Module 4

Module 4 - Object Oriented Programming

- Design with classes - Objects and Classes, Methods, Instance variables, Constructor, Accessor and Mutator,
- Data-Modeling Examples, Structuring classes with inheritance and polymorphism.
- Abstract classes, Interfaces, Exceptions - Handle a single exception, handle multiple exceptions

Define a Class in Python

- Primitive data structures—like numbers, strings, and lists—are designed to represent simple pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors, respectively.
- What if you want to represent something more complex?
- For example, let’s say you want to track employees in an organization.

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

A great way to make this type of code more manageable and more maintainable is to use **classes**.

How to Define a Class

- All class definitions start with the **class** keyword, which is **followed** by the **name of the class** and a **colon**.
- Any code that is indented below the class definition is considered part of the class’s body.
- Here’s an example of a Dog class:

```
class Dog:
    pass
```

How to Define a Class- Constructor

- The properties that all Dog objects must have are defined in a method called **.__init__()**.
- Every time a new Dog object is created, **.__init__() sets the initial state of the object** by assigning the values of the object's properties.
- That is, .__init__() initializes each new instance of the class.
- You can give .__init__() **any number of parameters**, but the **first parameter** will always be a variable called **self**.
- When a new class instance is created, the instance is automatically passed to the self parameter in .__init__() so that new attributes can be defined on the object.

How to Define a Class

- Let's update the Dog class with an .__init__() method that creates .name and .age attributes:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- In the body of .__init__(), there are two statements using the self variable:
  - self.name = name creates an attribute called name and assigns to it the value of the name parameter.
  - self.age = age creates an attribute called age and assigns to it the value of the age parameter.

class attributes

- Attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of .__init__().
- For example, the following Dog class has a class attribute called species with the value "Canis familiaris":

```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- They **must always be assigned an initial value**. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

Instantiate an Object in Python

- **Creating a new object** from a class is called **instantiating** an object.
- You can instantiate a new Dog object by typing the name of the class, followed by opening and closing parentheses:

```
Dog()
```

- The new Dog instance is located at a different memory address.
- That's because it's an entirely new instance and is completely unique from the first Dog object that you instantiated.

```
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

Class and Instance Attributes

```
>>> class Dog:
...     species = "Canis familiaris"
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
```

```
>>> buddy = Dog("Buddy", 9)
>>> miles = Dog("Miles", 4)
```

```
>>> Dog()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    Dog()
TypeError: __init__() missing 2 required positional arguments: 'name' and 'age'
```

Class and Instance Attributes

- After you create the Dog instances, you can access their instance attributes using **dot notation**:

```
>>> buddy.name
'Buddy'
>>> buddy.age
9

>>> miles.name
'Miles'
>>> miles.age
4
```

- You can access class attributes the same way:

```
>>> buddy.species
'Canis familiaris'
```

Class and Instance Attributes

- The values of the attributes can be changed dynamically

```
>>> buddy.age = 10
>>> buddy.age
10

>>> miles.species = "Felis silvestris"
>>> miles.species
'Felis silvestris'
```

Instance Methods

- Instance methods are functions that are defined inside a class and can only be called from an instance of that class.

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

This Dog class has two instance methods:

.description() returns a string displaying the name and age of the dog.
.speak() has one parameter called sound and returns a string containing the dog's name and the sound the dog makes.

Instance Methods

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

```
>>> miles = Dog("Miles", 4)

>>> miles.description()
'Miles is 4 years old'

>>> miles.speak("Woof Woof")
'Miles says Woof Woof'

>>> miles.speak("Bow Wow")
'Miles says Bow Wow'
```

Accessor and Mutator methods

- **Accessor Method:** This method is used to access the state of the object i.e, the data hidden in the object can be accessed from this method.
- However, this method cannot change the state of the object, it can only access the data hidden. We can name these methods with the word get.

- **Mutator Method**: This method is used to mutate/modify the state of an object i.e, it alters the hidden value of the data variable.
- It can set the value of a variable instantly to a new value. This method is also called as update method. Moreover, we can name these methods with the word set.

Accessor and Mutator methods

```
# Python program to illustrate the use of
# Accessor and Mutator methods

# Defining class Car
class Car:

        # Defining method init method with a parameter
        def __init__(self, carname):
                self.__make = carname

        # Defining Mutator Method
        def set_make(self, carname):
                self.__make = carname

        # Defining Accessor Method
        def get_make(self):
                return self.__make
```

Accessor and Mutator methods

```
# Creating an object
myCar = Car('Ford');

# Accesses the value of the variable
# using Accessor method and then
# prints it
print (myCar.get_make())

# Modifying the value of the variable
# using Mutator method
myCar.set_make('Porche')

# Prints the modified value
print (myCar.get_make())
```

Inheritance in Python

Inheritance

- The process of inheriting the properties of the parent class into a child class is called **inheritance**.
- The existing class is called a **base class or parent class** and the new class is called a **subclass or child class** or derived class.
- The main purpose of inheritance is the **reusability** of code because we can use the existing class to create a new class instead of creating it from scratch.

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

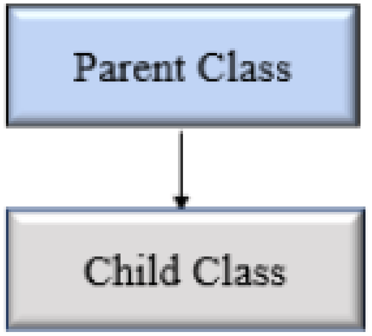
Types Of Inheritance

- In Python, based upon the number of child and parent classes involved, there are five types of inheritance. The type of inheritance are listed below:
  - Single inheritance
  - Multiple Inheritance
  - Multilevel inheritance
  - Hierarchical Inheritance
  - Hybrid Inheritance

Single Inheritance

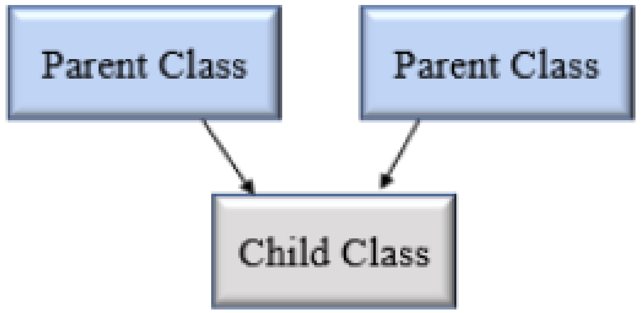
- In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class.

```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')
# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')
# Create object of Car
car = Car()
# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```



Multiple Inheritance

- In multiple inheritance, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.



Multiple Inheritance - Example

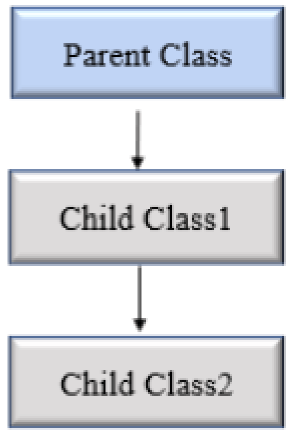
```python
# Parent class 1
class Person:
    def person_info(self, name, age):
        print('Inside Person class')
        print('Name:', name, 'Age:', age)

# Parent class 2
class Company:
    def company_info(self, company_name, location):
        print('Inside Company class')
        print('Name:', company_name, 'location:', location)

# Child class
class Employee(Person, Company):
    def Employee_info(self, salary, skill):
        print('Inside Employee class')
        print('Salary:', salary, 'Skill:', skill)
```

Multiple Inheritance – Example cont..

```python
# Create object of Employee
emp = Employee()

# access data
emp.person_info('Jessa', 28)
emp.company_info('Google', 'Atlanta')
emp.Employee_info(12000, 'Machine Learning')
```

Multilevel inheritance

- In multilevel inheritance, a class inherits from a child class or derived class.
- Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a **chain of classes** is called **multilevel inheritance**.



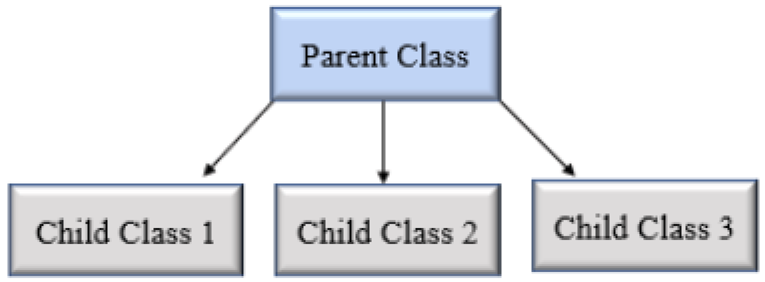
Multilevel inheritance - Example

```python
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Child class
class SportsCar(Car):
    def sports_car_info(self):
        print('Inside SportsCar class')

# Create object of SportsCar
s_car = SportsCar()
```

```python
# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

Hierarchical Inheritance

- In Hierarchical inheritance, more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes.

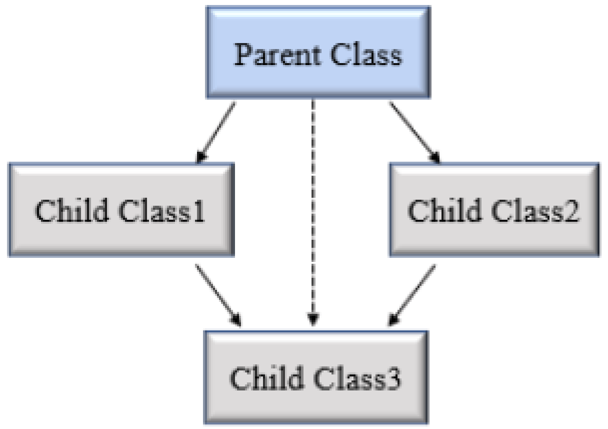


Hierarchical Inheritance - Example

```
class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)

class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)
car = Car()
car.info()
car.car_info('BMW’)

truck = Truck()
truck.info()
truck.truck_info(Tata')
```

Hybrid Inheritance

- When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.



Hybrid Inheritance - Example

```python
class Vehicle:
    def vehicle_info(self):
        print("Inside Vehicle class")

class Car(Vehicle):
    def car_info(self):
        print("Inside Car class")

class Truck(Vehicle):
    def truck_info(self):
        print("Inside Truck class")

# Sports Car can inherits properties of Vehicle and Car
class SportsCar(Car, Vehicle):
    def sports_car_info(self):
        print("Inside SportsCar class")
```

```python
# create object
s_car = SportsCar()

s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

Python super() function

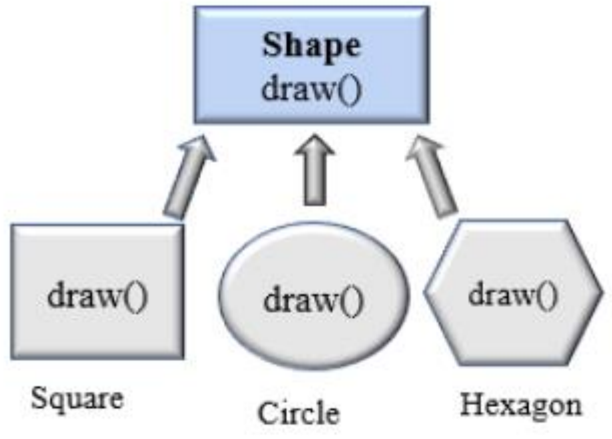
- In child class, we can refer to parent class by using the super() function.
- Benefits of using the super() function.
  - We are not required to remember or specify the parent class name to access its methods.
  - We can use the super() function in both single and multiple inheritances.
  - The super() function support code reusability as there is no need to write the entire function

Python super() function - Example

```
class Company:
    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self):
        # Calling the superclass method using 
super()function
        c_name = super().company_name()
        print("Jessa works at", c_name)

# Creating object of child class
emp = Employee()
emp.info()
```

Method Overriding

- In inheritance, all members available in the parent class are by default available in the child class.
- If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class.
- This concept is called **method overriding**.
- When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to **override** the method in the parent class.



Method Overriding - Example

```
# Function Overriding
class Employee:
    def WorkingHrs(self):
        self.hrs = 50

    def printHrs(self):
        print("Total Working Hrs : ", self.hrs)


class Trainee(Employee):
    def WorkingHrs(self):
        self.hrs = 60

    def resetHrs(self):
        super().WorkingHrs()
```

```
employee = Employee()
employee.WorkingHrs()
employee.printHrs()

trainee=Trainee()
trainee.WorkingHrs()
trainee.printHrs()

# Reset Trainee Hrs
trainee.resetHrs()
trainee.printHrs()
```

Exercises

- Write a Python class that has two methods: get_String and print_String , get_String accept a string from the user and print_String prints the string in upper case

Function Overloading in Python

- Function overloading is the feature when multiple functions have the same name, but the number of parameters in the functions varies. Python does not support function overloading as in other languages, and the functional parameters do not have a data type

```
class sumClass:
    def sum(self, a, b):
        print("First method:",a+b)
    def sum(self, a, b, c):
        print("Second method:", a + b + c)

obj=sumClass()
obj.sum(19, 8, 77) #correct output
obj.sum(18, 20) #throws error
```

```
Second method: 104
Traceback (most recent call last):
  File "<string>", line 9, in <module>
TypeError: sum() missing 1 required
positional argument: 'c'
```

Function Overloading in Python

```
class SumClass:

    def sum(self,a=None,b=None,c=None):
        if a!=None and b!=None:
            print("Function with two arguments")
            print("Sum=",a+b)
        elif a!=None and b!=None and c!=None:
            print("Function with three arguments")
            print("Sum=", a+b+c)
        else:
            print("Provide more numbers")
obj = SumClass()
obj.sum(19, 8, 77)
obj.sum(18, 20)
```

Abstract Class in Python

- An abstract class can be considered as a **blueprint for other classes**.
- It allows you to create a set of methods that must be created within any child classes built from the abstract class.
- A class which **contains one or more abstract methods** is called an abstract class.
- An **abstract method** is a method that has a declaration but **does not have an implementation**.
- By default, Python does not provide abstract classes.
- Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is ABC.

Abstract Class in Python

```python
# Python program showing
# abstract base class work

from abc import ABC, abstractmethod

class Polygon(ABC):

    @abstractmethod
    def noofsides(self):
        pass

class Triangle(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 3 sides")

class Pentagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 5 sides")

class Hexagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 6 sides")

class Quadrilateral(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 4 sides")
```

Abstract Class in Python

```
# Driver code
R = Triangle()
R.noofsides()

K = Quadrilateral()
K.noofsides()

R = Pentagon()
R.noofsides()

K = Hexagon()
K.noofsides()
```

Abstract Class in Python – Exercise

Create an Abstract Base Class called Shape that include abstract methods area() and circumference(). Then derive two classes Circle and Rectangle from the Shape class and implement the area() and circumference() methods . Write a Python program to implement above concept.

Interfaces in Python

Interfaces in Python

- In python there is no separate keyword to create an interface.
- We can create interfaces by using abstract classes which have only abstract methods.

Interfaces in Python

```
from abc import ABC, abstractmethod
class Bank(ABC):
   @abstractmethod
   def balance_check(self):
       pass
   @abstractmethod
   def interest(self):
       pass
```

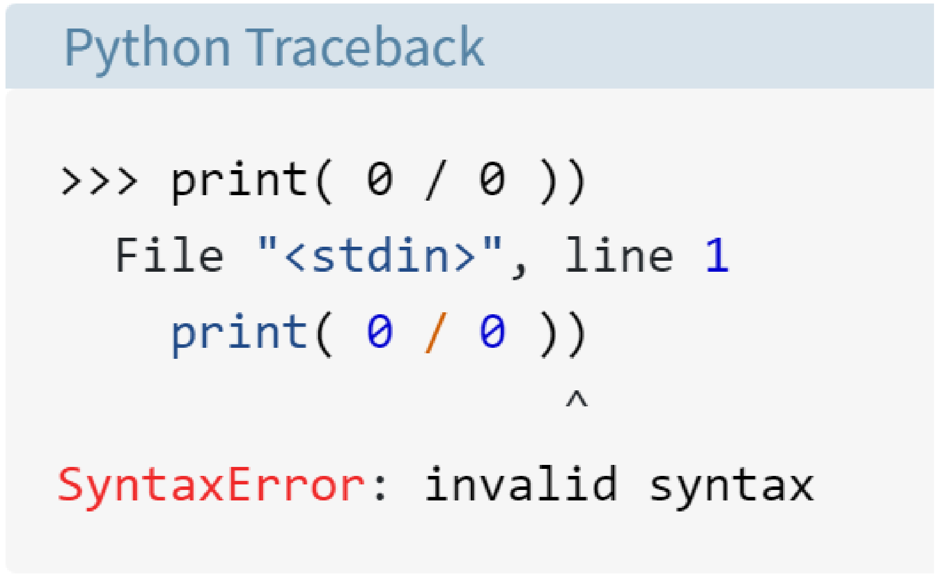
```
class SBI(Bank):
   def balance_check(self):
       print("Balance is 100 rupees")
   def interest(self):
       print("SBI interest is 5 rupees")
s = SBI()
s.balance_check()
s.interest()
```

Exceptions

Exceptions

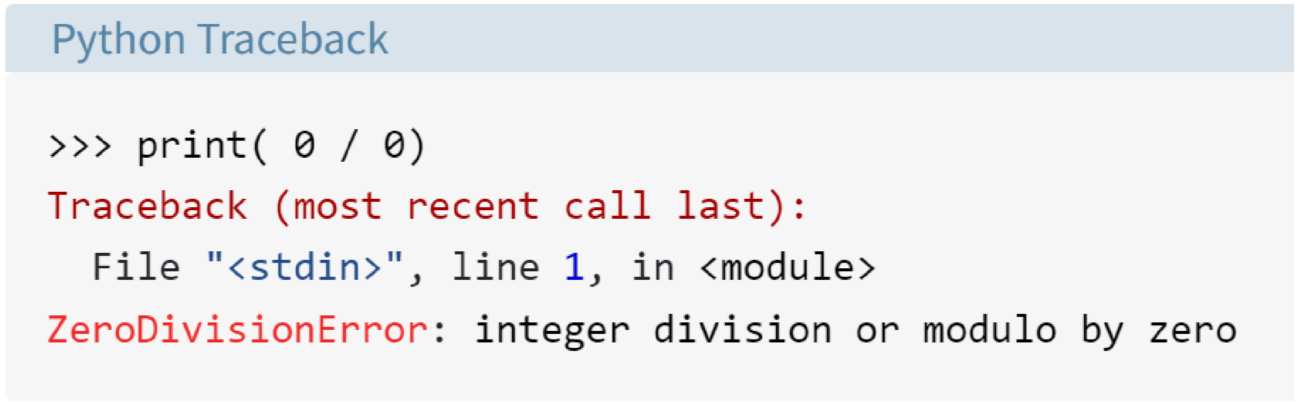
- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Exceptions versus Syntax Errors



Python Traceback

```
>>> print( 0 / 0 ))
  File "<stdin>", line 1
    print( 0 / 0 ))
                  ^
SyntaxError: invalid syntax
```

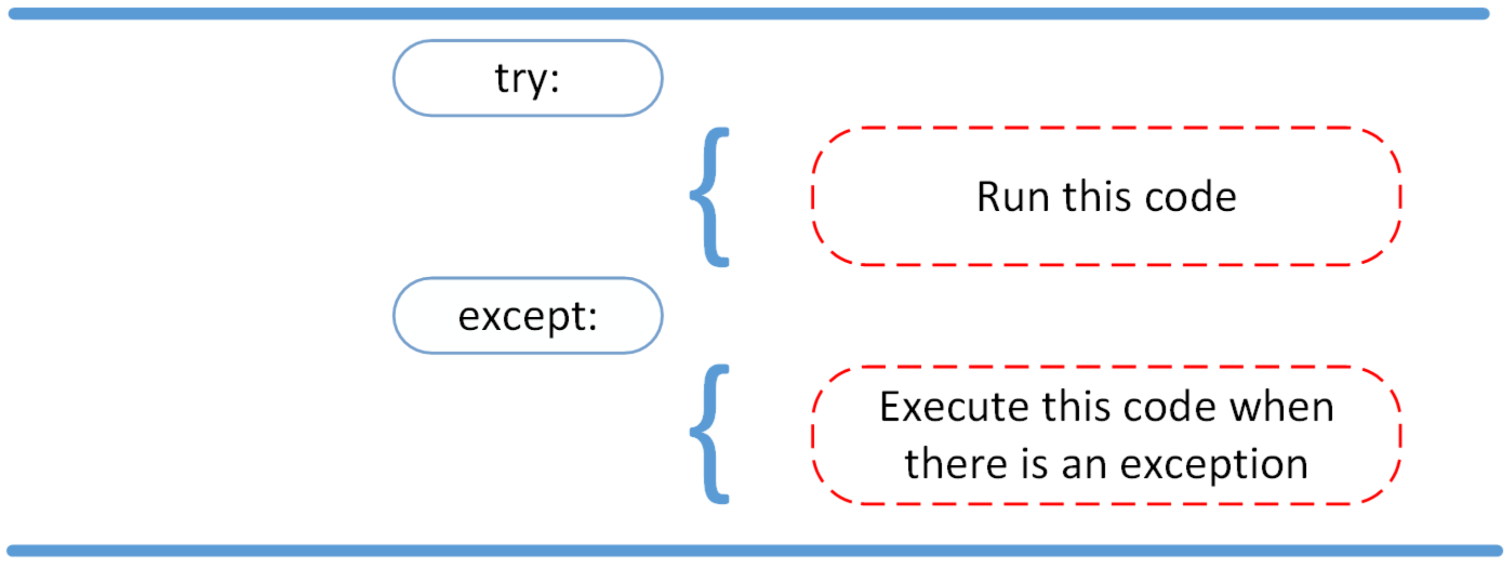


Python Traceback

```
>>> print( 0 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Try and Except Statement - Catching Exceptions

- In Python, we catch exceptions and handle them using try and except code blocks.
- The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception.



Try and Except Statement - Catching Exceptions

No exception, so the **try** clause will run.

```
# Python code to illustrate working of try()
def divide(x, y):
        try:
                # Floor Division : Gives only Fractional Part as Answer
                result = x // y
                print("Yeah ! Your answer is :", result)
        except ZeroDivisionError:
                print("Sorry ! You are dividing by zero ")

# Look at parameters and note the working of Program
divide(3, 2)
```

Try and Except Statement - Catching Exceptions

There is an exception so only **except** clause will run.

```
# Python code to illustrate working of try()
def divide(x, y):
        try:
                # Floor Division : Gives only Fractional Part as Answer
                result = x // y
                print("Yeah ! Your answer is :", result)
        except ZeroDivisionError:
                print("Sorry ! You are dividing by zero ")

# Look at parameters and note the working of Program
divide(3, 0)
```

Try and Except Statement - Catching Exceptions

The other way of writing except statement, is shown below and in this way, it only accepts exceptions that you’re meant to catch or you can check which error is occurring.

```
# Python code to illustrate working of try()
def divide(x, y):
      try:
            # Floor Division : Gives only Fractional Part as Answer
            result = x // y
            print("Yeah ! Your answer is :", result)
      except Exception as e:
            # By this way we can know about the type of error occurring
            print("The error is: ",e)
# Look at parameters and note the working of Program
divide(3, “CSE")
divide(3,0)
```

$\rightarrow$ The error is: unsupported operand type(s) for //: 'int' and 'str’

$\rightarrow$ The error is: integer division or modulo by zero

The except Clause with Multiple Exceptions

- You can also use the same except statement to handle multiple exceptions as follows

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

- Print one message if the try block raises a NameError and another for other errors:

The try-finally Clause

- You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

Exception handling with try, except, else, and finally

- **Try**: This block will test the excepted error to occur
- **Except**: Here you can handle the error
- **Else**: If there is no exception then this block will be executed
- **Finally**: Finally block always gets executed either exception is generated or not

```
try:
       # Some Code....

except:
       # optional block
       # Handling of exception (if
required)

else:
       # execute if no exception

finally:
      # Some code .....(always executed)
```

Exception handling with try, except, else, and finally

```python
# Python code to illustrate working of try()
def divide(x, y):
        try:
                # Floor Division : Gives only Fractional Part as Answer
                result = x // y
        except ZeroDivisionError:
                print("Sorry ! You are dividing by zero ")
        else:
                print("Yeah ! Your answer is :", result)
        finally:
                # this block is always executed regardless of exception generation.
                print('This is always executed')

# Look at parameters and note the working of Program
divide(3, 2)
divide(3, 0)
```

References

- https://pynative.com/python-inheritance/
- https://www.scaler.com/topics/function-overloading-in-python/