Python Sticker Collection

Object Oriented Version

In this exercise we will be using classes to encapsulate functions and their data. We will be writing a simulation of sticker collecting. My interest in this simulation came from the following question:

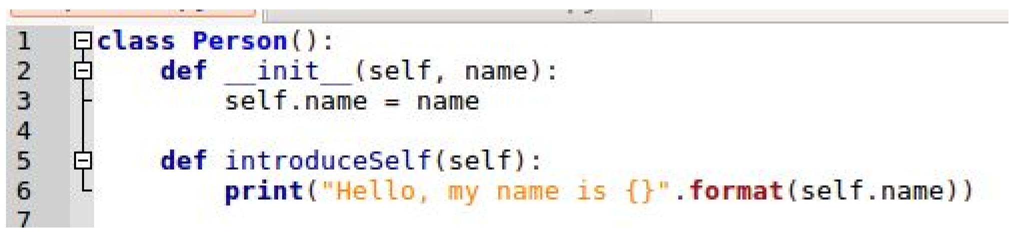
“Do they deliberately print a smaller number of some stickers to keep you buying them?”

By simulating the sticker collection on a computer, we can see how the collection stacks up with a truly random sticker printing process. I believe this program will demonstrate that there is no need for the sticker printers to make some ‘rarer’ than others, because even with a completely fair and even distribution, you will end up with many swaps, and many swaps of the same old faces.

Object Oriented Programming

We will be using the **class** keyword in Python to define our program. Classes are a means by which a programmer can group up a number of variables in a structure, and give functions to that structure which operate on a particular instance. If that is not clear (and who could blame you) let us look at an example.

If we wanted to simulate people introducing themselves, we could write a class called **Person**. The **Person** class might look like this.



```
1  class Person():
2      def __init__(self, name):
3          self.name = name
4
5      def introduceSelf(self):
6          print("Hello, my name is {}".format(self.name))
7
```

Let us break that down line by line.
Line 1 is the *class declaration*. It tells Python that we want to create a new type named **Person**. Any code indented within the Person is assumed to belong to that class.

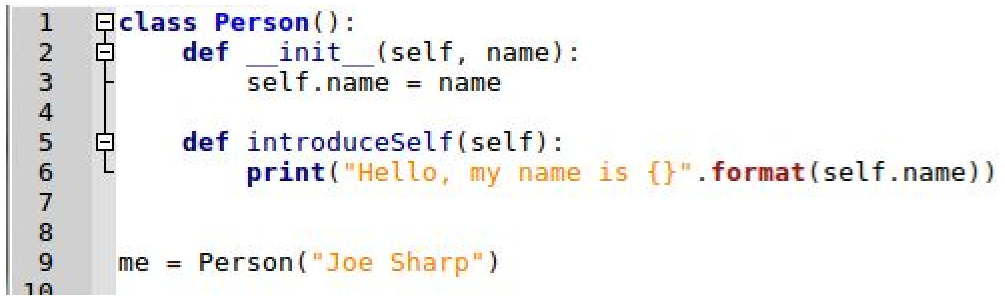
Line 2 is the start of the *constructor*. This is the function that is called when we create *instances* of the **Person** class. This function has a special name that Python recognises **__init__**. The first parameter is **self** which is a reference to the *instance* of the class. We will see that in action later. The second parameter is going to be the **name** of the person.

Line 3 shows us setting the variable **name** on our instance of the **Person**. Each **Person** will have a **name**. This belonging of variables is part of what makes a class powerful.

Line 5 is our own custom function called **introduceSelf**. It requires no actual parameters, but because it is a *class member function* it has to have the parameter **self**.

Line 6 is the implementation of our **introduceSelf** function. In this function we will simply print a greeting from the Person. Note the use of the format function on the string.

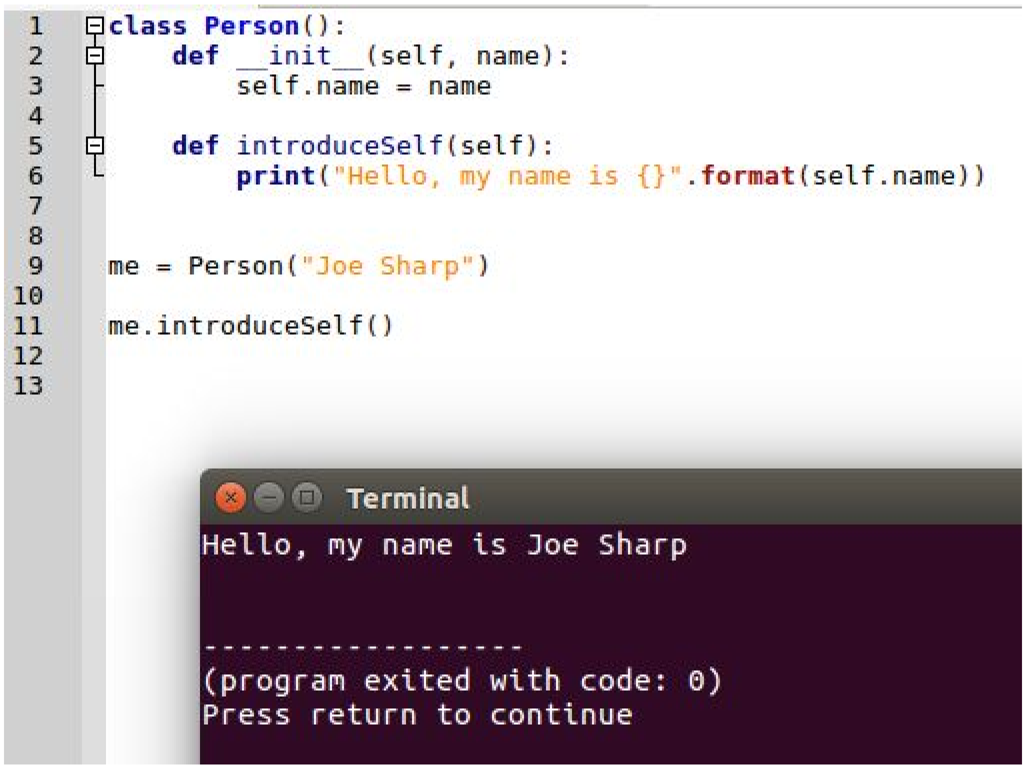
Now that we have defined a class, let us create *instances*. Here is how we create a single instance of Person.



```
1   class Person():
2       def __init__(self, name):
3           self.name = name
4
5       def introduceSelf(self):
6           print("Hello, my name is {}".format(self.name))
7
8
9   me = Person("Joe Sharp")
10
```

Line 9 shows the creation of a single instance of person and assigns it to a variable called **me**. Notice that we only passed in the **name** of the Person. We do not have to pass in **self** because Python understands this is a class and does that for us.

Now that we have created an instance we can call the **introduceSelf** function on the instance. How do we do this? See Line 11 below:

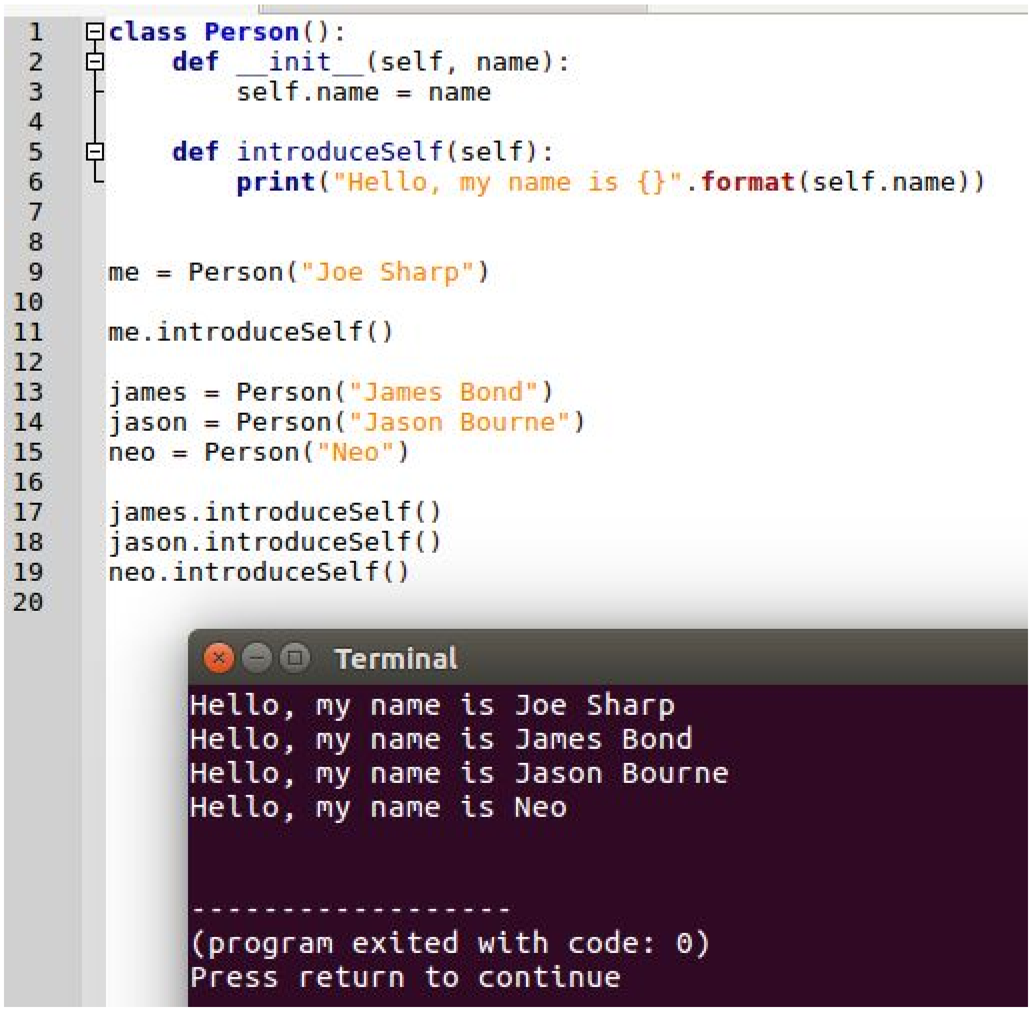


```python
class Person():
    def __init__(self, name):
        self.name = name

    def introduceSelf(self):
        print("Hello, my name is {}".format(self.name))


me = Person("Joe Sharp")

me.introduceSelf()
```

```
Hello, my name is Joe Sharp


------------------
(program exited with code: 0)
Press return to continue
```

You can now run this code and you should see the print out. Again note that we do not have to pass **self** into the **introduceSelf** function, this is done by Python because we have called the *method* on our *instance*.

Now that we have created a single instance let's create some more and call upon them to introduce themselves.



```python
class Person():
    def __init__(self, name):
        self.name = name

    def introduceSelf(self):
        print("Hello, my name is {}".format(self.name))


me = Person("Joe Sharp")

me.introduceSelf()

james = Person("James Bond")
jason = Person("Jason Bourne")
neo = Person("Neo")

james.introduceSelf()
jason.introduceSelf()
neo.introduceSelf()

```

```
Hello, my name is Joe Sharp
Hello, my name is James Bond
Hello, my name is Jason Bourne
Hello, my name is Neo


------------------
(program exited with code: 0)
Press return to continue
```

I have created 4 people now and called the **introduceSelf** function on each one. Notice that each instance uses its own name in the print out.

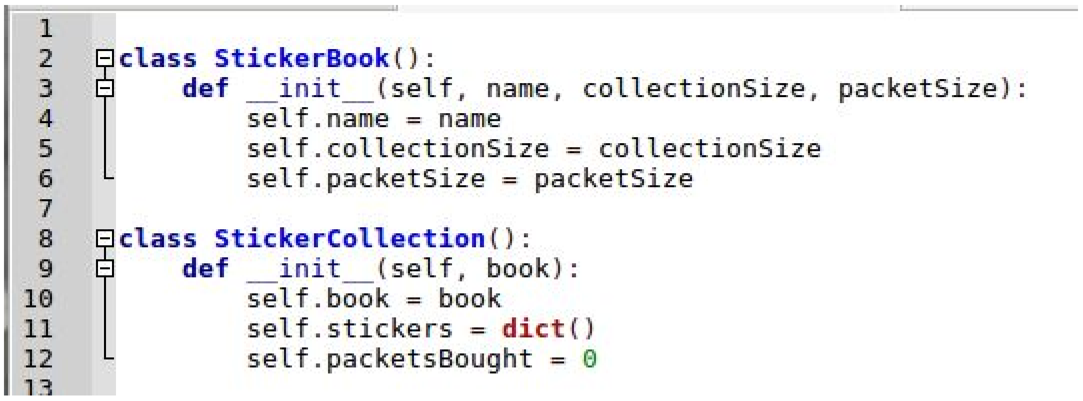
Sticker Book Design

Now that we have had a brief introduction to classes, we shall design our sticker book code. The first thing we will address is deciding which classes we want to create. I have decided on the following classes.

- Sticker Book - This will contain the following member variables
  - Name - the name of the collection (such as ‘Pokemon’ or ‘World Cup 2014’)
  - Collection Size - The total number of sticker available in the book
  - Packet Size - The number of stickers in a single packet.
- Sticker Collection
  - Book - The collection will be of a single book. So the book shall be passed in so that the collection can refer to it.
  - Stickers - The list of stickers collected so far

- Packets Bought - The number of packets bought for this collection

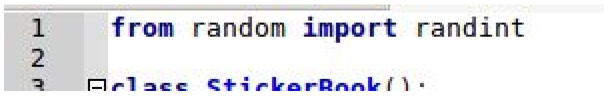
That should do for now. The following code declares these classes.



```python

class StickerBook():
    def __init__(self, name, collectionSize, packetSize):
        self.name = name
        self.collectionSize = collectionSize
        self.packetSize = packetSize

class StickerCollection():
    def __init__(self, book):
        self.book = book
        self.stickers = dict()
        self.packetsBought = 0

```

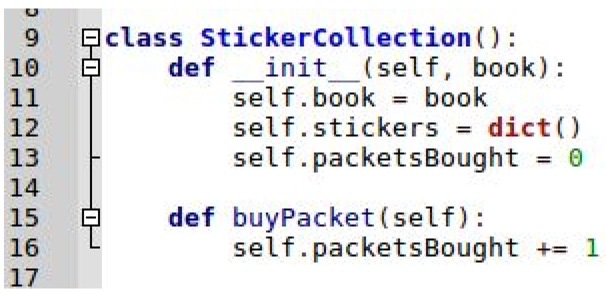
The **StickerCollection** class uses the **book** passed into the constructor. It then creates an empty **dictionary** and initialises the count of **packetsBought** to zero.

Now that we have our classes, let us write a *function* for the **StickerCollection** class. We will write a function called **buyPacket**. This will randomly choose a number of stickers to receive in a packet (according to the packet size) and put that data into our dictionary. First we will need to import the **randint** function from the **random** library. Do this as shown in Line 1:



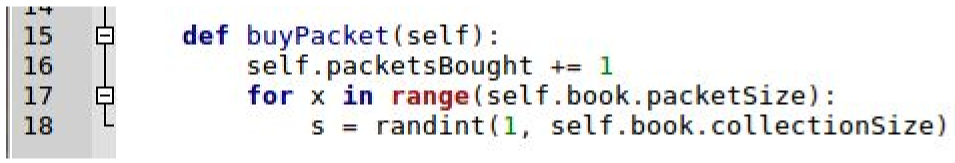
```python
1   from random import randint
2
3   class StickerBook():
```

Now add the member function to the StickerCollection class as shown in lines 15-16.



```python
 8
 9  class StickerCollection():
10      def __init__(self, book):
11          self.book = book
12          self.stickers = dict()
13          self.packetsBought = 0
14
15      def buyPacket(self):
16          self.packetsBought += 1
17
```

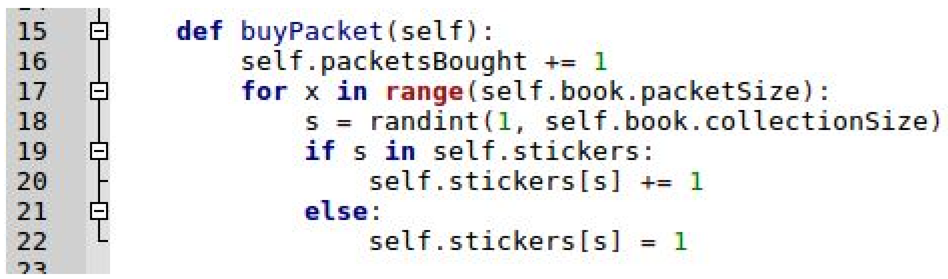
So far our function just increments the number of packets bought. Now we need to buy as many stickers as there are in a packet. We will use a **for loop** and the **packetSize** variable from the **book** tied to the collection. The code looks like this.



```
15    def buyPacket(self):
16        self.packetsBought += 1
17        for x in range(self.book.packetSize):
18            s = randint(1, self.book.collectionSize)
```

Line 17 shows the loop, and how we are accessing **packetSize** on our **book** *member variable.* Line 18 shows us choosing a random sticker between 1 and the collection size of our book.

Now that we have chosen a sticker, we will record it in our **stickers** dictionary that we created earlier.



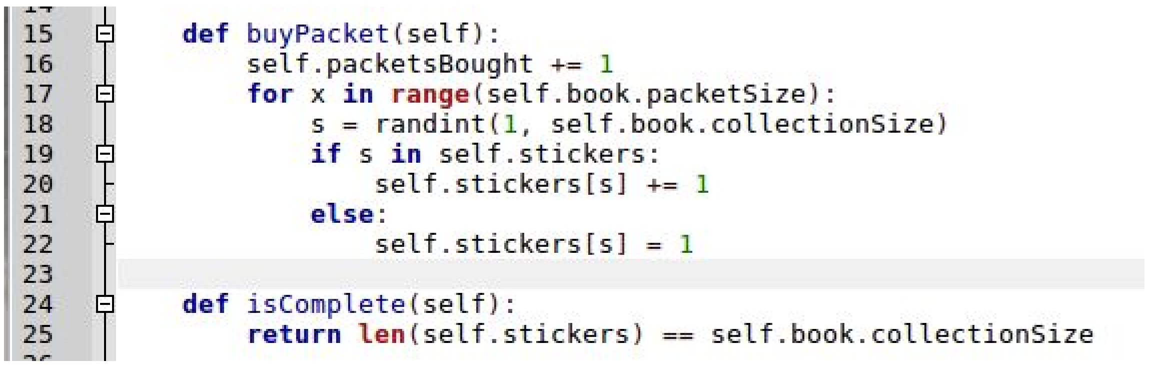
```
15    def buyPacket(self):
16        self.packetsBought += 1
17        for x in range(self.book.packetSize):
18            s = randint(1, self.book.collectionSize)
19            if s in self.stickers:
20                self.stickers[s] += 1
21            else:
22                self.stickers[s] = 1
```

At the start of our collection, the **self.stickers** dictionary is empty. So there are no keys 'in it'. As we collect more stickers we will count how many we have of each number so we can track our swaps. Once we have a sticker for every number in the collection we will assume the collection is complete.

Line 19 - This shows how we determine if a key is in a dictionary.
Line 20 - If the value exists in the dictionary we increment it by 1
Line 21 - The else, this means the sticker we have just bought it not already in our collection
Line 22 - If it is a new sticker, we simply count the first 1 of that number in our dictionary.

If we ran the **buyPacket** function repeatedly. Eventually the dictionary would contain every sticker in the collection and we would be finished.

We will now write a *member function* for the **StickerBook** class that we can use to check if the collection is complete. This function is shown in lines 24 and 25 below.

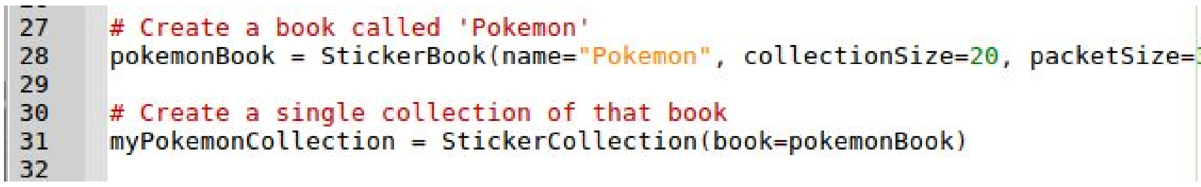


```python
    def buyPacket(self):
        self.packetsBought += 1
        for x in range(self.book.packetSize):
            s = randint(1, self.book.collectionSize)
            if s in self.stickers:
                self.stickers[s] += 1
            else:
                self.stickers[s] = 1

    def isComplete(self):
        return len(self.stickers) == self.book.collectionSize
```

I am using the length of the stickers dictionary and the collection size to check if the collection is complete. Since we are using a dictionary, when we get swaps it does not increase the number of keys in the dictionary. The keys are unique for each sticker number.

We now have enough code to complete a collection. We will write some code to create a sticker book, then create a collection, and cycle through purchasing stickers until the collection is complete.

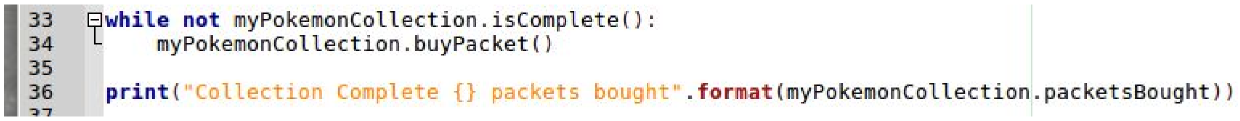
Note the indentation is back against the margin, this code exists outside either of our class definitions.



```python
27  # Create a book called 'Pokemon'
28  pokemonBook = StickerBook(name="Pokemon", collectionSize=20, packetSize=3)
29
30  # Create a single collection of that book
31  myPokemonCollection = StickerCollection(book=pokemonBook)
32
```

Line 28 creates the Sticker Book, I am using named parameters here so a developer can see the purpose of the various numbers.
Line 31 creates a single collection against that book. The cool thing about the encapsulation is that we could create multiple collections of the same book to simulate a group of friends collecting the same stickers.

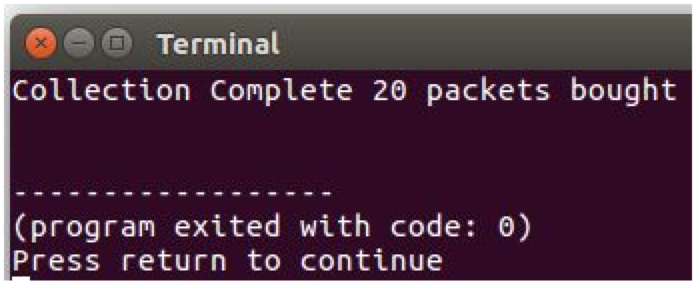
Let us now create a loop for buying the stickers.



```python
while not myPokemonCollection.isComplete():
    myPokemonCollection.buyPacket()

print("Collection Complete {} packets bought".format(myPokemonCollection.packetsBought))
```

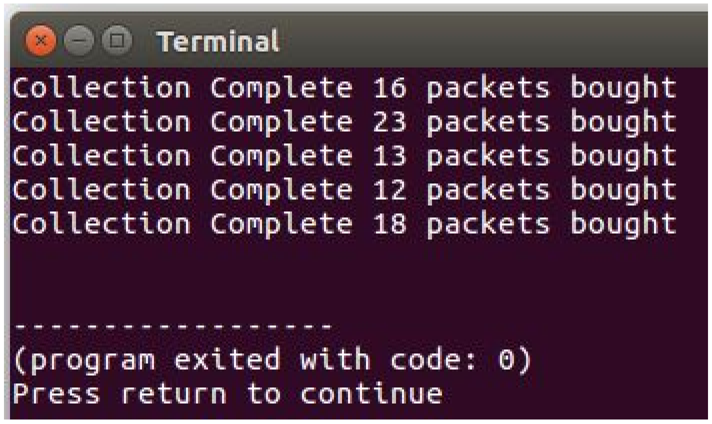
Note that I am calling **isComplete** on our instance of the sticker book called **myPokemonCollection**. While the collection is incomplete, keep buying stickers.

Line 36 shows us printing how many packets were bought by the end of the collection. Here is an example of that running.



```
Collection Complete 20 packets bought


-----------------
(program exited with code: 0)
Press return to continue
```

If you run this code several times you will get different numbers each time, that is the luck of the draw. Try adding a loop so it runs the entire collection simulation 5 times. Your print should look like this.



```
Collection Complete 16 packets bought
Collection Complete 23 packets bought
Collection Complete 13 packets bought
Collection Complete 12 packets bought
Collection Complete 18 packets bought


-----------------
(program exited with code: 0)
Press return to continue
```