Introduction to Python

Python is an easy-to-learn and a powerful Object-Oriented Programming language. It is a very high-level programming language.

Why Python?

1. **Easy to Use**: Python is comparatively an easier-to-use language as compared to other programming languages.
2. **Expressive Language**: The syntax of Python is closer to how you would write pseudocode. Which makes it capable of expressing the code's purpose better than many other languages.
3. **Interpreted Language**: Python is an interpreted language; this means that the Python installation interprets and executes the code a line-at-a-time.
4. Python is one of the most popular programming languages to be used in **Web Development** owing to the variety of Web Development platforms built over it like Django, Flask, etc.

Python Download

The very first step towards Python Programming would be to download the tools required to run the Python language. We will be using Python 3 for the course. You can download the latest version of Python 3 from **https://www.python.org/downloads/**

Note:- If you are using Windows OS, then while installing Python make sure that *"Add Python to PATH"* is checked.

Getting an IDE for writing programs:

You can use any IDE of your choice, however, you are recommended to use Jupyter Notebook. You can download it from **https://jupyter.org/install**

Working in Python

Once you have Python installed on your system, you are ready to work on it. You can work in Python in two different modes:-

a) **Interactive Mode:** In this mode, you type one command at a time and Python executes the same. Python's interactive interpreter is also called Python Shell.

b) **Script Mode**: In this mode, we save all our commands in the form of a program file and later run the entire script. After running the script, the whole program gets compiled and you'll see the overall output.

First Program in Python

As we are just getting started with Python, we will start with the most fundamental program which would involve printing a standard output to the console. The `print()` function is a way to print to the standard output. The syntax to use `print()` function is as follows:-

```
In[] : print(<Objects>)
```

- <Objects> means that it can be one or more comma-separated 'Objects' to be printed.
- <Objects> must be enclosed within parentheses.

Example: If we want to print "Hello, World!" in our code, we will write it in the following way:-

```
In[] : print("Hello, World!")
```

and, we get the output as:

```
Out[] : Hello, World!
```

Python executed the first line by *calling* the `print()` function. The string value of `Hello, World!` was *passed* to the function.

Note:- The quotes that are on either side of `Hello, World!` were not printed to the screen because they are used to tell Python that they contain a string. The quotation marks delineate where the string begins and ends.

Variables in Python

What are Variables?

A variable in Python represents a named location that refers to value and whose values can be used and processed during the program run. In other words, variables are labels/names to which we can assign value and use them as a reference to that value throughout the code.

Variables are fundamental to programming for two reasons:

- **Variables keep values accessible**: For example, The result of a time-consuming operation can be assigned to a variable, so that the operation need not be performed each time we need the result.
- **Variables give values context:** For example, The number 56 could mean lots of different things, such as the number of students in a class, or the average weight of all students in the class. Assigning the number 56 to a variable with a name like **num_students** would make more sense, to distinguish it from another variable **average_weight**, which would refer to the average weight of the students. This way we can have different variables pointing to different values.

How are Values Assigned to A Variable?

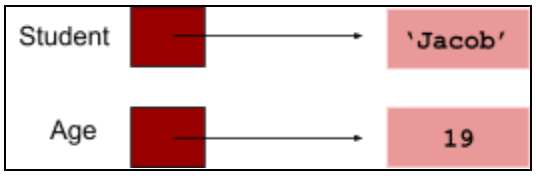
Values are assigned to a variable using a special symbol "=", called the **assignment operator**. An operator is a symbol, like = or +, that performs some operation on one or more values. For example, the + operator takes two numbers, one to the left of the

operator and one to the right, and adds them together. Likewise, the "=" operator takes a value to the right of the operator and assigns it to the name/label/variable on the left of the operator.

For Example: Now let us create a variable namely **Student** to hold a student's name and a variable **Age** to hold a student's age.

```
>>> Student = "Jacob"
>>> Age = 19
```

Python will internally create labels referring to these values as shown below:



Now, let us modify the first program we wrote.

```
greeting = "Hello, World!"
print(greeting)
```

Here, the Python program assigned the value of the string to a variable `greeting`, and then when we call `print(greeting)`, it prints the value that the variable, `greeting`, points to i.e. `"Hello, World!"`
We get the output as:-

```
Hello, World!
```

Naming a Variable

You must keep the following points in your mind while naming a variable:-

- Variable names can contain letters, numbers, and underscores.
- They cannot contain spaces.
- Variable names cannot start with a number.

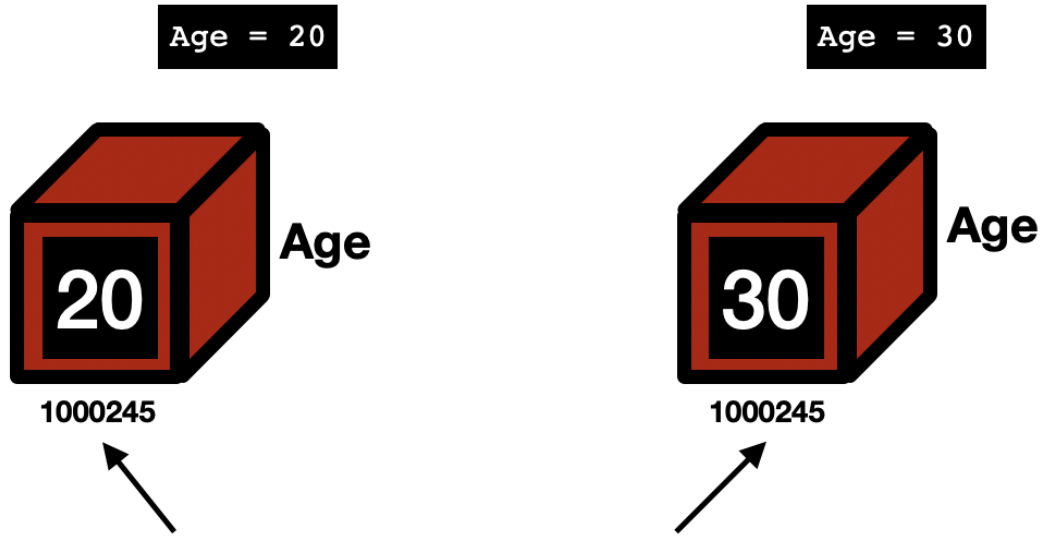
- Variable names are case sensitive. For example:- The variable names **Temp** and **temp** are different.
- While writing a program, creating self-explanatory variable names help a lot in increasing the readability of the code. However, too long names can clutter up the program and make it difficult to read.

Traditional Programming Languages' Variables in Memory

Let us study how variables and the values they are assigned, are represented in memory, in traditional programming languages like C, C++, Java, etc.
In these languages, variables are like **storage containers**. They are like named storage locations that store some value. In such cases, whenever we declare a new variable, a new storage location is given to that name/label and the value is stored at that named location. Now, whenever a new value is reassigned to that variable, the storage location remains the same. However, the value stored in the storage location is updated. This can be shown from the following illustration.

Consider the following script:

```
Age = 20
Age = 30 # Re-assigning a different value to the same variable
```



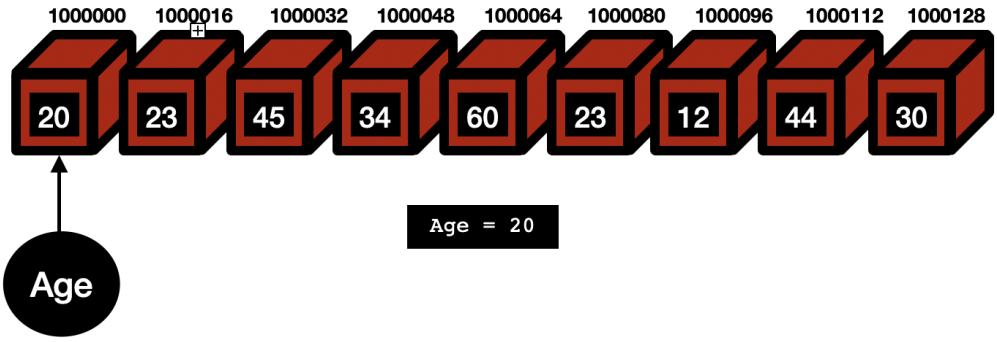
In the above script, when we declare a new variable `Age`, a container box/ Memory Location is named **Age** and the value 20 is stored in the memory address `1000245` with name/label, **Age**. Now, on reassigning the value 30 to `Age`, the value 30 is stored in the same memory location. This is how the variables behave in Traditional programming languages.

Python Variables in Memory

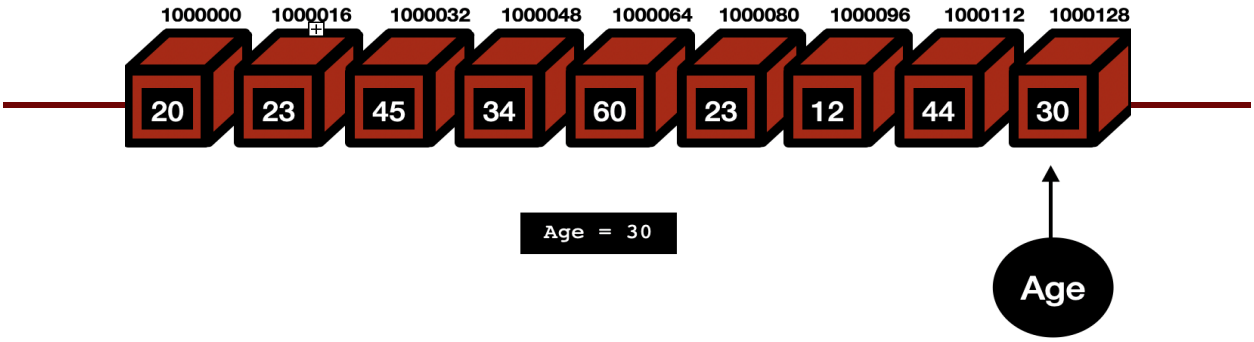
Python variables are not created in the form most other programming languages do. These variables do not have fixed locations, unlike other languages. The locations they refer/point to changes every time their value changes.

Python preloads some commonly used values in an area of memory. This memory space has values/literals at defined memory locations and all these locations have different addresses.

When we give the command, `Age = 20`, the variable `Age` is created as a label pointing to a memory location where 20 is already stored. If 20 is not present in any of the memory locations, then 20 is stored in an empty memory location with a unique address and then the `Age` is made to point to that memory location.



Now, when we give the second command, `Age = 30`, the label `Age` will not have the same location as earlier. Now it will point to a memory location where 30 is stored. So this time the memory location for the label `Age` is changed.



Data Types

Introduction

Data types are the classification or categorization of data items. Data types represent a kind of value that determines what operations can be performed on that data. Numeric, non-numeric, and Boolean (true/false) data are the most used data types. However, each programming language has its classification largely reflecting its programming philosophy. Python offers the following built-in data types:

- Numbers
    - Integers
    - Floating Point Numbers
    - Complex Numbers
- Strings
- Boolean Values
- List, Tuple, and Dictionary *(To be covered later in the course)*

Type Code	Description	Default Size (In Bytes)
`int`	Integers	4
`float`	Floating Point Numbers	4
`bool`	Boolean Values	1

Note:- If a variable has been assigned a value of some data type. It can be reassigned a value belonging to some other Data Type in the future.

```
a= "Raw" # String Data Type
a= 10 # Integer Data Type
a= 5.6 # Floating Point Number Data Type
a= 1+8j # Complex Number
a= True # Boolean Value
```

Python Numbers

Introduction

Number data types store numerical values. Python supports Integers, floating-point numbers, and complex numbers. They are defined as `int`, `float`, and `complex` classes.

- Integers can be of any length (Only limited by the memory available). They do not have a decimal point and can be positive or negative.

- A floating-point number is a number having a fractional part. The presence of a decimal point indicates a floating-point number. They have a precision of up to 15 digits.
- 1 is an integer, 1.0 is a floating-point number.
- Complex numbers are of the form, x + yj, where x is the real part and y is the imaginary part.

We can use the `type()` function to know which class a variable or a value belongs to. Similarly, the `isinstance()` function is used to check if an object belongs to a particular class.

Here are a few examples:-

```
b = 5
print(b, "is of type", type(b))
b = 2.0
print(b, "is of type", type(b))
b = 1+2j
print(b, "is complex number?", isinstance(b,complex))
```

And we will get the output as:

```
5 is of type <class 'int'>
```

```
2.0 is of type <class 'float'>
```

```
1+2j is complex number? True
```

Arithmetic Operators in Python

The Arithmetic Operators are used in Python in the same way as they are used in Mathematics.

OPERATOR	DESCRIPTION
+	Add two operands
-	Subtracts second operand from the first

*	Multiplies two operands
/	Divides numerator by denominator (Floating Point Division)
//	Divides numerator by denominator (Floor Division) - Acts as a floor function
**	**Exponent Operator**- The first operand raised to the power of the second operand
%	**Modulo Operator-** Calculates remainder left after dividing first by second

Let us see how these operators work:-

```
In[] : print(5 + 2) # Addition
Out[] : 7
In[] : print(5 - 2) # Subtraction
Out[] : 3
In[] : print(5 * 2) # Multiplication
Out[] : 10
In[] : print(5 / 2) # Floating Point Division
Out[] : 2.5
In[] : print(5 // 2) # Floor Division
Out[] : 2 # 5 divided by 2 gives 2.5 and value of floor(2.5) is 2
In[] : print(5 ** 2) # Calculate Exponent
Out[] : 25 # 5 raised to the power of 2 is 25
In[] : print(5 % 2) # Modulus
Out[] : 1 # Remainder 1 is left after dividing 5 by 2
```

Taking User Input

Developers often need to interact with users, either to get data or to provide some sort of result.

How to take User Input?

To get the input from the user interactively, we can use the built-in function, `input()`. This function is used in the following manner:

```
variable_to_hold_the_input_value = input(<Prompt to be displayed>)
```

For example:

```
In[] : age = input("What is your age?")
```

The above statement will display the prompt as:-

```
What is your age?_________ ←{User input here}
```

We will get the following interactive output:

```
In[] : name = input("Enter your name: ")
Enter your name: Rishabh #User Input
In[] : age = input("Enter your age: ")
Enter your age: 20 #User Input
In[] : name
Out[] : 'Rishabh'
In[] : age
Out[] : '19'
```

Note:- `input()` function always returns a value of the **String** type. Notice that in the above script the output for both name and age, Python has enclosed the output in quotes, like `'Rishabh'` and `'19'`, which implies that it is of **String** type. This is just because, whatever the user inputs in the `input()` function, it is treated as a **String**. This would mean that even if we input an integer value like 20, it will be treated like a string `'19'` and not an integer. Now, we will see how to read Numbers in the next section.

Reading Numbers

Python offers two functions `int()` and `float()` to be used with the `input()` function to convert the values received through `input()` into the respective numeric types integer and floating-point numbers. The steps will be:-

1. Use the `input()` function to read the user input.

2. Use the `int()` and `float()` function to convert the value *read* into integers and floating-point numbers, respectively. This process is called **Type Casting**.

The general way of taking Input:

```
variableRead = input(<Prompt to be displayed>)
updatedVariable = int(variableRead)
```

Here, `variableRead` is a String type that was read from the user. This string value will then be converted to Integer using the `int()` function and assigned to `updatedVariable.`
This can even be shortened to a single line of code as shown below:-

```
updatedVariable = int(input(<Prompt to be displayed>))
```

Let us take an example:-

```
In[] : age= int(input("Enter Your Age: "))
Enter Your Age: 19
In[] : age
Out[] : 19
```

Here, the output will be `19` and not `'19'`, i.e. the output is an Integer and not a String.

Similarly, if we want to read a floating-point number, we can do the following:-

```
In[] : weight= float(input("Enter Your Age: "))
Enter Your Weight: 65.5
In[] : weight
Out[] : 65.5
```

Conditionals and Loops

Boolean Data Type

A boolean expression (or logical expression) evaluates to one of the two states - `True` or `False`. Several functions and operations in Python return boolean objects.

The assignment of boolean datatype to some variable can be done similar to other data types. This is shown as follows:-

```
>>> a = True
>>> type(a)
<class 'bool'>
```

```
>>> b = False
>>> type(b)
<class 'bool'>
```

Note:- Here `True` and `False` are Reserved keywords. ie They are not written as "True" or "False", as then they will be taken as strings and not boolean values.

```
C = "True"
D = "False"
```

Here, C and D are of type String.

Also, note the keywords `True` and `False` must have an Upper Case first letter. Using a lowercase `true` or `false` will return an error.

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

Relational Operators

The operators which compare the values of their operands are called comparison/ relational operators. Python has 6 most common relational operators. Let X and Y be the two operands and let X = 5 and Y = 10.

Operator	Description	Example
==	If the values of two operands are equal, then the condition is **true**, otherwise, it is **false**. ***Common Mistake:-*** *Do not confuse it with the Assignment Operator(=).*	(X == Y) is **false**
!=	If the values of the two operands are not equal, then the condition is **true.**	(X != Y) is **true.**
>	If the value of the left operand is greater than the value of the right operand, then the condition is **true.**	(X > Y) is **false**
<	If the value of the left operand is less than the value of the right operand, then the condition is **true.**	(X < Y) is **true.**
>=	If the value of the left operand is greater than or equal to the value of the right operand, then the condition is **true.**	(X >= Y) is **false.**
<=	If the value of the left operand is less than or equal to the value of the right operand, then the condition is **true.**	(X <= Y) is **true.**

Logical Operators

The operators which act on one or two boolean values and return another boolean value are called logical operators. There are 3 key logical operators. Let X and Y be the two operands and let **X = True** and **Y = False.**

Operator	Description	Example
and	Logical AND: If both the operands are true then the condition is true.	(X and Y) is **false**
or	Logical OR: If any of the two operands are then the condition is true.	(X or Y) is **true**
not	Logical NOT: Used to reverse the logical state of its operand.	Not(X) is **false**

The Truth table for all combination of values of X and Y

X	Y	X and Y	X or Y	not(X)	not(Y)
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Let us consider an example code to understand the relational operators in Python:

```
x = 9
y = 13

print('x > y is',x > y) # Here 9 is not greater than 13

print('x < y is',x < y) # Here 9 is less than 13

print('x == y is',x == y) # Here 9 is not equal to 13

print('x != y is',x != y) # Here 9 is not equal to 13

print('x >= y is',x >= y) # Here 9 is not greater than or equal to 13

print('x <= y is',x <= y) # Here 9 is less than 13
```

And we get the output as:

```
x > y is False
x < y is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

Let us consider another example code to understand the logical operators in Python:

```
x = True
y = False

print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

And we get the output as:

```
x and y is False
x or y is True
not x is False
```

Introduction to If-Else

There are certain points in our code when we need to make some decisions and then based on the outcome of those decisions we execute the next block of code. Such conditional statements in programming languages control the flow of program execution.

Most commonly used conditional statements in Python are:

- Simple If statements
- If-Else statements
- If-Elif statements
- Nested Conditionals

Simple If statements

These are the most simple decision-making/conditional statements. It is used to decide whether a certain statement or block of statements will be executed or not.

- The most important part of any conditional statement is a condition or a boolean.
- And the second important thing is the code block to be executed.

In the case of simple `If` statements, if the conditional/boolean is true then the given code block is executed, else the code block is simply skipped and the flow of operation comes out of this If condition.

The **general syntax** of such statements in Python is:

```
if <Boolean/Condition>:
      <Code Block to be executed in case the Boolean is True>
<Code Block to be executed in case the Boolean is False>
```

An **example** of a simple `if` statement can be as follows:

```
Val = False
if Val == True:
      print("Value is True") # Statement 1
print("Value is False") # Statement 2
```

In the above code, the variable Val has a boolean value **False**, and hence the condition is not satisfied. Since the condition is not satisfied, it skips the `If` statement, and instead, the next statement is executed. Thus the output of the above code is:

```
Value is False
```

Importance of Indentation in Python

To indicate a block of code and separate it from other blocks in Python, you must indent each line of the block by the same amount. The two statements in our example of Simple if-statements are both indented four spaces, which is a typical amount of indentation used in the case of Python.

In most other programming languages, indentation is used only to improve the readability of the code. But in Python, it is required for indicating what block of code, a statement belongs to. **For instance**, *Statement 1* which is indented by 4 spaces is a part of the `if` statement block. On the other hand, *Statement 2* is not indented, and hence it is not a part of the `if` block. This way, indentation indicates which statements from the code belong together.

Any deviation from the ideal level of indentation for any statement would produce an indentation error. **For example:** On running the given script:

```
Val = False
if Val == True:
        print("Value is True") # Statement 1
    print("Value is False") # Statement 2
```

We get the output as:

```
IndentationError: unindent does not match any outer indentation level
```

This error is because *statement 1* is not in the indentation line for the `if` statement.

Else-If statements

The simple `if` statement, tells us that if a condition is true it will execute a block of statements, and if the condition is false it won't. But what if we want some other block of code to be executed if the condition is false. Here comes the *else* statement. We can use the `else` statement with `if` statement to execute a block of code when the condition is false. The general Syntax for the If-Else statement is:

```
if (Condition/Boolean):
    <Code block to be executed in case the condition is True>
else:
    <Code block to be executed in case the condition is False>
```

*** Keep in mind the indentation levels for various code blocks. Let us now take an example to understand If-Else statements in depth.

Distinguishing between Odd and Even numbers:

Problem Statement: Given a number, print whether it is odd or even.

Approach: In order fora number to be even, it must be divisible by 2. Which means that the remainder upon dividing the number by 2 must be 0. Thus, in order to distinguish between odd and even numbers, we can use this condition. The numbers which leave a remainder 0 on division with 2 will be categorized as even, else the number is odd. This can be written in Python as follows:-

```
num = 23
If num%2 == 0:
    print("Even Number")
else:
    print("Odd Number")
```

The output of this code will be:-

```
Odd Number
```

Since 23 is an odd number, it doesn't satisfy the `if` condition, and hence it goes to `else` and executes the command.

If-Elif-Else statements

So far we have looked at Simple If and a single If-Else statement. However, imagine a situation in which if a condition is satisfied, we want a particular block of code to be executed, and if some other condition is fulfilled we want some other block of code to run. However, if none of the conditions is fulfilled, we want some third block of code to be executed. In this case, we use an **if-elif-else** ladder.

In this, the program decides among multiple conditionals. The `if` statements are executed from the top down. As soon as one of the conditions controlling the `if` is true, the statement associated with that `if` is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

The general syntax of an **if-elif-else** ladder will be:

```
if <Condition 1>:
      <Execute this code block>
elif <Condition 2>:
      <Execute this code block>
.
.
elif <Condition X>:
      <Execute this code block>
else:
      <Execute this code block>
```

*** Keep in mind the indentation levels for various code blocks.

Note:- We can have as many `elif` statements as we want, between the `if` and the `else` statements. This means we can consider as many conditions as we want. It should be noted that once an `if` or `elif` condition is executed, the remaining `elif` and `else` statements will not be executed.

Let us now consider an example to understand **If-Elif-Else** statements in depth.

Finding the largest among three numbers

Problem Statement: Given three numbers A, B and C, find the largest among the three and print it.

Approach: Let A, B, and C, be 3 numbers. We can construct an **if-elif-else** ladder. We have to consider the following conditions:

- If A is greater than or equal to both B and C, then A is the largest Number.
- If B is greater than or equal to both A and C, then B is the largest number.
- However, if none of these conditions is true it means that C is the largest number.

Thus, we get the following implementation for the above approach.

```
A = 10
B = 20
C = 30
if A>=B and A>= C:
      print(A)
elif B>=C and B>=A:
      print(B)
else:
      Print(C)
```

- Here since 10 is not greater than 20 and 30, the first `if` the condition is not satisfied. The code goes on to the `elif` condition.
- Now, 20 is also not greater than both 10 and 30, thus even the `elif` condition is not true. Thus, the else code block will now be executed.
- Thus the output will be 30, as it is the largest among the three. The `else` conditional block is executed.

The output of the above code will be:

```
30
```

Nested Conditionals

A nested `if` is an `if` statement that is present in the code block of another `if` statement. In other words, it means- an `if` statement inside another `if` statement. Yes, Python allows such a framework for us to nest `if` statements. Just like nested `if` statements, we can have all types of nested conditionals. A nested conditional will be executed only when the parent conditional is true.

The general syntax for a very basic nested `if` statement will be:

```
if <Condition 1>:
    # If Condition 1 is true then execute this code block
    if <Condition 2>:
        < If Condition 2 is True then execute this code block>
    else:
        < If Condition 2 is False then execute this code block>

else:
    # If Condition 1 is False then execute this code block>
    if <Condition 3>:
        < If Condition 3 is True then execute this code block>
    else:
        < If Condition 3 is False then execute this code block>
```

Note:- The conditions used in all of these conditional statements can be comprised of relational or logical operators. For example:-

```
A = True
B = False
if( A and B ):   # True and False = False
      print("Hello")
else:
      print("Hi") # This code block will be executed
```

The output of the above code will be:

```
Hi # Else statement block is executed
```

Introduction to While Loops

The while loop is somewhat similar to an `if` statement, it executes the code block inside if the expression/condition is `True`. However, as opposed to the `if` statement, the while loop continues to execute the code repeatedly, as long as the expression is `True`. In other words, a while loop iterates over a block of code.

In Python, the body of a `while` loop is determined by the indentation. It starts with indentation and ends at the first unindented line.

The most important part of a `while` loop is the looping variable. This looping variable controls the flow of iterations. An increment or decrement in this looping variable is important for the loop to function. It determines the next iteration level. In case of the absence of such increment/decrement, the loop gets stuck at the current iteration and continues forever until the process is manually terminated.

The general syntax of a `while` loop is similar to an `if` statement with a few differences. It is shown below:

```
while(Expression/Condition/Boolean):
    <Execute this code block till the Expression is True>
    #Increment/Decrement in looping variable
```

Example:

Problem Statement: Given an Integer n, Find the Sum of first n Natural numbers.

```
n = 4
sum = 0
i = 1 #Initialising the looping variable to 1
while (i<=n): #The loop will continue till the value of i<number
        sum = sum + i
        i = i+1 #Value of i is updated at the end of every iteration
print(sum)
```

We get the output as:

```
10
```

Check Prime: Using While Loop and Nested If Statements

Problem Statement: Given any Integer, check whether it is Prime or Not.

Approach to be followed: A prime number is always positive so we are checking that at the beginning of the program. Next, we are dividing the input number by all the numbers in the range of `2 to (number - 1)` to see whether there are any positive divisors other than 1 and the number itself (Condition for Primality). If any divisor is found then we display, **"Is Prime",** else we display, **"Is Not Prime".**

Note:- We are using the **`break`** statement in the loop to come out of the loop as soon as any positive divisor is found as there is no further check required. The purpose of a break statement is to break out of the current iteration of the loop so that the loop stops. This condition is useful, as once we have found a positive divisor, we need not check for more divisors and hence we can break out of the loop. You will study about the **`break`** statement in more detail in the latter part of this course.

```
# taking input from the user
number = int(input("Enter any number: "))

isPrime= True #Boolean to store if number is prime or not
if number > 1: # prime number is always greater than 1
    i=2
    while i< number:
        if (number % i) == 0: # Checking for positive divisors
            isPrime= False
            break
        i=i+1

if(number<=1): # If number is less than or equal to 1
    print("Is Not Prime")
elif(isPrime): # If Boolean is true
    print("Is Prime")
else:
    print("Is Not Prime")
```

Nested Loops

Python programming language allows the usage of one loop inside another loop. The loops can be nested the same way, the conditional statements are. The general syntax of such an arrangement is:

```
while(Expression1):
    <Execute this code block till the Expression1 is True>
    while(Expression2):
        <Execute this code block till the Expression2 is True>
        #Increment/Decrement in looping variable
    #Increment/Decrement in looping variable
```

**Keep the indentations in mind

Print All Primes- Using Nested Loops

Problem Statement: Given an Integer, Print all the Prime Numbers between 0 and that Integer.

Approach to be followed: Run a loop from `2 - n`, in order to check which all numbers in this range are prime. Let the value of the looping variable for this loop in some iteration be `i`. Run another loop inside this loop which will check if `i` is prime or not. This loop will run from 2 to `i` (Similar to the Check Prime Problem). This way we have a loop nested inside another.

```
n=int(input()) # Taking User Input
k=2 # Looping variable starting from 2
While k<=n:# Loop will check all numbers till n
      d=2 # The inner loop also checks all numbers starting from 2
      isPrime = False
      While d<k:
            if(k%d==0):
                  isPrime = True
            d=d+1
      if(not(isPrime)):
            print(k)
      k=k+1
```

Patterns

Introduction

Patterns are a handy application of loops and will provide you with better clarity and understanding of the implementation of loops.

Before printing any pattern, you must consider the following three things:

- The first step in printing any pattern is to figure out the number of rows that the pattern requires.
- Next, you should know how many columns are there in the i^{th} row.
- Once, you have figured out the number of rows and columns, then focus on the pattern to print.

For eg. We want to print the following pattern for N rows: **(Pattern 1.1)**

```
#For N=4:
****
****
****
****
```

Approach:

From the above pattern, we can observe:

➔ **Number of Rows:** The pattern has 4 rows. We have to print the pattern for N rows.

➔ **Number of Columns:** All the rows have 4 columns. Thus, in a pattern of N rows, all the rows will have N columns.

➔ **What to print:** We have to print `*` 4 times in all the 4 rows. Thus, in a pattern of N rows, we will have to print `*` N times in all the rows.

Now, let us discuss how to implement such patterns using Python.

Python Implementation for Patterns

We generally need two loops to print patterns. The outer loop iterates over the rows, while the inner nested loop is responsible for traversing the columns. The **algorithm** to print any pattern can be described as follows:

- Accept the number of rows or size of the pattern from a user using the `input()` function.
- Iterate the rows using the outer loop.
- Use the nested inner loop to handle the column contents. The internal loop iteration depends on the values of the outer loop.
- Print the required pattern contents using the `print()` function.
- Add a new line after each row.

The implementation of **Pattern 1.1** in Python will be:

Step 1: Let us first use a loop to traverse the rows. This loop will start at the first row and go on till the N^{th} row. Below is the implementation of this loop:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N row
    #<Here goes the Nested Loop>
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row
```

Printing a New Line: Since we need to print the pattern in multiple lines, we will have to add a new line after each row. Thus for this purpose, we use an empty `print()` statement. The `print()` function in Python, by default, ends in a new line.

Step 2: Now, we need another loop to traverse the row during each iteration and print the pattern; this can be done as follows:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=N: #Loop will on for N columns
        print("*",end="") #Printing a (*) in all columns
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row is printed
```

Printing in the same line:

The `print()` function in Python, by default, ends in a new line. This means that `print("*")`, would print `*` and a new line character. Now if anything is printed after this, it will be printed in a new line. However, If we have to print something in the same line, we will have to pass another argument (`end=`) in the `print()` statement. Thus, when we write the command `print("*", end="")`, Python prints a `*` and it ends in an empty string instead of a new line; this means that, when the next thing is printed, it will be printed in the same line as `*`.

There are two popular types of patterns-related questions that are usually posed:

- Square Pattern - **Pattern 1.1** is square.
- Triangular Pattern

Let us now look at the implementation of some common patterns.

Square Patterns

Pattern 1.2

```
# N = 5
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** All the rows have 5 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- ➔ **What to print:** All the entries in any row, are the same as the corresponding row numbers. Thus in a pattern of N rows, all the entries of the i^{th} row are `i` (1^{st} row has all 1's, 2^{nd} row has all 2's, and so on).

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=N: #Loop will on for N columns
        print(row,end="") #Printing the row number in all columns
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
```

```
    print() #Add a new Line after each row
```

Pattern 1.3

```
# N = 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** All the rows have 5 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- ➔ **What to print:** All the entries in any row, are the same as the corresponding column numbers. Thus in a pattern of N rows, all the entries of the i^{th} column are `i` (1^{st} column has all 1's, 2^{nd} column has all 2's, and so on).

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=N: #Loop will on for N columns
        print(col,end="") #Printing the column number in all columns
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row
```

Pattern 1.4

```
# N = 5
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** All the rows have 5 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- ➔ **What to print:** All the entries in any row, are `N-columnNumber+1`. Thus in a pattern of N rows, all the entries of the i^{th} column are `N-i+1` (1^{st} column has all 5's `(5-1+1)`, 2^{nd} column has all 4's `(5-2+1)`, and so on).

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=N: #Loop will on for N columns
        print(N-col+1,end="") #Printing (N-col+1) in all columns
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row
```

This way there can be several other square patterns and you can easily print them using this approach- **By finding the number of Rows, Columns and What to print**.

Pattern 1.5

```
# N = 5
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** All the rows have 5 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- ➔ **What to print:** The first entry in the 1st row is 1, the first entry in the 2nd row is 2, and so on. Further, these values are incremented continuously by 1 in the remaining entries of any particular row. Thus in a pattern of N rows, the first entry of the `i`th row is `i`. The remaining entries in the `i`th row are `i+1,i+2, and so on`. It can be observed that any entry in this pattern can be written as `row+col-1`.

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=N: #Loop will on for N columns
        print(col+row-1,end="") #Printing row+col-1 in all columns
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
```

```
    print() #Add a new Line after each row is printed
```

Triangular Patterns

Pattern 1.6

```
# N = 5
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** The number of columns in any row is the same as the corresponding row number.1st row has 1 column, 2nd row has 2 columns, and so on. Thus, in a pattern of N rows, the `i`th row will have `i` columns.
- ➔ **What to print:** All the entries in any row, are the same as the corresponding row numbers. Thus in a pattern of N rows, all the entries of the `i`th row are `i` (1st row has all 1's, 2nd row has all 2's, and so on).

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=row: # Number of cols = The row number
        print(row,end="") #Printing the row number in all columns
        col=col+1 #Increment the current column (Inner Loop)
```

```
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row
```

Pattern 1.7

```
# N = 5
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** The number of columns in any row is the same as the corresponding row number.1st row has 1 column, 2nd row has 2 columns, and so on. Thus, in a pattern of N rows, the `i`th row will have `i` columns.
- ➔ **What to print:** All the entries in any row, are the same as the corresponding column numbers. Thus in a pattern of N rows, all the entries of the `i`th column are `i` (1st column has all 1's, 2nd column has all 2's, and so on).

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=row: # Number of cols = The row number
        print(col,end="") #Printing the column number in all columns
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
```

```
    print() #Add a new Line after each row
```

Pattern 1.8

```
# N = 5
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 5 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** The number of columns in any row is the same as the corresponding row number.1st row has 1 column, 2nd row has 2 columns, and so on. Thus, in a pattern of N rows, the `i`th row will have `i` columns.
- ➔ **What to print:** The pattern starts with 1 and then each column entry is incremented by 1. Thus, we will initialize a variable `temp=1`. We will keep printing the value of `temp` in the successive columns and upon printing, we will increment the value of temp by 1.

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
temp=1
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=row: #Number of cols = The row number
        print(temp,end="") #Printing value of temp in all columns
        temp=temp+1
        col=col+1 #Increment the current column (Inner Loop)
```

```
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row is printed
```

Character Patterns

Pattern 1.9

```
# N = 4
ABCD
ABCD
ABCD
ABCD
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 4 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** All the rows have 4 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- ➔ **What to print:** The 1st column has all A's, 2nd column has all B's, and so on. The **ASCII** value of A is `65`. In the 1st column, the character corresponds to the **ASCII** value 65 `(64+1)`. In the 2nd column, the character corresponds to the **ASCII** value 66 `(64+2)`. Thus, all the entries in the `i`th column are equal to the character corresponding to the **ASCII** value **`64+i`**. The `chr()` function gives the character associated with the integral ASCII value within the parentheses.

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=N: #Loop will on for N columns
        print(chr(64+col),end="") #Printing a (*) in all columns
```

```
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row is printed
```

Pattern 1.10

```
# N = 4
ABCD
BCDE
CDEF
DEFG
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 4 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** All the rows have 4 columns. Thus, in a pattern of N rows, all the rows will have N columns.
- ➔ **What to print:** This pattern is very similar to **Pattern 1.5.** We can implement this using a similar code with a minor change. Instead of integers, we need capital letters of the same order. Instead of 1, we need A, instead of 2, we need B and so on. **ASCII** value of A is 65. Thus if we add 64 to all the entries in **Pattern 1.5** and find their **ASCII** values, we will get our result. The `chr()` function gives the character associated with the integral **ASCII** value within the parentheses.

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=N: #Loop will on for N columns
        print(chr(64+col+row-1),end="") # Adding 64 to all entries
```

```
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row is printed
```

Practice Problems

Here are a few similar patterns problems for your practice. All the patterns have been drawn for N=4.

```
A
AB
ABC
ABCD
```

```
12344321
123**321
12****21
1******1
```

```
ABCD
ABC
AB
A
```

```
4555
3455
2345
1234
```

```
1
11
202
3003
```

```
A
BB
CCC
DDDD
```

Patterns

Some Advanced Patterns

Pattern 2.1 - Inverted Triangle

```
# N = 3
* * *
* *
*
```

Approach:

From the above pattern, **we can observe:**

- ➔ **Number of Rows:** The pattern has 3 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** The number of columns in any row is equal to `N-rowNumber+1`.1st row has 3 columns `(3-1+1)`, 2nd row has 2 columns `(3-2+1)`, and so on. Thus, in a pattern of N rows, the `i`th row will have `N-i+1` columns.
- ➔ **What to print:** All the entries in any row are `"*"`.

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=N-row+1: #Number of columns = N-rowNumber+1
        print("*",end="") #Printing a (*) in all columns
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row is printed
```

Pattern 2.2 - Reversed Pattern

```
# N = 3
    *
  * *
* * *
```

Approach:

From the above pattern, **we can observe:**

- ➔ **Number of Rows:** The pattern has 3 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** The number of columns in any row is equal to `N`.
- ➔ **What to print:** In the 1st row, while `columnNumber <= 2(3-1)`, we print a `" "` in every column. Beyond the 2nd column, we print a `"*"`. Similarly, in the 2nd row, we print a `" "` till `columnNumber <=1(3-2)` and beyond the 1st column, we print a `"*"`. We can easily notice that if `col <= N-rowNumber`, we are printing a `" " (Space)`. And if `col > N-rowNumber`, we are printing a `"*"`.

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    col=1; #The loop starts with the first column in the current row
    while col<=N:#The loop will go on for N columns
        if(col<=N-row):
            print(" ",end="") #Printing a (" ")
        else:
            print("*",end="") #Printing a (*)
        col=col+1 #Increment the current column (Inner Loop)
    row=row+1 #Increment the current row (Outer Loop)
    print() #Add a new Line after each row is printed
```

Pattern 2.3 - Isosceles Pattern

```
# N = 4
   1
  121
 12321
1234321
```

Approach:

From the above pattern **we can observe:**

- ➔ **Number of Rows:** The pattern has 3 rows. We have to print the pattern for N rows.
- ➔ **Number of Columns:** Similar to Pattern 2.2, we first have `N-rowNumber` columns of spaces. Following this, we have `2*rowNumber-1` columns of numbers.
- ➔ **What to print:** We can notice that if `col <= N-rowNumber`, we are printing a `" " (Space)`. Further, the pattern has two parts. First is the increasing part and second is the decreasing part. For the increasing part, we will initialise a variable `num=1`. In each row we will keep printing `num` till its value becomes equal to the `rowNumber` . We will increment `num` by 1 after printing it; ;this will account for the first part of the pattern. We have `num = rowNumber` at this stage. Now, for the decreasing part, we will again start printing `num` till `num>=1`. After printing `num` we will decrement it by 1.

Python Implementation:

```
N=int(input()) #Take user input, N= Number of Rows
row=1; #The loop starts with the 1st row
while row<=N: #Loop will on for N rows
    spaces =1 # Printing spaces
    while spaces<= N- row:
        print(" ",end="")
        spaces=spaces+1

    num=1 #Variable to print the numbers
    while num<=row: #Increasing Pattern
        print(num,end="")
        num=num+1;

    num=row-1 # We have to start printing the decreasing part from
one less than the rowNumber
    while num>=1: #Decreasing Pattern
        print(num,end="")
        num=num-1
    print()#New Line
    row=row+1
```

Practice Problems

Here are a few similar patterns problems for your practice. All the patterns have been drawn for N=4.

```
   *
  ***
 *****
*******
```

```
   1
  121
 12321
1234321
 12321
  121
   1
```

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

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

More On Loops

`range()` Function

The `range()` function in Python generates a **List** which is a special sequence type. A sequence in Python is a succession of values bound together by a single name.

The syntax of the `range()` function is given below:

```
range(<Lower Limit>, <Upper Limit>) # Both limits are integers
```

The function, `range(L,U)`, will produce a list having values `L, L+1, L+2....(U-1)`.
Note: The lower limit is included in the list but the upper limit is not.
For Example:

```
>>> range(0,5)
[0,1,2,3,4] #Output of the range() function
```

Note: The default step value is +1, i.e. the difference in the consecutive values of the sequence generated will be +1.

If you want to create a list with a step value other than 1, you can use the following syntax:

```
range(<Lower Limit>, <Upper Limit>, <Step Value>)
```

The function, `range(L, U, S)`, will produce a list `[L, L+S, L+2S...<= (U-1)]`.
For Example:

```
>>> range(0,5,2)
[0,2,4] #Output of the range() function
```

`in` Operator

The `in` operator tests if a given value is contained in a sequence or not and returns `True` or `False` accordingly. For eg.,

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

will return `True` as value 3 is contained in the list.

```
"A" in "BCD"
```

will return `False` as `"A"` is not contained in the String `"BCD"`.

`for` Loop

The `for` loop of Python is designed to process the items of any sequence, such as a list or a string, one by one. The syntax of a `for` loop is:

```
for <variable> in <sequence>:
      Statements_to_be_executed
```

For example, consider the following loop:

```
for a in [1, 4, 7]:
      print(a)
```

The given `for` loop will be processed as follows:

1. Firstly, the looping variable `a` will be assigned the first value from the list i.e. 1, and the statements inside the for loop will be executed with this value of `a`. Hence 1 will be printed.
2. Next, `a` will be assigned 4 and 4 will be printed,
3. Finally, `a` will be assigned 7, and 7 will be printed.
4. All the values in the list are executed, hence the loop ends.

Check Prime: Using For Loop

Problem Statement: Given any Integer, check whether it is Prime or Not.

Approach to be followed:

- A prime number is always positive so we are checking that at the beginning of the program.
- Next, we are dividing the input number by all the numbers in the range of `2 to (number - 1)` to see whether there are any positive divisors other than 1 and the number itself (Condition for Primality).
- If any divisor is found then we display, **"Is Prime",** else we display, **"Is Not Prime".**

```
# taking input from the user
number = int(input("Enter any number: "))

isPrime= True
if number > 1: # prime number is always greater than 1
    for i in range(2, number):
        if (number % i) == 0: # Checking for positive divisors
            isPrime= False
            break

if(number<=1): # If the number is less than or equal to 1
    print("Is Not Prime")
elif(isPrime):
    print("Is Prime")
else:
    print("Is Not Prime")
```

Jump Statements: `break` and `continue`

Python offers two jump statements (within loops) to jump out of the loop iterations. These are `break` and `continue` statements. Let us see how these statements work.

`break` Statement

The `break` statement enables a program to skip over a part of the code. A `break` statement terminates the very loop it lies within.

The working of a `break` statement is as follows:

```
while <Expression/Condition/Statement>:
      #Statement1
      if <condition1>:
            break #If “condition” is true, then code breaks out
      #Statement2
      #Statement3
#Statement4: This statement is executed if it breaks out of the loop
#Statement5
```

In the given code snippet, if `condition1` is true, then the flow of execution will break out of the loop. The next statement to be executed will be `Statement4`.

Note: This loop can terminate in 2 ways:

1. Throughout the iterations of the loop, if `condition1` remains false, the loop will go through its normal course of execution and stops when its `condition/expression` becomes false.
2. If during any iteration, `condition1` becomes true, the flow of execution will break out of the loop and the loop will terminate.

The following code fragment shows you an example of the `break` statement:

```
for val in "string":
    if val == "i":
        break
    print(val)
print("The end")
```

The output will be:

```
s
t
r
```

Here, as soon as the value of `val` becomes equal to `"i"`, the loop terminates.

`continue` Statement

The `continue` statement jumps out of the current iteration and forces the next iteration of the loop to take place.

The working of a `continue` statement is as follows:

```
while <Expression/Condition/Statement>:
      #Statement1
      if <condition1>:
            continue
      #Statement2
      #Statement3
#Statement4: This statement is executed if it breaks out of the loop
#Statement5
```

In the given code snippet, if `condition1` is true, the `continue` statement will cause the skipping of `Statement2` and `Statement3` in the current iteration, and the next iteration will start.

The following code fragment shows you an example of the `continue` statement:

```
for val in "string":
    if val == "i":
        continue
    print(val)
print("The end")
```

The output will be:

```
s
t
r
n
g
The end
```

Here, the iteration with `val = "i"`, gets skipped and hence `"i"` is not printed.

`pass` Statement

The **`pass`** statement is a `null` statement.

- It is generally used as a placeholder for future code i.e. in cases where you want to implement some part of your code in the future but you cannot leave the space blank as it will give a compilation error.
- Sometimes, **`pass`** is used when the user does not want any code to execute.
- Using the **`pass`** statement in loops, functions, class definitions, and **`if`** statements, is very useful as empty code blocks are not allowed in these.

The syntax of a `pass` statement is:

```
pass
```

Given below is a basic implementation of a conditional using a `pass` statement:

```
n=2
if n==2:
   pass #Pass statement: Nothing will happen
else:
   print ("Executed”)
```

In the above code, the `if` statement condition is satisfied because `n==2` is `True`. Thus there will be no output for the above code because once it enters the `if` statement block, there is a `pass` statement. Also, no compilation error will be produced.

Consider another example:

```
n=1
if n==2:
   print ("Executed")
else:
   pass #Pass statement: Nothing will happen
```

In the above code, the `if` statement condition is not satisfied because `n==2` is `False`. Thus there will be no output for the above code because it enters the `else` statement block and encounters the `pass` statement.

Functions

What Is A Function?

A Function is a sequence of statements/instructions that performs a particular task. A function is like a black box that can take certain input(s) as its **parameters** and can output a value after performing a few operations on the parameters. A function is created so that one can use a block of code as many times as needed just by using the name of the function.

Why Do We Need Functions?

- **Reusability:** Once a function is defined, it can be used over and over again. You can call the function as many times as it is needed. Suppose you are required to find out the area of a circle for 10 different radii. Now, you can either write the formula πr^2 10 times or you can simply create a function that takes the value of the radius as an input and returns the area corresponding to that radius. This way you would not have to write the same code (formula) 10 times. You can simply invoke the function every time.
- **Neat code:** A code containing functions is concise and easy to read.
- **Modularisation:** Functions help in modularizing code. Modularization means dividing the code into smaller modules, each performing a specific task.
- **Easy Debugging:** It is easy to find and correct the error in a function as compared to raw code.

Defining Functions In Python

A function, once defined, can be invoked as many times as needed by using its name, without having to rewrite its code.
A function in Python is defined as per the following syntax:

```
def <function-name>(<parameters>):
      """ Function's docstring """
      <Expressions/Statements/Instructions>
```

- Function blocks begin with the keyword `def` followed by the **function name** and **parentheses** (( )).
- The input **parameters** or **arguments** should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function is optional - the documentation string of the function or **docstring**. The **docstring** describes the functionality of a function.
- The code block within every function starts with a **colon** (:) and is **indented**. All statements within the same code block are at the same indentation level.
- The `return` statement exits a function, optionally passing back an expression/value to the function caller.

Let us define a function to add two numbers.

```
def add(a,b):
      return a+b
```

The above function returns the sum of two numbers **a** and **b**.

The `return` Statement

A `return` statement is used to end the execution of the function call and it "returns" the result (value of the expression following the `return` keyword) to the caller. The statements after the return statements are not executed. If the `return` statement is without any expression, then the special value `None` is returned.

In the example given above, the sum `a+b` is returned.

Note: In Python, you need not specify the return type i.e. the data type of returned value.

Calling/Invoking A Function

Once you have defined a function, you can call it from another function, program, or even the Python prompt. To use a function that has been defined earlier, you need to write a **function call**.

A **function call** takes the following form:

```
<function-name> (<value-to-be-passed-as-argument>)
```

The function definition does not execute the function body. The function gets executed only when it is called or invoked. To call the above function we can write:

```
add(5,7)
```

In this function call, `a = 5` and `b = 7`.

Arguments And Parameters

As you know that you can pass values to functions. For this, you define variables to receive values in the **function definition** and you send values via a function call statement. For example, in the `add()` function, we have variables **a** and **b** to receive the values and while calling the function we pass the values 5 and 7. We can define these two types of values:

- **Arguments:** The values being passed to the function from the function call statement are called arguments. Eg. **5** and **7** are arguments to the `add()` function.
- **Parameters:** The values received by the function as inputs are called parameters. Eg. **a** and **b** are the parameters of the `add()` function.

Types Of Functions

We can divide functions into the following two types:

1. **User-defined functions:** Functions that are defined by the users. Eg. The `add()` function we created.
2. **Inbuilt Functions:** Functions that are inbuilt in python. Eg. The `print()` function.

Scope Of Variables

All variables in a program may not be accessible at all locations in that program. Part(s) of the program within which the variable name is legal and accessible, is called the scope of the variable. A variable will only be visible to and accessible by the code blocks in its scope.

There are broadly two kinds of scopes in Python −

- Global scope
- Local scope

Global Scope

A variable/name declared in the top-level segment **(`__main__`)** of a program is said to have a global scope and is usable inside the whole program (Can be accessed from anywhere in the program).

In Python, a variable declared outside a function is known as a global variable. This means that a global variable can be accessed from inside or outside of the function.

Local Scope

Variables that are defined inside a function body have a local scope. This means that local variables can be accessed only inside the function in which they are declared.

The Lifetime of a Variable

The lifetime of a variable is the time for which the variable exists in the memory.

- The lifetime of a Global variable is the entire program run (i.e. they live in the memory as long as the program is being executed).
- The lifetime of a Local variable is their function's run (i.e. as long as their function is being executed).

Creating a Global Variable

Consider the given code snippet:

```
x = "Global Variable"
def foo():
    print("Value of x: ", x)
foo()
```

Here, we created a global variable `x = "Global Variable"`. Then, we created a function `foo` to print the value of the global variable from inside the function. We get the output as:

```
Global Variable
```

Thus we can conclude that we can access a global variable from inside any function.

What if you want to change the value of a Global Variable from inside a function?

Consider the code snippet:

```
x = "Global Variable"
def foo():
    x = x -1
    print(x)
foo()
```

In this code block, we tried to update the value of the global variable **x**. We get an output as:

```
UnboundLocalError: local variable 'x' referenced before assignment
```

This happens because, when the command `x=x-1`, is interpreted, Python treats this **x** as a local variable and we have not defined any local variable **x** inside the function `foo()`.

Creating a Local Variable

We declare a local variable inside a function. Consider the given function definition:

```
def foo():
    y = "Local Variable"
    print(y)
foo()
```

We get the output as:

```
Local Variable
```

Accessing A Local Variable Outside The Scope

```
def foo():
    y = "local"
foo()
print(y)
```

In the above code, we declared a local variable **y** inside the function `foo()`, and then we tried to access it from outside the function. We get the output as:

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

We get an error because the lifetime of a local variable is the function it is defined in. Outside the function, the variable does not exist and cannot be accessed. In other words, a variable cannot be accessed outside its scope.

Global Variable And Local Variable With The Same Name

Consider the code given:

```
x = 5
def foo():
    x = 10
    print("Local:", x)
foo()
print("Global:", x)
```

In this, we have declared a global variable `x = 5` outside the function `foo()`. Now, inside the function `foo()`, we re-declared a local variable with the same name, **x**. Now, we try to print the values of **x**, inside, and outside the function. We observe the following output:

```
Local: 10
Global: 5
```

In the above code, we used the same name **x** for both global and local variables. We get a different result when we print the value of **x** because the variables have been declared in different scopes, i.e. the local scope inside `foo()` and global scope outside `foo()`.

When we print the value of the variable inside `foo()` it outputs `Local: 10`. This is called the local scope of the variable. In the local scope, it prints the value that it has been assigned inside the function.

Similarly, when we print the variable outside `foo()`, it outputs global `Global: 5`. This is called the global scope of the variable and the value of the global variable **x** is printed.

Python Default Parameters

Function parameters can have default values in Python. We can provide a default value to a parameter by using the assignment operator (=). Here is an example.

```
def wish(name, wish="Happy Birthday"):

"""This function wishes the person with the provided message. If the
message is not provided, it defaults to "Happy Birthday" """

    print("Hello", name + ', ' + wish)

greet("Rohan")
greet("Hardik", "Happy New Year")
```

Output

```
Hello Rohan, Happy Birthday
Hello Hardik, Happy New Year
```

In this function, the parameter `name` does not have a default value and is required (mandatory) during a call.

On the other hand, the parameter `wish` has a default value of `"Happy Birthday"`. So, it is optional during a call. If an argument is passed corresponding to the parameter, it will overwrite the default value, otherwise it will use the default value.

Important Points to be kept in mind while using default parameters:

- Any number of parameters in a function can have a default value.
- The conventional syntax for using default parameters states that once we have passed a default parameter, all the parameters to its right must also have default values.
- In other words, non-default parameters cannot follow default parameters.

For example, if we had defined the function header as:

```
def wish(wish = "Happy Birthday", name):
...
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```

Thus to summarise, in a function header, any parameter can have a default value unless all the parameters to its right have their default values.

Lists

Introduction

A list is a standard data type of Python that can store a sequence of values belonging to any type. The **Lists** are contained within square brackets (**[ ]**). Following are some examples of lists in Python:

```
[ ] #Empty list
[1, 2, 3] #List of integers
[1, 2, 5.6, 9.8] #List of numbers (Floating point and Integers)
['a', 'b', 'c'] #List of characters
['a', 1, 4.3, "Zero"] #List of mixed data types
["One", "Two", "Three"] #List of strings
```

Creating Lists

In Python programming, a list is created by placing all the items (elements) inside square brackets **[ ]**, separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

```
list1 = [] #Empty list
list2 = [1, 2, 3] #List of integers
list3 = [1, "One", 3.4] #List with mixed data types
```

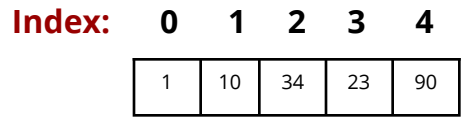
A list can also have another list as an element. Such a list is called a **Nested List**.

```
list4 = ["One", [8, 4, 6], ['Three']] #Nested List
```

Operations On Lists

Accessing Elements in a List

List indices start at 0 and go on till 1 less than the length of the list. We can use the index operator **[ ]** to access a particular item in a list. Eg.



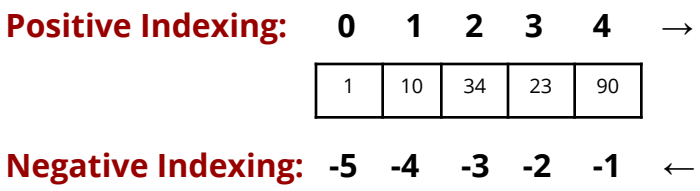
Note: Trying to access indexes out of the range `(0 ,lengthOfList-1)`, will raise an **IndexError**. Also, the index must be an integer. We can't use float or other types, this will result in **TypeError**.

Let us take an example to understand how to access elements in a list:

```
l1 = ['Mother', 'Father', 'Daughter', 10, 23]
>> print(l1[0]) #Output: 'Mother'
>> print(l1[2]) #Output: 'Daughter'
>> print(l1[4]) #Output: 23
```

Negative Indexing

Python allows negative indexing for its sequences. The index of **-1** refers to the last item, **-2** to the second last item, and so on. The negative indexing starts from the last element in the list.



Let us take an example to understand how to access elements using negative indexing in a list:

```
l1 = ['Mother', 'Father', 'Daughter', 10, 23]
>> print(l1[-1]) #Output: 23
>> print(l1[-2]) #Output: 10
>> print(l1[-6]) #Output: IndexOutOfRange error
```

Changing Elements of a List

Once a list is created, we can even change the elements of the list. This is done by using the assignment operator (**=**) to change the value at a particular list index. This can be done as follows:

```
l1 = ['Mother', 'Father', 'Daughter', 10, 23]
l1[-1] = "Daughter" #Changing the last element to "Daughter"
L1[3] = 12 #Changing the element at index 3 to 12
print(l1)
```

Output:

```
['Mother', 'Father', 'Daughter', 12, "Daughter"]
```

Concatenation of Lists

Joining or concatenating two list in Python is very easy. The concatenation operator (**+**), can be used to join two lists. Consider the example given below:

```
l1= [1,2,3] #First List
l2= [3,4,5] #Second List
```

```
l3= l1+l2 #Concatenating both to get a new list
print(l3)
```

Output:

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

Note: The **+** operator when used with lists requires that both the operands are of list types. You cannot add a number or any other value to a list.

Repeating/Replicating Lists

Like strings, you can use ***** operator to replicate a list specified number of times. Consider the example given below:

```
>>> l1 = [1,2, 10, 23]
>>> print(li*3)
[1,2,10,23,1,2,10,23,1,2,10,23] #Output
```

Notice that the above output has the same list `l1` repeated **3** times within a single list.

List Slicing

List slicing refers to accessing a specific portion or a subset of a list while the original list remains unaffected. You can use indexes of list elements to create list slices as per the following syntax:

```
slice= <List Name>[StartIndex : StopIndex : Steps]
```

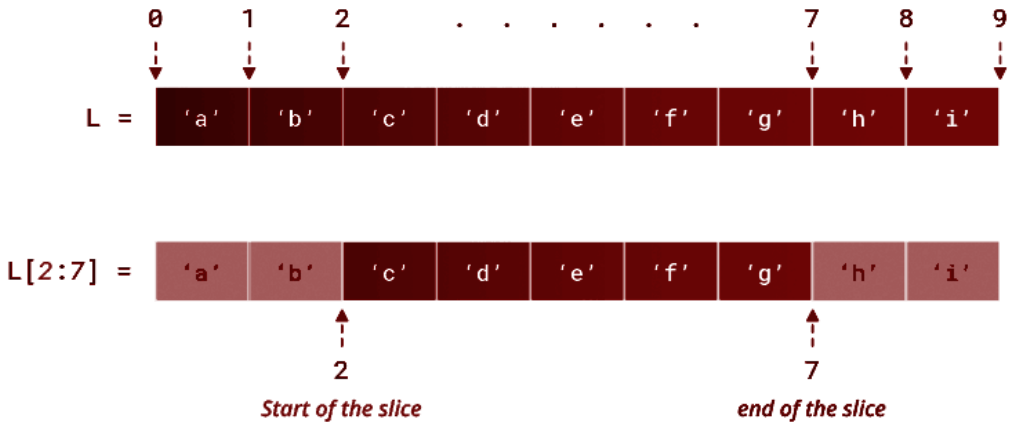
- The `StartIndex` represents the index from where the list slicing is supposed to begin. Its default value is 0, i.e. the list begins from index 0 if no `StartIndex` is specified.
- The `StopIndex` represents the last index up to which the list slicing will go on.

Its default value is `(length(list)-1)` or the index of the last element in the list.

- `steps` represent the number of steps. It is an optional parameter. `steps,` if defined, specifies the number of elements to jump over while counting from StartIndex to StopIndex. By default, it is 1.
- The list slices created, include elements falling between the indexes `StartIndex` and `StopIndex`, including `StartIndex` and not including `StopIndex`.

Here is a basic example of list slicing.

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[2:7])
```



As, you can see from the figure given above, we get the output as:

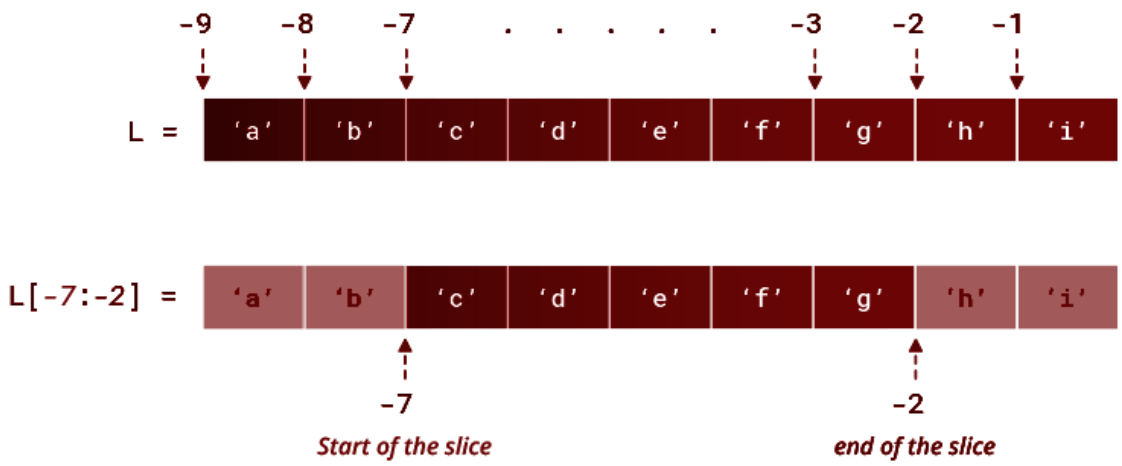
```
['c', 'd', 'e', 'f', 'g']
```

Slice Using Negative Indices

You can also specify negative indices while slicing a list. Consider the example given

below.

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[-7:-2])
```



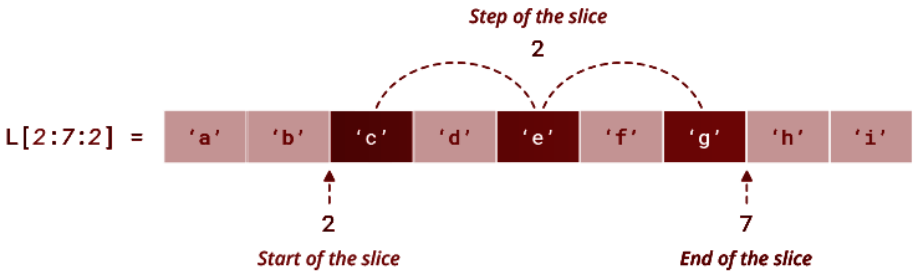
Thus, we get the output as:

```
['c', 'd', 'e', 'f', 'g]
```

Specify Step of the Slicing

You can specify the step of the slicing using the **`steps`** parameter. The **`steps`** parameter is optional and by default 1.

```
# Print every 2nd item between position 2 to 7
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[2:7:2])
```



The output will be:

```
['c','e','g']
```

You can even specify a negative step size:

```
# Print every 2nd item between position 6 to 1
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[6:1:-2])
```

The output will be:

```
['g','e','c']
```

Slice at Beginning & End

Omitting the `StartIndex` starts the slice from the index 0. Meaning, `L[:stop]` is equivalent to `L[0:stop]`.

```
# Slice the first three items from the list
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[:3])
```

Output

```
['a', 'b', 'c']
```

Whereas, omitting the **StopIndex** extends the slice to the end of the list. Meaning, **L[start:]** is equivalent to **L[start:len(L)]**.

```
# Slice the last three items from the list
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[6:])
```

Output

```
['g', 'h', 'i']
```

Reversing a List

You can reverse a list by omitting both **StartIndex** and **StopIndex** and specifying **steps** as -1.

```
L = ['a', 'b', 'c', 'd', 'e']
print(L[::-1])
```

Output:

```
['e', 'd', 'c', 'b', 'a'] #Reversed List
```

List Methods

append(): Used for appending/adding elements at the end of a list.

Syntax: `<ListName>.append(element)`

Example:

```
>>> li=[1,2,3,4]
>>> li.append(5) #Append 5 to the end of the list
>>> li
[1,2,3,4,5]
```

`extend()`: Adds the contents of **List2** to the end of **List1**.

Syntax: `<ListName1>.extend(<ListName2)`

Example:

```
>>> l1=[1,2,3,4]
>>> l2=[5,6,7,8]
>>> l1.extend(l2) #Adds contents of l2 to l1 at the end
>>> l1
[1,2,3,4,5,6,7,8]
```

`.append()` vs `.extend()`:

The only difference between `.append()` and `.extend()` is that, the `.append()` method adds an element at the back of a given list *(appends an element at the end of a list)*. On the other hand, the `.extend()` method adds the contents of another list at the end of the given list i.e. it merges two lists *(extends the given list).* See the examples given above for better clarity.

`insert()`: Inserts an element at a specified position/index in a list.

Syntax: `<ListName>(position, element)`

Example:

```
>>> li=[1,2,3,4]
>>> li.insert(2,5) #Insert 5 at the index no. 2
>>> li
[1,2,5,3,4]
```

`sum()` : Returns the sum of all the elements of a List. *(Used only for lists containing numerical values)*

Syntax: `sum(<ListName>)`

Example:

```
>>> l1=[1,2,3,4]
>>> sum1= sum(l1) #Finds the sum of all elements in l1
>>> sum1
10
```

`count()`: Returns the total number of times a given element occurs in a List.

Syntax: `<ListName>.count(element)`

Example:

```
>>> l1=[1,2,3,4,4,3,5,4,4,2]
>>> c= l1.count(4) #Number of times 4 occurs in the list
>>> c
4
```

`len()`: Returns the total length of a List.

Syntax: `len(<ListName>)`

Example:

```
>>> l1=[1,2,3,4,5]
>>> len(l1)
5
```

`index()`: Returns the index of first occurrence of an element in a list. If element is not present, it returns -1.

Syntax: `<ListName>.index(element)`

Example:

```
>>> l1=[1,2,3,4]
>>> l1.index(3)
2
```

`min()` : Returns the minimum element in a List.

Syntax: `min(<ListName>)`

Example:

```
>>> l1=[1,2,3,4]
>>> min(l1)
1
```

`max()`: Returns the maximum element in a List.

Syntax: `max(<ListName>)`

Example:

```
>>> l1=[1,2,3,4]
>>> max(l1)
4
```

`pop()`: It deletes and returns the element at the specified index. If we don't mention the index, it by default pops the last element in the list.

Syntax: `<ListName>.pop([index])`

Example:

```
>>> l1=[1,2,3,4]
```

```
>>> poppedElement= l1.pop(2)
>>> poppedElement #Element popped
3
>>> l1 #List after popping the element
[1,2,4]
```

Note: Index must be in range of the List, otherwise **IndexError** occurs.

`del()` : Element to be deleted is mentioned using list name and index.

Syntax: `del <ListName>[index]`

Example:

```
>>> l1=[1,1,12,3]
>>> del l1[2]
>>> l1
[1,1,3]
```

`remove()`: Element to be deleted is mentioned using list name and element.

Syntax: `<ListName>.remove(element)`

Example:

```
>>> l1=[1,1,12,3]
>>> l1.remove(12)
>>> l1
[1,1,3]
```

Looping On Lists

There are multiple ways to iterate over a list in Python.

Using `for` loop

```
li = [1, 3, 5, 7, 9]
# Using for loop
for i in li:
    print(i) #Print the element in the list
```

Output:

```
1
3
5
7
9
```

Using `for` loop and `range()`

```
list = [1, 3, 5, 7, 9]
length = len(list) #Getting the length of the list
for i in range(length): #Iterations from 0 to (length-1)
    print(i)
```

Output:

```
1
3
5
7
9
```

*You can even use **while()** loops. Try using the **while()** loops on your own.*

Taking Lists as User Inputs

There are two common ways to take lists as user inputs. These are:

- Space Separated Input of Lists
- Line Separated Input of Lists

Line Separated Input Of List

The way to take line separated input of a list is described below:

- Create an empty list.
- Let the number of elements you wish to put in the list be N.
- Run a loop for N iterations and during these N iterations do the following:
    - Take an element as user input using the `input()` function.
    - Append this element to the list we created.
- At the end of N iterations, you would have appended N desired elements to your list.
- In this, different elements will have to be entered by the user in different lines.

Consider the given example:

```
li=[] #Create empty list
for i in range(5): #Run the loop 5 times
      a=int(input()) #Take user input
      li.append(a) #Append it to the list
```

```
print(li) #Print the list
```

The above code will prompt the user to input 5 integers in 5 lines. These 5 integers will be appended to a list and the list will be printed.

Space Separated Input Of List

in Python, a user can take multiple values or inputs in one line by two methods.

- Using **`split()`** method
- Using List comprehension

Using `split()` method

This function helps in taking multiple inputs from the user in a single line . It breaks the given input by the specified **separator.** If a **separator** is not provided then any white space is treated as a separator.

Note: The `split()` method is generally used to split a string.

Syntax :

```
input().split(<separator>) #<separator> is optional
```

Example :

```
In[]: a= input().split()
In[]: print(a)
User[]: 1 2 3 4 5 #User inputs the data (space separated input)
Out[]: ['1','2','3','4','5']
```

Now, say you want to take comma separated inputs, then you will use `","` as the separator. This will be done as follows:

```
In[]: a= input().split(",")
In[]: print(a)
```

```
User[]: 1,2,3,4,5 #User inputs the data (space separated input)
Out[]: ['1','2','3','4','5']
```

Note: Observe that the elements were considered as characters and not integers in the list created using the `split()` function. *(What if you want a list of integers?- Think)*

Using List Comprehension

List comprehension is an elegant way to define and create a list in Python. We can create lists just like mathematical statements in one line only.

A common syntax to take input using list comprehension is given below.

```
inputList= [int(x) for x in input().split()]
```

Example :

```
In[1]: li= [int(x) for x in input().split()]
In[2]: print(li)
User[]: 1 2 3 4 5 #User inputs the data (space separated input)
Out[]: [1,2,3,4,5] #List has integers
```

Here, `In[1]` typecasts **x** in the list `input().split()` to an integer and then makes a list out of all these **x**'s.

Linear Search

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order.

Linear Search Algorithm

We have been given a `list` and a `targetValue`. We aim to check whether the given `targetValue` is present in the given `list` or not. If the element is present we are required to print the index of the element in the list and if the element is not present we print **-1**.

(First, let us implement this using a simple loop, then later we will see how to implement linear search using functions.)

The following are the steps in the algorithm:

1. Traverse the given list using a loop.
2. In every iteration, compare the `targetValue` with the value of the element in the list in the current iteration.
    - If the values match, print the current index of the list.
    - If the values do not match, move on to the next list element.
3. If no match is found, print -1.

Pseudo-Code

```
for each element in the array:#For loop to traverse the list
    if element == targetValue #Compare the targetValue to the element
         print(indexOf(item))
```

Python Code

```
li= [int(x) for x in input().split()] #Taking list as user input
targetValue= int(input()) #User input for targetValue
```

```
found = False #Boolean value to check if we found the targetValue
for i in li:
    if (i==targetValue): #If we found the targetValue
        print(li.index(i)) #Print the index
        found = True #Set found as True as we found the targetValue
        break #Since we found the targetValue, we break out of loop
if found is False:#If we did not find the targetValue
    print(-1)
```

We have:

```
User[1]: 1 2 4 67 23 12 #User input for list
User[2]: 4 #User input= targetValue
Out[]: 2 #Index of 4
```

Linear Search Through Functions

We will create a function `linearSearch`, which will have the following properties:

- Take `list` and `targetValue` as parameters.
- Run a loop to check the presence of `targetValue` in `list`.
- If it is present then it returns the index of the `targetValue`.
- If it is not present, then it returns **-1**.

We will call this function and then print the return value of the function.

Python Code

```
def linearSearch(li,targetValue):
```

```
    for i in li:
        if (i==targetValue): #If we found the targetValue
            return li.index(i) #Return the index
    return -1 #If not found, return -1
li= [int(x) for x in input().split()] #Taking list as user input
targetValue= int(input()) #User input for targetValue
print(linearSearch(li,targetValue)) #Print the return value
```

Mutable And Immutable Concept

The Python data types can be broadly categorized into *two* - ***Mutable*** and ***Immutable*** types. Let us now discuss these two types in detail.

- Since everything in Python is an Object, every variable holds an object instance.
- When an object is initiated, it is assigned a unique object id (Address in the memory).
- Its type is defined at runtime and once set it can never change.
- However, its state can be changed if it is **mutable**. In other words, the value of a **mutable** object can be changed after it is created, whereas the value of an **immutable** object can't be changed.

Note: *Objects of built-in types like (int, float, bool, str, tuple, Unicode) are* ***immutable****. Objects of built-in types like (list, set, dict) are* ***mutable****. A list is a mutable as we can insert/delete/reassign values in a list.*

Searching And Sorting

Searching

Searching means to find out whether a particular element is present in a given sequence or not. There are commonly two types of searching techniques:

- Linear search (We have studied about this in **Arrays and Lists**)
- Binary search

In this module, we will be discussing binary search.

Binary Search

Binary Search is a searching algorithm for finding an element's position in a sorted array. In a nutshell, this search algorithm takes advantage of a collection of elements of being already sorted by ignoring half of the elements after just one comparison.

Prerequisite: Binary search has one pre-requisite; unlike the linear search where elements could be any order, the array in **binary search** must be sorted,

The algorithm works as follows:

1. Let the element we are searching for, in the given array/list is X.
2. Compare X with the middle element in the array.
3. If X matches with the middle element, we return the middle index.
4. If X is greater than the middle element, then X can only lie in the right (greater) half subarray after the middle element, then we apply the algorithm again for the right half.
5. If X is smaller than the middle element, then X must lie in the left (lower) half, this is because the array is sorted. So we apply the algorithm for the left half.

Example Run

- Let us consider the array to be:

- Let x = 4 be the element to be searched.
- Set two pointers `low` and `high` at the first and the last element respectively.
- Find the middle element `mid` of the array ie. `arr[(low+high)/2] = 6`.

- If `x == mid`, then return `mid`. Else, compare the element to be searched with **m**.
- If `x > mid`, compare x with the middle element of the elements on the right side of mid. This is done by setting low to `low = mid + 1`.
- Else, compare x with the middle element of the elements on the left side of mid. This is done by setting high to `high = mid - 1`.

- Repeat these steps until low meets high. We found 4:

Python Code

```
#Function to implement Binary Search Algorithm
def binarySearch(array, x, low, high):

    #Repeat until the pointers low and high meet each other
    while low <= high:
        mid = low + (high - low) #Middle Index
        if array[mid] == x: #Element Found
            return mid
        elif array[mid] < x: #x is on the right side
            low = mid + 1
        else: #x is on the left side
            high = mid - 1
    return -1  #Element is not found

array = [3, 4, 5, 6, 7, 8, 9]
x = 4

result = binarySearch(array, x, 0, len(array)-1)

if result != -1: #If element is found
    print("Element is present at index " + str(result))
else: #If element is not found
    print("Not found")
```

We will get the **output** of the above code as:

```
Element is present at index 1
```

Advantages of Binary search:

- This searching technique is faster and easier to implement.
- Requires no extra space.
- Reduces the time complexity of the program to a greater extent. (The term **time complexity** might be new to you, you will get to understand this when you will be studying algorithmic analysis. For now, just consider it as the time taken by a particular algorithm in its execution, and time complexity is determined by the

number of operations that are performed by that algorithm i.e. time complexity is directly proportional to the number of operations in the program).

Sorting

Sorting is a permutation of a list of elements of such that the elements are either in increasing ***(ascending)*** order or decreasing ***(descending)*** order.

There are many different sorting techniques. The major difference is the amount of **space** and **time** they consume while being performed in the program.

For now, we will be discussing the following sorting techniques:

- Selection sort
- Bubble sort
- Insertion sort

Let us now discuss these sorting techniques in detail.

Selection Sort

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list. The detailed algorithm is given below.

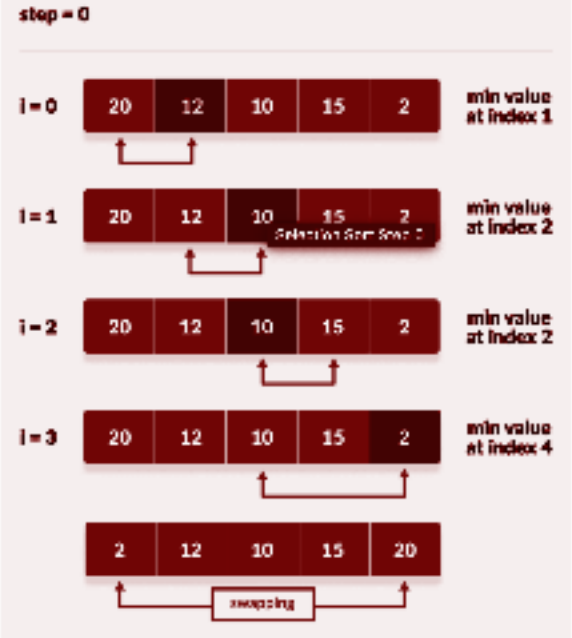
- Consider the given unsorted array to be:

- Set the first element as `minimum`.
- Compare `minimum` with the second element. If the second element is smaller than `minimum`, assign the second element as `minimum`.
- Compare `minimum` with the third element. Again, if the third element is smaller, then assign `minimum` to the third element otherwise do nothing. The process goes on until the last element.

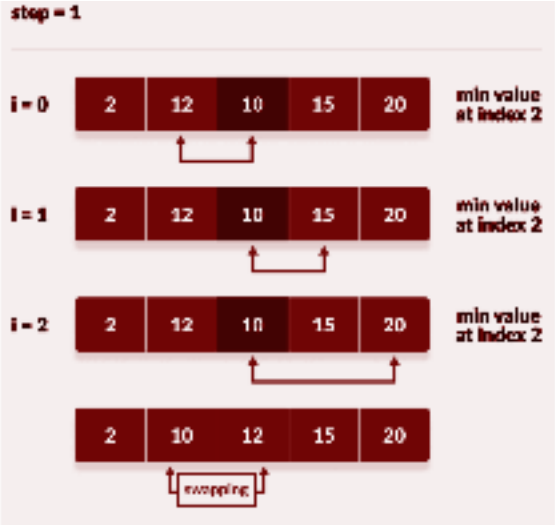
- After each iteration, `minimum` is placed in the front of the unsorted list.

- For each iteration, indexing starts from the first unsorted element. These steps are repeated until all the elements are placed at their correct positions.

First Iteration



Second Iteration:



Third Iteration

Fourth Iteration

Python Code

```
# Selection sort in Python
def selectionSort(arr, size):

    for step in range(size):
        minimum = step

        for i in range(step + 1, size):

            # to sort in descending order, change > to < in this line
            # select the minimum element in each loop
            if arr[i] < arr[minimum]:
                minimum = i

        # Swap
        (arr[step], arr[minimum]) = (arr[minimum], arr[step])


input = [20, 12, 10, 15, 2]
size = len(input)
selectionSort(input, size)
print('Sorted Array in Ascending Order: ')
print(input)
```

We will get the **output** of the above code as:

```
Sorted Array in Ascending Order: [2, 10, 12, 15, 20]
```

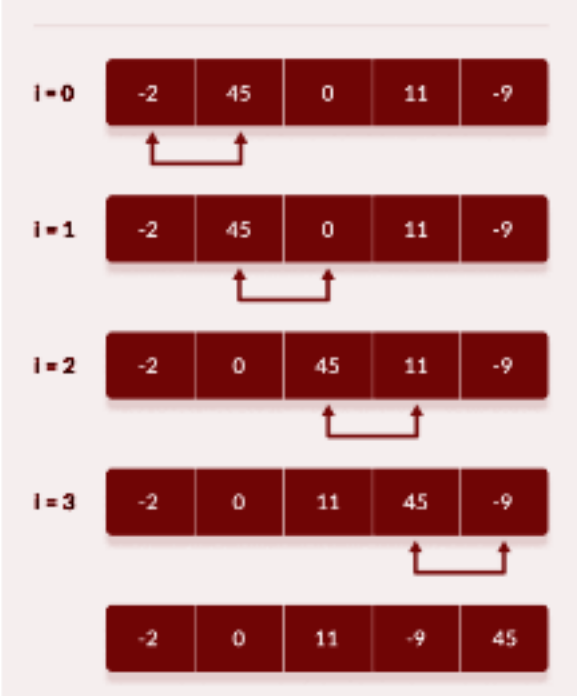
Bubble Sort

Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. The order can be ascending or descending.

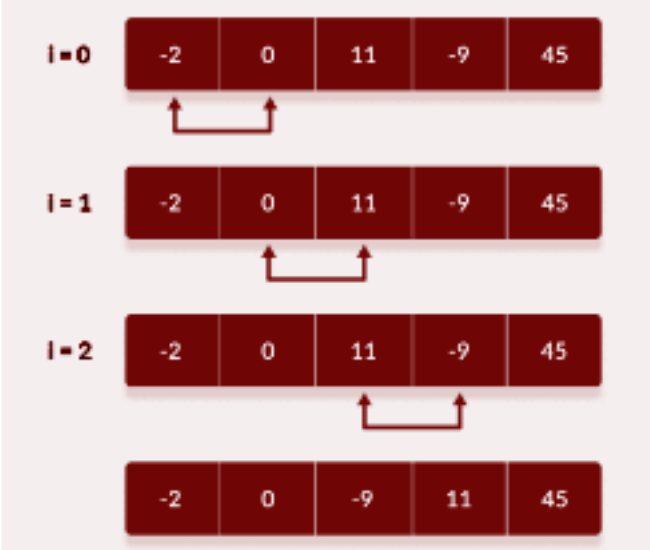
How does Bubble Sort work?

- Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are swapped.
- Now, compare the second and third elements. Swap them if they are not in order.
- The above process goes on until the last element.
- The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.
- In each iteration, the comparison takes place up to the last unsorted element.
- The array is sorted when all the unsorted elements are placed at their correct positions.

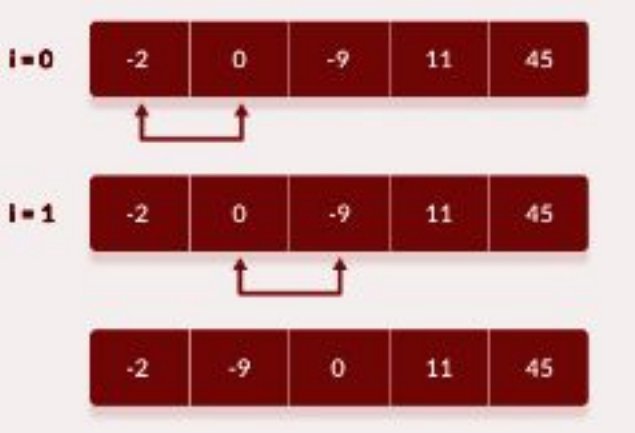
Let the array be `[-2, 45, 0, 11, -9]`.



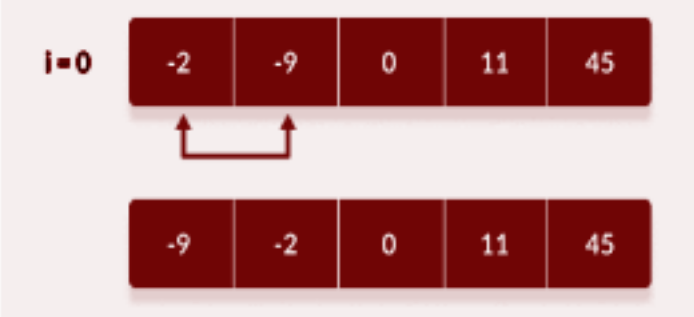
First Iteration



Second Iteration



Third Iteration



Fourth Iteration

Python Code

```
# Bubble sort in Python
def bubbleSort(arr):
    # run loops two times: one for walking through the array
    # and the other for comparison
```

```
    for i in range(len(arr)):
        for j in range(0, len(arr) - i - 1):
            # To sort in descending order, change > to < in this line.
            if arr[j] > arr[j + 1]:
                # swap if greater is at the rear position
                (arr[j], arr[j + 1]) = (arr[j + 1], arr[j])

input = [-2, 45, 0, 11, -9]
bubbleSort(input)
print('Sorted Array in Ascending Order:')
print(input)
```

We will get the **output** of the above code as:

```
Sorted Array in Ascending Order: [-9, -2, 0, 11, 45]
```

Insertion Sort

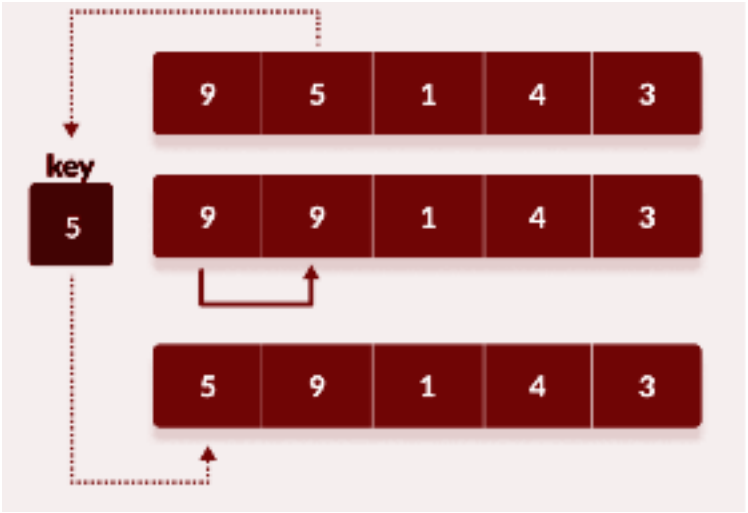
- Insertion sort works similarly as we sort cards in our hand in a card game.
- We assume that the first card is already sorted
- Then, we select an unsorted card.
- If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left.
- In the same way, other unsorted cards are taken and put in the right place. A similar approach is used by insertion sort.
- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration

Algorithm

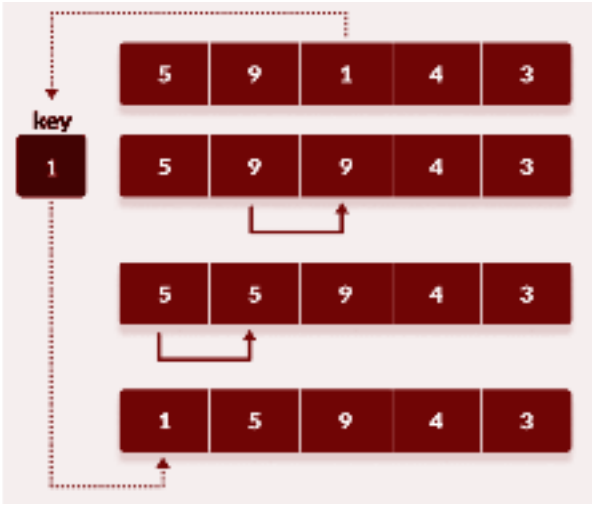
- Suppose we need to sort the following array.

- The first element in the array is assumed to be sorted. Take the second element and store it separately in **key**.
- Compare **key** with the first element. If the first element is greater than **key**, then **key** is placed in front of the first element.
- If the first element is greater than **key**, then **key** is placed in front of the first element.
- Now, the first two elements are sorted.
- Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.
- Similarly, place every unsorted element at its correct position.

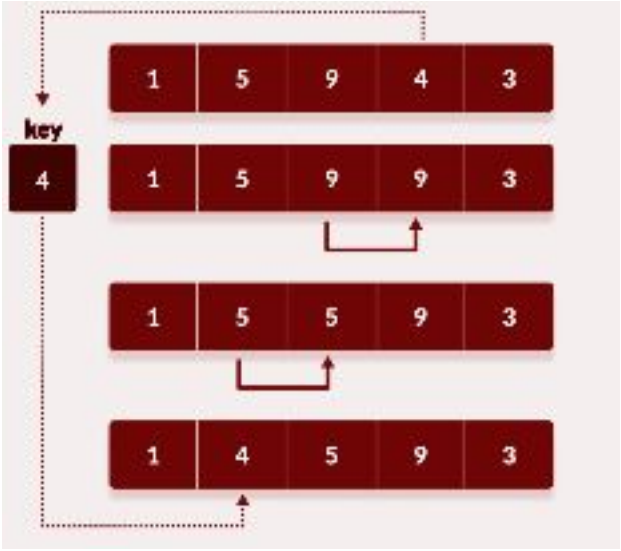
The various iterations are depicted below:



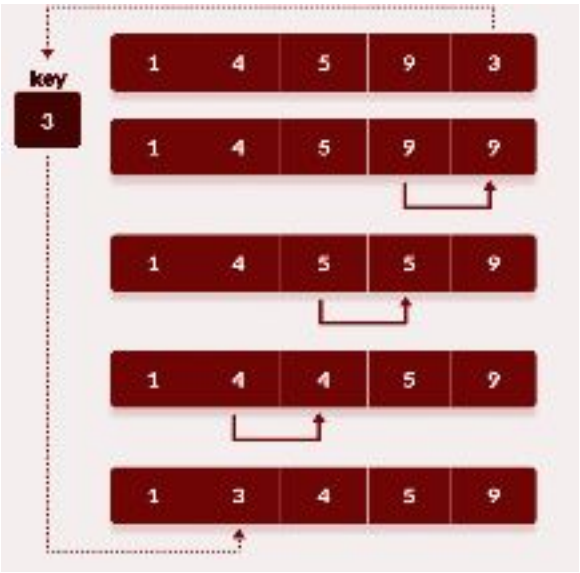
First Iteration



Second Iteration



Third Iteration



Fourth Iteration

Python Code

```
# Insertion sort in Python
def insertionSort(arr):

    for step in range(1, len(arr)):
        key = arr[step]
        j = step - 1

# Compare key with each element on the left of it until an element smaller
than it is found
        # For descending order, change key<array[j] to key>array[j].
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
```

```
            j = j - 1

        # Place the key at after the element just smaller than it.
        arr[j + 1] = key


input = [9, 5, 1, 4, 3]
insertionSort(input)
print('Sorted Array in Ascending Order:')
print(input)
```

We will get the **output** as:

```
Sorted Array in Ascending Order: [1, 3, 4, 5, 9]
```

Binary Search (Edge Case)

Remember the exploratory task we gave you?

Suppose your sorted array has duplicate numbers. For example, if the array is as follows: [1, 2, 2, 3, 3, 3, 4, 8, 9, 19, 19, 19].

Let's see how we can modify the code to find the first occurrence of the element **x** to be found.

In the current code, we exit when any of the following two conditions are satisfied

- When low < high, or
- When arr[mid] == x.

The modification we need is quite simple. Instead of exiting the code when arr[mid] == x, we shall assign **high** and **index** as **mid** as follows:

```
   index = mid
   high = mid
```

Let us see the entire code.

Python Code

```
#Function to implement Binary Search Algorithm
def binarySearch(array, x, low, high):

    low = 0;
    high = n
    mid = low
    index = -1

    while low < high :

        mid = low + (high - low) // 2

        if array[mid] > x :
            high = mid
        elif array[mid] < x :
            low = mid + 1
        else :
            index = mid
            high = mid

    return index

array = [1, 3, 3, 3, 3, 6, 7, 9, 9]
x = 3

result = binarySearch(array, x, 0, len(array)-1)

if result != -1: #If element is found
    print("Element is present at index " + str(result))
else: #If element is not found
    print("Not found")
```

Strings

Introduction

- A character is simply a symbol. For example, the English language has 26 characters; while a string is a contiguous sequence of characters.
- In Python, a string is a sequence of Unicode characters.
- Python strings are **immutable**, which means they cannot be altered after they are created.

Note: Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn more about Unicode from Python Unicode. Given below are some examples of Unicode Encoding:

`string ='pythön!'` → Default **UTF-8** Encoding: `b'pyth\xc3\xb6n!'`

`string ='python!'`→ Default **UTF-8** Encoding: `b'python!'`

How to create a string in Python?

Strings can be created by enclosing a sequence of characters inside a single quote or double-quotes.

```
# Defining strings in Python
# Both of the following are equivalent
s1 = 'Hello'
print(s1)
s2 = "Hello"
print(s2)
```

Output:

```
Hello
Hello
```

Note:

We can use triple quotes to create docstrings and/or multiline strings.

```
s3 = '''Hello
        World'''
print(s3) #Multiline string
```

Output

```
Hello
World
```

If you wish to print a string in a new line you can use the new line character `'\n'`. This is used within the strings. For example:

```
print('First line\nSecond line')
```

This would yield the result:

```
First Line
Second Line
```

Accessing Characters in a String

String indices start at 0 and go on till 1 less than the length of the string. We can use the index operator **[ ]** to access a particular character in a string. Eg.

Index:	**0**	**1**	**2**	**3**	**4**
String `"hello"`:	`"h"`	`"e"`	`"l"`	`"l"`	`"o"`

Note: Trying to access indexes out of the range `(0,lengthOfString-1)`, will raise an **IndexError**. Also, the index must be an integer. We can't use float or other data types;this will result in **TypeError**.

Let us take an example to understand how to access characters in a String:

```
s= "hello"
>>> print(s[0]) #Output: h
>>> print(s[2]) #Output: l
>>> print(s[4]) #Output: o
```

Negative Indexing

Python allows negative indexing for strings. The index of **-1** refers to the last character, **-2** to the second last character, and so on. The negative indexing starts from the last character in the string.

Positive Indexing:	**0**	**1**	**2**	**3**	**4**
String `"hello"`:	`"h"`	`"e"`	`"l"`	`"l"`	`"o"`
Negative Indexing:	**-5**	**-4**	**-3**	**-2**	**-1**

Let us take an example to understand how to access characters using negative indexing in a string:

```
s= "hello"
>>> print(s[-1]) #Output: o
>>> print(s[-2]) #Output: l
>>> print(s[-3]) #Output: l
```

Concatenation Of Strings

Joining of two or more strings into a single string is called string concatenation. The **+** operator does this in Python.

```
# Python String Operations
str1 = 'Hello'
str2 ='World!'
# using +
print('str1 + str2 = ', str1 + str2)
```

When we run the above program, we get the following output:

```
str1 + str2 =  HelloWorld!
```

Simply writing two string literals together also concatenates them.

```
>>> # two string literals together
>>> 'Hello ''World!'
'Hello World!'
```

If we want to concatenate strings in different lines, we can use parentheses.

```
>>> # using parentheses
>>> s = ('Hello '
...      'World')
>>> s
'Hello World'
```

Note: You cannot combine numbers and strings using the **+** operator. If you do so, you will get an error:

```
>>> "Hello" + 11
TypeError: can only concatenate str (not "int") to str
```

Repeating/Replicating Strings

You can use ***** operator to replicate a string specified number of times. Consider the example given below:

```
>>> s= "hello"
>>> print(s*3) #s*3 will produce a new string with s repeated thrice
hellohellohello #Output
```

Note: You cannot use ***** operator between two strings i.e. you cannot multiply a string by another string.

String Slicing

String slicing refers to accessing a specific portion or a subset of a string with the original string remaining unaffected. You can use the indexes of string characters to create string slices as per the following syntax:

```
slice= <String Name>[StartIndex : StopIndex : Steps]
```

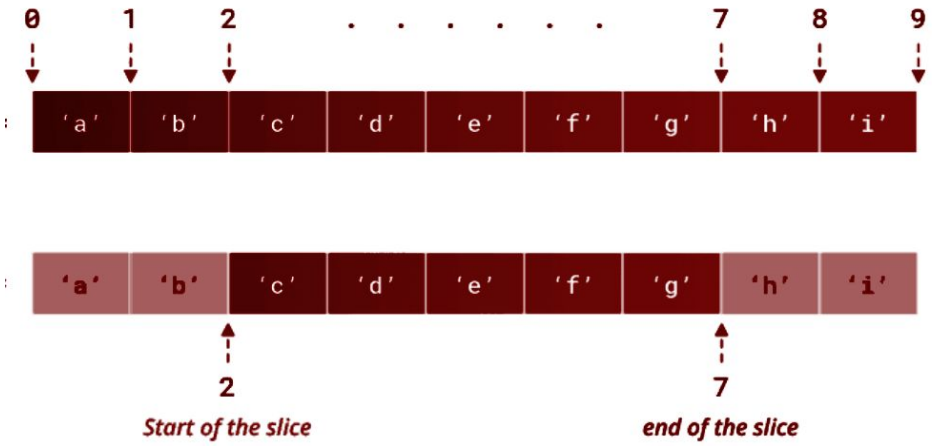
- The **`StartIndex`** represents the index from where the string slicing is supposed to begin. Its default value is 0, i.e. the string begins from index 0 if

no **`StartIndex`** is specified.

- The **`StopIndex`** represents the last index up to which the string slicing will go on. Its default value is **`(length(string)-1)`** or the index of the last character in the string.
- **`steps`** represent the number of steps. It is an optional parameter. **`steps,`** if defined, specifies the number of characters to jump over while counting from StartIndex to StopIndex. By default, it is 1.
- The string slices created, include characters falling between the indexes **`StartIndex`** and **`StopIndex`**, including **`StartIndex`** and not including **`StopIndex`**.

Here is a basic example of string slicing.

```
s = "abcdefghi"
print(s[2:7])
```



As you can see from the figure given above, we get the output as:

```
cdefg
```

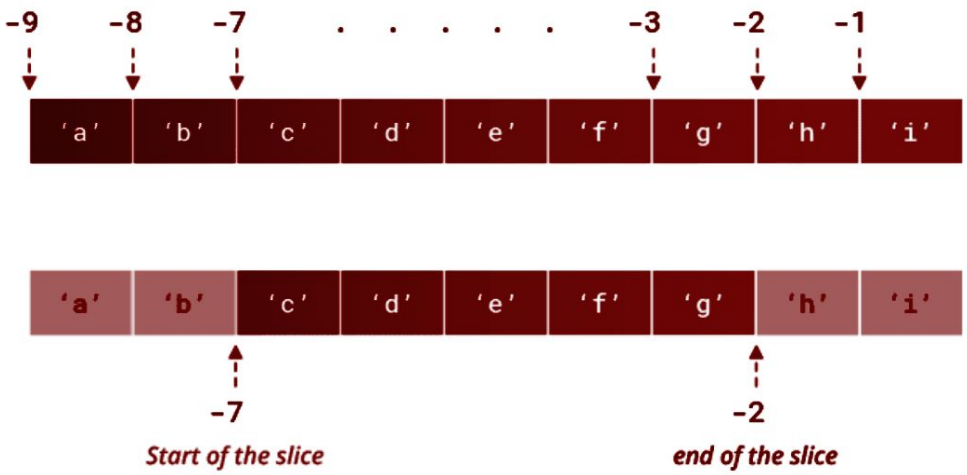
Slice Using Negative Indices

You can also specify negative indices while slicing a string. Consider the example given below.

```
s = "abcdefghi"
print(s[-7:-2])
```

Thus, we get the output as:

```
cdefg
```

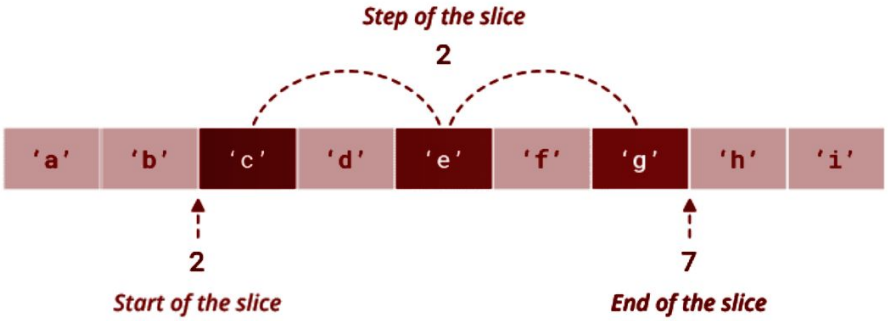


Specify Step of the Slicing

You can specify the step of the slicing using the `steps` parameter. The `steps` parameter is optional and by default 1.

```
# Every 2nd character between position 2 to 7
```

```
s = "abcdefghi"
print(s[2:7:2])
```



The output will be:

```
ceg
```

You can even specify a negative step size:

```
# Print every 2nd item between position 6 to 1
s = "abcdefghi"
print(s[6:1:-2])
```

The output will be:

```
gec
```

Slice at Beginning & End

Omitting the **`StartIndex`** starts the slice from the index 0. Meaning, **`S[:stop]`** is equivalent to **`S[0:stop]`**.

```
# Slice the first three items from the string
s = "abcdefghi"
print(s[:3])
```

Output

```
abc
```

Whereas, omitting the **`StopIndex`** extends the slice to the end of the string.

Meaning, **`S[start:]`** is equivalent to **`S[start:len(S)]`**.

```
# Slice the last three items from the string
s = "abcdefghi"
print(s[6:])
```

Output

```
ghi
```

Comparing Strings

1. In string comparison, we aim to identify whether two strings are equivalent to each other and if not, which one is greater.
2. String comparison in Python takes place character by character. That is, characters in the same positions are compared from both the strings.
3. If the characters fulfill the given comparison condition, it moves to the characters in the next position. Otherwise, it merely returns **False**.

Note: Some points to remember when using string comparison operators:

- The comparisons are **case-sensitive**, hence same letters in different letter cases(upper/lower) will be treated as separate characters.
- If two characters are different, then their Unicode value is compared; the character with the smaller Unicode value is considered to be lower.

This is done using the following operators:

- **`==`:** This checks whether two strings are equal
- **`!=`**: This checks if two strings are not equal

- `<`: This checks if the string on its left is smaller than that on its right
- `<=`: This checks if the string on its left is smaller than or equal to that on its right
- `>`: This checks if the string on its left is greater than that on its right
- `>=`: This checks if the string on its left is greater than or equal to that on its right

Iterating On Strings

There are multiple ways to iterate over a string in Python.

Using `for` loop

```
s="13579"
# Using for loop
for i in li:
    print(i) #Print the character in the string
```

Output:

```
1
3
5
7
9
```

Using `for` loop and `range()`

```
s="13579"
length = len(s) #Getting the length of the string
for i in range(length): #Iterations from 0 to (length-1)
    print(i)
```

Output:

```
1
3
5
7
9
```

*You can even use **while()** loops. Try using the **while()** loops on your own.*

Two Dimensional Lists

Introduction

- A Two-dimensional list is a list of lists.
- It represents a table/matrix with rows and columns of data.
- In this type of list, the position of a data element is referred to by two indices instead of one.

Consider an example of recording temperatures 4 times a day, 4 days in a row. Such data can be presented as a two-dimensional list as below.

```
Day 1 - 11, 12, 5, 2
Day 2 - 15, 6, 10, 6
Day 3 - 10, 8, 12, 5
Day 4 - 12, 15, 8, 6
```

The above data can be represented as a two-dimensional list as below.

```
T = [[11, 12, 5, 2], [15, 6, 10, 6], [10, 8, 12, 5], [12, 15, 8, 6]]
```

Accessing Values in a Two Dimensional list

The data elements in two-dimensional lists can be accessed using two indices. One index referring to the main or parent list (inner list) and another index referring to the position of the data element in the inner list. If we mention only one index then the entire inner list is printed for that index position.

The example below illustrates how it works.

```
T = [[11, 12, 5, 2], [15, 6, 10, 6], [10, 8, 12, 5], [12,15,8,6]]
print(T[0]) #List at index 0 in T
print(T[1][2]) #Element at index 2 in list at index 1 in T
```

When the above code is executed, it produces the following result −

```
[11, 12, 5, 2]
10
```

Updating Values in Two Dimensional list

We can update the entire inner list or some specific data elements of the inner list by reassigning the values using the list index.

```
T = [[11, 12, 5, 2], [15, 6, 10, 6], [10, 8, 12, 5], [12, 15, 8, 6]]
```

This can be done the same way as in 1D lists. Consider the examples given below:

```
T[0][1]= 1#Update the element at index 1 of list at index 0 of T to 1
T[1][1]= 7
```

This would modify the list as:

```
T= [[11, 1, 5, 2], [15, 7, 10, 6], [10, 8, 12, 5], [12, 15, 8, 6]]
```

Inserting Values in a Two Dimensional list

We can insert new data elements at any specific position by using the **`insert()`** method and specifying the index.

In the below example a new data element is inserted at index position 2.

```
T = [[11,12,5,2],[15,6,10,6],[10,8,12,5],[12,15,8,6]]
T.insert(2, [0,5,11,13])
```

This would modify the list as:

```
T = [[11,12,5,2],[15,6,10,6],[0,5,11,13],[10,8,12,5],[12,15,8,6]]
```

List Comprehension

List comprehensions are used for creating new lists from other iterable sequences. As list comprehensions return lists, they consist of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.

This is the basic syntax:

```
new_list = [expression for_loop_one_or_more conditions]
```

Finding squares of numbers in a list with the help of list comprehensions:

```
numbers = [1, 2, 3, 4]
squares = [n**2 for n in numbers]
print(squares)  # Output: [1, 4, 9, 16]
```

Here, square brackets signify that the output is a list. `n**2` is the expression executed for each element and `for n in numbers` is used to iterate over each element. In other words, we execute `n**2` (expression) for each element in **`numbers`**.

Find common numbers from two lists using list comprehension:

```
list_a = [1, 2, 3, 4]
list_b = [2, 3, 4, 5]
common_num = [a for a in list_a for b in list_b if a == b]
print(common_num) # Output: [2, 3, 4]
```

Here, we iterate over the two lists **`list_a`** and **`list_b`** and if a in **`list_a`** is equal to b in **`list_b`**, only then we add it to out output list.

Jagged Lists

Till now we have seen lists with an equal number of columns in all rows. Jagged lists are the two-dimensional lists with a variable number of columns in various rows. Various operations can be performed on such lists, the same way as in 2D lists with the same number of columns throughout. Some examples of jagged lists are shown below:

```
T1 = [[11,12,5,2],[10,6],[10,8,12],[12]]
T2 = [[1,2,3], 1, 3, [1,2]]
T3 = [[1],[],[1,2,1],0]
```

The elements in a jagged list can be accessed the same way as shown:

```
>>> T1[3]
[12]
>>> T1[0][2]
5
```

Note: Keep in mind the range of columns for various rows. Indexes out of the range can produce `indexOutOfRange` error.

Printing a Two Dimensional List

There are many ways to **print a 2D List**. Some of them are discussed below:

Simply Print the List

We can simple print a given list in a single line. This can be done by just using the `print()` function. This is shown as below:

```
T= [[11,12,5,2],[15,6,10,6],[0,5,11,13],[10,8,12,5],[12,15,8,6]]
print(T)
```

We get the output as:

```
[[11,12,5,2],[15,6,10,6],[0,5,11,13],[10,8,12,5],[12,15,8,6]]
```

Printing in Multiple Lines Like a Matrix

We can use nested `for` loops to print a 2D List. The outer loop iterates over the inner lists and the inner nested loop prints the elements of the lists. This can be shown as follows:

```
T= [[11,12,5,2],[15,6,10,6],[0,5,11,13],[10,8,12,5],[12,15,8,6]]
for row in T: #Every row in T is a list
    for col in row: #Every col is an element of the list row
        print(col, end=" ") #Print col
    print() #New Line
```

The above code will print T as:

```
11 12 5 2
15 6 10 6
0 5 11 13
10 8 12 5
12 15 8 6
```

Using List Comprehension and `.join()` method

`.join()`: The `.join()` string method returns a string by joining all the elements of an iterable sequence, separated by a string separator. The working is shown below:

```
>>> "-".join("abcd")
a-b-c-d
>>> "-".join(['a','b','c','d'])
a-b-c-d
```

The `join()` method produced a new string `"a-b-c-d"` from the given string and list.

Now, we can use this string method to print 2D Lists.

```
T= [[11,12,5,2],[15,6,10,6],[0,5,11,13],[10,8,12,5],[12,15,8,6]]
for row in T: #Every row in T is a list
    output = " ".join([str(col) for col in row])
    print(output)
```

The output will be:

```
11 12 5 2
15 6 10 6
0 5 11 13
10 8 12 5
12 15 8 6
```

Input Of Two Dimensional Lists

We will discuss two common ways of taking user input:

1. **Line Separated Input-** Different rows in different lines.
2. **Space Separated Input-** Taking input in a single line.

Line Separated Input:

We will use list comprehension to take user input. For every row, we take a list as input. The syntax would be:

```
n = int(input())
input= [[int(col) for col in input.split()] for row in range n]
print(input)
```

User Input:

```
5
11 12 5 2
15 6 10 6
0 5 11 13
10 8 12 5
12 15 8 6
```

Output:

```
[[11,12,5,2],[15,6,10,6],[0,5,11,13],[10,8,12,5],[12,15,8,6]]
```

Space Separated Input:

For this we will require the user to specify the number of rows (say `row`) and columns (say `col`) of the 2D list. The user will then enter all the elements of the 2D list in a single line. We will first convert this input stream to a list (say `inputL`). The next step would be using list comprehension to make a 2D list (say `fnlList`) from this list. It can be observed that `fnlList[i,j]=inputL[i*row+j]`. The python code for this can be written as follows:

```
row = int(input()) #Number of rows
col = int(input()) #Number of columns
inputL= input().split() #Converting input string to list
fnlList=[[int(inputL[i*col+j]) for j in range(col)] for i in range(row)]
print(fnlList)
```

User Input:

```
4
5
11 12 5 2 15 6 10 6 0 5 11 13 10 8 12 5 12 15 8 6
```

Output:

```
[[11,12,5,2],[15,6,10,6],[0,5,11,13],[10,8,12,5],[12,15,8,6]]
```

Tuples, Dictionaries, And Sets

Data Structures In Python

One can think of a data structure as a means of organizing and storing data such that it can be accessed and modified with ease. We have already seen the most conventional data types like strings and integers. Now let us take a deeper dive into the more unconventional data structures— **Tuples, Dictionaries, and Sets**.

Tuples

- A tuple is an ordered collection of elements/entries.
- Objects in a Tuple are immutable i.e. they cannot be altered once created.
- In a tuple, elements are placed within parentheses, separated by commas.
- The elements in a tuple can be of different data types like integer, list, etc.

How To Create A Tuple?

- A variable is simply assigned to a set of comma-separated values within closed parentheses.

```
>>> a=(1,2)
>>> b=("eat","dinner","man","boy")
>>> print(a)
(1,2)
>>> print(type(a))
<class 'tuple'>
```

- "()" parentheses are optional. If we assign a variable to a set of comma-separated values without parentheses, then by default, Python interprets it as a Tuple; This way of creating a Tuple is called **Tuple Packing**. It is always a good practice to use parentheses. This is shown in the given code snippet:

```
>>> c= 1,2,3,4,5
>>> print(c)
(1,2,3,4,5)
>>> print(type(c))
<class 'tuple'>
```

Note: If we assign a set of comma separated elements (or objects) to a set of comma separated variables, then these variables are assigned the corresponding values. **For Example:**

```
>>> e, f= "boy","man"
>>> print(e)
boy
>>> print(f)
man
```

Creating an Empty Tuple

In order to create an empty Tuple, simply assign a closed parentheses , which doesn't contain any element within, to a variable. This is shown below:

```
a=()
b=()
#Here 'a' and 'b' are empty tuples.
```

Creating A Tuple With A Single Element

- Creating a tuple with a single element is slightly complicated. If you try to create such a tuple by putting a single element within the parentheses, then it would not be a tuple, it would lead to the assignment of a single element to a variable.

```
>>> a=("Hamburger")
>>> print(type(a))
<class 'str'>
```

Here **'a'**, has a type **'string'**.

- If you try to create a tuple using the keyword `tuple`, you will get:

```
>>> tuple("Hamburger")
('H', 'a', 'm', 'b', 'u', 'r', 'g', 'e', 'r')
```

- To create such a tuple, along with a single element inside parentheses, we need a trailing comma. This would tell the system that it is in fact, a tuple.

```
>>> a=("Hamburger",)
>>> print(type(a))
<class 'tuple>
#Here a is indeed a tuple. Thus, the trailing comma is important in such an assignment.
```

Difference Between A Tuple and A List

Tuple	List
A Tuple is immutable	A list is mutable
We cannot change the elements of a Tuple once it is created	We can change the elements of a list once it is created
Enclosed within parentheses “( )”	Enclosed within square brackets “[ ]”

Accessing Elements of a Tuple

Indexing in a Tuple

- Indexing in a Tuple starts from index **0**.
- The highest index is the `NumberOfElementsInTheTuple-1`**.**
- So a tuple having **10** elements would have indexing from **0-9**.
- This system is just like the one followed in Lists.

Negative Indexing

- Python language allows negative indexing for its tuple elements.
- The last element in the tuple has an index **-1**, the second last element has index **-2**, and so on.

Let us define a Tuple:

```
myTemp=(1,15,13,18,19,23,4,5,2,3) #Number of elements=10
```

Element	1	15	13	18	19	23	4	5	2	3
Index	0	1	2	3	4	5	6	7	8	9
Negative indexing	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> print(myTemp[0])
1
>>> print(myTemp[8])
2
>>> print(myTemp[-1])
3
```

If any index out of this range is tried to be accessed, then it would give the following error.

```
>>> print(myTemp[10])
IndexError: tuple index out of range
>>> print(myTemp[20])
IndexError: tuple index out of range
>>> print(myTemp[-100])
IndexError: tuple index out of range
```

Slicing of a Tuple

- We can slice a Tuple and then access any required range of elements. Slicing is done by the means of a slicing operator which is ":".
- The basic format of slicing is:

```
<Name of Tuple>[start index: end index]
```

- This format by default considers the end index to be `endIndex-1`

```
>>> print(myTemp[1:4])
(15,13,18)
>>> print(myTemp[7:])
(5,2,3)
```

What Changes Can Be Made To A Tuple?

A tuple is immutable. This means that the objects or elements in a tuple cannot be changed once a tuple has been initialized. This is contrary to the case of lists - as a list is mutable, and hence the elements can be changed there.

- Python enables us to reassign the variables to different tuples.
- **For example**, the variable `myTemp` which has been assigned a tuple. To this variable some other tuple can be reassigned.

```
>>> myTemp = ([6,7],"Amity", 2, "boy")
```

- However, you cannot change any single element in the tuple.

```
>>> myTemp[2] = "Amity"
TypeError: 'tuple' object does not support item assignment
```

- Though, you can change items from any mutable object in the tuple.

```
>>> myTemp[0][1] = "Amity"
>>> print(myTemp)
([6,"Amity"],"Amity", 2, "boy")
```

- If you want to delete a tuple entirely, such an operation is possible.
- The keyword used is `del`

```
>>> del(myTemp)
>>> print(myTemp)
NameError: name 'myTemp' is not defined
# The name error shows that the tuple 'myTemp' has been deleted
```

Tuples Functions

- **We can use loops to iterate through a tuple:** For example, if we want to print all the elements in the tuple. We can run the following loop.

```
myTemp=(1,2,3)
for t in myTemp:
      print(t)
```

Output:

```
1
2
3
```

- **Checking whether the tuple contains a particular element:** The keyword used is `in`. We can easily get a boolean output using this keyword. It returns `True` if the tuple contains the given element, else the output is `False`.

```
>>> 10 in myTemp
False
>>> 4 in myTemp
True
```

- **Finding the length of a tuple:** We can easily find the number of elements that any tuple contains. The keyword used is `len`.

```
>>> print(len(myTemp))
6
```

- **Concatenation:** We can add elements from two different tuples and form a single tuple out of them. This process is concatenation and is similar to data types like string and list. We can use the **" + "** operator to combine two tuples.

```
a = (1,2,3,4)
b = (5,6,7,8)
d = a+b
print(d)
Out[]: (1,2,3,4,5,6,7,8)
```

We can also combine two tuples into another tuple in order to form a nested tuple. This is done as follows:

```
a = (1,2,3,4)
b = (5,6,7,8)
d = (a, b)
print(d)
--> ((1,2,3,4),(5,6,7,8))
# Here d is a nested tuple which is formed from 2 different tuples.
```

- **Repetition of a Tuple:** We can repeat the elements from a given tuple into a different tuple. The operator used is **" * "**, the multiplication operator. In other words, by using this operator, we can repeat the elements of a tuple as many times as we want.

```
>>> a = (1,2,3,4)
>>> print(a*4)
(1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4)
# The same tuple elements are repeated 4 times in the form of another
tuple.
```

- **Finding minimum or maximum element in a tuple:** To find the maximum element in a tuple, the keyword used is `max` and for finding the minimum element, the keyword used is `min`. These keywords return the maximum and the minimum elements of the tuple, respectively.

```
>>> a = (1,2,3,4)
>>> print(min(a))
1
>>> print(max(a))
4
```

Note: We can find the minimum and maximum of only those tuples, which have comparable entries. We cannot compare two different data types like a string and a tuple. Such comparisons would throw an error.

For example:

```
>>> s= (1,2,"string",4)
>>> print(min(s))
TypeError: '<' not supported between instances of 'string' and 'int'
```

```
>>> e= (1,2,2.6,4)
>>> print(min(e))
1
# Even though the data types are different , however since a float can be compared to a floating point value, hence this operation does not give an error.
```

- **Converting a list to a tuple:** We can typecast a list into a tuple. The keyword used is `tuple`. This is done as follows:

```
>>> myList= [1,2,3,4]
>>> myTuple= tuple(myList)
>>> print(myTuple)
(1,2,3,4)
```

Using Tuples For Variable Length Input And Output

Variable Length Inputs

There are some situations where we need to give a variable number of inputs to some functions. The use of tuples in such situations has proved to be highly efficient.

Task 1: Giving a variable number of inputs and printing them:

```
def printNum(a, b,*more):
      print(a)
      print(b)
      print(more)
printNum(1,2,3,4,5,5)
```

Output

```
1
2
(3,4,5,5)
```

- We use `*more` as the third parameter.
- The first two arguments are taken as the first two parameters and hence are printed individually. However, all the arguments after them, are taken as a single tuple and hence are printed in the form of a tuple.

Task 2: Finding the sum of a variable number of inputs:

Consider an example in which we have to calculate the sum of a variable number of inputs. In such a situation we cannot practically have multiple parameters in the function. This can be done as follows:

```
def printNum(a, b,*more):
      sum=a+b
      for t in more: #Traverse the tuple *more
            sum=sum+t #Add all elements in *more
      return sum
printNum(1,2,3,4,5,5)
Out[]: 20
```

Variable Length Outputs

- Following the conventional ways, we can return only a single value from any function. However, with the help of tuples, we can overcome this disadvantage.
- Tuples help us in returning multiple values from a single function.
- This can be done by returning comma-separated-values, from any function.
- On being returned, these comma-separated values act as a tuple.
- We can access the various entries from this returned tuple. This can be shown as:

```
def sum_diff(a, b):
      return a+b, a-b #Return the sum and difference together
print(sum_diff(1,2))
Out[]: (3,-1)
```

Dictionaries

- A Dictionary is a Python's implementation of a data structure that is more generally known as an associative array.
- A dictionary is an unordered collection of key-value pairs.
- Indexing in a dictionary is done using these **"keys".**
- Each pair maps the key to its value.
- Literals of the type dictionary are enclosed within curly brackets.
- Within these brackets, each entry is written as a key followed by a colon " : ", which is further followed by a value. This is the value that corresponds to the given key.
- These keys must be unique and cannot be any immutable data type. **Eg-** string, integer, tuples, etc. They are always mutable.
- The values need not be unique. They can be repetitive and can be of any data type (Both mutable and immutable)
- Dictionaries are mutable, which means that the key-value pairs can be changed.

What does a dictionary look like?

This is the general representation of a dictionary. Multiple key-value pairs are enclosed within curly brackets, separated by commas.

```
myDictionary = {
    <key>: <value>,
    <key>: <value>,
      .
      .
    <key>: <value>
}
```

Creating A Dictionary

Way 1- The Basic Approach

Simply put pairs of key-value within curly brackets. Keys are separated from their corresponding values by a colon. Different key-value pairs are separated from each other by commas. Assign this to a variable.

```
myDict= {"red":"boy", 6: 4, "name":"boy"}
months= {1:"January", 2:"February", 3: "March"}
```

Way 2- Type Casting

This way involves type casting a list into a dictionary. To typecast, there are a few mandates to be followed by the given list.

- The list must contain only tuples.
- These tuples must be of length **2**.
- The first element in these tuples must be the key and the second element is the corresponding value.

This type casting is done using the keyword **`dict`.**

```
>>> myList= [("a",1),("b",2),("c",2)]
>>> myDictionary= dict(myList)
>>> print(myDictionary)
{"a":1,"b":2,"c":2}
```

Way 3- Using inbuilt method `.copy()`

We can copy a dictionary using the method **`.copy()`**. This would create a shallow copy of the given dictionary. Thus the existing dictionary is copied to a new variable. This method is useful when we wish to duplicate an already existing dictionary.

```
>>> myDict= {"a":1,"b":2,"c":2}
>>> myDict2= myDict.copy()
>>> print(myDict2)
{"a":1,"b":2,"c":2}
```

Way 4- Using inbuilt method `.fromkeys()`

This method is particularly useful if we want to create a dictionary with variable keys and all the keys must have the same value. The values corresponding to all the keys is exactly the same. This is done as follows:

```
>>> d1= dict.fromkeys(["abc",1,"two"])
>>> print(d1)
{"abc":None ,1: None, "two": None}
# All the values are initialized to None if we provide only one argument
i.e. a list of all keys.
```

```
# We can initialise all the values to a custom value too. This is done by
providing the second argument as the desired value.
>>> d2= dict.fromkeys(["abc",1,"two"],6)
>>> print(d2)
["abc":6 ,1:6, "two":6]
```

How to access elements in a dictionary?

We already know that the indexing in a dictionary is done with the keys from the various key-value pairs present within. Thus to access any value we need to use its index i.e. it's **key**.

Similar to the list and tuples, this can be done by the square bracket operator **[ ]**.

```
foo= {"a":1,"b":2,"c":3}
print(foo["a"])
--> 1
print(foo["c"])
--> 3
```

If we want the value corresponding to any key , we can even use an inbuilt dictionary method called **`get`**.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> print(foo.get("a"))
1
>>> print(foo.get("c"))
3
```

A very unique feature about this method is that , incase the desired **key** is not present in the dictionary , it won't throw an error or an exception. It would simple return **`None`**.

We can make use of this feature in another way. Say we want the method to do the following action: If the key is present in the dictionary then return the value corresponding to the key. In case the key is not present, return a custom desired value (say 0).

This can be done as follows:

```
>>> foo= {"a":1,"b":2,"c":3}
>>> print(foo.get("a",0))
1
>>> print(foo.get("d",0))
0
```

Accessing all the available keys in the dictionary:

This can be done using the method **`.keys()`**. This method returns all the different keys present in the dictionary in the form of a list.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.keys()
dict_keys(["a","b","c"])
```

Accessing all the available values in the dictionary:

This can be done using the method **`.values()`**. This method returns all the different values present in the dictionary in the form of a list.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.values()
dict_values([1,2,3])
```

Accessing all the available items in the dictionary:

This can be done using the method `.items()`. This method returns all the different items (key-value pairs) present in the dictionary in the form of a list of tuples, with the first element of the tuple as the key and the second element as the value corresponding to this key .

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.items()
dict_items([("a",1),("b",2),("c",3)])
```

Checking if the dictionary contains a given key:

The keyword used is `in`. We can easily get a boolean output using this keyword. It returns True if the dictionary contains the given **key**, else the output is False. This checks the presence of the keys and not the presence of the values.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> "a" in foo
True
>>> 1 in foo
False
```

Iterating over a Dictionary:

In order to traverse through the dictionary, we can use a simple for loop. The loop will go through the keys of the dictionary one by one and do the required action.

```
bar= {2:1,3:2,4:3}
for t in bar:
      print(t)
Out[]:
2
3
4
# Here t is the key in the dictionary and hence when we print t in all
iterations then all the keys are printed.
```

```
for t in bar:
      print(t, bar[t])
Out[]:
2 1
3 2
4 3
# Here along with the keys, the values which are bar[t] are printed. In
this loop, the values are accessed using the keys.
```

Adding Elements In a Dictionary

Since a dictionary is mutable, we can add or delete entries from the dictionary. This is particularly useful if we have to maintain a dynamic data structure. To assign a value corresponding to a given key *(This includes over-writing the value present in the key or adding a new key-value pair)*, we can use the square bracket operators to simply assign the value to a key.

If we want to update the value of an already existing key in the dictionary then we can simply assign the new value to the given key. This is done as follows:

```
>>> bar= {2:1,3:2,4:3}
>>> bar[3]=4
# This operation updates the value of the key 3 to a new value i.e. 4.
>>> print(bar)
{2:1,3:4,4:3}
```

Now if we want to add a new key-value pair to our dictionary, then we can make a similar assignment. If we have to add a key-value pair as `"man":"boy"`, then we can make the assignment as:

```
>>> bar["man"]="boy"
>>> print(bar)
{2:1,3:2,4:3,"man":"boy"}
```

Adding or concatenation of two dictionaries:

If we have 2 dictionaries and we want to merge the contents of both the dictionaries and form a single dictionary out of it . It is done as follows:

```
a= {1:2,2:3,3:4}
b= {7:2,10:3,6:4}
a.update(b)
print(a)
--> {1:2,2:3,3:4,7:2,10:3,6:4}
```

In this process, the second dictionary is unchanged and the contents of the second dictionary are copied into the first dictionary. The uncommon keys from the second dictionary are added to the first with their corresponding values. However, if these dictionaries have any common key, then the value of the common key present in the first dictionary is updated to the new value from the second.

Deleting an entry:

In order to delete an entry corresponding to any key in a dictionary , we can simple pop the key from the dictionary. The method used here is `.pop()` . This method removes the key-value pair corresponding to any particular key and then returns the value of the removed key-value pair.

```
>>> c={1:2,2:(3,23,3),3:4}
>>> c.pop(2)
(3,23,3)
```

Deleting all the entries from the dictionary:

If we want to clear all the key-value pairs from the given dictionary and thus convert it into an empty dictionary we can use the `.clear()` method.

```
>>> c={1:2,2:(3,23,3),3:4}
>>> c.clear()
>>> print(c)
{}
```

Deleting the entire dictionary:

We can even delete the entire dictionary from the memory by using the `del` keyword. This would remove the presence of the dictionary. This is similar to tuples and lists.

Problem statement: Print all words with frequency k.

Approach to be followed:

First, we convert the given string of words into a list containing all the words individually. Some of these words are repetitive and to find all the words with a specific frequency, we convert this list into a dictionary with all the unique words as keys and their frequencies or the number of times they occur as their values.

To convert the string to a list, we use the `.split()` function. This gives us a list of words. Now, we run a loop through this list and keep making changes to the frequency in the dictionary. If the word in the current iteration already exists in the dictionary as a key, then we simply increase the value(or frequency) by 1. If the key does not exist, we create a new key-value pair with value as 1.

Now we have a dictionary with unique keys and their respective frequencies. Now we run another loop to print the keys with the frequency '**k**'.

Given below is a function that serves this purpose.

```
def printKFreqWords(string, k):
      # Converting the input string to a list
      myList= string.split()
      # Initialise an empty dictionary
      dict= {}
      # Iterate through the list in order to find frequency
      for i in myList:
            dict[i]=dict[i]+1
      else:
            dict[i]=1
      # Loop for printing the keys with frequency as k
      for t in dict:
            if dict[t]==k:
                  print(t)
```

Sets

Mathematically a set is a collection of items *(not in any particular order)*. A Python set is similar to this mathematical definition with below additional conditions.

- The elements in a set cannot be repeated i.e. an element can occur only once.
- The elements in the set are immutable*(cannot be modified)* but the set as a whole is mutable.
- There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

Set Operations

The sets in python are typically used for mathematical operations like union, intersection, difference, and complement, etc. We can create a set, access its elements, and carry out these mathematical operations as shown below.

Creating a set

A set is created by using the `set()` function or placing all the elements within a pair of curly braces, separated by commas.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days)
print(Months)
print(Dates)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb'])
set([17, 21, 22])
```

Note: The order of the elements has changed in the result.

Accessing Values in a Set

We cannot access individual values in a set as there is no specific order of elements in a set. We can only access all the elements together as shown. We can also get a list of individual elements by looping through the set.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
for d in Days:
      print(d)
```

When the above code is executed, it produces the following result.

```
Wed
Sun
Fri
Tue
Mon
Thu
Sat
```

Adding Items to a Set

We can add elements to a set by using the `add()` method. Again as discussed there is no specific index attached to the newly added element.

```
>>> Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
>>> Days.add("Sun")
>>> print(Days)
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Removing Item from a Set

We can remove elements from a set by using the `discard()` method. Again as discussed there is no specific index attached to the newly added element.

```
>>> Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
>>> Days.discard("Sun")
>>> print(Days)
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Union of Sets

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element `"Wed"` is present in both the sets. The union operator is **'|'**.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA|DaysB #Union of Sets
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one `"wed"`.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Intersection of Sets

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element `"Wed"` is present in both the sets. The intersection operator is **"&"**.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA & DaysB #Intersection of Sets
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one `"wed"`.

```
set(['Wed'])
```

Difference of Sets

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element `"Wed"` is present in both the sets so it will not be found in the result set. The operator used is **"-"**.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA - DaysB #Difference of Sets
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one `"wed"`.

```
set(['Mon', 'Tue'])
```

Compare Sets

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
SubsetRes = DaysA <= DaysB #Check Subset
SupersetRes = DaysB >= DaysA #Check Superset
print(SubsetRes)
print(SupersetRes)
```

When the above code is executed, it produces the following result.

```
True
True
```