

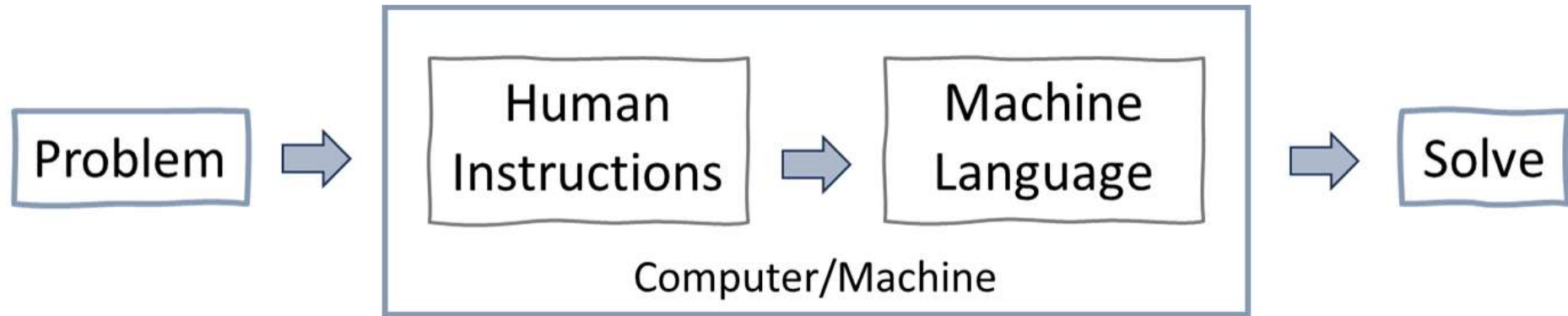
Python

Syllabus

- **Python Programming:**
- Python Interpreter and Interactive Mode-Data types-Statements- Expressions-Boolean Values and Operators-Strings-Arrays of Numbers- Lists-Tuples-Dictionaries-Functions-File Reading and Writing

Programming Language

- 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.



Python

- **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.

Applications



Web Development

Web Developer
Frontend Developer
Backend Developer
Python Developer



Software Development

Game Developer
Software Developer
DevOps Engineer
Software Testing & more



Data Science & Analytics

Data Analyst
Data Scientist
Machine Learning Engineer
Artificial intelligence Engineer

Python Interpreter and Interactive Mode

Python Interpreter

Python is an interpreted language developed by Guido van Rossum in the year of 1991.

As we all know Python is one of the most high-level languages used today because of its massive versatility and portable library & framework features.

It is an interpreted language because it executes line-by-line instructions.

- **Interpreted Languages:** Perl, BASIC, Python, JavaScript, Ruby, PHP.

Compiled Languages: C, C++, C#, COBOL and CLEO.

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 XC013 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.

Interactive Mode

- Interactive mode is a command line shell which gives immediate feedback for each statement, while running previously fed statements in active memory.

Start an interactive Python session

1. Select a device in Remote Manager that is configured to allow shell access to the admin user, and click Actions > Open Console. ...
2. At the shell prompt, use the python command with no parameters to enter an interactive Python session: ...
3. Type Python commands at the Python prompt.

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/P: `<class 'str'>`
 - 2. `value = 101`
 - `>>> type(value)`
 - O/P: `<class 'int'>`

Basic Data Types in Python

- Python can be used as a powerful calculator for performing a wide range of
- arithmetic operations.
- **1. Numeric: Integer, Float, Complex**
- **2. Sequence: String, List, Tuple**
- **3. Dictionary**
- **4. Set**
- **5. Boolean**
- **6. Binary: Bytes, bytearray, Memoryview**

Numeric

Int	float	complex
26, 101, 108	0.5, 1.5, 101.2	complex(3, 5) -> 3+5i

Sequence

str	list	tuple
"Madhav", "Rishabh", "26"	[1, 26, 101, 108]	(1, 26, 101, 108)

Dictionary

dict
{'name': 'Madhav', 'age': 26, 'city': 'Prayagraj'}

Sets

set
{1, 26, 101, 108}, {"abc", 34, True, 40, "male"}

Boolean

bool
True, False

Numeric1 The data that has a numeric value int: positive or negative whole numbers float: real number with a decimal point complex: number contains an arranged pair, $x + iy$, (real part) + (imaginary part)i	Sequence2 Collection of similar or different Python data types string: Collection of one or more characters put in a quote/quotes. list: can store objects of multiple data type, and encased inside square sections []. Mutable. tuple: store objects of multiple data type, and encased inside circle sections (). Immutable.	Dictionary3 A key-value pair set arranged in any order. Sets4 Unordered collection of elements. It is iterable, mutable, and has no duplicate elements Boolean5 True and False are the two default values	Binary6 Used to work with binary data. Such as image, audio, mp3 files, etc. bytes: immutable sequence type to represent sequences of bytes (8-bit values). bytearray: similar to bytes but can be modified . Memoryview: Used to create a “view” of memory containing binary data.
---	---	---	---

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** - $2+3j$

```
1  a = 5
2  print(type(a))
3
4  b = 5.0
5  print(type(b))
6
7  c = 2 + 4j
8  print(type(c))
```

Output

```
<class 'int'>
<class 'float'>
<class 'complex'>
<class 'float'>
<class 'complex'>
```

Sequence Data Types in Python

The sequence Data Type in Python is the ordered collection of similar or different Python data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence data types of Python:

- [Python String](#)
- [Python List](#)
- [Python Tuple](#)

String Data Type

Python Strings are arrays of bytes representing Unicode characters. In Python, there is no character data type Python, a character is a string of length one. It is represented by str class.

- Strings in Python can be created using single quotes, double quotes or even triple quotes. We can access individual characters of a String using index.

```
1 s = 'Welcome to the Geeks World'
2 print(s)
3
4 # check data type
5 print(type(s))
6
7 # access string with index
8 print(s[1])
9 print(s[2])
10 print(s[-1])
```

Output

```
Welcome to the Geeks World
<class 'str'>
e
l
d
```

List Data Type

- Lists are just like arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.

```
1  # Empty list
2  a = []
3
4  # list with int values
5  a = [1, 2, 3]
6  print(a)
7
8  # list with mixed int and string
9  b = ["Geeks", "For", "Geeks", 4, 5]
10 print(b)
```

Output

```
[1, 2, 3]
['Geeks', 'For', 'Geeks', 4, 5]
```

Tuple Data Type

Just like a list, a tuple is also an ordered collection of Python objects. The only difference between a tuple and a list is that tuples are immutable. Tuples cannot be modified after it is created.

Creating a Tuple in Python

- In Python Data Types, tuples are created by placing a sequence of values separated by a 'comma' with or without the use of parentheses for grouping the data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, lists, etc.).

```
1  # initiate empty tuple
2  tup1 = ()
3
4  tup2 = ('Geeks', 'For')
5  print("\nTuple with the use of String: ", tup2)
```

Output

```
Tuple with the use of String: ('Geeks', 'For')
```

Boolean Data Type in Python

- Python Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are `truthy` (true), and those equal to False are `falsy` (false). However non-Boolean objects can be evaluated in a Boolean context as well and determined to be true or false. It is denoted by the class `bool`.

```
1 print(type(True))
2 print(type(False))
3 print(type(true))
```

Output:

```
<class 'bool'>
<class 'bool'>
```

```
Traceback (most recent call last):
```

```
File "/home/7e8862763fb66153d70824099d4f5
```

```
print(type(true))
```

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

Set Data Type in Python

In Python Data Types, [Set](#) is an unordered collection of data types that is iterable, mutable, and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

Create a Set in Python

Sets can be created by using the built-in `set()` function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a '**comma**'. The type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

```
1  # initializing empty set
2  s1 = set()
3
4  s1 = set("GeeksForGeeks")
5  print("Set with the use of String: ", s1)
6
7  s2 = set(["Geeks", "For", "Geeks"])
8  print("Set with the use of List: ", s2)
```

Output

```
Set with the use of String: {'s', 'o', 'F', 'G', 'e', 'k', 'r'}
Set with the use of List: {'Geeks', 'For'}
```

Dictionary Data Type

- A dictionary in Python is a collection of data values, used to store data values like a map, unlike other Python Data Types that hold only a single value as an element, a Dictionary holds a key: value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon :, whereas each key is separated by a 'comma'.

```
1  # initialize empty dictionary
2  d = {}
3
4
5  d = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
6  print(d)
7
8  # creating dictionary using dict() constructor
9  d1 = dict({1: 'Geeks', 2: 'For', 3: 'Geeks'})
10 print(d1)
```

Output

```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

Operators

- 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

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

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

Operator	Name	Example (a = 5, b = 3)
+	Addition	a + b # o/p: 8
-	Substraction	a – b # o/p: 2
*	Multiplication	a * b # o/p: 15
/	Division	a / b # o/p: 1.6666
%	Modulus (returns remainder)	a % b # o/p: 2
//	Floor division	a // b # o/p: 1
**	Exponentiation	a ** b # o/p: 125

- Precedence of **Arithmetic Operators** in Python:
- P – Parentheses
- E – Exponentiation
- M – Multiplication
- D – Division
- A – Addition
- S – Subtraction

2. Comparison (Relational) Operators

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

Operator	Name	Example
==	Equal	a == b
!=	Not equal	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

3. Assignment Operators

Assignment operators are used to assign values to variables.

Operator	Eaample	Also written As
=	a = 5	a = 5
+=	a += 3	a = a + 3
-=	a -= 3	a = a - 3
*=	a *= 3	a = a * 3
/=	a /= 3	a = a / 3
%=	a %= 3	a = a % 3
//=	a //= 3	a = a // 3
**=	a **= 3	a = a ** 3
&=	a &= 3	a = a & 3 ... etc

Logical Operators

Logical operators are used to combine conditional statements.

Operator	Decription	Eaample
and	Returns True if both statements are true	<code>a > 2 and a < 5</code>
or	Returns True if one of the statements is true	<code>a > 5 or a < 10</code>
not	Reverse the result, returns False if the result is true	<code>not(a > 2 and a < 10)</code>

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.

Tbpe	Operator	Eaample	Also written As
Identity	is	Returns True if both variables are the same object	a is b
Identity	is not	Returns True if both variables are not the same object	a is not b
Membership	in	Returns True if a sequence with the specified value is present in the object	a in b
Membership	not in	Returns True if a sequence with the specified value is not present in the object	a not in b

Bitwise Operators

- Bitwise operators perform operations on binary numbers.

Operator	Name	Description	Also written As
&	AND	Sets each bit to 1 if both bits are 1	a & b
	OR	Sets each bit to 1 if one of two bits is 1	a b
^	XOR	Sets each bit to 1 if only one of two bits is 1	a ^ b
~	NOT	Inverts all the bits	~a
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	a << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	a >> 2

- Eg:1
- a = 5 # 0101
- b = 3 # 0011
- print(a & b)
- # Output: 1 # 0001
- Eg:2
- a = 5 # 0101
- b = 8 # 1000
- print(a & b)
- # Output: 0 # 0000

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 of 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:

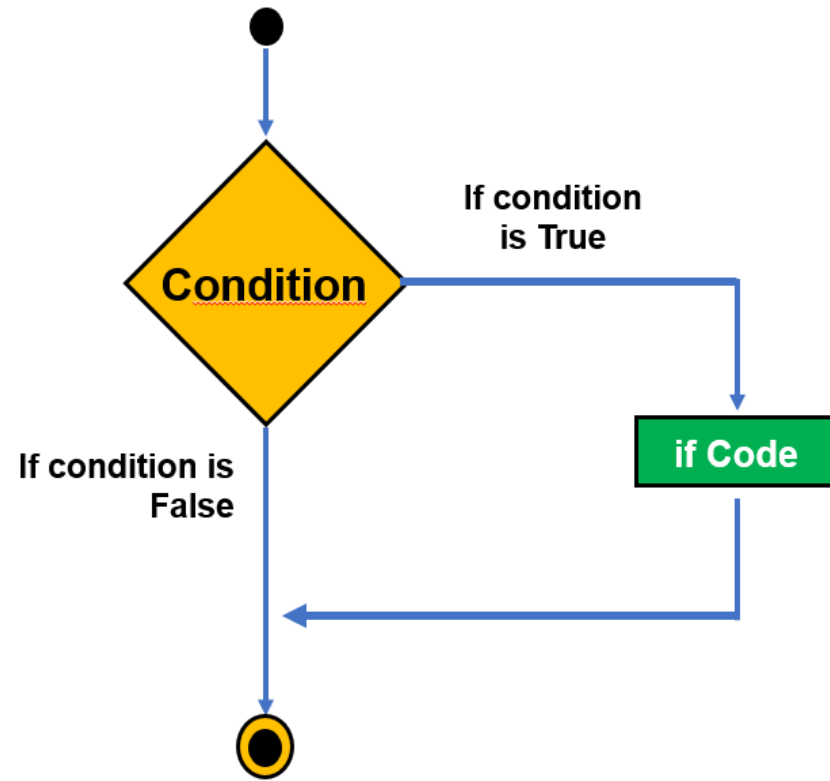
- # Code to execute if the condition is true

- **Example:**

- age = 26

- if age > 19:

- print("You are an adult")



'if-else' Conditional Statement

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

- **Syntax:**

- `if condition:`

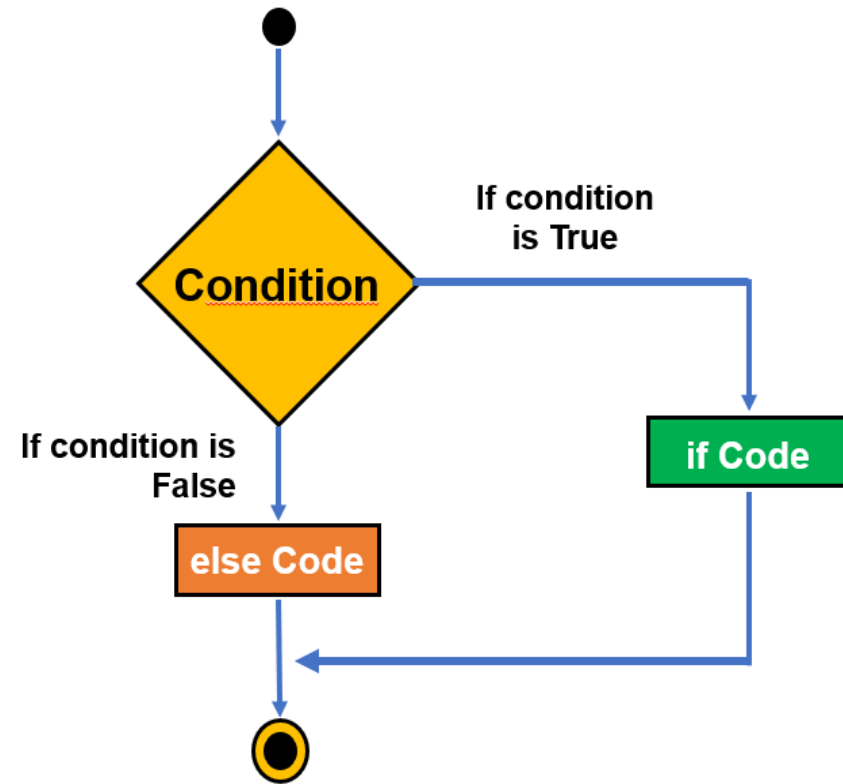
- `# Code to execute if the condition is true`

- `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-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")

Nested '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:`
- `# Code block for condition2 being False`
- `else:`
- `# Code block for condition1 being False`

- `number = 10`
- `if number > 0: # First check if 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.")`

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 :)

The diagram illustrates the components of a Python function definition. It shows the code: `def my_function(param):`, `instruction-1`, `instruction-2`, `...`, and `return result`. Hand-drawn orange arrows and brackets link parts of the code to their descriptions on the right. An arrow points from `def` to 'Keyword "def"'. Another points from `my_function` to 'Function name'. A bracket under `(param)` points to 'Parameters'. An arrow points from the colon `:` to 'Colon'. A bracket under the indented instructions points to 'Instructions'. Finally, an arrow points from `return` to 'Keyword "return", gives result'.

```
def my_function(param):  
    instruction-1  
    instruction-2  
    ...  
return result
```

Keyword "def"

Function name

Parenthesis

Parameters

Colon

Instructions

Keyword "return",
gives result

- *Create or Define Function*
- `def greetings():`
- `print("Welcome to Python tutorial by Rishabh")`
- *# Use or call this Function*
- `greetings()`
- *# Output: Welcome to Python tutorial by Rishabh*

- **Example:2**

- `# function to adds two numbers & 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 of a and b*

- `result = add(3, 5)`

- `print(result)`

Function with a Return value

- *# function to convert Celsius to Fahrenheit*
- `def celsius_to_fahrenheit(celsius):`
- `fahrenheit = (celsius * 9/5) + 32`
- `return Fahrenheit`
- *# Calling this function to return a value*
- `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 for now`

Methods	Description
len()	returns the length of a string (the number of characters).
upper()	Converts a string into upper case
lower()	Converts a string into lower case
strip()	Removes any leading and trailing whitespace (including spaces, tabs, or newline characters).
count()	Returns the number of times a specified value occurs in a string
find()	Searches the string for a specified value and returns the position of where it was found
title()	Converts the first character of each word to upper case
split()	Splits the string at the specified separator, and returns a list
replace(old, new)	Replaces all occurrences of a substring with a new substring

Methods	Description
<code>append()</code>	Adds a single element to the end of the list.
<code>extend()</code>	Adds multiple elements (from another iterable) to the end of the list.
<code>insert()</code>	Inserts an element at a specified position.
<code>remove()</code>	Removes the first occurrence of a specified element.
<code>pop()</code>	Removes and returns an element at a specified index. If no index is specified, it removes and returns the last element.
<code>clear()</code>	Removes all elements from the list, resulting in an empty list.
<code>index()</code>	Returns the index of the first occurrence of a specified element.
<code>count()</code>	Returns the number of occurrences of a specified element.
<code>sort()</code>	Sorts the list in ascending order by default. You can also sort in descending order and specify custom sorting criteria.
<code>reverse()</code>	Reverses the elements of the list in place.
<code>copy()</code>	Returns a shallow copy of the list.

Methods	Description
len()	Returns the number of items in a tuple.
sorted()	Returns a new sorted list from the items in the tuple
sum()	Sums up all the numeric items in the tuple.
min(), max()	Return the smallest and largest items in the tuple, respectively.

Methods	Description
values()	Returns a list of all values in the dictionary
fromkeys()	Returns a dictionary with specified keys and value
get()	Returns value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
update()	Updates the dictionary with the specified key-value pairs
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary

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 from 0 to 3*
- `count = 0`
- **while** `count < 4:` *# Condition*
- `print(count)`
- `count += 1`
- *# Output: 0 1 2 3*

- **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*
- `language = 'Python'`
- **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**

- **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

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

- **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 = open('example.txt', 'r')`

- **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 = open('example.txt', 'r')`

- `content = file.read()`

- `print(content)`

- `file.close()`

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

- `file = open('example.txt', 'r')`

- `line = file.readline()`

- `print(line)`

- `file.close()`

- **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.

- **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 = open('example.txt', 'r')`
- `content = file.read()`
- `print(content)`
- `file.close()`
- ***# Example:** Reading one line at a time*
- `file = open('example.txt', 'r')`
- `line = file.readline()`
- `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 = open('example.txt', 'w)`
- `file.write("Hello, world!")`
- `file.close()`

- **# Example:** *Appending to a file (add line to the end)*
- `file = open('example.txt', 'a')`
- `file.write("\nThis is an appended line.")`
- `file.close()`
- **# Close a file:**
- `file.close()`

- **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 open('example.txt', 'r') as file:**
- `content = file.read()`
- `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*
- **try:**
- **open**('example.txt', 'r') as file:
- content = file.**read**()
- print(content)
- **finally:**
- file.**close**()

