[Complete](https://www.youtube.com/playlist?list=PLdOKnrf8EcP384Ilxra4UlK9BDJGwawg9)

[Python for](https://www.youtube.com/playlist?list=PLdOKnrf8EcP384Ilxra4UlK9BDJGwawg9) [Beginners](https://www.youtube.com/playlist?list=PLdOKnrf8EcP384Ilxra4UlK9BDJGwawg9)

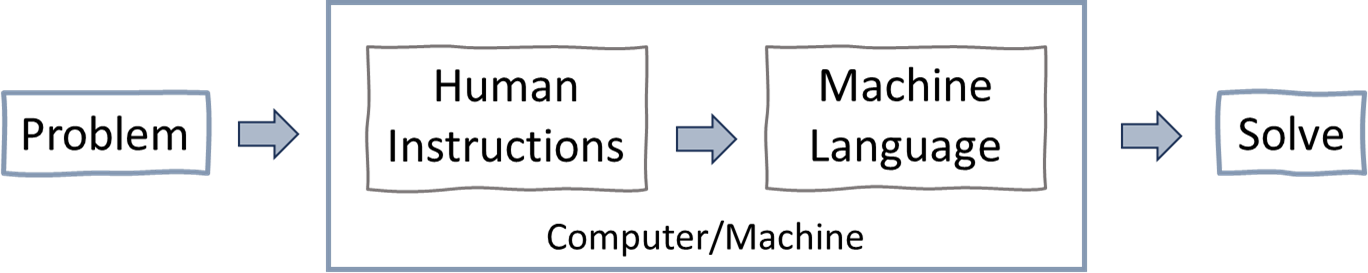
**Notes**

# PYTHON TUTORIAL FOR BEGINNERS

### Chapter - 01 Introduction to Python

* What is programming
* What is Python
* Popular programming languages
* Why Python
* Career with Python

### What is Programming Language?

**[](https://www.youtube.com/%40RishabhMishraOfficial/)**

Programming is the process of creating sets of instructions that tell a computer how to perform specific tasks. These instructions, known as code, are written in programming languages that computer understand and executes to carry out various operations, such as solving problems, analysing data, or controlling device.

Popular programming languages: Python, C, C++, Java, Go, C#, etc.

### What is Python?

Python is a high-level programming language known for its simplicity and readability.

Just like we use Hindi language to communicate and express ourselves,

Python is a language for computers to understand our instructions & perform tasks.

**Note:** Python was created by Guido van Rossum in 1991.

### Popular programming languages

As per statista survey, Python is the most popular programming language.

### Why Python?

Python is one of the easiest programming languages to learn and known for its versatility and user-friendly syntax, is a top choice among programmers.

Also, python is an open source (free) programming language and have extensive libraries to make programming easy. Python has massive use across different industries with excellent job opportunities.

### Python is Dynamically Typed Example

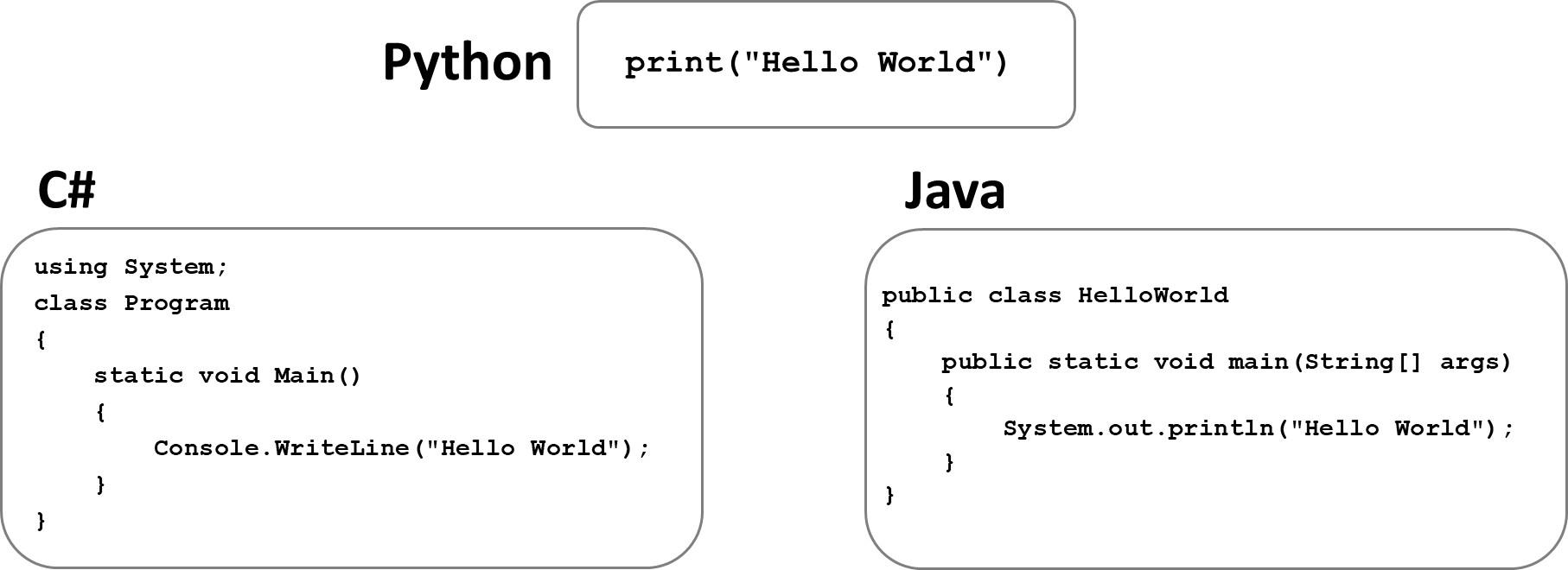
In Python there is no declaration of a variable, just an assignment statement. x = 8 *# here x is a integer*

x = "Python by Rishabh Mishra" *# here x is a string*

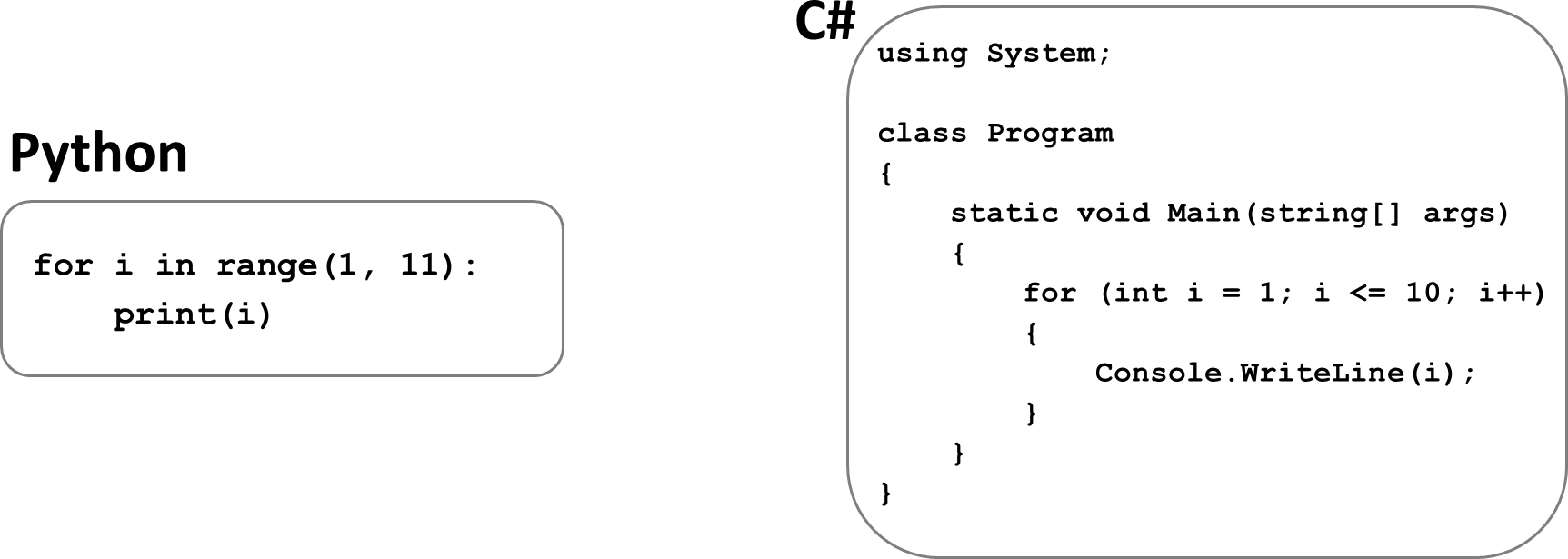
print(type(x)) *# you can check the type of x*

### Python - Easy to Read & Write

Ques1: Write a program to print “Hello World”

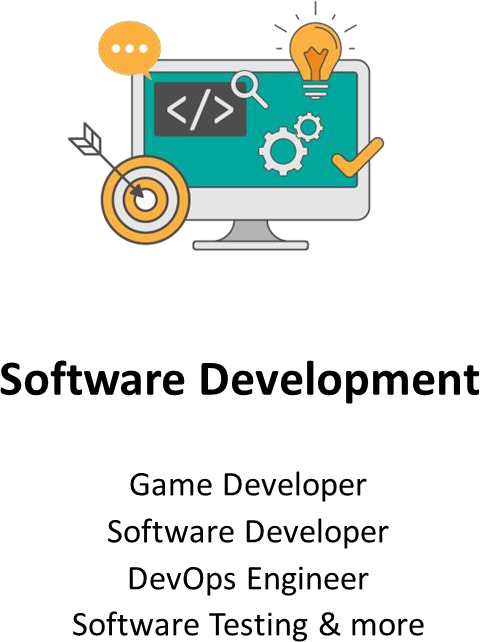
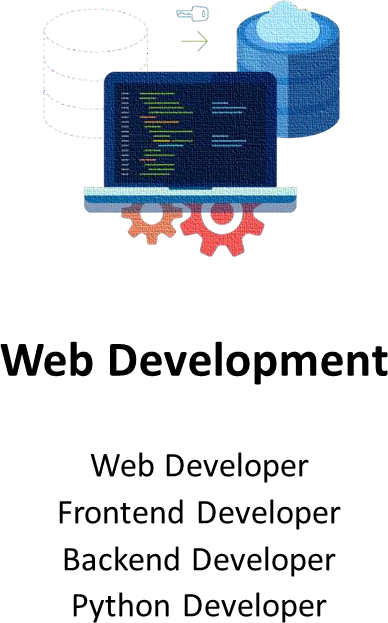
[](https://www.youtube.com/%40RishabhMishraOfficial/)

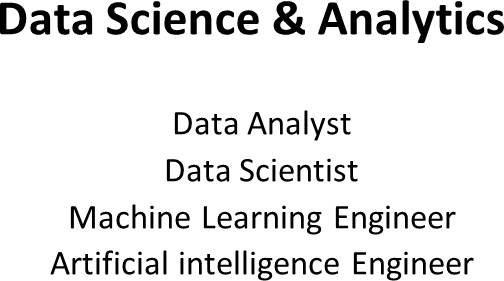
Ques2: Write a program to print numbers from 1 to 10

[](https://www.youtube.com/%40RishabhMishraOfficial/)

In above examples we can see that Python is simple in writing & reading the code.

### Careers with Python

Python is not only one of the most popular programming languages in the world, but it also offers great career opportunities. The demand for Python developers is growing every year.



# PYTHON TUTORIAL FOR BEGINNERS

**Chapter - 02**

### Python Installation & Setup + Visual Studio Code Installation

* Python installation
* Visual Studio Code installation

### Install Python

Step 1: Go to website: <https://www.python.org/downloads/> Step 2: Click on “Download Python” button

(Download the latest version for Windows or macOS or Linux or other) Step 3: Run Executable Installer

Step 4: Add Python to Path

Step 5: Verify Python Was Installed on Windows

Open the command prompt and run the following command:

### python --version

**Install Python IDE (code editor)**

Step 1: Go to website: <https://code.visualstudio.com/download> Step 2: Click on “Download” button

(Download the latest version for Windows or mac or Linux or other) Step 3: Run Executable Installer

Step 4: Click “Add to Path”

Step 5: Finish & launch the app

**Popular Python IDE**: VS Code, PyCharm, Jupyter Notebook & more

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 03**

**First Python Program**

* Print- Hello World!
* Python As a Calculator
* Running the Python code
* Python Execution Steps
* Interpreter v/s Compiler

## First Python Program - Hello World

Printing "Hello World" as the first program in Python.

**print** is a keyword word that has special meaning for Python. It means, "Display what’s inside the parentheses."

**print("Hello World")**

**Instructor = "Faizan Ali"**

**print("flython by", Instructor, sep="-")**

## Python As a Calculator

Python can be used as a powerful calculator for performing a wide range of arithmetic operations.

**2+5 ** **add two numbers print(10/5) ** **divide two numbers**

 **print sum of two numbers**

**a = 2**

**b = 5 print(a+b)**

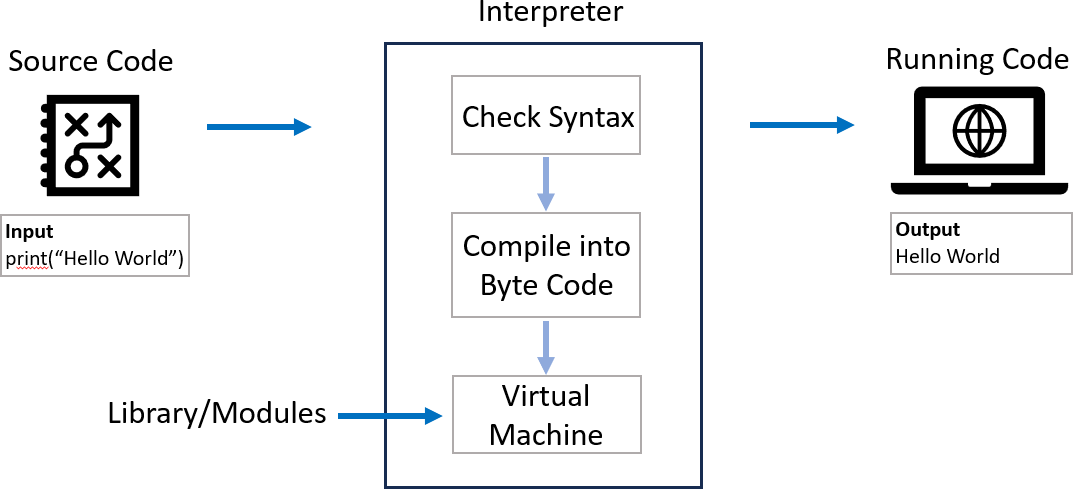
**Comments:** Comments are used to annotate codes, and they are not interpreted by Python. It starts with the hash character #

Comments are used as notes or short descriptions along with the code to increase its readability.

## Running the Python Code

* Create a new text file and inside it write – print(“Welcome to the Python Course by Rishabh Mishra”)
* Save file with extension .py – firstcode.py
* Open command prompt on windows (or Terminal on MacOS)
* Enter the location where firstcode.py file is saved – cd downloads
* Finally run the file as – python firstcode.py

## Python Execution Flow



**Python Code Execution Steps**

1. Lexical Analysis: The interpreter breaks down the code into smaller parts called tokens, identifying words, numbers, symbols, and punctuation.
2. Syntax Parsing: It checks the structure of the code to ensure it follows the rules of Python syntax. If there are any errors, like missing parentheses or incorrect indentation, it stops and shows a SyntaxError.
3. Bytecode Generation: Once the code is validated, the interpreter translates it into a simpler set of instructions called bytecode. This bytecode is easier for the computer to understand and execute.
4. Execution by PVM: The Python Virtual Machine (PVM) takes the bytecode and runs it step by step. It follows the instructions and performs calculations, assigns values to variables, and executes functions.
5. Error Handling and Output: If there are any errors during execution, like trying to divide by zero or accessing a variable that doesn't exist, the

interpreter raises an exception. If the code runs without errors, it displays any output, such as printed messages or returned values, to the user.

## Python Syntax

The syntax of the Python programming language, is the set of rules that defines how a Python program will be written and interpreted (by both the runtime system & by human readers).

my\_name = "Madhav" ⬛ my\_name = Madhav +

 **Use quotes "" for strings in flython**

## Interpreter vs Compiler

|  |  |
| --- | --- |
| **Interpreter** | **Compiler** |
| An interpreter translates and executes a source code line by line as the code runs. | A compiler translates the entire code into machine code before the program runs. |
| **Execution:** Line by line. | **Execution:** Entire program at once. |
| **Speed:** Slower execution because it translates each line on the fly. | **Speed:** Faster execution because it translates the entire program at once. |
| **Debugging:** Easier to debug as it stops at the first error encountered. | **Debugging:** Harder to debug  because errors are reported after the entire code is compiled |
| **Examples: Python**, Ruby, JavaScript, and PHP. | **Examples:** C, C++, Java, and Go. |

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 04**

**Variables in Python**

* What is a Variable
* Variables - examples
* Variable Naming Rules

## Variables in Python

A variable in Python is a **symbolic name** that is a reference or pointer to an object.

In simple terms, variables are like **containers** that you can fill in with different types of data values. Once a variable is **assigned** a value, you can use that variable in place of the value.

We assign value to a variable using the **assignment operator** (=).

**Syntax: variable\_name = value Example: greeting = "Hello World"**

**print(greeting)**

## Variable Examples

Python can be used as a powerful calculator for performing a wide range of arithmetic operations.

|  |  |  |  |
| --- | --- | --- | --- |
| **flythonLevel** | **=** | **"Beginner"** | **pascal case** |
| **pythonLevel** | **=** | **"Beginner"** | **camel case** |
| **pythonlevel** | **=** | **"Beginner"** | **flat case** |
| **python\_level** | **=** | **"Beginner"** | **Snake case** |

**x = 10**

**print(x+1) ** **add number to a variable**

**a, b, c = 1, 2, 3**

**print(a, b, c) ** **assign multiple variables**

## Variable Naming Rules

1. Must start with a letter or an underscore ( **\_** ).
2. Can contain letters, numbers, and underscores.
3. Case-sensitive (my\_name and my\_Name are different).
4. Cannot be a reserved keyword (like for, if, while, etc.).

## \_my\_name = "Madhav" ⬛ for = 26 +

### ‘for’ is a reserved word in flython

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 05**

**Data Types in Python**

* What are Data types
* Types of data types
* Data types examples

## Data Types in Python

In Python, a data type is a **classification** that specifies the **type of value** a variable can hold. We can check data type using type() function.

**Examples:**

1. **my\_name = "Madhav"**

**>>> type(my\_name) O/fl: <class 'str’>**

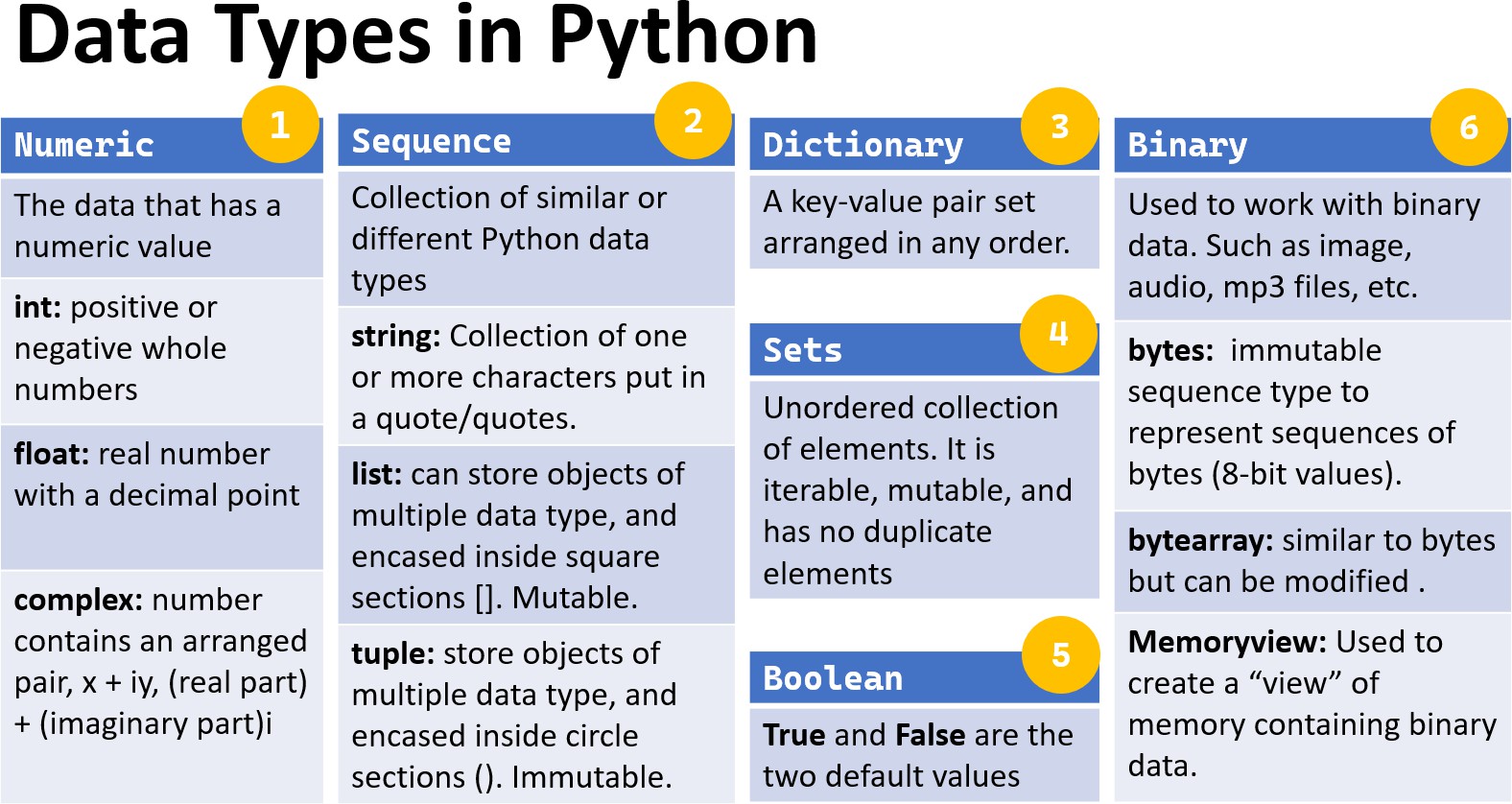
1. **value = 101**

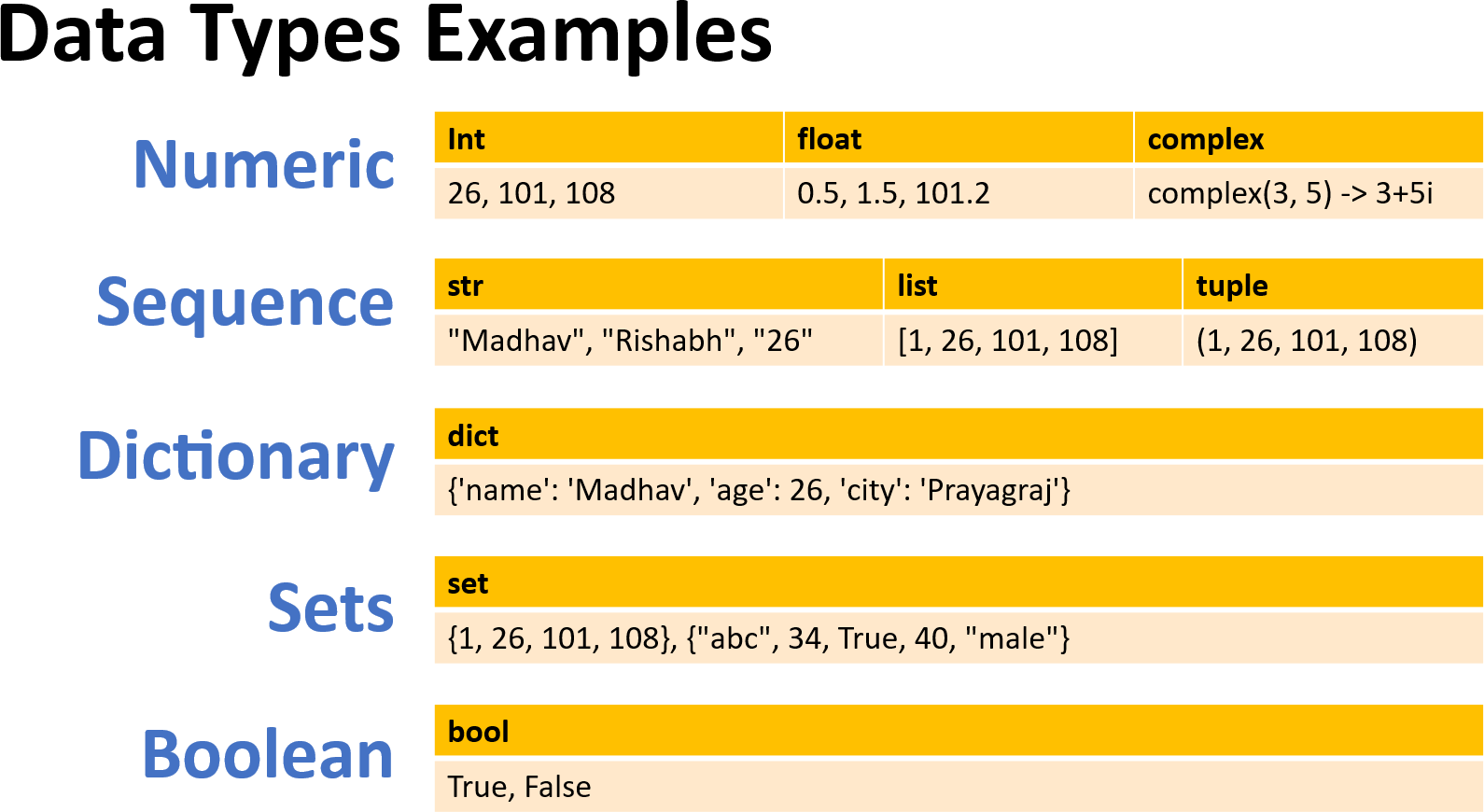
**>>> type(value) O/fl: <class 'int'>**

## Basic Data Types in Python

Python can be used as a powerful calculator for performing a wide range of arithmetic operations.

1. **umeric: Integer, Float, Complex**
2. **Sequence: String, List, Tuple**
3. **Dictionary**
4. **Set**
5. **Boolean**
6. **Binary: Bytes, Bytearray, Memoryview**



****

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 06**

**Type Casting in Python**

* What is type casting
* Type Casting examples
* Type Casting - Types

## Type Casting

Type casting in Python refers to the process of **converting** a value from **one data type to another**. This can be useful in various situations, such as when you need to perform operations between different types or when you need to format data in a specific way. Also known as **data type conversion**.

Python has several built-in functions for type casting:

**int(): Converts a value to an integer.**

**float(): Converts a value to a floating-point number. str(): Converts a value to a string.**

**list(), tuple(), set(), dict() and bool()**

## Type Casting Examples

Basic examples of type casting in python:

 Converting String to Integer: str\_num = "26"

int\_num = int(str\_num) print(int\_num)  Output: 26

print(type(int\_num))  Output: <class 'int'>

 Converting Float to Integer:

float\_num = 108.56 int\_num = int(float\_num)

print(int\_num)  Output: 108 print(type(int\_num))  Output: <class 'int'>

## Types of Typecasting

There are two types of type casting in python:

* Implicit type casting
* Explicit type casting

## Implicit Type Casting

Also known as coercion, is performed automatically by the Python interpreter. This usually occurs when performing operations between different data types, and Python implicitly converts one data type to another to avoid data loss or errors.

 Implicit type casting from integer to float num\_int = 10

num\_float = 5.5

result = num\_int + num\_float  Integer is automatically converted to float

print(result)  Output: 15.5 print(type(result))  Output: <class 'float'>

## Explicit Type Casting

Also known as **type conversion**, is performed **manually** by the programmer using built-in functions. This is done to ensure the desired type conversion and to avoid unexpected behavior.

 Converting String to Integer: str\_num = "26"

int\_num = int(str\_num) print(int\_num)  Output: 26

print(type(int\_num))  Output: <class 'int’>

 Converting a value to boolean: bool(0)  Output: False

bool(1)  Output: True

[](https://www.youtube.com/%40RishabhMishraOfficial)

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 07**

**Input Function in Python**

* Input Function – Definition
* Input Function – Example
* Handling Different Data Types

## Input Function in Python

The input function is an essential feature in Python that allows to take **input from the user**. This is particularly useful when you want to create interactive programs where the user can provide data during execution.

Also known as **user input function**. **How input Function Works:**

The input function waits for the user to type something and then press Enter. It reads the input as a string and returns it.

**Example:**

 flrompting the user for their name name = input("Enter your name: ")

 Displaying the user's input print("Hello, " + name + "!")

## Input Function – Add 2 Numbers

A simple program that takes two numbers as input from the user and prints their sum.

 flrompting the user for the first and second number num1 = input("Enter the first number: ")

num2 = input("Enter the second number: ")

 Since input() returns a string, we need to convert it to an integer

num1 = int(num1) num2 = int(num2)

 Calculating the sum and display the result

sum = num1 + num2

print("The sum of", num1, "and", num2, "is:", sum)

## Multiple Input from User & Handling different Data Types

 input from user to add two number and print result x = input("Enter first number: ")

y = input("Enter second number: ")

 casting input numbers to int, to perform sum print(f"Sum of {x} fi {y} is {int(x) + int(y)}")

## Home Work – User input and print result

Write a program to input student name and marks of 3 subjects. Print name and percentage in output.

 flrompting the user for their name and 3 subject marks name = input("Enter your name: ") hindi\_marks = input("Enter Hindi Marks: ") maths\_marks = input("Enter Maths Marks: ")

science\_marks = input("Enter Science Marks: ")

 Calculating percentage for 3 subjects

percentage = ((int(hindi\_marks) + int(maths\_marks) + int(science\_marks))/300)\*100

 flrinting the final results

print(f"{name}, have {percentage}%. Well done fi keep working hard!!")

[](https://www.youtube.com/%40RishabhMishraOfficial)

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 08**

**Operators in Python**

* What are Operators
* Types of Operators
* Operators Examples

## Operators in Python

Operators in Python are **special symbols or keywords** used to perform operations on operands (variables and values).

**Operators**: These are the special symbols/keywords. Eg: + , \* , /, etc.

**Operand**: It is the value on which the operator is applied.

 **Examples**

|  |  |  |
| --- | --- | --- |
| **Addition operator '+':** | **a** | **+ b** |
| **Equal operator '==':** | **a** | **== b** |
| **and operator 'and':** | **a** | **> 10 and b < 20** |

## Types of Operators

Python supports various types of operators, which can be broadly categorized as:

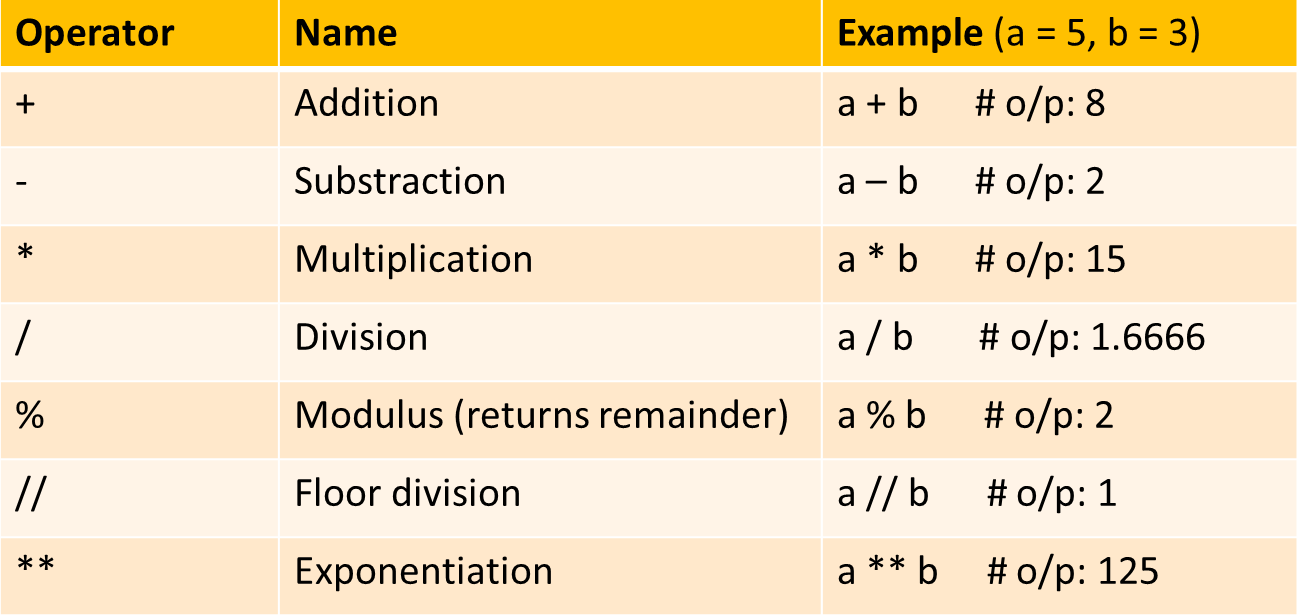
1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators
5. Bitwise Operators
6. Identity Operators
7. Membership Operators

## Operators Cheat Sheet

|  |  |
| --- | --- |
| **Operator** | **Description** |
| () | Parentheses |
| \*\* | Exponentiation |
| +, -, ~ | Positive, Negative, Bitwise NOT |
| \*, /, //, % | Multiplication, Division, Floor Division, Modulus |
| +, - | Addition, Subtraction |
| ==, !=, >, >=, <, <= | Comparison operators |
| is, is not, in, not in | Identity, **Membership** Operators |
| NOT, AND, OR | Logical NOT, Logical AND, Logical OR |
| <<, >> | Bitwise Left Shift, Bitwise Right Shift |
| &, ^, | | Bitwise AND, Bitwise XOR, Bitwise OR |

1. **Arithmetic Operators**

Arithmetic operators are used with numeric values to perform mathematical operations such as addition, subtraction, multiplication, and division.



Precedence of **Arithmetic Operators** in Python:

P – Parentheses

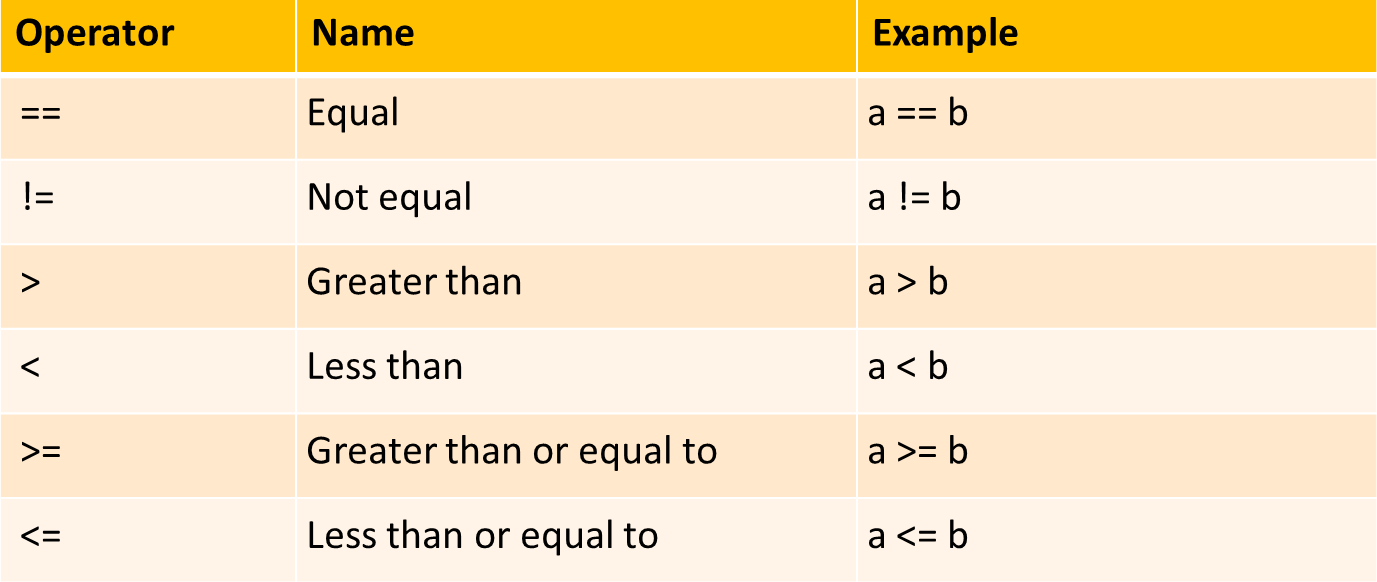
E – Exponentiation M – Multiplication D – Division

A – Addition

S – Subtraction

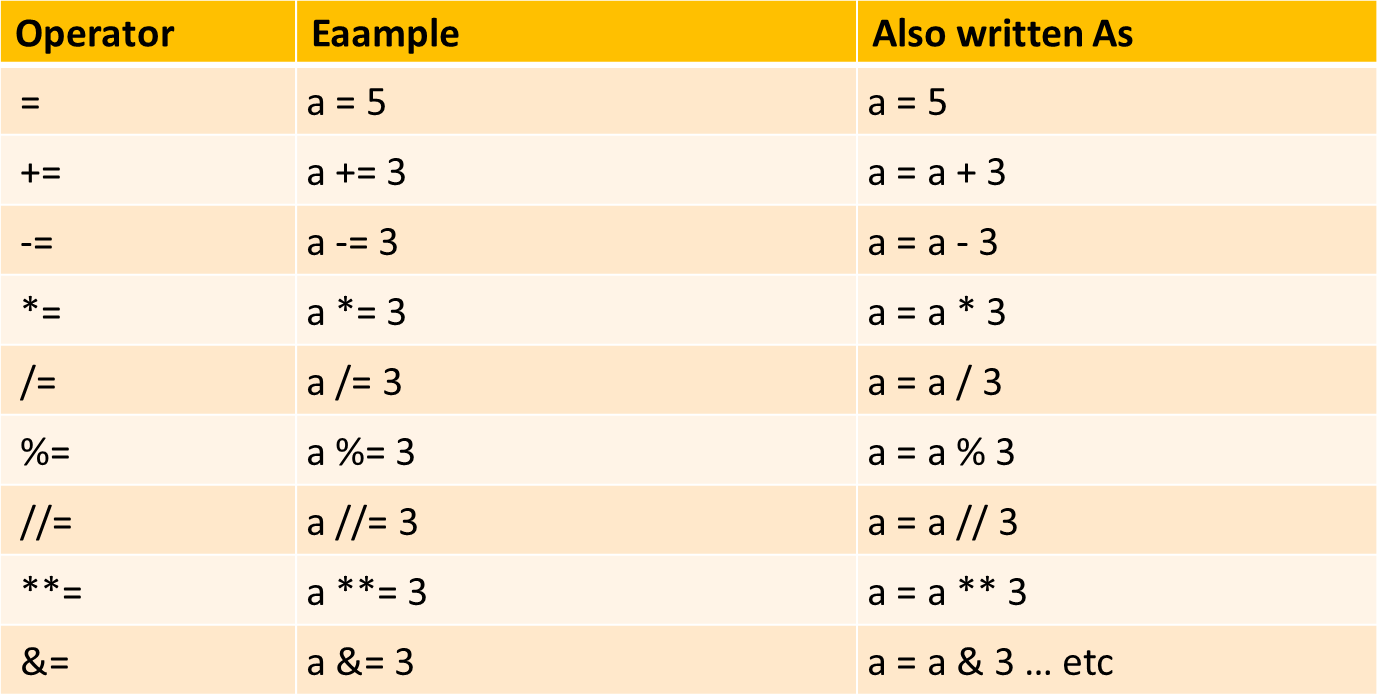
## Comparison (Relational) Operators

Comparison operators are used to compare two values and return a Boolean result (True or False).



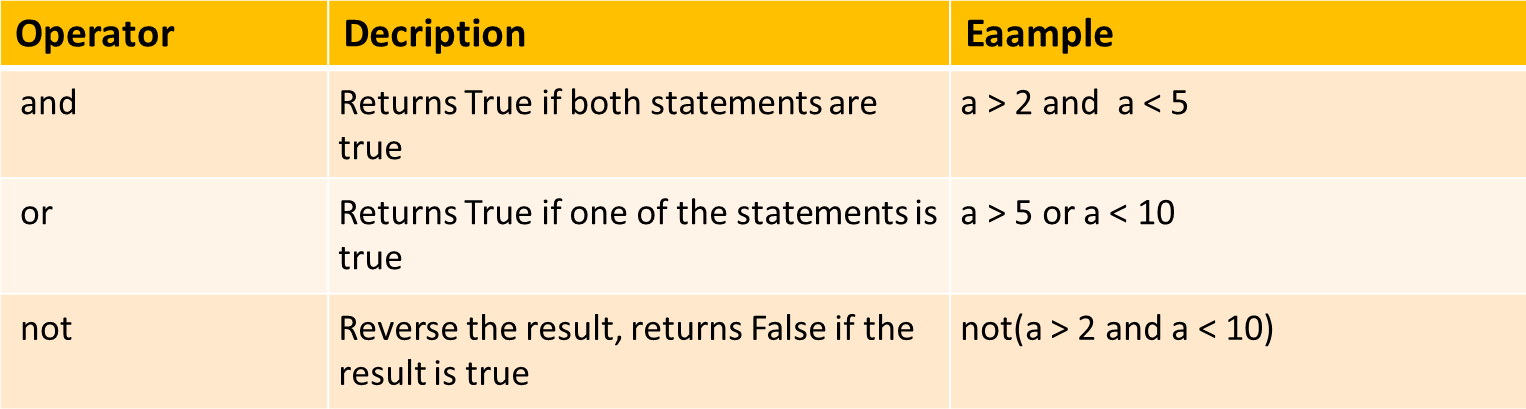
## Assignment Operators

Assignment operators are used to assign values to variables.



## Logical Operators

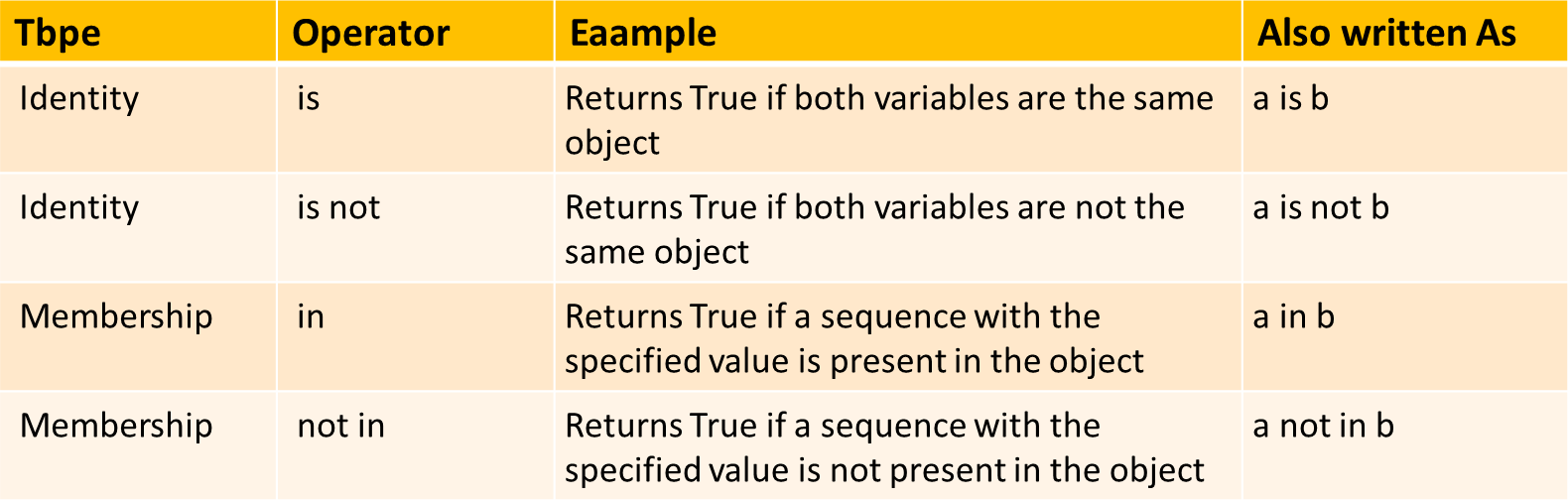
Logical operators are used to combine conditional statements.



## Identity & Membership Operators

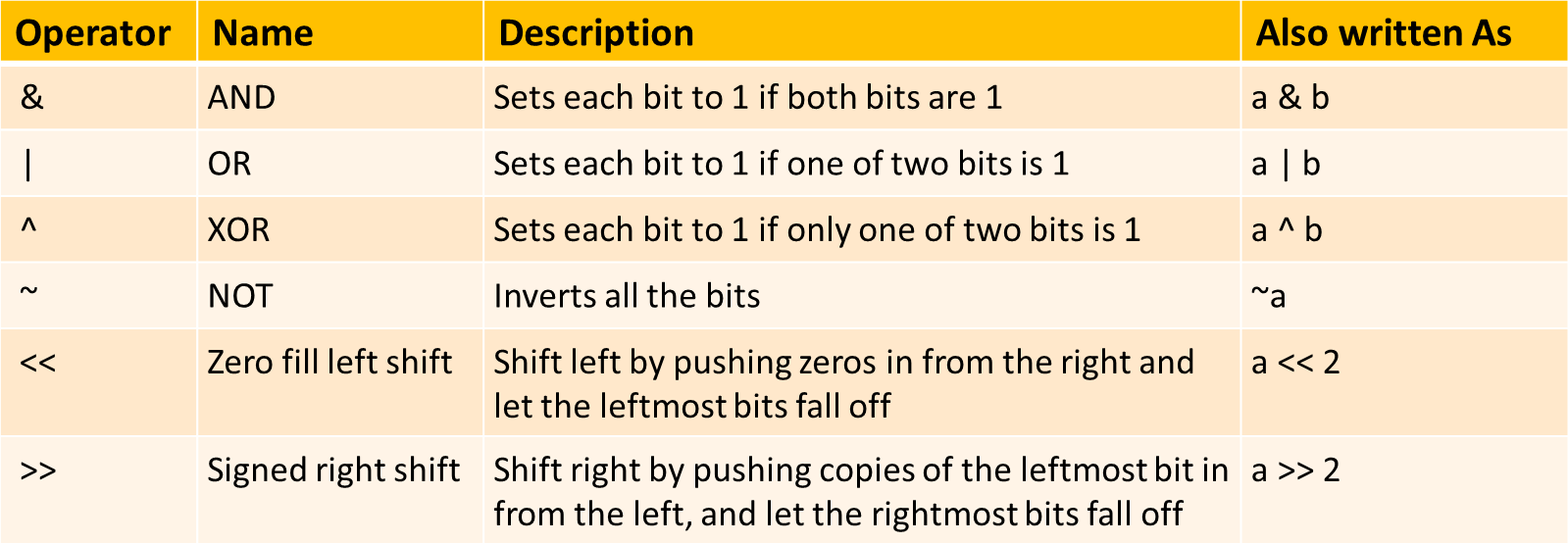
Identity operators are used to compare the memory locations of two objects, not just equal but if they are the same objects.

Membership operators checks whether a given value is a member of a sequence (such as strings, lists, and tuples) or not.



## Bitwise Operators

Bitwise operators perform operations on binary numbers.



## Bitwise Operators Example:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ------- |  |  |  |  |  | print(a fi | b) |  | |
| 0001 (This | is | the result of | 5 | fi | 3) | Output: | 1 |  | 0001 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Compare each bit in these numbers. | | | | | | Eg:1 | |
| 0101 | (This | is | 5 | in | binary) | a = 5 | 0101 |
| 0011 | (This | is | 3 | in | binary) | b = 3 | 0011 |

****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rules: 0 – False, 1 | - True | Eg:2 |  | |
| True + True = True |  | a = 5 |  | 0101 |
| True + False = False |  | b = 8 |  | 1000 |
| False + False = False |  | print(a | fi b) |  |
|  | | Output: 0  0000 | | |

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 09**

**Conditional Statements in Python**

* Conditional Statement definition
* Types of Conditional Statement
* Conditional Statement examples

## Conditional Statements in Python

Conditional statements allow you to execute code based on **condition** evaluates to True or False. They are essential for **controlling the flow** of a program and making decisions based on different inputs or conditions.

 **Examples**

a = 26

b = 108

**if** b > a**:**

print("b is greater than a")

 *Indentation - whitespace at the beginning ofi a line*

## Types of Conditional Statements

There are 5 types of conditional statements in Python:

* 1. 'if' Statement
  2. 'if-else' statement
  3. 'if-elif-else' statement
  4. Nested 'if else' statement
  5. Conditional Expressions (Ternary Operator)

## 'if' Conditional Statement

The **if** statement is used to test a condition and execute a block of code **only if the condition is true**.

### Syntax:

if condition:

 Code to execute if the condition is true

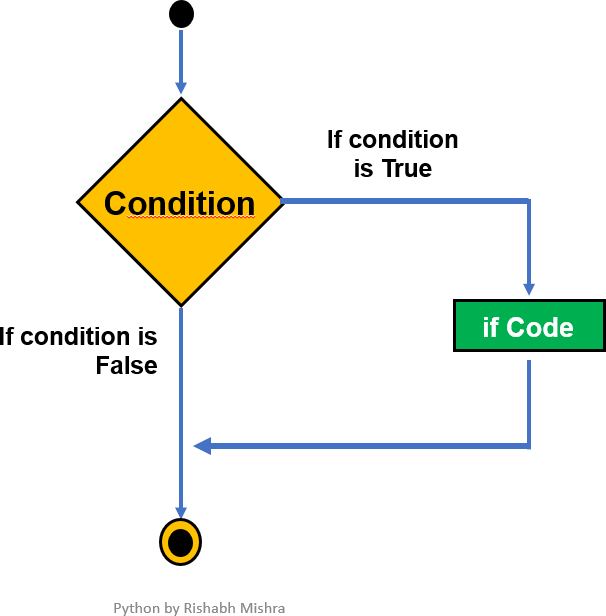
### Example:

age = 26

if age > 19**:**

print("You are an adult")

## 'if' statement flow diagram:

****

1. **'if-else' Conditional Statement**

The **if-else** statement provides an alternative block of code to execute if the condition is **false**.

### Syntax:

if condition**:**

 Code to execute if the condition is true

else**:**

 Code to execute if the condition is false

### Example:

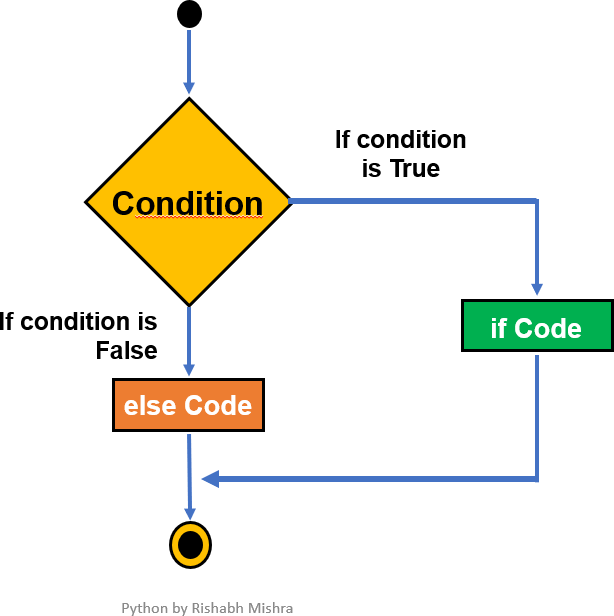
temperature = 30

if temperature > 25**:** print("It's a hot day.")

else**:**

print("It's a cool day.")

## 'if-else' statement flow diagram:

****

1. **'if-elif-else' Conditional Statement**

The **if-elif-else** statement allows to check **multiple conditions** and execute different blocks of code based on which condition is true.

### Syntax:

if condition1**:**

 Code to execute if condition1 is true elif condition2**:**

 Code to execute if condition2 is true

else**:**

 Code to execute if none of the above conditions are true

### Example:

**Grading system:** Let’s write a code to classify the student’s grade based on their total marks (out of hundred).

score = 85

if score >= 90: print("Grade - A")

elif score >= 80: print("Grade - B")

elif score >= 70: print("Grade - C")

else:

print("Grade - D")

## ested 'if-else' Conditional Statement

A **nested if-else** statement in Python involves placing an if-else statement **inside** another if-else statement. This allows for more complex decision-making by checking **multiple conditions that depend on each other**.

### Syntax:

if condition1**:**

 Code block for condition1 being True if condition2**:**

 Code block for condition2 being True

else**:**

****

else**:**

Code block for condition2 being False

 Code block for condition1 being False

**... ..**

### Example:

**Number Classification:** Let's say you want to classify a number as positive, negative, or zero and further classify positive numbers as even or odd.

number = 10

if number > 0:  *First check ifi the number is positive*

if number % 2 == 0:

print("The number is positive and even.") else:

print("The number is positive and odd.") else:  *The number is not positive*

if number == 0:

print("The number is zero.") else:

print("The number is negative.")

## Conditional Expressions

Conditional expressions provide a shorthand way to write simple if-else statements. Also known as Ternary Operator.

### Syntax:

value\_if\_true **if** condition **else** value\_if\_false

### Example:

age = 16

status = "Adult" **if** age >= 18 **else** "Minor" print(status)

## Conditional Statements- HW

**Q1: what is expected output and reason?**

value = one

if value:

print("Value is True") else:

print("Value is False")

## Q2: write a simple program to determine if a given year is a leap year using user input.

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 10**

**Functions in Python**

* Functions definition
* Types of Functions
* Function examples

## Functions in Python

A function is a block of code that **performs a specific task**. You can use it whenever you want by calling its name, which saves you from writing the same code multiple times.

**Benefits** of Using Function**:** Increases code **Readability** & **Reusability**. **Basic Concepts:**

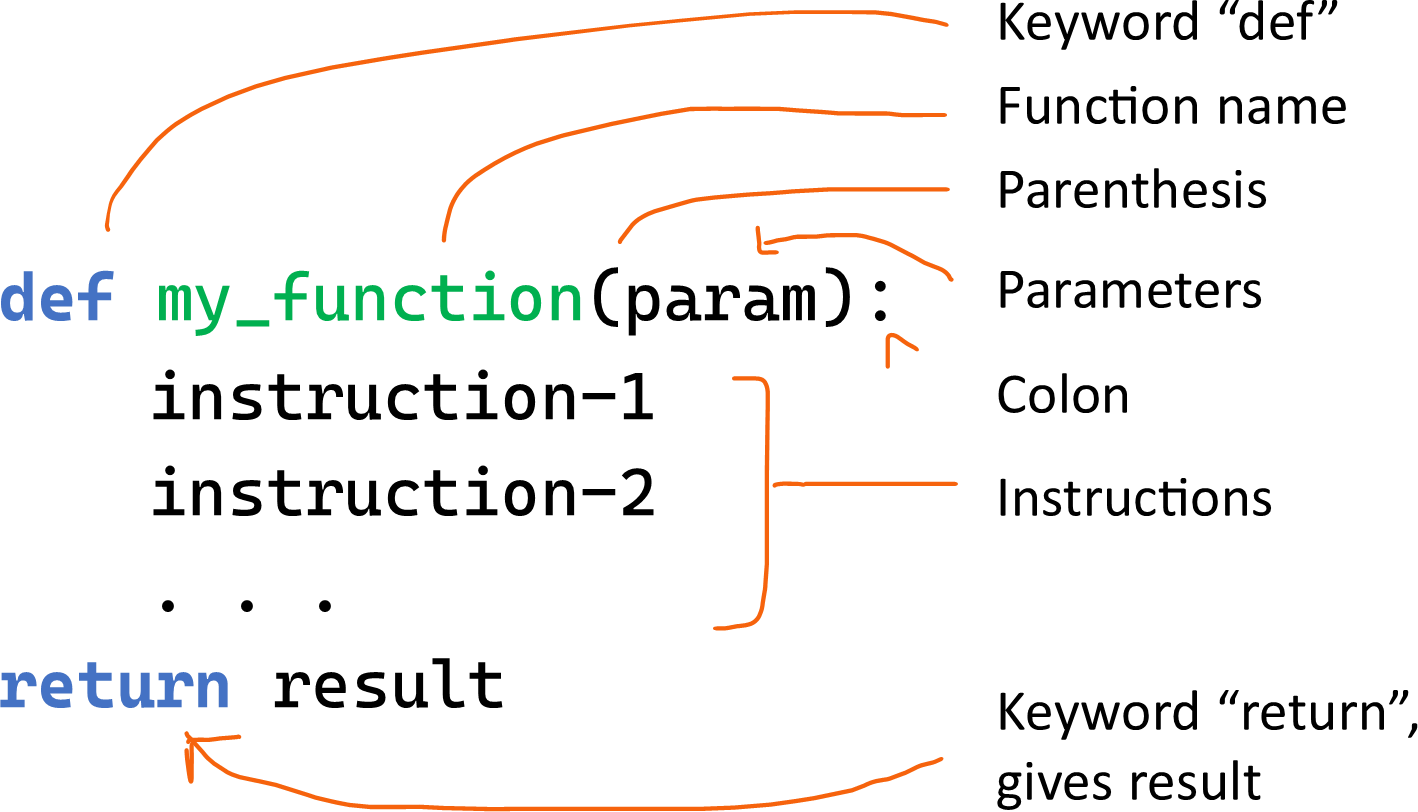
* + **Create** function: Use the def keyword to define a function.
  + **Call** function: Use the function's name followed by () to run it.
  + **Parameter**: The variable listed inside parentheses in function definition.
  + **Argument**: The actual value you pass to function when you call it.

## Types of Functions

Below are the two types of functions in Python:

1. Built-in library function:
   * These are Standard functions in Python that are available to use.
   * Examples: print(), input(), type(), sum(), max(), etc
2. User-defined function:
   * We can create our own functions based on our requirements.
   * Examples: create your own function :)

**Syntax:**

****

**# *return*** *result is optional, Use if you want the function to give back a value*

## Function without Parameters

**Example:1**

 *Create or Defiine Function*

def greetings():

print("Welcome to flython tutorial by Rishabh")

 *Use or call this Function*

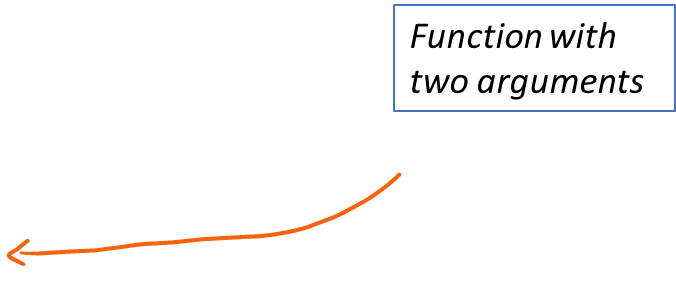
greetings()

 Output: Welcome to flython tutorial by Rishabh

## Function with Parameters

**Example:2**

 function to adds two numbers fi print result. def add2numbers(a, b):

result = a + b

print("The sum is:", result)

 Calling this function with arguments add2numbers(**5**, **3**)

 Output: The sum is: 8

## The return Statement

The return statement is used in a function to **send a result back** to the place where the function was called. When return is executed, the function **stops running** and immediately returns the specified value.

**Example:**

def add(a, b):

**return** a + b  *This line sends back sum ofi a and b*

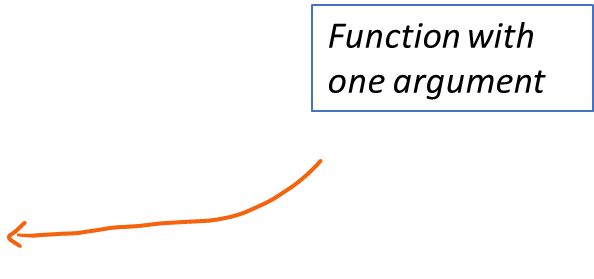
result = add(3, 5) print(result)

 *Output: 8*

## Function with a Return value

**Example:3**

 *fiunction to convert Celsius to Fahrenheit*

def celsius\_to\_fahrenheit(celsius): fahrenheit = (celsius \* 9/5) + 32 return Fahrenheit

*value*

 *Calling this fiunction to return a*

temp\_f = celsius\_to\_fahrenheit(**25**) print("Temperature in Fahrenheit:", temp\_f)

 Output: Temperature in Fahrenheit: 77.0

## The pass Statement

The pass statement is a placeholder in a function or loop. It **does nothing** and is used when you need to write code that will be added **later** or to define an **empty** function.

**Example:**

### def myfunction():

**pass ** ***This does nothing fior now***

## Functions – HW

Write a Python program to create a **calculator** that can perform at least **five**

[](https://www.youtube.com/%40RishabhMishraOfficial)different mathematical operations such as addition, subtraction, multiplication, division and average. Ensure that the program is user-friendly, prompting for input and displaying the results clearly.

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 11**

**Function Arguments in Python**

* Function Arguments
* Types of Functions Arguments
* Function Arguments examples

## Arguments in Function

Arguments are the values that are passed into a function when it’s called. A function must be called with the right number of arguments. If a function has 2 parameters, you must provide 2 arguments when calling it.

**Example**: function defined using one parameter (variable)

def greetings(name):  *name is a parameter*

print("Hello, " + name + "!")

greetings("Madhav")  *adhav as argument*

 Output: Hello, Madhav!

## Types of Function Arguments

Python supports various types of arguments that can be passed at the time of the function call.

1. Required arguments (Single/Multiple arguments)
2. Default argument
3. Keyword arguments (named arguments)
4. Arbitrary arguments (variable-length arguments \*args and \*\*kwargs)

**Required Arguments (same as above)**

Required arguments are the arguments passed to a function in correct positional order. A function must be called with the right number of arguments. If a function has 2 parameters, you must provide 2 arguments when calling it.

**Example**: function defined using one parameter (variable)

def greetings(name):  *name is a parameter*

print("Hello, " + name + "!")

greetings("Madhav")  *adhav as argument*

 Output: Hello, Madhav!

## Default Arguments

You can assign **default** values to arguments in a function definition. If a value **isn't**

provided when the function is called, the default value is used.

**Example:** function defined using one parameter & default value

|  |  |  |  |
| --- | --- | --- | --- |
| def greetings(name | = | "World"): | *defiault value* |
| print("Hello, " | + | name + "!") |  |

greetings()  *No argument passed*

 Output: Hello, World!

greetings("Madhav")  *adhav as argument*

 Output: Hello, Madhav!

## Keyword Arguments

When calling a function, you can specify arguments by the parameter **name**. These are called **keyword arguments** and can be given in **any order**.

**Example:** function defined using two parameters

def divide(a, b):  *a,b are 2 parameters*

return a / b

result = divide(b=10, a=20)  *with keyword arguments*

print(result)  *Output: 2*

result = divide(10, 20)  *positional arguments*

print(result)  *Output: 0.5*

## Arbitrary Positional Arguments (\*args)

If you're unsure how many arguments will be passed, use \*args to accept any number of positional arguments.

**Purpose:** Allows you to pass a variable number of **positional arguments**. **Type:** The arguments are stored as a **tuple**.

**Usage:** Use when you want to pass multiple values that are accessed by position.

### Example 1:

def add\_numbers(\*args): return sum(args)

 *Any number ofi arguments*

result = add\_numbers(1, 2, 3, 4)

print(result)  Output: 10

**Note**: Here, \*args collects all the passed arguments into a tuple, & sum() function adds them.

### Example 2:

def greetings(\*names): for name in names:

print(f"Hello, {name}!") greetings("Madhav", "Rishabh", "Visakha")  *Output:*

Hello, Madhav! Hello, Rishabh! Hello, Visakha!

## Arbitrary Keyword Arguments (\*\*kwargs)

If you want to pass a variable number of keyword arguments, use \*\*kwargs.

**Purpose:** Allows you to pass a variable number of **keyword arguments** (arguments with names).

**Type:** The arguments are stored as a **dictionary**.

**Usage:** Use when you want to pass multiple values that are accessed by name.

### Example 1:

def print\_details(\*\*kwargs):

for key, value in kwargs.items(): print(f"{key}: {value}")

print\_details(name="Madhav", age=26, city="Delhi")

 *Output:*

name: Madhav age: 26 city: Delhi

### Example 2:

def shopping\_cart(\*\*products): total = 0

print("Items flurchased:")

for item, price in products.items():

print(f"{item}: ₹{price}")

total += price

print(f"Total: ₹{total}")

 *multiple keyword arguments*

shopping\_cart(apple=15, orange=12, mango=10)

 *Output:*

Items flurchased: apple: ₹15 orange: ₹12 mango: ₹10 Total: ₹37

[](https://www.youtube.com/%40RishabhMishraOfficial)

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 12**

**Strings in Python (Part-1)**

* Strings and Examples
* Formatted Strings
* Escape Characters
* String Operators

## Strings in Python

A string is a sequence of **characters**. In Python, strings are enclosed within single (') or double (") or triple (""") quotation marks.

### Examples:

print('Hello World!')  *use type() to check data type*

print("Won’t Give Up!")

print('''"Quotes" and 'single quotes' can be tricky.''') print("\"Quotes\" and 'single quotes' can be tricky.")

## Types of Function Arguments

A formatted string in Python is a way to **insert variables** or expressions inside a string. It allows you to format the output in a readable and controlled way.

There are multiple ways to format strings in Python:

1. Old-style formatting (% operator)
2. str.format() method
3. F-strings (formatted string literals)

## Formatted String - % Operator

### Old-style formatting (% operator)

This approach uses the **%** operator and is similar to string formatting in languages like C.

**Syntax:** "string % value"

### Example:

name = "Madhav" age = 16

print("My name is %s and I’m %d." **%** (name, age))

 *%s, %d are placeholders fior strings and integers*

## Formatted String - str.format()

### str.format() method

In Python 3, the **format**() method is more powerful and flexible than the old-style **%**

formatting.

**Syntax:** "string {}"**.format**(value)

### Example:

name = "Madhav" age = 16

|  |  |  |
| --- | --- | --- |
| print("My | name | is {} and I’m {}."**.format**(name, age)) |
| *You can* | *also* | *refierence the variables by index or keyword:* |
| print("My | name | is {0} and I’m {1}."**.format**(name, age)) |
| print("My | name | is {name} and I’m {age}."**.format**(name="Madhav", |
| age=28)) |  |  |

## Formatted String – F-strings

### F-strings (formatted string literals)

In Python 3.6, F-strings are the most concise and efficient way to format strings. You

prefix the string with an f or F, and variables or expressions are embedded directly within curly braces **{}**.

**Syntax:** f"string **{**variable**}**"

### Example:

name = "Madhav" age = 16

print(f"My name is {name} and I’m {age}.")

 *You can also perfiorm expressions inside the placeholders:*

print(f"In 5 years, I will be {age + 5} years old.")

## Escape Characters

Escape characters in Python are **special** characters used in strings to represent

whitespace, symbols, or control characters that would otherwise be difficult to include. An escape character is a **backslash \** followed by the character you want to insert.

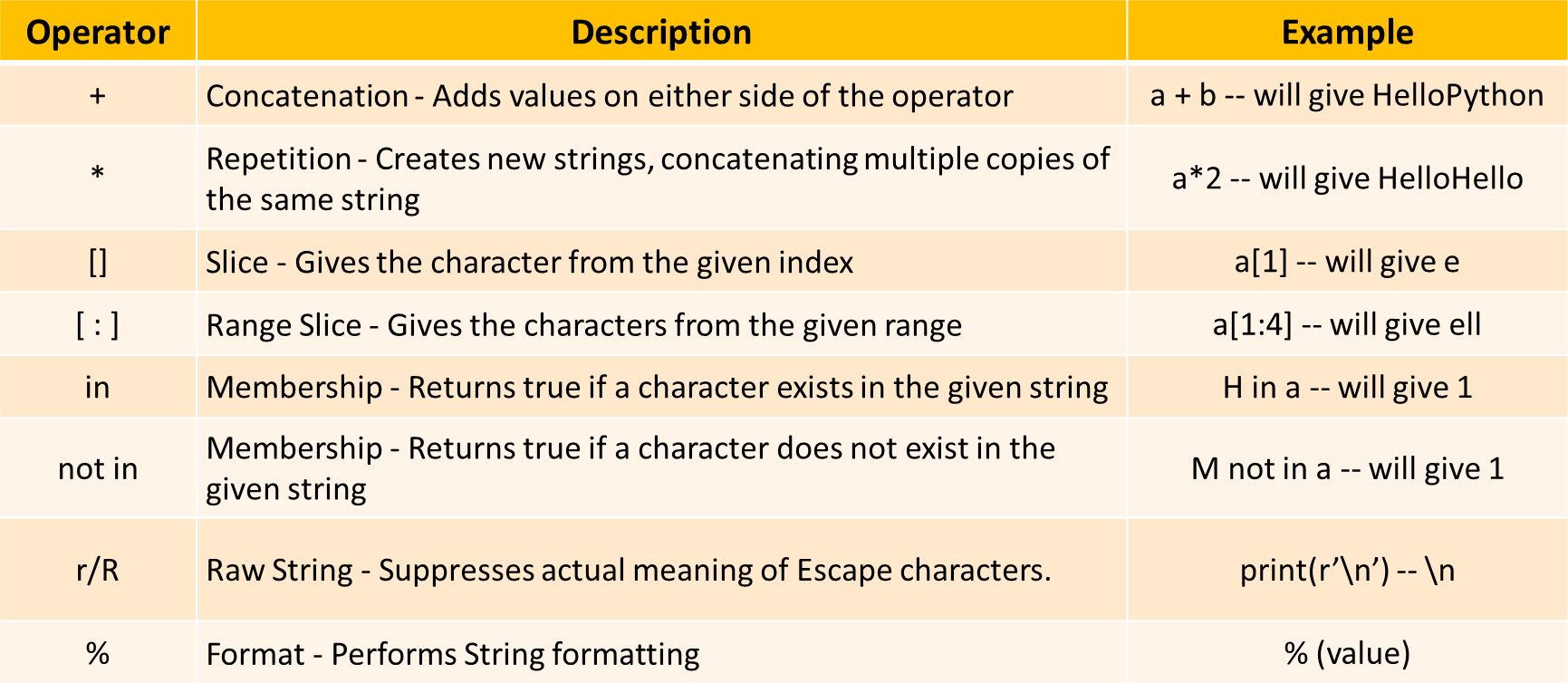
### Examples:

print('Hello\nWorld!')  *\n fior new line*

print('Hello\tWorld!')  *\t fior tab*

print("\"Quotes\" and 'single quotes' can be tricky.")  *print single and double quotes*

## String Operators

****

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 13**

**Strings in Python (Part-2)**

* String Indexing
* String Slicing
* String Methods

## String Indexing

You can access individual characters in a string using their **index**. Python uses **zero-based**

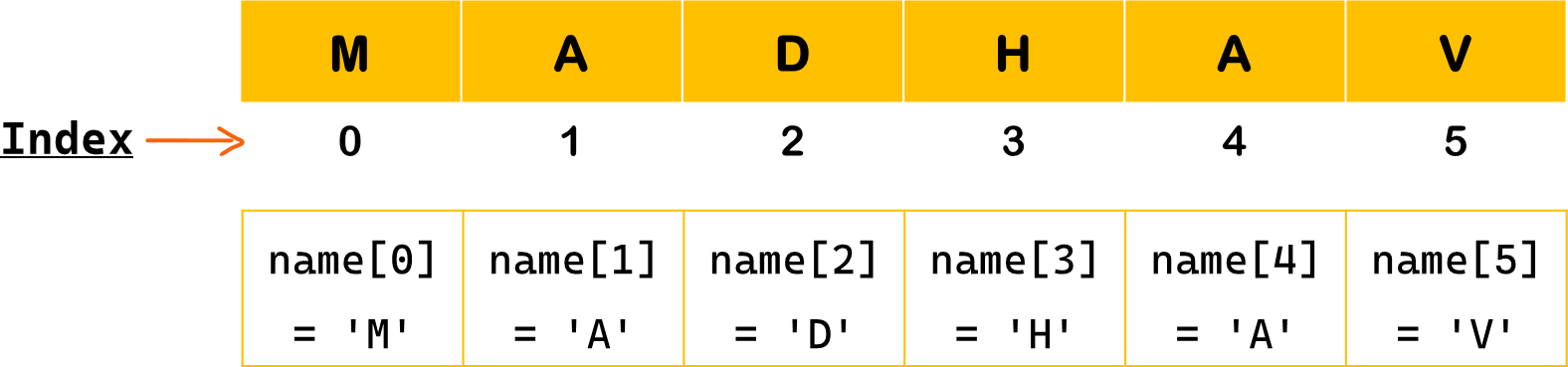
indexing, meaning the first character has an index of **0**. **Index:** Position of the character.

**Syntax:**

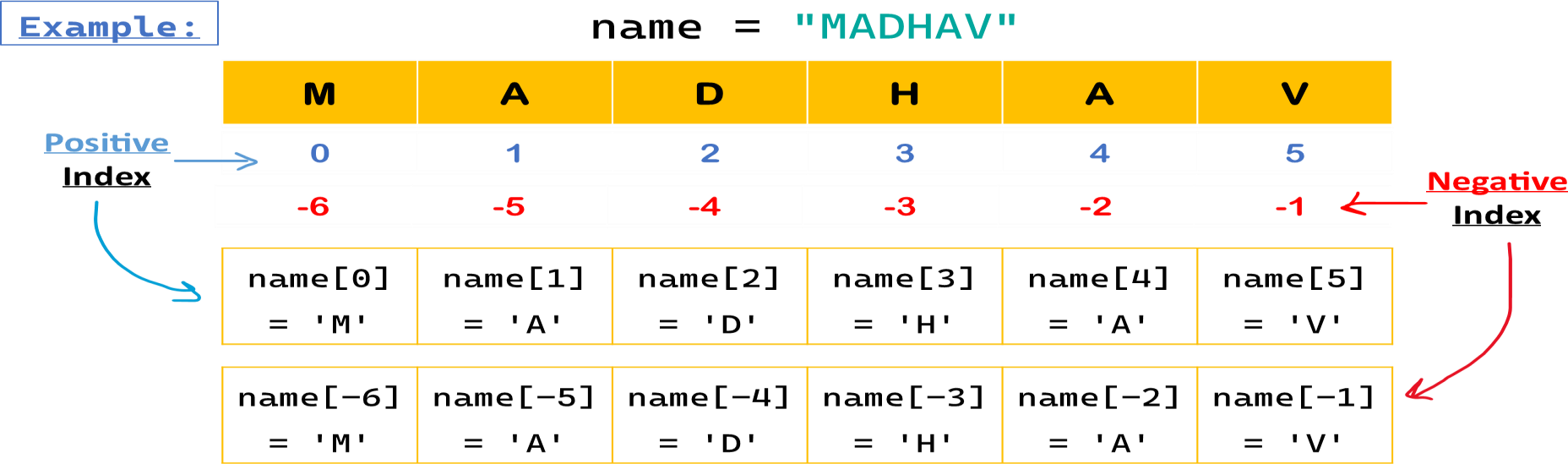
**string[Index\_Value]**

**Example:**

name = "MADHAV"



## String Indexing – Positive & Negative Index

****

**String Slicing**

Slicing in Python is a feature that enables **accessing parts** of the sequence. String slicing allows you to get subset of characters from a string using a specified **range of indices**.

### Syntax:

**string[start : end : step]**

* **start** : The index to start slicing (inclusive). Default value is 0.
* **end** : The index to stop slicing (exclusive). Default value is length of string.
* **Step** : How much to increment the index after each character. Default value is 1.

### Example:

name = "MADHAV" name[0:2] = 'MA'

name[0:5:2] = 'MDA'

## String Slicing - Examples

### Example:

name = "MADHAV"

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| name[0:1] | = | name[:1] | = | 'M' | *fiirst* | *char* |
| name[0:2] | = | name[:2] | = | 'MA' | *fiirst* | *2 chars* |
| name[2:5] | = | 'DHA' |  |  | *third* | *to fiifith chars* |
| name[5:] = name[-1:] | | | = | 'V' | *last char* | |
| name[4:] = name[-2:] | | | = | 'AV' | *last 2 chars* | |

name[0:5:2] = name[0::2] = 'MDA'  *every second chars* name[1:-1] = 'ADHA'  *exclude fiirst & last chars* name[:] = name[::] = 'MADHAV'  *all chars*

name[::-1] = 'VAHDAM'  *reverse the string*

## String Methods

****

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 14**

**Loops in Python**

* Loops & Types
* While Loop
* For Loop
* Range Function
* Loop Control Statements

## Loops in Python

Loops enable you to perform **repetitive tasks** efficiently without writing redundant code. They

iterate over a sequence (like a list, tuple, string, or range) or execute a block of code as long as a specific **condition is met**.

### Types of Loops in Python

1. While loop
2. For loop
3. Nested loop

## While Loop

The while loop **repeatedly** executes a block of code as long as a given condition remains

True. It checks the condition before each iteration.

**Syntax:**

**while** condition**:**

 *Code block to execute*

**Example:** *Print numbers firom 0 to 3*

count = 0

**while** count < 4:  *Condition*

print(count) count += 1

 *Output: 0 1 2 3*

## While Loop Example

**else Statement:** An else clause can be added to loops. It executes after the loop finishes normally (i.e., not terminated by break). ***Example:***

count = 3

**while** count > 0:  *Condition* print("Countdown:", count) count -= 1

### else:

print("Liftoff!")  *Run afiter while loop ends*

## For Loop

The for loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string) and execute a block of code for **each element** in that sequence.

**Syntax:**

**for** variable **in** sequence**:**

 *Code block to execute*

**Example:** *iterate over each character in language*

lat:ua:v = 'flQtkot’

**for** x **in** language:

print(x)  *Output: P y t h o n*

## Using range() Function

To **repeat** a block of code a specified number of times, we use the range() function. The range() function returns a sequence of numbers, starting from 0 by default,

increments by 1 (by default), and stops before a specified number.

### Syntax:

**range**(stop) **range**(start, stop) **range**(start, stop, step)

* + **start**: (optional) The beginning of the sequence. Defaults is 0. (inclusive)
  + **stop**: The end of the sequence (exclusive).
  + **step**: (optional) The difference between each number in the sequence. Defaults is 1.

## range() Function Example

**Example1:** *Basic usage with One Argument - Stop*

**for** i in range(5): print(i)

 *Output: 0 1 2 3 4*

**Example2:** *Basic usage with Start, Stop and Step*

**for** i in range(1, 10, 2): print(i)

 *Output: 1 3 5 7 9*

## For Loop Example

**else Statement:** An else clause can be added to loops. It executes after the loop finishes normally (i.e., not terminated by break).

**Example:**

**for** i **in** range(3): print(i)

### else:

print("Loop completed")

 *Output: 0 1 2 Loop Completed*

## while loop VS for loop

### while loop

* + - A while loop keeps running as long as a **condition is true**.
    - It is generally used when you **don’t know** how many iterations will be needed beforehand, and loop continues based on a condition.

### for loop

* + - A for loop **iterates over a sequence** (like a strings, list, tuple, or range) and runs the loop for each item in that sequence.
    - It is used when **you know** in advance how many times you want to repeat a block of code.

## Loop Control Statements

Loop control statements allow you to **alter** the normal flow of a loop. Python supports 3 clauses within loops:

* + - **pass statement**
    - **break Statement**
    - **continue Statement**

## Loop Control - pass Statement

**pass Statement:** The pass statement is used as a **placeholder** (**it does nothing**) for the future code, and runs entire code without causing any syntax error. *(already covered in functions)*

**Example:**

**for** i **in** range(5):

 code to be updated

### pass

Above example, the loop **executes** without error using **pass** statement

## Loop Control - break Statement

**break Statement:** The break statement terminates the loop entirely, exiting from it immediately.

**Example:**

**for** i **in** range(5):

**if** i == 3:

### break

print(i)  *Output: 0 1 2*

Above example, the loop **terminated** when condition met true for i == 3

## Loop Control - continue Statement

**continue Statement:** The continue statement **skips** the current iteration and moves to the next one.

**Example:**

**for** i **in** range(5):

**if** i == 3:

### continue

print(i)  *Output: 0 1 2 4*

Above example, the loop **skips** when condition met true for i == 3

## break vs continue Statement break Statement example

 *pass statement*

count = 5

while count > 0: if count == 3:

### pass

else:

print(count) count -= 1

 *Output: 5 4 2 1*

## continue Statement example

 *continue statement: don't try - infiinite loop*

count = 5

while count > 0: if count == 3:

### continue

else:

print(count) count -= 1

 *Output: 5 4 3 3…….*

## Validate User Input

 ***validate user input:*** *controlled infiinite* ***while*** *loop using*

***break*** *statement*

**while** True:

user\_input = input("Enter 'exit' to STOfl: ")

**if** user\_input == 'exit':

print("congarts! You guessed it right!")

### break

print("sorry, you entered: ", user\_input)

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 15**

**Nested Loops in Python**

* Nested Loops Definition
* Nested Loops Examples
* Nested Loops Interview Ques

## Nested Loops in Python

**Loop inside another loop** is nested loop. This means that for every single time the outer loop runs, the inner loop **runs all of its iterations**.

## Why Use Nested Loops?

* Handling Multi-Dimensional Data: Such as matrices, grids, or lists of lists.
* Complex Iterations: Operations depend on multiple variables or dimensions.
* Pattern Generation: Creating patterns, such as in graphics or games.

## Nested Loop Syntax

**Syntax:**

### Outer\_loop:

**inner\_loop:**

 *Code block to execute -* ***innsr*** *loop * *Code block to execute -* ***outsr*** *loop*

## Nested Loop Example

**Example:** *Print numbers firom 1 to 3, fior 3 times using* ***fior-fior***

*nested loop*

**for i** in range(3)**: ** *Outer fior loop (runs 3 times)*

**for j** in range(1,4)**:** print(**j**)

print()

**Example:** *Print numbers firom 1 to 3, fior 3 times using* ***wkils- fior*** *nested loop*

**i** = 1

**while i** < 4:  *Outer while loop (runs 3 times)*

**for j** in range(1, 4): print(**j**)

print()

**i** += 1

## Nested Loop Interview Question

**Example:** *Print* **p*ri½s*** *numbers firom 2 to 10*

**for** num in range(2, 10):

**for** i in range(2, num): if num % i == 0:

break

else:

print(num)

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 16 List in Python**

* What is List
* Create Lists
* Access List: Indexing & Slicing
* Modify List
* List Methods
* Join Lists
* List Comprehensions
* Lists Iteration

### List in Python

A **list** in Python is a collection of items (elements) that are **ordered**, **changeable**

(mutable), and allow **duplicate** elements.

Lists are one of the most versatile data structures in Python and are used to store multiple items in a single variable.

**Example:**

fruits = ["apple", "orange", "cherry", "apple"] print(fruits)

 Output: ['apple', 'orange', 'cherry', 'apple']

### Create List in Python

You can **create lists** in Python by placing **comma-separated** values between **square brackets []**. Lists can contain elements of different data types, including other lists.

**Syntax: list\_name** = [element1, element2, element3, **...**]

 *List ofi strings*

colors = ["red", "green", "blue"]

** *List ofi integers* numbers = [1, 2, 3, 4, 5]  *ixed data types*

mixed = [1, "hello", 3.14, True]

 *Nested list*

nested = [1, [2, 3], [4, 5, 6]]

### Accessing List Elements - Indexing

You can **access elements** in a list by referring to their **index**. Python uses **zero-based**

indexing, meaning the first element has an index of **0**.

|  |  |  |
| --- | --- | --- |
| **Syntax:** | **list\_name**[index] |  |
| **Example:** |  |  |
| fruits = | ["apple", "orange", | "cherry", "apple", "mango"] |



 *Access fiirst element* print(fruits[0])  *Output: apple * *Access third element* print(fruits[2])  *Output: cherry*

 *Access last element using negative index*

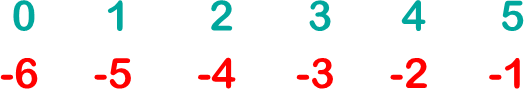
print(fruits[-1])  *Output: mango*

### List Slicing

**Slicing** allows you to **access a range of elements** in a list. You can specify the **start and stop indices**, and Python returns a new list containing the specified elements.

**Syntax: list\_name**[start:stop:step]

**Example:** numbers = [10, 20, 30, 40, 50, 60]



 *Slice firom index 1 to 3*

print(numbers[1:4])  *Output: [20, 30, 40]*

 *Slice firom start to index 2*

print(numbers[:3])  *Output: [10, 20, 30]*

 *Slice all alternate elements*

print(numbers[0::2])  *Output: [10, 30, 50]*

 *Slice with negative indices*

print(numbers[-4:-1])  *Output: [30, 40, 50]*

 *Reverse list*

print(numbers[::-1])  *Output: [60,50,40,30,20,10]*

### Modifying List

Lists are **mutable**, meaning you can change their content after creation. You can **add**, **remove**, or **change** elements in a list.

 **Initial list:** fruits = ["apple", "banana", "cherry"]

 *Changing an element*

fruits**[1]** = "blueberry"

print(fruits)  *Output: ['apple', 'blueberry', 'cherry']*

 *Adding an element*

fruits**.append**("mango")

print(fruits)  *Output: ['apple', 'blueberry', 'cherry’, 'mango']*

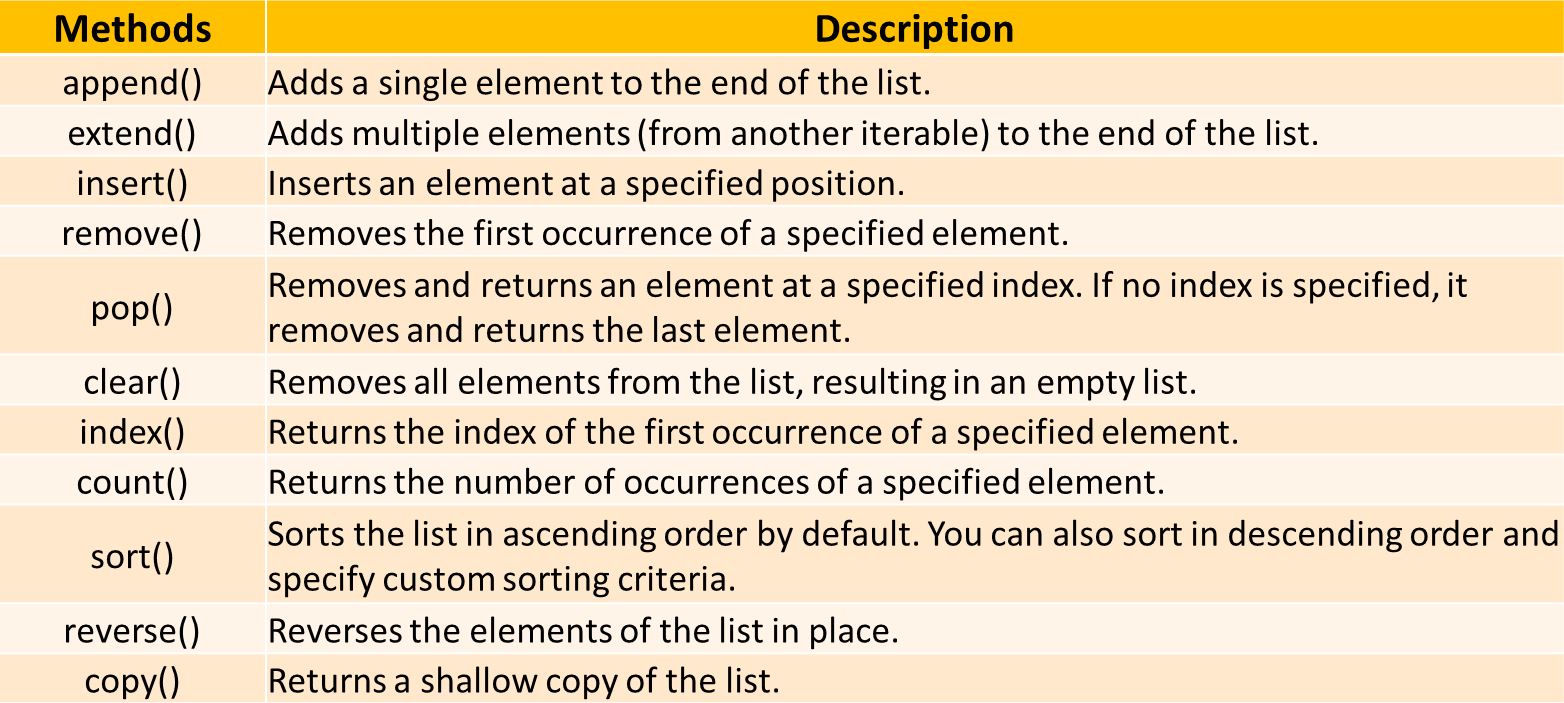
 *Removing an element*

fruits**.remove**("cherry")

print(fruits)  *Output: ['apple', 'blueberry', 'mango']*

### List Methods

Python provides several **built-in methods** to **modify** and **operate** on lists. Eg:



### Join Lists

There are several ways to **join**, or **concatenate**, two or more lists in Python.

list1 = [1, 2] list2 = ["a", "b"]

 One of the easiest ways are by using the + operator list3 = list1 + list2

print(list3)  *Output: [1, 2, 'a', 'b']*

 *using append method*

**for** x in list2: list1**.append**(x)

 *appending all the items firom list2 into list1, one by one*

print(list1)  *Output: [1, 2, 'a', 'b']*

 *using extend method*

list1**.extend**(list2)  *add elements firom one list to another list*

print(list1)  *Output: [1, 2, 'a', 'b']*

### List Comprehensions

List comprehensions provide a **concise way to create lists**. They consist of brackets containing an expression followed by a for clause, and optionally if clauses.

### Syntax:

new\_list = [expression **for** item **in** iterable **if** condition]

 *Creating a list ofi squares:*

squares = [x\*\*2 for x in range(1, 6)] print(squares)  *Output: [1, 4, 9, 16, 25]*

 *Filtering even numbers:*

even\_numbers = [x for x in range(1, 11) if x % 2 == 0] print(even\_numbers)  *Output: [2, 4, 6, 8, 10]*

 *Applying a fiunction to each element:*

fruits = ["apple", "banana", "cherry"] uppercase\_fruits = [fruit.upper() for fruit in fruits]

print(uppercase\_fruits)  *Output: ['APPLE', 'BANANA', 'CHERRY']*

### List Comprehensions - Flatten a List

**Flatten a ested List - using List Comprehension**

def flatten\_list(lst):

return [item **for** sublist **in** lst **for** item **in** sublist]

#### Example

nested\_list = [[1, 2], [3, 4], [5, 6]] flattened = flatten\_list(nested\_list) print(flattened)

 *Output: [1, 2, 3, 4, 5, 6]*

### Iterating Over Lists

Iterating allows you to traverse each element in a list, typically using loops.

**Example:** fruits = ["apple", "banana", "cherry"]

 **Using for loop for** fruit in fruits:

print(fruit)

### Using while loop

index = 0

**while** index < len(fruits): print(fruits[index]) index += 1

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 17**

**Tuple in Python**

* What is Tuple
* Create Tuples
* Access Tuples: Indexing & Slicing
* Tuple Operations
* Tuple Iteration
* Tuple Methods
* Tuple Functions
* Unpack Tuples
* Modify Tuple

## Tuple in Python

A **tuple** is a collection of items in Python that is **ordered**, **unchangeable** (**immutable**) and allow **duplicate** values.

Tuples are used to store multiple items in a single variable.

**Note**: Ordered – Tuple items have a defined order, but that order will not change.

**Example:**

fruits = **(**"apple", "orange", "cherry", "apple"**)** print(fruits)

 Output: **(**'apple', 'orange', 'cherry', 'apple'**)**

### Create Tuple in Python

There are several ways to create a **tuple** in Python:

### Using flarentheses ()

colors = ("red", "green", "blue") numbers = (1, 2, 3, 4, 5)

mixed = (1, "hello", 3.14, True)

nested = (1, [2, 3], (4, 5, 6))

### Without flarentheses (Comma-Separated)

also\_numbers = 1, 2, 3, 4, 5

### Using the tuple() Constructor

new\_tuple = tuple(("apple", "banana", "cherry"))  *use doble brackets*

list\_items = ["x", "y", "z"]  *Creating a tuple firom a list*

tuple\_items = **tuple**(list\_items)  *('x', 'y', 'z’)*

### Single-Item Tuple

tuplesingle = ("only"**,**)

### Accessing Tuple Elements - Indexing

You can **access elements** in a tuple by referring to their **index**. Python uses **zero-based**

indexing, meaning the first element has an index of **0**.

**Syntax: tuple\_name**[index]

**Example:**

fruits = **(**"apple", "orange", "cherry", "apple", "mango"**)**

****

 *Access fiirst element*

print(fruits[0])  *Output: apple*

 *Access third element*

print(fruits[2])  *Output: cherry*

 *Access last element using negative index*

print(fruits[-1])  *Output: mango*

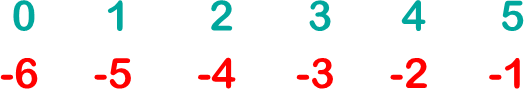
### Tuple Slicing

**Slicing** allows you to **access a range of elements** in a tuple. You can specify the **start and stop indices**, and Python returns a new tuple containing the specified elements.

**Syntax: tuple\_name**[start:stop:step]

**Example:**

numbers = (10, 20, 30, 40, 50, 60)



 *Slice firom index 1 to 3*

print(numbers[1:4])  *Output: (20, 30, 40)*

 *Slice firom start to index 2*

print(numbers[:3])  *Output: (10, 20, 30)*

 *Slice all alternate elements*

print(numbers[0::2])  *Output: (10, 30, 50)*

 *Slice with negative indices*

print(numbers[-4:-1])  *Output: (30, 40, 50)*

 *Reverse list*

print(numbers[::-1])  *Output: (60,50,40,30,20,10)*

**Tuple Operations**

1. **Concatenation**

 **You can join two or more tuples using the + operator.**

tuple1 = (1, 2, 3)

tuple2 = (4, 5)

combined = tuple1 + tuple2

print(combined)  *Output: (1, 2, 3, 4, 5)*

### Repetition

 **You can repeat a tuple multiple times using the \* operator.**

tuple3 = ("hello",) \* 3

print(tuple3)  *Output: ('hello', 'hello', 'hello’)*

### Checking for an Item

 **Use the in keyword to check if an item exists in a tuple.**

numbers = (10, 20, 30, 40)

print(20 in numbers)  *Output: True*

### Iterating Over Tuple

Iterating allows you to traverse each element in a tuple, using loops.

**Example:** fruits = ("apple", "mango", "cherry")

 **Using for loop for** fruit in fruits:

print(fruit)

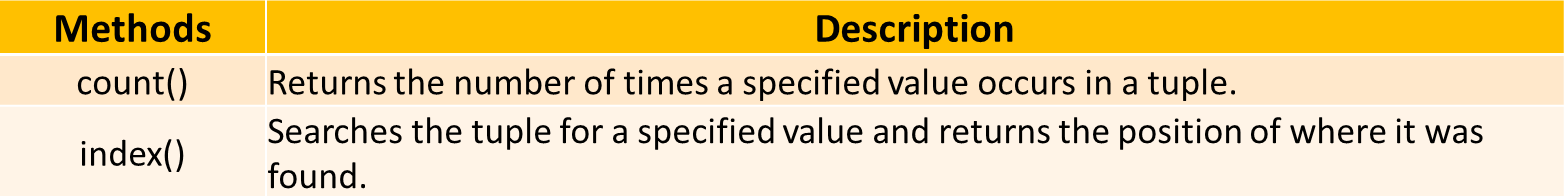
### Using while loop

i = 0

**while** i < len(fruits): print(fruits[i]) index += 1

### Tuple Methods

Python provides two **built-in methods** to use on tuples.



#### count

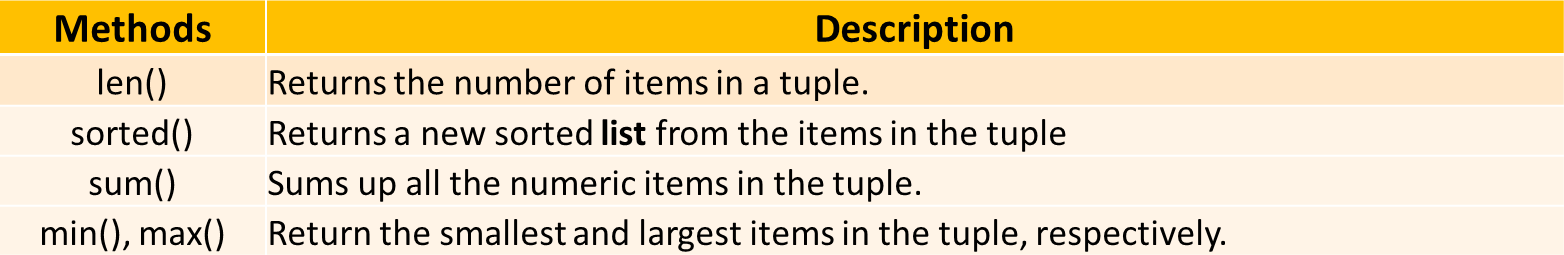
colors = ("red", "green", "blue", "green") print(colors**.count**("green"))  *Output: 2*

#### index

colors = ("red", "green", "blue", "green") print(colors**.index**("blue"))  *Output: 2*

### Tuple Functions

Python provides several **built-in functions** to use on tuples.



numbers = (2, 3, 1, 4)

print(**len**(numbers))  *Output: 4*

sorted\_num = **sorted**(numbers) print(sorted\_num)  *Output: [1,2,3,4]*

|  |  |  |
| --- | --- | --- |
| print(**sum**(numbers)) | *Output:* | *10* |
| print(**min**(numbers)) | *Output:* | *1* |
| print(**max**(numbers)) | *Output:* | *4* |

### Packing and Unpacking Tuples

* 1. **Packing** is the process of putting multiple values into a **single** tuple.

a = "Madhav"

b = 21

c = "Engineer"

pack\_tuple = a,b,c  *Packing values into a tuple*

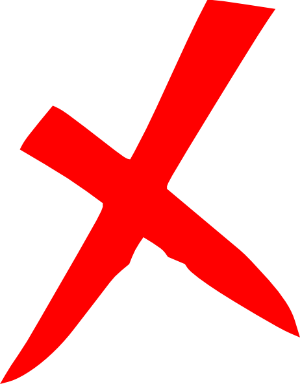
print(pack\_tuple)

* 1. **Unpacking** is extracting the values from a tuple into **separate** variables.

name, age, profession = person  *Unpacking a tuple*

|  |  |  |
| --- | --- | --- |
| print(name) | *Output:* | *adhav* |
| print(age) | *Output:* | *21* |
| print(profession) | *Output:* | *Engineer* |

### Modifying Tuple - Immutable

Once a tuple is created, you **cannot** modify its elements. This means you **cannot** add, remove, or change items.

 *Creating a tuple*

numbers = (1, 2, 3)

 *Attempting to change an item*

numbers[0] = 10

 *This will raise an error*

### Modifying Tuple

But there is a **trick**. You can **convert the tuple into a list, change the list, and convert the list back into a tuple.**

my\_tuple = ("apple", "mango", "cherry")

 *type cast tuple to list* y = **list**(my\_tuple) y.append("orange")

 *type cast back list to tuple* my\_tuple = **tuple**(y) print(my\_tuple)

### Tuple Use Case - Examples

**Storing Fixed Data (Immutable Data)**

Example: Storing geographic coordinates (latitude, longitude) or RGB color values, where the values shouldn’t be changed after assignment.

coordinates = (40.7128, -74.0060)  *Latitude and longitude fior NYC*

rgb\_color = (255, 0, 0)  *RfiB value fior red*

### Using Tuples as Keys in Dictionaries

Since tuples are immutable and hashable, they can be used as keys in dictionaries, unlike lists.

location\_data = {

(40.7128, -74.0060): "ew York City",

(34.0522, -118.2437): "Los Angeles"

}

print(location\_data[(40.7128, -74.0060)])  *Output: New York City*

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 18**

**Set in Python**

* What is a Set
* Create Sets
* Set Operations
* Set Methods
* Set Iteration
* Set Comprehensions

### Set in Python

A **set** is a collection of **unique items** in Python. Sets **do not** allow **duplicate** items and do not maintain any particular order so it **can’t be indexed**.

### Characteristics of Sets:

* **Unordered:** Elements have no defined order. You cannot access elements by index.
* **Unique Elements:** No duplicates allowed. Each element must be distinct.
* **Mutable:** You can add or remove elements after creation.
* **Immutable Elements:** individual elements inside a set cannot be modified/replaced

**Example:**

vowels = {'a', 'e', 'i', 'o', 'u'}

### Create Set in Python

There are two primary ways to create a set in Python:

### Using Curly Braces {}

my\_set = {1, 2, 3, 4, 5}

print(my\_set)  *Output: {1, 2, 3, 4, 5}*

### Using the set() Constructor

my\_set = set([1, 2, 3, 4, 5])

print(my\_set)  *Output: {1, 2, 3, 4, 5}*

**Note:** An empty set cannot be created using {} as it creates an empty dictionary. Use set() instead.

empty\_set = set() print(empty\_set)  *Output: set()*

### Set Operations

1. **Adding Elements :** Use the add() method to add a single element to a set.

fruits = {'apple', 'banana'} fruits**.add**('cherry')

print(fruits)  *Output: {'apple', 'banana', 'cherry’}*

1. **Removing Elements:** Use the remove() or discard() methods to remove elements.
   * **remove()** raises an error if the element is not found.
   * **discard()** does not raise an error if the element is missing.

fruits = {'apple', 'banana', 'cherry'}

### Using remove()

fruits**.remove**('banana')

print(fruits)  *Output: {'apple', 'cherry'}*

### Using discard()

|  |  |  |
| --- | --- | --- |
| fruits**.discard**('orange') | *No error even ifi 'orange' is* | *not* |
| *in the set* |  |  |
| print(fruits)  *Output:* | *{'apple', 'cherry’}* |  |

**Set Methods**

1. **Union:** Combines elements from two sets, removing duplicates.

set\_a = {1, 2, 3}

set\_b = {3, 4, 5}

union\_set = set\_a**.union**(set\_b) print(union\_set)  *Output: {1, 2, 3, 4, 5}*

**Alternative Syntax:** union\_set = set\_a **|** set\_b

1. **Intersection:** Includes only elements present in both sets.

set\_a = {1, 2, 3}

set\_b = {2, 3, 4}

intersection\_set = set\_a**.intersection**(set\_b) print(intersection\_set)  *Output: {2, 3}*

**Alternative Syntax:** intersection\_set = set\_a **fi** set\_b

1. **Difference:** Elements present in the first set but not in the second.

set\_a = {1, 2, 3, 4}

set\_b = {3, 4, 5}

difference\_set = set\_a**.difference**(set\_b) print(difference\_set)  *Output: {1, 2}*

**Alternative Syntax:** difference\_set = set\_a **-** set\_b

1. **Symmetric Difference:** Elements in either set, but not in both.

set\_a = {1, 2, 3}

set\_b = {3, 4, 5}

sym\_diff\_set = set\_a**.symmetric\_difference**(set\_b) print(sym\_diff\_set)  *Output: {1, 2, 4, 5}*

**Alternative Syntax:** sym\_diff\_set = set\_a **^** set\_b

### Set Iterations – Loop

You can use a **for loop** to go through each element in a set.

 **Using for loop -** Printing each number from a set

numbers = {1, 2, 3, 4, 5}

**for** number in numbers: print(number)

 **Using while loop -** first convert **set** to a **list** then use while loop because sets do not support indexing.

### Set Comprehension

Set comprehensions allow concise and readable creation of sets. Similar to list comprehensions but for sets.

### Syntax:

new\_set = {expression **for** item **in** iterable **if** condition}

### Example:

squares = {x\*\*2 for x in range(1, 6)} print(squares)  *Output: {1, 4, 9, 16, 25}*

### Set Common Use Cases

* **Removing Duplicates:** Easily eliminate duplicate entries from data.
* **Membership Testing:** Quickly check if an item exists in a collection.
* **Set Operations:** Perform mathematical operations like union, intersection, and difference.
* **Data Analysis:** Useful in scenarios requiring unique items, such as tags, categories, or unique identifiers.

 **Example:** Removing Duplicates from a List numbers = [1, 2, 2, 3, 4, 4, 5] unique\_numbers = **set**(numbers)

print(unique\_numbers)  *Output: {1, 2, 3, 4, 5}*

*[](https://www.youtube.com/%40RishabhMishraOfficial)*

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 19**

**Dictionary in Python**

* What is a Dictionary
* Create Dictionary
* Access Dictionary Values
* Dictionary Methods
* Dictionary – Add, Modify & Remove Items
* Dictionary Iteration
* Nested Dictionary
* Dictionary Comprehensions

## Dictionary in Python

A **dictionary** is a data structure in Python that stores data in **key-value pairs**. Dictionary items (key – value pair) are ordered, changeable, and do not allow duplicates.

* **Key:** Must be unique and immutable (strings, numbers, or tuples).
* **Value:** Can be any data type and does not need to be unique.

**Example:** *Simple dictionary with three key-value pairs*

student = **{**

1**:** "Class-X",

"name"**:** "Madhav", "age"**:** 20

**}**

### Create Dictionary in Python

**Method-1:** We create a dictionary using **curly braces {}** and separating keys and values with a **colon**.

### Syntax

my\_dict =

{"key1": "value1", "key2": "value2", "key3": "value3", **…**}

### Empty dictionary

empty\_dict = {}

### Dictionary with data

cohort = **{**

"course"**:** "flython"**,** "instructor"**:** "Rishabh Mishra"**,** "level"**:** "Beginner"

**}**

### Method-2: Using dict() constructor

Pass key-value pairs as keyword arguments to dict()

person = **dict**(name="Madhav", age=20, city="Mathura") print(person)

 *Output: {'name': 'adhav', 'age’: 20, 'city':*

*'athura'}*

### Method-3: Using a List of Tuples

Pass a list of tuples, where each tuple contains a key-value pair.

student = **dict**([("name", "Madhav"), ("age", 20), ("grade", "A")])

print(student)

 *Output: {'name': 'adhav', 'age’: 20, 'grade': 'A'}*

### Access Dictionary Values

Access dictionary values by using the **key** name inside **square brackets**.

**Example:**

student = **{**

1**:** "Class-X",

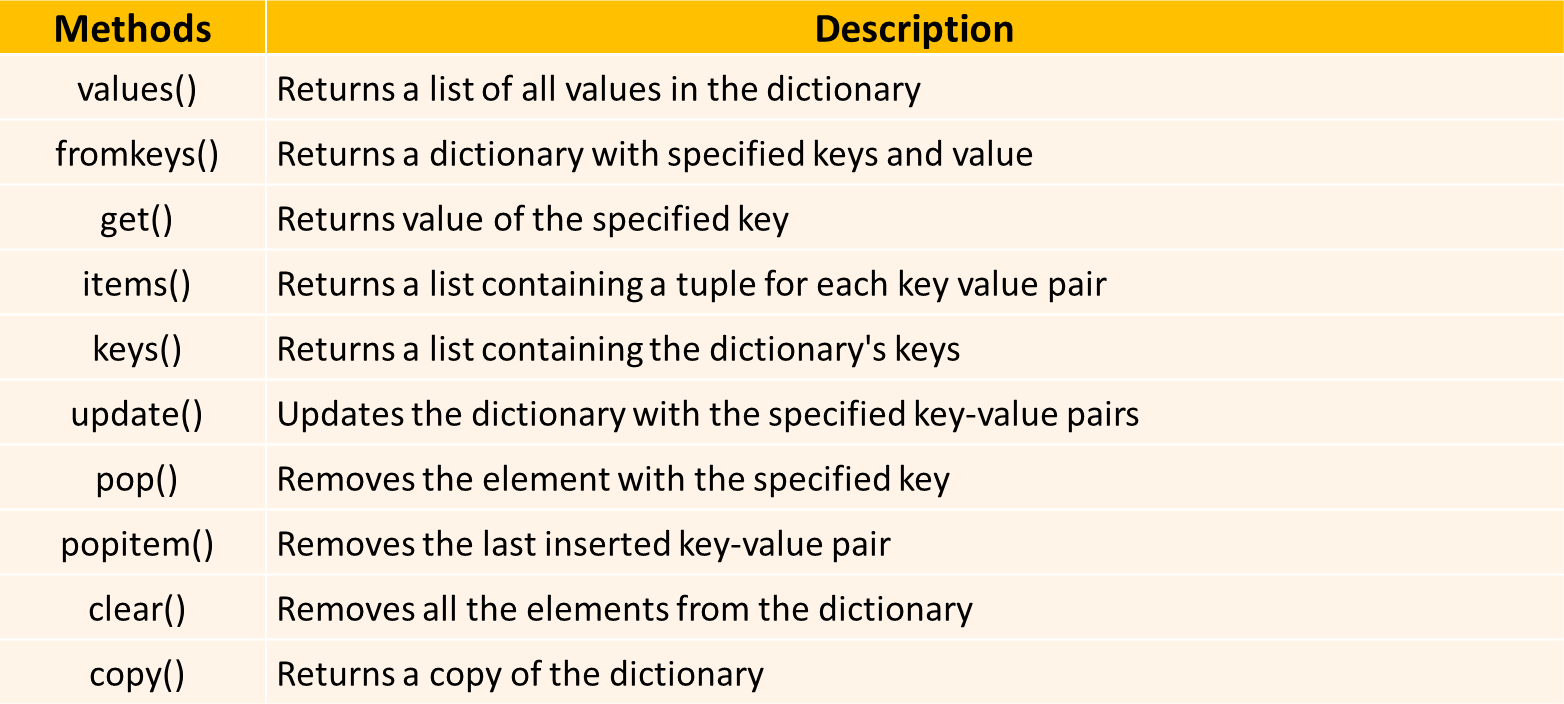
"name"**:** "Madhav", "age"**:** 20

**}**

 *print value based on respective key-names* print(student["name"])  *Output: adhav* print(student["age"])  *Output: 20*

### Dictionary Methods

Python provides several **built-in methods** to use on dictionary.



Here are a few useful methods:

* + **.keys():** Returns all keys in the dictionary.
  + **.values():** Returns all values in the dictionary.
  + **.items():** Returns all key-value pairs.
  + **.get():** Returns value for a key (with an optional default if key is missing).

|  |  |  |
| --- | --- | --- |
| **Examples**  print(student.**keys**()) | *All* | *keys* |
| print(student.**values**()) | *All* | *values* |
| print(student.**items**()) | *All* | *key-value pairs* |

print(student.**get**("name"))  *Safie way to access a value*

### Dictionary – Add, Modify & Remove Items

1. **Add** or **Modify Item:** Use **assign-operator '='** to add/modify items in a dictionary.

*# Adding a new key-value pair*

student["email"] **=** "[madhav@example.com](mailto:madhav@example.com)"

*# Modifying an existing value*

student["age"] **=** 25

1. **Remove Item:** Use **del** or **.pop()** to remove items from a dictionary.

*# Remove with del*

**del** student["age"]

*# Remove with pop() and store the removed value*

email = student**.pop**("email")

print(email)  *Output:* [*madhav@example.com*](mailto:madhav@example.com)

### Dictionary Iterations

A dictionary can be iterated using **for loop**. We can loop through dictionaries by keys, values, or both.

 *Loop through keys*

for key in student: print(key)

 *Loop through values*

for value in student: print(student[value])

 *Loop through values: using values() method*

for value in student**.values()**: print(value)

 *Loop through both keys and values*

for key, value in student**.items()**: print(key, value)

### Nested Dictionary

Dictionaries can contain other dictionaries, which is useful for storing more complex data.

*# nested dictionaries*

students = {

"student1": {

"name": "Madhav", "age": 20,

"grade": "A"

},

"student2": {

"name": "Keshav", "age": 21,

"grade": "B"

}

}

print(students["student1"]["name"])  *Output: adhav*

### Dictionary Comprehension

A dictionary comprehension allows you to create dictionaries in a concise way.

### Syntax:

new\_dict =

{key\_expression**:** value\_expression **for** item **in** iterable **if**

condition}

|  |  |  |
| --- | --- | --- |
| **Example: Creating** | **a dictionary with** | **square numbers** |
| squares = {x: x \* x | for x in range(1, | 6)} |
| print(squares) |  |  |

 *Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}*

### Dictionary Common Use Cases

* + **User Profiles in Web Applications:** Store user details like name, email, etc.
  + **Product Inventory Management:** Keep track of stock levels for products in an e- commerce system.
  + **API Responses**: Parse JSON data returned from APIs (e.g., weather data).
  + **Grouping Data:** Organize data into categories. Example: grouped = {"fruits": ["apple", "banana"], "veggies": ["carrot"]}
  + **Caching:** Store computed results to reuse and improve performance. Example: cache = {"factorial\_5": 120}
  + **Switch/Lookup Tables:** Simulate switch-case for decision-making.

### Example:

actions = {"start": start\_fn, "stop": stop\_fn} actions["start"]()

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 20**

**OOPs in Python**

* What is OOPs
* Why OOP is required
* Class and Object
* Attributes and Methods
* init Method (Constructor)
* Abstraction
* Encapsulation
* Inheritance
* Polymorphism

## OOPs in Python

### Two ways of programming in Python:

1. **Procedural** Programming,

### OOPs

**OOPs: Object Oriented Programming**

A way of organizing code by creating "blueprints" (called **classes**) to represent real-world things like a student, car, or house. These blueprints help you create **objects** (individual examples of those things) and define their behavior.

**Class:** A class is a blueprint or template for creating objects.

It defines the properties (**attributes**) & actions/behaviors (**methods**) that objects of this type will have.

**Object:** An object is a specific instance of a class.

It has actual data based on the blueprint defined by the class.

**OOPs Example in Python *Example***: **Constructing a building Class:** Blueprint for a floor.

**Object:** Actual house built from the blueprint. Each house (object) can have different features, like paint color or size, but follows the same blueprint.



### Why OOPs?

* **Models Real-World Problems:**

Mimics real-world entities for easier understanding.

### Code Reusability:

Encourages reusable, modular, and organized code.

### Easier Maintenance:

OOP organizes code into small, manageable parts (classes and objects). Changes in one part don’t impact others, making it easier to maintain.

### Encapsulation:

Encapsulation **protects data** integrity and **privacy** by bundling data and methods within objects.

### Flexibility & Scalability:

OOP makes it easier to add new features without affecting existing code.

### OOPs – Question

**Write a Python program to:**

1. Define a Student class with attributes name, grade, percentage, and team.
   * Include an init method to initialize these attributes.
   * Add a method student\_details that prints the student’s details in the format: "<name> is in <grade> grade with <percentage>%, from team <team>".
2. Create two teams (team1 and team2) as string variables.
3. Create at least two student objects, each belonging to one of the teams.
4. Call the student\_details method for each student to display their details.

### Class and Object Example

**Class:** A class is a blueprint or template for creating objects.

**Object:** An object is a specific instance of a class.

*Example*

**class** Student**: pass**

 *Create an object* student1 = Student() print(type(student1))

 *Output:* <class ' main .Student'>

### Attributes and Methods

**Attributes**: Variables that hold data about the object.

**Methods**: Functions defined inside a class that describe its behavior.

*Example*

**class** Student**:**

**def**  init (self, name, grade)**:** self.name = name  ***Attribute*** self.grade = grade  ***Attribute***

**def** get\_grade(self)**: ** ***ethod***

return f"{self.name} is in grade {self.grade}."

 *Object creation*

student1 = Student("Madhav", 10)

print(student1.get\_grade())  *Output:* Madhav is in grade 10.

### The \_\_init\_\_ Method (Constructor)

Whenever we create/construct an object of a class, there is an inbuilt method **init**

which is automatically called to **initialize** attributes.

The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

*Example*

**class** Student:

**def init** (self, name, grade): self.name = name

self.grade = grade

 *Initialize object with attributes* student1 = Student("Madhav", 10) print(student1.name)  *Output*: Madhav

**Abstraction in Python:** Hiding unnecessary details

Abstraction hides implementation details and shows only the relevant functionality to the user.

*Example*

**class** Student:

**def**  init (self, name, grade, percentage): self.name = name

self.grade = grade self.percentage = percentage

**def** is\_honors(self):  *Abstracting the logic*

return self.percentage > 90  *Logic hidden * *Abstract method in use*

student1 = Student("Madhav", 10, 98) print(student1.is\_honors())  *Output: True*

**Encapsulation in Python:** Restricting direct access to attributes & methods

Encapsulation restricts access to certain attributes or methods to protect the data and enforce controlled access.

*Example*

**class** Student:

**def**  init (self, name, grade, percentage): self.name = name

self.grade = grade

self. percentage = percentage  flrivate attribute (hidden)

**def** get\_percentage(self):  *Public method to access the private attribute*

return self. percentage

 *Creating a student object*

student1 = Student("Madhav", 10, 98)

 *Accessing the private attribute using the public method*

print(f"{student1.name}'s percentage is

{student1.get\_percentage()}%.") print(student1. percentage)  *error*

**Inheritance in Python:** Reusing Parent’s prop & methods

Inheritance (parent-child), allows one class (child) to reuse the properties and methods of another class (parent). This avoids duplication and helps in code reuse.

*Example*

**class** Student:

**def**  init (self, name, grade, percentage): self.name = name

self.grade = grade self.percentage = percentage

**def** student\_details(self):  method

print(f'{self.name} is in {self.grade} grade with

{self.percentage}%')

**class** GraduateStudent(Student):  *firaduateStudent inherits firom Student*

**def** init (self, name, grade, percentage, stream):

**super()**. init (name, grade, percentage)  Call parent class initializer

self.stream = stream  *New attribute specifiic to firaduateStudent*

**def** student\_details(self): **super()**.student\_details() print(f"Stream: {self.stream}")

 *Create a graduate student*

grad\_student = GraduateStudent("Vishakha", 12, 94, "flCM")

 *Vishakha is in 12 grade with 94%*

grad\_student.student\_details()  *Stream: PC*

**Polymorphism in Python:** Same method but different output

Polymorphism allows methods in different classes to have the same name but behave differently depending on the object.

*Example*

**class** GraduateStudent(Student):

**def** student\_details(self):  *Same method as in parent class*

print(f"{self.name} is a graduate student from final

year.")

 *Polymorphism in action*

student1 = Student("Madhav", 10, 98)

grad\_student = GraduateStudent("Sudevi", 12, 99, "flCM") student1**.**student\_details()

 *Output: adhav is in 10 grade with 98%*

grad\_student**.**student\_details()

 *Output: Sudevi is a graduate student firom fiinal year.*

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 21**

**Modules, Packages & Libraries in Python**

* What is Module
* Create & Use a Module
* What is Package
* What is Library
* Python pip
* Most used Libraries

## Modules in Python

A module is a single Python file (.py) containing Python code. It can include functions, classes, and variables that you can reuse in other programs.

### Why use modules?

* To organize code into smaller, manageable chunks.
* To reuse code across multiple programs.

#### # Create a module:

* Save the following as mymodule.py

def say\_hello(name):

return print(f"Hello, {name}!")

#### # Use the module:

**import** mymodule greetings.say\_hello("Madhav") *# Output: Hello, Madhav!*

## Packages in Python

A package is a **collection** of modules organized in **directories** (folders) with an init .py file. It allows you to structure your Python projects logically.

### Why use packages?

* To group related modules together.
* To create larger applications or libraries.

#### # Structure Example:

my\_package/

init .py math\_utils.py string\_utils.py

#### # Use the package:

Syntax: **from** my\_package **import** <package\_name>

Example: **from** my\_package **import** math\_utils, string\_utils

## Libraries in Python

A library is a collection of modules and packages that provide pre-written functionality for your program. Libraries are typically larger and more feature-rich than packages or modules.

### Why use libraries?

To avoid writing common functionality from scratch.

To leverage powerful tools developed by the community.

**Example**: Python has many popular libraries, such as:

* Pandas: For data manipulation.
* Matplotlib: For plotting and visualization.

#### # Using a library (Pandas):

**import** pandas as pd

## Python PIP

pip stands for "Pip Installs Packages". It is the package manager for Python that allows you to install, update, and manage Python libraries (packages) from the Python Package Index (PyPI).

*Think of pip as an app store for Python libraries. You use it to search, install, and manage Python tools, just like downloading apps on your phone.*

When you use pip install <package\_name>, it:

* Connects to PyPI (Python Package Index) online.
* Downloads the specified library or package.
* Installs it into your Python environment.

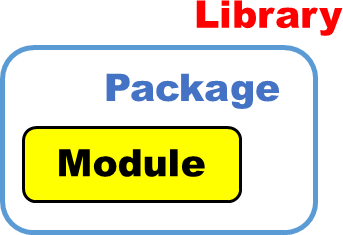
To install packages, we use: **pip install** *<library\_name>*

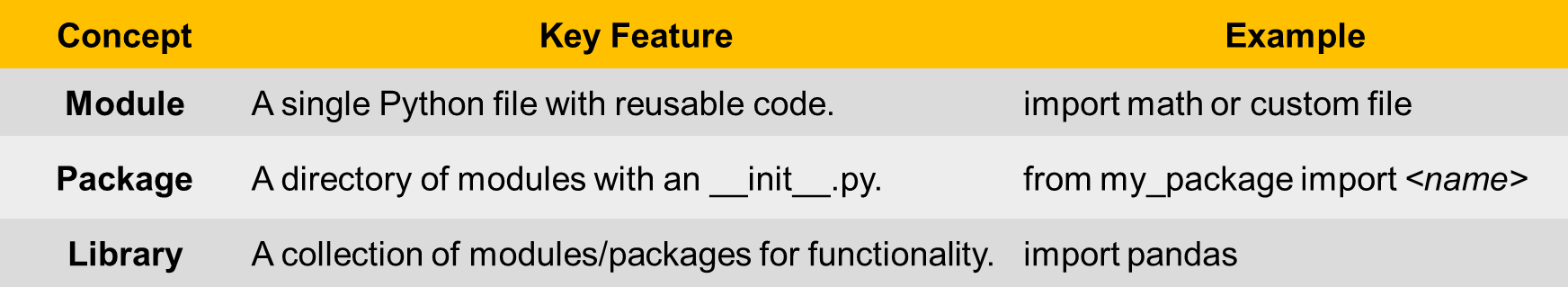
**Example**: installing pandas to work on dataframe:

pip install pandas

## Summary: Module, Package and Library

* **Module:** A single page.
* **Package:** A book containing multiple pages.
* **Library:** A book store with many books.





## Most Used Python Libraries

### Data Analytics, data visualization and ML

**Application Library Description Install Command**

**Data Analytics**

**Pandas** Data manipulation and analysis.

**NumPy** Numerical computing with array support.

**SciPy** Scientific computing and technical computing.

**Statsmodels** Statistical modeling and

testing.

**Dask** Parallel computing for large datasets.

pip install pandas pip install numpy pip install scipy

pip install statsmodels pip install dask

**Data Visualization**

**Matplotlib** Basic plotting and visualization. pip install matplotlib

**Seaborn** Statistical data visualization. pip install seaborn

**Plotly** Interactive graphs and dashboards.

**Scikit-learn** Classic machine learning

algorithms.

pip install plotly

pip install scikit-learn

**Machine Learning**

**TensorFlow** Deep learning and ML models. pip install tensorflow

Deep learning with dynamic

**fi Deep Learning**

**PyTorch**

pip install torch torchvision

computation.

**Keras** High-level deep learning API. pip install keras

**XGBoost** Gradient boosting for structured data.

pip install xgboost

### Web Scraping, web development and game development

|  |  |  |  |
| --- | --- | --- | --- |
| **Application** | **Library** | **Description** | **Install Command** |
|  | **BeautifulSoup** | Parsing HTML and XML for data pip install beautifulsoup4  extraction. | |
|  | **Scrapy** | Advanced web scraping framework. | pip install scrapy |
| **Web Scraping** | **Selenium** | Browser automation for scraping dynamic sites. | pip install selenium |
|  | **Requests** | HTTP library for fetching web pages. | pip install requests |
|  | **Lxml** | Fast XML and HTML parsing. | pip install lxml |
|  | **Django** | Full-stack web framework. | pip install django |
| **Web Development** | **Flask** | Lightweight web framework. | pip install flask |
|  | **FastAPI**  **Pygame** | High-performance API framework.  Game development library. | pip install fastapi  pip install pygame |
|  |
| **Game Development** | **Arcade** | Advanced 2D game development library. | pip install arcade |
|  | **Panda3D** | Real-time 3D rendering and game creation. | pip install panda3d |

**[](https://www.youtube.com/%40RishabhMishraOfficial)**

# PYTHON TUTORIAL FOR BEGINNERS



**Chapter - 22**

**File Handling in Python**

* What is File Handling
* Open & Read a File
* Write & Append to a File
* Create a File
* Close a file
* Work on txt, csv, excel, pdf files

## File handling in Python

File handling in Python allows you to read from and write to files. This is important when you want to store data permanently or work with large datasets.

Python provides built-in functions and methods to interact with files.

### Steps for File Handling in Python:

* Opening a file
* Reading from a file
* Writing to a file
* Closing the file

### Open a File

To perform any operation (read/write) on a file, you first need to open the file using Python’s open() function.

**Syntax:** file\_object = open('filename', 'mode’)

* ***'filename***': Name of the file (can be relative or absolute path).
* ***'mode***': Mode in which the file is opened (read, write, append, etc.).

### File Modes:

* + 'r': Read (default mode). Opens the file for reading.
  + 'w': Write. Opens the file for writing (if file doesn’t exist, it creates one).
  + 'a': Append. Opens the file for appending (if file doesn’t exist, it creates one).
  + 'rb'/'wb': Read/Write in binary mode.

**Example:** Opening a file for reading

file = **opvt**('example.txt', '**u**')

### Read from a File

Once a file is open, you can read from it using the following methods:

* + read(): Reads the entire content of the file.
  + readline(): Reads one line from the file at a time.
  + readlines(): Reads all lines into a list.

***# Example:*** *Reading the entire file*

file = **opvt**('example.txt', '**u**') content = file**.uvaa()** print(content)

file.close()

***# Example:*** *Reading one line at a time*

file = **opvt**('example.txt', '**u**') line = file**.uvaalitv()** print(line)

file.close()

### Write to a File

To write to a file, you can use the write() or writelines() method:

* + write(): Writes a string to the file.
  + writelines(): Writes a list of strings.

***# Example:*** *Writing to a file (overwrites existing content)* file = **opvt**('example.txt', '**w**') file.**wuitv**("Hello, world!") file.close()

***# Example:*** *Appending to a file (add line to the end)*

file = **opvt**('example.txt', '**a**') file.**wuitv**("\nThis is an appended line.") file.close()

#### # Close a file:

file.**closv()**

### Close a File

Instead of manually opening and closing a file, you can use the **with** statement, which automatically handles closing the file when the block of code is done.

***# Example:*** *Reading with with statement*

**with opvt**('example.txt', '**u**') as file: content = file.**uvaa**() print(content)

*In this case, you don’t need to call file.close(), because Python automatically closes the file when the block is finished.*

***# Example:*** *Using* ***exception handling*** *to close a file*

### tuQ:

**opvt**('example.txt', '**u**') as file: content = file.**uvaa**() print(content)

### fitallQ:

file.**closv**()

### Working with Difft Format Files

***# csv -*** *Using csv module*

import **csv**

file = open('file.csv', mode='r') reader = **csv**.**uvaavu**(file)

***# csv -*** *Using pandas library*

import **pataas** as pd

df = **pa**.**uvaa\_csv**('file.csv')

***# excel -*** *Using pandas library*

import **pataas** as pd

df = **pa**.**uvaa\_vxcvl**('file.xlsx')

***# PDF*** *Using PyPDF2 library****:***

import **flQflKF?**

file = **opvt**('file.pdf', 'b') pdf\_reader = **flQflKF?**.**flafRvaavu**(file)

[](https://www.youtube.com/%40RishabhMishraOfficial)