PYTHON PROGRAMMING

UNIT 02

What are Modules in Python?

Module is a file that contains code to perform a specific task. A module may contain Python code, definitions of functions, statements, or classes.

We employ modules to divide complicated programs into smaller, more understandable pieces. Modules also allow for the reuse of code.

Rather than duplicating their definitions into several applications, we may define our most frequently used functions in a separate module and then import the complete module.

An **example_module.py** file is a module we will create and whose name is **example_module**.

Let's construct a module. Save the file as **example_module.py** after entering the following.

Example 1:

```
# Here, we are creating a simple Python program to show how to create a module.

# defining a function in the module to reuse it
def square( number ):
   # here, the above function will square the number passed as the input
    result = number ** 2
    return result     # here, we are returning the result of the function
```

Here, a module called example_module contains the definition of the function **square()**. The function returns the square of a given number.

Example 2:

Type the following and save it as example.py.

```
# Python Module addition


def add(a, b):


   result = a + b

   return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

How to Import Modules in Python?

In Python, we may import functions from one module into our program, or as we say into, another module.

For this, we make use of the **import** keyword. In the Python window, we add the next to import keyword, the name of the module we need to import. We will import the module we defined earlier **example_module**.

Syntax:

1. **import** example_module

   The functions that we defined in the example_module are not imported into the present program. Only the name of the module, i.e., example_ module, is imported here.

   Using the module name we can access the function using the dot . operator. For example:

   Example:

```
1. # here, we are calling the module square method and passing the value 4
2. result = example_module.square(  4  )
3. print("By using the module square of number is: ", result )
```

   Output:

By using the module square of number is: 16

For example 2: example.add(4,5) # returns 9

There are several standard modules for Python. The complete list of Python standard modules is available. The list can be seen using the help command.

Similar to how we imported our module, a user-defined module, we can use an import statement to import other standard modules.

Importing a module can be done in a variety of ways. Below is a list of them.

Python import Statement

Using the import Python keyword and the dot operator, we may import a standard module and can access the defined functions within it. Here's an illustration.

Suppose we want to get the value of pi, first we import the math module and use math.pi.

For example:

```
    import math                          # import standard math module
print("The value of pi is", math.pi)     # use math.pi to get value of pi
```

Output

The value of pi is 3.141592653589793

Importing and also Renaming

While importing a module, we can change its name too. Here is an example to show.

Code

```
# import module by renaming it

import math as m


print(m.pi)

                    # Output: 3.141592653589793
```

Output:

Here, We have renamed the math module as m. This can save us typing time in some cases.

Note that the name math is not recognized in our scope. Hence, math.pi is invalid, and m.pi is the correct implementation.

Python from...import Statement

We can import specific names from a module without importing the module as a whole. Here is an example.

Code

```
# import only pi from math module

from math import pi


print(pi)

                    # Output: 3.141592653589793
```

Here, we imported only the pi attribute from the math module.

We avoid using the dot (.) operator in these scenarios. As follows, we may import many attributes at the same time:

Code

```
1. # Here, we are creating a simple Python program to show how to import multiple

2. # objects from a module
3. from math import e, tau
4. print( "The value of tau constant is: ", tau )
5. print( "The value of the euler's number is: ", e )
```

Output:

The value of tau constant is: 6.283185307179586
The value of the euler's number is: 2.718281828459045

Import all Names - From import * Statement

In Python, we can import all names(definitions) from a module using the following construct:

```
# import all names from the standard module math

from math import *

print("The value of pi is", pi)
```

Here, we have imported all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).

Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

Locating Path of Modules

The interpreter searches numerous places when importing a module in the Python program. Several directories are searched if the built-in module is not present. The list of directories can be accessed using sys.path. The Python interpreter looks for the module in the way described below:

The module is initially looked for in the current working directory. Python then explores every directory in the shell parameter PYTHONPATH if the module cannot be located in the current directory. A list of folders makes up the environment variable known as PYTHONPATH. Python examines the installation-dependent set of folders set up when Python is downloaded if that also fails.

Here is an example to print the path.

Code

```
1. # Here, we are importing the sys module
2. import sys
3. # Here, we are printing the path using sys.path
4. print("Path of the sys module in the system is:", sys.path)
```

Output:

Path of the sys module in the system is:
['/home/pyodide', '/home/pyodide/lib/Python310.zip', '/lib/Python3.10', '/lib/Python3.10/lib-dynload', '', '/lib/Python3.10/site-packages']

The dir() Built-in Function

In Python, we can use the dir() function to list all the function names in a module.

For instance, we have the following names in the standard module str. To print the names, we will use the dir() method in the following way:

Code

```
1. # Here, we are creating a simple Python program to print the directory of a module
```

2. **print**("List of functions:\n ", dir(str), end=", ")

   Output:

List of functions:
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).

Strings:

In Python, a string is a sequence of characters. It is a simple data structure that serves as the foundation for data manipulation. Strings in Python are "immutable." They can't be modified once they're created.
For example, `"hello"` is a string containing a sequence of characters `'h'`, `'e'`, `'l'`, `'l'`, and `'o'`.

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello"

You can display a string literal with the **print()** function:

Example 1:

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

Example 2:

```
# create string type variables

    name = "Python"

    print(name)


    message = "I love Python."

    print(message)
```

Output:

```
Python

I love Python.
```

In the above example, we have created string-type **variables**: name and message with **values** "Python" and "I love Python" respectively.

Here, we have used double quotes to represent strings, but we can use single quotes too.

What is string manipulation?

String manipulation is the process of manipulating and analyzing strings. It includes a variety of processes involving the modification and parsing of strings to use and change their data.

Concatenation of Strings

Concatenation means to join two or more strings into a single string. + operator is used to concatenate strings. For example,

```
# concatenating words Coding and Ninjas.
word1 = 'Coding'
word2 = ' Ninjas'
print(word1 + word2)
```

Output:

Coding Ninjas

Replace Part of a String

The replace() method is used to replace a part of the string with another string. As an example, let's replace 'ing' in Cod**ing** with 'ers,' making it Coders.

```
string = 'Coding'
string = string.replace('ing', 'ers')
print(string)
```

Output:

Coders

Count characters

count() method is used to count a particular character in the string.

```
string = 'Coding Ninjas'

# count the character 'i'.
count1 = string.count('i')
print("Count of 'i': ", count1)

# Count spaces.
count2 = string.count(' ')
print("Number of spaces:", count2)
```

Output:

Count of ‘I’: 2

Number of spaces: 1

Split a String

It is a very common practice to split a string into smaller strings. In Python, we use the split() function to do this. It splits the string from the identified separator and returns all the pieces in a list.

Syntax:

```
string.split(separator, maxsplit)
```

- separator: the string splits at this separator.
- maxsplit (optional): tells how many splits to do.

For example:

```
string = 'Welcome to Coding Ninjas'
# Splitting across the whitespace.
myList = string.split(" ")
print(myList)
```

Output:

[‘Welcome’, ‘to’, ‘Coding’, ‘Ninjas’]

find( ) function

The find() function looks for the first occurrence of the provided value. If the value is not found in that string, it returns -1. The value can be any character or a substring.

```
string = 'Welcome to Coding Ninjas'
index = string.find('t')

print(index)
```

Output:

8

join() Function

This method is used to join a sequence of strings with some operator.
The join() operator will create a new string by joining every character of the sequence with a specified character, including whitespaces.
Note: The join() method only accepts strings. If any element in the iteration is of a different type, an error will be thrown.

Syntax:

```
"character".join(seq)
```

- character: The character from which the strings will be joined.
- seq: Sequence of the strings to be joined.

```
string = 'Welcome to Coding Ninjas'
# joining the above-given characters of the string with ‘,’.
string = ",".join(string)

print(string)
```

Output:

W,e,l,c,o,m,e, ,t,o, ,C,o,d,I,n,g, ,N,i,,n,j,a,s

Remove a Prefix or a Suffix

You can use the removeprefix() method to remove a prefix from a string.

Syntax:

```
string.removeprefix('prefix')
```

You can use the removesuffix() method to remove a suffix from a string.

Syntax:

```
string.removesuffix('suffix')
```

- **lower():** Converts all uppercase characters in a string into lowercase
- **upper():** Converts all lowercase characters in a string into uppercase
- **title():** Convert string to title case
- **swapcase():** Swap the cases of all characters in a string
- **capitalize():** Convert the first character of a string to uppercase

Example:

```
text = 'geeKs For geEkS'
print(text.upper())                    #GEEKS FOR GEEKS
print(text.lower())                    #geeks for geeks
print(text.title())                    #Geeks For Geeks
print(text.swapcase())                 #GEEkS fOR GEeKs
print(text.capitalize())          #Geeks for geeks
print(text)                      #original
```

Object-oriented programming:

In Python object-oriented Programming (OOPs) is a programming concept that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of object-oriented Programming (OOPs) or oops concepts in Python is to bind the data and the functions that work together as a single unit so that no other part of the code can access this data.

OOP in Python revolves around four main principles: **encapsulation**, **inheritance**, **polymorphism**, and **abstraction**.

Overall, OOP in Python enables developers to create modular, organized, and scalable code. It promotes the design of complex systems using smaller, reusable, and understandable building blocks.

OOPs Concepts in Python

- Class in Python
- Objects in Python
- Polymorphism in Python
- Encapsulation in Python
- Inheritance in Python
- Data Abstraction in Python

Python Class:

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods. It does not occupy memory. It's a way to group related properties and methods under a single unit.

To understand the need for creating a class let’s consider an example, let’s say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog’s breed while the second element could represent its age. Let’s suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if

you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
   # Statement-1
   .
   .
   .
   # Statement-N
```

Python Objects

An object is an instance of a class. The object is an entity that has a state and behavior associated with it. Object takes memory. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object.

The number 12 is an object, the string "Hello, world" is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Creating an Object:

This will create an object named obj of the class Dog defined above.

```
obj = Dog()
```

The Python self parameter:

Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it

If we have a method that takes no arguments, then we still have to have one argument.

This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

The Python __init__ Method:

The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object. Now let us define a class and create some objects using the self and __init__ method.

Creating a class and object with class and instance attributes

```
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

# Driver code
```

```
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

Output

```
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy
```

Creating Classes and objects with methods

Here, The Dog class is defined with two attributes:

- **attr1** is a class attribute set to the value "**mammal**". Class attributes are shared by all instances of the class.
- __init__ is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog.
  The speak method is defined within the Dog class. This method prints a string that includes the name of the dog instance.

The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The __init__ method is called for each instance to initialize their name attributes with the provided names. The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

```
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("My name is {}".format(self.name))

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

Output:

My name is Rodger

My name is Tommy

Program to understand class and object in python:

```
# Defining a class
class Car:
    # Constructor to initialize the object's attributes
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    # Method to describe the car
    def description(self):
        return f"{self.year} {self.make} {self.model}"

    # Method to start the car
    def start(self):
        return f"The {self.model} is now starting."

# Creating objects (instances of the Car class)
car1 = Car("Toyota", "Corolla", 2020)
car2 = Car("Honda", "Civic", 2019)
```

 # Accessing object properties and methods

print(car1.description()) # Output: 2020 Toyota Corolla

print(car2.start()) # Output: The Civic is now starting.

Comparison between class and object:

Aspect	Class	Object
Definition	A template or blueprint for creating objects.	An instance of a class created using the blueprint.
Memory	Does not consume memory until an object is instantiated.	Consumes memory as it holds the actual data.
Attributes	Defines the structure (attributes/methods) of objects.	Holds actual data for those attributes.
Methods	Defines behaviors that objects can perform.	Invokes methods defined by the class.
Example	Car (class)	car1, car2 (objects/instances)

In the example above, Car is the class, and car1 and car2 are objects (instances of the Car class). Each object can have its own values for the attributes like make, model, and year.

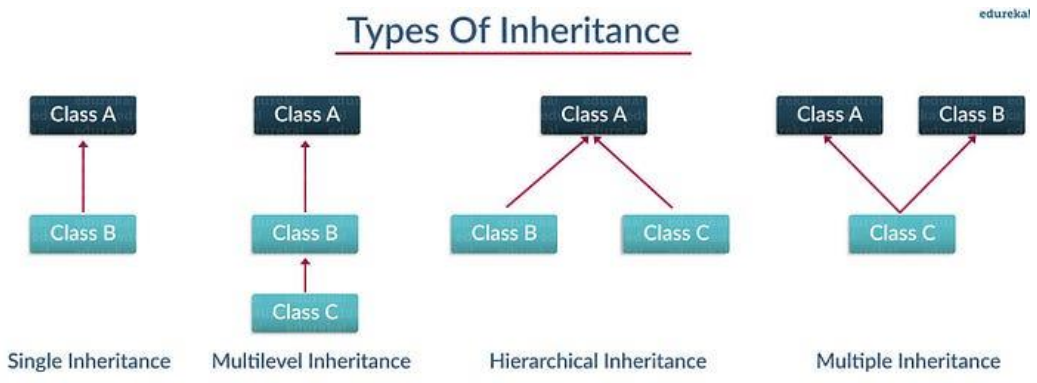
Python Inheritance

In Python object oriented Programming, Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance:

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
- **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.



Program to understand inheritance:

```
# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
```

```
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```

The super() Function:

The super() function allows you to call methods of the parent class from the child class. It is often used in class inheritance to avoid explicitly referencing the parent class. One primary use of super() is to ensure that the parent class's __init__ method is called when initializing an instance of the child class.

Using super() ensures that the parent class is properly initialized without having to hardcode the parent class's name, making the code more flexible.

Python program to demonstrate the use of super() function :

```
# Parent class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Child class inheriting from Person
class Employee(Person):
    def __init__(self, name, age, employee_id, position):
        # Call the parent class's __init__ method using super()
        super().__init__(name, age)
        self.employee_id = employee_id
        self.position = position

    def display_info(self):
        # Call the parent class's display_info method using super()
        super().display_info()  # Displays name and age
        print(f"Employee ID: {self.employee_id}, Position: {self.position}")

# Testing the functionality
person = Person("Alice", 30)
employee = Employee("Bob", 35, "E12345", "Software Developer")

# Display information for person and employee
person.display_info()  # Output: Name: Alice, Age: 30
```

```
print()  # Just to separate the outputs
employee.display_info()
```

```
# Output:
# Name: Bob, Age: 35
# Employee ID: E12345, Position: Software Developer
```

Polymorphism

Polymorphism contains two words "**poly**" and "**morphs**". **Poly** means **many**, and **morph** means **shape**. By polymorphism, we understand that one task can be performed in different ways. For example - you have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in a sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

The advantages of polymorphism in Python include:

- **Code Reusability:** You can write flexible and generic code that works with different object types, reducing duplication.
- **Maintainability:** Polymorphism helps you organize code better, making it easier to modify or extend without changing existing code.
- **Flexibility and Extensibility:** It allows for easy addition of new functionality or object types without altering existing functions or code structures.
- **Dynamic Behavior:** The same function can behave differently based on the object type, providing dynamic and context-specific behavior.

Example:

Here's a simple example of polymorphism in Python using a `speak` method for different animal classes:

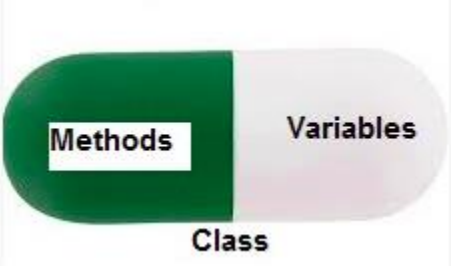
```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

# Function that takes any animal and calls its speak method
def animal_sound(animal):
    print(animal.speak())

# Create instances of Dog and Cat
dog = Dog()
cat = Cat()

# Call the same function, but with different objects
animal_sound(dog)          # Output: Woof!
animal_sound(cat)          # Output: Meow!
```

In this example, both `Dog` and `Cat` have a `speak` method, but they return different sounds. The `animal_sound` function can take any object (dog or cat) and call their `speak` method, showcasing polymorphism.

Encapsulation:

In Python object oriented programming, Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



Example of Encapsulation in Python:

In the above example, we have created the c variable as the private attribute. We cannot even access this attribute directly and can't even change its value.

Python

```
# Python program to
# demonstrate private members
# "__" double underscore represents private attribute.
# Private attributes start with "__".

# Creating a Base class
class Base:
    def __init__(self):
        self.a = "GeeksforGeeks"
        self.__c = "GeeksforGeeks"

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
```

```
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)


# Driver code
obj1 = Base()
print(obj1.a)

# Uncommenting print(obj1.c) will
# raise an AttributeError

# Uncommenting obj2 = Derived() will
# also raise an AtrributeError as
# private member of base class
# is called inside derived class
```

Output

```
GeeksforGeeks
```

Abstraction in Python

Abstraction in Object-Oriented Programming (OOP) refers to the concept of hiding the complex code details and exposing only the essential parts of a program. This allows the user to interact with an object without needing to understand the underlying complexity, making code more modular and easier to manage.

Why is Abstraction Useful?

- Reduces complexity by hiding the unnecessary details.
- Encourages code reusability and modular design.
- Provides a layer of security by restricting access to certain data or functionality.
- It simplifies the interface for the users of the class.

Example to understand abstraction in python:

```
class Rectangle:
    def __init__(self, length, width):
        self.__length = length  # Private attribute
        self.__width = width    # Private attribute

    def area(self):
        return self.__length * self.__width

    def perimeter(self):
        return 2 * (self.__length + self.__width)


rect = Rectangle(5, 3)
print(f"Area: {rect.area()}")          # Output: Area: 15
print(f"Perimeter: {rect.perimeter()}")  # Output: Perimeter: 16

# print(rect.__length)  # This will raise an AttributeError as length and width are private attributes
```

Output

```
Area: 15
Perimeter: 16
```

Exception Handling:

Exception handling is a way to manage errors that occur while a program is running. Instead of letting the program crash, you can use exception handling to respond to these errors in a controlled way.

What Are Exceptions?

- **Exceptions** are errors that happen during the execution of your program. An exception is an event that disrupts the normal flow of a program. Python provides a mechanism to catch and handle such exceptions to avoid program crashes.
- Examples include dividing by zero (ZeroDivisionError), trying to open a file that doesn't exist (FileNotFoundError), or entering a wrong type of data (ValueError).

Types of Exceptions:

There are two types of exceptions in python.

1. Built-in exceptions.
2. User defined exceptions or custom exceptions

Built-in exceptions:

Built-in exceptions are predefined exceptions that Python provides out of the box. They cover a wide range of common errors that can occur during program execution. Here are some key built-in exceptions:

- **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
- **TypeError**: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
- **NameError**: This exception is raised when a variable or function name is not found in the current scope.
- **IndexError**: This exception is raised when an index is out of range for a list, tuple, or other sequence types.
- **KeyError**: This exception is raised when a key is not found in a dictionary.
- **ValueError**: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
- **AttributeError**: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
- **IOError**: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
- **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.
- **ImportError**: This exception is raised when an import statement fails to find or load a module.

User defined exceptions or custom exceptions:

User-defined exceptions are custom exceptions created by the programmer. These exceptions can be used to represent specific error conditions that are relevant to your application. To create a user-defined exception, you typically subclass the built-in Exception class. Here's how to do it:

Example:

```
class NegativeNumberError(Exception):
 pass


def check_positive(number):
    if number < 0:
        raise NegativeNumberError("Negative numbers are not allowed.")
    return number


try:
    number = int(input("Enter a positive number: "))
    print(check_positive(number))
except NegativeNumberError as e:
    print(e)
```

How Exception Handling Works

Exception handling in Python allows you to manage errors gracefully without crashing your program.

When an error occurs in Python, an exception is raised. If the exception is not handled, it will propagate up the call stack, potentially terminating the program. To manage exceptions, you can use **try**, **except**, **else**, and **finally** blocks.

- **try**: Write the code that might cause an error.
- **except**: Write the code that runs if an error happens.
- **else**: (Optional) Write code that runs if no errors happen.
- **finally**: (Optional) Write code that always runs, whether an error happened or not

Try Block:

The try block contains code that may raise an exception. Python will attempt to execute this code.

```
try:
    # Code that might raise an exception
    result = 10 / 0  # This will raise ZeroDivisionError
```

Except Block:

The except block allows you to catch and handle specific exceptions. You can have multiple except blocks to handle different exceptions. If an error happens, the program runs the code inside the matching except block.

Else Block:

The else block executes if no exceptions are raised in the try block. It's useful for code that should run when the try block is successful.

Finally Block:

The finally block will execute regardless of whether an exception was raised or not. This is often used for cleanup actions, such as closing files or releasing resources.

Example 1:

```
def divide_numbers(num1, num2):
```

```
    try:
        result = num1 / num2
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
    except TypeError:
        print("Error: Please provide numbers.")
    else:
        print("Result:", result)
    finally:
        print("Operation completed.")

# Testing the function
divide_numbers(10, 2)  # Should print the result
divide_numbers(10, 0)  # Should handle division by zero
divide_numbers(10, 'a')  # Should handle type error
```

Example 2:

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ValueError:                  # This line might cause a ValueError
    print("Please enter a valid number!")
except ZeroDivisionError:                   # This line might cause a ZeroDivisionError
    print("You can't divide by zero!")
else:
    print(f"The result is {result}")
finally:
    print("This runs no matter what.")
```

Built-in functions:

Python provides several **built-in functions** to help handle exceptions effectively. These functions allow you to raise exceptions, catch them, and introspect them. Here are the main ones related to exception handling:

1. **raise**

- Used to manually raise an exception when a specific condition occurs.

  Usage:

  raise ValueError("This is a custom error message")

  You can also raise the same exception from within an except block using raise without arguments to re-raise the caught exception:

```
try:
    1 / 0
except ZeroDivisionError:
    print("Caught an error")
    raise  # Re-raise the original exception
```

2. **assert**

- The assert statement is used for debugging purposes. It tests a condition and raises an AssertionError if the condition is false. You can optionally provide an error message.

  Usage:

```
assert 2 + 2 == 4  # This will pass without error

assert 2 + 2 == 5, "Math error!"  # This will raise an AssertionError with the message
```

When the condition evaluates to False, the AssertionError is raised.

3. **try**: Write the code that might cause an error.
4. **except**: Write the code that runs if an error happens.
5. **else**: (Optional) Write code that runs if no errors happen.
6. **finally**: (Optional) Write code that always runs, whether an error happened or not

Multithreading in Python

In Python, a **thread** is the smallest unit of a process that can be scheduled and executed by the operating system. Threads allow concurrent execution of multiple operations within a single process, meaning you can run multiple tasks at the same time within a single Python program. In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process! A thread contains all this information in a **Thread Control Block (TCB).**

Python provides a built-in library called **threading** that allows you to create and manage threads.

Multiple threads can exist within one process where:

- Each thread contains its own **register set** and **local variables (stored in the stack)**.
- All threads of a process share **global variables (stored in heap)** and the **program code**.

How to create and run a thread in Python:

```
import threading

def print_numbers():
    for i in range(5):
        print(i)
```

```
# Create a thread
thread = threading.Thread(target=print_numbers)

thread.start()                          # Start the thread
thread.join()                           # Wait for the thread to finish
print("Thread has finished executing")
```

Multithreading:

Multithreading in Python is a technique that allows multiple threads to be executed concurrently.

In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed **context switching**. In context switching, the state of a thread is saved and the state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running parallelly (this is termed **multitasking**).

Multithreading Example 1:

```
import threading
def print_cube(num):
    print("Cube: {}" .format(num * num * num))

def print_square(num):
    print("Square: {}" .format(num * num))
```

```
if __name__ =="__main__":
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    print("Done!")
```

Multithreading Example 2:

Program demonstrating multithreading in Python using the `threading` module. The example will create two threads, each printing a message multiple times.

```
import threading
import time

# Function to be executed by each thread
def print_message(message):
    for _ in range(3):
        print(message)
        time.sleep(1)                    # Simulate a delay
```

```
# Create two threads
thread1 = threading.Thread(target=print_message, args=("Hello from Thread 1!",))
thread2 = threading.Thread(target=print_message, args=("Hello from Thread 2!",))

thread1.start()                 # Start both threads
thread2.start()
thread1.join()                  # Wait for both threads to complete
thread2.join()
print("Both threads have finished executing")
```

These notes are prepared by **Suhail Abass Hurrah** (S.P College Srinagar, batch 2019)
For any queries, you can email me at Hurrahsuhail1@gmail.com