Iterators and generators

Estimated time for completion: 45 minutes

Overview:

It's time to work with collections. Python has a host of features and types that make working with collections delightful. We will explore some of these in this lab.

These instructions assume you have **Python 3** installed. You can use any OS you choose (OS X, Windows, or Linux).

Goals:

- Use generator methods to simplify iteration of complex types
- Add custom iteration to a type
- Use list comprehensions and generator expressions to filter and transform data

Part 1 – Simple generator methods

Open the python files (or project if you are using PyCharm) in **Iterator_Zen/Before/iterator_zen_before**. Run program.py and observe that it outputs a message warning that certain elements are not yet implemented. The first one has to do with custom iteration of types (that is, using a custom type in a for loop).

Open **custom_iter.py** and review the TODOs. First you need to fill out two classes **ShoppingCart** and **CartItem**. The shopping cart should hold a list of **CartItem**s and **CartItem**s should have names and prices. For bonus points, add verification and raise exceptions when they are violated (e.g. adding a None cart item is not allowed).

Once that is done, add an **__iter__** method to **ShoppingCart**. Its implementation should use the yield keyword to return all the cart items.

Here is the expected output from this step:

```
Shopping cart details:


Tesla P85 - Yellow ............................... $120,000
Tesla - White .................................... $69,000
Lotus Evora ...................................... $79,890

_____________________________________________________________

Subtotal                                          $268,890
Tax                                               $40,333
Total                                             $309,223
```

Part 2 – Building efficient sequences with generators

Turn your attention to the **generators.py**. There are two parts here. The first part is to implement the Fibonacci numbers as a sequence. We could approximate them using lists and fixed values, but recall this sequence is infinite. It turns out generator methods are perfect for these types of scenarios.

Open the **get_fibonacci** method. Follow the hints in the comments to implement the sequence using the yield keyword. The expected output is:

```
Some (efficient) information about fibonacci numbers:


1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,

Sum of fib less than 100,000 => 196,417
```

That was interesting, but pretty academic. Let's see how we can use yield to make working with real data structures easier.

The next part of the generators.py file works with an ordered tree class defined in tree.py. This is mostly done for you already but the iteration, which is supposed to return the items in order, is not done. We can navigate trees depth first with recursion, but sometimes a simple sequence is what you want (it is here).

Your job is to add iteration to the tree so for looping over the tree simply returns the items in order. The method **Tree. items_in_order** is the place to do your work (notice how it's used in **__iter__**).

Once you get that working, you should see something like this (the numbers are random so they vary):

```
Here is a ordered tree (raw):

[left=[left=None         data=5  right=[left=None         data=15         right=None]]    data=18
        right=[left=[left=[left=None     data=43         right=[left=[left=None   data=49
        right=None]     data=56         right=None]]    data=58         right=[left=[left=None
```

```
        data=65         right=None]     data=67         right=None]]    data=68         right=None]]
```

This might make more sense (in order):

5,15,18,43,49,56,58,65,67,68,

Part 3 – List comprehensions and generator expressions

Finally, we'll look at filtering and transforming data with list comprehensions and generator expressions. We will be working with a Bike class and a set of bikes:

```
bikes = [
    Bike(name="Specialized FSR Comp", front_suspension="FOX Forks", rear_suspension="FOX shock + swing arm"),
    Bike(name="Specialized FSR", front_suspension="Basic Forks", rear_suspension="spring shock + swing arm"),
    Bike(name="Specialized Stump Jumper", front_suspension="Basic Forks", rear_suspension=None),
    Bike(name="Specialized Stump Jumper", front_suspension="Basic Forks", rear_suspension=None),
    Bike(name="Giant Anthem Advanced XC", front_suspension="High-end Rockshox Forks",
         rear_suspension="Rockshox shock + swing arm"),
    Bike(name="Stromer ST1 Elite", front_suspension="High-end Forks", rear_suspension=None, wattage=500),
    Bike(name="Stromer ST1", front_suspension=None, rear_suspension=None, wattage=350),
    Bike(name="Basic Walmart Bike", front_suspension="Spring Forks", rear_suspension="cheap spring + swing arm"),
]
```

Open up the **comprehensions.py** file and see there are two basic questions being answered about these bikes:

1. What are the **names** of the **electric** bikes? Use a list comprehension to return exactly this data as a list (it's printed in the next statement)
2. Which **bikes** are suitable for off-road riding? An off-road bike includes:
    a. They must have dual suspension

b. The suspension cannot be not fake, cheap suspension (like the Wal-Mart bikes come with)
(answer this question with a generator expression)