Chapter 3

Lab 3: Sequences and Series

3.1 Introduction

In this lab, we will apply for loops to study sequences and series. If you already have programming experience, you can complete a challenge problem in lieu of some of the other problems: see the final problem for the details.

3.2 Preparation

This lab will rely on the material from Sections 1.2.1 to 1.2.4 of the Scientific Python Lecture notes. Most of the problems can be completed using a simple functions containg a single for loop, such as in this function:

```
  def loop(n):
    for i in range(n):
        print(i)
```

To run the code in the function, you call the function, usually in a different cell:

```
loop(5)
```

△ **Jupyter Notebook Exercise 3.1:** Create a new function:

```
def mult(a,n):
   # your code here ...
```

that prints the first n multiples of a. For example `mult(3,4)` should output:

```
3
6
9
12
```

In future problems, we'll describe this output simply as 3, 6, 9, 12. We won't be picky about whitespace unless we discuss it explicitly. One way to complete this is to use the three arguments of `range(start,stop,step)`.

3.3 Fibonacci Sequence

The Fibonacci numbers are a sequence of numbers satisfying the recursion relationship:

$$F_{n+2} = F_n + F_{n+1}$$

with $F_0 = 0$ and $F_1 = 1$. The sequence is:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

This sequence can be generated numerically by an algorithm such as this one:

```
fa := 0 # set fa to 0
fb := 1 # set fb to 1
repeat n times:
    fc := fa + fb
    print fc to screen
    fa := fb
    fb := fc
```

Note that this is not python syntax. What is the importance of the last two lines? Would the algorithm work if we exchanged their order?

△ **Jupyter Notebook Exercise 3.2:** Use the algorithm described above to implement a new function `fib(n)` which prints the next n Fibbonacci numbers after the initial 0 and 1.

3.4 Arithmetic Series

The finite arithmetic series

$$S_n = \sum_{k=0}^{n}(a + kd) = a + (a + d) + (a + 2d) + \ldots + (a + nd)$$

sums to the average of the first and last terms times the number of terms:

$$S_n = (n + 1)\frac{a + (a + nd)}{2} \tag{3.1}$$

We will assume $a = d = 1$ and calculate this finite series numerically using the following function:

```python
def arith(n):
     sum = 0
     for k in range(n):
          sum = sum + 1 + k
          #print("k: ", k, "\t sum: ", sum)
    return sum
```

Type in this function and see how it works by uncommenting the print statement (delete the # symbol that starts a comment) and calling it as `arith(5)`. The use of print statements in a loop like this or at each stage of a calculation is a simple, effective and classic debugging technique. You test your code with the print statements included, keeping n small so you don't fill your whole screen with output. Once your code is working, you comment out the unneeded print statements so that the interpreter ignores them and you no longer see the unneeded output. Why not just delete them? You can, but experience shows that if you do, you will need the line again shortly!

△ **Jupyter Notebook Exercise 3.3:** Obtain the sum of the first n terms of arithmetic series with `sum = arith(n)` for three different values of n. Each time, show that sum returned by the function matches the expected sum.

3.5 Geometric Series

The geometric series

$$\sum_{k=0}^{\infty} ar^k = a + ar + ar^2 + ar^3 + \dots$$

converges for $|r| < 1$ to:

$$\frac{a}{1-r}. \tag{3.2}$$

We will demonstrate this numerically.

△ **Jupyter Notebook Exercise 3.4:** Implement a function `geom(a,r,n)` which calculates sum of the first n terms of the geometric series with kth term ar^k. Show that it agrees with Eqn. 3.2 for $a = 2$, $r = 0.5$ $n = 100$.

△ **Jupyter Notebook Exercise 3.5:** Call you geometric series function again for $a = 3$, $r = 0.8$ and $n = 100$. Compare with the expected output calculate within python and with pencil and paper. Do they agree exactly? If not, do they agree within the floating point precision?

△ **Jupyter Notebook Exercise 3.6:** Now compare your calculated sum with Eqn. 3.2 for $a = 1$, $r = -0.9$ $n = 100$. How is the agreement? Increase n and see what happens. Why do you suppose this series is slower to converge?

3.6 Refinements

There are a few refinements you can make to your code. Don't change your working code from previous examples! Instead, copy the previous version to a new cell and make your refinements there. You don't even need to change the name of the function, Python will happily overwrite the old function implementation when it reaches the cell with the new version. Make these code improvements:

△ **Jupyter Notebook Exercise 3.7:** Improve your Fibbonacci function so that prints the first n numbers including the initital two numbers "0" and "1". Make sure it works properly for $n = 0$, $n = 1$, $n = 2$, and so on.

△ **Jupyter Notebook Exercise 3.8:** Extend the Arithmetic series function to include parameters a and d. Show that it works.

3.7 Maclaurin series

A Taylor series is an expansion of of a function about a point based on the derivatives at that point. A Maclaurin series is a special case of a Taylor series, which is expanded about 0. It is defined by

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 \ldots$$

This is equivalent of approximate a function with a polynomial. Larger is the polynomial, better is the approximation even at points of the function far away from the referent point.

Let's star defining a function `f(x) = cos(x)`. For this exercise, we will use the numpy package that we will use extensively later in the course.

```
import numpy as np
def myfunc(x):
    return np.cos(x)
```

Let's now define a second order polynomial that approximate this function (around 0) and let's write a python function with this polynomial. To find this function you compute the derivatives of f(x) and can use the Maclaurin series:

$$P(x) = 1 - \frac{1}{2!}x^2$$

```
def polynomialorder2(x):
    return  1 - 1/2. * x**2
```

△ **Jupyter Notebook Exercise 3.9:** Compute the derivative of the function f(x) around the point 0 and define the polynomial that approximate the function f(x) with order 4, 6 and 8. Define a new function for each of these polynomials.

△ **Jupyter Notebook Exercise 3.10:** Investigate how good is your polynomial or second order to approximate f(x) = cos(x) when you start to go far away from x=0. In particular, fix a value for x (x = 1) and compare in absolute value the difference between `f(x) - polynomialorder2(x)`

```
x = 1
print(np.abs(myfun(x) - polynomialorder2(x)))
```

△ **Jupyter Notebook Exercise 3.11:** Investigate how good are the polynomial of orders 2 (P2), 4 (P4), 6 (P6) and 8 (P8) to approximate the function f(x). Find how far can you go with x and still have difference between f(x) and your polynomials (P2,P4,P6,P8) < than .1. You can do this by trial and error, but try to find the x value corresponding to a .1 error within .1.

3.8 Fibonacci Integer Right Triangles

Starting with the number 5, every second Fibonacci number is the length of the hypotenuse of a right triangle with integer sides. The first two are:

$$5^2 = 3^2 + 4^2$$

and

$$13^2 = 5^2 + 12^2.$$

Furthermore, from the second triangle onward, the middle side is the sum of the lengths of the sides of the previous triangle, for example:

$$12 = 3 + 4 + 5.$$

△ **Jupyter Notebook Exercise 3.12:** Explicitly verify these properties for the first four Fibonacci integer right triangles.