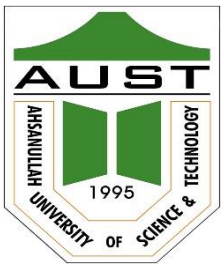


Ahsanullah University of Science and Technology (AUST)
Department of Computer Science and Engineering

LABORATORY MANUAL

Course No.: CSE 1278
Course Title: Computer Programming and Algorithms Sessional

For the students of 1st Year, 2nd semester of
B.Sc. in Industrial and Production Engineering program

TABLE OF CONTENTS

COURSE LEARNING OUTCOMES

After the successful completion of this course, students will be able to:

1. Explain the basic concepts of programming
2. Apply the program solving skills and algorithms to solve a wide range of common programming problems
3. Analyze and synthesize different problem-solving methodologies

PREFERRED TOOL(S)

1. Python IDLE
2. VS Code
3. Google Colab

TEXT/REFERENCE BOOK(S)

1. Luciano Ramalho, ***Fluent Python***
2. Charles Severance, ***Python for Everybody: Exploring Data in Python 3***

ADMINISTRATIVE POLICY OF THE LABORATORY

1. Students must perform class assessment tasks individually
2. Viva will be taken for each assignment and marks on assignment will substantially depend on viva
3. Plagiarism is strictly forbidden and will be dealt with punishment

Python Programming Language

It was created by Guido van Rossum, and released in 1991. Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc). It has a simple syntax similar to the English language that allows developers to write programs with fewer lines than some other programming languages. Python can be used to handle big data and perform complex mathematics. The most recent major version of Python is Python 3. Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses. It relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

First Python Program

Code	Output
print("Hello, world!")	Hello, world!

Python Indentation

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code. It will give you an error if you skip the indentation.

For Example,

Correct	Error
if 5 > 2: print("Five is greater than two!")	if 5 > 2: print("Five is greater than two!")

The number of spaces is up to you as a programmer, the most common use is four, but it has to be at least one. You have to use the same number of spaces in the same block of code, otherwise Python will give you an error.

Correct	Error

if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
  print("Five is greater than two!")
	if 5 > 2:
	 print("Five is greater than two!")
	  print("Five is greater than two!")

Python Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For Example,

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character.

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Quotations in Python

Python accepts single ('), double (") and triple ('" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string. The triple quotes are used to span the string across multiple lines. For example,

```
word = 'word'
print (word)

sentence = "This is a sentence."
print (sentence)

paragraph = """This is a paragraph. It is
 made up of multiple lines and sentences."""
print (paragraph)
```

Comments in Python

A comment is a programmer-readable explanation or annotation in the Python source code. They are added with the purpose of making the source code easier for humans to understand, and are ignored by Python interpreter. Just like most modern languages, Python supports single-line (or end-of-line) and multi-line (block) comments.

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

Code	Output
# First comment print ("Hello, World!") # Second comment	Hello, world!

Python does not really have a syntax for multiline comments. To add a multiline comment, you can insert a # for each line.

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Not quite as intended, you can use a multiline string. Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it.

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block.

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called suites in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite.

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

Python Keywords

Keywords are predefined, reserved words used in Python programming that have special meanings to the compiler. We cannot use a keyword as a variable name, function name, or any other identifier. They are used to define the syntax and structure of the Python language.

All the keywords except **True**, **False** and **None** are in lowercase and they must be written as they are.

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter **A to Z** or **a to z** or an **underscore (_)** followed by zero or more letters, underscores and digits (0 to 9). Python does not allow punctuation characters such as @, $, and % within identifiers.

Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Rules for Naming an Identifier

- Identifiers cannot be a keyword.
- Identifiers are case-sensitive.
- It can have a sequence of letters and digits. However, it must begin with a letter or _. The first letter of an identifier cannot be a digit.
- It's a convention to start an identifier with a letter rather _.
- Whitespaces are not allowed.
- We cannot use special symbols like !, @, #, $, and so on.

Valid Identifiers	Invalid Identifiers
score	@core
return_value	return
highest_score	highest score
name1	1name
convert_to_string	convert to_string

Python Variables

In programming, a variable is a container (storage area) to hold data. Python variables are the reserved memory locations used to store values within a Python Program. This means that when you create a variable you reserve some space in the memory.

Creating Python Variables

Python variables do not need explicit declaration to reserve memory space or you can say to create a variable. A Python variable is created automatically when you assign a value to it.

The equal sign (=) is used to assign values to variables. The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

```
counter = 100          # Creates an integer variable
miles   = 1000.0       # Creates a floating point variable
name    = "AUST CSE"   # Creates a string variable
```

Python is a type-inferred language, so you don't have to explicitly define the variable type. It automatically knows that *AUST CSE* is a string and declares the *name* variable as a string.

Assigning multiple values to multiple variables

```
a, b, c = 5, 3.2, 'Hello'

print(a)  # prints 5
print(b)  # prints 3.2
print(c)  # prints Hello
```

If we want to assign the same value to multiple variables at once, we can do this as:

```
name1 = name2  = 'AUST CSE'

print(name1)  # prints AUST CSE
print(name2)  # prints AUST CSE
```

Variables do not need to be declared with any particular type, and can even change type after they have been set.

```
x = 4       # x is of type int
x = "AUST" # x is now of type str
print(x)
```

If you want to specify the data type of a variable, this can be done with casting.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called unpacking.

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Output Variables

The Python print() function is often used to output variables.

```
x = "Python is awesome"
print(x)
```

In the print() function, you can output multiple variables, separated by a comma.

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

Local Variables

Python Local Variables are defined inside a function. We cannot access variable outside the function.

```
def sum(x,y):
   sum = x + y
   return sum
print(sum(5, 10))
```

Global Variables

Variables that are created outside of a function are known as global variables. Global variables can be used by everyone, both inside of functions and outside.

```
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

```
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function. To create a global variable inside a function, you can use the global keyword.

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

Use the global keyword if you want to change a global variable inside a function.

```
x = "awesome"

def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

Deleting Python Variables

You can delete the reference to a number object by using the del statement.

```
del var
del var_a, var_b
```

If we try to use a deleted variable then Python interpreter will throw an error.

```
counter = 100
print (counter)

del counter
print (counter)
```

Data Types in Python

Variables can store data of different types, and different types can do different things. Data types specify the type of data that can be stored inside a variable. Python has the following data types built-in by default, in these categories:

Text Type: str

Numeric Types: int, float, complex

Sequence Types: list, tuple, range

Mapping Type: dict

Set Types: set, frozenset

Boolean Type: bool

Binary Types: bytes, bytearray, memoryview

None Type: NoneType

You can get the data type of any object by using the **type()** function:

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

In Python, the data type is set when you assign a value to a variable:

x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview
x = None	NoneType

If you want to specify the data type, you can use the following constructor functions:

```
x = str("Hello World")
x = int(20)
x = float(20.5)
```

```
x = complex(1j)
x = list(("apple", "banana", "cherry"))
x = tuple(("apple", "banana", "cherry"))
x = range(6)
x = dict(name="John", age=36)
x = set(("apple", "banana", "cherry"))
x = frozenset(("apple", "banana", "cherry"))
x = bool(5)
x = bytes(5)
x = bytearray(5)
x = memoryview(bytes(5))
```

Python Data Type Conversion

To convert data between different Python data types, you simply use the type name as a function.

```
a = int(1)     # a will be 1
b = int(2.2)   # b will be 2
c = int("3.3")   # c will be 3

a = float(1)     # a will be 1.0
b = float(2.2)   # b will be 2.2
c = float("3.3") # c will be 3.3

a = str(1)     # a will be "1"
b = str(2.2)   # b will be "2.2"
c = str("3.3") # c will be "3.3"
```

In certain situations, Python automatically converts one data type to another. This is known as implicit type conversion.

```
integer_number = 123
float_number = 1.23

new_number = integer_number + float_number

print("Value:",new_number)
print("Data Type:",type(new_number))
```

Python User Input

In Python, we can use the **input()** function to take the input from the user.

Syntax: input(prompt)

Here, **prompt** is the string we wish to display on the screen. It is **optional**.

num = input('Enter a number: ') print('You Entered:', num) print('Data type of num:', type(num))	Enter a number: 10 You Entered: 10 Data type of num: <class 'str'>

Python always read the user input as a string. It is important to note that the entered value 10 is a string, not a number. So, type(num) returns <class 'str'>.

To convert user input into a number we can use **int()** or **float()** functions as:

```
num = int(input('Enter a number: '))
```

Example Codes

1. Add Two Numbers With User Input

```
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')

sum = float(num1) + float(num2)

print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

2. Calculate the Area of a Triangle

```
a = float(input('Enter first side: '))
b = float(input('Enter second side: '))
c = float(input('Enter third side: '))

s = (a + b + c) / 2

area = (s*(s-a)*(s-b)*(s-c)) ** 0.5

print('The area of the triangle {0:.3f} and s={1:.3f}' .format(area,s))
```

Python Operators

Operators are special symbols that perform operations on variables and values. These operations can be arithmetic, logical, comparison, assignment, membership, or identity operations. Understanding operators is crucial for manipulating data effectively and writing concise code.

Types of Python Operators

- Arithmetic Operators
- Comparison Operators
- Assignment Operators
- Logical Operators
- Membership Operators
- Identity Operators
- Bitwise Operators

Python Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

For Example,

Assume, variable a holds 10 and variable b holds 20.

Operator	Name	Example
+	Addition	a + b = 30
-	Subtraction	a – b = -10
*	Multiplication	a * b = 200
/	Division	b / a = 2
%	Modulus	b % a = 0
**	Exponent	a**b =10**20
//	Floor Division	9//2 = 4

```
a = 21
b = 10
c = 0

c = a + b
print ("a: {} b: {} a+b: {}".format(a,b,c))

c = a - b
print ("a: {} b: {} a-b: {}".format(a,b,c) )

c = a * b
print ("a: {} b: {} a*b: {}".format(a,b,c))

c = a / b
print ("a: {} b: {} a/b: {}".format(a,b,c))

c = a % b
print ("a: {} b: {} a%b: {}".format(a,b,c))

a = 2
b = 3
c = a**b
print ("a: {} b: {} a**b: {}".format(a,b,c))

a = 10
b = 5
c = a//b
print ("a: {} b: {} a//b: {}".format(a,b,c))
```

Python Comparison Operators

Comparison operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

Assume, variable a holds 10 and variable b holds 20.

Operator	Name	Example
==	Equal	(a == b) is not true
!=	Not equal	(a != b) is true
>	Greater than	(a > b) is not true

<	Less than	(a < b) is true
>=	Greater than or equal to	(a >= b) is not true
<=	Less than or equal to	(a <= b) is true

```
a = 21
b = 10
if ( a == b ):
   print ("Line 1 - a is equal to b")
else:
   print ("Line 1 - a is not equal to b")

if ( a != b ):
   print ("Line 2 - a is not equal to b")
else:
   print ("Line 2 - a is equal to b")

if ( a < b ):
   print ("Line 3 - a is less than b" )
else:
   print ("Line 3 - a is not less than b")

if ( a > b ):
   print ("Line 4 - a is greater than b")
else:
   print ("Line 4 - a is not greater than b")

a,b=b,a #values of a and b swapped. a becomes 10, b becomes 21

if ( a <= b ):
   print ("Line 5 - a is either less than or equal to  b")
else:
   print ("Line 5 - a is neither less than nor equal to  b")

if ( b >= a ):
   print ("Line 6 - b is either greater than  or equal to b")
else:
   print ("Line 6 - b is neither greater than  nor equal to b")
```

Python Assignment Operators

Assignment operators are used to assign values to variables. For Example,

Operator	Example	Equivalence
=	a = 10	a = 10
+=	a += 30	a = a + 30
-=	a -= 15	a = a - 15
*=	a *= 10	a = a * 10
/=	a /= 5	a = a / 5
%=	a %= 5	a = a % 5
**=	a **= 4	a = a ** 4
//=	a //= 5	a = a // 5
&=	a &= 5	a = a & 5
|=	a |= 5	a = a | 5
^=	a ^= 5	a = a ^ 5
>>=	a >>= 5	a = a >> 5
<<=	a <<= 5	a = a << 5
:=	print(x := 3)	x = 3 print(x)

Python Logical Operators

Python logical operators are used to combine two or more conditions and check the final result.

Assume, variable a holds 10 and variable b holds 20.

Operator	Name	Example
and	AND	a>5 and b<25
or	OR	a>5 or b<25
not	NOT	not(a>5 or b<25)

```
var = 5

print(var > 3 and var < 10)
print(var > 3 or var < 4)
print(not (var > 3 and var < 10))
```

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

Operator	Description	Example
in	returns True if it finds a variable in the specified sequence, false otherwise	a in b
not in	returns True if it does not find a variable in the specified sequence and false otherwise	a not in b

```
a = 10
b = 20
list = [1, 2, 3, 4, 5]

print ("a:", a, "b:", b, "list:", list)

if ( a in list ):
   print ("a is present in the given list")
else:
   print ("a is not present in the given list")

if ( b not in list ):
   print ("b is not present in the given list")
else:
   print ("b is present in the given list")

c=b/a
print ("c:", c, "list:", list)
if ( c in list ):
   print ("c is available in the given list")
else:
    print ("c is not available in the given list")
```

Python Identity Operators

Identity operators compare the memory locations of two objects.

Operator	Description	Example
is	returns True if both variables are the same object and false otherwise	a is b
is not	returns True if both variables are not the same object and false otherwise	a is not b

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4, 5]
c = a

print(a is c)
print(a is b)

print(a is not c)
print(a is not b)
```

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. These operators are used to compare binary numbers.

Operator	Name	Example
&	AND	a & b
|	OR	a | b
^	XOR	a ^ b
~	NOT	~a
<<	Zero fill left shift	a << 3
>>	Signed right shift	a >> 3

```
a = 20
b = 10

print ('a=',a,':',bin(a),'b=',b,':',bin(b))
c = 0

c = a & b;
print ("result of AND is ", c,':',bin(c))

c = a | b;
print ("result of OR is ", c,':',bin(c))

c = a ^ b;
print ("result of EXOR is ", c,':',bin(c))

c = ~a;
print ("result of COMPLEMENT is ", c,':',bin(c))

c = a << 2;
print ("result of LEFT SHIFT is ", c,':',bin(c))

c = a >> 2;
print ("result of RIGHT SHIFT is ", c,':',bin(c))
```

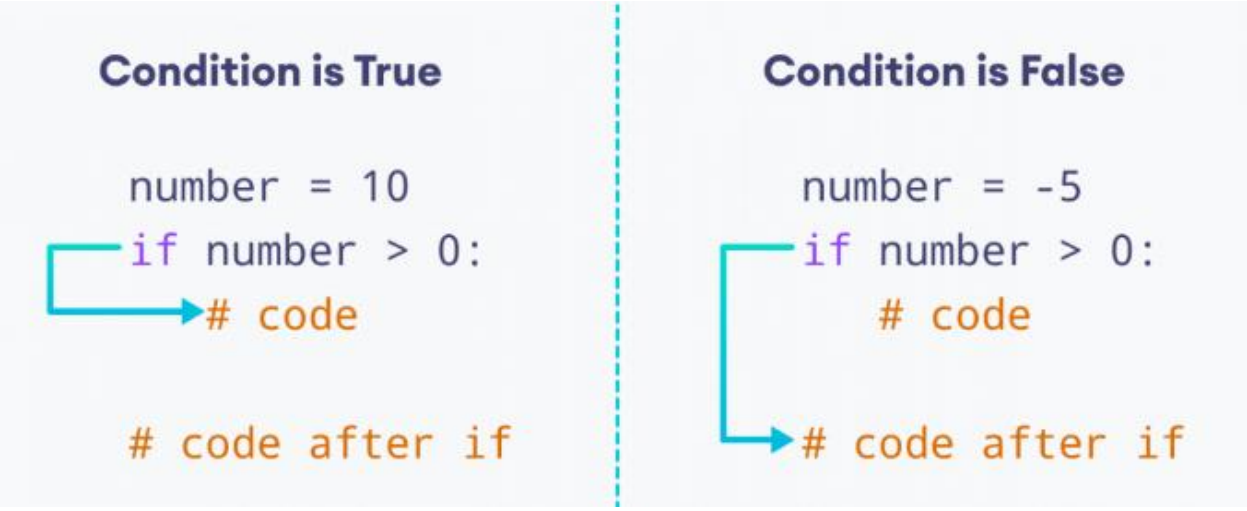
Python if Statement

An if statement executes a block of code only if the specified condition is met.

```
if condition:
    # body of if statement
```

Here, if the condition of the if statement is:

- True - the body of the if statement executes.
- False - the body of the if statement is skipped from execution.




```
number = 10

# check if number is greater than 0
if number > 0:
    print('Number is positive')

print('This statement always executes')
```

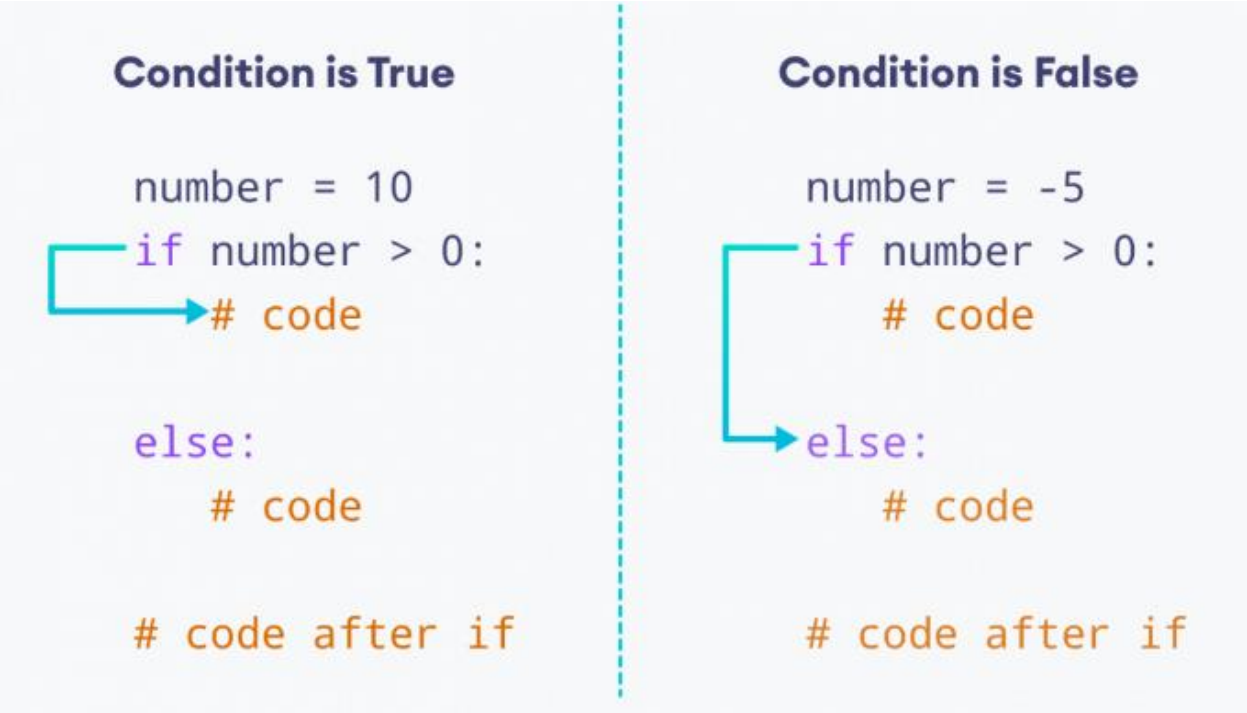
Python if...else Statement

An if statement can have an optional else clause. The else statement executes if the condition in the if statement evaluates to False.

```
if condition:
    # body of if statement

else:
    # body of else statement
```

Here, if the condition inside the if statement evaluates to

- True - the body of if executes, and the body of else is skipped.
- False - the body of else executes, and the body of if is skipped.




```
number = 10

if number > 0:
    print('Positive number')

else:
    print('Negative number')

print('This statement always executes')
```

Python if…elif…else Statement

The if...else statement is used to execute a block of code among two alternatives. However, if we need to make a choice between more than two alternatives, we use the if...elif...else statement.

```
if condition1:
    # code block 1
elif condition2:
    # code block 2
else:
    # code block 3
```

```
number = 0

if number > 0:
    print('Positive number')

elif number <0:
    print('Negative number')

else:
    print('Zero')

print('This statement is always executed')
```

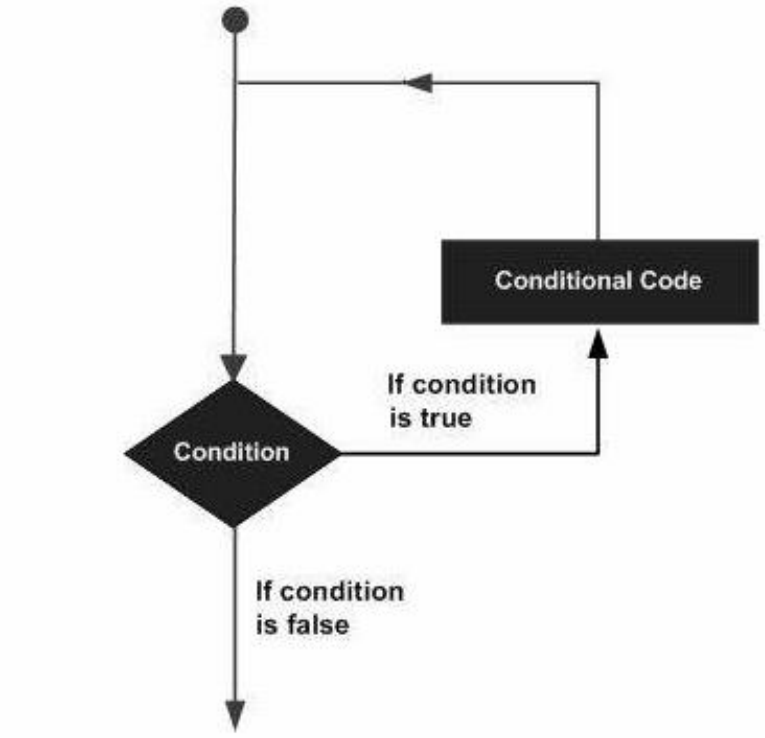
Python Nested if Statements

It is possible to include an if statement inside another if statement. For example,

```
number = 5

# outer if statement
if number >= 0:
    # inner if statement
    if number == 0:
      print('Number is 0')

    # inner else statement
    else:
        print('Number is positive')

# outer else statement
else:
    print('Number is negative')
```

Python Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times. Python loops allow us to execute a statement or group of statements multiple times.



Types of Loops in Python

- while loop
- for loop
- nested loops

The while Loop

With the while loop we can execute a set of statements as long as a condition is true. The while loop requires relevant variables to be ready.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

The else Statement

With the else statement we can run a block of code once when the condition no longer is true.

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string). With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc. The for loop does not require an indexing variable to set beforehand.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

Looping Through a String

```
for x in "banana":
  print(x)
```

The break Statement

With the break statement we can stop the loop before it has looped through all the items.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)
```

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

The range() Function

To loop through a set of code a specified number of times, we can use the range() function. The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):
  print(x)
```

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6).

```
for x in range(2, 6):
  print(x)
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3).

```
for x in range(2, 30, 3):
  print(x)
```

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished.

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

The else block will NOT be executed if the loop is stopped by a break statement.

```
for x in range(6):
  if x == 3:
    break
  print(x)
else:
  print("Finally finished!")
```

The pass Statement

Python pass statement is used when a statement is required syntactically but you do not want any command or code to execute. It is a null operation; nothing happens when it executes.

```
for x in [0, 1, 2]:
  pass
```

Nested Loops

A nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop".

```
# outer loop
for i in range(2):
    # inner loop
    for j in range(2):
        print(f"i = {i}, j = {j}")
```

Infinite while Loop

If the condition of a while loop is always True, the loop runs for infinite times, forming an infinite while loop.

```
age = 32
while age > 18:
   print('You can vote')
```

for loop vs while loop

The for loop is usually used in the sequence when the number of iterations is known.

```
for i in range(4):
   print(i)
```

The while loop is usually used when the number of iterations is unknown.

```
while True:
   user_input = input("Enter password: ")
   if user_input == 'exit':
      print('Status: Entry Rejected')
      break
   print('Status: Entry Allowed')
```

Modifying 'range()' function to generate infinite for loop

```
from itertools import count

for i in count():
   print("Value of i: ", i)
```

Sample Codes

- 1+2+3+…+n

```
n = int(input("Enter value of n: "))
sum = 0
for i in range(1,n+1):
    sum+=i
print(sum)
```

- 3+11+19+…+n

```
n = int(input("Enter value of n: "))
sum = 0
for i in range(3,n+1,8):
    sum+=i
print(sum)
```

❖ Guess the output!

```
for i in range(3):
  for j in range(3):
    if j==1:
      break
    print(i,j)
```

❖ Guess the output!

```
for i in range(3):
  for j in range(3):
    if j==1:
      continue
    print(i,j)
```

❖ Guess the output!

```
def fun(n):
        for i in range(1, n + 1):
                for j in range(n - i):
                        print(" ", end="")
                for k in range(1, 2*i):
                        print("*", end="")
                print('\n')
n = 5
fun(n)
```

Python Functions

A function is a block of code which only runs when it is called and performs a specific task. You can pass data, known as parameters, into a function. A function can return data as a result. A Python function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. Dividing a complex problem into smaller chunks makes our program easy to understand and reuse. A Python function may be invoked from any other function by passing required data (called parameters or arguments). The called function returns its result back to the calling environment.

Types of Python Functions

Python provides the following types of functions:

- Built-in functions
- Functions defined in built-in modules
- User-defined functions

Creating a Function

In Python a function is defined using the **def** keyword.

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis.

```
def my_function():
  print("Hello from a function")

my_function()
```

Arguments

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Function to Add Two Numbers

```
def add_numbers(num1, num2):
    sum = num1 + num2
    print("Sum: ", sum)

add_numbers(5, 4)
```

The return Statement

We return a value from the function using the **return** statement.

```
def find_square(num):
    result = num * num
    return result

square = find_square(3)

print('Square:', square)
```

Multiple Function Program

```
def cse():
    print("CSE")
def eee():
    print("EEE")
def ipe():
    print("IPE")
print("Welcome to AUST")
ipe()
eee()
cse()
```

```
def cse():
    print("CSE")
def eee():
    print("EEE")
    cse()
def ipe():
    print("IPE")
    eee()
print("Welcome to AUST")
ipe()
```

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

```
total = 0;

def sum( arg1, arg2 ):
   total = arg1 + arg2;
   print ("Inside the function local total : ", total)
   return total;

sum( 10, 20 );
print ("Outside the function global total : ", total)
```

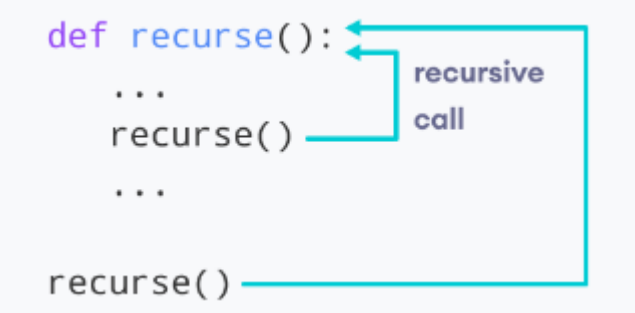
Guess The Output!

```
x = 10
def func1():
    x = 1100
    print("x inside function1: ",x)
    func2(100)
def func2(x):
    print("x inside function2: ",x)
    func3()
def func3():
    global x
    print("x inside function3: ",x)
    x=550
```

```
def func4(x):
    print("x inside function4: ",x)
    x=1500
    return x
print("x outside function: ",x)
func1()
print("x outside function: ",x)
x = 110
print("x outside function: ",x)
x = func4(750)
print("x outside function: ",x)
```

Python Recursion

Recursion is the process of defining something in terms of itself. In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.




```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Find Sum of Natural Numbers Using Recursion

```
def recur_sum(n):
   if n <= 1:
      return n
   else:
      return n + recur_sum(n-1)
num = 5
if num < 0:
   print("Enter a positive number")
else:
   print("The sum is",recur_sum(num))
```

Display Fibonacci Sequence Using Recursion

```
def getFibo(n, n1, n2):
    n3=0
    if n>0 :
        n3 = n1 + n2
        print(n3)
        n1 = n2
        n2 = n3
        getFibo(n-1, n1, n2)
n=7
n1,n2=0,1
print(n1, n2)
getFibo(n-2,n1,n2)
```

Find total digits in an integer number using recursive function

```
count = 0
def digitCount(n):
    if n//10 == 0:
       global count
       count = count + 1
       return
    else:
       n = n//10
       count = count + 1
       digitCount(n)

number = 5231
digitCount(number)
print("No of digit: ",count)
```

Python Collections

There are four collection data types in the Python programming language.

- **List**: ordered and changeable. Allows duplicate members.
- **Tuple**: ordered and unchangeable. Allows duplicate members.
- **Set**: unordered, unchangeable and unindexed. No duplicate members.
- **Dictionary**: ordered and changeable. No duplicate members.

List

Lists are used to store multiple items in a single variable. List is one of the built-in data types in Python used to store collections of data. Lists are created using square brackets.

```
list = ["apple", "banana", "cherry"]
print(list)
```

List items are ordered, changeable, and allow duplicate values. List items are indexed, the first item has index [0], the second item has index [1] etc.

When we say that lists are ordered, it means that the items have a defined order, and that order will not change. If you add new items to a list, the new items will be placed at the end of the list.

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Since lists are indexed, lists can have items with the same value.

```
list = ["apple", "banana", "cherry", "apple", "cherry"]
print(list)
```

To determine how many items a list has, use the len() function.

```
list = ["apple", "banana", "cherry"]
print(len(list))
```

List items are indexed and you can access them by referring to the index number.

```
list = ["apple", "banana", "cherry"]
print(list[1])
```

Negative indexing means start from the end. -1 refers to the last item, -2 refers to the second last item etc.

```
list = ["apple", "banana", "cherry"]
print(list[-1])
```

You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items.

```
list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(list[2:5])
```

By leaving out the start value, the range will start at the first item.

```
list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(list[:4])
```

By leaving out the end value, the range will go on to the end of the list.

```
list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(list[2:])
```

Specify negative indexes if you want to start the search from the end of the list.

```
list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(list[-4:-1])
```

To determine if a specified item is present in a list use the in keyword.

```
list = ["apple", "banana", "cherry"]
if "apple" in list:
  print("Yes, 'apple' is in the fruits list")
```

To add an item to the end of the list, use the append() method.

```
list = ["apple", "banana", "cherry"]
list.append("orange")
print(list)
```

The remove() method removes the specified item.

```
list = ["apple", "banana", "cherry"]
list.remove("banana")
print(list)
```

To change the value of a specific item, refer to the index number.

```
list = ["apple", "banana", "cherry"]
list[1] = "blackcurrant"
print(list)
```

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

Python Lambda

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

```
lambda arguments : expression
```

The expression is executed and the result is returned.

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments.

```
x = lambda a, b : a * b
print(x(5, 6))
```

The power of lambda is better shown when you use them as an anonymous function inside another function.

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Python Arrays

Arrays are used to store multiple values in one single variable. Python does not have built-in support for Arrays, but Python Lists can be used instead. Array in Python can be created by importing an array module. But we are interested to learn about NumPy.

NumPy

NumPy is a Python library which is short for "Numerical Python". NumPy is used for working with arrays. In Python, we have lists that serve the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

```
import numpy
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
```

NumPy is usually imported under the np alias. In Python, alias are an alternate name for referring to the same thing.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

The array object in NumPy is called **ndarray**. To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray.

```
import numpy as np
arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```
import numpy as np
arr = np.array(42)
print(arr)
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array. These are the most common and basic arrays.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

3-D Arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

```
import numpy as np
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

An array can have any number of dimensions. When the array is created, you can define the number of dimensions by using the ndmin argument.

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

Access Array Elements

You can access an array element by referring to its index number. The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element. Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(arr[0, 1])
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

The first number represents the first dimension, which contains two arrays:
[[1, 2, 3], [4, 5, 6]]
and:
[[7, 8, 9], [10, 11, 12]]
Since we selected 0, we are left with the first array:
[[1, 2, 3], [4, 5, 6]]
The second number represents the second dimension, which also contains two arrays:
[1, 2, 3]
and:
[4, 5, 6]
Since we selected 1, we are left with the second array:
[4, 5, 6]
The third number represents the third dimension, which contains three values:

4
5
6
Since we selected 2, we end up with the third value:
6

Negative Indexing

Use negative indexing to access an array from the end.

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(arr[1, -1])
```

Slicing arrays

Slicing in python means taking elements from one given index to another given index. We pass slice instead of index like this: [start:end]. We can also define the step, like this: [start:end:step]. If we don't pass start its considered 0. If we don't pass end its considered length of array in that dimension. If we don't pass step its considered 1.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
print(arr[4:])
print(arr[:4])
print(arr[-3:-1])
print(arr[1:5:2])
print(arr[::2])
```

Slicing 2-D Arrays

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
print(arr[0:2, 2])
print(arr[0:2, 1:4])
```

Shape of an Array

The shape of an array is the number of elements in each dimension. NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
```

Reshaping arrays

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
arr2D = arr.reshape(4, 3)
print(arr2D)
arr3D = arr.reshape(2, 3, 2)
print(arr3D)
```

We can Reshape Into any Shape as long as the elements required for reshaping are equal in both shapes. We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(3, 3)
print(newarr)
```

You are allowed to have one "unknown" dimension. Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method. Pass -1 as the value, and NumPy will calculate this number for you.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(-1, 1, 2)
print(newarr)
```

Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
newarr = arr.reshape(-1)
print(newarr)
```

Iterating Arrays

Iterating means going through elements one by one.

```
import numpy as np
arr = np.array([1, 2, 3])
for x in arr:
  print(x)
```

In a 2-D array it will go through all the rows.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
  print(x)
```

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
  for y in x:
    print(y)
```

Iterating Arrays Using nditer()

In basic for loops, iterating through each scalar of an array we need to use n for loops which can be difficult to write for arrays with very high dimensionality.

```
import numpy as np
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
for x in np.nditer(arr):
  print(x)
```

We can use filtering and followed by iteration.

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for x in np.nditer(arr[:, ::2]):
  print(x)
```

Enumerated Iteration Using ndenumerate()

Enumeration means mentioning sequence number of somethings one by one. Sometimes we require corresponding index of the element while iterating, the ndenumerate() method can be used for those use cases.

```
import numpy as np
arr = np.array([1, 2, 3])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

For 2-D,

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Searching Arrays

You can search an array for a certain value, and return the indexes that get a match. To search an array, use the where() method.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

Find the indexes where the values are even.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 0)
print(x)
```

There is a method called **searchsorted()** which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order. The searchsorted() method is assumed to be used on sorted arrays. Find the indexes where the value 7 should be inserted.

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7)
print(x)
```

The method starts the search from the left and returns the first index where the number 7 is no longer larger than the next value.

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7, side='right')
print(x)
```

To search for more than one value, use an array with the specified values.

```
import numpy as np
arr = np.array([1, 3, 5, 7])
x = np.searchsorted(arr, [2, 4, 8])
print(x)
```

Sorting Arrays

Sorting means putting elements in an ordered sequence.

```
import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

This method returns a copy of the array, leaving the original array unchanged. If you use the sort() method on a 2-D array, both arrays will be sorted.

```
import numpy as np
arr = np.array([[3, 2, 4], [5, 0, 1]])
print(np.sort(arr))
```

❖ Write a python code to find second largest number in a NumPy array.

```
import numpy as np
def find_second_largest(arr):
  largest = second_largest = arr[0]
  for num in arr[1:]:
    if num > largest:
      second_largest = largest
      largest = num
    elif num > second_largest and num != largest:
      second_largest = num
  return second_largest

arr =np.array([10, 5, 8, 15, 2])
second_largest = find_second_largest(arr)
print("The second largest element is:", second_largest)
```

❖ Write a python code to add two 2D matrices using NumPy array.

```
import numpy as np
matrix1 = np.array([[1, 2, 3], [4, 5, 6]])
matrix2 = np.array([[7, 8, 9], [10, 11, 12]])
sum_matrix = matrix1 + matrix2
print("Sum of the matrices:\n", sum_matrix)
```

❖ Write a python code to multiply two 2D matrices using NumPy array.

```
import numpy as np
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
result_matrix = np.dot(matrix1, matrix2)
print("Product of the matrices:\n", result_matrix)
```

Python Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

```
print("Hello")
print('Hello')
print("It's alright")
print("AUST 'CSE' ")
print('AUST "IPE" ')
```

Assigning a string to a variable is done with the variable name followed by an equal sign and the string.

```
a = "Hello"
print(a)
```

You can assign a multiline string to a variable by using three quotes.

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

```
a = "Hello, World!"
print(a[1])
```

Since strings are arrays, we can loop through the characters in a string, with a for loop.

```
for x in "banana":
  print(x)
```

To get the length of a string, use the len() function.

```
a = "Hello, World!"
print(len(a))
```

To check if a certain phrase or character is present in a string, we can use the keyword **in**.

```
txt = "The best things in life are free!"
print("free" in txt)
```

To check if a certain phrase or character is NOT present in a string, we can use the keyword **not in**.

```
txt = "The best things in life are free!"
print("expensive" not in txt)
```

Slicing Strings

You can return a range of characters by using the slice syntax. Specify the start index and the end index, separated by a colon, to return a part of the string.

```
b = "Hello, World!"
print(b[2:5])
```

By leaving out the start index, the range will start at the first character.

```
b = "Hello, World!"
print(b[:5])
```

By leaving out the end index, the range will go to the end.

```
b = "Hello, World!"
print(b[2:])
```

Use negative indexes to start the slice from the end of the string.

```
b = "Hello, World!"
print(b[-5:-2])
```

Modify Strings

The upper() method returns the string in upper case.

```
a = "Hello, World!"
print(a.upper())
```

The lower() method returns the string in lower case.

```
a = "Hello, World!"
print(a.lower())
```

The swapcase() method returns a string where all the upper case letters become lower case and vice versa.

```
txt = "Hello My Name Is PETER"
x = txt.swapcase()
print(x)
```

The zfill() method adds zeros (0) at the beginning of the string, until it reaches the specified length. If the value of the len parameter is less than the length of the string, no filling is done.

```
a = "hello"
b = "welcome to the jungle"
c = "10.000"
```

```
print(a.zfill(10))
print(b.zfill(10))
print(c.zfill(10))
```

The strip() method removes any whitespace from the beginning or the end.

```
a = " Hello, World! "
print(a.strip())
```

The replace() method replaces a string with another string.

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

The split() method returns a list where the text between the specified separator becomes the list items.

```
a = "Hello, World!"
print(a.split(","))
```

To concatenate, or combine, two strings you can use the + operator.

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

To add a space between them, add a " ".

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

We can combine strings and numbers by using f-strings.

```
age = 21
txt = f"My name is John, I am {age}"
print(txt)
```

A placeholder can include a modifier to format the value.

```
price = 59
txt = f"The price is {price:.2f} dollars"
print(txt)
```

A placeholder can contain Python code, like math operations.

```
txt = f"The price is {20 * 59} dollars"
print(txt)
```

The capitalize() method returns a string where the first character is upper case, and the rest is lower case.

```
txt = "python is FUN!"
x = txt.capitalize()
print (x)
```

The count() method returns the number of times a specified value appears in the string.

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple")
print(x)
```

Search in a specified range,

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple", 10, 24)
print(x)
```

The startswith() method returns True if the string starts with the specified value, otherwise False.

```
txt = "Hello, welcome to my world."
x = txt.startswith("Hello")
print(x)
```

The endswith() method returns True if the string ends with the specified value, otherwise False.

```
txt = "Hello, welcome to my world."
x = txt.endswith(".")
print(x)
```

The find() method finds the first occurrence of the specified value. It returns -1 if the value is not found.

```
txt = "Hello, welcome to my world."
x = txt.find("e")
print(x)
```

The rfind() method finds the last occurrence of the specified value. It returns -1 if the value is not found.

```
txt = "Hello, welcome to my world."
x = txt.rfind("e")
print(x)
```

The islower() method returns True if all the characters are in lower case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.

```
a = "Hello world!"
b = "hello 123"
```

```
c = "mynameisPeter"
print(a.islower())
print(b.islower())
print(c.islower())
```

The isupper() method returns True if all the characters are in upper case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.

```
a = "Hello World!"
b = "hello 123"
c = "MY NAME IS PETER"
print(a.isupper())
print(b.isupper())
print(c.isupper())
```

The join() method takes all items in an iterable and joins them into one string. A string must be specified as the separator.

```
myTuple = ("John", "Peter", "Vicky")
x = ", ".join(myTuple)
print(x)
```

The lstrip() method removes any leading characters (space is the default leading character to remove).

```
txt = ",,,,,ssaaww.....banana"
x = txt.lstrip("b.,saw")
print(x)
```

The rstrip() method removes any trailing characters (characters at the end of a string), space is the default trailing character to remove.

```
txt = "banana,,,,,ssqqqww....."
x = txt.rstrip("an,.qsw")
print(x)
```

Escape Character

To insert characters that are illegal in a string, use an escape character. An escape character is a **backslash** \ followed by the character you want to insert. An example of an illegal character is a double quote inside a string that is surrounded by double quotes.

```
txt = "We are the so-called "Vikings" from the north." #Wrong
txt = "We are the so-called \"Vikings\" from the north." #Correct
```

Other escape characters used in Python:

```
txt = 'It\'s alright.'
print(txt)
txt = "This will insert one \\ (backslash)."
```

```
print(txt)
txt = "Hello\nWorld!"
print(txt)
txt = "Hello\tWorld!"
print(txt)
txt = "Hello \bWorld!"
print(txt)
```

Searching Algorithms

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

1. Linear Search
2. Binary Search

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found. It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1. Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

For Example,

10	14	19	26	27	31	33	35	40	47

Find 33.

Pseudocode

```
function linear_search(data, target):
  for i in range(len(data)):
    if data[i] == target:
      return i
  return -1
```

Try to implement Linear Search!

```
def linear_search(data, target):
  for i in range(len(data)):
    if data[i] == target:
      return i
  return -1

my_list = [10, 14, 19, 26, 27, 31, 33, 35, 40, 47]
target_element = 33
```

```
result = linear_search(my_list, target_element)
if result != -1:
  print("Element found at index:", result)
else:
  print("Element not found")
```

Binary Search

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

For Example,

10	14	19	26	27	31	33	35	40	47

Find 33.

Pseudocode

```
function binary_search(data, target):
  low = 0
  high = len(data) - 1
  while low <= high:
    mid = (low + high) // 2
    if data[mid] == target:
      return mid
    elif data[mid] < target:
      low = mid + 1
    else:
      high = mid - 1
  return -1
```

Try to implement Binary Search!

```
def binary_search(data, target):
  low = 0
  high = len(data) - 1
  while low <= high:
    mid = (low + high) // 2
    if data[mid] == target:
      return mid
    elif data[mid] < target:
      low = mid + 1
    else:
      high = mid - 1
  return -1

my_data = [10, 14, 19, 26, 27, 31, 33, 35, 40, 47]
target_element = 33
result = binary_search(my_data, target_element)
if result != -1:
  print("Element found at index:", result)
else:
  print("Element not found")
```

Linear Search vs Binary Search

Feature	Linear Search	Binary Search
Data Structure	Works on any list	Requires sorted list
Search Time	O(n) - Iterates through all elements in worst case	O(log n) - Divides search space in half each iteration
Implementation Complexity	Simpler to implement	More complex due to loop and index adjustments
Use Case	Suitable for small datasets or unsorted data	Ideal for large sorted datasets where efficiency matters

Insertion Sort Algorithm

Require: List A, Size of list n

```
for j = 1 to n - 1
   key = A[j]
   i = j - 1
   while i >= 0 and A[i] > key
      A[i + 1] = A[i]
      i = i - 1
   end while
   A[i + 1] = key
end for
```

Selection Sort Algorithm

Require: List A, Size of list n

```
for i = 0 to n - 2 do
   min = i
   for j = i + 1 to n - 1 do
      if A[j] < A[min] then
         min = j
      end if
   end for
   swap A[i] and A[min]
end for
```

FINAL EXAMINATION

There will be a one-hour written examination. Different types of questions will be included such as MCQ, writing a program, finding outputs, correcting errors etc.