420-LCW-05 Programming Techniques and Applications - Assignment 5

April 20, 2018

Here's a reminder of the general requirements for all of the course assignments:

1. **Identification section** Do this for *every* Python file in every assignment in this course. This section must be either in a comment, with a '#' preceding each line, or enclosed within triple quotes ("'). The grader and I need this section for the *accurate processing of your assignment*. Assignments missing this may lose up to 5% of the total mark.

   Example:

   ```
   """
   Justin Trudeau, 1234567
   Sunday, February 31
   R. Vincent, instructor
   Assignment 2
   """
   ```

   Obviously substitute your name, Marianopolis ID, and the correct date for the appropriate fields!

2. Always include additional comments with your code. These need not explain every individual line of your program, but consider using comments for the following situations:

   - A brief explanation of a particular variable's purpose, included on the first line where the variable is defined, e.g.:

     ```
     hi = 100 # Define the upper limit of the range.
     ```

   - A note mentioning any website or person you may have consulted with to help with the assignment.
   - A comment describing any constant value that appears in your code.

   Please do not create an unreasonable or excessive number of comments - either extreme is to be avoided.

   In addition to your comments, each `def` statement, whether for a global function or class method should include at least a brief docstring. *You should also provide docstrings for any new classes you create*.

3. Your submission for assignment will typically include multiple Python files, which have the extension `.py`. Before submission, these files must be combined into a single ZIP archive file (extension `.zip`). If you do not know how to create a ZIP file, I will demonstrate this in lab.

4. Be sure to respect other instructions specified in the assignment. Part of each assignment is to correctly follow the instructions as closely as possible.

Introduction

There are two major parts to this assignment - regular expressions and a brief exercise on *generators*, an advanced Python features.

1 Exercise 1 - Regular expressions

As we discussed in class, there are many programming problems that are easily (or more easily) solved using regular expressions. In this part of the assignment you must use Python's regular expression module to solve a few example problems.

For reference, look at the slides (which will be posted shortly), or the information at this web site: https://developers.google.com/edu/python/regular-expressions. The information on this site is good quality, but the examples are coded using Python 2. This primarily matters that the arguments to `print` are not surrounded by parentheses. Keep this in mind as you follow their discussion.

1.1 1a - Finding a genetic sequence

Fragile X syndrome is a relatively common genetic problem with significant developmental effects. In this exercise, you will write a program to determine which of two DNA sequences came from a patient with fragile X syndrome.

As you probably know, DNA is made up of four nucleotides, referred to by the letters 'a', 'g', 'c', and 't'. DNA sequence data typically takes the form of a string containing primarily these four characters. A common form is known as FASTA format (see `https://en.wikipedia.org/wiki/FASTA_format`. FASTA data is case-insensitive, so your code should consider upper and lower case to be identical. FASTA files also include newline characters that you must ignore, so your program has to strip them out somehow.

Fragile X syndrome manifests as an excess number of repetitions of a particular 3-base pattern on FMR1 gene the X chromosome. The pattern always begins with the sequence `gcg`, followed by any number of repetitions of either `cgg` or `agg`, ending with the three letters `ctg`. In typical people the number of repetitions is less than 40. In fragile X syndrome, the number of repetitions of the `cgg` or `agg` is excessive, generally over 200.

Your job is to write a program that examines two FASTA sequences, named `patient1.fasta` and `patient2.fasta`, and determine which one is normal and which one shows signs of the disease.

Your program should be called `a5ex1a.py`. It should just print the number of `cgg/agg` repeats found in the two example files.

1.2 Exercise 1b - Identifying floating-point numbers

Imagine you are working on a program written in the C language, and as part of fixing the program you want to examine all of the lines that contain a floating-point constant, as some of these values are suspected of being incorrect.

Floating-point constants in C follow the same rules as those in Python, except that they can end in a lower-case 'f' character.

Write a program called `a5ex1b.py` that opens the file `trace.c` and uses regular expressions to find and print all of the lines that contain a floating-point constant.

2 Exercise 2 - Generators

One interesting feature of Python that we have not covered is the `yield` statement, which is used to turn a function into something called a *generator*.

The `yield` statement works sort of like the `return` statement, in that it "returns" a value to the calling program. The surprising thing about `yield` is that, unlike `return`, it does not immediately end the execution of the function. Instead, the function continues to do its work, but it continually "feeds" values back to the caller in an iterable fashion.

As an example, consider the `range` function in Python. It is a function that returns an iterable. We can mimic the operation of `range` like this:

```python
def simple_range(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Now we can use this simplified version of `range` like this:

```python
for x in simple_range(10)
    print(x)
```

Try this - your program should print the numbers from 0 to 9 just as if you used the standard `range` function.

The `yield` statement transforms a function into an iterable - so you can call the function, and it immediately returns a *generator* object. If you iterate over the returned generator, Python automatically re-invokes your function and gets the next value in the sequence. This allows for the easy creation of specialized iterable objects and functions, which is what you'll try in this exercise

2.1 Exercise 2a - an Fibonacci generator

In a file named `a5ex2a.py`, write a function named `fib(n)` that generates the first `n` Fibonacci numbers. Remember that `fib(0)` is 0, `fib(1)` is 1, and `fib(n)` is `fib(n-1) + fib(n-2)`. Write your function iteratively, but using the `yield` statement. Test your function by adding the following code at the end of your file:

```
for n in fib(10):
    print(n)
```

Run this program - it should print the first ten Fibonacci numbers, each on a line by itself.

You may be tempted to imagine that your function runs once and just creates a list, which is quietly returned somehow. The operation of generators is stranger than this, however. To see what I mean, add a call to `print('Hello')` in the loop of your `fib()` function. Then run your program again and see what prints out. In the comments of your file, describe what seems to happen.

2.2 Exercise 2b - an iterable linked list

Generators make it easy to add iteration to a new sequence type. In Python, any object that appears after the reserved word `in` in a for loop needs to provide an implementation of the `__iter__(self)` method. This method just returns an iterable object that "knows" how to step through the items in the structure.

Try running the included file `a5ex2b.py`. It contains the linked list class we saw back at the beginning of the semester. The test code will fail with the message:

```
TypeError: 'SinglyLinkedList' object is not iterable
```

when it tries to run the second `for` loop.

You will now fix this. Create an `__iter__(self)` method. This method must return a generator that steps through each item in the linked list. It's actually very simple to do, just write standard loop to iterate over the nodes in the list, and yield each value. Once you've done this, the file should run and the last `for` loop should work!

What to hand in

A zip file containing:

- `a5ex1a.py`
- `a5ex2b.py`
- `a5ex2a.py`
- `a5ex2b.py`