function_name' is the name of the function. It should follow the same naming conventions as variables.
- 'parameters' are the inputs to the function. They are optional and can be empty.
- The 'docstring' is a string that describes the function. It should be enclosed in triple quotes. The code block contains the actual code that the function performs.
- The 'return' statement returns the output of the function. It is optional and can be empty. Here's an example of a function that takes two numbers as inputs and returns their sum:

In [119…
```
def add_numbers(num1, num2):
    """
    This function takes two numbers as input and returns their sum.
    """
    result = num1 + num2
    return result
```

To call this function, we can simply use its name with the inputs:

In [120…
```
sum = add_numbers(3, 5)
print(sum)
# This will print the output 8,
#which is the sum of the two input numbers.
```

```
8
```

In [121…
```
sum = add_numbers(30, 5)
print(sum)
# This will print the output 35,
#which is the sum of the two input numbers.
```

```
35
```

Function Arguments Functions in Python can take input arguments that are passed when the function is called. These arguments are enclosed in parentheses and separated by

commas. For example, in the function add_numbers that we defined earlier, num1 and num2 are the input arguments.

There are three types of input arguments in Python functions:

Required arguments: These are the arguments that are mandatory to pass to the function in the correct positional order.

Default arguments: These are the arguments that have a default value specified in the function definition. If the value is not passed when the function is called, the default value is used.

Keyword arguments: These are the arguments that are identified by their parameter names. They can be passed in any order, as long as the parameter names are specified.

Here is an example of a function that uses all three types of arguments:

In [122…
```
def calculate_bill(amount, tax_rate=0.08, tip_amount=0):
    """
    This function calculates the total bill amount based on the subtotal, tax rate
    """
    tax = amount * tax_rate
    tip = amount * tip_amount
    total = amount + tax + tip
    return total
```

To call this function, we can pass in the required arguments and optionally pass in the other arguments:

In [123…
```
total_bill = calculate_bill(100, tip_amount=0.1)
print(total_bill)    # Output: 118.0
```

```
118.0
```

Variable Scopes Variables that are defined inside a function are only accessible within the function's scope. This means that they cannot be accessed outside of the function, and they do not exist before the function is called. Variables that are defined outside of a function's scope are called global variables, and they can be accessed and modified from anywhere in the code. Here is an example that demonstrates variable scopes in Python:

In [124…
```
global_var = "I am a global variable"

def function():
    local_var = "I am a local variable"
    print(local_var)
    print(global_var)

function()
print(global_var)
print(local_var)    # This will raise a NameError
```

```
I am a local variable
I am a global variable
I am a global variable
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-124-c7367a528cb0> in <cell line: 10>()
      8 function()
      9 print(global_var)
---> 10 print(local_var)    # This will raise a NameError

NameError: name 'local_var' is not defined
```

In this example, global_var is a global variable that can be accessed from inside the function. local_var is a local variable that is defined inside the function and can only be accessed within the function's scope. When we try to access local_var outside of the function, it raises a NameError because the variable does not exist in the current scope.

Function Decorators A function decorator is a special type of function that can modify the behavior of other functions. Decorators are used to add functionality to existing functions without modifying their source code. In Python, decorators are denoted by the @decorator_name syntax, where decorator_name is the name of the decorator function.

Here is an example of a function decorator that adds timing information to a function:

In [125…

```python
import time

def time_it(func):
    """
    This decorator adds timing information to a function.
    """
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Execution time: {end_time - start_time} seconds")
        return result
    return wrapper

@time_it
def calculate_sum(n):
    """
    This function calculates the sum of the first n numbers.
    """
    total = 0
    for i in range(n):
        total += i
    return total

sum = calculate_sum(1000000)
```

```
Execution time: 0.14351224899291992 seconds
```

This is an example of a Python decorator that adds timing information to a function. The decorator takes a function as input and returns a new function that wraps the original function with timing code.

The time module is imported to allow for the measurement of the execution time of the decorated function.

The time_it function is the decorator function that takes in a single argument, func, which is the function that is to be decorated.

The wrapper function is defined inside time_it. This function takes in any number of arguments *args and keyword arguments **kwargs. It measures the execution time of the decorated function by recording the start time before calling the function and the end time after the function completes. It then calculates the difference between the start and end
times to obtain the total execution time.

After measuring the execution time, the original function func is called with the same arguments and keyword arguments as were passed to the wrapper function. The result is stored in the result variable.

Finally, the execution time is printed to the console along with a message stating how long the function took to execute. The result of the original function is then returned.

The @time_it decorator is applied to the calculate_sum function by placing it above the function definition. This causes the calculate_sum function to be passed as an argument to the time_it function, which returns a new function that wraps the original function with timing code.

More using def function

P-1: Basic Function

Write a function that takes two integers as input and returns their sum.

In [ ]:
```
def add_numbers(x, y):
    return x + y

print(add_numbers(2, 3)) # Output: 5
```

```
5
```

Explanation:

The def keyword is used to define a function in Python.

add_numbers is the name of the function.

x and y are the parameters of the function.

The return keyword is used to return a value from the function.

P-2: Function with Default Argument

Write a function that takes a name as input and prints a greeting message with the name. The function should use "Hello" as the default greeting if no greeting is provided.

In [ ]:
```
def greet(name, greeting="Hello"):
    print(greeting + ", " + name)

greet("Alice") # Output: Hello, Alice
greet("Bob", "Hi") # Output: Hi, Bob
```

```
Hello, Alice
Hi, Bob
```

Explanation:

The greeting parameter has a default value of "Hello".

If a value is not provided for greeting, the default value is used.

If a value is provided for greeting, the provided value is used.

P-3:Function with Variable-length Arguments

Write a function that takes any number of integers as input and returns their sum.

In [ ]:
```
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3)) # Output: 6
print(add_numbers(1, 2, 3, 4, 5)) # Output: 15
```

```
6
15
```

Explanation:

The *args syntax is used to allow the function to accept any number of arguments.

The sum function is used to calculate the sum of the input integers.

P-4: Recursive Function

Write a function that calculates the factorial of a number using recursion.

In [ ]:
```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5)) # Output: 120
```

```
120
```

Explanation:

The function calls itself with a smaller value of n until n reaches 0.

The base case is when n is 0, in which case the function returns 1.

The function multiplies n with the result of calling itself with n-1.

P-5: Higher-Order Function

Write a function that takes a function as input and returns a new function that applies the input function to the output of the original function.

In [ ]:
```python
def apply_twice(func, x):
    return func(func(x))

def add_one(x):
    return x + 1

print(apply_twice(add_one, 2)) # Output: 4
```

```
4
```

Explanation:

The apply_twice function takes a function func and an input value x.

It calls func with x and then calls func again with the result of the first call.

The add_one function takes a number and returns the number plus one.

apply_twice(add_one, 2) first calls add_one(2) which returns 3, and then calls add_one(3) which returns 4.

P-6: Lambda Function

Write a function that takes a list of integers and returns a new list containing only the even integers. Use a lambda function to filter the list.

In [ ]:
```
def get_even_numbers(numbers):
    return list(filter(lambda x: x % 2 == 0, numbers))

print(get_even_numbers([1, 2, 3, 4, 5, 6])) # Output: [2, 4, 6]
```

```
[2, 4, 6]
```

Explanation:

The filter function is used to create a new list containing only the elements of the input list that satisfy a condition.

The condition is defined by the lambda function lambda x: x % 2 == 0, which returns True for even numbers and False for odd numbers.

P-7: Generator Function

Write a function that generates the first n Fibonacci numbers.

In [ ]:
```
def fibonacci(n):
    a, b = 0, 1
    for i in range(n):
        yield a
        a, b = b, a + b

for number in fibonacci(10):
    print(number) # Output: 0 1 1 2 3 5 8 13 21 34
```

```
0
1
1
2
3
5
8
13
21
34
```

Explanation:

The yield keyword is used to define a generator function in Python.

The fibonacci function generates the first n Fibonacci numbers using a loop and the formula a, b = b, a + b.

The generator function is called in a loop to print the Fibonacci numbers.

P-8: Decorator Function

Write a decorator function that logs the time taken by a function to execute.

In [ ]:
```
import time

def time_it(func):
    def wrapper(*args, **kwargs):
```

```
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.5f} seconds to execute.")
        return result
    return wrapper

@time_it
def add_numbers(x, y):
    return x + y

print(add_numbers(2, 3)) # Output: 5
```

```
add_numbers took 0.00000 seconds to execute.
5
```

Explanation:

A decorator is a function that takes another function as input and returns a new function that modifies the behavior of the original function.

The time_it function is a decorator that takes a function as input and returns a new function that logs the time taken by the original function to execute.

The wrapper function is the new function returned by the decorator. It calls the original function, measures the time taken to execute it, logs the time, and returns the result.

The @time_it decorator is applied to the add_numbers function to modify its behavior. When add_numbers is called, the decorated version of the function is executed, which logs the time taken to execute the function.

P-9: Memoization

Write a function that calculates the nth Fibonacci number using memoization.

In [ ]:
```
def memoize(func):
    cache = {}
    def wrapper(n):
        if n not in cache:
            cache[n] = func(n)
        return cache[n]
    return wrapper

@memoize
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(10)) # Output: 55
```

```
55
```

Explanation:

Memoization is a technique for optimizing functions by caching the results of expensive function calls and returning the cached result when the same inputs occur again.

The memoize function is a decorator that takes a function as input and returns a new function that caches the results of the original function.

The wrapper function is the new function returned by the decorator. It checks if the result for a given input is already in the cache, and if not, it calls the original function, caches the result, and returns it.

The @memoize decorator is applied to the fibonacci function to cache its results. When fibonacci is called, the decorated version of the function is executed, which caches the results of previous function calls and returns the cached result for subsequent calls.

P-10: Password Generator

Write a program that generates a random password of length n, where n is an integer input by the user.

In [ ]:
```
import random
import string

def generate_password(n):
    chars = string.ascii_letters + string.digits + string.punctuation
    return ''.join(random.choice(chars) for _ in range(n))

n = int(input("Enter password length: "))
print("Password:", generate_password(n))
```

```
Enter password length: 8
Password: :MV_UzA&
```

Explanation:

The generate_password function takes an integer n as input and generates a random password of length n.

The chars variable is a string containing all the possible characters that can be used to generate the password. It includes letters (both uppercase and lowercase), digits, and punctuation characters.

The random.choice(chars) function selects a random character from the chars string, and the for loop generates n random characters and concatenates them into a string using the join method.

The program prompts the user to enter the length of the password, converts the input to an integer using the int function, and generates a password of the specified length using the generate_password function.

P-11: URL Shortener

Write a program that shortens a URL by generating a random short code and storing the URL-short code pair in a dictionary. The program should also be able to retrieve the original URL using the short code.

In [ ]:
```
import random
import string
```

```
url_db = {}

def shorten_url(url):
    chars = string.ascii_letters + string.digits
    short_code = ''.join(random.choice(chars) for _ in range(6))
    url_db[short_code] = url
    return f"http://myurl.com/{short_code}"

def expand_url(short_code):
    return url_db.get(short_code, "Short code not found.")

url = input("Enter URL: ")
short_url = shorten_url(url)
print("Short URL:", short_url)

short_code = short_url.split("/")[-1]
original_url = expand_url(short_code)
print("Original URL:", original_url)
```

```
Enter URL: learn.gitam.edu
Short URL: http://myurl.com/YPOMUV
Original URL: learn.gitam.edu
```

Explanation:

The shorten_url function takes a URL as input, generates a random 6-character short code using letters and digits, stores the URL-short code pair in the url_db dictionary, and returns the short URL.

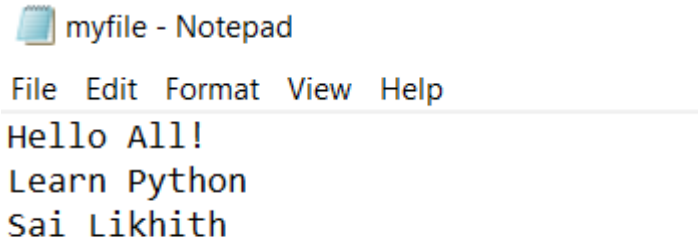
The expand_url function takes a short code as input, retrieves the original URL from the url_db dictionary using the get method, and returns the original URL or a "Short code not found" message if the short code is not in the dictionary.

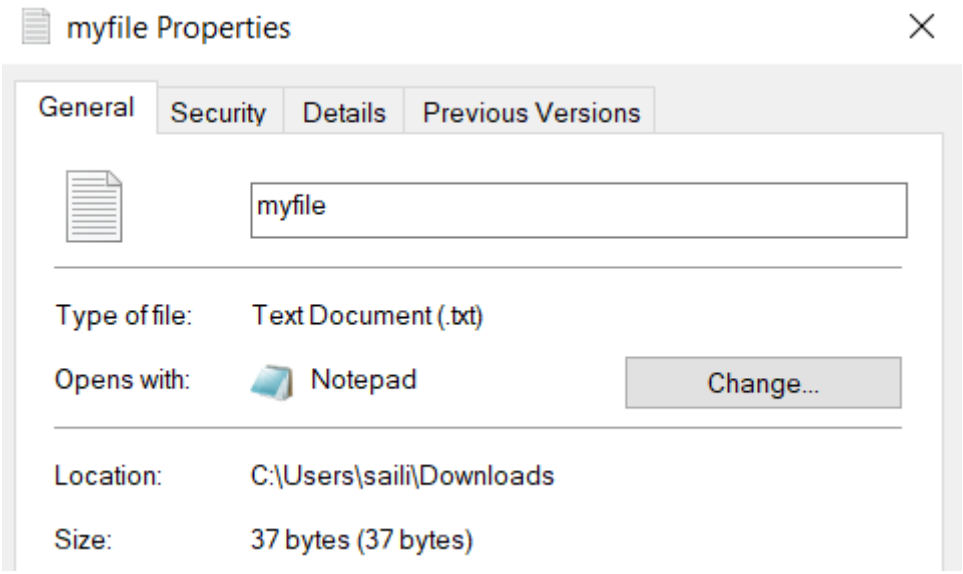
The program prompts the user to enter a URL, shortens the URL using the shorten_url function, and prints the short URL.

The program extracts the short code from the short URL using the split method, retrieves the original URL using the expand_url function, and prints the original URL or a "Short code not found" message.

File Handling

Method Name	Use	Explanation
`open`	`open(filename,'r')`	Open a file called filename and use it for reading. This will return a reference to a file object.
`open`	`open(filename,'w')`	Open a file called filename and use it for writing. This will also return a reference to a file object.
`close`	`filevariable.close()`	File use is complete.





Renaming a File

In [1]:
```
import os
os.rename('myfile.txt','Python.txt')
```

This Python code imports the os module, which provides a way to interact with the operating system. The os.rename() function is used to rename a file on the file system. In this case, the file named myfile.txt is being renamed to Python.txt.

So, when this code is executed, it will rename the file myfile.txt to Python.txt in the current directory. If myfile.txt doesn't exist or the current user doesn't have the necessary permissions to modify the file, this code will raise an error.

It's important to note that if a file with the new name already exists in the directory, it will be overwritten without any warning or confirmation.

Opening and Reading data from a file

In [2]:
```
# f=open('path of the file','access mode')
f=open('Python.txt','r')
data=f.read()
print(data)
f.close()
```

```
Hello All!
Learn Python
Sai Likhith
```

This Python code opens a file named Python.txt for reading using the open() function and assigns the resulting file object to the variable f. The 'r' argument specifies that the file should be opened in read mode.

The contents of the file are then read using the read() method, and the resulting data is stored in the data variable.

Finally, the print() function is used to output the contents of the file to the console, and the file is closed using the close() method of the file object.

It's important to note that if the file doesn't exist or the current user doesn't have the necessary permissions to read the file, this code will raise an error.

writing data to a file

In [3]:
```
msg1='This is Likhith\n'
msg2='Linkedin\n'
msg3='Positive'
f=open('Python.txt','w')
f.write(msg1)
f.write(msg2)
f.write(msg3)
f.close()
```

This code snippet opens a file named Python.txt for writing using the open() function with the 'w' mode. This mode specifies that the file should be opened for writing, and if the file already exists, its contents will be overwritten.

Then, the code writes three strings (msg1, msg2, and msg3) to the file using the write() method. Each string is written on a separate line, as the newline character (\n) is included at the end of the first two strings.

Finally, the close() method is called to close the file and ensure that any changes made to the file are saved. If the file didn't exist before this code was executed, it will be created in the current directory.

Overall, this code is an example of how to create and write data to a file using Python.

In [4]:
```
f=open('Python.txt','r+')
data=f.read()
```

```
f.write('\nThis is Likhith.')
print(data)
f.close()
```

```
This is Likhith
Linkedin
Positive
```

This code snippet opens a file named Python.txt in read-write mode using the 'r+' mode. This mode specifies that the file should be opened for reading and writing, allowing the code to both read data from and write data to the file.

Next, the read() method is used to read the contents of the file into a string variable named data.

Then, the write() method is used to append the string '\nThis is Likhith.' to the end of the file. This string will be added on a new line due to the newline character (\n) included at the beginning of the string.

After that, the contents of the original file are printed using the print() function, which will output the value of data.

Finally, the close() method is called to close the file and ensure that any changes made to the file are saved.

Overall, this code is an example of how to read and write data to a file in Python using the 'r+' mode. Note that when using this mode, any data written to the file will be appended to the end of the existing data, rather than overwriting it.

In [5]:
```
f=open('Python.txt','r+')
f.seek(15,0)
f.write('Yes, I am Good Boy')
f.seek(0)
data=f.read()
print(data)
f.close()
```

```
This is LikhithYes, I am Good Boy
This is Likhith.
```

This code snippet opens a file named Python.txt in read-write mode using the 'r+' mode, just like the previous example.

Then, the seek() method is used to move the file pointer to the 16th byte (starting from the beginning of the file). This is done using the parameters (15,0) to specify the byte offset and the origin, where 0 means the beginning of the file.

Next, the write() method is used to overwrite the data at the current position of the file pointer with the string 'Yes, I am Good Boy'. This will replace the text that was originally at the 16th byte position with the new string.

After that, the seek() method is called again with the parameters (0) to move the file pointer to the beginning of the file.

Then, the read() method is used to read the contents of the file into the string variable data.

Finally, the contents of the file are printed using the print() function, which will output the value of data.

In [6]:
```
f=open('Python.txt','r')
data=f.read()
print(data)
f.close()
```

```
This is LikhithYes, I am Good Boy
This is Likhith.
```

In [7]:
```
f=open('Python.txt','w')
f.write('Apple')
f.close()
```

In [8]:
```
f=open('Python.txt','r+')
f.write('SAI LIKHITH')
f.close()
```

In [9]:
```
msg1='This is Likhith\n'
msg2='Career\n'
msg3='Positive'
f=open('Python.txt','w')
f.write(msg1)
f.write(msg2)
f.write(msg3)
f.close()
```

In [10]:
```
f=open('Python.txt','a')
f.write('\nHappy')
f.close()
```

In [11]:
```
f=open('Python.txt','r')
data=f.read()
print(data)
f.close()
```

```
This is Likhith
Career
Positive
Happy
```

Introduction to .csv format

CSV (Comma Separated Values) is a popular file format used to store and exchange data between systems. It's a plain text format that separates values with commas and uses line breaks to separate records. The CSV format is often used for storing tabular data, such as spreadsheets or databases, and can be opened and edited with spreadsheet applications like Microsoft Excel or Google Sheets.

In a CSV file, each line represents a single record or row of data, and each field or column is separated by a comma. For example, consider the following CSV data:

```
Name, Age, City
John, 25, New York
Emily, 30, London
```

In this example, the first line contains the column headers for the data, and the next two lines represent individual records. Each record contains values for the Name, Age, and City fields, separated by commas.

CSV files can also include other types of delimiters besides commas, such as tabs or semicolons, depending on the needs of the data being stored. Additionally, CSV files may include a header row, which provides column names for the data, or may simply list the data without any column labels.

CSV files are a flexible and widely used format for storing and exchanging data, particularly in the context of spreadsheets and databases.

To work with CSV files in Python, you first need to open the file using the open() function, just like you would with any other file. You can then use the csv.reader() or csv.writer() methods to read or write data from the file. Here's an example of reading data from a CSV file using the csv.reader() method:

In [ ]:
```python
import csv

# Open the CSV file
with open('example.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Loop through each row of the CSV file
    for row in reader:
        # Print each row of data
        print(row)
```

In this example, we're opening a file named example.csv in read mode using the open() function, and creating a CSV reader object using the csv.reader() method. We can then loop through each row of the CSV file using a for loop, and print out each row of data using the print() function.

Similarly, you can use the csv.writer() method to write data to a CSV file. Here's an example of writing data to a CSV file:

In [ ]:
```python
import csv

# Open the CSV file in write mode
with open('output.csv', 'w', newline='') as file:
    # Create a CSV writer object
    writer = csv.writer(file)

    # Write some data to the CSV file
    writer.writerow(['Name', 'Age', 'City'])
    writer.writerow(['John', '25', 'New York'])
    writer.writerow(['Emily', '30', 'London'])
```

In this example, we're opening a file named output.csv in write mode using the open() function, and creating a CSV writer object using the csv.writer() method. We can then write some data to the CSV file using the writerow() method. Each call to writerow() writes a new row of data to the file, and each argument to writerow() represents a single field or column of data.

Overall, the csv module provides a simple and effective way to work with CSV files in Python, allowing you to read and write data to and from CSV files with ease.

Object Oriented Programming

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviours are bundled into individual objects.

Class: A class is a blueprint or a template for creating objects. It defines the properties (attributes) and behaviors (methods) of an object.

Object: An object is an instance of a class. It is a specific entity that has its own set of attributes and methods.

Inheritance: Inheritance is a mechanism in which one class inherits the properties and methods of another class. The class that inherits is called the subclass or derived class, and the class from which it inherits is called the superclass or base class.

Polymorphism: Polymorphism is the ability of an object to take on many forms. It allows objects of different classes to be treated as if they are of the same class.

Encapsulation: Encapsulation is the process of hiding the internal details of an object and providing a public interface for accessing and manipulating the object.

Abstraction: Abstraction is the process of reducing complexity by hiding unnecessary details and exposing only the essential features of an object.

Method Overloading: Method overloading is the ability to define multiple methods with the same name but different parameters. Python does not support method overloading directly, but it can be achieved using default arguments or variable-length arguments.

Method Overriding: Method overriding is the ability of a subclass to provide its own implementation of a method that is already defined in its superclass.

Class and Object

Classes:

A class is a blueprint or a template for creating objects that define a set of attributes and methods.

Attributes are variables that store data and methods are functions that perform actions on that data.

Classes are defined using the class keyword followed by the class name, and can include an optional ***init*** method that initializes the object's attributes when it is created.

class members are variables and methods that belong to a class and are shared by all instances of the class. Class members can be divided into two types:

class variables and class methods

Class Variables: A class variable is a variable that is defined within a class, but outside of any method. It is shared by all instances of the class and can be accessed using the class name or the instance name.

Class Methods: A class method is a method that is bound to the class and not the instance of the class. It is defined using the @classmethod decorator and takes the class as its first argument.

Objects:

An object is an instance of a class that has its own unique set of attributes and methods.

Objects are created using the class name followed by parentheses, and can be assigned to variables for later use.

Terminology:

- **`self`** : `self` is a reference to the object being created or manipulated. It is used to access the object's attributes and methods.
- **`__init__`**: `__init__`is a special method that is called when an object is created from the class. It initializes the object's attributes with the values passed as arguments to the constructor.
- **Inheritance**: Inheritance is a way to create new classes based on existing classes. A subclass inherits the attributes and methods of its parent class and can also define its own attributes and methods.
- **Polymorphism**: Polymorphism is the ability of objects of different classes to be used interchangeably. This allows for greater flexibility and modularity in programming.
- **Encapsulation**: Encapsulation is the practice of hiding the internal workings of an object and providing a public interface for interacting with it. This helps to prevent unintended modification of the object's attributes and methods.

- **Abstraction**: Abstraction is the practice of reducing complexity by hiding unnecessary details and focusing on the essential features of an object or system.

Programming :)

In [1]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I'm {self.age} years old.")
```

In this example, the Person class has two attributes (name and age) and one method (greet).

The **init** method is a special method that is called when an object is created from the class. It initializes the object with the values passed as arguments to the constructor. The self parameter refers to the object being created and is used to access its attributes and methods.

The greet method is a simple method that prints a greeting message using the object's name and age attributes.

To create an object of the Person class, you can use the following code:

In [2]:
```
person = Person("Alice", 30)
```

This creates a new Person object with the name "Alice" and age 30.

You can access the object's attributes using dot notation:

In [3]:
```
print(person.name)   # Output: "Alice"
print(person.age)    # Output: 30
```

```
Alice
30
```

You can also call the object's methods using dot notation:

In [4]:
```
person.greet()
# Output: "Hello, my name is Alice and I'm 30 years old."
```

```
Hello, my name is Alice and I'm 30 years old.
```

Classes provide a powerful way to organize code and create reusable components in Python. By encapsulating data and functionality into a single unit, you can create more maintainable and scalable programs.

P-1: Student class with data members student rollno, name,address, course. Include aconstructor to initialize data members. Add a method to print the student details.

In [5]:
```
class student:
  RollNo=0
  Name=''
  Address=''
  Course=''
```

```
  def__init__(self, RollNo, Name, Address, Course):
    self.RollNo= RollNo
    self.Name=Name
    self.Address=Address
    self.Course=Course

  def Print(self):
    print('{}, holding {} Registration Number lives in {} enrolled in {} Course.'
    .format(self.Name, self.RollNo, self.Address, self.Course))
```

In [6]:
```
Likhith=student(122010405056, 'Sai Likhith Panuganti','Rushikonda', 'ECE-AIML')
Likhith.Print()
```

```
Sai Likhith Panuganti, holding 122010405056 Registration Number lives in Rushikond
a enrolled in ECE-AIML Course.
```

P-2: Book class with data members book_id,name,cost and publisher. Include constructor and a method to display the book details. Create 3 objects and displaytheir details.

In [7]:
```
class Book:
  BookId=''
  Name=''
  Cost=0
  Publisher =''
  def___init__(self,BookId,Name,Cost,Publisher):
    self.BookId = BookId
    self.Name = Name
    self.Cost = Cost
    self.Publisher = Publisher
  def showDetails(self):
    print(self.BookId,self.Name,":  ₹",self.Cost,"from",self.Publisher)
```

In [8]:
```
harryPotter = Book("978-1408855652","Harry Potter and the Philosopher's Stone",309
percyJackson = Book("978-0141346809","Percy Jackson and the Lightning Thief",283,"
wimpyKid = Book("978-0141324906","Diary Of A Wimpy Kid",251,"Penguin")

harryPotter.showDetails()
percyJackson.showDetails()
wimpyKid.showDetails()
```

```
978-1408855652 Harry Potter and the Philosopher's Stone : ₹ 309 from Bloomsbury
978-0141346809 Percy Jackson and the Lightning Thief : ₹ 283 from Penguin
978-0141324906 Diary Of A Wimpy Kid : ₹ 251 from Penguin
```

P-3: Account class with data members acc_no,name,balance. Include a constructor andmethods to perform deposit and withdraw operations on account. Create account object perform some operations and display the account details.

In [9]:
```
class Account:
    acc_no = 0
    name = ''
    balance = 0
    def__init__(self,acc_no,name,balance):
        self.acc_no = acc_no
        self.name = name
        self.balance = balance
    def deposit(self,amount):
        self.balance =self.balance+ amount
    def withdraw(self,amount):
```

```
        self.balance= self.balance- amount
    def display(self):
        print(self.balance)
```

In [10]:
```
myAccount = Account(5056,'Sai Likhith',100000)
myAccount.display()
myAccount.deposit(50000)
myAccount.display()
myAccount.withdraw(25000)
myAccount.display()
```

```
100000
150000
125000
```

P-4: Product class with data members product_id, product_name, price, expiry_date. Include constructor to initialize data members and a method to print products details.

In [11]:
```
class Product:
    product_id = 0
    product_name = ''
    price = 0
    expiry_date = ''
    def___init_(self,product_id,product_name,price,expiry_date):
        self.product_id = product_id
        self.product_name = product_name
        self.price = price
        self.expiry_date = expiry_date
    def showProductDetails(self):
        print('The Product-{} named {} costs {}/- Rupees can be used by {}.'
        .format(self.product_id,self.product_name,self.price,self.expiry_date))
```

In [12]:
```
noodles = Product(1,"Munch",10,"22/12/2023")
biscuit = Product(2,"Jim Jam",25,"1/9/2023")
cake = Product(3,"Corn Flakes",100,"10/12/2023")
noodles.showProductDetails()
biscuit.showProductDetails()
cake.showProductDetails()
```

```
The Product-1 named Munch costs 10/- Rupees can be used by 22/12/2023.
The Product-2 named Jim Jam costs 25/- Rupees can be used by 1/9/2023.
The Product-3 named Corn Flakes costs 100/- Rupees can be used by 10/12/2023.
```

P-5: Complex_Number with data members real_part and imaginary_part. Include constructor to initialize complex number. Add a method which adds two complexnumbers.

In [13]:
```
class ComplexNumber:
    real_part = 0
    imaginary_part = 0
    def___init_(self,real_part,imaginary_part):
        self.real_part = real_part
        self.imaginary_part = imaginary_part
def add(a,b):
    c = ComplexNumber(0,0)
    c.real_part = a.real_part + b.real_part
    c.imaginary_part = a.imaginary_part + b.imaginary_part
    return c

a = ComplexNumber(25,28)
b = ComplexNumber(25,28)
```

```
sum = add(a,b)
print(sum.real_part,'+',sum.imaginary_part,'i')
```

```
50 + 56 i
```

P-6: Employee class with data members eno,ename,sal,designation. Include constructor to initialize employee details and count the number of employee objectscreated.

In [14]:
```
class Count():
 c = 0
count = Count()
class Employee:
  eno = 0
  ename = ""
  sal = 0
  designation = ""
  def___init__(self,eno,ename,sal,designation):
    self.eno = eno
    self.ename = ename
    self.sal = sal
    self.designation = designation
    count.c += 1
Stefan = Employee(0,"Stefan Salvatore",1,"CEO")
Elena = Employee(1,"Elena Gilbert",0,"Co-founder")
Ric = Employee(100,"Alaric Saltzman",100000000,"Fellow")
print(count.c)
```

```
3
```

P-7: Create a class called Distance. A person has to travel a certain distace and he usedtwo cars. Now create two objects "cardist1" and "cardist2" for the class Distance. Addthe two objects distances and put the total distance in the third object of class Distance "totaldist". Take one data member, which will accept the distance input inkm. Take two functions, for accepting the distance and the other for displaying.
Display the total distance in meters.

In [15]:
```
class Distance:
 distance = 0
 def inputDistance(self):
   self.distance = input("Enter the distance in km:")
 def showDistance(self):
   print(self.distance)
cardist1 = Distance()
cardist1.inputDistance()
cardist2 = Distance()
cardist2.inputDistance()
totaldist = Distance()
totaldist.distance = str(int(cardist1.distance)+int(cardist2.distance))
print("Total distance=",int(totaldist.distance)*1000,"m")
```

```
Enter the distance in km:68
Enter the distance in km:75
Total distance= 143000 m
```

Inheritance

Inheritance is a way to create new classes based on existing classes. A subclass inherits the attributes and methods of its parent class and can also define its own attributes and methods.

In [16]:
```
class Animal:
    def___init__(self, name):
        self.name = name

    def speak(self):
        print("Unknown  sound")

class  Dog(Animal):
    def___init_(self, name):
        super().__init__(name)

    def speak(self):
        print("Woof")

class  Cat(Animal):
    def___init_(self, name):
        super().__init__(name)

    def speak(self):
        print("Meow")

dog = Dog("Fido")
dog.speak()   # Output: "Woof"

cat = Cat("Whiskers")
cat.speak()   # Output: "Meow"
```

```
Woof
Meow
```

In this example, the Animal class has a constructor that takes a name parameter and a speak method that prints "Unknown sound". The Dog and Cat classes are subclasses of Animal and override the speak method with their own implementation.

When you create a Dog object and call its speak method, it prints "Woof". Similarly, when you create a Cat object and call its speak method, it prints "Meow".

P-1: Develop a program to Perform Python Multi-Level and multiple inheritances.

In [17]:
```
class Apple:
 manufacturer = 'Apple Inc'
 contact_website  =  'www.apple.com/contact'
 name = 'Apple'
 def  contact_details(self):
   print('Contact us at ', self.contact_website)
class  MacBook(Apple):
 def___init__(self):
   self.year_of_manufacture  =  2018
 def  manufacture_details(self):
   print('This MacBook was manufactured in {0}, by {1}.'
         .format(self.year_of_manufacture,  self.manufacturer))
```

```
macbook = MacBook()
macbook.manufacture_details()


class OperatingSystem:
 multitasking = True
 name = 'Mac OS'

class MacTower(OperatingSystem, Apple):
 def___init__(self):
   if self.multitasking is True:
     print('Multitasking system')
     print('Name: {}'.format(self.name))

mactower = MacTower()

# Multilevel inheritence
class MusicalInstrument:
 num_of_major_keys = 12

class StringInstrument(MusicalInstrument):
 type_of_wood = 'Tonewood'

class Guitar(StringInstrument):
 def___init__(self):
   self.num_of_strings = 6
   print('The guitar consists of {0} strings,' +'it is made of {1} and can play {2
   .format(self.num_of_strings,self.type_of_wood, self.num_of_major_keys))
guitar = Guitar()
```

```
This MacBook was manufactured in 2018, by Apple Inc.
Multitasking system
Name: Mac OS
The guitar consists of {0} strings,it is made of Tonewood and can play 12 keys.
```

Polymorphism

Polymorphism is the ability of objects of different classes to be used interchangeably. This allows for greater flexibility and modularity in programming.

In [18]:
```
class Shape:
    def area(self):
        pass

class  Square(Shape):
    def___init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

class  Circle(Shape):
    def___init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

def  print_area(shape):
    print("Area:",  shape.area())

square = Square(5)
circle = Circle(3)

print_area(square)   # Output: "Area: 25"
print_area(circle)   # Output: "Area: 28.26"
```

```
Area: 25
Area:  28.259999999999998
```

In this example, the Shape class has an area method that is implemented by its subclasses Square and Circle. The print_area function takes an object of type Shape as a parameter and calls its area method, regardless of whether it's a Square or Circle object.

When you create a Square object and pass it to print_area, it prints the area of the square. Similarly, when you create a Circle object and pass it to print_area, it prints the area of the circle.

Encapsulation

Encapsulation is the practice of hiding the internal workings of an object and providing a public interface for interacting with it. This helps to prevent unintended modification of the object's attributes and methods.

In [19]:
```python
class BankAccount:
    def __init__(self, balance):
        self._balance = balance

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if amount <= self._balance:
            self._balance -= amount
        else:
            print("Insufficient balance")

    def get_balance(self):
        return self._balance

account = BankAccount(1000)
account.withdraw(500)   # Output: "Balance: 500"
account.deposit(1000)   # Output: "Balance: 1500"
```

This is an example of encapsulation. The BankAccount class has a private attribute _balance, which can only be accessed or modified by the class methods. The **init** method is the constructor that takes a balance parameter and initializes the _balance attribute.

The deposit method takes an amount parameter and adds it to the _balance attribute. The withdraw method takes an amount parameter and subtracts it from the _balance attribute, but only if the balance is sufficient. If the balance is insufficient, it prints a message saying so.

The get_balance method returns the _balance attribute.

When you create a BankAccount object with an initial balance of 1000 and call its withdraw method with an amount of 500, it subtracts 500 from the balance and prints "Balance: 500". When you call its deposit method with an amount of 1000, it adds 1000 to the balance and prints "Balance: 1500".

This example shows how encapsulation can be used to protect the object's internal state and provide a public interface for interacting with it.

Abstraction

Abstraction is a technique in object-oriented programming that allows us to hide the complexity of an object and provide a simpler interface for interacting with it. In Python, we can use abstract classes and methods to achieve abstraction.

An abstract class is a class that cannot be instantiated, and it is designed to be subclassed by other classes. Abstract classes often define abstract methods, which are methods that have a declaration but no implementation. The implementation of an abstract method is provided by its concrete subclasses.

Here is an example of an abstract class Shape that defines an abstract method area:

In [20]:
```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

In this example, Shape is an abstract class that inherits from the ABC class provided by the abc module. The area method is decorated with the abstractmethod decorator, which indicates that it is an abstract method and must be implemented by its concrete subclasses.

Here is an example of a concrete subclass Rectangle that implements the area method:

In [21]:
```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

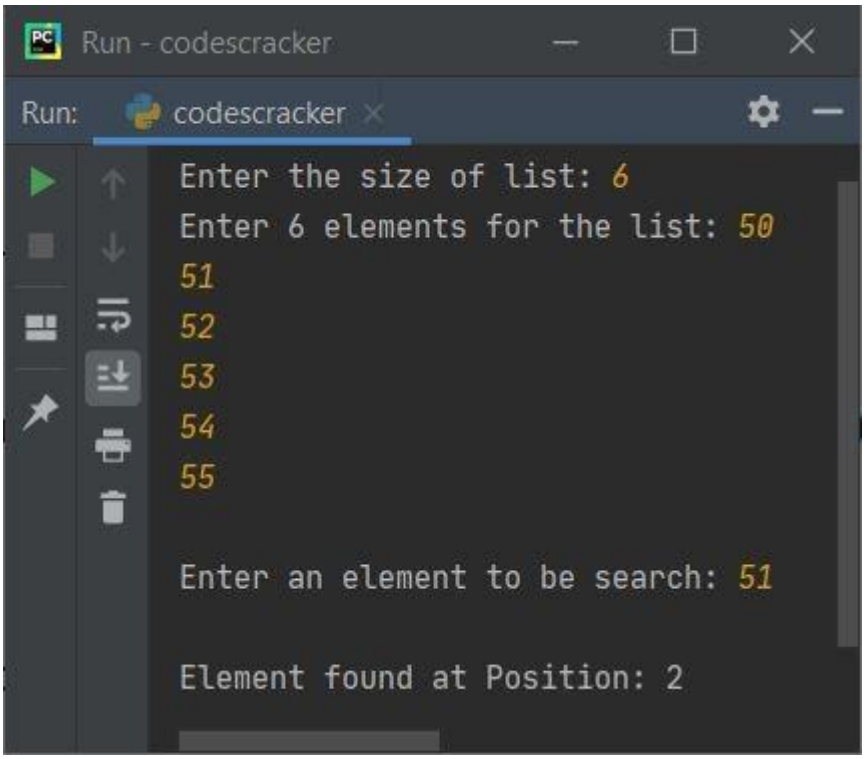
In this example, Rectangle is a concrete subclass of Shape that implements the area method by calculating the area of a rectangle.

By using abstraction, we can create a simpler interface for interacting with complex objects. In this example, we can create instances of Rectangle and call its area method to get its area, without having to worry about the details of how the area is calculated. We can also create other subclasses of Shape that implement the area method differently, such as Circle or Triangle, without affecting the code that uses Shape objects.

Introduction to Searching

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in data structure is listed below:

1. Linear Search
2. Binary Search



Sequential Search or Linear Search

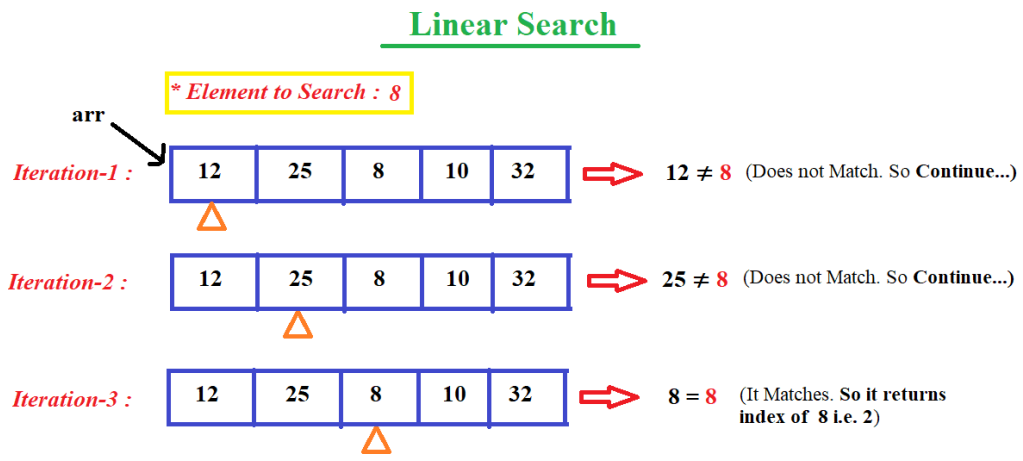
Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Features of Linear Search Algorithm:

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of O(n), which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
3. It has a very simple implementation



The following Python implementation of the linear search algorithm, which is used to find the position of a given element in an array. Here's how the algorithm works:

1. Iterate over each element in the array.
2. Compare the current element with the given element x.
3. If the current element is equal to x, return its index in the array.
4. If the element is not found in the array, return -1.

In [22]:
```
def linearSearch(array, n, x):
    for i in range(0, n):
        if (array[i] == x):
            return i
    return -1
```

This line defines a function linearSearch that takes three arguments: array, which is the array to be searched, n, which is the length of the array, and x, which is the element to be found.

This line starts a loop that iterates over each element in the array. The loop variable i is used to keep track of the index of the current element.

This line checks if the current element is equal to x. If it is, the function immediately returns the index of the current element.

This line is executed if the function does not find the element x in the array. It returns -1 to indicate that the element was not found.

this function has a time complexity of O(n), since in the worst case it may have to iterate over every element in the array. However, it has a space complexity of O(1), since it does not need to allocate any additional memory beyond the input parameters.

In [23]:
```
array = [11,22,32,40,51,60,74,80,95,90,85,76] #SAI LIKHITH 122010405056
x = 85
n = len(array)
result = linearSearch(array, n, x)
if(result == -1):
    print("Search element not found")
else:
    print("Element found at Index: ", result)
```

```
Element found at Index:  10
```

In [24]:
```
def LS(array, n):
    print(len(array))
    if n in array:
        for i in range(len(array)):
            if array[i]==n:
                a=i
        print('The number {} is at index {}'.format(n,a))
    else:
        print('The number {} is not in the array.'.format(n))
```

In [25]:
```
LS([1,2,3,4,5,6],6)
```

```
6
The number 6 is at index 5
```

Binary search

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

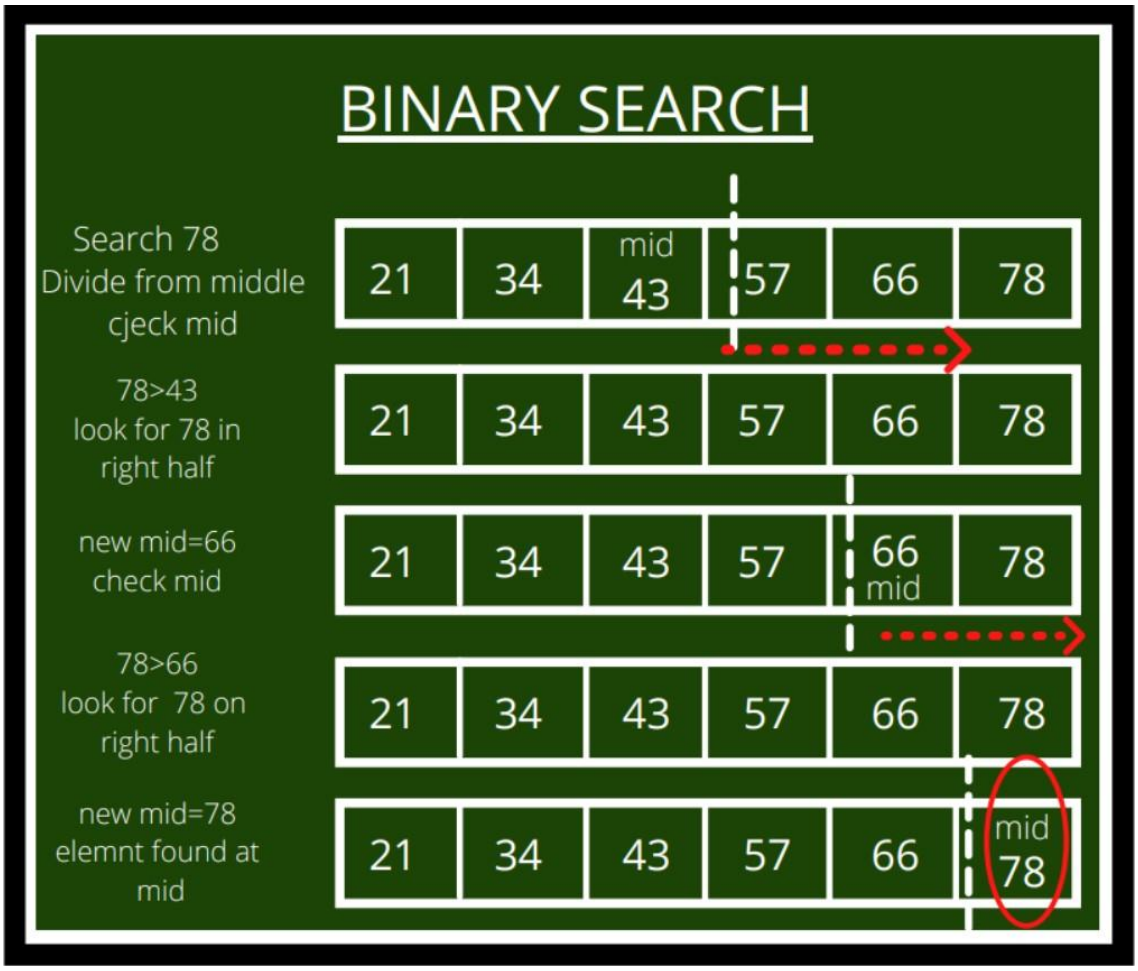
Binary Search is useful when there are large number of elements in an array and they are sorted. So a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search:

1. It is great to search through large sorted arrays.
2. It has a time complexity of O(log n) which is a very good time complexity. It has a simple implementation.

Binary search is a fast search algorithm with run-time complexity of Ï(log n). This search algorithm works on the principle of divide and conquers. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looksfor a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the subarray as well until the size of the sub array reduces to zero.



The following is the implementation of the binary search algorithm, which is used to find the position of a given element in a sorted array. Here's how the algorithm works:

1. Initialize the start and end indices of the array.
2. Compute the middle index of the array.
3. Compare the middle element with the given element `x`.
4. If the middle element is equal to `x`, return its index in the array.
5. If the middle element is less than `x`, the element must be in the right half of the array. Update the start index to the right of the middle index.
6. If the middle element is greater than `x`, the element must be in the left half of the array. Update the end index to the left of the middle index.
7. Repeat steps 2-6 until the element is found or the search space is exhausted.

In [1]:
```
def binary_search(arr, x):
    # Initialize start and end indices of the array
    start = 0
    end = len(arr) - 1

    while start <= end:
        mid = (start + end) // 2  # Find the middle index of the array

        if arr[mid] == x:
            return mid  # Found x, return the index
        elif arr[mid] < x:
            start = mid + 1  # x is in the right half of the array
        else:
            end = mid - 1  # x is in the left half of the array
```

```
    return -1  # x is not in the array
```

Here's a line-by-line explanation of the code:

```
def binary_search(arr, x):
```

This line defines a function `binary_search` that takes two arguments: `arr`, which is the sorted array to be searched, and `x`, which is the element to be found.

```
start = 0
end = len(arr) - 1
```

These lines initialize the start and end indices of the search space to the first and last indices of the array, respectively.

```
while start <= end:
```

This line starts a loop that continues until the search space is exhausted (i.e., the start index is greater than the end index).

```
mid = (start + end) // 2
```

This line computes the middle index of the search space.

```
if arr[mid] == x:
    return mid
```

This line checks if the middle element of the search space is equal to the element `x`. If it is, the function immediately returns the index of the middle element.

```
elif arr[mid] < x:
    start = mid + 1
```

This line checks if the middle element of the search space is less than `x`. If it is, the element must be in the right half of the search space. The start index is updated to the right of the middle index.

```
else:
    end = mid - 1
```

This line is executed if the middle element of the search space is greater than `x`. In this case, the element must be in the left half of the search space. The end index is updated to the left of the middle index.

```
return -1
```

This line is executed if the function does not find the element `x` in the array. It returns `-1` to indicate that the element was not found.

Overall, this function has a time complexity of O(log n), since the size of the search space is divided by 2 with each iteration of the loop. However, it has a space complexity of O(1), since it does not need to allocate any additional memory beyond the input parameters.

In [2]:
```
binary_search([1,2,3,4,5,6,7],69)
```

Out[2]:
```
-1
```

In [3]:
```
def BS(array):
    for j in range(len(array)):
        for i in range(0,len(array)-1):
            if  array[i]>array[i+1]:
                array[i],array[i+1]  =  array[i+1],array[i]
    print(array)
```

In [4]:
```
BS([91,222,33,31,43,5,6])
```

```
[5, 6, 31, 33, 43, 91, 222]
```

Introduction to Sorting

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly and for producing human - readable input Sorting

algorithms are often classified by :

- Computational complexity (worst, average and best case) in terms of the size of the list (N). For typical sorting algorithms good behaviour is O(NlogN) and worst case behaviour is O(N2) and the average case behaviour is O(N).
- Memory Utilization
- Stability - Maintaining relative order of records with equal keys.
- No. of comparisions.
- Methods applied like Insertion, exchange, selection, merging etc.

Sorting is a process of linear ordering of list of objects. Sorting techniques are categorized into

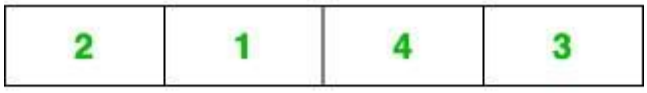
1. Internal Sorting
2. External Sorting

Internal Sorting takes place in the main memory of a computer.

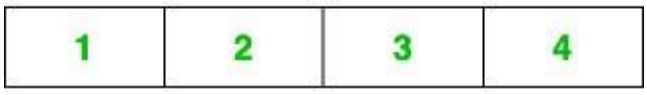
eg : - Bubble sort, Insertion sort, Shell sort, Quick sort, Heap sort, etc.

External Sorting, takes place in the secondary memory of a computer, Since the number of objects to be sorted is too large to fit in main memory.

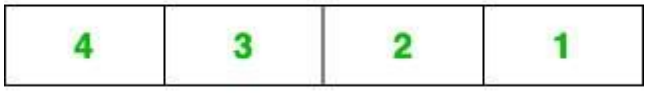
eg : - Merge Sort, Multiway Merge, Polyphase merge



Unsorted Array



Array sorted in ascending order



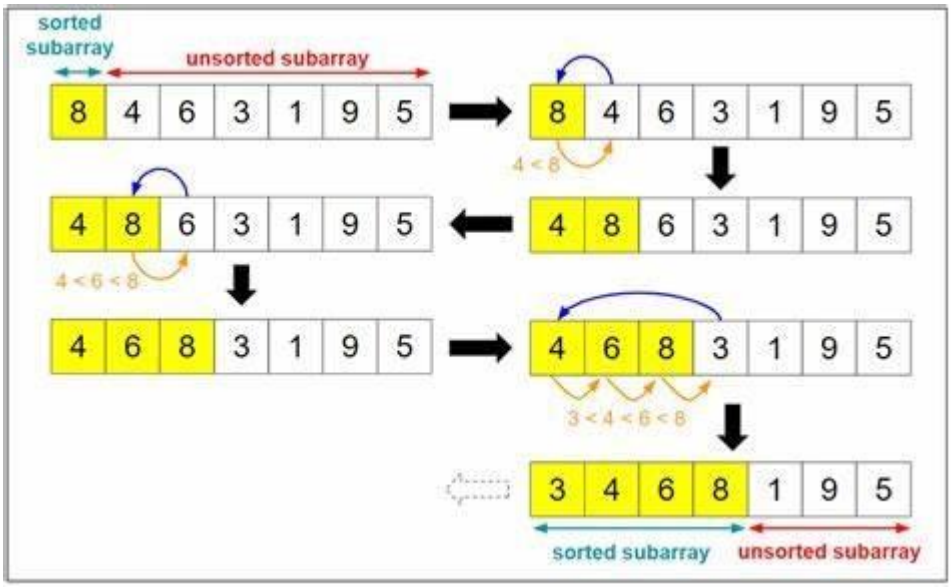
Array sorted in descending order

Insertion Sort

Insertion sorts works by taking elements from the list one by one and inserting them in their current position into a new sorted list. Insertion sort consists of N - 1 passes, where N is the number of elements to be sorted. The ith pass of insertion sort will insert the ith element A[i] into its rightful place among A[1], A[2] --- A[i - 1]. After doing this insertion the records occupying A[1]...A[i] are in sorted order.

Limitations Of Insertion Sort :

- It is relatively efficient for small lists and mostly - sorted lists.
- It is expensive because of shifting all following elements by one.



The following is the implementation of the Insertion Sort algorithm in Python. Here's how it works:

1. The function takes an unsorted array as input.
2. It starts a loop that iterates over each element in the array, starting at the second element (index 1).
3. For each element, it stores its value in a variable called "key".
4. It then compares the key with the element to its left (i.e., the element at index j).
5. If the key is smaller than the element to its left, the function moves the element to the right one position, creating a "hole" for the key.
6. It then decrements the value of j and repeats the comparison with the element to the left until it finds an element that is smaller than the key or reaches the beginning of the array.
7. Finally, it inserts the key into the hole created in step 5.

This process continues until all elements have been processed and the array is sorted in ascending order.

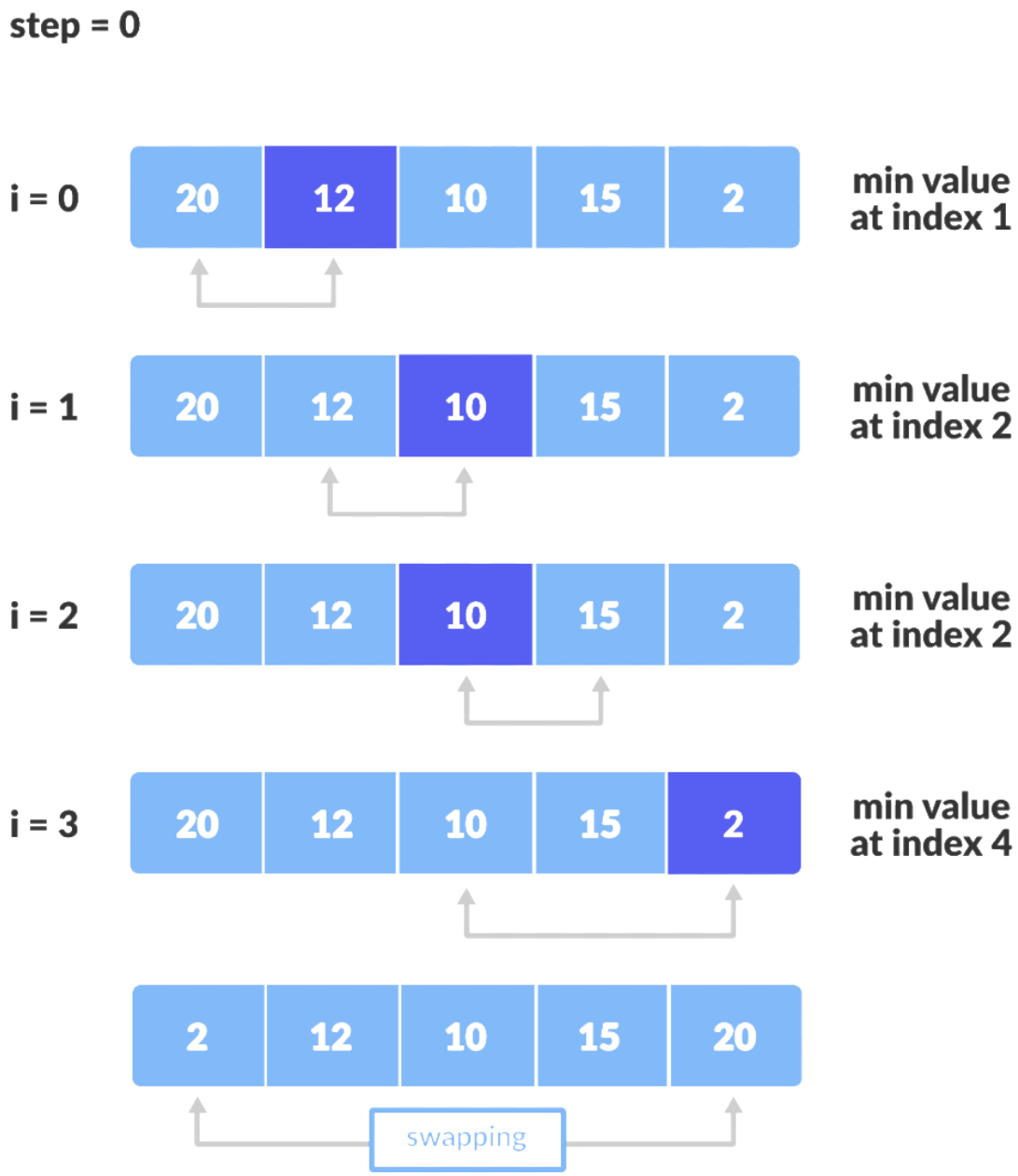
In [5]:
```
def  insertionSort(array):
    for step in range(1, len(array)):
        key = array[step]
        j = step - 1
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j = j - 1
        array[j + 1] = key
```

In [6]:
```
data  = [11,22,32,40,51,60,74,80,95,90,85,76]
insertionSort(data)
print('Sorted Array in Ascending Order is:')
print(data)
```

```
Sorted Array in Ascending Order is:
[11, 22, 32, 40, 51, 60, 74, 76, 80, 85, 90, 95]
```

Selection sort

The selection sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires n”1n”1passes to sort n items, since the final item must be in place after the (n”1)(n”1) last pass.



The following is the implementation of the binary search algorithm, which is used to find the position of a given element in a sorted array. Here's how the algorithm works:

1. Initialize the start and end indices of the array.
2. Compute the middle index of the array.
3. Compare the middle element with the given element `x`.

4. If the middle element is equal to `x`, return its index in the array.
5. If the middle element is less than `x`, the element must be in the right half of the array. Update the start index to the right of the middle index.
6. If the middle element is greater than `x`, the element must be in the left half of the array. Update the end index to the left of the middle index.
7. Repeat steps 2-6 until the element is found or the search space is exhausted.

implementation of the Selection Sort algorithm in Python. Here's how it works:

8. The function takes an unsorted array as input.
9. It starts a loop that iterates over each element in the array, starting at the first element (index 0).
10. For each element, it finds the index of the minimum value in the rest of the array (i.e., the elements to the right of the current element).
11. If the minimum value is smaller than the current element, the function swaps the positions of the minimum value and the current element.
12. This process continues until all elements have been processed and the array is sorted in ascending order.

In [7]:
```
def  SelectionSort(array):
    size=len(array)
    for step in range(size):
        min_idx = step
        for i in range(step + 1, size):
            if array[i] < array[min_idx]:
                min_idx = i
        (array[step], array[min_idx]) = (array[min_idx], array[step])
    print(array)
```

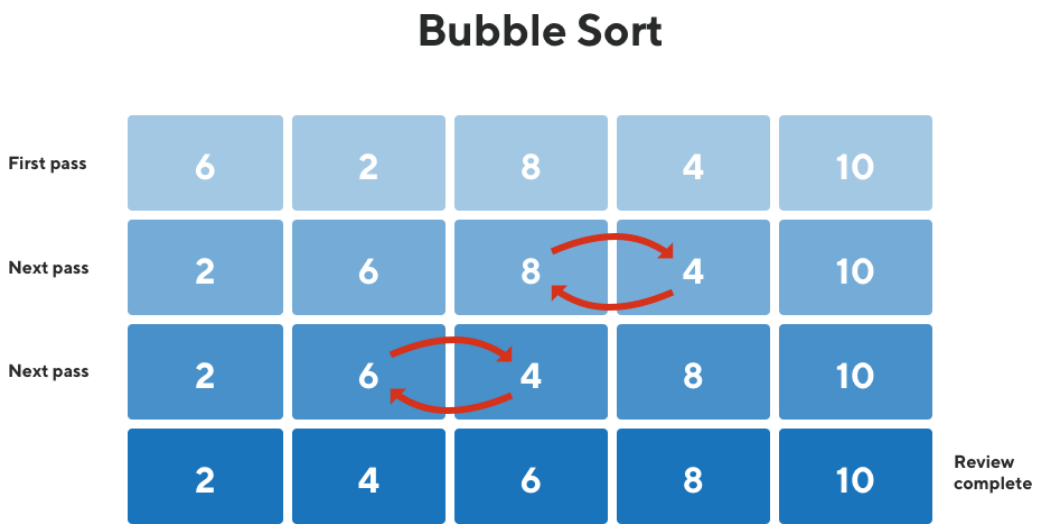
In [8]:
```
SelectionSort([91,222,33,31,43,5,6])
```

```
[5, 6, 31, 33, 43, 91, 222]
```

Bubble sort

The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item “bubbles” up to the location where it belongs.

A Bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These “wasted” exchange operations are very costly. However, because the bubble sort makes passes through the entire unsorted portion of the list, it has the capability to do something most sorting algorithms cannot. In particular, if during a pass there are no exchanges, then we know that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted. This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop



The following is a basic implementation of the Bubble Sort algorithm. Here's how it works:

1. The outer loop iterates through the array n times, where n is the length of the array.
2. The inner loop iterates through the array from index 0 to index n - i - 1, where i is the current iteration of the outer loop. This is because after each iteration of the outer loop, the largest element will "bubble up" to the end of the array, so we don't need to compare it again.
3. If the current element is greater than the next element, we swap them.
4. After all iterations are complete, the array is sorted in ascending order.

In [26]:
```
def bubbleSort(array):
  for i in range(len(array)):
    for j in range(0, len(array) - i - 1):
      if array[j] > array[j + 1]:
        temp = array[j]
        array[j] = array[j+1]
        array[j+1] = temp
```

In [27]:
```
data = [11,22,32,40,51,60,74,80,95,90,85,76]  #SAI LIKHITH 122010405056
bubbleSort(data)
print('Sorted Array in Ascending Order is:')
print(data)
```

```
Sorted Array in Ascending Order is:
[11, 22, 32, 40, 51, 60, 74, 76, 80, 85, 90, 95]
```

Introduction to Linked Lists

A linked list is a data structure used to store a collection of elements or nodes.

Each node in a linked list contains two fields: data and a pointer to the next node in the list. The first node in the list is called the head, while the last node is called the tail.

The main advantage of a linked list over an array is that a linked list can be dynamically resized without having to allocate a new block of memory.

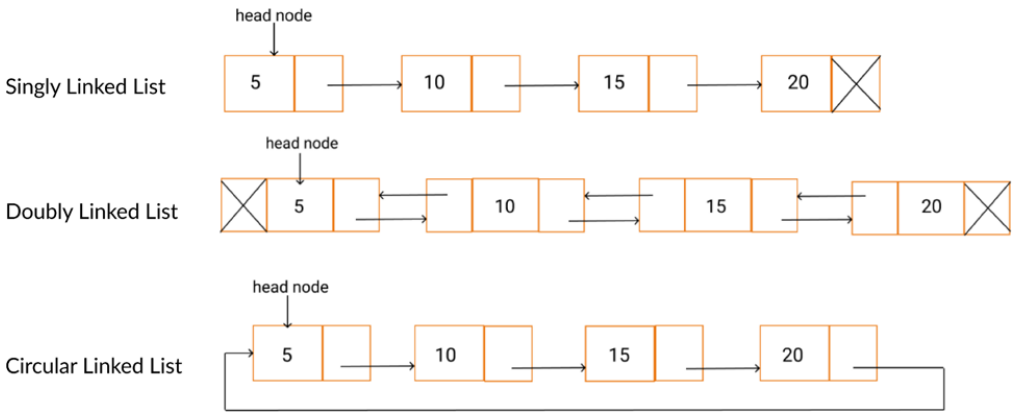
However, linked lists have slower access times than arrays since elements are not stored contiguously in memory.

There are several types of linked lists, including singly linked lists, doubly linked lists, and circular linked lists. In a singly linked list, each node only has a pointer to the next node in the list. In a doubly linked list, each node has a pointer to both the next and the previous node in the list. In a circular linked list, the last node in the list has a pointer to the first node, creating a loop.

Linked lists are commonly used in many programming applications, such as implementing stacks, queues, and hash tables. They are also frequently used in computer science education to teach concepts such as recursion, pointers, and dynamic memory allocation.

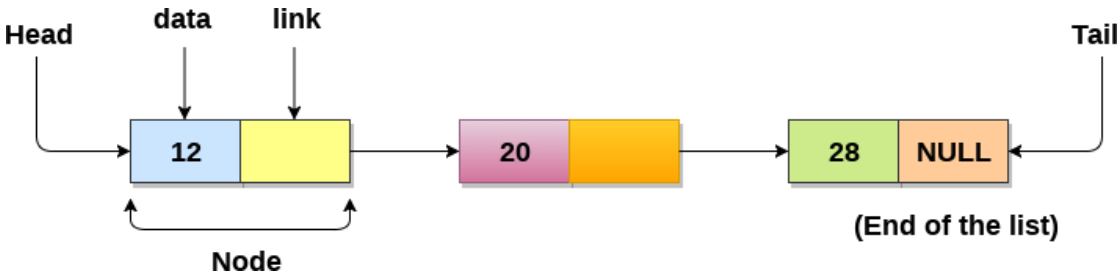
Types of LinkedLists:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List



Single linked list

A singly linked list is a linear data structure in which each node contains a data element and a pointer to the next node in the list. It only allows for traversal of the list in one direction (forward), from the head to the tail.



Singly linked list is a data structure that consists of a sequence of nodes. Each node contains a data element and a reference (or link) to the next node in the sequence. The first node is called the head and the last node is called the tail.

Here are some key concepts and terminology associated with singly linked lists:

- Node: A node is a basic unit of a linked list. It consists of a data element and a reference to the next node in the list.
- Head: The head is the first node in the list. It is the entry point into the list.
- Tail: The tail is the last node in the list. It has a null reference to indicate the end of the list.
- Linked List Length: The length of a linked list is the number of nodes it contains.
- Traversal: Traversing a linked list means visiting each node in the list exactly once. This is typically done using a loop that iterates over the nodes in the list.
- Insertion: Inserting a new node into a linked list involves creating a new node, updating the links between adjacent nodes, and updating the head or tail if necessary.
- Deletion: Deleting a node from a linked list involves updating the links between adjacent nodes and freeing the memory used by the deleted node.

Singly linked lists are commonly used in programming because they provide efficient insertion and deletion operations compared to other data structures like arrays. However, they have slower random access times because they cannot be indexed directly like an array.

In [28]:
```
class Node:
    def__init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def__init_(self):
        self.head = None

    def  print_list(self):
        temp = self.head
        while temp:
            print(temp.data)
```

```
            temp = temp.next

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node

    def insert_at_position(self, data, position):
        if position == 0:
            self.insert_at_beginning(data)
            return
        new_node = Node(data)
        temp = self.head
        for i in range(position - 1):
            if temp is None:
                raise Exception("Position out of range")
            temp = temp.next
        new_node.next = temp.next
        temp.next = new_node

    def delete_at_position(self, position):
        if self.head is None:
            raise Exception("List is empty")
        temp = self.head
        if position == 0:
            self.head = temp.next
            temp = None
            return
        for i in range(position - 1):
            if temp is None:
                raise Exception("Position out of range")
            temp = temp.next
        if temp is None or temp.next is None:
            raise Exception("Position out of range")
        next_node = temp.next.next
        temp.next = None
        temp.next = next_node
```

Explanation:

- The `Node` class represents a node in the linked list. Each node has two attributes: `data`, which stores the value of the node, and `next`, which points to the next node in the list.
- The `LinkedList` class represents the linked list itself. It has a single attribute `head`, which points to the first node in the list.
- The `print_list` method is used to print the contents of the linked list. It starts at the head node and iterates through the list, printing the data value of each node.
- The `insert_at_beginning` method inserts a new node at the beginning of the list. It creates a new node with the given data, sets its `next` attribute to the current head node, and updates the head node to point to the new node.

- The `insert_at_end` method inserts a new node at the end of the list. It creates a new node with the given data, and iterates through the list until it finds the last node. It then sets the `next` attribute of the last node to the new node.
- The `insert_at_position` method inserts a new node at a specified position in the list. It creates a new node with the given data, and iterates through the list until it finds the node at the specified position. It then sets the `next` attribute of the new node to the next node in the list, and sets the `next` attribute of the current node to the new node.
- The `delete_at_position` method deletes a node at a specified position in the list. It iterates through the list until it finds the node at the specified position, and sets the `next` attribute of the previous node to the next node in the list, effectively removing the current node from the list. If

In [29]:
```
# Create a new linked list
llist = LinkedList()

# Insert some nodes
llist.insert_at_beginning(5)
llist.insert_at_beginning(7)
llist.insert_at_end(10)
llist.insert_at_position(8,  2)

# Print the linked list
llist.print_list()

# Delete some nodes
llist.delete_at_position(2)

# Print the linked list again
llist.print_list()
```

```
7
5
8
10
7
5
10
```

Double linked list

A doubly linked list is a type of linked list where each node contains two links, one to the previous node and one to the next node in the list. This allows for more efficient traversal in both directions, as each node can be accessed from both the previous and next nodes.

One important concept in doubly linked lists is the use of a sentinel node. A sentinel node is a special node that is added to the beginning and/or end of the list and serves as a marker for the start and end of the list. This can simplify certain operations, such as inserting or deleting nodes at the beginning or end of the list.

Another important concept is the use of pointers to maintain the links between nodes. Each node contains two pointers, one to the previous node and one to the next node. These pointers must be updated whenever a node is added, removed, or moved within the list. It is important to properly manage these pointers to avoid memory leaks or other issues.

In [9]:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    # Add a new node to the beginning of the list
    def insert_at_start(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    # Add a new node to the end of the list
    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
            new_node.prev = current

    # Delete a node from the list
    def delete_node(self, data):
```

```
        current = self.head
        if current is None:
            print("List is empty")
            return
        elif current.data == data:
            self.head = current.next
            current.next.prev = None
            return
        else:
            while current.next:
                if current.next.data == data:
                    temp = current.next
                    current.next = temp.next
                    if temp.next:
                        temp.next.prev = current
                    temp = None
                    return
                current = current.next
            print("Data not found")

    # Display the list
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print()
```

In this implementation, the `Node` class represents a node in the doubly linked list, with each node containing a `data` attribute, as well as `prev` and `next` attributes that point to the previous and next nodes in the list, respectively.

The `DoublyLinkedList` class represents the doubly linked list itself, with a `head` attribute that points to the first node in the list. The class provides four methods:

- `insert_at_start`: Adds a new node containing the given `data` to the beginning of the list.
- `insert_at_end`: Adds a new node containing the given `data` to the end of the list.
- `delete_node`: Removes the node containing the given `data` from the list.
- `display`: Displays the contents of the list in order.

In addition, each of the `insert_at_start` and `insert_at_end` methods checks if the list is empty, and if so, sets the new node to be the head of the list. The `delete_node` method also checks if the node to be deleted is the head of the list, and updates the `prev` attribute of the new head node accordingly. If the node to be deleted is not the head, the method iterates through the list until it finds the node with the given `data` value and removes it from the list by updating the `prev` and `next` attributes of the surrounding nodes. If the node is not found, the method prints a message indicating that the data was not found. Finally, the `display` method simply iterates through the list and prints out the data in each node.

In [10]:
```
# Creating a new doubly linked list
dll = DoublyLinkedList()

# Inserting nodes
dll.insert_at_start(5)
dll.insert_at_start(3)
```

```
dll.insert_at_end(7)
dll.insert_at_end(9)

# Displaying the list
dll.display()

# Deleting nodes
dll.delete_node(5)
dll.delete_node(9)

# Displaying the list again
dll.display()
```

```
3 5 7 9
3 7
```

Circular linked list

A circular linked list is a type of linked list where the last node of the list points back to the first node, creating a circular structure. This data structure has several important properties that make it useful in certain applications:

1. Circular linked lists can be traversed indefinitely: because the last node points back to the first node, it is possible to traverse the entire list starting from any node and ending up back at the same node. This makes circular linked lists useful in certain algorithms where it is necessary to repeatedly process the nodes of a list in a circular fashion.

2. Insertion and deletion are efficient: in a singly linked list, inserting or deleting a node requires updating the next pointer of the previous node. In a circular linked list, however, the last node points back to the first node, so insertion and deletion can be done simply by updating the next pointer of a single node.

3. Circular linked lists can be used to implement circular buffers: a circular buffer is a data structure that allows data to be stored and retrieved in a circular fashion. Circular linked lists can be used to implement circular buffers, since the last node of the list can be thought of as wrapping around to the first node.

4. Memory management can be simplified: in a circular linked list, all nodes are linked together, so it is possible to allocate a contiguous block of memory for all nodes. This can simplify memory management in certain applications, since it is not necessary to allocate individual blocks of memory for each node.

circular linked lists are a useful data structure that can simplify certain algorithms and applications. However, they are not appropriate for all use cases, and care must be taken to ensure that circular dependencies are avoided, since these can lead to infinite loops and other problems.

In [30]:
```
class Node:
    def__init__(self, data):
        self.data = data
        self.next = None

class  CircularLinkedList:
    def__init_(self):
        self.head = None
        self.tail = None

    # Method to append a node at the end of the list
```

```
    def append(self, data):
        new_node = Node(data)
        if self.head is None:  # List is empty
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.next = self.head
            self.tail = new_node

    # Method to insert a node at the beginning of the list
    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:  # List is empty
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.tail.next = new_node
            self.head = new_node

    # Method to remove a node from the list
    def remove(self, data):
        if self.head is None:  # List is empty
            return

        # Special case for the head node
        if self.head.data == data:
            self.tail.next = self.head.next
            self.head = self.head.next
            return

        curr = self.head
        while curr.next != self.head:
            if curr.next.data == data:
                curr.next = curr.next.next
                return
            curr = curr.next

    # Method to print the list
    def print_list(self):
        if self.head is None:  # List is empty
            return
        curr = self.head
        while curr.next != self.head:
            print(curr.data, end=' -> ')
            curr = curr.next
        print(curr.data, end=' -> ')
        print('HEAD')
```

Here are the explanations of the methods:

- `__init_(self)` : Initializes an empty circular linked list with `head` and `tail` pointers set to `None` .
- `append(self, data)` : Appends a node with the given data at the end of the list. If the list is empty, sets both `head` and `tail` pointers to the new node. Otherwise, sets the `next` pointer of the current tail node to the new node, and the `next` pointer of the new node to the `head` , and sets `tail` pointer to the new node.
- `prepend(self, data)` : Prepends a node with the given data at the beginning of the list. If the list is empty, sets both `head` and `tail` pointers to the new node.

Otherwise, sets the `next` pointer of the new node to the current `head` node, the `next` pointer of the current `tail` node to the new node, and sets `head` pointer to the new node.

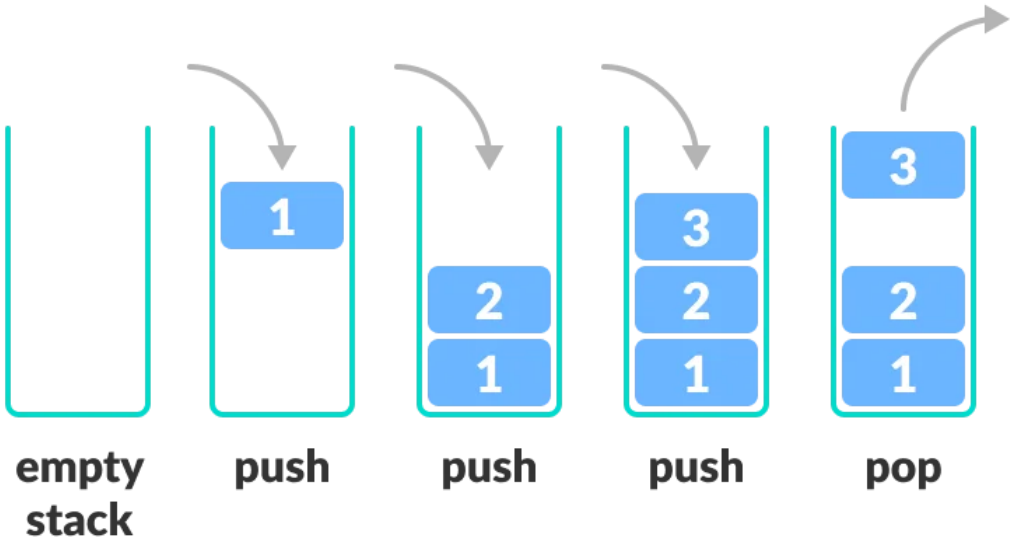
- `remove(self, data)` : Removes the first node with the given data from the list. If the list is empty, does nothing. If the node to be removed is the head node, sets the `next` pointer of the current tail node to the `next` pointer of the head node, and sets the `head` pointer to the `next` node. Otherwise, iterates over the list and removes the node by setting the `next` pointer of the previous node to the `next` pointer of the current node.
- `print_list(self)` :

In [31]:
```
# Create a circular linked list and test its methods
clist = CircularLinkedList()

# Append some nodes to the list
clist.append(5)
clist.append(10)
clist.append(15)
clist.append(20)

# Prepend a node to the list
clist.prepend(2)

# Print the list
clist.print_list()  # Expected output: 2 -> 5 -> 10 -> 15 -> 20 -> HEAD

# Remove a node from the list
clist.remove(15)

# Print the list again
clist.print_list()  # Expected output: 2 -> 5 -> 10 -> 20 -> HEAD
```

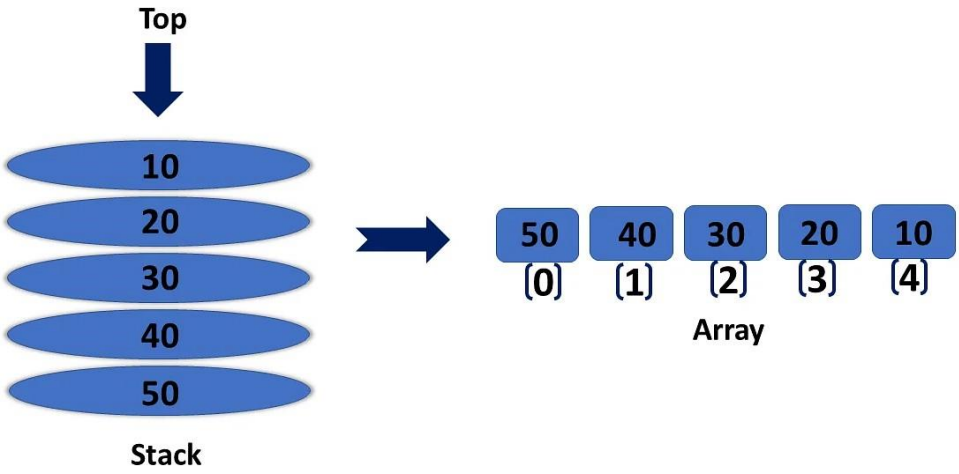
```
2 -> 5 -> 10 -> 15 -> 20 -> HEAD
2 -> 5 -> 10 -> 20 -> HEAD
```

Introduction to Stacks

Stacks in Data Structures is a linear type of data structure that follows the LIFO (Last-In-First-Out) principle and allows insertion and deletion operations from one end of the stack data structure, that is top



Implementation of Stacks using Arrays



In [32]:
```
# BY USING PRE-DEFINED FUNCTIONS:

class Stack:
    def__init__(self):
        self.stack = []

    def push(self, data):
        self.stack.append(data)
        print("Element {} is pushed into the Stack".format(data))

    def Pop(self):
        if self.stack:
            print("Popped item is :",self.stack.pop())
        else:
            return "Stack is Empty"

    def peep(self):
        if self.stack:
            print("The Top most element is : ",self.stack[-1])
        else:
            return "Stack is Empty"

    def is_empty(self):
        return len(self.stack) == 0

    def__str__(self):
        return str(self.stack)

s = Stack()
s.push(5)
s.push(0)
s.push(5)
s.push(6)
print(s)          # [5,0,5,6]
s.peep()          # 6 Since it is the last data enetered
s.Pop()           # 6 is popped Out
s.Pop()           # 5 is popped Out
```

```
s.Pop()          # 0 is popped Out
s.Pop()          # 5 is popped Out
s.is_empty()     # True as there are no data elements in the Stack
s.push(5056)     # 5 is pushed into the stack
s.is_empty()     # False, as 5 is present in the Stack
print(s)         # Prints [5], as it is entered into the stack
```

```
Element 5 is pushed into the Stack
Element 0 is pushed into the Stack
Element 5 is pushed into the Stack
Element 6 is pushed into the Stack
[5, 0, 5, 6]
The Top most element is : 6
Popped item is : 6
Popped item is : 5
Popped item is : 0
Popped item is : 5
Element 5056 is pushed into the Stack
[5056]
```

This code defines a class `Stack` which is used to implement a stack data structure in Python. The class has four methods: `push()`, `Pop()`, `peep()`, and `is_empty()`, and an `__init__()` method to initialize the stack.

The `push()` method takes an input parameter `data`, which is added to the top of the stack. The `Pop()` method removes the top element from the stack if it is not empty, otherwise returns the message "Stack is Empty". The `peep()` method returns the top-most element of the stack, if it is not empty, otherwise returns "Stack is Empty". The `is_empty()` method checks whether the stack is empty or not.

The code creates an instance `s` of the `Stack` class and pushes some data elements into the stack using the `push()` method. Then, it prints the stack using the `__str__()`

method. The `peep()` method is called to print the top-most element of the stack. The `Pop()` method is called four times to remove the elements from the stack. Finally, it calls the `is_empty()` method to check if the stack is empty, and again adds an element to the stack using the `push()` method.

The output shows that the data elements are added to the stack in the order of 5, 0, 5, and 6. The `peep()` method returns 6, which is the last element added to the stack. The `Pop()` method is called four times to remove the elements from the stack. After removing all the elements, the `is_empty()` method returns `True`. Finally, a new element 5056 is added to the stack, and the `is_empty()` method returns `False`.

In [33]:
```
# WITHOUT USING PRE-DEFINED FUNCTIONS:

class Stack:
    def __init__(self):
        self.stack = []

    def push(self, data):
        self.stack += [data]
        print("Element {} is pushed into the Stack".format(data))

    def Pop(self):
        if self.stack:
            data = self.stack[-1]
            self.stack = self.stack[:-1]
```

```
            print("Popped item is :",data)
        else:
            print("Stack is Empty")

    def peep(self):
        if self.stack:
            print("The Top most element is : ",self.stack[-1])
        else:
            print("Stack is Empty")

    def is_empty(self):
        return len(self.stack) == 0

    def__str__(self):
        return str(self.stack)


s = Stack()
s.push(5)
s.push(0)
s.push(5)
s.push(6)
print(s)          # [5,0,5,6]
s.peep()          # 6 Since it is the last data enetered
s.Pop()           # 6 is popped Out
s.Pop()           # 5 is popped Out
s.Pop()           # 0 is popped Out
s.Pop()           # 5 is popped Out
s.is_empty()      # True as there are no data elements in the Stack
s.push(5056)      # 5 is pushed into the stack
s.is_empty()      # False, as 5 is present in the Stack
print(s)          # Prints [5], as it is entered into the stack
```

```
Element 5 is pushed into the Stack
Element 0 is pushed into the Stack
Element 5 is pushed into the Stack
Element 6 is pushed into the Stack
[5, 0, 5, 6]
The Top most element is :  6
Popped item is : 6
Popped item is : 5
Popped item is : 0
Popped item is : 5
Element 5056 is pushed into the Stack
[5056]
```

This code defines a class `Stack` which is used to implement a stack data structure in Python. The class has four methods: `push()`, `Pop()`, `peep()`, and `is_empty()`, and an `__init__()` method to initialize the stack.

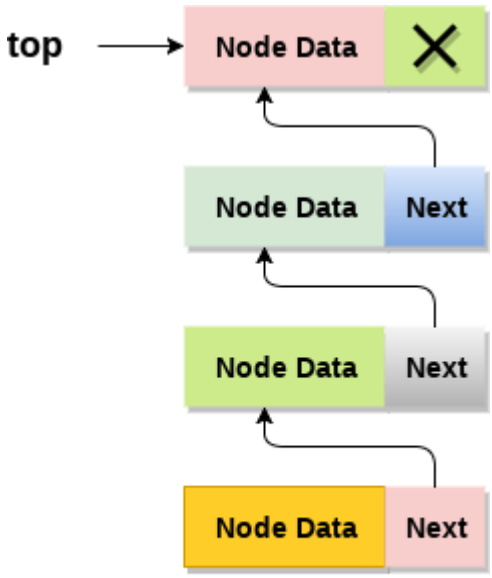
The `push()` method takes an input parameter `data`, which is added to the top of the stack using the `+=` operator. The `Pop()` method removes the top element from the stack by getting the last element of the list and then removing it using the slicing technique. If the stack is empty, it prints the message "Stack is Empty". The `peep()` method returns the top-most element of the stack using the indexing technique. If the stack is empty, it prints the message "Stack is Empty". The `is_empty()` method checks whether the stack is empty or not by checking the length of the stack.

The code creates an instance `s` of the `Stack` class and pushes some data elements into the stack using the `push()` method. Then, it prints the stack using the `__str__()`

method. The `peep()` method is called to print the top-most element of the stack. The `Pop()` method is called four times to remove the elements from the stack. Finally, it calls the `is_empty()` method to check if the stack is empty, and again adds an element to the stack using the `push()` method.

The output shows that the data elements are added to the stack in the order of 5, 0, 5, and 6. The `peep()` method returns 6, which is the last element added to the stack. The `Pop()` method is called four times to remove the elements from the stack. After removing all the elements, the `is_empty()` method returns `True`. Finally, a new element 5056 is added to the stack, and the `is_empty()` method returns `False`.

Implementation of Stacks usingLinked Lists



Stack

In [34]:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = Node(data)
        if self.top is None:
            self.top = new_node
        else:
            new_node.next = self.top
            self.top = new_node

    def pop(self):
        if self.top is None:
            print("Stack  Underflow")
        else:
            popped = self.top.data
            self.top = self.top.next
            return popped

    def peek(self):
        if self.top is None:
            print("Stack  Underflow")
        else:
            return  self.top.data

    def  is_empty(self):
        return self.top is None
```

```
s = Stack()

s.push(5)
s.push(0)
s.push(5)
s.push(6)
s.pop()
s.pop()
print(s.peek())
print(s.is_empty())
s.pop()
s.pop()
print(s.is_empty())

s.push(5)
s.push(0)
s.push(5)
s.push(6)
s.pop()
s.pop()
print(s.peek())
print(s.is_empty())
s.pop()
s.pop()
print(s.is_empty())  # should print True
```

```
0
False
True
0
False
True
```

This code defines a `Stack` class that implements a stack data structure using a singly linked list. The `Stack` class has three methods: `push`, `pop`, `peek`, and `is_empty`.

The `push` method adds a new element to the top of the stack. It creates a new `Node` object with the given data and sets the `next` field of the new node to the current `top` node. Then it sets the `top` to the new node.

The `pop` method removes and returns the top element of the stack. It first checks if the stack is empty. If the stack is not empty, it sets the `popped` variable to the `data` of the `top` node, sets the `top` to the `next` node, and returns `popped`.

The `peek` method returns the top element of the stack without removing it. It first checks if the stack is empty. If the stack is not empty, it returns the `data` of the `top` node.

The `is_empty` method returns `True` if the stack is empty, `False` otherwise.

The code creates an instance of the `Stack` class and performs some operations on it. It pushes four elements onto the stack, pops two elements from the stack, peeks at the top element, checks if the stack is empty, and pops the remaining elements from the stack. It then performs the same operations again to demonstrate that the stack is empty at the end.

Overall, this code implements a simple stack data structure using a singly linked list, allowing for efficient addition and removal of elements from the top of the stack.

Introduction to Queues

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. In a queue, the elements are inserted at one end, called the rear or tail, and removed from the other end, called the front or head. This makes a queue resemble a real-world queue, like people waiting in a line for a ticket, where the first person to enter the line is the first to leave it.

The basic operations that can be performed on a queue are:

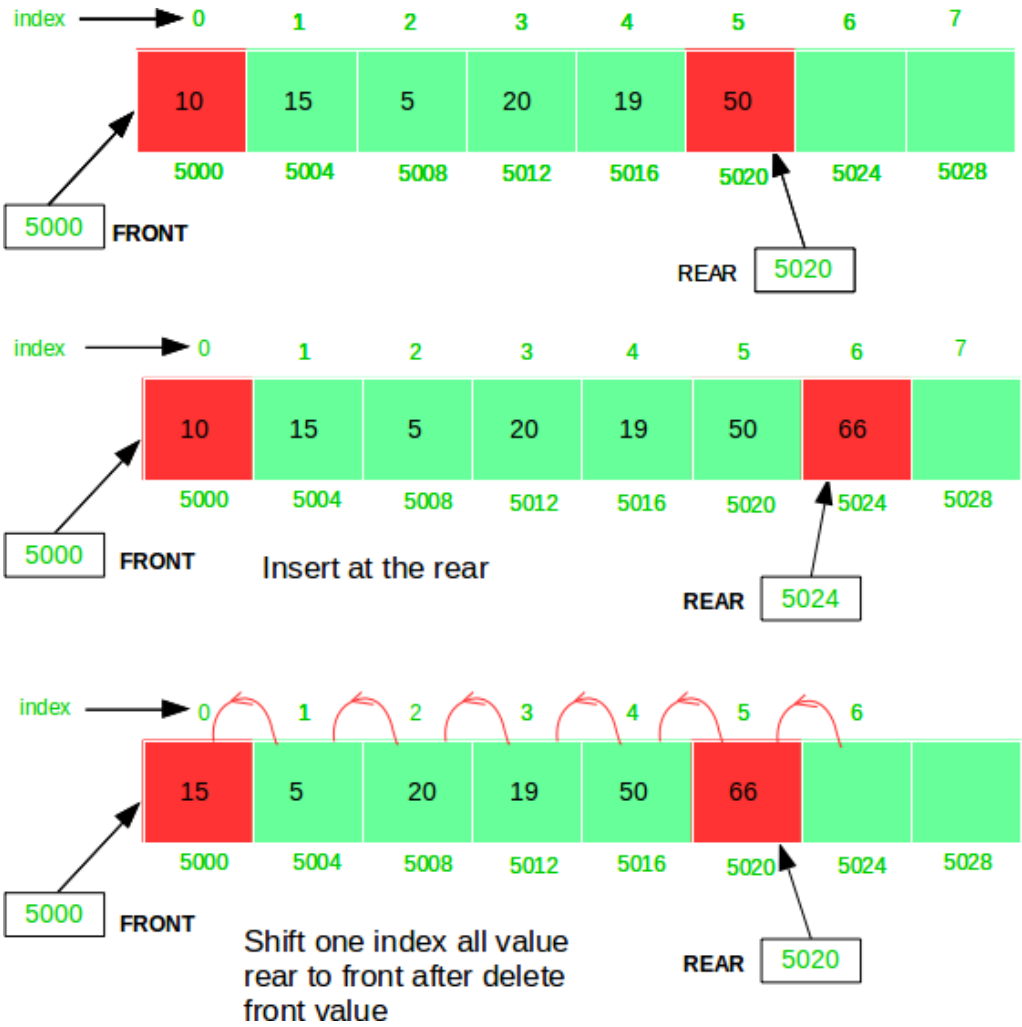
1. Enqueue: Adds an element to the rear of the queue.
2. Dequeue: Removes an element from the front of the queue.
3. Peek/Front: Returns the element at the front of the queue without removing it.
4. Is Empty: Checks whether the queue is empty or not.

Some applications of queues include job scheduling, network data packets, and printer queues.

Queues can be implemented using arrays, linked lists, or other data structures. When implementing a queue, it is important to ensure that elements are added to the rear and removed from the front to maintain the correct order of the queue.

Queue Data Structure

Implementation of Queues using Arrays



In [11]:
```
class Queue:
    def___init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def  dequeue(self):
        if  self.is_empty():
            return None
        return  self.items.pop(0)

    def  is_empty(self):
        return len(self.items) == 0

    def size(self):
        return  len(self.items)

    def front(self):
        if  self.is_empty():
            return None
        return  self.items[0]
```

The `Queue` class has a single instance variable `items`, which is an array that stores the elements of the queue.

The `enqueue` method adds an element to the rear of the queue by appending it to the end of the `items` array.

The `dequeue` method removes and returns an element from the front of the queue. If the queue is empty, it returns `None`. It does this by checking if the queue is empty and then using the `pop` method on the `items` array with an index of 0, which removes and returns the first element.

The `is_empty` method checks whether the queue is empty by checking the length of the `items` array.

The `size` method returns the number of elements in the queue by returning the length of the `items` array.

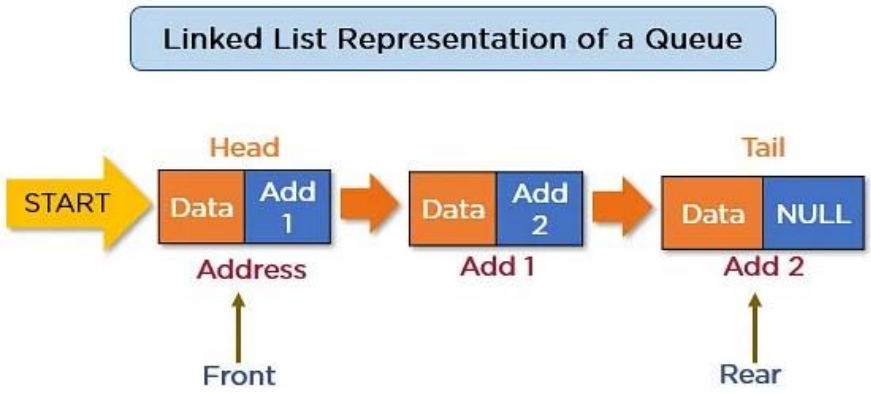
The `front` method returns the element at the front of the queue without removing it. If the queue is empty, it returns `None`. It does this by checking if the queue is empty and then accessing the first element of the `items` array with an index of 0.

Overall, this implementation of a queue using arrays in Python is simple and efficient, but it has some limitations. One limitation is that if the `items` array becomes full, it cannot store any more elements, even if the queue is not yet full. Another limitation is that removing an element from the front of the queue using `pop(0)` can be inefficient for large queues, since it requires shifting all the remaining elements in the array one position to the left.

In [12]:
```
q = Queue()

# Add elements to the queue
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)

# Get the element at the front of the queue
print("Front of queue:", q.front())

# Remove the first element from the queue
print("Removed from queue:", q.dequeue())

# Check if the queue is empty
print("Queue is empty:", q.is_empty())

# Get the size of the queue
print("Queue size:", q.size())

# Add another element to the queue
q.enqueue(4)

# Remove all elements from the queue
while not q.is_empty():
    print("Removed from queue:", q.dequeue())
```

```
Front of queue: 1
Removed from queue: 1
Queue is empty: False
Queue size: 2
Removed from queue: 2
Removed from queue: 3
Removed from queue: 4
```

Implementation of Queues using Linked Lists



In [13]:
```
class Node:
    def___init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def___init_(self):
        self.head = None
        self.tail = None
        self.count = 0

    def is_empty(self):
        return self.head is None

    def size(self):
        return  self.count

    def enqueue(self, item):
        node = Node(item)
        if self.tail:
            self.tail.next = node
        else:
            self.head = node
        self.tail = node
        self.count += 1

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        item = self.head.data
        self.head = self.head.next
        if not self.head:
            self.tail = None
        self.count -= 1
        return item

q = Queue()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
```

```
print(q.size())
print(q.dequeue())
print(q.dequeue())
print(q.size())
q.enqueue(40)
print(q.dequeue())
print(q.dequeue())
print(q.is_empty())
```

```
3
10
20
1
30
40
True
```

This code implements a queue data structure using a linked list. The `Queue` class has three instance variables: `head`, `tail`, and `count`. The `head` variable points to the first node in the linked list, the `tail` variable points to the last node in the linked list, and the `count` variable keeps track of the number of nodes in the queue.

The `is_empty` method checks if the queue is empty by checking if the `head` variable is `None`. The `size` method returns the number of nodes in the queue by returning the value of the `count` variable.

The `enqueue` method adds an item to the end of the queue by creating a new `Node` object with the given item and setting the `next` attribute of the current `tail` node to the new node. If the queue is empty, the `head` variable is set to the new node. The `tail` variable is then updated to point to the new node, and the `count` variable is incremented.

The `dequeue` method removes and returns the item at the front of the queue. It raises an exception if the queue is empty. Otherwise, it gets the data of the first node in the linked list and sets the `head` variable to point to the next node in the list. If the `head` variable is `None` after removing the first node, the `tail` variable is also set to `None`. Finally, the `count` variable is decremented.

The driver code creates a new queue object `q` and enqueues three items using the `enqueue` method. It then prints the size of the queue, dequeues two items using the `dequeue` method, prints the new size of the queue, enqueues another item, dequeues the remaining items using the `dequeue` method, and checks if the queue is empty.

Priority Queue

A priority queue is a data structure similar to a queue or stack, but each element in the queue has a priority assigned to it. When elements are added to the queue, they are placed in order based on their priority, and the element with the highest priority is always at the front of the queue.

Priority queues are useful in situations where items need to be processed in a specific order based on their priority. For example, in an operating system, processes may be placed in a priority queue based on their importance or urgency, with higher priority processes being executed before lower priority ones.

There are several ways to implement a priority queue, including using arrays or linked lists with sorted elements or using heaps. A binary heap is a common data structure used to implement a priority queue, as it provides efficient insertion and removal of elements while maintaining the heap property that the highest priority element is always at the front.

Operations that can be performed on a priority queue include inserting an element with a priority, removing the element with the highest priority, checking if the priority queue is empty, and peeking at the highest priority element without removing it. Some priority queues may also allow updating the priority of an element that is already in the queue.

Priority queues provide a useful way to manage and process items in a specific order based on their priority, making them a valuable tool in many computer science applications.

In Python, we can implement a priority queue using the heapq module. The heapq module provides functions for creating and manipulating heap data structures, which can be used to implement a priority queue.

Here is an example implementation of a priority queue using heapq:

In [35]:
```python
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

In this implementation, each item in the priority queue is a tuple containing three elements: the priority (which is negated so that higher priorities are given lower values), an index to maintain the order of insertion, and the item itself. The _index variable is used to ensure that items with the same priority are popped in the order that they were added.

To insert an item into the priority queue, we use the heappush function from the heapq module, passing in the tuple containing the item, priority, and index. To remove an item from the priority queue, we use the heappop function, which removes and returns the item with the highest priority.

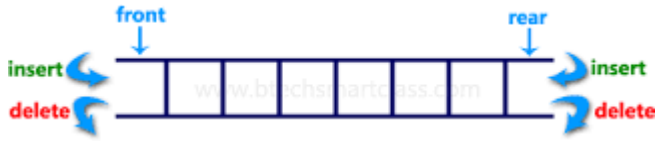
Here is an example usage of the priority queue:

In [36]:
```
q = PriorityQueue()
q.push('Task 1', 5)
q.push('Task 2', 1)
q.push('Task 3', 3)

print(q.pop()) # Output: 'Task 2'
print(q.pop()) # Output: 'Task 3'
print(q.pop()) # Output: 'Task 1'
```

```
Task 1
Task 3
Task 2
```

In this example, we create a PriorityQueue object, and push three tasks into the queue with different priorities. We then pop items from the queue, which are returned in order based on their priorities.

Double Ended Queues



A double-ended queue (deque) is a data structure that allows insertion and removal of elements from both ends. It can be thought of as a hybrid of a stack and a queue. It is also sometimes called a head-tail linked list.

A deque can be implemented using various data structures, such as an array, a linked list, or a dynamic array. Some common operations supported by a deque include:

- insertFront(x): Insert an element x at the front of the deque.
- insertLast(x): Insert an element x at the back of the deque.
- deleteFront(): Remove and return the element at the front of the deque.
- deleteLast(): Remove and return the element at the back of the deque.
- getFront(): Return the element at the front of the deque without removing it.
- getLast(): Return the element at the back of the deque without removing it.
- isEmpty(): Check if the deque is empty.
- size(): Return the number of elements in the deque.

Deques can be used in a variety of applications, such as implementing algorithms like BFS (breadth-first search) and maintaining a sliding window of elements in an array or a stream of data.

In Python, a deque can be implemented using the collections module. The deque class provides an implementation of a double-ended queue, with all the above-mentioned operations available.

Here is an example usage of a deque in Python:

In [14]:
```
from collections import deque

# Create an empty deque
d = deque()

# Insert elements at the back
d.append(1)
d.append(2)
d.append(3)

# Insert elements at the front
d.appendleft(0)
d.appendleft(-1)
d.appendleft(-2)

# Remove elements from the front
```

```
print(d.popleft())
print(d.popleft())

# Remove elements from the back
print(d.pop())
print(d.pop())

# Check the front and back elements without removing them
print(d[0])
print(d[-1])

# Check if the deque is empty and its size
print(d)
print(len(d))
```

```
-2
-1
3
2
0
1
deque([0, 1])
2
```

This code demonstrates the use of a double-ended queue in Python using the `deque` class from the `collections` module.

First, an empty deque is created using the `deque()` function. Elements can be added to the deque from the back using the `append()` function, and from the front using the `appendleft()` function.

To remove elements from the deque, the `popleft()` function is used to remove and return the leftmost element, while the `pop()` function is used to remove and return the rightmost element.

To access the front and back elements of the deque without removing them, we can use the square bracket indexing operator. For example, `d[0]` returns the first element (leftmost), while `d[-1]` returns the last element (rightmost).

Finally, the code prints the deque to check its contents and length using the `print()` and `len()` functions, respectively.

Trees

A tree is a widely used abstract data type in computer science, particularly in the field of computer science algorithms and data structures. A tree data structure is a collection of nodes that are arranged in a hierarchical fashion. Each node in a tree can have zero or more child nodes, except the root node, which has no parent.

The following are the essential terminologies associated with trees:

- Root: The topmost node of a tree.
- Parent: A node that has one or more child nodes.
- Child: A node that has a parent node.
- Sibling: Nodes that share the same parent node.
- Leaf: A node that does not have any child nodes.
- Internal node: A node that has one or more child nodes.
- Depth: The number of edges from the root to a node.
- Height: The number of edges from a node to the deepest leaf in the subtree rooted at that node.

There are several types of trees, some of which include:

1. Binary Tree: A binary tree is a tree in which each node can have at most two children.
2. Binary Search Tree (BST): A binary search tree is a binary tree in which for each node, all nodes in the left subtree are less than the node, and all nodes in the right subtree are greater than the node.
3. AVL Tree: AVL tree is a self-balancing binary search tree in which the difference between the heights of the left and right subtrees of a node is at most one.
4. Red-Black Tree: A red-black tree is a binary search tree in which each node is either red or black and satisfies the following properties:
    - The root node is always black.
    - Every leaf node is black.
    - If a node is red, then its children must be black.
    - Every simple path from a node to a descendant leaf contains the same number of black nodes.
5. B-Tree: A B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
6. Trie: A trie is a tree data structure used for efficient searching of strings. Each node in a trie represents a prefix of one or more strings, and each edge represents a character.

Tree data structures have various applications in computer science, including:

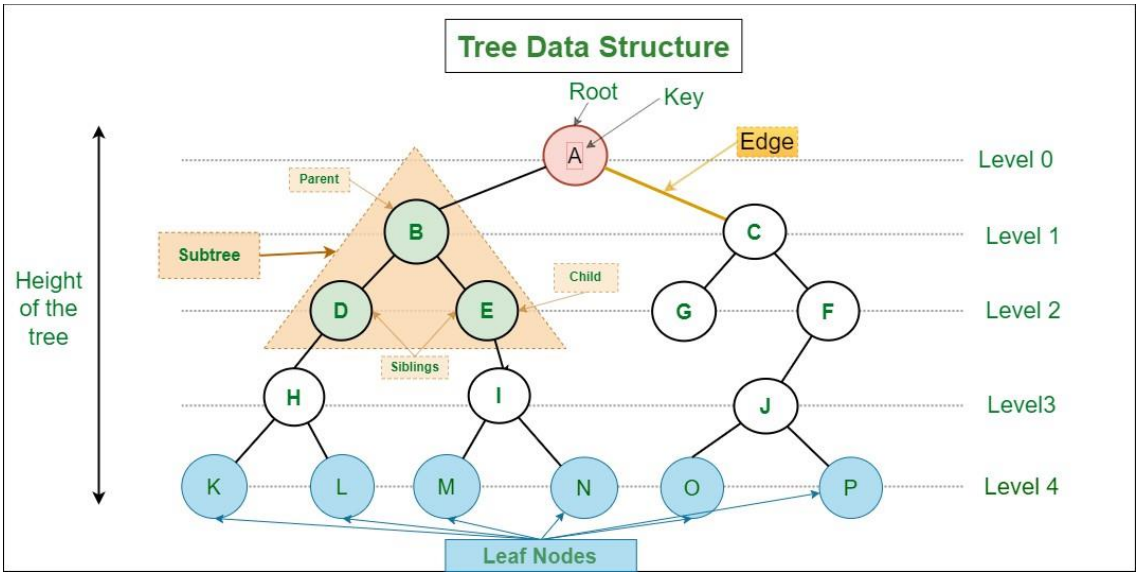
- File systems
- Network routing algorithms

- Data compression algorithms
- Decision trees and game trees
- Databases and indexing structures

Trees can be traversed in different ways, including:

- Depth-first search: A method to traverse a tree by exploring as far as possible along each branch before backtracking.
- Breadth-first search: A method to traverse a tree by exploring all the neighbor nodes at the current depth before moving on to the nodes at the next depth.

In summary, a tree is a collection of nodes that are arranged in a hierarchical fashion. It is a widely used data structure in computer science and has many applications, including file systems, network routing algorithms, and data compression algorithms. There are several types of trees, including binary trees, binary search trees, AVL trees, red-black trees, B-trees, and tries. Trees can be traversed in different ways, including depth-first search and breadth-first search.



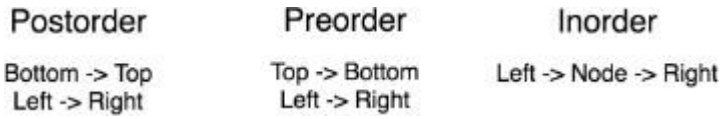
Tree Traversals

Create a complete binary tree with the digits of a 12 digit Number

1.inorder sequence

2.pre order sequence

3.post order sequence



1. In-order Tree traversal and Sequence

In-order traversal is a type of tree traversal that visits all the nodes in a binary search tree (BST) in a specific order. It involves visiting the left subtree, then the root node, and then the right subtree.

Here is the algorithm for in-order tree traversal:

1. Traverse the left subtree by calling in-order on the left child of the current node (if it exists).
2. Visit the current node.
3. Traverse the right subtree by calling in-order on the right child of the current node (if it exists).

In-order traversal has the following properties:

- It visits all nodes of a BST exactly once.
- It visits nodes in ascending order of their values if the BST is constructed such that smaller values are placed to the left of the root and larger values are placed to the right.

In-order traversal is useful in various applications such as searching for a value in a BST, printing out values in ascending order, and evaluating arithmetic expressions in postfix notation.

Here is an implementation of in-order tree traversal in Python:

```
class Node:
    def___init_(self, val):
        self.val = val
        self.left = None
        self.right = None

def inorder_traversal(node):
    if not node:
        return
    inorder_traversal(node.left)
```

```
        print(node.val, end=' ')
        inorder_traversal(node.right)
```

Here, the `inorder_traversal` function takes a node as an argument and performs in-order traversal on the subtree rooted at that node. If the node is None, the function returns immediately. Otherwise, it calls `inorder_traversal` on the left child of the node, prints the value of the node, and then calls `inorder_traversal` on the right child of the node. The `end=' '` argument in the `print` statement is used to print the values on a single line separated by a space.

In [15]:
```
class Node:
  def__init__(self, key):
    self.left = None
    self.right = None
    self.val = key

def printInorder(root):
  if root:
    printInorder(root.left)
    print(root.val, end = " ")
    printInorder(root.right)

if__name__== "__main__":
  root = Node(1)
  root.left = Node(2)
  root.right = Node(2)
  root.left.left = Node(0)
  root.left.right = Node(1)
  root.right.left = Node(0)
  root.right.right = Node(4)
  root.left.left.left = Node(0)
  root.left.left.right = Node(5)
  root.left.right.left = Node(0)
  root.left.right.right = Node(5)
  root.right.left.right = Node(6)
  print ("In-Order traversal of binary tree is : ")
  printInorder(root)
```

```
In-Order traversal of binary tree is :
0 0 5 2 0 1 5 1 0 6 2 4
```

This code defines a binary tree and prints its in-order traversal.

First, the `Node` class is defined with three attributes: `left` and `right` pointers, which point to the left and right subtrees of the node, respectively, and a `val` attribute which stores the value of the node.

Next, a function `printInorder` is defined that takes a binary tree root node as input and recursively prints its in-order traversal. In-order traversal visits the left subtree, then the root node, and then the right subtree. This is done by calling the `printInorder` function on the left subtree, printing the value of the root node, and then calling the `printInorder` function on the right subtree.

Finally, the `main` function creates a binary tree with some values and calls the `printInorder` function to print its in-order traversal.

2. Pre-order Tree traversal and Sequence

Pre-order traversal is a type of depth-first search (DFS) traversal algorithm that traverses the nodes of a tree or graph in a specific order. In the pre-order traversal algorithm, the root node is visited first, followed by the left subtree, and then the right subtree.

The pre-order traversal algorithm can be implemented using recursion or iteration. The recursive implementation is simpler and more intuitive. It follows the following steps:

1. Visit the root node.
2. Recursively traverse the left subtree.
3. Recursively traverse the right subtree.

The iterative implementation of the pre-order traversal algorithm uses a stack data structure. It follows the following steps:

1. Push the root node onto the stack.
2. While the stack is not empty, do the following: a. Pop the top node from the stack. b. Visit the popped node. c. Push the right child of the popped node onto the stack. d. Push the left child of the popped node onto the stack.

The time complexity of the pre-order traversal algorithm is O(n), where n is the number of nodes in the tree or graph. This is because each node is visited exactly once. The space complexity of the recursive implementation is O(h), where h is the height of the tree or graph, because the function call stack can grow as tall as the height of the tree or graph. The space complexity of the iterative implementation is O(n), because the stack can hold all the nodes of the tree or graph.

here's the code for pre-order traversal:

```
class Node:
    def___init_(self, val):
        self.val = val
        self.left = None
        self.right = None

def preorder_traversal(node):
    if not node:
        return
    print(node.val, end=' ')
    preorder_traversal(node.left)
    preorder_traversal(node.right)
```

In pre-order traversal, the current node is visited first, followed by its left subtree and then its right subtree.

In [16]:
```
class Node:
  def___init__(self, key):
    self.left = None
    self.right = None
    self.val = key

def printPreorder(root):
```

```
  if root:
    print(root.val,end = " ")
    printPreorder(root.left)
    printPreorder(root.right)

if__name__== "__main__":
  root = Node(1)
  root.left = Node(2)
  root.right = Node(2)
  root.left.left = Node(0)
  root.left.right = Node(1)
  root.right.left = Node(0)
  root.right.right = Node(4)
  root.left.left.left = Node(0)
  root.left.left.right = Node(5)
  root.left.right.left = Node(0)
  root.left.right.right = Node(5)
  root.right.left.right = Node(6)

print("Pre-Order traversal of binary tree is : ")
printPreorder(root)
```

```
Pre-Order traversal of binary tree is :
1 2 0 0 5 1 0 5 2 0 6 4
```

This is a Python code that defines a binary tree using a Node class and prints its pre-order traversal. Here is an explanation of the code:

- The Node class defines a node of the binary tree, which has three attributes: left, right, and val. The left and right attributes point to the left and right children of the node, respectively, and the val attribute stores the value of the node.
- The printPreorder function is a recursive function that takes a node as an argument and prints its value first, then recursively calls itself on the left child of the node, and finally recursively calls itself on the right child of the node. The function continues this process until it reaches a leaf node (i.e., a node with no children).
- In the if **name** == "**main**" block, the code creates a binary tree with the root node having a value of 1.
- The code then calls the printPreorder function on the root node, which prints the pre-order traversal of the binary tree: 1 2 0 0 5 1 0 5 2 0 4 6. The pre-order traversal visits the root node first, then recursively visits the left subtree in pre-order, and finally recursively visits the right subtree in pre-order.

3. Post-order Tree traversal and Sequence

Post-order tree traversal is a depth-first search algorithm used to traverse a binary tree. In this algorithm, the traversal starts from the root node and visits the left subtree followed by the right subtree and finally the root node. The traversal is performed in such a way that the left child node is processed before the right child node, and the parent node is processed after the left and right child nodes.

The post-order traversal algorithm can be implemented using recursion or a stack data structure. Here is the recursive algorithm for post-order traversal:

1. Traverse the left subtree using post-order traversal

2. Traverse the right subtree using post-order traversal
3. Visit the root node

In other words, the algorithm first recursively traverses the left subtree, then the right subtree, and finally, it visits the root node. The post-order traversal is used to print the nodes in the order they would be deleted in a binary search tree.

The post-order traversal has a time complexity of O(n), where n is the number of nodes in the binary tree, since it visits each node exactly once. It is also worth noting that post-order traversal is used in many applications, including evaluating expressions in a binary expression tree and deleting nodes from a binary search tree.

Sure! Here's the code for a post-order traversal:

```
class Node:
    def__init_(self, val):
        self.val = val
        self.left = None
        self.right = None

def postorder_traversal(node):
    if not node:
        return
    postorder_traversal(node.left)
    postorder_traversal(node.right)
    print(node.val, end=' ')
```

In a post-order traversal, we first visit the left subtree, then the right subtree, and finally the root node. So we recursively traverse the left subtree first using `postorder_traversal(node.left)`, then traverse the right subtree using `postorder_traversal(node.right)`, and finally print the root node's value using `print(node.val, end=' ')`.

In [17]:
```
class Node:
  def__init__(self, key):
    self.left = None
    self.right = None
    self.val = key

def printPostorder(root):
  if root:
    printPostorder(root.left)
    printPostorder(root.right)
    print(root.val,end = " ")

if__name__== "__main__":
  root = Node(1)
  root.left = Node(2)
  root.right = Node(2)
  root.left.left = Node(0)
  root.left.right = Node(1)
  root.right.left = Node(0)
  root.right.right = Node(4)
  root.left.left.left = Node(0)
  root.left.left.right = Node(5)
  root.left.right.left = Node(0)
  root.left.right.right = Node(5)
```

```
  root.right.left.right = Node(6)

print("Post-Order traversal of binary tree is")
printPostorder(root)
```

```
Post-Order traversal of binary tree is
0 5 0 0 5 1 2 6 0 4 2 1
```

This is a Python code that defines a Node class and a function called "printPostorder" that performs a post-order traversal of a binary tree.

The Node class has three instance variables: left, right, and val. The left and right variables are used to point to the left and right subtrees of the node, and the val variable stores the value of the node.

The printPostorder function takes a root node as input and recursively traverses the binary tree in a post-order fashion. In a post-order traversal, we first visit the left subtree, then the right subtree, and finally the root node. The function uses recursion to traverse the left and right subtrees first, and then prints the value of the root node.

In the driver code, a binary tree is created with the root node having a value of 1, and the left and right subtrees having values of 2. The left subtree of the root node has two children with values 0 and 1 respectively, and the right subtree also has two children with values 0 and 4 respectively. The left subtree of the node with value 2 also has two children with values 0 and 5, and the right subtree has a single child with value 6.

Finally, the printPostorder function is called with the root node as input, and it prints the values of the nodes in the binary tree in post-order traversal.

Binary Search Tree BST

A Binary Search Tree (BST) is a data structure used for storing a collection of elements such as integers or strings. It is a tree-based data structure in which each node has at most two children, referred to as the left child and the right child.

The binary search tree has the property that for every node, the value of all nodes in its left subtree is less than or equal to the node's value, and the value of all nodes in its right subtree is greater than the node's value. This property allows for efficient search, insertion, and deletion operations.

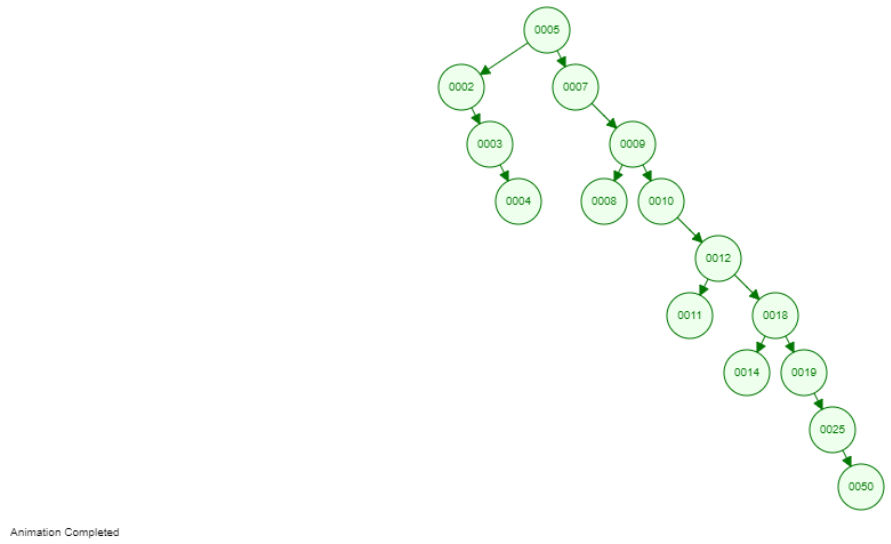
The search operation works by comparing the value of the element to be searched with the value of the current node. If the value matches, the search is successful. If the value is less than the current node, the search continues in the left subtree. If the value is greater than the current node, the search continues in the right subtree. This process is repeated until either the value is found or the subtree becomes empty.

The insertion operation works by traversing the tree to find the appropriate location to insert the new element. The element is inserted as a leaf node in the appropriate position based on its value.

The deletion operation is a bit more complicated. There are three cases to consider: (1) the node to be deleted has no children, (2) the node to be deleted has one child, and (3) the node to be deleted has two children. In the first case, the node is simply removed from the tree. In the second case, the child node replaces the deleted node. In the third case, the leftmost node in the right subtree is found and replaces the deleted node.

The BST has several advantages over other data structures such as arrays or linked lists. It allows for efficient search, insertion, and deletion operations in O(log n) time complexity, where n is the number of nodes in the tree. In addition, it can be used for various applications such as sorting, searching, and ranking of data.

However, the performance of the BST is highly dependent on the shape of the tree. If the tree is unbalanced, i.e., it is skewed to the left or right, the search operation may take O(n) time complexity, where n is the number of nodes in the tree. To prevent this, balanced tree structures such as AVL trees and Red-Black trees can be used.



Implementation of BST using Python:

In [1]:

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

class BST:
    def __init_(self):
        self.root = None

    def insert(self, val):
        if self.root is None:
            self.root = Node(val)
        else:
            self._insert(val,  self.root)

    def _insert(self, val, node):
        if val < node.val:
            if node.left is None:
                node.left = Node(val)
            else:
                self._insert(val,  node.left)
        else:
            if node.right is None:
                node.right = Node(val)
            else:
                self._insert(val,  node.right)

    def delete(self, val):
        self.root  =  self._delete(val,  self.root)

    def _delete(self, val, node):
        if not node:
            return None
        elif val < node.val:
            node.left  =  self._delete(val,  node.left)
        elif val > node.val:
            node.right  =  self._delete(val,  node.right)
        else:
            if not node.left:
                return node.right
            elif not node.right:
```

```
                return node.left
            else:
                temp = self._find_min(node.right)
                node.val = temp.val
                node.right = self._delete(temp.val, node.right)
        return node

    def _find_min(self, node):
        while node.left:
            node = node.left
        return node

    def inorder_traversal(self, node):
        if not node:
            return
        self.inorder_traversal(node.left)
        print(node.val, end=' ')
        self.inorder_traversal(node.right)
```

This is the implementation of a binary search tree (BST) in Python.

A BST is a binary tree where each node has a key (value) and the keys in the left subtree are smaller than the key in the root node, while the keys in the right subtree are greater than the key in the root node. This makes searching, insertion, and deletion of keys in the tree efficient.

The class `Node` represents a node in the BST, with a value (`val`) and two child nodes (`left` and `right`).

The class `BST` represents the BST data structure, with a root node (`root`) and the following methods:

- `insert(val)` : inserts a new node with the given value into the BST.
- `_insert(val, node)` : a helper method for `insert()` , which recursively finds the correct place to insert the new node.
- `delete(val)` : removes the node with the given value from the BST.
- `_delete(val, node)` : a helper method for `delete()` , which recursively finds the node to be deleted and rearranges the tree accordingly.
- `_find_min(node)` : a helper method for `_delete()` , which finds the minimum value node in the right subtree of the node to be deleted.
- `inorder_traversal(node)` : performs an inorder traversal of the BST, printing the values of the nodes in sorted order.

Overall, this implementation allows for efficient searching, insertion, and deletion of keys in a BST, making it a useful data structure for many applications.

In [2]:
```
bst = BST()
data = [5, 7, 2, 9, 3, 4, 8, 10, 12, 18, 11, 14, 19, 25, 50]
for val in data:
    bst.insert(val)

print('Before deletion:', end=' ')
bst.inorder_traversal(bst.root)

bst.delete(12)
```

```
print('\nAfter deletion:', end=' ')
bst.inorder_traversal(bst.root)

bst.delete(4)
print('\nAfter deletion:', end=' ')
bst.inorder_traversal(bst.root)
```

```
Before deletion: 2 3 4 5 7 8 9 10 11 12 14 18 19 25 50
After deletion: 2 3 4 5 7 8 9 10 11 14 18 19 25 50
After deletion: 2 3 5 7 8 9 10 11 14 18 19 25 50
```

This code creates a binary search tree (BST) and inserts a list of values into it. It then calls the `inorder_traversal` method to print out the values of the nodes in the tree in ascendingorder.

Afterwards, it deletes the node with value 12 from the tree and prints out the updated values using `inorder_traversal` again.

Finally, it deletes the node with value 4 and prints out the updated values again.

Introduction to Numpy

NumPy (Numerical Python) is a powerful library in Python for numerical computing. It provides efficient data structures and functions for working with large arrays and matrices.

NumPy is a fundamental package for scientific computing in Python. It provides a multidimensional array object, various derived objects, and a collection of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation, and more.

Installing NumPy

NumPy is typically installed alongside Python, but you can install it separately using `pip`, the Python package installer. Open your command prompt or terminal and run the following command:

```
pip install numpy
```

NumPy Arrays

At the core of NumPy is the `ndarray` object, which stands for n-dimensional array. It is a table of elements (usually numbers) of the same type, indexed by a tuple of positive integers.

In [1]:
```
import numpy as np

# Create a 1D array
arr_1d = np.array([1, 2, 3, 4, 5])
print(arr_1d)

# Create a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr_2d)
```

```
[1 2 3 4 5]
[[1 2 3]
 [4 5 6]]
```

Array Creation

You can create NumPy arrays in various ways:

- Using the `array` function: `np.array([1, 2, 3])`
- Using functions like `zeros`, `ones`, `empty`: `np.zeros((2, 3))`
- Using `arange` and `linspace` functions: `np.arange(0, 10, 2)` or `np.linspace(0, 1, 5)`

- Reading arrays from files: `np.loadtxt('data.txt')`

In [2]:
```
import numpy as np

# Create an array of zeros
zeros_arr = np.zeros((3, 4))
print(zeros_arr)

# Create an array of ones
ones_arr = np.ones((2, 3))
print(ones_arr)

# Create an empty array
empty_arr = np.empty((2, 2))
print(empty_arr)

# Create an array with a range of values
range_arr = np.arange(0, 10, 2)
print(range_arr)

# Create an array with evenly spaced values
linspace_arr = np.linspace(0, 1, 5)
print(linspace_arr)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]]
[[2.89944835e-316 0.00000000e+000]
[6.90404044e-310 6.90401671e-310]]
[0 2 4 6 8]
[0.   0.25 0.5  0.75 1.  ]
```

Array Attributes

NumPy arrays have several attributes that provide information about the array, such as its shape, size, and data type. Some commonly used attributes include `ndim`, `shape`, `size`, and `dtype`.

In [3]:
```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr.ndim)  # Number of dimensions
print(arr.shape)  # Shape of the array
print(arr.size)  # Number of elements
print(arr.dtype)  # Data type of the array
```

```
2
(2, 3)
6
int64
```

Array Indexing

You can access elements of a NumPy array using indexing. Indexing in NumPy arrays starts from 0, and you can use positive and negative indices. For example, `arr[0]` accesses the

first element, and `arr[-1]` accesses the last element.

In [4]:
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr[0])  # Access the first element
print(arr[-1])  # Access the last element
print(arr[1:4])  # Access a range of elements
```

```
1
5
[2 3 4]
```

Array Slicing

NumPy arrays also support slicing, which allows you to extract specific portions of an array. Slicing is done using the colon operator (`:`). For example, `arr[1:5]` retrieves elements from index 1 to 4.

In [5]:
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr[1:4])  # Slice a range of elements
print(arr[:3])  # Slice from the beginning to index 2
print(arr[3:])  # Slice from index 3 to the end
```

```
[2 3 4]
[1 2 3]
[4 5]
```

Array Reshaping

You can reshape an array using the `reshape` method or the `reshape` function. Reshaping changes the shape of the array without modifying its data. For example, `arr.reshape((2, 3))` reshapes a 1D array into a 2D array with 2 rows and 3 columns.

In [6]:
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

reshaped_arr = arr.reshape((2, 3))
print(reshaped_arr)

# Note: The reshaping should be compatible with the number of elements in the arra
```

```
[[1 2 3]
 [4 5 6]]
```

Array Operations

NumPy provides various mathematical operations that can be applied to arrays. These include basic arithmetic operations like addition, subtraction, multiplication, division, and

more. For example, `arr1 + arr2` performs element-wise addition of two arrays.

In [7]:
```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition
result = arr1 + arr2
print(result)

# Element-wise multiplication
result = arr1 * arr2
print(result)

# Matrix multiplication
result = np.dot(arr1, arr2)
print(result)
```

```
[5 7 9]
[ 4 10 18]
32
```

Broadcasting

Broadcasting is a powerful mechanism in NumPy that allows arrays with different shapes to be used in arithmetic operations. In broadcasting, the smaller array is broadcast across the larger array to make them compatible for the operation. For example, you can add a scalar value to an array, and NumPy will automatically broadcast the scalar to all elements in the array.

In [8]:
```
import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])
scalar = 2

# Add a scalar to an array
result = arr1 + scalar
print(result)

# Multiply an array by a scalar
result = arr1 * scalar
print(result)
```

```
[[3 4 5]
 [6 7 8]]
[[ 2  4  6]
 [ 8 10 12]]
```

Aggregation Functions

NumPy provides several aggregation functions that operate on arrays and return a single value. These include functions like `sum`, `mean`, `std`, `var`, `min`, `max`, and more. For example, `np.sum(arr)` computes the sum of all elements in the array.

In [9]:
```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Sum of all elements in the array
total_sum = np.sum(arr)
print("Sum:", total_sum)

# Minimum value in the array
min_value = np.min(arr)
print("Minimum:", min_value)

# Maximum value in the array
max_value = np.max(arr)
print("Maximum:", max_value)

# Mean of the array
mean_value = np.mean(arr)
print("Mean:", mean_value)

# Standard deviation of the array
std_value = np.std(arr)
print("Standard Deviation:", std_value)

# Variance of the array
var_value = np.var(arr)
print("Variance:", var_value)
```

```
Sum: 15
Minimum: 1
Maximum: 5
Mean: 3.0
Standard Deviation: 1.4142135623730951
Variance: 2.0
```

Array Manipulation

NumPy provides several functions for manipulating arrays, such as `transpose`, `concatenate`, `stack`, `split`, and more. These functions allow you to modify the shape and structure of arrays. For example, `np.transpose(arr)` returns a new array with the axes transposed.

In [11]:
```python
import numpy as np

arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([[7, 8, 9], [10, 11, 12]])

# Transpose of the array
arr_transposed = np.transpose(arr1)
print("Transposed Array:")
print(arr_transposed)

# Concatenate arrays along a specified axis
arr_concatenated = np.concatenate((arr1, arr2), axis=0)
print("Concatenated Array:")
print(arr_concatenated)

# Split the array into multiple sub-arrays
arr_split = np.array_split(arr_concatenated, 2, axis=1)
print("Split Array:")
```

```
print(arr_split)

# Flatten the array into a 1D array
arr_flattened = arr1.flatten()
print("Flattened Array:")
print(arr_flattened)
```

```
Transposed Array:
[[1 4]
 [2 5]
 [3 6]]
Concatenated Array:
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
Split Array:
[array([[ 1,  2],
       [ 4,  5],
       [ 7,  8],
       [10, 11]]), array([[ 3],
       [ 6],
       [ 9],
       [12]])]
Flattened Array:
[1 2 3 4 5 6]
```

Universal Functions (ufunc)

NumPy provides universal functions (ufunc) that operate on arrays element-wise. These functions include mathematical functions like `sin`, `cos`, `exp`, `log`, and more. For example, `np.sin(arr)` computes the sine of each element in the array.

In [12]:
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Square root of each element
sqrt_arr = np.sqrt(arr)
print("Square Root:")
print(sqrt_arr)

# Exponential of each element
exp_arr = np.exp(arr)
print("Exponential:")
print(exp_arr)

# Sine of each element
sin_arr = np.sin(arr)
print("Sine:")
print(sin_arr)

# Logarithm of each element
log_arr = np.log(arr)
print("Logarithm:")
print(log_arr)
```

```
Square Root:
[1.         1.41421356 1.73205081 2.         2.23606798]
Exponential:
[  2.71828183   7.3890561   20.08553692  54.59815003 148.4131591 ]
Sine:
[ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427]
Logarithm:
[0.         0.69314718  1.09861229  1.38629436  1.60943791]
```

File Input/Output

NumPy provides functions for reading and writing arrays to files. These include functionslike `loadtxt` , `savetxt` , `load` , `save` , and more. For example, `np.savetxt('data.txt', arr)` saves the array to a text file.

Introduction to Pandas

Pandas is a fast, powerful, and flexible open-source data manipulation and analysis library for Python. It provides data structures like Series and DataFrame, which allow you to work with labeled and tabular data efficiently.

Pandas is a versatile library that provides powerful tools for data manipulation, cleaning, analysis, and visualization. This comprehensive guide covered the basics to advanced topics in Pandas, equipping you with the knowledge and skills to effectively work with data in Python. By mastering Pandas, you'll have a valuable tool for data exploration, transformation, and analysis, enabling you to gain insights and make data-driven decisions in your projects and analyses.

Installing Pandas

Pandas can be installed using `pip`, the Python package installer. Open your command prompt or terminal and run the following command:

```
pip install pandas
```

Pandas Series

Pandas Series is a one-dimensional labeled array that can hold any data type. It is similar to a column in a spreadsheet or a dictionary. Learn how to create and manipulate Series objects, perform basic operations, and access elements.

In [ ]:
```python
import pandas as pd

# Create a Series
s = pd.Series([10, 20, 30, 40, 50])

print(s)
```

In this example, we created a Series s with values [10, 20, 30, 40, 50]. The default index is created automatically, starting from 0.

Pandas DataFrames

Pandas DataFrame is a two-dimensional labeled data structure with columns of potentially different types. It is similar to a table or a spreadsheet. Explore how to create DataFrames, load data from various sources, and perform operations on them.

In [ ]:
```python
import pandas as pd

# Create a DataFrame
```

```
data = {'Name': ['John', 'Alice', 'Bob'],
        'Age': [25, 30, 35],
        'City': ['New York', 'Paris', 'London']}

df = pd.DataFrame(data)

print(df)
```

In this example, we created a DataFrame df from a dictionary data. Each key-value pair in the dictionary represents a column in the DataFrame.

Data Input and Output

Learn how to read and write data in different formats using Pandas. This includes CSV files, Excel spreadsheets, SQL databases, and more. Discover different options and parameters for efficient data input/output operations.

Pandas provides various functions to read and write data in different formats. Here's an example of reading data from a CSV file and writing data to a CSV file:

```
import pandas as pd

# Read data from CSV file
df = pd.read_csv('data.csv')

# Write data to CSV file
df.to_csv('output.csv', index=False)
```

In this example, we used `pd.read_csv` to read data from a CSV file into a DataFrame `df`. We then used `df.to_csv` to write the DataFrame `df` to a CSV file named 'output.csv'. The `index=False` argument is used to exclude the index column in the output.

Data Selection and Indexing

Understand various techniques to select, slice, and index data in Pandas. Learn how to use labels, indices, conditions, and logical operators to filter and retrieve specific subsets of data from Series and DataFrames.

In [ ]:
```
import pandas as pd

# Select a single column
col = df['Name']

# Select multiple columns
cols = df[['Name', 'Age']]

# Select rows using boolean indexing
selected_rows = df[df['Age'] > 30]

# Select rows and columns using loc
selected_data = df.loc[2, 'Name']

print(col)
print(cols)
```

```
print(selected_rows)
print(selected_data)
```

In this example, we demonstrated different ways to select data. We selected a single column using df['Name'], multiple columns using df[['Name', 'Age']], and rows using boolean indexing df[df['Age'] > 30]. We also used df.loc to select specific rows and columns based on labels.

Data Manipulation and Cleaning

Discover techniques to clean and manipulate data in Pandas. Learn how to handle missing data, remove duplicates, rename columns, apply functions, sort and rank data, and perform string operations.

In [ ]:
```
import pandas as pd

# Rename columns
df.rename(columns={'Name': 'Full Name'}, inplace=True)

# Drop rows with missing values
df.dropna(inplace=True)

# Replace values
df['City'].replace('New York', 'NYC', inplace=True)

# Apply a function to a column
df['Age'] = df['Age'].apply(lambda x: x + 1)

# Sorting the DataFrame
df.sort_values(by='Age', ascending=False, inplace=True)

# Removing duplicates
df.drop_duplicates(subset='Full Name', inplace=True)

# Resetting the index
df.reset_index(drop=True, inplace=True)

# Checking for null values
null_values = df.isnull().sum()

print(df)
print(null_values)
```

In this example, we demonstrated various data manipulation and cleaning techniques. We renamed the 'Name' column to 'Full Name' using rename(). We dropped rows with missing values using dropna(). We replaced a specific value in the 'City' column using replace(). We applied a function to the 'Age' column using apply(). We sorted the DataFrame based on the 'Age' column using sort_values(). We removed duplicates based on the 'Full Name' column using drop_duplicates(). We reset the index using reset_index(). Finally, we checked for null values in the DataFrame using isnull().sum().

Data Aggregation and Grouping

Explore how to aggregate and summarize data using Pandas. Learn about grouping data based on criteria, performing calculations, applying functions, and generating descriptive statistics using groupby operations.

In [ ]:
```
import pandas as pd

# Grouping and calculating mean
grouped_data = df.groupby('City')['Age'].mean()

# Aggregating with multiple functions
aggregated_data = df.groupby('City')['Age'].agg(['mean', 'min', 'max'])

print(grouped_data)
print(aggregated_data)
```

In this example, we grouped the data by the 'City' column and calculated the mean age for each city using groupby() and mean(). We also aggregated the data by 'City' column, calculating the mean, minimum, and maximum age using groupby() and agg().

Handling Missing Data

Learn techniques to handle missing data in Pandas. Understand how to identify missing values, handle them by dropping or filling with appropriate values, and perform imputation using various methods.

In [ ]:
```
import pandas as pd

# Checking for missing values
missing_values = df.isnull().sum()

# Dropping rows with missing values
df.dropna(inplace=True)

# Filling missing values with a specific value
df['Age'].fillna(0, inplace=True)

# Filling missing values with the mean
mean_age = df['Age'].mean()
df['Age'].fillna(mean_age, inplace=True)

print(missing_values)
print(df)
```

In this example, we first checked for missing values in the DataFrame using isnull().sum(). We then dropped rows with missing values using dropna(). We filled missing values in the 'Age' column with a specific value (0) and with the mean age using fillna().

Combining DataFrames

Discover different methods to combine multiple DataFrames in Pandas. Learn about concatenation, merging, and joining operations to combine data horizontally and vertically based on common keys or indices.

In [ ]:
```python
import pandas as pd

# Concatenating DataFrames vertically
df_concat = pd.concat([df1, df2])

# Concatenating DataFrames horizontally
df_concat = pd.concat([df1, df2], axis=1)

# Merging DataFrames based on a common column
df_merged = pd.merge(df1, df2, on='ID')

print(df_concat)
print(df_merged)
```

First, we used pd.concat() to concatenate DataFrames vertically by passing a list of DataFrames as the argument. This creates a new DataFrame df_concat with the rows from df1 followed by the rows from df2. We can also concatenate DataFrames horizontally by specifying axis=1, which results in the columns of df2 being added next to the columns of df1.

Second, we used pd.merge() to merge DataFrames based on a common column, 'ID', using the on parameter. This creates a new DataFrame df_merged with rows that have matching 'ID' values from both df1 and df2.

Reshaping and Pivoting Data

Understand how to reshape and pivot data in Pandas. Learn about techniques to transform data from wide to long format and vice versa, perform stacking and unstacking, and pivot tables to summarize and reorganize data.

In [ ]:
```python
import pandas as pd

# Reshaping data from wide to long format
df_long = pd.melt(df, id_vars=['ID'], value_vars=['var1', 'var2'], var_name='Varia

# Pivoting data from long to wide format
df_wide = df_long.pivot(index='ID', columns='Variable', values='Value')

print(df_long)
print(df_wide)
```

In this example, we used pd.melt() to reshape the data from wide to long format. We specified the 'ID' column as the identifier variable and selected 'var1' and 'var2' columns as value variables. The resulting DataFrame df_long has 'Variable' as the column containing the variable names and 'Value' as the column containing the corresponding values.

We then used pivot() to pivot the data from long to wide format. We specified the 'ID' column as the index, 'Variable' column as the column names, and 'Value' column as the values. The resulting DataFrame df_wide has 'ID' as the index and 'var1' and 'var2' as the columns.

Merging and Joining Data

Learn advanced techniques for merging and joining DataFrames in Pandas. Explore different types of joins, handling overlapping column names, merging on multiple keys, and dealing with data mismatches.

In [ ]:
```python
import pandas as pd

# Inner join
df_inner = pd.merge(df1, df2, on='ID', how='inner')

# Left join
df_left = pd.merge(df1, df2, on='ID', how='left')

# Right join
df_right = pd.merge(df1, df2, on='ID', how='right')

# Outer join
df_outer = pd.merge(df1, df2, on='ID', how='outer')

print(df_inner)
print(df_left)
print(df_right)
print(df_outer)
```

In this example, we used pd.merge() to perform different types of joins between df1 and df2 based on the common column 'ID'. We specified the join type using the how parameter.

Inner join (how='inner') returns only the rows with matching 'ID' values in both DataFrames. Left join (how='left') returns all the rows from the left DataFrame (df1) and the matching rows from the right DataFrame (df2). Right join (how='right') returns all the rows from the right DataFrame (df2) and the matching rows from the left DataFrame (df1). Outer join (how='outer') returns all the rows from both DataFrames, filling missing values with NaN when there is no match.

Time Series Analysis

Discover how to work with time series data in Pandas. Learn how to handle dates and time, resample and interpolate time series, perform advanced operations like shifting, lagging, and rolling calculations, and analyze time-based patterns and trends.

In [ ]:
```python
import pandas as pd

# Convert a column to datetime
df['Date'] = pd.to_datetime(df['Date'])

# Set the column as the DataFrame index
df.set_index('Date', inplace=True)

# Resample data to a different frequency
df_resampled = df.resample('W').sum()

# Rolling calculations
df['Rolling Mean'] = df['Value'].rolling(window=3).mean()

# Shifting data
```

```
df['Previous Value'] = df['Value'].shift(1)

print(df)
print(df_resampled)
```

In this example, we performed time series analysis using Pandas. First, we converted a column named 'Date' to a datetime data type using pd.to_datetime(). Then, we set the 'Date' column as the index of the DataFrame using set_index(). This allows us to work with time-based operations.

We resampled the data to a different frequency using resample(). In this example, we resampled the data to a weekly frequency ('W') and calculated the sum of values for each week. The resulting DataFrame df_resampled contains the resampled data.

We performed rolling calculations on the 'Value' column using the rolling() function. In this case, we calculated the rolling mean over a window of size 3.

We also demonstrated shifting the data in the 'Value' column using shift(). We shifted the values by 1 position, resulting in a new column named 'Previous Value' that contains the previous value for each row.

Handling Categorical Data

Explore techniques to handle categorical data in Pandas. Learn how to encode categorical variables, perform one-hot encoding, deal with ordinal data, and apply categorical operations for analysis and modeling.

In [ ]:
```
import pandas as pd

# Encoding categorical variables
df['Gender'] = df['Gender'].astype('category')
df['Gender Codes'] = df['Gender'].cat.codes

# One-hot encoding
one_hot_encoded = pd.get_dummies(df['City'])

# Handling ordinal data
ordinal_mapping = {'Low': 1, 'Medium': 2, 'High': 3}
df['Priority'] = df['Priority'].map(ordinal_mapping)

print(df)
print(one_hot_encoded)
```

In this example, we demonstrated different techniques for handling categorical data. We encoded the 'Gender' column using the astype('category') and cat.codes functions. This converts the column to a categorical data type and assigns a numerical code to each category. We performed one-hot encoding on the 'City' column using pd.get_dummies(). This createsnew binary columns for each unique value in the original column.

We handled ordinal data in the 'Priority' column by creating a mapping dictionary and using the map() function to replace the categories with corresponding numerical values. These techniques enable efficient handling and manipulation of categorical data in Pandas.

Thank you…

About Author,

Sai Likhith Panuganti

sailikithpaanuganti@gmail.com

https://www.linkedin.com/in/sailikhith