In-class worksheet 17

Mar 26, 2019

Introduction to classes in Python

Python is an object-oriented programming language. In computer science, the term "object" describes the combination of data and methods that can manipulate the data. It is often convenient to keep the methods that operate on a given type of data together with the data itself, and therefore object-oriented programming is widely used and is pervasive throughout Python.

In fact, most built-in data types in Python are objects. For example, we have previously worked with lists. Lists are objects. They contain data (the elements of the list) and methods that can manipulate the data (for example, the function `append()`):

In [1]:
```
mylist = [1, 2, 3]  # create a list object
print(mylist)       # output: [1, 2, 3]


mylist.append(4)    # call append() function of the list object
                    # the append() function modified the list object
print(mylist)       # output: [1, 2, 3, 4]
```

```
[1, 2, 3]
[1, 2, 3, 4]
```

Note that to call a function on a given object, we use a period (as in `mylist.append(4)`). This is different from R, where the period has no special meaning and can be part of regular variable or function names. In Python, you cannot use periods in variable or function names.

Importantly, we are not limited to using only built-in objects, we can also define our own, using the `class` mechanism. Classes are collections of variables and functions. For example, the following class implements a counter that we can increment or reset.

(Note: it is convention to name classes with a starting upper-case letter but the objects instantiating a given class with a lower-case letter.)

In [2]:
```
# a simple Counter class
class Counter:
    count = 0       # variable that holds the current count

    # A function that increments the counter by 1. Note the `self` argument.
    # The function needs its own object as argument, so it can work on
    # the appropriate data. The contents of `self` is provided by Python
    # when the function is called on an object.
    def increment(self):
        self.count += 1  # increment the counter by 1

    # A function that resets the counter to 0.
    def reset(self):
        self.count = 0   # set the counter to 0

# define two counter variables
c1 = Counter()
c2 = Counter()

# use c1 to count the number of characters in a string
for c in "The quick brown fox jumps over the lazy dog.":
    c1.increment()

# increment c2 just once
c2.increment()

print("We incremented c1", c1.count, "times.")
print("We incremented c2", c2.count, "times.")

# reset c1 to 0
c1.reset()
print("After reset, c1 count is at", c1.count)
```

```
We incremented c1 44 times.
We incremented c2 1 times.
After reset, c1 count is at 0
```

In object-oriented programming, we generally want to properly initialize an object when it is created. In Python, this is done with the `__init()__` function. We should define this function for every class we write.

As an example, we'll define a class `Dog`, which stores the name of the dog and a list of tricks the dog can do. The name of the dog needs to be defined when the dog is created, and hence it is an argument to the `__init()__` function. The list of tricks is initially empty, until tricks are added to the dog.

In [3]:
```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

dog1 = Dog('Fido')
dog2 = Dog('Buddy')
dog1.add_trick('sit down')
dog1.add_trick('roll over')
dog2.add_trick('play dead')
print(dog1.name, "can do these tricks:", dog1.tricks)
print(dog2.name, "can do these tricks:", dog2.tricks)
```

```
Fido can do these tricks: ['sit down', 'roll over']
Buddy can do these tricks: ['play dead']
```

Problems

Problem 1:

(a) Take the `Counter` class defined above and add a function that decrements the counter by 1. Then make up a simple example that uses this decrement function.

In [4]:
```
# Code for Problem 1a goes here.

# a simple Counter class
class Counter:
    count = 0       # variable that holds the current count

    def increment(self):       # increment the counter by 1
        self.count += 1

    def reset(self):     # set the counter to 0
        self.count = 0

    def decrement(self):       # decrement the counter by 1
        self.count -= 1

c1 = Counter()
c1.decrement()
c1.decrement()
print("The counter c1 now has the value:", c1.count)
```

```
The counter c1 now has the value: -2
```

(b) It is good programming style to always implement a `__init__()` function for every class. Write a `__init__()` function for the `Counter` class. This function should have an optional argument (with default value of 0) defining the initial value of the counter.

In [5]:
```
# a simple Counter class
class Counter:
    def __init__(self, starting_count = 0):
        self.count = starting_count

    def increment(self):       # increment the counter by 1
        self.count += 1

    def reset(self):     # set the counter to 0
        self.count = 0

    def decrement(self):       # decrement the counter by 1
        self.count -= 1

c1 = Counter(5)
c2 = Counter()
print("The counter c1 now has the value:", c1.count)
print("The counter c2 now has the value:", c2.count)
```

```
The counter c1 now has the value: 5
The counter c2 now has the value: 0
```

(c) Take the exact code from your solutions to part (b), but now add a `print()` statement to the `__init()__` function to see when the `__init()__` function actually gets called.

In [6]:
```
# a simple Counter class
class Counter:
    def __init__(self, starting_count = 0):
        self.count = starting_count
        print("New counter initialized to:", starting_count)

    def increment(self):       # increment the counter by 1
        self.count += 1

    def reset(self):     # set the counter to 0
        self.count = 0

    def decrement(self):       # decrement the counter by 1
        self.count -= 1

c1 = Counter(5)
c2 = Counter()
print("The counter c1 now has the value:", c1.count)
print("The counter c2 now has the value:", c2.count)
```

```
New counter initialized to: 5
New counter initialized to: 0
The counter c1 now has the value: 5
The counter c2 now has the value: 0
```

Problem 2:

Take the `Dog` class defined above and add a function `knows_trick` that checks whether the dog knows a given trick or not.

In [7]:
```
# Code for Problem 2 goes here.

class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

    def knows_trick(self, trick):
        return trick in self.tricks

dog1 = Dog('Fido')
dog2 = Dog('Buddy')
dog1.add_trick('sit down')
dog1.add_trick('roll over')
dog2.add_trick('play dead')
print(dog1.knows_trick('roll over'))
print(dog2.knows_trick('sit down'))
```

```
True
False
```

If this was easy

Problem 3:

Take the modified `Dog` class from Problem 2 and add a function `add_tricks_from` that takes as argument another dog and adds all the tricks from that dog to the current dog. Make sure that tricks are not added twice; if the dog knows a trick already then that trick shouldn't be added again. Also, don't access the list `self.tricks` directly in the new function you are adding. Use the functions `add_trick` and `knows_trick` instead.

In [8]:
```
# Your code goes here.

class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

    def knows_trick(self, trick):
        return trick in self.tricks

    def add_tricks_from(self, dog):
        for trick in dog.tricks:
            if not self.knows_trick(trick):
                self.add_trick(trick)

dog1 = Dog('Fido')
dog2 = Dog('Buddy')
dog1.add_trick('sit down')
dog1.add_trick('roll over')
dog2.add_trick('sit down')
dog2.add_trick('play dead')
dog2.add_tricks_from(dog1)
print(dog1.name, "can do these tricks:", dog1.tricks)
print(dog2.name, "can do these tricks:", dog2.tricks)
```

```
Fido can do these tricks: ['sit down', 'roll over']
Buddy can do these tricks: ['sit down', 'play dead', 'roll over']
```

Problem 4:

Take the basic `Dog` class (with functions `__init__()` and `add_trick()`) and modify it so that it keeps track of the total number of dogs in existence. You should be able to retrieve this number from a member variable `Dog.num_dogs`. Make sure you not only increment the number of dogs when new dog instances are created, but also decrement it when dog instances are removed.

Note: Techniques such as these are commonly used in programming, in particular in the context of reference counting. (https://en.wikipedia.org/wiki/Reference_counting)

In [9]:
```
class Dog:
    num_dogs = 0 # we start out with 0 dogs

    def __init__(self, name):
        Dog.num_dogs += 1   # we just created a new dog, increment the number b
y 1
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

    def __del__(self):
        Dog.num_dogs -=1    # we're losing one dog, need to keep track of that

# make 10 new dogs
dogs = []
for i in range(10):
    dogs.append(Dog(i))
print("We now have", Dog.num_dogs, "different dogs.")

# empty the list
dogs = []
print("We now have", Dog.num_dogs, "different dogs.")
```

```
We now have 10 different dogs.
We now have 0 different dogs.
```