Object-Oriented Programming (OOPS-3)

Abstract Classes

An abstract class can be considered as a blueprint for other classes. Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that has a declaration but does not have an implementation. This set of methods must be created within any child classes which inherit from the abstract class. *A class that contains one or more abstract methods is called an* ***abstract class****.*

Creating Abstract Classes in Python

- 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.**
- **abc** works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base.
- A method becomes abstract when decorated with the keyword `@abstractmethod`.
- You are required to import **ABC** superclass and `abtractmethod` from the **abc** module before declaring your abstract class.
- An abstract class cannot be directly instantiated i.e. we cannot create an object of the abstract class.
- However, the subclasses of an abstract class that have definitions for all the abstract methods declared in the abstract class, can be instantiated.
- While declaring abstract methods in the class, it is not mandatory to use the `@abstractmethod` decorator (i.e it would not throw an exception). However, it is considered a good practice to use it as it notifies the compiler that the user has defined an abstract method.

The given Python code uses the **ABC** class and defines an abstract base class:

```
from abc import ABC, abstractmethod #Importing the ABC Module

class AbstractClass(ABC):

    def __init__(self, value): #Class Constructor
        self.value = value
        super().__init__()

    @abstractmethod
    def do_something(self): #Our abstract method declaration
        pass
```

Note:

- You are required to define (implement) all the abstract methods declared in an Abstract class, in all its subclasses to be able to instantiate the subclass.

For example, We will now define a subclass using the previously defined abstract class. You will notice that since we haven't implemented the `do_something` method, in this subclass, we will get an exception.

```
class TestClass(AbstractClass):
    pass #No definition for do_something method
x = TestClass(4)
```

We will get the output as:

```
TypeError: Can’t instantiate abstract class TestClass with abstract
methods do_something
```

We will do it the correct way in the following example, in which we define two classes inheriting from our abstract class:

```
class add(AbstractClass):
    def do_something(self):
        return self.value + 42

class mul(AbstractClass):
    def do_something(self):
        return self.value * 42

x = add(10)
y = mul(10)

print(x.do_something())
print(y.do_something())
```

We get the output as:

```
52
420
```

Thus, we can observe that a class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

Note: Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

- An abstract method can have an implementation in the abstract class.
- However, even if they are implemented, this implementation shall be overridden in the subclasses.
- If you wish to invoke the method definition from the abstract superclass, the abstract method can be invoked with `super()` call mechanism. (*Similar to cases of "normal" inheritance*).

- Similarly, we can even have concrete methods in the abstract class that can be invoked using `super()` call. Since these methods are not abstract it is not necessary to provide their implementation in the subclasses.
- Consider the given example:

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    @abstractmethod
    def do_something(self): #Abstract Method
        print("Abstract Class AbstractMethod")

    def do_something2(self): #Concrete Method
        print("Abstract Class ConcreteMethod")

class AnotherSubclass(AbstractClass):
    def do_something(self):
        #Invoking the Abstract method from super class
        super().do_something()

    #No concrete method implementation in subclass
x = AnotherSubclass()
x.do_something() #Calling abstract method
x.do_something2() #Calling concrete method
```

We will get the output as:

```
Abstract Class AbstractMethod
Abstract Class ConcreteMethod
```

Another Example

The given code shows another implementation of an abstract class.

```
# Python program showing how an abstract class works
from abc import ABC, abstractmethod
class Animal(ABC): #Abstract Class
    @abstractmethod
    def move(self):
        pass

class Human(Animal): #Subclass 1
    def move(self):
        print("I can walk and run")

class Snake(Animal): #Subclass 2
    def move(self):
        print("I can crawl")

class Dog(Animal): #Subclass 3
    def move(self):
        print("I can bark")

# Driver code
R = Human()
R.move()
K = Snake()
K.move()
R = Dog()
R.move()
```

We will get the output as:

```
I can walk and run
I can crawl
I can bark
```

Exception Handling

Error in Python can be of two types i.e. Syntax errors and Exceptions.

- Errors are the problems in a program due to which the program will stop the execution.
- On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Difference between Syntax Errors and Exceptions

Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

Example:

Consider the given code snippet:

```
amount = 10000
if(amount>2999)
    print("Something")
```

We will get the output as:

```
SyntaxError: invalid syntax
```

The syntax error is because there should be a **":"** at the end of an **if** statement. Since it is not present, it gives a syntax error.

Exceptions: Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

```
marks = 10000
a = marks / 0
print(a)
```

Output:

```
ZeroDivisionError: division by zero
```

The above example raised the **ZeroDivisionError** exception, as we are trying to divide a number by 0 which is not defined.

Exceptions in Python

- Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
- When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled.
- If not handled, the program will crash.
- For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is displayed and the program comes to a sudden unexpected halt.

Some Common Exceptions

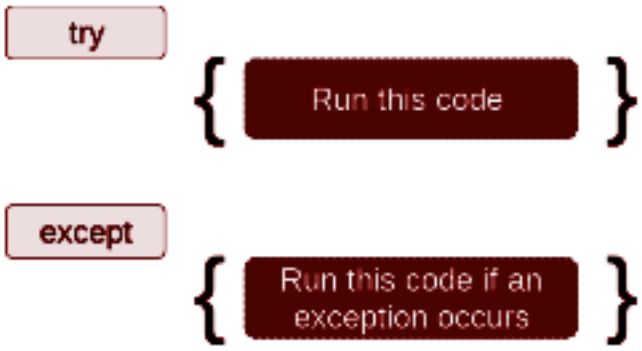
A list of common exceptions that can be thrown from a standard Python program is given below.

- **ZeroDivisionError:** This occurs when a number is divided by zero.
- **NameError:** It occurs when a *name* is not found. It may be local or global.
- **IndentationError:** It occurs when incorrect indentation is given.
- **IOError:** It occurs when an Input-Output operation fails.
- **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

Catching Exceptions

In Python, exceptions can be handled using `try-except` blocks.

- If the Python program contains suspicious code that may throw the exception, we must place that code in the `try` block.
- The `try` block must be followed by the `except` statement, which contains a block of code that will be executed in case there is some exception in the `try` block.
- We can thus choose what operations to perform once we have caught the exception.



- Here is a simple example:

```
# import module sys to get the type of exception
import sys
l = ['a', 0, 2]
for ele in l:
    try:#This block might raise an exception while executing
        print("The entry is", ele)
        r = 1/int(ele)
        break
    except:#This block executes in case of an exception in "try"
        print("Oops!", sys.exc_info()[0], "occurred.")
        print()

print("The reciprocal of", ele, "is", r)
```

We get the output to this code as:

```
The entry is a
Oops! <class 'ValueError'> occurred.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.

The entry is 2
The reciprocal of 2 is 0.5
```

- In this program, we loop through the values of a list `l`.
- As previously mentioned, the portion that can cause an exception is placed inside the try block.
- If no exception occurs, the except block is skipped and normal flow continues(for last value).
- But if any exception occurs, it is caught by the except block (first and second values).
- Here, we print the name of the exception using the `exc_info()` function inside `sys` module.
- We can see that element “a” causes ValueError and 0 causes ZeroDivisionError.

Every exception in Python inherits from the base **Exception** class. Thus we can write the above code as:

```
l = ['a', 0, 2]
for ele in l:
    try:
        print("The entry is", ele)
        r = 1/int(ele)
    except Exception as e: #Using Exception class
        print("Oops!", e.__class__, "occurred.")
        print("Next entry.")
        print()
```

```
print("The reciprocal of", ele, "is", r)
```

This program has the same output as the above program.

Catching Specific Exceptions in Python

- In the above example, we did not mention any specific exception in the `except` clause.
- This is not a good programming practice as it will catch all exceptions and handle every case in the same way.
- We can specify which exceptions an `except` clause should catch.
- A try clause can have any number of `except` clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
- You can use multiple `except` blocks for different types of exceptions.
- We can even use a tuple of values to specify multiple exceptions in an `except` clause. Here is an example to understand this better:

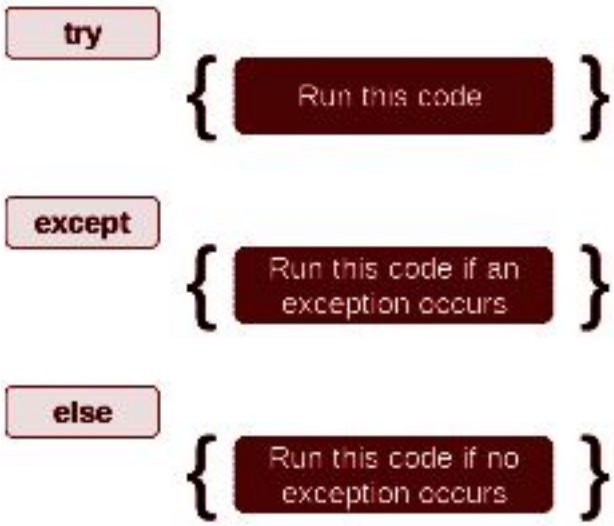
```
try:
    a=10/0;
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
else:
    print("Successfully Done")
```

Output:

```
Arithmetic Exception
```

`try-except-else` Statements

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the `else` block. The syntax is given below:



Consider the example code to understand this better:

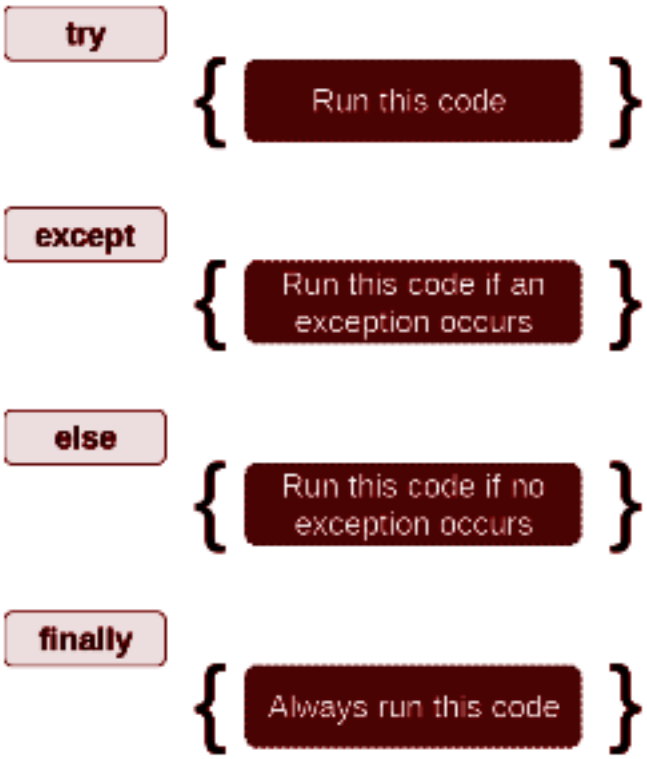
```
try:
    c = 2/1
except Exception as e:
    print("can't divide by zero")
    print(e)
else:
    print("Hi I am else block")
```

Output:

```
Hi I am else block
```

We get this output because there is no exception in the try block and hence the else block is executed. If there was an exception in the `try` block, the `else` block will be skipped and `except` block will be executed.

`finally` Statement



The `try` statement in Python can have an optional `finally` clause. This clause is executed no matter what and is generally used to release external resources. Here is an example of file operations to illustrate this:

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.

Raising Exceptions in Python

In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the `raise` keyword. We can optionally pass values to the exception to clarify why that exception was raised. Given below are some examples to help you understand this better

```
>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt
```

```
>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument
```

Now, consider the given code snippet:

```
try:
    a = -2
    if a <= 0:
        raise ValueError("That is not a positive number!")
except ValueError as ve:
    print(ve)
```

The output of the above code will be:

```
That is not a positive number!
```