UNIT 2 PPS

by Prof. Pradeep Barde

Python Interpreter

The Python interpreter is a software program that executes Python code. It reads Python source code, interprets it, and carries out the instructions specified in the code. The Python interpreter plays a central role in running Python programs and scripts.
Here are some key aspects of the Python interpreter:

Interactive Mode:

The Python interpreter can run in interactive mode, where we can enter Python statements and expressions directly and see their results immediately. We can start the interactive mode by running the python command in wer terminal:

Example

```
>>> print("Hello, World!")
Hello, World!
>>> 3 + 5
8
>>> exit() # To exit the interactive mode
```

Variable Definition

In programming, a variable is a memory location used to store a value. When you store a value in a variable, the value is actually stored in memory at a physical location.

Creating a Variable

In Python, it is very simple to construct a variable. For this function, the assignment operator "=" is used. The variable identifier or name is the value to the left of the assignment operator. The value assigned to the variable is the value to the right of the operator.

```
Age = 15 # An integer variable

Score = 102.5 # A floating type variable

Pass = True # A boolean Variable
```

Depending upon the value being stored in a variable, Python assigns type to the variable at runtime. For instance when Python interpreter interprets the line “Age = 15”, it checks the type of the value which is integer in this case. Hence, Python understands that Age is an integer type variable.

To check type of a variable, pass the variable name to “type” function as shown below:

```
type(Age)
```

Python Data Types:

A programming application must store a wide range of data. Consider the case of a banking application that requires the storage of customer information. For example, a person's name and phone number; whether he is a defaulter or not; a list of items lent by him/her, and so on. Different data types are needed to store such a wide range of information. Though custom data types can be created in the form of classes, Python comes with six standard data types out of the box. They are as follows:

- Strings
- Numbers
- Booleans
- Lists
- Tuples
- Dictionaries

Strings

Python treats string as sequence of characters. To create strings in Python, you can use single as well as double quotes. Take a look at the following script:

```
first_name = 'mike' # String with single quotation
last_name = " johns" # String with double quotation
full_name = first_name + last_name
# string concatenation using +
print(full_name)
```

Numbers

There are four types of numeric data in python:

```
int (Stores integer e.g 10)
float (Stores floating point numbers e.g 2.5)
long (Stores long integer such as 48646684333)
complex (Complex number such as 7j+4847k)
```

To create a numeric Python variable, simply assign a number to variable.

```
int_num = 10 # integer
float_num = 156.2 #float
long_num = -0.5977485613454646 #long
complex_num = -.785+7J #Complex
print(int_num)
print(float_num)
print(long_num)
print(complex_num)
```

Boolean

Boolean variables are used to store Boolean values. True and False are the two Boolean values in Python. Take a look at the following example:

```
defaulter = True
has_car = False
print(defaulter and has_car)
```

Lists

The List data form in Python is used to store a set of values. In every other programming language, lists are identical to arrays. Python lists, on the other hand, can store values of various types. Opening and closing square brackets are used to build a list. A comma separates each item in the list from the next.

```
cars = ['Honda', 'Toyota', 'Audi', 'Ford', 'Suzuki', 'Mercedez']
print(len(cars))  # finds total items in string
print(cars)
```

Tuples
Tuples are similar to lists, but there are two main distinctions. To begin, instead of using square brackets to create lists, opening and closing braces are used to create tuples. **Second, once formed, a tuple is permanent, which means that its values cannot be changed**.

Eg.

```
cars = ['Honda', 'Toyota', 'Audi', 'Ford', 'Suzuki']   # lists

cars2 = ('Honda', 'Toyota', 'Audi', 'Ford', 'Suzuki')  # Tupple
```

Dictionaries
Dictionaries are collections of data that are stored in the form of key-value pairs. A comma separates each key-value pair from the next. The keys and values are separated by a colon. Indexes and keys can also be used to access dictionary objects. To make dictionaries, place key-value pairs inside the opening and closing parenthesis. Consider the following example.

```
cars = {'Name':'Audi', 'Model': 2008, 'Color':'Black'}
print(cars['Color'])
print(cars.keys())
print(cars.values())
```

Operators:

In programming, operators are literals that are used to perform particular logical, relational, or mathematical operations on operands.

Python operators are classified into the five groups mentioned below:

- Arithmetic
- Logical
- Comparison
- Assignment
- Membership operators

Arithmetic:

+(Addition) , -(Subtraction) ,*(multiplication), /(division), %(modulus operator), **(Exponent operator)

Logical Operators

Logical operators are used to perform logical functions such as AND, OR and NOT on the operands(data).

```
N1 = True
N2 = False
print(N1 and N2)
print(N1 or N2)
print(not(N1 and N2))
```

Comparison Operators

Comparison operators evaluate the values in the operands and return true or false based on the relationship between the operands. Relational operators are another name for comparison operators.
== (Equality)
!=(inequality
> (greater than)
< (smaller than)
>=(greater than equal)
<= (smaller than equal

```
N1=10
N2=5
```

```
print(N1 == N2)
print(N1 != N2)
print(N1 > N2)
print(N1 < N2)
print(N1 >= N2)
print(N1 <= N2)
```

Assignment Operators

To assign values to operands, assignment operators are used.

=	(Assignment)
+=	(Add and assign)
-=	(Subtract and assign)
*=	(multiply and assign)
%=	(modulus and assign)
**=	(exponent and assign)

Eg,

```
N1=10;N2=5
N1 += N2
print(N1)
N1=10;N2=5
N1 -= N2
print(N1)
N1=10;N2=5
N1 *= N2
print(N1)
```

Membership Operators

Membership operators are used to determine whether or not the value stored in the operand remains in a given sequence. In Python, there are two groups of membership operators: **‘in'** and **‘not in.'** If a value is contained in a specific sequence, the in operator returns real.

```
cars = ['Honda', 'Toyota', 'Audi', 'Ford', 'Camery']
print('Honda' in cars)

print('Honda' not in cars)
```

function:

Definition: Functions are the subprograms that perform specific task. Functions are the small modules.

Types of Functions:

1. Library Functions (Built in functions)

2. Functions defined in modules

3. User Defined Functions

1. Library Functions: These functions are already built in the python library.

For example: type( ), len( ), input( ) etc.

2. Functions defined in modules: These functions defined in particular modules. When you want to use these functions in program, you have to import the corresponding module of that function.
To work with the functions of math module, we must import math module in program.

import math

```
sqrt()    eg.  print(math.sqrt(49)
pow()     eg.  print(math.pow(2,3))

floor()   eg.  print(math.floor(81.3)
```

3. User Defined Functions: The functions those are defined by the user are called user defined functions.

USER DEFINED FUNCTIONS:

The **syntax** to define a function is:

```
def function-name ( parameters) :
        #statement(s)
        return(expression/values)    #optional
```

Example :
#Subprogram

```
def ADD(x, y): #Defining a function and x and y are parameters
z=x+y
print("Sum = ", z)
```

#Main program

```
a=float(input("Enter first number: " ))
b=float(input("Enter second number: " ))
ADD(a,b)   # Calling the function by passing actual arguments
```

Calling the function:

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

Syntax:

```
function-name(arguments)
```

The return statement:

The **return** statement is used to exit a function and go back to the place from where it was called.

There are two types of functions according to return statement:

a. Function returning some value

b. Function not returning any value

Syntax:
return expression/value

a. Function returning some value :

Example-1: Function returning one value

```
def my_function(x):
    return 5 * x
```

Example-2 Function returning multiple values:

```
def sum(a,b,c):
        return a+5, b+4, c+7
S=sum(2,3,4) # S will store the returned values as a tuple
print(S)
```

OUTPUT:

```
(7, 7, 11)
```

b. Function not returning any value :

The function that performs some operations but does not return any value, called void function.

```
def message():
        print("Hello")
m=message()
print(m)
```

Expression:

In Python, an expression is a combination of values, variables, operators, and function calls that can be evaluated to produce a result. Expressions can be as simple as a single constant or variable, or they can be complex combinations of multiple elements. Python allows we to create a wide variety of expressions to perform calculations and make decisions in wer code.

Here are some common types of expressions in Python:

 Arithmetic Expressions:
These expressions involve basic arithmetic operations such as addition, subtraction,
multiplication, division, and more.

```
x = 10
y = 5
result = x + y # Addition expression
```

 Comparison Expressions:
These expressions compare values and return Boolean results (True or False) based on the comparison.

```
a = 10
b = 20
is_greater = a > b # Comparison expression (False)
```

 Logical Expressions:
Logical expressions involve Boolean operators like and, or, and not to combine or negate Boolean values.

```
is_sunny = True
is_warm = False
is_good_weather = is_sunny and not is_warm # Logical expression
(True)
```

 Conditional Expressions (Ternary Operator):
This expression provides a compact way to write conditional statements in a single line.

```
age = 18
if age >= 18
status = "Adult"
else:
"Minor" # Ternary conditional expression
```

Precedence of Operators:

In Python, operator precedence determines the order in which operators are evaluated in an expression. When an expression contains multiple operators, some operators are evaluated
before others based on their precedence level. If operators have the same precedence, they are typically evaluated from left to right.

Creating a Module:
We can create a Python module by creating a .py file. For example, create a file named my_module.py with the following content:

```
# my_module.py
    def greet(name):
        return f"Hello, {name}!"
```

Using a Module:
To use the functions or variables defined in a module, we need to import it in wer Python -script using the import statement:

```
import my_module
message = my_module.greet("Alice")
print(message) # Output: "Hello, Alice!"
```

Flow of Execution:
The flow of execution in a Python program refers to the order in which the statements within the program are executed. Understanding the flow of execution is crucial for comprehending how a program works and for debugging issues. Here's an overview of how the flow of execution typically occurs in a Python program:

1. Sequential Execution:
Python programs are executed sequentially, meaning that statements are executed one after the other in the order they appear in the script, from top to bottom. This is the default behavior unless control flow statements (like conditionals and loops) are used to alter the sequence.

```
Example
statement_1
statement_2
statement_3
```

2. Conditional Execution (if, elif, else):

Conditional statements (if, elif, else) allow we to execute different blocks of code based on whether certain conditions are met. The code block associated with the first True condition is executed, and then the program continues after the conditional block.

Example:

```
if condition_1:
    # Code block executed if condition_1 is True
elif condition_2:
   # Code block executed if condition_2 is True
else:
   # Code block executed if none of the conditions are
True
```

3 Looping (for and while loops):

Loops (for and while) allow we to repeat a block of code multiple times. The program executes the loop body until the loop condition becomes False, and then it continues with the next statement after the loop.

Example:

```
for item in iterable:
# Code block executed for each item in the iterable

while condition:
# Code block executed while the condition is True
```