Object-Oriented Programming and Python Classes

Object-Oriented (OO) Programming and Python Classes

- programming paradigm based on the concept of "objects" rather than "actions"
  - objects may contain data, i.e., fields, often called *attributes*
  - objects may contain code, i.e., functions, often called *methods*
  - an object's methods can access and (often) modify the data fields of the object with which they are associated (objects have a notion of *this* or *self*)
- OO programs are created from objects that interact with one another
- Python, like most popular OO programming languages, is class-based
  - i.e., objects are instances of classes, which typically also determine their type

Our first class... BankAccount

- let's consider a simple class, *BankAccount*, which represents, unsurprisingly, a bank account
- what kinds of data (attributes) should a bank account have?
  - owner's name
  - current balance
  - and of course many others, but those are a good start
- what kind of operations (methods) should we be able to perform on a bank account?
  - deposit money
  - withdraw money
  - again, we can think of others, but that's a good minimum set

Things to Know About Classes (Objects) in Python

- some languages, such as Java and C++, use the keyword *this* inside methods, in order to refer to the object itself
- in Python, we use *self*, which, oddly, must be the first argument to every method in the class (with some exceptions we will see in Part 2)
  - *self* is not a reserved word in Python, it is just a naming convention that everyone follows

- when calling an object's methods, Python passes in a reference to that object as the first parameter
- you therefore don't *pass* the parameter, but you must *declare* it
  - will take some getting used to but eventually it will be second nature

In [1]:
```
'''
The class (or classes) in parentheses are classes
from which this class inherits (we will discuss the
concept of inheritance shortly).

In Python 2.0, classes had issues that weren't
corrected until Python 2.2. Beginning with Python
2.2, in order to get the "new style" classes, you
had to inherit from object. If you don't inherit
from object in Python 2, you'll get the "old style"
classes. In Python 3, this no longer matters, because
all classes are "new style" classes in Python 3.
'''
class BankAccount(object):
    def __init__(self, name, initial_balance):
        self.name = name
        self.balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            return self.balance
        else:
            print("can't deposit nonpositive amount!")

    def withdraw(self, amount):
        if amount > 0:
            if amount <= self.balance:
                self.balance -= amount
                return self.balance
            else:
                print("can't withdraw", amount, "or you would be overdrawn!")
        else:
            print("can't withdraw nonpositive amount!")
```

Creating (Instantiating) a BankAccount Object

- to create or *instantiate* an object of type BankAccount, we call the class as if it were a function

- note that an *instance* of the class is different from the class itself

In [2]:
```
marc = BankAccount('Marc Benioff', 500)
ranveer = BankAccount('Ranveer Singh', 300)
print(marc, ranveer, sep='\n')
```

```
<__main__.BankAccount object at 0x7f7e4c056490>
<__main__.BankAccount object at 0x7f7e4c056d00>
```

In [3]:
```
print(type(BankAccount), type(ranveer), type(int), sep='\n')
```

```
<class 'type'>
<class '__main__.BankAccount'>
<class 'type'>
```

What happened when we *instantiated* a BankAcount object?

 alt-text  alt-text

In [4]:
```
print(marc)
print(marc.name, marc.balance, sep='\n')
marc.deposit(25)
```

```
<__main__.BankAccount object at 0x7f7e4c056490>
Marc Benioff
500
```

Out[4]:
```
525
```

In [5]:
```
marc.withdraw(40)
```

Out[5]:
```
485
```

What happens when you call a method such as `deposit()`?

- Python calls the method inside the *class* and passes a reference (the memory address) to the specific object for you which you want to call that method
- Let's consider making a deposit to Ranveer's account...

- Instead of calling **`ranveer.deposit(45)`**, we can invoke the **`deposit()`** method inside the **`BankAccount`** class and pass a reference to the object we want to affect (**`ranveer`** in this case)

In [6]:
```
BankAccount.deposit(ranveer, 45) # ranveer.deposit(45)
```

Out[6]:
```
345
```

Magic Methods

- methods whose name is of the form `__foo__` are called "magic methods" in Python
- you already know one of them: `__init__`
  - `__init__` is called automatically when the object is instantiated
  - sometimes called a constructor (see https://docs.python.org/3/reference/datamodel.html#basic-customization for details of why it's a bit more complicated)

- **`__str__()`** returns a string representation of the object (i.e., for humans)
  - maps to **`str()`** function
  - what you get when you **`print()`** an object
- **`__repr__()`** returns an unambiguous representation of the object which could be fed to Python interpreter to recreate the object
  - maps to **`repr()`** function
  - what you get when you have the interpreter print the value of an object
- a good example of the difference between **`str()`** and **`repr()`** can be demonstrated with a **`datetime`** object...

In [7]:
```
import datetime
today = datetime.datetime.now()
print(type(today), today, sep='\n') # str()
```

```
<class 'datetime.datetime'>
2021-09-20 07:24:31.773380
```

In [8]:
```
today.__repr__() # repr()
```

Out[8]:
```
'datetime.datetime(2021, 9, 20, 7, 24, 31, 773380)'
```

In [9]:
```
str(today) # same as __str__() function in the object
```

Out[9]:
```
'2021-09-20 07:24:31.773380'
```

In [10]:
```
today.__str__()
```

Out[10]:
```
'2021-09-20 07:24:31.773380'
```

In [11]:
```
repr(today) # same as __repr__() function in the object
```

Out[11]:
```
'datetime.datetime(2021, 9, 20, 7, 24, 31, 773380)'
```

In [12]:
```
today.__repr__()
```

Out[12]:
```
'datetime.datetime(2021, 9, 20, 7, 24, 31, 773380)'
```

Let's augment our BankAccount class with `str()` and `repr()` functions...

In [13]:
```
class BankAccount2(object):
    def __init__(self, name, initial_balance):
        self.name = name
        self.balance = initial_balance

    def __repr__(self):
        '''unambiguous representation of the object'''
        return self.__class__.__name__ + '(' + repr(self.name) + ', ' + repr(self.balance) + ')'

    def __str__(self):
        '''string representation of object, for humans
        __repr__ is used if __str__ does not exist'''
        return self.name + ' has ₹' + str(self.balance) + ' in the bank'

    def __add__(self, other):
        return self.__class__(self.name + ' + ' + other.name,
                              self.balance + other.balance - 5.95)

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            return self.balance
```

```
        else:
            print("can't deposit nonpositive amount!")

    def withdraw(self, amount):
        if amount > 0:
            if amount <= self.balance:
                self.balance -= amount
                return self.balance
            else:
                print("can't withdraw", amount, "or you would be overdrawn!")
        else:
            print("can't withdraw nonpositive amount!")
```

In [14]:
```
account1 = BankAccount2('Michael D. Higgins', 150)
account2 = BankAccount2('Theresa May', 150)
print(account1)
```

```
Michael D. Higgins has ₹150 in the bank
```

In [15]:
```
account1
```

Out[15]:
```
BankAccount2('Michael D. Higgins', 150)
```

In [16]:
```
account1 + account2
```

Out[16]:
```
BankAccount2('Michael D. Higgins + Theresa May', 294.05)
```

In [17]:
```
repr(account2)
```

Out[17]:
```
"BankAccount2('Theresa May', 150)"
```

In [18]:
```
code_str = repr(account2)
print(repr(code_str))
dupe_object = eval(code_str)
dupe_object
```

```
"BankAccount2('Theresa May', 150)"
```

Out[18]:
```
BankAccount2('Theresa May', 150)
```

Other Magic Methods

- **__ add __** = add two objects together
- **__ eq __** = implementation of ==
- **__ ne __** = implementation of !=
- **__ len __** = implementation of len() method

Lab: OO Programming

1. Add a **__ eq __()** method to the BankAccount class
   - How you define **__ eq __()** is up to you

- Add a **__ mul __()** method to the BankAccount class
  - it should create a new BankAccount which does something to the name and multiplies the balance by the second operand
- Add a **__ len __()** method to the BankAccount class
- Create a class **`Calculator`** which acts like a calculator
  - Your class should have methods `add()`, `sub()`, `mult()`, `div()`, `pow()`, and `log()`
  - Each of the above methods (except `log`) should take 1 or 2 arguments
    - for 1 argument, e.g., `add(1)`, your method should add to the running total
    - for 2 arguments, your method should act on those 2 arguments to create a new running total
    - e.g., `add(2, 4)` should produce 6, and then if followed by `multiply(5)`, the result should be 30
- All calculations should be stored, and should be accessible to the caller via the `showcalc()` method (kind of like a printing calculator)
- You should also have an `ac()` "all clear" method which clears the running total and the list of calculations (i.e., showcalc() should produce no output, or "0.0" when preceded by a call to `ac()`)

In [19]:
```
from calculator import Calc
mycalc = Calc()
mycalc.add(5)
mycalc.add(4, 3)
print(mycalc.showcalc())
```

```
0 + 5 = 5
4 + 3 = 7
```

In [20]:
```
c = Calc()
```

```
c.add(2, 5) # we are asking one single calculator to add two numbers
c.add(9)
c.mult(13)
c.mult(20, 5)
print(c)
c.ac()
c.add(1)
```

```
2 + 5 = 7
7 + 9 = 16
16 * 13 = 208
20 * 5 = 100
```

Out[20]:
```
1
```

Inheritance

- a Python class can inherit from one or more other classes
- a class which inherits from a class is called a *subclass*
  - the class from which the *subclass* inherits is called the *superclass*
- a subclass which defines a method which exists in the superclass *overrides* the superclass's method

`Word`: A Class Which Inherits from Python's Builtin `str` Class

- unlike strings, `Word`s are ordered by their length, rather than alphabetical order
  - for example...

    ```
    'apple' < 'fig'
    Word('apple') > Word('fig')
    ```

- in all other ways, `Word`s are the same as strings
  - all we need to do is inherit from `str` and *override* the concepts of >, <, >=, <=

In [21]:
```
'apple' < 'fig'
```

Out[21]:
```
True
```

In [22]:
```

```

```
class Word(str):
    def __lt__(self, other):
        # compute length of each Word (string)
        # ask if length of first Word < length of second Word
        print(f'{self} < {other} = {len(self) < len(other)}')
        return len(self) < len(other)

    def __gt__(self, other):
        return len(self) > len(other)

    def __ge__(self, other):
        return len(self) >= len(other)

    def __le__(self, other):
        return len(self) <= len(other)

    def __eq__(self, other):
        return len(self) == len(other)
```

In [23]:

```
my_words = [Word('pomegranate'), Word('apple'), Word('fig'),
            Word('pear')]
print(my_words)
```

```
['pomegranate', 'apple', 'fig', 'pear']
```

In [24]:

```
my_words.sort()
# sort() iterates through the list
# compares pairs of items from the list
# uses less than (<) to do that comparison
print(my_words)
```

```
apple < pomegranate = True
fig < apple = True
pear < fig = False
pear < apple = True
pear < fig = False
['fig', 'pear', 'apple', 'pomegranate']
```

In [25]:

```
Word('apple') == Word('peach'), 'apple' == 'peach'
```

Out[25]:

```
(True, False)
```

Lab: Inheritance

- create a type called FunnyList which has all the chocolately goodness of a list, but adds the following wrinkle:
  - if two lists have same items but in different orders, they are considered equal
  - e.g., `[1, 2, 3] == [3, 1, 2]`

Another inheritance example: Polygon

- a polygon is a multi-sided object
- triangles and squares are polygons with specific properties

In [26]:
```
class Polygon:
    def __init__(self, num_sides):
        self.num_sides = num_sides
        self.sides = [0] * num_sides

    def __repr__(self):
        return ", ".join(str(i) for i in self.sides)

    def input_sides(self):
        self.sides = [float(input("Enter side " + str(i + 1) + ": "))
                      for i in range(self.num_sides)]

    def area(self):
        raise ValueError("Can't compute area of unknown polygon!")
```

In [27]:
```
class Triangle(Polygon):
    def __init__(self):
        '''
        use super() to call __init__ in base class and
        be sure we have 3 sides
        '''
        super().__init__(3)
        # super(Triangle, self)...

    def area(self):
        from math import sqrt
        a, b, c = self.sides
        '''compute semi-perimeter'''
        s = sum(self.sides) / 2
```

```
        '''compute area using Heron's formula'''
        area = sqrt((s * (s - a) * (s - b) * (s - c)))
        return area
```

In [28]:

```
t = Triangle()
t.input_sides()
print(t.area())
```

```
6.0
```

In [29]:

```
p = Polygon(5)
p.input_sides()
p.area()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_22618/2989182706.py in <module>
      1 p = Polygon(5)
      2 p.input_sides()
----> 3 p.area()

/tmp/ipykernel_22618/2206560661.py in area(self)
     12 
     13     def area(self):
---> 14         raise ValueError("Can't compute area of unknown polygon!")

ValueError: Can't compute area of unknown polygon!
```

In [ ]:

```
class Square(Polygon):
    def __init__(self):
        super().__init__(4)

    def input_sides(self):
        '''For a square only need to enter one side'''
        # only need one side length for a square
        s = float(input("Enter length of side: "))
        # only need to store one side
        self.sides = [s] * 4

    def area(self):
        return self.sides[0] ** 2
```

In [ ]:
```
s = Square()
s.input_sides()
print(repr(s))
print(s.area())
```

```
Enter length of side: 5
5.0, 5.0, 5.0, 5.0
25.0
```

Lab: Inheritance

- Create a "ZanyInt" class which inherits from int and redefines certain methods:
- **`len()`** doesn't work for standard ints but make sure it works for a **`ZanyInt`**
- make it so the **`str()`** version of a ZanyInt is something odd, e.g., **`str(3)`** return 'three', but **`str()`** of other numbers returns the number with some leading and trailing spaces
- make it so + usually gives the right answer, but not always (use the **`random`** module)

In [ ]:
```

```

Class variables vs. Instance variables

- variables set outside `__init__` belong to the *class* (as opposed to the *instance*) and are shared by all instances of the class
  - these variables can be accessed via **`ClassName.var`** and **`classInstance.var`**
- variables created inside **init** (and all other method functions) and prefaced with **`self.`** belong to the object *instance* and cannot be accessed via **`ClassName.`**

In [ ]:
```
class Person:
    '''How about that?'''
    species = 'Human'

    def __init__(self, name):
        print('__dict__', self.__dict__)
        self.name = name
        print('__dict__', self.__dict__)
        print(f"{self.name}'s species is {Person.species}")
```

In [ ]:
```
p1 = Person('Godzilla')
```

```
__dict__ {}
__dict__ {'name': 'Godzilla'}
Godzilla's species is Human
```

In [ ]:
```
p1.__dict__
```

Out[ ]:
```
{'name': 'Godzilla'}
```

In [ ]:
```
Person.species, p1.species, p1.name
```

Out[ ]:
```
('Human', 'Human', 'Godzilla')
```

In [ ]:
```
print(p1.__dict__, Person.__dict__, sep='\n')
```

```
{'name': 'Godzilla'}
{'__module__': '__main__', '__doc__': 'How about that?', 'species': 'Human', '__init__': <function Person.__init__ at 0x1
0bf23268>, '__dict__': <attribute '__dict__' of 'Person' objects>, '__weakref__': <attribute '__weakref__' of 'Person' ob
jects>}
```

In [ ]:
```
help(Person)
```

```
Help on class Person in module __main__:

class Person(builtins.object)
 |  Person(name)
 |
 |  How about that?
 |
 |  Methods defined here:
 |
 |  __init__(self, name)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

```
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  species = 'Human'
```

In [ ]:
```
p2 = Person('Mothra')
print(p2.name, p2.species, sep='\n')
```

```
__dict__ {}
__dict__ {'name': 'Mothra'}
Mothra's species is Human
Mothra
Human
```

In [ ]:
```
Person.species = 'animal'
```

In [ ]:
```
print(p1.species, p2.species, Person.species, sep='\n')
```

```
animal
animal
animal
```

In [ ]:
```
p1.species = 'monster'
p1.__dict__
```

Out[ ]:
```
{'name': 'Godzilla', 'species': 'monster'}
```

In [ ]:
```
print(Person.species, p1.species, p2.species, sep='\n')
```

```
animal
monster
animal
```

In [ ]:
```
p1.__dict__
```

In [ ]:
```
Person.species = 'Benioff'
Person.species, p1.species, p2.species
```

Lab: Class variables vs. Instance variables

- create a class with an instance variable called **`name`** which does the following:
  - uses a class variable to keep track of the **`name`** s of the objects that have been created
- what if we wanted to know the names of the instances that exist currently, as opposed to the names of instances which have *ever* been created
  - hint: there is a **`__del__()`** function

Accessing Attributes of an Object

- **`__getattr__(self, name)`**
  - called when you attempt to get the value of an attribute
  - you can add code to deal with attributes that don't exist (perhaps to catch common misspellings or just to avoid exceptions)
- **`__setattr__(self, name, value)`**
  - called when you set the value of an attribute

In [ ]:

```
class Demo:
    def __init__(self):
        self.one = 1
        self.two = 2
        # self.readonly = 'do not change'
        super().__setattr__('readonly', 'do not change')
        # super(Demo, self).__setattr__...

    def __getattr__(self, attr):
        if attr in self.__dict__:
            return self.__dict__[attr]
        else:
            return 'whoop!'

    def __setattr__(self, attr, value):
        print('setting attribute', attr)
        if attr != 'readonly':
            self.__dict__[attr] = value
        else:
            print('nyah!')
```

In [ ]:
```
d = Demo()
d.one, d.two, d.__dict__
```

```
setting attribute one
setting attribute two
```
Out[ ]:
```
(1, 2, {'one': 1, 'two': 2, 'readonly': 'do not change'})
```

In [ ]:
```
d.readonly
```

Out[ ]:
```
'do not change'
```

In [ ]:
```
d.readonly = 1
```

```
setting attribute readonly
nyah!
```

In [ ]:
```
d.readonly
```

Out[ ]:
```
'do not change'
```

In [ ]:
```
d.foo
```

Out[ ]:
```
'whoop!'
```

Lab: __getattr and __setattr

- create an object which holds a value and has a "modification" counter which keeps track of how many times the object has been modified
- for example, the value could be in an attribute called **`value`**, so you'll want to notice when you make changes to **`value`** and increment the counter
- if you allow modifications to other attributes, you won't increment the counter
- consider rejecting attempts to modify the **`counter`** attribute directly
- you will need to use **`super()`**, Python's way to call a method in the parent (superclass) in order to actually modify the attribute...why?

In [ ]:
```
```

```
class ConstantBlock:
    """
    A simple object that lets you set its attributes ONCE.
    Any further attempt to set an attribute results in a
    RuntimeError exception.
    """

    def __setattr__(self, name, value):
        if name in self.__dict__:
            raise RuntimeError("Can't set constant!")
        else:
            super().__setattr__(name, value)

    def __delattr__(self, attr):
        raise RuntimeError("Can't delete constant")
```

In [ ]:
```
my_constants = ConstantBlock()
my_constants.IP_ADDRESS = '127.0.0.1'
my_constants.FOO = 'bar'
```

In [ ]:
```
my_constants.__dict__
```

Out[ ]:
```
{'IP_ADDRESS': '127.0.0.1', 'FOO': 'bar'}
```

In [ ]:
```
my_constants.FOO = 'salesforce'
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-76-0e935da63fd5> in <module>
----> 1 my_constants.FOO = 'salesforce'

<ipython-input-71-9000ca5590bb> in __setattr__(self, name, value)
      8     def __setattr__(self, name, value):
      9         if name in self.__dict__:
---> 10             raise RuntimeError("Can't set constant!")
     11         else:
     12             super().__setattr__(name, value)

RuntimeError: Can't set constant!
```

In [ ]:
```
del my_constants.FOO
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-77-154236cc7683> in <module>
----> 1 del my_constants.FOO

<ipython-input-71-9000ca5590bb> in __delattr__(self, attr)
     13
     14     def __delattr__(self, attr):
---> 15         raise RuntimeError("Can't delete constant")

RuntimeError: Can't delete constant
```