Unit - 5

Object Oriented Programming (OOP)

- This is a programming paradigm(a programming idea) that uses Classes and Objects

- OOPs have the following concepts you have to get around:
    - Classes
    - Objects
    - Polymorphism
    - Encapsulation
    - Inheritance
    - Data Abstraction

A Class

- A class is basically a collection of functions and variables. Some functions inside a class are special (constructors) and do special things.
- A class is a blueprint for creating an object
- An object is an instance of a class. An object has all the functions and methods of the class it is an instance of.
- This might all sound like gibberish but you'd understand it better when you see code.

An Object

- An object is an instance of a class. A class can have any number of objects that are built on it. As told before, A class is a blueprint of a object. You can use this blueprint to build any number of objects.

Q. Create a class called `fruit` having at least one method and at least one feature and create an object of type `fruit`. Invoke your method.

Q. Write a python program to create a class called employee having a minimum of two class variables and a minimum of one class function.

The Constructor function

- The `__init__(self)` function inside a class is called a constructor function.
- This function is run every time an object of that class is created.

Inheritance

- It's a concept in OOP that allows you to inherit methods of one class in another class.
- For example:

PYTHON

```
class A:
    def fun1(self):
        print("fun1 class A")
    def fun2(self):
        print("fun2 class A")


class B(A):
    def fun3(self):
        print("fun3 class B")


obj = B()

obj.fun3()
obj.fun1()
obj.fun2()

# Output:
# fun3 class B
# fun1 class A
# fun2 class A
```

- Here although `fun1` and `fun2` are part of `class A`, we can access it from `class B`. This is because class B is a *child class* or a *subclass* of class A. class A is the *parent class* or a *superclass*
- You can inherit multiple parent classes as well

Polymorphism

- Consider the same program as above but I add a `fun1` inside class B, and add a class C

PYTHON

```
class A:
    def fun1(self):
        print("fun1 class A")
    def fun2(self):
        print("fun2 class A")


class B(A):
    def fun3(self):
        print("fun3 class B")
    def fun1(self):
        print("fun1 class B")

class C(A):
    def fun4(self):
        print("fun4 class C")

objB = B()
```

```
objA = A()
objC = C()



objB.fun3()
objB.fun1()
objB.fun2()

objA.fun1()



objC.fun1()
objC.fun4()

# Output:
# fun3 class B
# fun1 class B
# fun2 class A
# fun1 class A
# fun1 class A
# fun4 class C
```

- Here, when you call `objA.fun1()` although there is a `fun1()` in class A, there is a `fun1()` in class B as well. When `objB.fun1()` is called, it will call the `fun1()` inside class B itself.
- When we call from class C, there is no `fun1` present in class C, so it looks for a `fun1` in class A.
- The whole point of this is that although objB and objC are child classes of objA, they behave differently when fun1 is called.
- *Morphism* means 'behaviour' and *poly* means 'many' => So polymorphism means *many-behaviours*
- Calling the same fun1 generated different behaviours, and this is called *polymorphism*

Composition

Consider the below code:

PYTHON

```
class A:
    def fun1(self):
        print("fun1 class A")
class B(A):
    def __init__(self):
        self.objA = A()
    def fun2(self):
        print("fun2 class B")
        self.objA.fun1()



objB = B()
objB.fun2()

# Output:
```

```
# fun2 class B
# fun1 class A
```

- This allows us to call methods of A inside B. This is different from inheritance because in inheritance, we can only call methods of class A during a function call. We did not have a statement that we can execute to call functions of class A *inside* class B
- Here, class B *had a* class A instance inside it; and in inheritance, we had class A as a *parent class* of class B.
- Because of this reason: *Composition is called a **has-a relationship** and Inheritance is called a **is-a relationship***
  - class B **has a** class A instance
  - class B **is a** class A child

Private Methods/Variables

- Objects that you name starting with a *dunder(`__`)* are private.
- 'Objects' includes functions and variables
- This just means that you can not access these methods from outside the class, but you can inside the class.
- You might have observed that the `__init__()` method qualifies as a private method and you cannot access that as a normal function from the outside of that class
- For example,

PYTHON

```
class Car:
    def accelerate(self): # normal method
        print("accelerating")
    def __tellSpeed(self): # private method
        print("100Kmph")
    def callTellSpeed(self): # normal method
        self.__tellSpeed() # will work as called from inside the class


ciaz = Car()
ciaz.accelerate() # works
try:
    ciaz.__tellSpeed() # will not work; on to except block
except:
    print("__tellSpeed failed") # executed

ciaz.callTellSpeed() # will work


# output
# accelerating
# __tellSpeed failed
# 100Kmph
```

- This is the same for variables as well. You cannot access a variable starting with `__` from the outside of a class.

- This is useful when you want to hide some information from the outside of a class.

Name Mangling

- When you do try to access a private method from outside, you get an error like this:

```python
class Car:
    def accelerate(self): # normal method
        print("accelerating")
    def __tellSpeed(self): # private methor
        print("100Kmph")
    def callTellSpeed(self): # normal method
        self.__tellSpeed() # will work as called from inside the class


ciaz = Car()

ciaz.__tellSpeed()

# output
# Traceback (most recent call last):
#   File "/Users/anuragrao/test.py", line 12, in <module>
#     ciaz.__tellSpeed()
#     ^^^^^^^^^^^^^^^^^
# AttributeError: 'Car' object has no attribute '__tellSpeed'. Did you mean:
'_Car__tellSpeed'?
```

Observe how it asks you if you mean `_Car__tellSpeed`
This is called *name mangling*

- All of the private methods are renamed to this syntax: `_<class name>__<privateMethod>`
- So here, `__tellSpeed` became `_Car__tellSpeed`
- This is how private methods are implemented under the hood of python - It just renames those methods implicitly.
- This however *does not mean* that you'll need to use this new name **INSIDE** the class
- But, you can use this new name to kinda fool python and still access a private method:

```python
class Car:
    def accelerate(self): # normal method
        print("accelerating")
    def __tellSpeed(self): # private method
        print("100Kmph")
    def callTellSpeed(self): # normal method
        self.__tellSpeed() # will work as called from inside the class


ciaz = Car()

ciaz._Car__tellSpeed()
```

```
# output
# 100 Kmph
```

The `super()` Method

- The `super()` method is used whenever you want to access the methods of a parent class forcefully inside a child class
- The Python super() function returns objects represented in the parent's class and is very useful in multiple and multilevel inheritances to find which class the child class is extending first.
- For example:

PYTHON

```
class Car:
    def __init__(self):
        self.topSpeed = 100

    def printSelf(self):
        print("car")


class Ciaz(Car):
    def __init__(self):
        self.topSpeed = 220

    def printSelf(self):
        print("ciaz")

    def callParentPrintSelf(self):
        super().printSelf() # skips it's own and calls the parent class's method


ciaz = Ciaz()
ciaz.printSelf() # calls the method of ciaz
ciaz.callParentPrintSelf() # calls the method of car (parent)

# output
# ciaz
# car
```

The Method Resolution Order (MRO)

- Look at this code:

PYTHON

```
class A:
    def test(self):
        print("test of A called")

class B(A):
```

```
    def test(self):
        print("test of B called")
        super().test()

class C(A):
    def test(self):
        print("test of C called")
        super().test()

class D(B,C):
    def test2(self):
        print("test of D called")

obj = D()
obj.test()
```

By intuition, and just by following the concepts, you'd expect the output to be:

```shell
test of B called
test of A called
```

But this *does not* happen. Because Python weird 🙂🔫
Instead, you get this output:

```shell
test of B called
test of C called
test of A called
```

- The `D` class is derived from both `B` and `C`.
- When `obj.test()` is called, the method `test` in `B` is executed first which calls `super().test()` which resolves to the `test` method in `C`.
- The `test` method in `C` is then executed, which calls `super().test()` which resolves to the `test` method in `A`.
- Finally, the `test` method in `A` is executed, printing `test of A called`.
  Why is the test method in C called when the parent of B is A
  The method resolution order (MRO) of a class is determined by the C3 linearisation algorithm, which determines the order in which parent classes are searched for a method.

In the code, `D` is derived from `B` and `C`, and `B` is derived from `A`. The MRO of `D` is `[D, B, C, A]`, meaning that when the `test` method is called on an instance of `D`, the interpreter searches for the method in the following order:

1. `D`
2. `B`
3. `C`
4. `A`

Since the method `test` is defined in `B`, it is found and executed. This calls `super().test()`, which resolves to the next class in the MRO, `C`, and the `test` method defined in `C` is executed, which calls `super().test()` and resolves to the `test` method defined in `A`.

Exception Handling

- You might encounter statements of code that are prone to raising an error in certain conditions.
- You might be trying to connect to a database, it might not always succeed. You want your code to handle this exception instead of just throwing a run time error.
- There are different types of exceptions like ZeroDivision, FileNotFound, NameError, etc.
- *You can use a `try` and `except` block to handle these exceptions.*
- Example:

```python
print("abc") # executes successfully
print(a) # raises a NameError because 'a' is not defined

# output:
# abc
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# NameError: name 'a' is not defined
```

To handle this, you can put `print(a)` inside a `try` block.
Do note that here we are absolutely sure `print(a)` will raise an error, but in real code, this is not the case. You just suspect a statement to be problematic.

```python
print("abc") # executes successfully
try:
    print(a) # raises an error
except NameError:
    print("Name Error Raised") # This is executed
except IndexError:
    print("Index Error Raised")

# output:
# abc
# Name Error Raised
```

- Another Example:

```python
l = [1,2,3] # executes successfully
try:
```

```
    print(l[4]) # raises an error
except NameError:
    print("Name Error Raised")
except IndexError:
    print("Index Error Raised") # This is executed

# output:
# Index Error Raised
```

- When an exception is encountered, an object is encountered implicitly. This object is matched with every except block. If the `class` in front of the `except` block is matched, that except block is executed.

- If there is no error raised in the statements inside the `try` block, The `except` block is never run.

- Note that when there are multiple exceptions, the first matched `except` is executed.
  Example:

PYTHON
```
l = [1,2,3] # executes successfully
try:
    print(l[4]) # raises an error
except Exception:
    print("Generic Exception Raised") # This is executed
except IndexError:
    print("Index Error Raised")

# output:
# Generic Exception Raised
```

- Now if you put the `except IndexError` before `exception Exception`:

PYTHON
```
l = [1,2,3] # executes successfully
try:
    print(l[4]) # raises an error

except IndexError:
    print("Index Error Raised") # This is executed
except Exception:
    print("Generic Exception Raised")


# output:
# Index Error Raised
```

- You can also have an optional `else` block for a `try` block.
  This code is pretty self explanatory:

```python
try:
    print("executed")
except Exception:
    print("executed when something goes wrong")
else:
    print("executed when nothing goes wrong")

# output
# executed
# executed when nothing goes wrong
```

- Also note that, if there is no matching `except` block, the error gets raised just like normal - as if there was no try and except block.

The `raise` keyword

- You can raise your own exception when a certain condition is met.
- *Explicitly* raising an error in a *forceful* way by the programmer (keyword mumbo jumbo given in the slides, just keep these in mind)
- Syntax: `raise <ExceptionName>('<ExceptionMessage>')`
- Example: `raise Exception("MyError Raised!")`
- Code:

```python
def fun(num):
    raise ZeroDivisionError
try:
    fun(3)
except ArithmeticError:
    print("error")

print("terminated")

# output
# error
# terminated
```

- Here the function raises a `ZeroDivisionError` and so the `ArithmeticError` except block is triggered since `ArithmeticError` *is a parent class* of `ZeroDivisionError` and then finally `terminated` is printed - outside the try and except block.

Raising Your Own Exceptions

You can raise your own exceptions along with a message when needed like this:

```python
a = 10
b = 10
if a == b:
    raise Exception("a is equal to b!")
```

```
# output
# Traceback (most recent call last):
#   File "<stdin>", line 2, in <module>
# Exception: a is equal to b!
```

The `finally` keyword

- You can use the `finally` keyword to execute something irrespective of if an exception is raised or not:

PYTHON

```
try:
    print("executed")
except:
    print("executed if something went wrong")
else:
    print("executed if nothing went wrong")
finally:
    print("executed irrespective of if an error was raised or not")


# output
# executed
# executed if nothing went wrong
# executed irrespective of if an error was raised or not
```

Why use the `finally` block when you can just put the code outside the `try and except` block?

- You can put the code outside of the try-except block, but it may not be executed if an exception is raised and not handled within the try block. In such cases, the program may terminate before reaching the code that is outside the try-except block.
- Using a finally block guarantees that the code in that block will be executed regardless of whether an exception was raised or not, which can make it more robust and less prone to bugs. Also, having the code that needs to be executed no matter what happens inside try-catch block makes the code more readable and easy to understand.
- Example:

PYTHON

```
try:
    raise NameError
except ZeroDivisionError:
    print("executed if something went wrong")
else:
    print("executed if nothing went wrong")
finally:
    print("executed irrespective of if an error was raised or not")


# output
# executed irrespective of if an error was raised or not
```

```
# Traceback (most recent call last):
#  File "/Users/anuragrao/realTest.py", line 2, in <module>
#    raise NameError
# NameError
```

- This is useful when you have to release resources: like close a database connection, or close a file, etc.

How do you make your own custom Exception?

- Make a class derived from the `Exception` class.
- Example:

PYTHON

```
class CustomException(Exception):
    def __init__(self, string ):
        self.string = string

    def __str__(self) -> str:
        return self.string

a = 10
b = 10

try:
    if a == b : raise CustomException("A is equal to B!")

except ZeroDivisionError:
    print("zero division error") # not executed

except CustomException as e:
    print("custom exception. Exception: ", e)

# output
# custom exception. Exception:  A is equal to B!
```

Exception Propagation

- This describes how an exception *travels* in your code - that doesn't make sense without an example:

PYTHON

```
def func4():
    func3()

def func3():
    func2()

def func2():
    func1()
```

```
def func1():
    print(100/0) # should produce ZeroDivision error

func4()
```

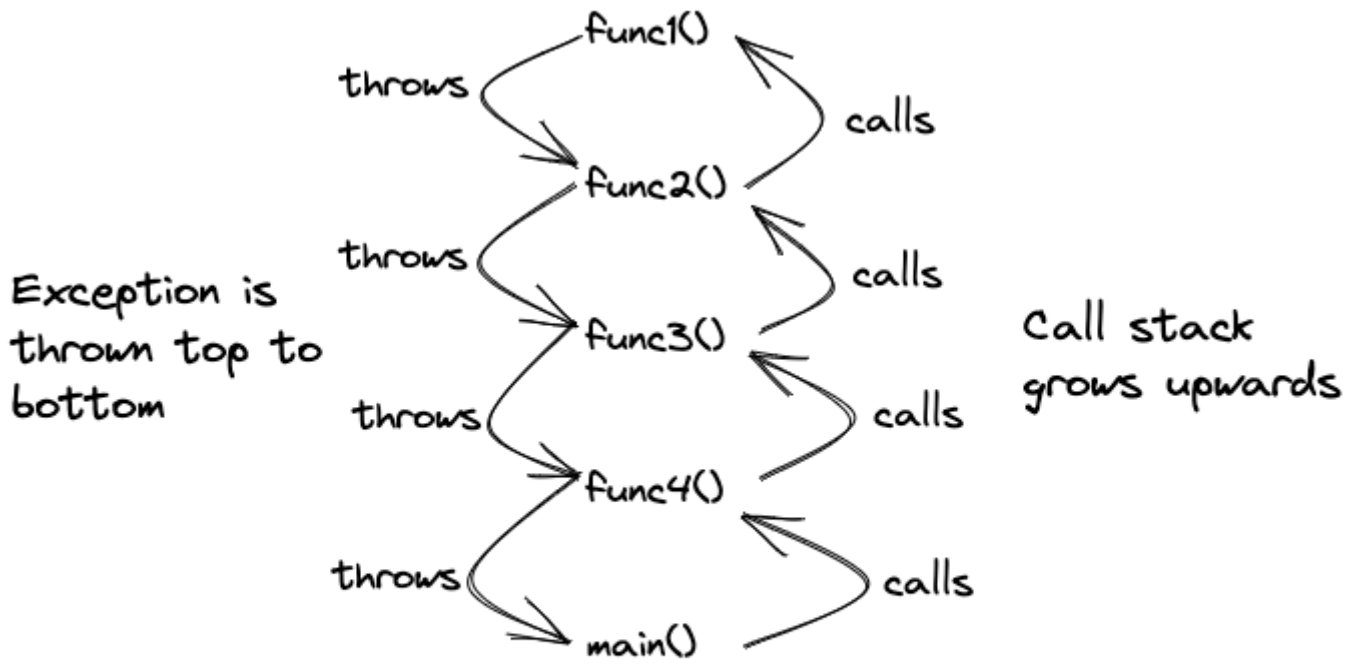
This produces:

SHELL

```
Traceback (most recent call last):
  File "/Users/anuragrao/test.py", line 20, in <module>
    func4()
  File "/Users/anuragrao/test.py", line 3, in func4
    func3()
  File "/Users/anuragrao/test.py", line 6, in func3
    func2()
  File "/Users/anuragrao/test.py", line 9, in func2
    func1()
  File "/Users/anuragrao/test.py", line 12, in func1
    print(100/0) # should produce ZeroDivision error
          ~~~^~
ZeroDivisionError: division by zero
```

Observe the `Traceback (most recent call last)` and the stuff below it, That describes the call stack.

- `func4()` *calls* `func3()` *calls* `func2()` *calls* `func1()` where the exception is generated
- But when an exception is thrown in `func1()` it travels the opposite way.
- `func1()` throws the exception at `func2()` which throws it at `func3()` and finally at `func4()`



CALL STACK

Made with ♡ by Anurag Rao
Find all the answers to the questions in this pdf at my GitHub Repo