Pillars of object-oriented programming (OOP)

TheMalteseSailor

Outline

1. Introduction
2. Inheritance
3. Encapsulation
4. Abstraction
5. Polymorphism

Introduction

Introduction

There are 4 pillars of object-oriented programming:

1. Inheritance
2. Encapsulation
3. Abstraction
4. Polymorphism

Introduction (vocabulary)

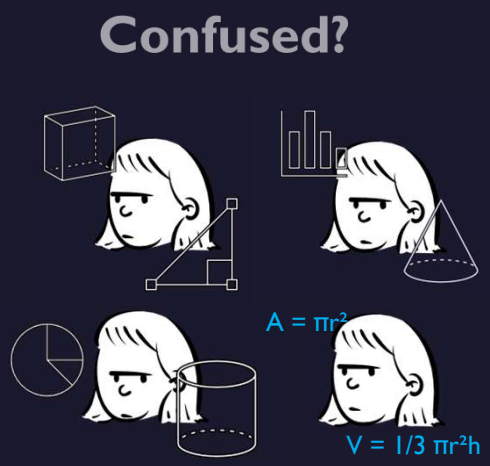
What is a class?

A class is a template that defines the attributes and abilities of a specific type of object.

What's the difference between a class and an object?

An object is a specific instance created using one or more class templates.

Introduction



Introduction (vocabulary)

Helpful way to view the terms:

Class : blueprint (ignoring static classes)

- Think of these like Lego©™®(etc) instructions. It has instructions to construct and object instance, but also contains all the instruction on how operate its functionality (methods).

Object : noun

- These are specific instances where the class/template was used to generate an object that you can interact with.

Function/Method : verb

- Methods are functions that belong to an object. These are actions/procedure that can be performed.

Attributes : adjectives

- These are characteristics of an object that typically is used to track state.

Introduction

Picturing it:

Imagine a conveyor belt in an assembly line where cheap plastic toys are manufactured. The assembly line includes the process of vacuum forming plastic and snapping the parts together before packaging it.

The conveyor belt is the underlying operating system assigning resources for the construction and operation of the object. The vacuum forming process emulates the template object paradigm. The vacuum components detail the shape and form of the object, but it itself is not one. The assembling is completed by the underlying operating system but can be directed using something called a constructor. Once the object instance is generated the operating system presents the caller with a method to access/reference the new object.

Introduction

Code Example:

On OOP_Game_lab.py:8-33 the scenario class is defined. Inside of this class definition you will see 'def __init__(self, name):'. The usage of the double underline is pronounced 'dunder'. This method definition, 'def', defines what an object created using this class 'template' will do right after it's created. This is not a constructor as it is not executed during object creation, but after the object is completed constructed.

The first argument is 'self' which is a reference to the current object. The second is name. This is how arguments are passed during object instantiation. This is seen on line OOP_Game_lab.py:48 (world = scenario(name)). The variable 'world' now contains a reference to an instance of the scenario class. On line 11 you see that the argument passed to '__init__' is used to define an attribute of that object. In this instance it's the self.hero_name variable.

In sequence, the class definition is created. Then from inside the execution flow an instance of the scenario class is needed to be created

Introduction

Now you can see later within the while loop (ln. 87) that the object 'world' is referenced as containing all the things seen inside the class definition earlier. A big difference is that the world object contains data resulting from the method calls seen in the class definition.

Notice in OOP_Game_lab.py:87-127, ln. 127 is a while loop nested inside another while loop. This is used so that the outer while loop handles the larger scenario details and the inner while loop handles user input and decisions.

Try to trace some what you see in these two while loops to what you see in the class definition at the top of the script file.

TAKE 5 MINUTES TO TRACE

Introduction

QUESTIONS?

Inheritance

Inheritance

Inheritance is the ability for on class to inherit or derive methods and properties from another class. Typically referenced as parent and child classes, respectively.

Inheritance in python can come in several basic forms: (There are more, but these are the basic ones)

1. **Single Inheritance** – child classes inherit from a single parent class
2. **Multiple Inheritance** – child classes inherit from multiple parent classes
3. **Multilevel Inheritance** – child classes inherit from classes that also inherit from their own parent classes.

Inheritance (Single Inheritance)

```
class parent():
        def meth1(self):
                print("parent derp.")
        def meth2(self):
                print("I'm the parent.")

class child(parent):
        def __init__(self):
                parent.__init__(self)
        def child_meth1(self):
                print("child derp.")
        def meth2(self):
                print("I'm the child.")
```

```
>> real_child = child()
>> real_child.meth1()
>> parent derp.
>> real_child.child_meth1()
>> child derp.
>> real_child.meth2()
>> I'm the child.
```

Inheritance (Single Inheritance w/ super())

```
parent():
        def meth1(self):
                print("parent derp.")
        def meth2(self):
                print("I'm the parent.")

class child(parent):
        def __init__(self):
                super.__init__(self)
        def child_meth1(self):
                print("child derp.")
        def meth2(self):
                print("I'm the child.")
```

```
>> real_child = child()
>> real_child.meth1()
>> parent derp.
>> real_child.child_meth1()
>> child derp.
>> real_child.meth2()
>> I'm the child.
```

'super' is a function that references the single base/parent class.

Inheritance (Multiple Inheritance)

```
class parent1():
        def p1Meth(self):
                print("p1Meth")
        def meth2(self):
                print("p1Meth2")

class parent2():
        def p2Meth(self):
                print("p2Meth")
        def meth2(self):
                print("p2Meth2")

class child(parent1,parent2):
        def __init__(self):
                parent1.__init__(self)
                parent2.__init__(Self)
        def child_meth1(self):
                print("child derp.")
        def meth2(self):
                print("c1Meth2")
```

```
>> kid = child()
>> kid.p1Meth()
>> p1Meth
>> kid.p2Meth()
>> p2Meth
>> kid.meth2()
>> c1Meth2
```

Inheritance (Multilevel Inheritance)

```
class grandparent():
        def __init__(self, gpname):
                self.gpname = gpname
        def get_gpname(self):
                print(self.gpname)

class parent(grandparent):
        def __init__(self, pname):
                grandparent.__init__(gpname)
                self.pname = pname
        def get_pname(self):
                print(self.pname)

class child(parent):
        def __init__(self, name, pname, gpname):
                parent.__init__(self, pname, gpname)
                self.name = name
        def get_name(self):
                print(self.name)
```

```
>> kid = child('Bill','Bob','Ben')
>> kid.get_gpname()
>> Ben
>> kid.get_pname()
>> Bob
>> kid.get_name()
>> Bill
```

Inheritance

Why would you utilize inheritance in your programming?

- Building a base class with generic functionality is largely used to reduce code duplication. It also makes extensible code easier to manage and generate.

Inheritance

QUESTIONS?

Encapsulation

Encapsulation

Encapsulation is the idea of moving object data manipulation methods and attributes outside of the reach of external calls. This is used to prevent unauthorized modifications to protected or critical object data.

Encapsulation

Python3 does have the concept of a private method or member, but it's different from C++/C#/JAVA/etc. This is because it's not truly a private method or member. The name is simply "mangled" by the interpreter by it adding the base class name to the method.

ex. object.__validate_account() becomes:

object. _<base class>__validate_account()

Historically the understood rule was that variables or methods prepended with a single underscore were considered for internal object operations and shouldn't be called.

Encapsulation

You will often see methods called getter and setter functions. The getter function facilitates a safe method of retrieving protected information from an object. Setter functions are a safe way to modify protected data without out giving an external caller direct access to that specific object property.

```
ex. balance = thing._get_account_ballance()
    thing._set_account_balance(balance + 1500)
```

Encapsulation

```
class SavingsAccount:
        def __init__(self,name,acct_num):
                self.name = name
                self._balance = 0
                self.account_number = acct_num

        def get_account_balance(self):
                print(self._balance)

        def set_account_balance(self, amt, calc_type):
                if not isinstance(amt, int):
                        return -1
                if calc_type == "add":
                        self._balance = self._balance + amt
                        self._save_account_details()
                        print(self._balance)
                elif calc_type == "sub":
                        self._balance = self._balance - amt
                        self._save_account_details()
                        print(self._balance)
                else:
                        return -2
        def _save_account_details(self):
                ''' saves to disk '''
```

```
>> Account = SavingsAccount("Guy",1245)
>> Account.get_account_balance()
>> 0
>> Account.set_account_balance(15,"add")
>> Account.get_account_balance()
>> 15
>> Account.set_account_balance(2,"sub")
>> Account.get_account_balance()
>> 13
```

Encapsulation

QUESTIONS?

Abstraction

Abstraction

Abstraction is the idea of hiding unnessiary complexity from the user. This allows a programmer to tackle complex operations, compartmentalize it, and expose it with a simple interface for further and expanded usage.

It is seen as a good practice to isolate functionality within its own function/method. This allows for modularity regardless of the complexity. For example, if you were to have a program that took user input, parsed the data, performed desired processing, and output the data. Each of these activities would be contained within its own function or functions. If certain elements are sufficiently complex it may make sense to break that functionality up into smaller pieces.

An example of extremely complex processing could be robust math calculations, hardware intense data parsing or processing, etc.

Abstraction

OOP_Game_lab.py:32 is inside a method called createHero() belonging to the scenario() class. This abstracts the process of generating the hero object by simply passing the name variable that's used by the object in its '__init__' method.

Moving over to utilz.py:121 you can see the Hero class definition referenced from within the scenario class. The work done from within this class is abstracted away from the caller. It simply needs to pass the name variable and it is completely ignorant of what's occurring to produce its expected result. Here you may notice that there are no "return"s explicitly expressed at the bottom of the methods. The return is implicit when there is no more code.

You can see how there are multiple layers of abstraction present here. Inside utils.py on line 128, within method definition of create_hero_json_object() the weapons and armor generation is abstraction by calling generic weapon and armor selection functions. These are also used when generating the dungeon enemies during the execution of scenario():__init__():instantiateWorld().

Abstraction (bringing it all together)

OOP_Game_lab.py

```
class scenario():
    def __init__(self,name):
        self.realm_name = "kingdom of derp"
        self.hero_name = name
        self.dungeon_boss_key_captured = False
        self.instantiateWorld()
        self.greatHero()

    def instantiateWorld(self):
        …

    def createHero(self):
        self.hero = utils.Hero(self.hero_name)
        return
```

Utils.py

```
class Hero():
    def __init__(self,name):
        self.name = name
        self.create_hero_json_object()
        self.write_hero_object_to_disk()

    def create_hero_json_object(self):
        weapons = get_weapon()
        armor = get_armor()
        hero_attribs = {
            "name": self.name,
            "health": 100,
            "armor": armor,
            "weapons": weapons,
            "boss_key": False
        }

    def write_hero_object_to_disk(self):
        serializedHero = json.dumps(self.hero_attribs, indent=4)
        with open("HeroObject.json", 'w') as _wf:
            _wf.write(serializedHero)
```

Utils.py

```
def get_weapon():
    weapons = {
        "short_sword": 15,
        "long_sword": 20,
        "dagger": 5,
        "spear": 20
    }
    item_number = random.randint(0,3)
    for index, value in enumerate(weapons.keys()):
        if index == item_number:
            return [value,weapons[value]]
```

Utils.py

```
def get_armor():
    armor = {
        "breast_plate": 10,
        "helmet": 5,
        "shield": 10,
        "greaves": 5,
        "gauntlets": 5
    }
    item_number = random.randint(0,4)
    for index, value in enumerate(armor.keys()):
        if index == item_number:
            return [value,armor[value]]
```

Abstraction

QUESTIONS?


Polymorphism

Polymorphism

Polymorphism means to have many forms. In programming it means something having the same name, but a different signature.

From a "C" perspective this means you can have many functions with the same name, but they must have a different signature. This means that you can have multiple functions called derp as long as the combined nature of their return type, argument count, and argument type(s) are unique. This is called function overloading. During compilation, the compiler recognizes them as totally different functions regardless of what the duplicate name the programmer has assigned it.

In python this can be see when utilizing the '+' symbol. When placing the plus sign between two integers it will perform a math operation. If you place the plus sign between two string variables it will concatenate the two.

Polymorphism

```
>> print(11223344) # 11223344 is of type int()
>> 11223344
>>
>> print('string',1122)
>> string 1122
>>
>> 'string'+1122
TypeError: Must be str, not int
>>
>> 'string'+str(1122)
>> string1122
>>
>> len("derp")
>> 4
>> len([0,1,2,3,4,5])
>> 6
>>
```

Abstraction

QUESTIONS?

Summary

Understanding the pillars of Object-Oriented Programming is critical to becoming anything beyond a basic programmer. It is also a foundational skill for many more advanced skills including hacking, reverse engineering, etc.

Thank You

TheMalteseSailor