Introduction to python

Python and its Features

Python is an interpreted, high-level language known for its simplicity and readability. Key features include:

- **Dynamic Typing**: Variables are assigned types at runtime.
- **Extensive Libraries**: Python offers a vast standard library.
- **Cross-platform**: Compatible with Windows, macOS, Linux, etc.

# Python's simplicity in syntax

print("Hello, World!")

History of Python

Python was created by Guido van Rossum in the late 1980s and first released in 1991. It was designed for readability and productivity, inspired by the ABC language.

Keywords & Identifiers

Keywords are reserved words with special meanings (e.g., if, for, True). **Identifiers** are names for variables, functions, etc., which should follow naming conventions (letters, numbers, and underscores but cannot start with a number).

Variables & Operators

Variables store data values. **Operators** perform operations on these values, such as:

- Arithmetic (+, -)
- Comparison (==, !=)
- Logical (and, or)

Data Types

Python has various data types: integers, floats, strings, lists, tuples, dictionaries, and sets.

Example:

```
# Different data types

integer_val = 10

float_val = 10.5

string_val = "Python"

list_val = [1, 2, 3]
```

Numeric

Python's numeric data types include:

- **Integers**: Whole numbers.
- **Floats**: Decimal numbers.
- **Complex Numbers**: Numbers with a real and imaginary part.

Example:

```
int_val = 42       # Integer
float_val = 3.14   # Float
complex_val = 2 + 3j  # Complex
```

Sequence

Sequence types in Python include **lists**, **tuples**, and **strings**. These are ordered collections of items.

Example:

```
# List, tuple, and string examples
my_list = [1, 2, 3]
my_tuple = (1, 2, 3)
my_string = "Python"
```

Boolean

The Boolean type represents True or False values. It's useful for conditions and logical operations.

Example:

```
is_active = True
if is_active:
    print("Boolean example: Active")
```

Control Structure

If Statement

The if statement checks a condition and executes a block of code if the condition is True.

Example:

```
x = 10

if x > 5:

    print("x is greater than 5")
```

If-Else Statement

The if-else statement provides an alternative block to execute if the condition is False.

Example:

```
x = 3

if x > 5:

    print("x is greater than 5")

else:

    print("x is 5 or less")
```

If-Elif-Else Statement

This allows for multiple conditions to be checked sequentially using elif (else if).

Example:

```
x = 7

if x > 10:

    print("x is greater than 10")

elif x > 5:

    print("x is greater than 5 but not greater than 10")

else:

    print("x is 5 or less")
```

Control Structure

Control structures manage the flow of a program, such as decision-making and loops. **If statements** and **loops** (for, while) are common control structures.

For Loop

The for loop iterates over a sequence (like a list or range) and executes code for each element.

Example:

```
for i in range(3):
    print(i)
# Output: 0, 1, 2
```

While Loop

A while loop continues as long as a specified condition is True.

Example:

```
x = 0
while x < 3:
    print(x)
    x += 1
# Output: 0, 1, 2
```

Nested Loop

A loop inside another loop, useful for working with multi-dimensional data.

Example:

```
for i in range(2):
    for j in range(3):
        print(f"i={i}, j={j}")
```

Break, Continue & Pass

- **Break**: Exits the loop.
- **Continue**: Skips the current iteration.
- **Pass**: Acts as a placeholder with no action.

Example:

```
for i in range(5):
    if i == 2:
        continue  # Skip 2
    elif i == 4:
        break   # Stop the loop at 4
    else:
        print(i)
```

# Output: 0, 1, 3

Input and Output

The input() function takes user input, and print() outputs data to the console.

Example:

```
name = input("Enter your name: ")

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

Introduction to Lists

A list is an ordered, mutable sequence of elements defined with square brackets [].

Example:

```
fruits = ["apple", "banana", "cherry"]

print(fruits)
```

List Methods and Slicing

- **List Methods**: append(), remove(), sort().
- **Slicing**: Extracts parts of a list using index ranges.

Example:

```
fruits = ["apple", "banana", "cherry"]

fruits.append("orange")

print(fruits[1:3])  # Slicing from index 1 to 2
```

Introduction to Dictionaries & Dictionary Methods

Dictionaries store key-value pairs and are defined with curly braces {}.

Example:

```
person = {"name": "Alice", "age": 25}

print(person["name"])


# Dictionary methods

person["age"] = 26  # Update value

print(person.get("age"))
```

Introduction to Set & Set Methods

Sets are unordered collections of unique elements, defined with {}.

Example:

```
numbers = {1, 2, 3, 3}  # Duplicates are removed
numbers.add(4)
print(numbers)
```

Introduction to Map & Map Methods

map() applies a function to each item in an iterable, creating a new iterable with the results.

Example:

```
numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))
# Output: [1, 4, 9, 16]
```

Functions

Mapping Function

A **mapping function** like map() applies a function to each item in an iterable, returning an iterator with the results.

Example:

```
numbers = [1, 2, 3, 4]

squared = map(lambda x: x ** 2, numbers)

print(list(squared))

# Output: [1, 4, 9, 16]
```

String Function

String functions are built-in methods to manipulate strings, such as upper(), lower(), and replace().

Example:

```
text = "hello world"

print(text.upper())      # Output: HELLO WORLD

print(text.replace("world", "Python"))  # Output: hello Python
```

Number Function

Number functions include operations like abs(), round(), and pow(), as well as math module functions for advanced operations.

Example:

```
num = -7.5

print(abs(num))          # Output: 7.5

print(round(3.14159, 2)) # Output: 3.14
```

Date and Time Function

The datetime module provides functions to work with dates and times.

Example:

```
from datetime import datetime

now = datetime.now()

print(now.strftime("%Y-%m-%d %H:%M:%S"))

# Output: Current date and time in "YYYY-MM-DD HH:MM:SS" format
```

Python Functions

Functions in Python are reusable blocks of code, defined with the def keyword.

Example:

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

Default Argument Values

Default values are specified in function arguments so that the caller doesn't need to provide them.

Example:

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

Keyword Arguments

Keyword arguments are explicitly named when calling a function, allowing you to specify arguments out of order.

Example:

```
def introduce(name, age):
    print(f"My name is {name} and I am {age} years old.")

introduce(age=25, name="Alice")
# Output: My name is Alice and I am 25 years old.
```

Special Parameters

Special parameters (like / and *) in function definitions clarify positional-only or keyword-only arguments.

Example:

```
def func(a, b, /, c, *, d):
    print(a, b, c, d)


func(1, 2, c=3, d=4)  # Works
```

```
# func(a=1, b=2, c=3, d=4)  # Would raise an error
```

- Here, a and b are positional-only, while d is keyword-only.

Arbitrary Argument Lists

Arbitrary argument lists allow a function to accept any number of arguments using *args (for positional arguments) and **kwargs (for keyword arguments).

Example:

```
def add_all(*args):
    return sum(args)

print(add_all(1, 2, 3, 4))  # Output: 10

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

display_info(name="Alice", age=25)
# Output:
# name: Alice
# age: 25
```

Lambda Expressions

Lambda expressions are anonymous functions defined with lambda, often used for short, simple functions.

Example:

```
square = lambda x: x ** 2
print(square(5))   # Output: 25

# Using with filter function
numbers = [1, 2, 3, 4, 5]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4]
```

OOPS

OOP is a programming paradigm that uses **objects** to represent real-world entities. It promotes code organization and reusability through key concepts like **classes**, **inheritance**, **polymorphism**, and **encapsulation**.

Class and Object

A **class** is a blueprint for creating objects, defining their properties and behaviors. An **object** is an instance of a class, with specific values for its attributes.

Example:

```
class Car:
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color

    def drive(self):
        print(f"The {self.color} {self.brand} is driving.")

# Creating an object of the Car class
my_car = Car("Toyota", "Red")
my_car.drive()
# Output: The Red Toyota is driving.
```

Access Specifiers

Access specifiers control the accessibility of class attributes and methods:

- **Public** (name): Accessible from anywhere.
- **Protected** (_name): Intended for internal use, but can be accessed from outside (convention).
- **Private** (__name): Not accessible outside the class.

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name        # Public
        self._age = age         # Protected
        self.__ssn = "123-45-6789"  # Private
```

```
p = Person("Alice", 30)
print(p.name)        # Accessible
print(p._age)        # Accessible but discouraged
# print(p.__ssn)     # Raises AttributeError (private)
```

Constructor

A **constructor** is a special method __init__() that initializes an object's attributes when the object is created.

Example:

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

my_book = Book("1984", "George Orwell")
print(my_book.title)   # Output: 1984
```

Inheritance

Inheritance allows a class (child) to inherit attributes and methods from another class (parent), promoting code reuse.

Example:

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):  # Dog inherits from Animal
    def bark(self):
        print("Dog barks")

d = Dog()
d.speak()  # Output: Animal speaks
d.bark()   # Output: Dog barks
```

Polymorphism

Polymorphism means "many forms" and allows different classes to use the same method name with different implementations.

Example:

```
class Bird:
    def sound(self):
        print("Bird chirps")

class Dog:
    def sound(self):
        print("Dog barks")

# Polymorphism in action
for animal in (Bird(), Dog()):
    animal.sound()
# Output:
# Bird chirps
# Dog barks
```

Method Overriding

Method overriding allows a child class to provide a specific implementation of a method already defined in its parent class.

Example:

```
class Animal:
    def speak(self):
        print("Animal sound")

class Cat(Animal):
    def speak(self):   # Overriding parent method
        print("Cat meows")


c = Cat()
c.speak()  # Output: Cat meows
```

File Handling

Python provides built-in functions for file operations like reading, writing, and appending.

Example:

```
# Writing to a file
with open("example.txt", "w") as file:
    file.write("Hello, File Handling!")

# Reading from a file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# Output: Hello, File Handling!
```

Exception Handling

Exception handling helps manage runtime errors using try, except, finally blocks, preventing the program from crashing.

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
finally:
    print("Execution complete.")

# Output:
# Cannot divide by zero!
# Execution complete.
```

Modules & Package

Python Modules

A **module** is a file containing Python code (usually .py) that defines functions, classes, or variables. Modules allow you to organize code logically and reuse it in multiple programs.

Example (mymodule.py): # A simple module with a function

```
def greet(name):
    return f"Hello, {name}!"
```

To use the module in another script:

```
import mymodule

print(mymodule.greet("Alice"))  # Output: Hello, Alice!
```

User-defined Modules

User-defined modules are custom Python files created to organize code into reusable components. You create them just like regular Python files, but with reusable functions, classes, or variables.

Example (math_util.py): # A custom module for math utilities

```
def add(a, b):
    return a + b
```

Using the Module:

```
import math_util

print(math_util.add(3, 4))  # Output: 7
```

Executing Modules as Scripts

Modules can be executed as standalone scripts by including the special __name__ variable check. This allows you to test a module's functionality by running it directly.

Example (mymodule.py):

```
def greet(name):
    return f"Hello, {name}!"

if __name__ == "__main__":
    # This code only runs if the module is executed directly
    print(greet("Alice"))  # Output: Hello, Alice!
```

When run as a script, this outputs "Hello, Alice!". When imported, only the function definitions are accessible, and the if block won't run.

Standard Modules

Python includes many **standard modules** in its standard library, such as math, datetime, and random, which provide additional functionality.

Example:

```
import math

print(math.sqrt(16))  # Output: 4.0


import datetime

print(datetime.date.today())  # Outputs today’s date
```

Packages

A **package** is a directory containing multiple modules (or sub-packages) and a special __init__.py file, which makes it importable as a package.

Example Package Structure:

```
my_package/
│
├── __init__.py       # Initializes the package
├── module1.py        # A module in the package
└── module2.py        # Another module in the package
```

Using the Package:

```
from my_package import module1

module1.some_function()
```

Importing * From a Package

Using from package import * imports all public symbols defined in the package's __init__.py file. It is generally better to import specific modules or functions to avoid namespace conflicts.

Example (__init__.py):

```
# Specify what to import with *

__all__ = ["module1", "module2"]
```

Usage:

```
from my_package import *
```

Intra-package References

Intra-package references allow modules within a package to import each other using relative imports (like . or ..).

Example:

```
# Inside module2.py in my_package

from .module1 import some_function  # Relative import within package
```

This lets module2.py access some_function from module1.py without fully qualifying the package name.

Packages in Multiple Directories

Packages can span multiple directories by adding each directory to sys.path, the list of paths Python searches for modules. You can use the PYTHONPATH environment variable or modify sys.path at runtime.

Example:

```
import sys

sys.path.append("/path/to/another_directory")

import another_package.module
```

This lets Python find and import modules from the specified directory.