# Data Science Fundamentals with Python

## MCA Semester – 1

### Unit: 1

Contact: 76000 44051 E-Mail: rupareleducation@gmail.com

## What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

## It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

## What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

## Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

## Good to know

- The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

## Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.

- Python 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.

## Python Install

Many PCs and Macs will have python already installed.

To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

C:\Users*Your Name*>python --version

To check if you have python installed on a Linux or Mac, then on linux open the command line or on Mac open the Terminal and type:

python --version

If you find that you do not have Python installed on your computer, then you can download it for free from the following website: https://www.python.org/

## Python Quickstart

Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

The way to run a python file is like this on the command line:

C:\Users*Your Name*>python helloworld.py

Where "helloworld.py" is the name of your python file.

Let's write our first Python file, called helloworld.py, which can be done in any text editor.

***helloworld.py***

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

Simple as that. Save your file. Open your command line, navigate to the directory where you saved your file, and run:

C:\Users*Your Name*>python helloworld.py

The output should read:

Hello, World!

Python Version

To check the Python version of the editor, you can find it by importing the sys module:

```
import sys

print(sys.version)
```

The Python Command Line

To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.

Type the following on the Windows, Mac or Linux command line:

C:\Users*Your Name*>python

Or, if the "python" command did not work, you can try "py":

C:\Users*Your Name*>py

***From there you can write any python, including our hello world example from earlier in the tutorial:***

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
```

Which will write "Hello, World!" in the command line:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
```

Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

```
exit()
```

## Python Components

As a popular high-level programming language, Python is widely used in many applications, from web development and data analysis to artificial intelligence and machine learning. Whether you're a beginner or an experienced developer, it's essential to understand the different components that make up a Python program. In this article, we'll explore the key elements of a Python program (**basic structure of a Python program**) and explain their role in building successful applications.

**1. Expressions:**
An expression is any legal combination of symbols that represents a value. An expression represents something, which Python evaluates and which then produce a value. Some examples of expressions are

These are: if x>5:

**2. Statement:**
A statement is a programming instruction that does something.
Following are some examples of statements:

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

It is not necessary that a statement result in a value; it may not yield a value.
Some statement in below sample code is:

```
a = 15
b = a – 10
print (a + 3)
if b > 5:
```

**3. Comments:**
Comments are the additional readable information, which is read by the programmers but ignored by Python interpreter. In Python, comments begin with symbol # (Pound or hash character) and end with the end of physical line.
In the below code, you can see single comments:

- The physical lines beginning with # are the full line comments. There are three full line comments in the program are:

```
#This program shows a program" s components
#Definition of function see you () follows
    #Main program code follows now
```

**Multi- line Comments**
What if you want to enter a multi- line comment or a block comment? You can enter a multi – line comment in Python code in two ways:

- Type comment as a triple – quoted multi-line string e.g.,

```
"' Multi-line comment are useful for detailed additional information
related to the program in question. It helps clarify certain
important things "'
```

This type of multi-line comment is also known as docstring. You can either use triple – apostrophe (”””) or triple quotes (”””) to write docstrings. The docstring are very useful in documentation.

**4. Functions**

A function is a code that has a name and it can be reused (executed again) by specifying its name in the program, where needed.

```
deffun():
        print(“execution code”)
```

In the above sample program, there is one function namely fun(). The statements indented below its def statement are part of the function. [All statements indented at the same level below deffun() are part of fun().] This function is executed in main code through following statement (Refer to sample program code given above)

```
        fun()          # function – calls statement
```

Calling of a function becomes a statement e.g. , print is a function but when you call print () to print something , then that function call becomes a statement.

**5. Blocks and Indentation**

Sometimes a group of statements is part of another statement or function. Such a group of one or more statements is called block or code – block or suite. For example,

```
        If (condition):
                block
```

Many languages such as C, C++, Java etc., use symbols like curly brackets to show blocks but Python does not use any symbol for it, rather it uses indentation. Consider the following example:

```
        If b>a:
                print(b)
        for i in range(1,10):
                print(i)
```

A group of individual statements which make a single code-block is also called a suite in Python.

**6. Variables Variables**

are used to store data in a Python program. They are like containers that hold values, such as numbers, strings, or objects. Variables can be assigned values using the "=" operator, and their names can be chosen freely as long as they follow some naming conventions.

**7. Data Types**

Python supports several built-in data types, including numbers, strings, lists, tuples, and dictionaries. Each data type has its own characteristics and methods, and they can be combined and manipulated to perform various operations.

**8. Operators**

Operators are symbols or keywords that perform mathematical, logical, or comparison operations on data. Python supports a wide range of operators, from basic arithmetic (+, -, *, /) to advanced ones like bitwise, assignment, and logical operators.

**9. Control Structures**

Control structures are used to control the flow of a Python program. They include conditionals (if/else statements), loops (for/while statements), and exceptions (try/except blocks). Control structures are essential for creating dynamic and interactive applications.

## Installing Jupyter Notebook

Installing Python itself directly via Jupyter Notebook isn't possible since Jupyter Notebook is primarily a tool for interactive computing and does not manage system-level installations. However, you can set up a Python environment and install packages, including Jupyter Notebook itself, using Python code and terminal commands. Here's how you can approach it:

**1. Install Python**

First, you need to install Python on your system. You can download and install Python from the official Python website. Choose the appropriate installer for your operating system and follow the installation instructions.

**2. Install Jupyter Notebook**

Once Python is installed, you can use pip (Python's package installer) to install Jupyter Notebook. You typically do this from your terminal or command prompt, not from within Jupyter Notebook.

Open your terminal or command prompt and run:

pip install notebook

**3. Start Jupyter Notebook**

After installing Jupyter Notebook, you can start it by running the following command in your terminal or command prompt:

jupyter notebook

This command will open Jupyter Notebook in your default web browser.

**4. Use Jupyter Notebook to Install Additional Packages**

Once Jupyter Notebook is running, you can install additional Python packages from within a notebook using magic commands or by running shell commands. Here's how:

### *Using ! to Run Shell Commands*

In a Jupyter Notebook cell, you can run shell commands by prefixing them with !. For example, to install a package such as numpy, you can run:

!pip install numpy

*Using %pip Magic Command*

For IPython environments, including Jupyter Notebook, %pip is a more integrated way to install packages:

%pip install numpy

**5. Verify Installation**

You can verify that the package has been installed and is available for use in your notebook by importing it:

importnumpyas np

print(np.__version__)

This will print the version of numpy that has been installed, confirming that the installation was successful.

**Summary**

1. **Install Python:** Download and install Python from the official website.
2. **Install Jupyter Notebook:** Run pip install notebook in your terminal.
3. **Start Jupyter Notebook:** Execute jupyter notebook in your terminal.
4. **Install Packages in Jupyter Notebook:**Use !pip install <package> or %pip install <package> in notebook cells.

These steps should help you get Python and Jupyter Notebook set up and allow you to manage additional packages within Jupyter Notebook itself.

## Reading input from the console

In Python, reading input from the console is commonly done using the `input()` function. This function allows you to capture user input as a string. Here's a basic example of how you can use it:

```
X=input("Enter value :")

Print(x)
```

**Key Points:**

- **`input(prompt)`**: Displays a prompt (optional) to the user and waits for the user to type something and press Enter. The entered value is returned as a string.
- **Conversion**: If you need to work with numerical input, you'll often need to convert the string to the desired type (e.g., `int`, `float`).

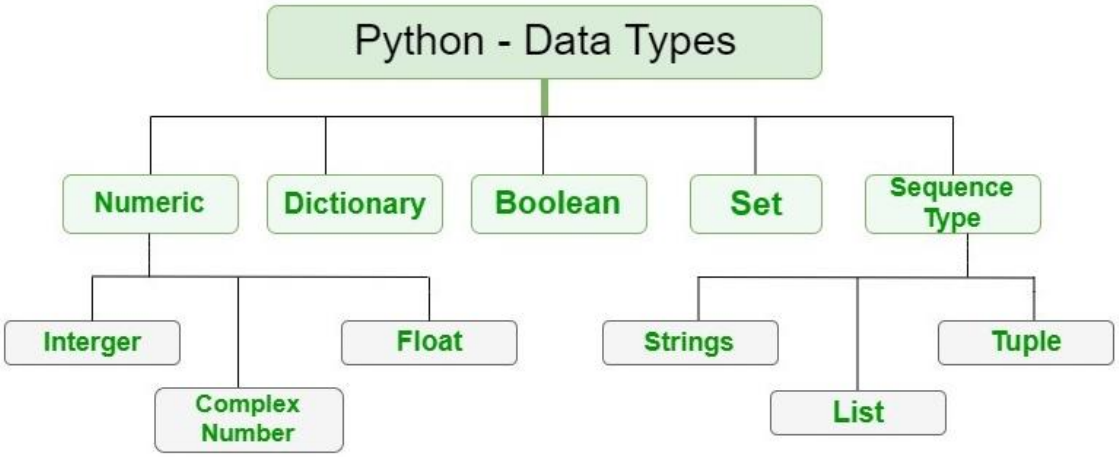
## Built-in Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

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 |




Python has a set of built-in functions.

## Different Forms of Assignment Statements in Python

We use **Python assignment statements** to assign objects to names. The target of an assignment statement is written on the left side of the equal sign (=), and the object on the right can be an arbitrary expression that computes an object.

There are some important properties of assignment in Python :-

- Assignment creates object references instead of copying the objects.
- Python creates a variable name the first time when they are assigned a value.
- Names must be assigned before being referenced.
- There are some operations that perform assignments implicitly.

## Assignment statement forms :-

**1. Basic form:**

This form is the most common form.

```
student = 'IMCA'

print(student)
```

**OUTPUT**
**IMCA**

**2. Tuple assignment:**

```
# equivalent to: (x, y) = (50, 100)

x, y = 50, 100


print('x = ', x)

print('y = ', y)
```

**OUTPUT**
**x = 50**
**y = 100**

When we code a tuple on the left side of the =, Python pairs objects on the right side with targets on the left by position and assigns them from left to right. Therefore, the values of x and y are 50 and 100 respectively.

**3. List assignment:**

This works in the same way as the tuple assignment.

```
[x, y] = [2, 4]

 print('x = ', x)

print('y = ', y)
```

**OUTPUT**
**x = 2**
**y = 4**

**4. Sequence assignment:**

In recent version of Python, tuple and list assignment have been generalized into instances of what we now call sequence assignment – any sequence of names can be assigned to any sequence of values, and Python assigns the items one at a time by position.

```
a, b, c = 'HEY'

print('a = ', a)

print('b = ', b)


print('c = ', c)
```

**OUTPUT**

**a = H**

**b = E**

**c = Y**

**5. Multiple- target assignment:**

```
x = y = 75

print(x, y)
```

In this form, Python assigns a reference to the same object (the object which is rightmost) to all the target on the left.

**OUTPUT**

**75 75**

**6. Augmented assignment :**

The augmented assignment is a shorthand assignment that combines an expression and an assignment.

```
x = 2

# equivalent to: x = x + 1

x += 1

print(x)
```

**OUTPUT**
**3**

There are several other augmented assignment forms:

**-=, **=, &=, etc.**

## What are constants in Python?

Variables that hold a value and cannot be changed are called constants.

- Constants are rarely used in Python – this helps to hold a value for the whole program.
- Constants are usually declared and assigned for different modules or assignments.

## Numeric Data Types in Python

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number. These values are defined as Python int, Python float, and Python complex classes in Python.

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.
- **Float** – This value is represented by the float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- **Complex Numbers** – A complex number is represented by a complex class. It is specified as *(real part) + (imaginary part)j*. For example – 2+3j

**Note** – type() function is used to determine the type of Python data type.

**Example:** This code demonstrates how to determine the data type of variables in Python using the **type() function**. It prints the data types of three variables**: a (integer)**, **b (float)**, and **c (complex)**. The output shows the respective data type Python for each variable.

```
a = 5
print("Type of a: ", type(a))
b = 5.0
print("\nType of b: ", type(b))
c = 2 + 4j
print("\nType of c: ", type(c))
```

**Output:**

```
Type of a:  <class 'int'>
Type of b:  <class 'float'>
Type of c:  <class 'complex'>
```

The '**&**' symbol is a bitwise AND operator in Python, it is also known as a **bitwise AND operator**. It operates on the bitwise representation of integers.

**Example:**

```
num1 = 14
num2= 10
print (num1 & num2)
```

**Output:**

**10**

**14**

in binary is 1110 and 10 in binary is 1010, bitwise AND on these two will give us 1010 which is 10 in integers. This is how to use the & operator in Python.

Difference between 'AND' Operator and '&' Symbol

From the above examples, you can see the clear difference between AND and& operators in Python. Let's use these operators together to see the difference between them:

**Example:**

```
# Python program to demonstrate
# the difference between and, &
# operator

a = 14
b = 4

print(b and a)  # print_stat1
print(b & a)  # print_stat2
```

**Output**

**14**

**4**

This is because '**and**' tests whether both expressions are logically True while '&' performs bitwise AND operation on the result of both statements. In print statement 1, the compiler checks if the first statement is True. If the first statement is False, it does not check the second statement and returns False immediately. This is known as **"lazy evaluation"**. If the first statement is True then the second condition is checked and according to the rules of AND operation, True is the result only if both the statements are True. In the case of the above example, the compiler checks the 1st statement which is True as the value of b is 4, then the compiler moves towards the second statement which is also True because the value of a is 14. Hence, the output is also 14.

In print statement 2, the compiler is doing bitwise & operation of the results of statements. Here,

the statement is getting evaluated as follows:
The value of 4 in binary is 0000 0100 and the value of 14 in binary is 0000 1110. On performing bitwise and we get –
00000100 & 00001110 = 00000100

Hence, the output is 4. To elaborate on this, we can take another example.
**Example:**

```
# Python program to demonstrate
# the difference between and, &
# operator

a, b = 9, 10
print(a & b)  # line 1
print(a and b)  # line 2
```

**Output**
**8**

**10**

The first line is performing bitwise AND on a and b and the second line is evaluating the statement inside print and printing answer. In line 1, a = 1001, b = 1010, Performing & on a and b, gives us 1000 which is the binary value of decimal value 8.
In line 2, the expression 'a and b' first evaluates a; if a is False (or Zero), its value is returned immediately because of the "lazy evaluation" explained above, else, b is evaluated. If b is also non-zero then the resulting value is returned. The value of b is returned because it is the last value where checking ends for the truthfulness of the statement. Hence the use of boolean and 'and' is recommended in a loop.
This is the main difference between AND and& operator in Python. Both operators appear to be the same, but they have very different functionalities in Python Programming language. Practice with each operator to completely grasp their working in Python.

## What are Methods in Python?

Python methods has various uses:

- Methods in Python are used to define the behaviour of the Python objects.
- Methods are used to improve the readability and maintainability of code.
- They help in breaking down complex tasks into smaller, more manageable tasks

Now, let's understand types of Python methods.

## Types of methods in Python

- Instance Methods
- Class Methods
- Static Methods

***Comparison between Python Methods***

## Types of Methods in Python

In Python, there are three types of methods: instance, class, and static.

1. Instance methods are associated with an instance of a class and can modify the data stored within the instance.
2. Class methods are associated with the class rather than an instance and can access and modify class-level data.
3. Static methods are similar to functions outside of the class and cannot access any instance or class data.

## Instance Methods

**Characteristics:**

- Instance methods are defined within the class definition and are called on an instance of the class using the dot notation.
- "Self" is often the first parameter of an instance method.
- They can access and modify the attributes of the instance on which they are called.
- They can also access and call other instance and class methods of the same class.

**Example:**

Here are some examples of instance methods:

**Example 1: Example to find the area of a circle using instance methods.**

```
class Circle:
def __init__(self, radius):
self.radius = radius
def area(self):
    return 3.14 * self.radius ** 2
circle1 = Circle(5)
print(circle1.area())
```

**Output:**

**78.5**

**Example 2: Creating a Car class to print the specifications of the particular car using instance methods.**

```
 # Define the Car class
class Car:
   # Initialize the Car object with the make, model, and year
def __init__(self, make, model, year):
self.make = make
self.model = model
self.year = year
      # Set the odometer reading to 0
self.odometer = 0
   # Method to simulate driving the car and add to the odometer reading
defdrive(self, miles):
self.odometer += miles
   # Method to return the car's info including make, model, year, and
odometer reading
defget_info(self):
      return f"{self.year} {self.make} {self.model} with {self.odometer} miles"
# Create an instance of the Car class with a make of "Toyota", model of
"Corolla", and year of 2022
my_car = Car("Toyota", "Corolla", 2022)
# Simulate driving the car for 50 miles
my_car.drive(50)

# Print the car's info
print(my_car.get_info())
```

**Output:**
**2022 Toyota Corolla with 50 miles**

**Advantages**

- Instance methods allow for the encapsulation of behaviour specific to an instance of a class, improving code organisation and making it easier to manage complex code.
- Instance methods are inherited by subclasses, which makes it easy to reuse code and extend functionality.
- Instance methods are Python's most commonly used method, and their purpose is straightforward to understand.

**Disadvantages**

- Each class instance contains a copy of all instance variables, which can be inefficient if the class has many instances.

- Instance methods are tightly coupled to the data stored within an instance, making it difficult to change the behaviour of a class without affecting other classes that use that class.
- Instance methods often rely on an instance's state, making them difficult to test in isolation.

## Class Methods

Class methods operate on the class itself rather than on an instance. Class methods are defined within the class definition and are called on the class rather than on an instance.

**Characteristics**

- Python class methods are associated with the class and are defined using the @classmethod decorator.
- They are conventionally named "cls" and have access to class-level data.
- Class methods are frequently used to create alternative constructors that allow for easier creation of class instances with attributes set differently.
- Class methods have access to class-level variables but not instance-level variables. However, because they are associated with the class, they are often used to modify class-level data.

**Examples of class methods**

**Example 1: simple example illustrates the use of class methods**

```
import datetime
class Person:
   # Define a class variable all_people as an empty list
all_people = []

   # Define an instance method called __init__ that takes two arguments: name and age
def __init__(self, name, age):
      self.name = name
self.age = age
      # Add the instance to the all_people list
Person.all_people.append(self)
   # Define a class method called from_birth_year that takes two arguments: cls and name and
birth_year
   @classmethod
deffrom_birth_year(cls, name, birth_year):
      # Calculate the age based on the current year and birth year
  age = datetime.date.today().year - birth_year
      return cls(name, age)
# Create two instances of the Person class
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
```

```
# Create a new instance of the Person class using the class method from_birth_year
person3 = Person.from_birth_year("Charlie", 1990)
# Print the names and ages of all people
for person in Person.all_people:
    print(person.name, person.age)
```

**Output:**
**Alice 30**
**Bob 25**
**Charlie 33**

**Example 2: Employee class using class methods**

```
 class Employee:
raise_amount = 1.05  # class variable

def __init__(self, first, last, salary):
self.first = first
self.last = last
self.salary = salary

    @classmethod
defset_raise_amount(cls, amount):
cls.raise_amount = amount  # class method to set raise amount

emp1 = Employee("John", "Doe", 50000)  # creating objects
emp2 = Employee("Jane", "Smith", 60000)

print(Employee.raise_amount)
Employee.set_raise_amount(1.10)  # calling class method to set raise amount
print(Employee.raise_amount)
```

**Output:**
**1.05**
**1.10**

**Advantages**

- The advantages of class methods include that they can access and modify class-level data, which makes them useful for creating alternative constructors and performing operations that involve the class as a whole.
- They can be invoked on the class itself rather than an instance of the class, which means they can be called even if no class instances have been created.
- They can also improve code organization by grouping related functionality in the class definition.

**Disadvantages**

- Disadvantages of class methods include that they cannot access instance-level data, which may limit their usefulness in certain situations.
- They also require a clear understanding of class-level and instance-level data, which can make them more complex to work with.
- Overuse of class methods can also lead to tightly coupled code, which may make the code harder to maintain and test.

## Static Methods

Static methods do not take the self parameter and cannot access the instance, class variables, or methods.

**Characteristics**

- Static methods are independent of instances of the class and the class itself. They don't take any implicit first parameter like self or cls.
- Static methods can't access class-level data or instance-level data. They work like normal functions outside of the class and have no access to data except the data passed as arguments.
- To define a static method, you must use the @staticmethod decorator before the method definition.
- Static methods are often utilised for utility functions that don't depend on any data of the class or instance. They can be used to perform calculations or operations related to the class but not dependent on it.

**Examples:**

**Example 1: Static methods to add, multiply and find factorial of numbers.**

```
class MathOperations:
    # Static method that adds two numbers
    @staticmethod
defadd_numbers(num1, num2):
        return num1 + num2
    # Static method that multiplies two numbers
    @staticmethod
```

```
defmultiply_numbers(num1, num2):
        return num1 * num2
    # Static method that computes the factorial of a number
    @staticmethod
def factorial(num):
        if num == 0:
            return 1
  else:
            return num * MathOperations.factorial(num - 1)

# Example usage
print(MathOperations.add_numbers(2, 3))
print(MathOperations.multiply_numbers(4, 5))
print(MathOperations.factorial(4))
```

**Output:**
**5**
**20**
**24**

In this example, we have a MathOperations class with three static methods: add_numbers(), multiply_numbers(), and factorial().

- The add_numbers() method takes two numbers as parameters and returns their sum.
- The multiply_numbers() method takes two numbers as parameters and returns their product.
- The n factorial() method takes a single number as a parameter and computes its factorial.

**Example 2: Static methods to add two numbers**

```
 class Math:
# Define a static method named "add_numbers" that takes two parameters "x" and "y"
    @staticmethod
defadd_numbers(x, y):
        # Returns the sum of the two numbers
        return x + y
#Create an instance of the Math class and call the "add_numbers" method, passing in two
arguments
print(Math.add_numbers(2, 3))
```

**Output:**
**5**

**Note:** All three methods are defined using the @staticmethod decorator, indicating that they are static.

**Advantages**

- Static methods can execute operations independent of a specific instance or the class.
- They can be called without creating an instance of the class, making them more memory-efficient than instance methods in cases where only a specific functionality is required.
- They can be easily tested and mocked since they do not rely on the state of a specific instance or the class itself.

**Disadvantages**

- As static methods cannot access the instance or class, they cannot modify any instance or class-level data.
- They are less flexible than instance and class methods since they cannot be overridden in subclasses.
- They may not provide a clear separation between the functionality that should be part of the class and the functionality that standalone functions should provide.

**Comparison between Python Methods**

- Instance methods are bound to the instance of a class, and they can access and modify instance-level data. They are the most commonly used method type and are invoked on the class instance using dot notation.
- On the other hand, class methods are bound to the class itself, and they can access and modify class-level data. They are defined using the @classmethod decorator and are invoked on the class rather than the instance of the class.
- Static methods are not bound to the instance or the class and cannot access instance or class-level data. They are defined using the @staticmethod decorator and are used when the method doesn't depend on the instance or class-level data.
- All three method types have access to class-level data and are defined using the same syntax. However, instance and class methods can access and modify instance-level and class-level data, respectively, while static methods do not.
- In terms of invocation, instance methods are invoked on the class instance, class methods are invoked on the class itself, and static methods are invoked on the class or instance of the class, but they do not have access to either.

## Python String Methods

Python has a set of built-in methods that you can use on strings.

**Note:** All string methods returns new values. They do not change the original string.

| Method | Description |
|---|---|
| capitalize() | Converts the first character to upper case |
| casefold() | Converts string into lower case |
| center() | Returns a centered string |
| count() | Returns the number of times a specified value occurs in a string |
| encode() | Returns an encoded version of the string |
| endswith() | Returns true if the string ends with the specified value |
| expandtabs() | Sets the tab size of the string |
| find() | Searches the string for a specified value and returns the position of where it was found |
| format() | Formats specified values in a string |
| format_map() | Formats specified values in a string |
| index() | Searches the string for a specified value and returns the position of where it was found |
| isalnum() | Returns True if all characters in the string are alphanumeric |
| isalpha() | Returns True if all characters in the string are in the alphabet |
| isascii() | Returns True if all characters in the string are ascii characters |
| isdecimal() | Returns True if all characters in the string are decimals |
| isdigit() | Returns True if all characters in the string are digits |
| isidentifier() | Returns True if the string is an identifier |
| islower() | Returns True if all characters in the string are lower case |
| isnumeric() | Returns True if all characters in the string are numeric |
| isprintable() | Returns True if all characters in the string are printable |
| isspace() | Returns True if all characters in the string are whitespaces |
| istitle() | Returns True if the string follows the rules of a title |
| isupper() | Returns True if all characters in the string are upper case |
| join() | Converts the elements of an iterable into a string |
| ljust() | Returns a left justified version of the string |
| lower() | Converts a string into lower case |
| lstrip() | Returns a left trim version of the string |
| maketrans() | Returns a translation table to be used in translations |
| partition() | Returns a tuple where the string is parted into three parts |
| replace() | Returns a string where a specified value is replaced with a specified value |
| rfind() | Searches the string for a specified value and returns the last position of where it was found |
| rindex() | Searches the string for a specified value and returns the last position of where it was found |
| rjust() | Returns a right justified version of the string |
| rpartition() | Returns a tuple where the string is parted into three parts |
| rsplit() | Splits the string at the specified separator, and returns a list |

| | |
|---|---|
| rstrip() | Returns a right trim version of the string |
| split() | Splits the string at the specified separator, and returns a list |
| splitlines() | Splits the string at line breaks and returns a list |
| startswith() | Returns true if the string starts with the specified value |
| strip() | Returns a trimmed version of the string |
| swapcase() | Swaps cases, lower case becomes upper case and vice versa |
| title() | Converts the first character of each word to upper case |
| translate() | Returns a translated string |
| upper() | Converts a string into upper case |
| zfill() | Fills the string with a specified number of 0 values at the beginning |

## Python | Output Formatting

In Python, there are several ways to present the output of a program. Data can be printed in a human-readable form, or written to a file for future use, or even in some other specified form. Users often want more control over the formatting of output than simply printing space-separated values.

Output Formatting in Python

There are several ways to format output using String Method in Python.

- Using String Modulo Operator(%)
- Using Format Method
- Using The String Method
- Python's Format Conversion Rule

### Formatting Output using String moduloOperator (%)

The Modulo % operator can also be used for string formatting. It interprets the left argument much like a printf()-style format as in C language strings to be applied to the right argument. In Python, there is no printf() function but the functionality of the ancient printf is contained in Python. To this purpose, the modulo operator % is overloaded by the string class to perform string formatting. Therefore, it is often called a string modulo (or sometimes even called modulus) operator. The string modulo operator ( % ) is still available in Python(3.x) and is widely used. But nowadays the old style of formatting is removed from the language.

```
# Python program showing how to use string modulo operator(%)

print("IMCA : %2d, Portal : %5.2f" % (1, 05.333))

print("Total students : %3d, Boys : %2d" % (240, 120))   # print integer value

print("%7.3o" % (25))   # print octal value

print("%10.3E" % (356.08977))   # print exponential value
```

**Output**

**IMCA :  1, Portal : 5.33**

**Total students : 240, Boys : 120**

**031**
**3.561E+02**

## Output Formatting using Modulo Operator

There are two of those in our example: “%2d” and “%5.2f”. The general syntax for a format placeholder is:
%[flags][width][.precision]type

Let’s take a look at the placeholders in our example.

- The first placeholder ‘%2d’ is used for the first component of our tuple, i.e. the integer 1. It will be printed with 2 characters, and as 1 consists of only one digit, the output is padded with 1 leading blank.
- The second placeholder ‘%5.2f’ is for a float number. Like other placeholders, it’s introduced with the % character. It specifies the total number of digits the string should contain, including the decimal point and all the digits, both before and after the decimal point.
- Our float number 05.333 is formatted with 5 characters and a precision of 2, denoted by the number following the ‘.’ in the placeholder. The last character ‘f’ indicates that the placeholder represents a float value.

## Formatting Output using The Format Method

The format() method was added in Python(2.6). The format method of strings requires more manual effort. Users use {} to mark where a variable will be substituted and can provide detailed formatting directives, but the user also needs to provide the information to be formatted. This method lets us concatenate elements within an output through positional formatting. For Example –

**Example 1:** The code explain various Python string formatting techniques.The values are either explicitly supplied or referred to by the order in which they appear in the format() procedure.f-Strings enable the use of curly braces and the f prefix to embed expressions inside string literals. The f-Strings’ expressions are assessed and their appropriate values are substituted for them.

```
print('I learns {} from "{}!"'.format('Noble, 'Ruparel’))

# using format() method and referring a position of the object
print('{0} and {1}'.format('Ruparel', 'Junagadh'))

print('{1} and {0}'.format('IMCA', 'BCA'))

print(f"I am learning  {'Python'} for \"{'IT'}!\"")

# using format() method and referring a position of the object
print(f"{'IMCA'} and {'BCA'}")
```

**Output**

**I learnsNoble for "Ruparel!"**
**Ruparel and Junagadh**
**IMCA and BCA**

**I amlearingPython for "IT!"**
**IMCA and BCA**

The brackets and characters within them (called **format fields**) are replaced with the objects passed into the format() method. A number in the brackets can be used to refer to the position of the object passed into the format() method.

**Example 2:**With the help of positional parameters and a named argument ('other') in the first line, the values 'Ruparel', 'For', and 'City Campus' are added to the string template.'Girls:12, Portal: 0.55' is printed, with the first value appearing as a 2-digit integer and the second number having 2 decimal places and an 8-bit width. The format() method's named arguments, denoted by specific labels ('a' and 'p') for the numbers '453' and '59.058',

```
# combining positional and keyword arguments
print('Number one portal is {0}, {1}, and {other}.'
    .format('Ruparel', 'For', other ='City Campus'))

# using format() method with number
print("Girls :{0:2d}, Portal :{1:8.2f}".
     format(12, 00.546))

# Changing positional argument
print("Second argument: {1:3d}, first one: {0:7.2f}".
     format(47.42, 11))

print("IMCA: {a:5d},  Boys: {p:8.2f}".
    format(a = 453, p = 59.058))
```

**Output**

**Number one portal is Ruparel, For, and City Campus.**
**Geeks :12, Portal :  0.55**
**Second argument:  11, first one:  47.42**
**IMCA:  453, Boys:  59.06**

## Output Formatting using Format method

**Formatting Output using The String Method**

This output is formatted by using **string method i.e.** slicing and concatenation operations. The string type has some methods that help in formatting output in a fancier way. Some methods

which help in formatting an output are str.ljust(), str.rjust(), and str.centre()

```
cstr = "I am learning Python"
 # Printing the center aligned string with fillchr
print("Center aligned string with fillchr: ")
print(cstr.center(40, '#'))
# Printing the left aligned string with "-" padding
print("The left aligned string is : ")
print(cstr.ljust(40, '-'))

# Printing the right aligned string with "-" padding
print("The right aligned string is : ")
print(cstr.rjust(40, '-'))
```

**Output**

```
Center aligned string with fillchr:
########## I am learning Python##########
The left aligned string is :
I am learning Python----------
The right aligned string is :
---------- I am learning Python
```

Python's Format Conversion Rule

This table lists the standard format conversion guidelines used by Python's format() function.

| Conversion | Meaning |
|---|---|
| D | Decimal integer |
| b | Binary format |
| o | octal format |
| u | Obsolete and equivalent to 'd' |
| x or X | Hexadecimal format |
| e or E | Exponential notation |
| f or F | Floating-point decimal |
| g or G | General format |
| c | Single Character |
| r | String format(using repr()) |
| s | String Format(using str())) |
| % | Percentage |