Python

1. History of Python

Python was created by **Guido van Rossum** in **1991**. It is designed to be easy to read and write, with a focus on reducing complexity.

Python 2 is released in 2000, introduced features like list comprehensions. It was officially discontinued in 2020.

Python 3 is released in 2008, brought major improvements but was not backward-compatible with Python 2.

2. Features of Python

1. **Easy to Learn and Use**: Python has a simple syntax that resembles English, making it beginner-friendly.
2. **Interpreted Language**: Python code is executed line by line, making debugging easier.
3. **Dynamically Typed**: No need to declare variable types explicitly.
4. **Platform Independent**: Python code runs on various operating systems like Windows, macOS, and Linux without modification.
5. **Extensive Libraries**: Python has a rich standard library and third-party modules for various tasks.
6. **Supports Multiple Paradigms**: Procedural Programming, Object-Oriented Programming (OOP) and Functional Programming
7. **Open Source**: Python is free to use and distribute, even for commercial purposes.
8. **Automatic Memory Management**: Python manages memory allocation and garbage collection automatically.

3. Applications of Python

1. **Web Development**: Frameworks like Django and Flask make it easy to build robust web applications.
2. **Data Science and Machine Learning**: Libraries like NumPy, Pandas, and Scikit-learn are widely used for data analysis and modeling.
3. **Automation and Scripting**: Automates repetitive tasks using simple scripts.
4. **Game Development**: Libraries like Pygame help in creating games.
5. **Scientific Computing**: Used in research and simulations with libraries like SciPy and Matplotlib.
6. **Artificial Intelligence**: Frameworks like TensorFlow and PyTorch are popular for AI applications.
7. **Embedded Systems**: Python is used in IoT devices and microcontroller programming.

Python Syntax

Python's syntax is minimalistic and emphasizes readability. It is similar to writing in English, which makes it accessible to beginners.

1. **No Semicolons:** Statements end without semicolons.
2. **Case Sensitivity:** Python is case-sensitive, Variables like name and Name are different.
3. **Dynamic Typing:** No need to declare variable types.
4. **Line Continuation:**
    - Implicit: Parentheses, brackets, or braces allow multi-line statements.
    - Explicit: Use a backslash (\) for multi-line statements.

Indentation in Python

Python uses **indentation** to define blocks of code, replacing the use of curly braces {} or begin-end keywords seen in other languages.

Mandatory Indentation:

- Use the same number of spaces or tabs for all statements in a block. **Use 4 spaces** for each level of indentation.
- Indentation is not optional in Python. Omitting or inconsistent indentation will result in a **IndentationError.**
- Increase the indentation level for nested blocks.
- Mixing tabs and spaces can cause errors. Use either spaces or tabs consistently.

Comments in Python

Comments are used to explain the code, making it easier to understand. They are ignored during execution.

Types of Comments:

1. **Single-Line Comments:** Begin with a # symbol.
   Example: # This is a single-line comment

2. **Multi-Line Comments:** Python does not have a specific syntax for multi-line comments. However, you can use triple quotes (''' or """) to create block comments.
   Example:

```
"""
This is a multi-line comment.
It spans multiple lines.
"""
```

Python Variables and Constants

Variables in Python

Python variables has dynamic typing. Explicit type declaration does not require. The type is determined at runtime.

Constants in Python

Python does not have a built-in constant declaration mechanism, but naming conventions (e.g., all uppercase) are used to indicate constants.

Basic Data Types in Python

Python provides several built-in data types to handle different kinds of data. Below is a table explaining the basic data types with examples:

Data Type	Description	Example
int	Represents integers (whole numbers).	x = 10
float	Represents floating-point numbers (numbers with decimal points).	pi = 3.14
str	Represents strings (sequence of characters).	name = "Alice"
bool	Represents Boolean values (True or False).	is_valid = True
complex	Represents complex numbers (numbers with a real and imaginary part).	z = 2 + 3j

Note: Use type() function to check the data type of a variable.

Example: print(type(10)) # Output: <class 'int'>

Type Conversion and Type Casting

Python allows you to convert data from one type to another. This is essential for operations involving different data types.

Type Conversion vs. Type Casting

Aspect	Type Conversion	Type Casting
Definition	Implicit conversion done by Python automatically.	Explicit conversion done by the programmer using built-in functions.
Control	Performed automatically by Python when compatible types are involved.	Requires the programmer to specify the target data type explicitly.
Example	Adding an integer to a float results in a float automatically.	Converting a string to an integer using int().
Code Example	result = 5 + 2.0 (Result: 7.0)	num = int("10") (Converts string "10" to integer 10).

Common Type Casting Functions

Function	Description	Example	Explanation
int()	Converts to an integer.	int("123") → 123	Converts strings or floats to integers, truncating decimals.
float()	Converts to a floating-point number.	float("3.14") → 3.14	Converts integers or strings to floats.
str()	Converts to a string.	str(10) → "10"	Converts numbers or other types to string.
bool()	Converts to a Boolean value.	bool(0) → False	0, None, or empty values are False; others are True.
complex()	Converts to a complex number.	complex(2, 3) → 2+3j	Creates a complex number with real and imaginary parts.

Input and Output Functions

1. **Input Function (input()):** Used to take user input as a string.
   Syntax: variable = input(prompt)

Property	Description
Always returns a string	Data entered through input() is always returned as a string.
Type conversion required	Convert the input to the required type using type casting.
Example of conversion	age = int(input("Enter your age: ")) (Converts the input string to an integer).

2. **Output Function (print()):** Used to display output on the screen.
   Syntax: print(value1, value2, ..., sep=' ', end='\n')
   Parameters:
   - **sep:** Defines the separator between values (default is space).
   - **end:** Defines what to print at the end of the output (default is newline).

Feature	Example	Explanation
Basic Output	print("Hello, World!")	Prints the string Hello, World!.
Multiple Values	print("Name:", "Alice")	Prints Name: Alice with a space as the separator.
Custom Separator	print(1, 2, 3, sep="-")	Prints 1-2-3 using - as the separator.
Custom Ending	print("Hello", end=" ")	Prints Hello without a newline, appending a space instead.
Formatted Output	print(f"Age: {age}")	Uses f-strings to embed variables or expressions in strings.

Conditional Statements in Python

Conditional statements control the flow of execution based on conditions.

Statement	Syntax	Description
if	python if condition: statement(s)	Executes the block if the condition is True.
elif	python elif condition: statement(s)	Checks another condition if the previous one is False.
else	python else: statement(s)	Executes the block if all previous conditions are False.

Example:

```
age = int(input("Enter your age: "))
if age < 18:
   print("You are a minor.")
elif age == 18:
   print("You are just 18.")
else:
   print("You are an adult.")
```

Input	Output
15	You are a minor.
18	You are just 18.
21	You are an adult.

Loops in Python

Loops are used to execute a block of code multiple times.

Loop Type	Syntax	Description
for Loop	python for variable in iterable: statement(s)	Iterates over items in a sequence (e.g., list, range, string).
while Loop	python while condition: statement(s)	Repeats as long as the condition is True.

Example of for loop:

```
# Iterating through a range
   for i in range(5):
    print(i, end=" ")
# Output: 0 1 2 3 4
```

Example of while Loop

```
# Countdown using while loop
   n = 5
  while n > 0:
   print(n, end=" ")
   n -= 1
# Output: 5 4 3 2 1
```

Loop Control Statements

These statements modify the flow of a loop.

Statement	Syntax	Description
break	python break	Exits the loop immediately, regardless of the condition.
continue	python continue	Skips the current iteration and continues with the next one.
pass	python pass	Does nothing; used as a placeholder for future code.

Examples of Loop Control Statements

1. **break Example**

```
for i in range(10):
    if i == 5:
        break
    print(i, end=" ")
# Output: 0 1 2 3 4
```

2. **continue Example**

```
for i in range(5):
    if i == 2:
        continue
    print(i, end=" ")
# Output: 0 1 3 4
```

3. **pass Example**

```
for i in range(3):
    pass        # Placeholder for future code
print("Loop executed.")
# Output: Loop executed.
```

Functions in Python

Functions are reusable blocks of code that perform a specific task. They enhance modularity and reusability in programming.

1. Defining and Calling Functions

A function is defined using the **def keyword** followed by the function name and parentheses.

Syntax: python def function_name(parameters): statement(s)

Calling a Function: Functions are invoked using their name followed by parentheses.

Return Statement: The return keyword is used to send a value back to the caller.

Example:

```
# Function definition
def greet(name):
    return f"Hello, {name}!"
# Function call
message = greet("Alice")
print(message)
# Output: Hello, Alice!
```

2. Parameters and Return Values

Parameters: Parameters are variables used in the function definition to accept input values when the function is called.

Type of Parameter	Explanation	Example

Positional Parameters	Values are passed in the order they appear in the function definition.	def add(a, b): return a + b
		add(2, 3) (Result: 5)

Default Parameters	Assigns a default value to a parameter if no value is provided during the function call.	def greet(name="Guest"): print("Hello, {name}")
		greet() (Output: Hello, Guest)

Keyword Parameters	Allows specifying parameters by name during the function call.	def greet(name, msg): print(f"{msg}, {name}")
		greet(name="Alice", msg="Hi")

Variable-Length Args	Allows a function to accept any number of positional or keyword arguments.	def print_all(*args): print(args)
		print_all(1, 2, 3) (Output: (1, 2, 3))

Return Values

The return statement is used to send the result of a function back to the caller.

Aspect	Explanation	Example

Single Value Return	Returns a single value to the caller.	def square(x): return x ** 2
		square(4) (Result: 16)

Multiple Values Return	Returns multiple values as a tuple.	def calc(a, b): return a + b, a * b
		calc(2, 3) (Result: (5, 6))

No Return Value	If return is omitted, the function returns None by default.	def say_hello(): print("Hello")
		say_hello() (Result: None)

Example: Parameters and Return Values

```
# Function with parameters and return values
def calculate_area(length, width):
    return length * width

# Calling the function
area = calculate_area(5, 3)
print(f"Area: {area}")
# Output: Area: 15
```

Default and Keyword Arguments in Python

Default Arguments

Default arguments allow a function to use default values if no value is provided for the parameter during the function call.

- Parameters are assigned default values in the function definition.
  Example: def greet(name="Guest"): print(f"Hello, {name}")
- If no argument is passed, the default value is used.
  Example: greet() → Output: Hello, Guest
- Default arguments must follow non-default arguments in the function definition.
  Example: def func(a, b=10): pass is valid; def func(a=10, b): pass is invalid.

Example of Default Arguments

```
def greet(name="Guest"):
    print(f"Hello, {name}")
greet()  # Output: Hello, Guest
greet("Alice")  # Output: Hello, Alice
```

Keyword Arguments

Keyword arguments allow you to pass values to function parameters by explicitly naming them during the function call.

- Arguments are passed by explicitly specifying parameter names.
  Example: def greet(name, msg): print("{msg}, {name}")
- The order of arguments does not matter when using keywords.
  Example: greet(name="Alice", msg="Hi") → Output: Hi, Alice
- Positional arguments must precede keyword arguments in the function call.
  Example: greet("Alice", msg="Hi") is valid; greet(msg="Hi", "Alice") is invalid.

Example of Keyword Arguments

```
def greet(name, msg="Hello"):
   print(f"{msg}, {name}")
greet(name="Alice", msg="Hi")   # Output: Hi, Alice
greet("Bob")  # Output: Hello, Bob
```

Introduction to Modules

A module is a file containing Python code (functions, classes, or variables) that can be reused in other programs.

Built-in Modules

Module Name	Purpose	Common Functions/Classes	Example Usage
math	Provides mathematical functions.	sqrt(), ceil(), floor(), pow(), pi, sin(), cos(), tan()	import math print(math.sqrt(16)) → 4.0
random	Generates random numbers and performs random selections.	randint(), random(), choice(), shuffle(), uniform()	import random print(random.randint(1, 10)) → Random integer between 1 and 10
os	Interacts with the operating system.	getcwd(), listdir(), mkdir(), remove(), rename(), system()	import os print(os.getcwd()) → Current working directory
sys	Provides access to system-specific parameters and functions.	argv, exit(), path, platform, version	import sys print(sys.version) → Python version
datetime	Handles date and time operations.	datetime(), date(), time(), timedelta(), now(), strftime()	import datetime print(datetime.datetime.now()) → Current date and time
time	Handles time-related operations.	time(), sleep(), ctime(), strftime(), localtime()	import time time.sleep(2) → Pauses execution for 2 seconds
re	Provides support for regular expressions (pattern matching).	search(), match(), findall(), sub(), compile()	import re print(re.findall(r'\d+', 'abc123')) → ['123']

Importing and Using Libraries

Python allows importing libraries (modules or packages) using the import statement.

Method	Syntax	Explanation
Full Import	import module_name	Imports the entire module.
Selective Import	from module_name import function_name	Imports specific functions or variables.
Alias Import	import module_name as alias	Provides an alias for the module name.

Examples of Importing Libraries

```
# Full Import
import math
print(math.sqrt(9))  # Output: 3.0

# Selective Import
from math import pi, pow
print(pi)  # Output: 3.141592653589793
print(pow(2, 3))  # Output: 8.0

# Alias Import
import random as rnd
print(rnd.randint(1, 5))  # Output: Random integer between 1 and 5
```

Data Structures and Their Operations

1. **Lists:** A list is an **ordered**, **mutable** collection of elements. It can contain items of different data types and allows duplicates.

Indexed: Elements can be accessed using indices (starting from 0).

Dynamic: Size can grow or shrink dynamically.

Mutable: Items can be modified after creation.

Operation	Description	Example Code
Create	Use square brackets [].	my_list = [1, 2, 3, 'Python']
Access	Use indices.	print(my_list[0]) # Output: 1
Add	append(), extend(), insert()	my_list.append(4) my_list.extend([5, 6])
Remove	remove(), pop(), clear()	my_list.remove(2) my_list.pop() # Removes last element
Update	Direct assignment.	my_list[0] = 'Updated'
Iterate	Use a for loop.	for item in my_list: print(item)

2. **Tuples:** A tuple is an **ordered**, **immutable** collection of elements. It can also contain mixed data types and allows duplicates.

Immutable: Elements cannot be modified after creation.

Lightweight: Tuples are faster than lists.

Suitable for fixed data that should not change.

Operation	Description	Example Code
Create	Use parentheses ().	my_tuple = (1, 2, 3, 'Python')
Access	Use indices.	print(my_tuple[0]) # Output: 1
Immutability	Cannot add, remove, or update elements.	# my_tuple[0] = 'Updated' # Error
Iterate	Use a for loop.	for item in my_tuple: print(item)

3. **Sets:** A set is an **unordered**, **mutable** collection of unique elements. It does not allow duplicates.

Unordered: No indexing or slicing.

Unique: Duplicate elements are automatically removed.

Supports mathematical operations like union and intersection.

Operation	**Description**	**Example Code**
Create	Use curly braces {} or set().	my_set = {1, 2, 3}
Add	Use add() or update().	my_set.add(4) my_set.update([5, 6])
Remove	Use remove(), discard(), pop().	my_set.remove(2) my_set.pop()
Iterate	Use a for loop.	for item in my_set: print(item)
Union	Combines two sets.	set1 = {1, 2} set2 = {2, 3} print(set1.union(set2)) # Output: {1, 2, 3}
Intersection	Finds common elements.	print(set1.intersection(set2)) # Output: {2}

4. **Dictionaries:** A dictionary is an **unordered**, **mutable** collection of key-value pairs. Keys must be unique and immutable, while values can be mutable.

Fast lookups using keys.

Dynamic: Can grow or shrink dynamically.

Keys are unique; values can be duplicated.

Operation	**Description**	**Example Code**
Create	Use curly braces {} or dict().	my_dict = {'a': 1, 'b': 2}
Access	Use keys.	print(my_dict['a']) # Output: 1
Add/Update	Assign key-value pairs.	my_dict['c'] = 3
Remove	Use pop(), clear().	my_dict.pop('a')
Iterate	Use for loop with keys(), values(), items().	for key, value in my_dict.items(): print(key, value)

Key Differences between List, Tupple, Set and Dictionary

Feature	**List**	**Tuple**	**Set**	**Dictionary**
Ordered	Yes	Yes	No	No
Mutable	Yes	No	Yes	Yes
Allows Duplicates	Yes	Yes	No	Keys: No, Values: Yes
Use Case	General-purpose storage	Fixed data	Unique elements	Key-value mappings

5. **Comprehensions:**

Comprehensions are a concise way to create data structures.

Type	**Example Code**	**Output**
List Comprehension	squares = [x**2 for x in range(5)] print(squares)	[0, 1, 4, 9, 16]
Set Comprehension	unique_squares = {x**2 for x in [1, 2, 2, 3]} print(unique_squares)	{1, 4, 9}
Dictionary Comprehension	square_dict = {x: x**2 for x in range(5)} print(square_dict)	{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **"objects"**, which can contain **data** (attributes) and **methods** (functions) that operate on the data.
It makes programming more modular, reusable, and organized.

1. **Class**: A blueprint for creating objects. It defines the attributes and methods that the objects of the class will have.
   A class is a user-defined data type that serves as a blueprint for creating objects.
   It is defined using the class keyword.

2. **Object**: An instance of a class. Objects are created using the class blueprint and can have their own data and behavior.
3. **Methods**: Functions defined inside a class that operate on the attributes of the class.

Syntax for Defining a Class

```
class ClassName:
   # Constructor method (optional, used to initialize attributes)
   def __init__(self, attribute1, attribute2):
      self.attribute1 = attribute1
      self.attribute2 = attribute2
   # Method to perform an action
   def method_name(self):
      # Code for the method
      pass
# Creating an object
object_name = ClassName(attribute1_value, attribute2_value)
```

Defining a Class and Creating an Object

```
class Student:
   # Constructor to initialize attributes
   def __init__(self, name, age):
      self.name = name
      self.age = age

   # Method to display student details
   def display_details(self):
      print(f"Name: {self.name}, Age: {self.age}")

# Creating an object of the Student class
student1 = Student("Alice", 20)

# Accessing the method using the object
student1.display_details()
```

Output:

```
yaml
Name: Alice, Age: 20
```

Constructor (__init__ Method)

A **constructor** is a special method in Python, defined using the __init__ method, that is automatically called when an object of a class is created.

It is used to **initialize the attributes** of the class.

The constructor is defined using the __init__ method.

It is automatically invoked when an object is created.

It typically initializes the attributes of the object.

Syntax

```
class ClassName:
    def __init__(self, parameter1, parameter2):
        self.attribute1 = parameter1
        self.attribute2 = parameter2
```

Example: Constructor

```
class Person:
    def __init__(self, name, age):
        # Initializing attributes
        self.name = name
        self.age = age
    def display_details(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Creating objects
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# Accessing methods
person1.display_details()  # Output: Name: Alice, Age: 25
person2.display_details()  # Output: Name: Bob, Age: 30
```

Inheritance

- Inheritance allows a class (called the **child class**) to inherit the attributes and methods of another class (called the **parent class**).
- It helps in **reusing code** and extending the functionality of existing classes.
- **Parent Class (Base Class)**: The class whose properties are inherited.
- **Child Class (Derived Class)**: The class that inherits the properties of the parent class.
- A child class can add new attributes or methods and can override the methods of the parent class.

Syntax

```
class ParentClass:
    # Parent class methods and attributes
    pass
class ChildClass(ParentClass):
    # Child class methods and attributes
    pass
```

Example: Inheritance

```
class Animal:
    def __init__(self, species):
        self.species = species

    def display_species(self):
        print(f"Species: {self.species}")

class Dog(Animal):  # Dog class inherits from Animal
    def __init__(self, species, breed):
        super().__init__(species)  # Calling the parent class constructor
        self.breed = breed

    def display_details(self):
        print(f"Species: {self.species}, Breed: {self.breed}")

# Creating an object of the Dog class
dog1 = Dog("Mammal", "Golden Retriever")

# Accessing parent and child methods
dog1.display_species()    # Output: Species: Mammal
dog1.display_details()    # Output: Species: Mammal, Breed: Golden Retriever
```

Polymorphism

Polymorphism means "many forms." It allows the same method or operator to behave differently based on the object or data type.

Types of Polymorphism

1. **Method Overriding**: A child class redefines a method of the parent class.
2. **Method Overloading (Not natively supported in Python)**: Achieved by default parameters or handling multiple argument types.

Example: Method Overriding

```
class Vehicle:
    def start(self):
        print("Vehicle is starting...")

class Car(Vehicle):  # Car class inherits from Vehicle
    def start(self):  # Overriding the start method
        print("Car is starting...")

# Creating objects
vehicle = Vehicle()
car = Car()

# Calling methods
vehicle.start()  # Output: Vehicle is starting...
car.start()      # Output: Car is starting...
```

Example: Polymorphism with a Common Interface

```
class Bird:
    def sound(self):
        print("Birds chirp.")

class Dog:
    def sound(self):
        print("Dogs bark.")

# Polymorphic behavior
def make_sound(animal):
    animal.sound()

# Using the same function for different objects
make_sound(Bird())  # Output: Birds chirp.
make_sound(Dog())   # Output: Dogs bark.
```

File Handling in Python

File handling is a way to work with files (e.g., text files) to perform operations like reading, writing, and closing files. Python provides built-in functions and methods to make file handling simple and efficient.

Basic File Operations

1. **Opening a File**: Use the open() function.
2. **Reading from a File**: Use methods like read(), readline(), or readlines().
3. **Writing to a File**: Use write() or writelines().
4. **Closing a File**: Use the close() method to release resources.

File Modes

Mode	Description
'r'	Opens a file for reading (default mode). The file must exist.
'w'	Opens a file for writing. If the file exists, it is overwritten. If it doesn’t exist, a new file is created.
'a'	Opens a file for appending. Data is added to the end of the file. If the file doesn’t exist, it creates a new file.
'r+'	Opens a file for both reading and writing. The file must exist.
'w+'	Opens a file for both writing and reading. If the file exists, it is overwritten. If it doesn’t exist, a new file is created.
'a+'	Opens a file for both appending and reading. Data is added to the end of the file. If the file doesn’t exist, a new file is created.

Opening and Closing Files

Syntax

```
file = open("filename", "mode")  # Opens the file
file.close()       # Closes the file
```

Example

```
file = open("example.txt", "w")  # Open file in write mode
file.write("Hello, World!")      # Write to the file
file.close()                # Close the file
```

Reading from a File

Method	Description	Example	When to Use
read(size)	Reads the specified number of characters from the file. If size is not specified, reads the entire file.	with open("file.txt", "r") as f: print(f.read(5)) # Reads first 5 chars	Use when you need a portion of the file or want to control memory usage for large files.
readline()	Reads one line from the file, including the newline character (\n).	with open("file.txt", "r") as f: print(f.readline()) # Reads first line	Use when processing files line by line, such as reading logs or structured text files.
readlines()	Reads all lines in the file and returns them as a list of strings.	with open("file.txt", "r") as f: print(f.readlines()) # List of all lines	Use when you need all lines at once and can afford to load the entire file into memory.

Writing to a File

Method	Description	Short Example	When to Use
write(string)	Writes a string to the file. If the file doesn't exist, it creates one.	with open("file.txt", "w") as f: f.write("Hello, World!")	Use when writing a single string or appending a specific line to a file.
writelines(list)	Writes a list of strings to the file without adding newlines automatically.	with open("file.txt", "w") as f:\ f.writelines(["Line1", "Line2"])	Use when writing multiple lines from a list. Ensure lines include \n for proper formatting.

Using with Statement

The with statement is used to automatically close the file after the block of code is executed, even if an exception occurs.

Example

```
with open("example.txt", "r") as file:
   content = file.read()
   print(content)
# No need to call file.close() explicitly
```

Handling Exceptions during File Operations in Python

When performing file operations, errors can occur, such as trying to read a non-existent file, writing to a file without proper permissions, or encountering unexpected issues.

Python provides a mechanism to handle such errors using **exception handling**.

Common Exceptions in File Handling

Exception	Description
FileNotFoundError	Raised when trying to open a file that doesn't exist.
PermissionError	Raised when trying to access a file without the required permissions.
IOError	Raised for general I/O errors, such as reading from a closed file.
ValueError	Raised when an invalid mode is passed to open().

Using try-except for Exception Handling

```
try:
   # Code that may raise an exception
except ExceptionType:
   # Handle the exception
finally:
   # Optional: Code to execute regardless of an exception
```

Nested try-except Blocks: Sometimes, multiple operations on a file may require separate exception handling.