Intermediate Python Programming

Description: Iterators, Generators and Function Currying

Instructions

- Please follow all instructions closely and read this in its entirety before writing ANY code. There is no penalty for going "above and beyond" the assignment requirements. However, failure to meet all required parts can affect your grade. Please consult the rubric for each section.
- Either raw Python (*.py) or Jupyter Notebook (*.ipynb) files are acceptable for this lab.
- A single file may be used for this project.

Background

Part of this lab requires a basic understanding of infinite series. An infinite series is a mathematical formula that can never be fully calculated, but as more and more iterations are performed the result converges on a value. In this lab we are going to be using two well-known infinite series to estimate the value of pi (π), which is roughly 3.14159265.

Leibniz Series

The first series we will look at is the Leibniz series (named after Gottfried Leibniz) which has alternating additions and subtractions of fractions to approximate pi.

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} - \cdots$$

It should be relatively simple to guess the next few terms. The numerator is always 4 and the denominator is the next odd number in the series. The terms alternate between being added to the result or subtracted. This is one of the simplest series to implement, but it takes many millions of iterations before the value starts looking like pi. After 10,000,000 iterations, the value is about: 3.1415925535891397 (the red being where the value diverges from pi)

Nilakantha Series

Like the Leibniz series, the Nilakantha series (named after Kelallur Nilakantha Somayaji) also calculates pi by using a repeating pattern of adding and subtracting fractions where the numerator is also always 4.

$$\pi = 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \frac{4}{8 \times 9 \times 10} + \frac{4}{10 \times 11 \times 12} - \cdots$$

This series is also easy to calculate but has the advantage of converging on pi much quicker. After 10,000 iterations, the value is: 3.14159265358956 (again, red indicates where it diverges). As you can see it gets closer to pi in FAR less time than Leibniz.

Part I

To make this project easier, you can utilize the Fraction module which will have a much higher resolution then using floating point types. So, use the following import command:

```
from Fraction import *
```

Additionally, we will use the Decimal data type which has more precision than floats:

```
from decimal import *
```

Next, we will need some module-level variables for calculations:

Pi out to 50 digits:

```
pi50 = Decimal("3.14159265358979323846264338327950288419716939937510")
```

Number of iterations to perform:

```
iterations = 1000000
```

Create an iterator class to calculate the value of pi using the Leibniz series:

```
class LeibnizPiIterator:
```

- __init__() signature:
  `def __init__(self):`
  - For this method, simply use a pass command. All of the setup will occur in __iter__.
- __iter__() signature:
  `def __iter__(self):`
  - This method initializes the values we will need for the iterator
  - Create an instance variable called self.fraction and assign to it a Fraction object with a numerator of 0 and denominator of 1. This will represent the running total for the series.
  - Create an instance variable called self.n and assign 1 to it. This will represent the denominator to be used in the next iteration (see documentation above).
  - Create an instance variable called self.add_next and assign to it True. This is a Boolean value indicating if the next iteration will be an add or subtract.
  - Return self
- __next__() signature:
  `def __next__(self):`
  - This method is where the work is done for each iteration.
  - If self.add_next is True then the next value is to be added to self.fraction. Otherwise, subtract it.
  - The next value is a new Fraction object with the value 4 / self.n
  - After updating self.fraction do the following:
    - Change the value of self.add_next to its opposite.
    - Add 2 to self.n
    - Return self.fraction.value

Test the Iterator:

- Set iterations to 100,000 (don’t use the thousands separator)
- Create a new LeibnizPiIterator object and loop through it “iterations” times.
    - You can implement using a for loop or while loop.
- Report the result of the final value to the terminal
- Then calculate the difference between the final value from the iterator and pi50 and report that value to the terminal as well.
- Change the iterations variable to 10,000,000 and re-run the test (it will take much more time, maybe get a coffee!)

Sample Output:

```
pi after 100000 iterations: 3.14158265358985315979006807645440766789826367641700
Difference: 0.00000999999999400786725753068250950000000000000000
pi after 10000000 iterations: 3.14159255358913974898979182536369204750639121844740
Difference: 0.00000010000065348947285155791581080000000000000000
```

Scoring:

Criteria	**Points**
Lab setup (imports, variables, etc.)	5
LeibnizPiIterator class definition	3
__init__() implementation/functionality	2
__iter__() implementation/functionality	5
__next__() implementation/functionality	10
Tester implementation/functionality	5
TOTAL	**30**

Part II

Now you are going to create a generator that calculates pi using Nilakantha's Series.

```
def NilakanthaPiGenerator():
```

- In the function, create the following variables:
    - fraction: set it to a new Fraction object with the numerator set to 3 and the denominator to 1. This represents the current value of pi after each iteration.
    - num: set it to 2. This represents the first factor in the denominator calculation at each iteration.
    - add_next: set it to True. This represents whether or not to add or subtract the next value to fraction.
- Create an infinite loop. Inside the loop:
    - Calculate the next Fraction object to add/subtraction to fraction. Store it in a variable called operand.
    - Add or subtract operand to fraction based on the value of add_next
    - Flip the value of add_next
    - Add 2 to num
    - yield fraction.value

- Next, test this generator in much the same way as the iterator created in the first section. You must test it at 100,000 and 10,000,000 iterations. The output should show how much closer to pi this series gets.

Scoring:

Criteria	**Points**
NilakanthaPiGenerator function definition	1
Variable setup	4
Loop implementation	5
Next value yield calculation	15
Tester implementation/functionality	5
TOTAL	**30**

Part III

Use currying and composition to calculate distance conversions.

- The first thing you will need in this section is the compose() function which accepts a series of functions and reduces it to a single function call. You can write the compose function any way you like, but here are two common implementations:

```
def compose(*functions):
    return reduce(lambda f, g: lambda x: g(f(x)), functions)
```

OR

```
compose = lambda *F: reduce(lambda f, g: lambda x: g(f(x)), F)
```

- Next, implement a series of functions to convert from one unit of measure to another. The idea is that we have the ability to convert from any one unit to another by tying a combination of these methods together in the right order.
    - Use the following chart to implement the methods.
    - The conversion factor is the value you must multiply by to convert from the input value to the output result.
    - Each method accepts a single value and returns the resulting calculation.

Function	Description	Conversion Factor
milesToYards	Converts miles to yards	1760
yardsToMiles	Inverse of milesToYards	0.0005681818181818
yardsToFeet	Converts yards to feet	3
feetToYards	Inverse of yardsToFeet	0.3333333333333333
feetToInches	Converts feet to inches	12
inchesToFeet	Inverse of feetToInches	0.0833333333333333
inchesToCm	Converts inches to centimeters	2.54
cmToInches	Inverse of inchesToCm	0.3937007874015748
cmToMeters	Converts centimeters to meters	0.01
metersToCm	Inverse of cmToMeters	100
metersToKm	Converts meters to kilometers	0.001
kmToMeters	Inverse of metersToKm	1000
kmToAu	Converts kilometers to astronomical units (average distance from Earth to Sun)	0.000000006684587122268445
auToKm	Inverse of kmToAu	149597870.700
auToLy	Converts AUs to light years (distance light travels in a year in a vacuum)	0.00001581250740982065847572
lyToAu	Inverse of auToLy	63241.07708426628026865358

- Once all of the functions are implemented, you can start tying them together using compose(). To convert from miles to feet, you would use compose in the following manner:
  `milesToFeet = compose(milesToYards, yardsToFeet)`
- Now you can execute the milesToFeet() function variable passing in the number of miles you want to convert to feet.
- To go from miles to light years will require many functions to be chained together.

Here are the conversions you must perform in this exercise and the approximate results (your results may vary, but not by much):

- Convert 2 miles to inches (result: 126720)
- Convert 5 feet to meters (result: 1.524)
- Convert 1 meter to inches (result: 39.37007874015748)
- Convert 10 miles to kilometers (result: 16.09344)
- Convert 1 kilometer to miles (result: 0.6213711922373341)
- Convert 12.7 kilometers to inches (result: 500000.0)
- Convert 500000 inches to kilometers (result: 12.7)
- Convert 9,460,730,472,580,800 meters to light years (result: 1)

Extra Credit:

There are a lot of missing metric system conversions in the list. Implement the following and demonstrate their usage in your code (you will need to look up the conversion factors on your own):

cm ➔ mm (centimeter to millimeter)

mm ➔ cm (millimeter to centimeter)
mm ➔ µm (millimeter to micrometer)
µm ➔ mm (micrometer to millimeter)
µm ➔ Å (micrometer to Angstrom)
Å ➔ µm (Angstrom to micrometer)

Scoring:

Criteria	**Points**
Implement each conversion method (2 points each)	32
Use compose() to implement the required conversions	8
TOTAL	**40**
EXTRA CREDIT	
Extra conversions	5

Hints and Tips:

- In parts I and II, you can put periodic output statements in your loop so you can see your progress. After all, 10,000,000 iterations takes a long time!
```
if counter % 100000 == 0: print(x)  # print the value every 100000 iterations
```
- Recall, to use the reduce function, include the following import statement:
```
from functools import reduce
```