Chapter 05 - When To Use Object Oriented Programming

April 4, 2021

0.1 Treating Objects as Objects

- give separate objects in your domain a special class in your code
- first identify objects in the problem and then model the data and behavior
- objects have both `data` and `behavior`
- if we are working with just data, we are better just storing it as a list, set, dictionary or other Python data structure
- if we are working with just behavior, we can just use a function

- generally, unless there is a reason too, avoid using a class
- if we are passing the same set of related variables to a set of functions, we might want to think about grouping the variables and functions into a class

- one of the benefits of object oriented code is that it is relatively self-documenting
- key point is still if you have multiple functions that take the same variables, then a `class` is apporpiate

[5]:

```python
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, p2):
        return math.sqrt((self.x - p2.x)**2 + (self.y - p2.y)**2)

class Polygon:
    def __init__(self):
        self.vertices = []

    def add_point(self, point):
        self.vertices.append((point))

    def perimeter(self):
        perimeter = 0
        points = self.vertices + [self.vertices[0]]
        for i in range(len(self.vertices)):
            perimeter += points[i].distance(points[i+1])
        return perimeter

square = Polygon()
square.add_point(Point(1, 1))
square.add_point(Point(1, 2))
square.perimeter()
```

```
[5]: 2.0
```

- notice in the class above, we could just have `distance` and `perimeter` as seprate functions
- but if there was another function, then the class definition becomes more apeasing

- also look for interaction between objects
- look for inheritance relationships
- inheritance is hard to model elegantly without classes

0.2 Adding Behaviors to Class Data With Properties

- in python, the distinction between behavior and data is blurry
- some languages like java want you to `set` and then `get` variables, instead of just having accessable variables

Java Version

```
[9]: class Color:
         def __init__(self, rgb_value, name):
             self._rgb_value = rgb_value
             self._name = name

         def set_name(self, name):
             self._name = name
         def get_name(self):
             return self._name

     c = Color("#ff0000", "bright red")
     print(c.get_name())
     c.set_name("red")
     c.get_name()
```

```
bright red
```

```
[9]: 'red'
```

Python Version

```
[10]: class Color:
          def __init__(self, rgb_value, name):
              self.rgb_value = rgb_value
              self.name = name

      c = Color("#ff000", "bright red")
```

```
print(c.name)
c.name = "red"
print(c.name)
```

```
bright red
red
```

- the python `property` keyword can make methods that `look` like attributes
- thus we can write our code to use direct memeber access

[19]:

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self._name = name

    def _set_name(self, name):
        if not name:
            raise Exception("Invalid Name")
        self._name = name

    def _get_name(self):
        return self._name

    name = property(_get_name, _set_name)
```

- in the code above, we first changed the name attribute into a semi-private `_name` attribute
- then we add two more semi-private methods to get and set that variable
- the `property` declaration create a new attribute on `Color` class called name to replace the direct name attribute
- it sets the attribute to be a **proprty**
- under the hood, `property` calls the two methods we just created whenever the value is accessed or changes
- this new version of the `Color` class can be used exactly the same way as the earlier version, yet it now performs validation when we set the `name` attribute

[20]:

```
c = Color("#000fff", 'bright red')
print(c.name)
```

```
bright red
```

[21]:

```
c.name = "red"
print(c.name)
```

```
red
```

[22]:

```
c.name = ""
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-22-d163a332e13e> in <module>
```

```
----> 1 c.name = ""

<ipython-input-19-2cafe111ebdb> in _set_name(self, name)
      6     def _set_name(self, name):
      7         if not name:
----> 8             raise Exception("Invalid Name")
      9         self._name = name
     10

Exception: Invalid Name
```

- the whole purposse of the `property` is to keep accessing the variables safe

0.3 Properties In Detail

- think of the property function as returning an object that proxies any request to set or access the attribute value through the methods we have specified
- the property built-in is like a constructor for such an object and that object is set as the public-facing memeber for the given attribute
- `property` constructor can also detect two additional arguments, a `delete` function and a docstring for the property
- the `delete` function is rarely used but can keep track of if a value has been deleted or not

[29]:

```python
class Bacteria:

    def _get_bacteria(self):
        print("You are getting silly")
        return self._bacteria

    def _set_bacteria(self, value):
        print(f"You are making bacteria {value}")
        self._bacteria = value

    def _del_bacteria(self):
        print("Woah, you killed bacteria")

    bacteria = property(_get_bacteria, _set_bacteria, _del_bacteria, "This is␣
 ↪bacteria property")

b = Bacteria()
b.bacteria = "Alpha"
print(b.bacteria)
del b.bacteria
```

```
You are making bacteria Alpha
You are getting silly
Alpha
Woah, you killed bacteria
```

0.4 Decorators - another way to create properties

- the `property` function can be used with the decorator syntax to turn a `get` function into a property function
- code below is the equilivant to `foo = property(foo)`

[34]:

```python
class Bacteria:
    @property
    def Bacteria(self):
        return "bar"

    @Bacteria.setter
    def Bacteria(self, value):
        self._Bacteria = value

    @Bacteria.deleter
    def Bacteria(self):
        print("Woah, you deleted foo")
        del self._Bacteria
```

- first we decorate the `foo` method as a getter
- then we decorate a second method with exactly the same name by applying the `setter` attribute of the orginally decorated `foo` method
- the `property` function returns an object; this object always comes with its own `setter` attribute, which can then be applied as a decorator to other functions
- usingg the same name for the get and set method is not required, but it does help to group the multiple methods that access one property

0.5 Deciding When To Use Properties

- when do you choose an attribute, a method or a property

- for property, there is a pretty common use case
- we have some data on a class that we later want to add behavior to
- functions and methods themselves are normal objects
- `methods` are just callable attributes and properties are just `customizable` attributes
- `methods` should represet action; things that can be done to or performed by the object
- if the attribute is not an action, we need to decide between data attributes and properties
- always use standard attribute untill you need to control access to that property in some way

- **the only difference between an attribute and a property is that we can invoke custom actions automatically when a property is retrieved, set, or deleted**

- an example is when you are caching
- the first time the value is retrieved, we perform the lookup calculation
- then we can locally cache the value as a private attribute on our object
- the next time the value is requested, we return the stored data

[43]:

```python
import time
from urllib.request import urlopen
```

```
class WebPage:
    def __init__(self, url):
        self.url = url
        self._content = None

    @property
    def content(self):
        if not self._content:
            print("Retriveing New Page...")
            print("")
        self._content = urlopen(self.url).read()
        return self._content

webpage = WebPage('https://www.google.com/')
now = time.time()
content1 = webpage.content

print('first time accesing url')
print(time.time() - now)
print("")
now = time.time()
content2 = webpage.content

print('second time accessing url')
print(time.time() - now)
print("")
print('content1 == content2')
print(content1 == content2)
```

```
Retriveing New Page…

first time accesing url
0.1132657527923584

second time accessing url
0.11429214477539062

content1 == content2
False
```

0.6 Manager Objects

- the attributes on a management class tend to refer to other objects that do the *visible* work
- the behavior on such a class delegate to those other classes at the right time and pass messages between them

- as an example we will write a program that find text, replaces it and then stores the file in a compressed zip file

- the manager object will just orchestrates the events
- for us, the manager will be responsible for the following
  - unzipping the compressed file
  - performing the find-and-replace action
  - zipping up the new files

[45]:

```python
import sys
import shutil
import zipfile
from pathlib import Path

class ZipReplace:
    def __init__(self, filename, search_string, replace_string):
        self.filename = filename
        self.search_string = search_string
        self.replace_string = replace_string
        self.temp_directory = Path(f"unzipped-{filename}")

    def zip_find_replace(self):
        self.unzip_files()
        self.find_replace()
        self.zip_files()
```

- we could have done everything above without having ever created an object but it does offer the following benefits

- **Readability**: the code for each step is in a self-contained unit that is easy to read and understand. the method name describes what the method does, and less additional documentation is required to understand what is going on
- **Extensibility**: If a subclass wanted to use compressed `TAR` files instead of ZIP files, it could override `zip` and `unzip` methods without having to duplicate the `find_replace` method
- **partitioning**: an external class could create an instance of this class and call the `find_replace` method directly on some folder without having to `zip` the content

0.7 Removing Duplicate Code

- duplicate code is bad for readability and maintainability reasons
- to solve code duplication, the simplest solution is often to move the code into a function that accepts parameters to account for whatever parts are different
- try to make use of inheritance, composition also