Object-Oriented Programming (Part 2)

Object-Oriented Programming (Part 2)

- Now that we've looked at decorators, we can delve deeper into object-oriented programming

In [1]:
```
class Duck:
    def __init__(self, name):
        self.hidden_name = name

    def get_name(self):
        '''getter for name attribute'''
        print('Inside the getter')
        return self.hidden_name

    def set_name(self, val):
        '''setter for name attribute'''
        print('Inside the setter')
        self.hidden_name = val

    # the property() class returns a special descriptor object
    name = property(get_name, set_name)
```

In [2]:
```
property()
```

Out[2]:
```
<property at 0x7f4815f24810>
```

In [3]:
```
property().getter
```

Out[3]:
```
<function property.getter>
```

In [4]:
```
property().setter
```

Out[4]:
```
<function property.setter>
```

In [5]:
```
fowl = Duck('Donald')
```

In [6]:
```
fowl.name = 'foo' # invokes the set_name function
```

```
Inside the setter
```

In [7]:
```
fowl.get_name()
```

```
Inside the getter
```

Out[7]:
```
'foo'
```

In [8]:
```
fowl.name = 'Daffy'
fowl.name
```

```
Inside the setter
Inside the getter
```

Out[8]:
```
'Daffy'
```

In [9]:
```
class Duck:
    def __init__(self, name):
        self._hidden_name = name

    @property
    def name(self): #
        '''getter for name attribute'''
        print('Inside the getter')
        return self._hidden_name

    #name = property(name)

    @name.setter
    def name(self, val):
        '''setter for name attribute'''
        print('Inside the setter')
        self._hidden_name = val
```

In [10]:
```
fowl = Duck('Donald')
fowl.name # we no longer have get_name and set_name functions
```

```
Inside the getter
'Donald'
```

Out[10]:

In [11]:

```python
# but hidden_name can still be accessed from outside
fowl.name = 'Marc'
fowl.name, fowl._hidden_name
```

```
Inside the setter
Inside the getter
('Marc', 'Marc')
```

Out[11]:

In [12]:

```python
class Duck():
    def __init__(self, name):
        # data which is intended to be truly private can be preceded with "dunder"
        self.__name = name

    @property
    def name(self):
        '''getter for name attribute'''
        print('hi')
        return self.__name

    # name = property(name)

    @name.setter
    def name(self, val):
        '''setter for name attribute'''
        self.__name = val
```

In [13]:

```python
d = Duck('Donald')
d.name
```

```
hi
'Donald'
```

Out[13]:

In [14]:

```python
d.__name # finally private?
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipykernel_22879/1976914427.py in <module>
```

```
----> 1 d.__name # finally private?

AttributeError: 'Duck' object has no attribute '__name'
```

In [ ]:
```
d.__dict__
```

Out[ ]:
```
{'_Duck__name': 'Donald'}
```

In [ ]:
```
# not quite ... __name is mangled cannot be accessed
# except by its mangled name
d._Duck__name
```

Out[ ]:
```
'Donald'
```

Static and Class Methods

- static methods are methods that don't operate on an instance of the object and therefore are shared by all instances of the object
- class methods are methods that operate on the class itself, rather than instance of the class

In [ ]:
```
class Duck:
    _species = 'fowl' # class data

    def __init__(self, name):
        # data which is intended to be truly private can be preceeded with "dunder"
        self.__name = name

    @property
    def name(self):
        '''getter for name attribute'''
        print('in getter')
        return self.__name

    @name.setter
    def name(self, val):
        '''setter for name attribute'''
        print('IN SETTER')
        self.__name = val

    @staticmethod
    def myprint(thing):
```

```
        '''note that self is NOT the first param'''
        print('-' * len(thing), thing, '-' * len(thing), sep='\n')

    #myprint = staticmethod(myprint)
```

In [ ]:
```
d = Duck('Marc')
Duck.myprint('Marc Benioff')
```

```
------------
Marc Benioff
------------
```

In [ ]:
```
d.name = 'Jeff'
```

```
IN SETTER
```

In [ ]:
```
d.name
```

```
in getter
```

Out[ ]:
```
'Jeff'
```

In [ ]:
```
d.__dict__
```

Out[ ]:
```
{'_Duck__name': 'Jeff'}
```

In [ ]:
```
d._Duck__name
```

Out[ ]:
```
'Jeff'
```

In [ ]:
```
class Example:
    __some_data = 'blah'
    __how_many = 0

    def __init__(self, val):
        print('in init for Example')
        self.name = val # instance data
        self.__class__.__how_many += 1 # get from object to class
        print('__how_many =', self.__class__.__how_many)
```

```
    def __del__(self):
        self.__class__.__how_many -= 1

    # We can use a static (or class) method to get around
    # a brittle __init__ that doesn't quite do what we want.
    @staticmethod
    def list_init(somelist):
        '''allow me to send in a list, and "flatten" it
        into a string with intervening commas'''
        obj = Example('')
        obj.name = ', '.join(somelist)
        return obj

    @classmethod
    def get_some_data(cls):
        return cls.__some_data

    @classmethod
    def get_count(cls):
        return cls.__how_many
```

In [ ]:
```
a = Example('foo')
```

```
in init for Example
__how_many = 1
```

In [ ]:
```
b = Example.list_init(['a', 'b', 'c'])
```

```
in init for Example
__how_many = 2
```

In [ ]:
```
Example.get_count()
```

Out[ ]:
```
2
```

In [ ]:
```
b.name
```

In [ ]:
```
e = Example('foo')
e2 = Example.list_init(['foo', 'bar', 'baz'])
```

```
print(type(e), e.name, e2.name, type(e2))
print(Example.get_count())
```

In [ ]:
```
del e
print(Example.get_count())
```

In [ ]:
```
e3 = e2
```

In [ ]:
```
del e3
```

In [ ]:
```
e2
```

Lab: Class Methods

- add class methods to your class which keeps track of all the instance names which have been created
  - **`.allnames()`** should return a list of all the names of objects which exist
  - **`.count()`** should return the number of objects that have ever been created
  - we will need `__del__` to accomplish this

The Python Data Model

- let's return to our Pythonic deck of cards
- we used named tuples to represent each card
- the 'deck' is simply a list of cards

In [ ]:
```
import collections

Card = collections.namedtuple('Card', 'rank suit')

class Deck:
    # ranks and suits are class attributes because they
    # should be shared by all decks
    __ranks = [str(n) for n in range(2, 11)] + list('JQKA')
```

```
    __suits = 'clubs diamonds hearts spades'.split()

    def __init__(self):
        self._cards = [Card(rank, suit)
            for suit in self.__class__.__suits
                for rank in self.__class__.__ranks]
```

In [ ]:
```
d = Deck()

# We can create a deck of cards, but it turns out it's not iterable...

for card in d:
    print(card)
```

In [ ]:
```
# ...unless we refer to _cards directly

for card in d._cards:
    print(card, end=' ')
```

In [ ]:
```
# we also cannot find the length of the deck
# ...at least not without referring to `_cards` directly
print(len(d._cards))
print(len(d))
```

Making our deck iterable

- the Python data model allows us to accomplish quite a bit, just by implement the `__len__()` and `__getitem__()` methods

In [ ]:
```
# a deck of cards, round two
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class Deck(object):
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'clubs diamonds hearts spades'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
```

```
                                       for rank in self.ranks]
    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
        #return list.__getitem__(self._cards, position)
```

In [ ]:
```
deck = Deck()
len(deck)
```

In [ ]:
```
for card in deck:
    print(card, end=' ')
```

...but just by implementing `__getitem__()`, we get so much more!

In [ ]:
```
# like indexing
deck[0], deck[-1]
```

In [ ]:
```
# ...and slicing!
deck[9:13]
```

In [ ]:
```
deck[12::13]
```

What about a method to pick a random card?

- no need because Python already has a function to choose a random item from a sequence

In [ ]:
```
from random import choice
choice(deck)
```

Two big advantages of using special methods to leverage the Python data model

- users of your classes don't have to memorize arbitrary method names for standard operations ("How to get the number of items? Is it **`.size()`**, **`.length()`**, or what?")
- it's easier to benefit from the rich Python standard library and avoid reinventing the wheel, e.g., **`random.choice()`**

Private Class Methods?

In [ ]:
```
'''Python's name-mangling feature allows us to have somewhat
   private methods and data. As we'll see, though, they can
   still be accessed outside the class, if you're determined.
'''
class MyClass(object):
    def __init__(self, name):
        self.name = name

    def public(self):
        print('This is a public method...name =', self.name)
        print('It can call its own private method, of course...')
        self.__class__.__private()

    @staticmethod
    def __private():
        print('\tThis is a "private" method!')
```

In [ ]:
```
c = MyClass('Dave')
c.public()
```

In [ ]:
```
# Try to call the private method...
c.__private()
```

In [ ]:
```
# ...but we *can* access the private method if we understand
# "name mangling", which adds _classname at the beginning...
c._MyClass__private()
```