Attributes

March 13, 2021

0.1 Getting Attributes

- in other languages if you wanted `.color` you would have getters and setters
- in python you just write `Object.color`
- `.` is called an attribute

- `__getattr__` is a little more funky then the `__getattribute__`
- getattr will run if it cannot find the attribute but getattribute will always run

[19]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'<Pair (x={self.x}, y={self.y})>'

    def __getattr__(self, attrname):
        if attrname == 'z':
            return 'cheese'
        else:
            return 'Singularity has occured'

    def __getattribute__(self, attrname):
        print('__getattribute__ ' + attrname)
        if attrname == 'z':
            return 'cheese'
        return super().__getattribute__(attrname)

    def __dir__(self):
        return [key for key in self.__dict__] + ['z']


point = Point(3, 7)
point.z
point.a
```

```
__getattribute__ z
__getattribute__ a
```

[19]:
```
'Singularity has occured'
```

0.2 Setting Attributes

- we can normally assing items using the = sign
- but maybe we want to hijack it
- __setattr__ runs everytime we set an attribute

[36]:
```
class Bacteria:
    def __init__(self, size, color, habitat):
        self.size = size
        self.color = color
        self.habitat = habitat

    def __repr__(self):
        return f'<Resturant name={self.size} color={self.color}>'

    def __setattr__(self, name, value):
        print(f'{name}:{value}')
        if name == 'habitat' and hasattr(self, 'habitat'):
            print('No!!')
            return
        self.__dict__[name] = value

    def __delattr__(self, name):
        if name == 'location':
            del self.__dict[name]
        else:
            print('You cannot delete that')

bacteria = Bacteria(5, 'red', 'hot springs')
bacteria.habitat = 'swamp'

del bacteria.size
```

```
size:5
color:red
habitat:hot springs
habitat:swamp
No!!
You cannot delete that
```

0.3 Descriptors Property Decorators

- we hang new functionality over functions using decorators
- we can use decorators to set up descriptors, which allows us to create a property that has certain behaviors
- descriptors are objects of properties that have a `get`, `set` and `delete`
- property decorator is actually a class that's creating an object for the class your setting up

[49]:
```
help(property)
```

```
Help on class property in module builtins:

class property(object)
 |  property(fget=None, fset=None, fdel=None, doc=None)
 |
 |  Property attribute.
 |
 |    fget
 |      function to be used for getting an attribute value
 |    fset
 |      function to be used for setting an attribute value
 |    fdel
 |      function to be used for del'ing an attribute
 |    doc
 |      docstring
 |
 |  Typical use is to define a managed attribute x:
 |
 |  class C(object):
 |      def getx(self): return self._x
 |      def setx(self, value): self._x = value
 |      def delx(self): del self._x
 |      x = property(getx, setx, delx, "I'm the 'x' property.")
 |
 |  Decorators make defining new properties or modifying existing ones easy:
 |
 |  class C(object):
 |      @property
 |      def x(self):
 |          "I am the 'x' property."
 |          return self._x
 |      @x.setter
 |      def x(self, value):
 |          self._x = value
 |      @x.deleter
 |      def x(self):
 |          del self._x
 |
 |  Methods defined here:
 |
 |  __delete__(self, instance, /)
 |      Delete an attribute of instance.
 |
 |  __get__(self, instance, owner, /)
 |      Return an attribute of instance, which is of type owner.
 |
```

```
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __set__(self, instance, value, /)
|      Set an attribute of instance to value.
|
|  deleter(…)
|      Descriptor to change the deleter on a property.
|
|  getter(…)
|      Descriptor to change the getter on a property.
|
|  setter(…)
|      Descriptor to change the setter on a property.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __isabstractmethod__
|
|  fdel
|
|  fget
|
|  fset
```

- the `proproty` decorator lets you fine point control how users interact with your class
- you maybe do not want them to change around stuff
- by the way, you need to make your function name match your decorator name

[52]:
```python
class Bacteria:
    def __init__(self, color):
        self._color = color

    @property
    def color(self):
        print('some calculation')
        return self._color
```

```
    @color.setter
    def color(self, value):
        if value not in {'red', 'green', 'blue'}:
            raise ValueError(f'the color {value} is not valid')
        self._color = value

    @color.deleter
    def color(self):
        raise AttributeError("you cannot delete")
        del self._color


bacteria = Bacteria('red')
bacteria.color
```

```
some calculation
```

[52]:
```
'red'
```

0.4 Descriptors Creating A Custom Descriptor

- you can basically have a class with data and that class basically forces you to adhere to some rule if you want to get or assign to that class attributes
- in other programmming languages, you have to use `get` and `set` but not in python

[68]:
```
import re

class BATCH:
    _validation_regex = re.compile('\d{3}')
    def __init__(self, batch='000'):
        self._batch = batch

    def __get__(self, instance, owner):
        print(self)
        print(instance)
        print(owner)
        print('getting the vin')
        return self._batch

    def __set__(self, instance, new_value):
        print('setting the vin to ' + new_value)
        if not self._validation_regex.match(new_value):
            print('your batch must adhere to the appropriate formatting')
            return
        self._batch = new_value

    def __delete__(self):
```

```
        print('deleting the vin')

class Bacteria:

    batch = BATCH()

    def __init__(self, color, batch=None):
        self._color = color
        if batch is not None:
               self.batch = BATCH(batch)

    @property
    def color(self):
        print('some calculation')
        return self._color

    @color.setter
    def color(self, value):
        if value not in {'red', 'green', 'blue'}:
            raise ValueError(f'the color {value} is not valid')
        self._color = value

    @color.deleter
    def color(self):
        raise AttributeError("you cannot delete")
        del self._color


bacteria = Bacteria('red')
bacteria.color
bacteria.batch
bacteria.batch = '012'
```

```
some calculation
<__main__.BATCH object at 0x000001DBC67730D0>
<__main__.Bacteria object at 0x000001DBC67735E0>
<class '__main__.Bacteria'>
getting the vin
setting the vin to 012
```