oop_notes

September 17, 2020

1 Object Oriented Programming on Python

- Within the class, the object itself is passed as self.
- Outside the class, the object is passed automatically, so there is omitted from the input parameters.

1.1 Class vs Instance Variables

- Class variables are same for each instance of the object
- Instance variables are different for unique object instances.

[1]:
```
# Define a class
class Employee:

        # Class variable
        num_of_emps = 0
        raise_amount = 1.04

        # Constructor (initialise members)
        def __init__(self, first, last, pay):
                self.first = first
                self.last = last
                self.pay = pay
                # self.email = first + last + "@company.com"
                Employee.num_of_emps += 1 # don't use self, will be different␣
 ↪for different unique instances.

        # Arbitrary rerpresentations
        def __repr__(self):
                return "Employee('{}', '{}','{}')".format(self.first, self.
 ↪last, self.pay)

        def __str__(self):
                return "{} - {}".format(self.full_name, self.email)

        # Return combined salary
        def __add__(self, other):
                return self.pay + other.pay

        def __len__(self):
                return len(self.full_name)

        # Regular Methods
        @property
        def full_name(self):
                return '{} {}'.format(self.first, self.last)

        @full_name.setter
        def full_name(self, name):
                first, last = name.split(" ")
                self.first = first
                self.last = last

        @full_name.deleter
        def full_name(self):
                print("Delete Name!")
                self.first = None
                self.last = None

        def apply_raise(self):
                # self.pay = int(self.pay*1.04)
                self.pay = int(self.pay * self.raise_amount)

        @property # access it like an attribute/ member!
        def email(self):
                return "{}{}@company.com".format(self.first, self.last)

        # Class Method
        @classmethod # this means that class is the first argument. cls used␣
↪for convention (can't use class)
        def set_raise_amt(cls, amount):
                cls.raise_amount = amount

        @classmethod
        # Alternative constructor from long string
        def from_string(cls, emp_str):
                first, last, pay = emp_str.split("-")
                return cls(first, last, pay)

        @staticmethod
        # just takes in arguments in needs, for when you don't need to access␣
↪the class.
        def is_workday(day):
                if day.weekday() == 5 or  day.weekday() == 6: # Saturday or␣
↪Sunday
                        return False
                return True

# Unique instance variables of employees (preferred)
emp_1 = Employee("Corey", "Schafer", 5000)
emp_2 = Employee("Yanni", "Chau", 10000)

# Assign members manually
emp_1.first = "Corey"
emp_1.last = "schafer"
# emp_1.email = "coreyschafer@company.com"
emp_1.pay = 5000

# Call print method (2 ways)
print(emp_1.full_name)
# print(Employee.full_name(emp_1))
```

```
Corey schafer
```

Having a look at the entire dictionary of each instance of the employee object.

[2]:
```
print(emp_1.__dict__)
print(emp_2.__dict__)
```

```
{'first': 'Corey', 'last': 'schafer', 'pay': 5000}
{'first': 'Yanni', 'last': 'Chau', 'pay': 10000}
```

Printing the dictionary of the entire employee object.

[3]:
```
print(Employee.__dict__)
```

```
{'__module__': '__main__', 'num_of_emps': 2, 'raise_amount': 1.04, '__init__':
<function Employee.__init__ at 0x10fac34c0>, '__repr__': <function
Employee.__repr__ at 0x10fac3c10>, '__str__': <function Employee.__str__ at
0x10fac3ca0>, '__add__': <function Employee.__add__ at 0x10fac3d30>, '__len__':
<function Employee.__len__ at 0x10fac3dc0>, 'full_name': <property object at
0x10faf8c20>, 'apply_raise': <function Employee.apply_raise at 0x10fafa040>,
'email': <property object at 0x10f991c70>, 'set_raise_amt': <classmethod object
at 0x10faf7e80>, 'from_string': <classmethod object at 0x10faf7eb0>,
'is_workday': <staticmethod object at 0x10faf7ee0>, '__dict__': <attribute
'__dict__' of 'Employee' objects>, '__weakref__': <attribute '__weakref__' of
'Employee' objects>, '__doc__': None}
```

This is handy - setting the raise amount for one employee only.

[4]:
```
emp_1.raise_amount = 1.05
print(emp_1.__dict__)
```

```python
print(emp_2.__dict__)
```

```
{'first': 'Corey', 'last': 'schafer', 'pay': 5000, 'raise_amount': 1.05}
{'first': 'Yanni', 'last': 'Chau', 'pay': 10000}
```

[5]:
```python
print(Employee.num_of_emps)
```

```
2
```

2 Class Methods and Static Methods

Static methods are just regular functions within classes.

Now let's have a look at what the raise amount is. Only employee 1 is 1.05. The default (alongside emp_2) is 1.04

- `@classmethod`
- `@staticmethod`

[6]:
```python
print(Employee.raise_amount)
print(emp_1.raise_amount)
print(emp_2.raise_amount)
```

```
1.04
1.05
1.04
```

This time we set the raise_amount for the entire employee class to 1.05

[7]:
```python
# Alternative methods to do the same thing.
Employee.raise_amount = 1.05
Employee.set_raise_amt(1.05)

print(Employee.raise_amount)
print(emp_1.raise_amount)
print(emp_2.raise_amount)
```

```
1.05
1.05
1.05
```

Here we try and deploy the from string method.

[8]:
```python
emp_str_1 = "John-Doe-70000"
emp_str_2 = "Steve-Smith-30000"
emp_str_3 = "Jane-Doe-90000"

first, last, pay = emp_str_1.split("-")
new_emp_1 = Employee(first, last, pay)
```

[9]:
```
emp_3 = Employee.from_string(emp_str_1)
emp_4 = Employee.from_string(emp_str_2)
emp_5 = Employee.from_string(emp_str_3)
print(emp_3.__dict__)
print(emp_4.__dict__)
print(emp_5.__dict__)
```

```
{'first': 'John', 'last': 'Doe', 'pay': '70000'}
{'first': 'Steve', 'last': 'Smith', 'pay': '30000'}
{'first': 'Jane', 'last': 'Doe', 'pay': '90000'}
```

[10]:
```
import datetime
my_date = datetime.date(2016, 7, 10)
print(Employee.is_workday(my_date))
```

```
False
```

3 Inheritance and subclasses

Subclasses, by default, inherit methods and attributes and methods from their parent classes. Will help make things easier to maintain, inherits a lot of attributes from the parent class (but with modifications)

[11]:
```
# Has all attributes and methods of the employee class
class Developer(Employee):

    # Def new constructor
    def __init__(self, first, last, pay, prog_lang):
        super().__init__(first, last, pay) # handle that with constructor of␣
↪parent class
        # Employee().__init__(self, first, last, pay) # alternative for classes␣
↪with multiple inhertiances
        self.prog_lang = prog_lang

    # Things defined in subclass aoverrides parent class
    raise_amount = 1.10
```

[12]:
```
class Manager(Employee):

    # Def new constructor
    def __init__(self, first, last, pay, employees = None):
        super().__init__(first, last, pay) # handle that with constructor of␣
↪parent class
        if employees is None:
            self.employees = []
        else:
```

```python
            self.employees = employees

    def add_employee(self, emp):
        if emp not in self.employees:
            self.employees.append(emp)

    def remove_employee(self, emp):
        if emp in self.employees:
            self.employees.remove(emp)

    def print_employee(self):
        for emp in self.employees:
            print('-->', emp.full_name)
```

[13]:
```python
# dev_1 =  Employee("Corey", "Schafer", 50000)
dev_1 =  Developer("Erlich", "Bachman", 50000, "Python")
dev_2 = Developer("Tah", "Kitikul", 10000, "Java")

print(dev_1.email)
print(dev_1.prog_lang)
print(dev_2.email)
print(dev_2.prog_lang)
```

```
ErlichBachman@company.com
Python
TahKitikul@company.com
Java
```

[14]:
```python
print(help(Developer))
```

```
Help on class Developer in module __main__:

class Developer(Employee)
 |  Developer(first, last, pay, prog_lang)
 |
 |  Method resolution order:
 |      Developer
 |      Employee
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, first, last, pay, prog_lang)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
```

```
|
|  raise_amount = 1.1
|
|  ----------------------------------------------------------------------
|  Methods inherited from Employee:
|
|  __add__(self, other)
|      # Return combined salary
|
|  __len__(self)
|
|  __repr__(self)
|      Return repr(self).
|
|  __str__(self)
|      Return str(self).
|
|  apply_raise(self)
|
|  ----------------------------------------------------------------------
|  Class methods inherited from Employee:
|
|  from_string(emp_str) from builtins.type
|
|  set_raise_amt(amount) from builtins.type
|      # Class Method
|
|  ----------------------------------------------------------------------
|  Static methods inherited from Employee:
|
|  is_workday(day)
|
|  ----------------------------------------------------------------------
|  Readonly properties inherited from Employee:
|
|  email
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from Employee:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  full_name
|
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from Employee:
 |
 |  num_of_emps = 8

None
```

[15]:
```
print(dev_1.pay)
dev_1.apply_raise()
print(dev_1.pay)
```

```
50000
55000
```

[16]:
```
mgr_1 = Manager("Sue", "Smith", 90000, [dev_1])
mgr_1.add_employee(dev_2)
print(mgr_1.email)
mgr_1.print_employee()
```

```
SueSmith@company.com
--> Erlich Bachman
--> Tah Kitikul
```

[17]:
```
mgr_1.remove_employee(dev_1)
mgr_1.print_employee()
```

```
--> Tah Kitikul
```

Is instance will tell us is if an object is an instance of a class

[18]:
```
print(isinstance(mgr_1, Manager))
print(isinstance(mgr_1, Employee))
print(isinstance(mgr_1, Developer))
```

```
True
True
False
```

Is subclass will tell us if an object is a subclass of a parent class

[19]:
```
print(issubclass(Manager, Employee))
print(issubclass(Employee, Manager))
print(issubclass(Developer, Employee))
print(issubclass(Employee, Developer))
```

```
True
False
True
False
```

4 Magic/ Dunder Methods

- Operator overloading
- Override some built in operations such as print, as well as double underscore (dunder) functions which have special meanings in Python

e.g. - `repr` is an unambiguous representation of the object, used for debugging.
- `str` is a readable representation of an object, display to the end user.

[20]:
```
print(repr(emp_1))
print(str(emp_1))

# Alternative
print(emp_1.__repr__())
print(emp_1.__str__())
```

```
Employee('Corey', 'schafer','5000')
Corey schafer - Coreyschafer@company.com
Employee('Corey', 'schafer','5000')
Corey schafer - Coreyschafer@company.com
```

[21]:
```
print(1+2)
print(int.__add__(1,2))
print(str.__add__("a","b"))
```

```
3
3
ab
```

[22]:
```
print("Combined Salaries")
print(emp_1 + emp_2)
```

```
Combined Salaries
15000
```

[23]:
```
print(len("test"))
print("test".__len__())
```

```
4
4
```

[24]:
```
print(len(emp_1))
```

```
13
```

One problem: things constructed in the constructor don't change automatically!

Solution: property decorators (like getters, setters and deleters)

- `@property` makes it easy to get (but not set) the attribute, for instance the email

- `@(nameofgetter).setter` and then method with the same name.

[25]:
```
print(emp_5)
```

```
Jane Doe - JaneDoe@company.com
```

[26]:
```
emp_5.first = "Janet"

print(emp_5.first)
print(emp_5.email)
print(emp_5.full_name)
```

```
Janet
JanetDoe@company.com
Janet Doe
```

[27]:
```
del emp_1.full_name
```

```
Delete Name!
```