

# MATH 580A 6/21 Notes

## 1 What is the Graduate Computing Seminar?

This is an experimental summer course that we're running for the first time (and hopefully not the last!). The goal will be to learn fundamental programming concepts, be comfortable working in Python, and then do a whirlwind tour of some areas of mathematics that benefit from the inclusion of programming as a tool in our toolbox.

We'll be meeting twice a week in C-38 on Tuesdays and Thursdays from 11:00 AM - 12:30 PM. There will be assignments each week, alternating between a set of multiple small programming problems and a single larger project. Grading will be credit / no credit and essentially based on whether or not you turn in something for the assignments; don't stress out about the grading, but the assignments should be fun! Different people will have different definitions of fun of course, but I hope that the assignments touch on a lot of topics and feel rewarding to implement.

## 2 Python Setup

We will be programming in Python 3.8 on <https://repl.it> which will handle all the environment set up for you and ensure that your code is backed up online. Python is a widely used modern programming language. It is *interpreted* as opposed to *compiled*, which means that the code doesn't get turned into machine code (i.e. 0's and 1's or assembly code). Instead, there is a Python *interpreter*, which is a separate program that reads your Python code, parses it, and executes the code you write. In general, interpreted programming languages tend to run slower since the interpreter needs to spend time parsing and then executing the code on your behalf. In order to speed up the actual running of your code, some projects on repl.it will run your code using PyPy3 which does something called JIT (just in time) compilation to speed up how it runs your code.

You may also wish to set up Python on your own computer for any projects you do or just to have a sense of how the installation process goes. For that, take a look at the official Python website instructions on <https://wiki.python.org/moin/BeginnersGuide/Download>.

Python itself comes with a rich *standard library* which is a suite of code that adds a lot of functionality. For example, the standard library in Python allows you to do things ranging from mathematical operations such as computing the greatest common divisor of a set of numbers or iterating over the Cartesian product of two sets to more pragmatic things such as reading CSV (comma separated value) files or drawing graphics using the Turtle library. More pre-written code is bundled in *packages* and is available from something called the Python Package Index (PyPI).

### 3 Variables and Flow Control

A program is essentially an ordered set of instructions for the computer to execute. For example, an instruction might be to add two numbers together and return the result. Another instruction might be to find the first occurrence of a 0 in an  $n \times n$  matrix.

Intermediate results get stored in *variables* which you can think of as locations in a large warehouse (i.e. the computer's memory or RAM) and you can store objects at those locations. Variables are not intrinsically tied to the values stored in them and you can reassign their values.

Instructions you can tell Python to run generally consist of two types. One are *operators* which are symbols like  $+$ ,  $-$ ,  $*$ ,  $/$  that handle very fundamental operations like arithmetic. The other, more general category of instructions are called *functions*. A function can take a certain number of *arguments* and compute a result based on all of them. Traditionally, the first program introduced in books and such is usually called "Hello World!" and just consists of outputting the phrase "Hello World!" onto the screen. We're going to do just a little more interesting, namely output the (approximate) solutions to the quadratic equation  $\pi x^2 + x - e = 0$ . This program is shown in Listing 1.

```

1 import math
2
3 x = (-1 + math.sqrt(1 + 4*math.pi*math.e))/(2*math.pi)
4 print("Solution 1:", x)
5
6 # Note that you can use ** for exponentiation as well.
7 x = (-1 - (1 + 4*math.pi*math.e)**0.5)/(2*math.pi)
8 print("Solution 2:", x)

```

Listing 1: Solutions to  $\pi x^2 + x - e = 0$ .

Let us break down this example. Line 1 has an `import` statement. A lot of functionality in Python is in its standard library and in order to make use of them, you need to *import* them in your file to load them for use. In this example, we are importing the `math` library which provides access to the square root `sqrt()` function and the mathematical constants `pi` and `e`. Lines 3 and 7 are the main instructions for calculating the solutions to the quadratic equation. We use a mix of operators for addition, subtraction, and division as well as *calling* the function `math.sqrt()`. The right hand side of the equation is one big *expression* that Python evaluates and stores the result of in the variable `x`. In line 7, we demonstrate the built-in Python operator `**` for exponentiation instead of `math.sqrt()`. Notice that in order to assign a value to a variable we have to write the variable name on the left hand side and the value we're assigning to it on the right hand side. On line 6, there is a *comment*. Any line prefixed with a `#` is a comment and won't be interpreted as code. It is a way to add some documentation and explanations in your program. Finally, lines 4 and 8 called the built-in `print()` function that displays the text `Solution 1:` or `Solution 2:` followed by the value in the variable `x`. Text inside quotation marks are called *strings* and represent the literal text itself as opposed to say variable names.

Let us take a look at another example program, one that attempts a crude approximation to  $\pi$  using the identity  $\pi = 4 \cdot \arctan(1)$  and the Taylor series expansion for  $\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots$ . This program is in Listing 2.

```

1 pi = 4
2 pi = pi - 4/3
3 pi = pi + 4/5
4 pi -= 4/7
5 pi += 4/9
6
7 print(pi)

```

Listing 2: Approximating  $\pi$ .

There are a few things of note here. First is that even though there is a `pi` variable in the `math` module, we can define our own `pi` variable. In addition, our variable names don't have to be a single character. They can be any combination of letters, numbers, and underscores. Next, observe that `pi` appears on both sides of the `=` in line 2. The way variable assignment works is you use the current value of `pi` to evaluate the right hand side *and then* assign that value to the variable on the left hand side. So in line 2, the right hand side is equivalent to  $4 - 4/3$  since `pi` has value 4 at the point in the program. Similarly, the right hand side in line 3 is equivalent to  $4 - 4/3 + 4/5$ . Lines 4 and 5 introduce some shorthand operator. Namely, `pi -= 4/7` is equivalent to `pi = pi - 4/7`. Similarly, `pi += 4/9` is equivalent to `pi = pi + 4/9`.

Our programs have shown that we can use programs to help us do computations, but so far they seem like slightly fancier calculators. We can do a lot more by learning to define our own *functions* and using some flow control tools such as `if-else` statements and `for` loops. We will first define our own general function for solving quadratic equations in Listing 3.

```

1 import math
2
3 def solve_quadratic(a, b, c):
4     if a == 0:
5         if b == 0:
6             return
7         else:
8             return -c/b
9     else:
10        discriminant = (b**2 - 4*a*c)**0.5
11        return (-b + discriminant)/(2*a), (-b-discriminant)/(2*a)
12
13 print(solve_quadratic(1,2,1))
14 print(solve_quadratic(1,0,1))
15 print(solve_quadratic(math.pi, 1, -math.e))
16 print(solve_quadratic(0,5,4))
17 print(solve_quadratic(0,0,4))

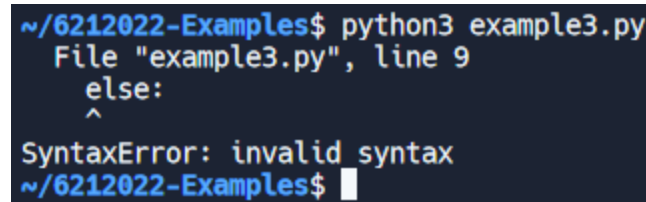
```

Listing 3: A General Quadratic Equation Solving Function.

Lines 3 through 11 are how we *define* a custom function called `solve_quadratic()`. This function has 3 *arguments* which are the input values it takes. These three arguments will be

stored in variables called `a`, `b`, and `c`. Next, lines 4 through 11 are called the *function body* and describe what our function does.

Before discussing how the function body is written, let us take a small detour to talk about whitespace in Python. In Python, the whitespace in your code matters, and the function body has to be indented the same amount throughout. If you indent the `else:` in line 9 a bit further in or don't indent it, then Python will complain. If I try indenting line 9 by two extra spaces, I'll see the error in Fig. 1. The error message is `SyntaxError: invalid syntax` which unfortunately isn't that helpful in this case. However, it provides the line number which is quite helpful with debugging.



```
~/6212022-Examples$ python3 example3.py
File "example3.py", line 9
    else:
    ^
SyntaxError: invalid syntax
~/6212022-Examples$
```

Figure 1: Example Python Error Message.

Returning to the function body, we have 3 cases to handle for a quadratic equation of the form  $ax^2 + bx + c = 0$ , depending on if  $a = b = 0$ ,  $a = 0$  but  $b \neq 0$ , and  $a \neq 0$ . To handle these three cases, in Python we use `if-else` statements. The general syntax is `if <condition>: <statements>` followed by `else: <statements>`. In line 4, we check if the variable `a` is 0. Notice that to compare we use `==` as opposed to `=`, since in Python `=` is used for assigning values to a variable. Lines 5 through 8 are the statements that get executed if `a` is 0 and lines 10 and 11 are the statements that get executed if `a` is not 0. In lines 5 through 8, we further distinguish between the cases where `b` is 0 or not. Finally, lines 6, 8, and 11 have a keyword `return` which ends and returns the result of the function computation. In Python you can return multiple values (or none). Line 6 returns no values in the case where  $a = b = 0$  and the result gets printed out as the special Python value `None`. Line 7 returns one result in the case of a linear equation, and line 11 returns both solutions in the case of a quadratic equation. Python also has built-in support for complex numbers, though it uses the notation `j` instead of `i` for imaginary numbers. Thus, you would write `1-1j` for  $1 - 1i$  in Python.

Lines 13 through 17 show how we call our new function `solve_quadratic()`. Defining functions is very powerful since it lets us abstract away logic into functions. Each time we call our function in lines 13 through 17, we are actually executing lines 1 through 11, but with different values of `a`, `b`, and `c`. If nothing else, it means we don't have to repeat those lines five times with different values.

We will now look at an improvement to Listing 2 for computing  $\pi$ . In that program, we only approximated  $\pi$  using the first 5 terms of the Taylor expansion of `arctan` and had to manually write out the terms we were adding and subtracting. We will use a `for` loop in Listing 4 to let us easily compute  $\pi$  using the first 100 terms of the Taylor expansion.

Line 4 is the beginning of the `for`-loop. The general syntax is `for <variable> in range(<length>): <statements>`. Here we chose to call our loop variable `i` and we loop a total of `num_terms` times. The `for`-loop will execute the statements inside the *loop body* (line 5 in this example) repeatedly, but the value of `i` will be different on each iteration of the loop. In this basic `for`-loop example, the values of `i` will range from 0, 1, ..., `num_terms-1`.

```

1 num_terms = 100
2 pi = 0
3
4 for i in range(num_terms):
5     pi += 4 * (-1)**i / (2*i + 1)
6
7 print(pi)

```

Listing 4: Computing  $\pi$  Using A for Loop.

Like defining custom functions, this allows us to run a large number of instructions in our program without having to explicitly write out all the instructions. We could for example, copy and paste line 5 one hundred times and replace the value of  $i$  appropriately. We would obtain the same result but the code would be very long and tedious.

Work on `exercise1.py`, which involves filling in the function body for two functions. One is to compute  $\pi$  via a Monte Carlo approximation. The other is to compute use the trapezoidal rule to integrate.

Work through any `repl.it` issues and solidify basic understanding of variables, functions, and for loops.

## 4 Recursion and Lists

We've seen that we can call functions and define our own custom functions. We can in fact call a function in its own function body and use this to recursively do computations. Listing 5 is a recursive example of computing the Fibonacci numbers defined by  $F_0 = F_1 = 1$  and  $F_n = F_{n-1} + F_{n-2}$  for  $n \geq 2$ .

```

1 def fibonacci(n):
2     if n == 0 or n == 1:
3         return 1
4     else:
5         return fibonacci(n-1) + fibonacci(n-2)
6
7 for i in range(10):
8     print(f"F_{i} = {fibonacci(i)}")

```

Listing 5: Computing the First 10 Fibonacci Numbers.

Lines 2 and 3 constitute the base cases  $F_0 = F_1 = 1$  and line 5 is the recurrence  $F_n = F_{n-1} + F_{n-2}$ . For example, when `fibonacci(3)` is called, it needs to execute 3 lines of code (lines 2, 4, 5) and then before returning the value, it needs to compute `fibonacci(2)` and `fibonacci(1)`. The latter takes 2 lines of code (lines 2 and 3) while the former takes another 3 lines of code followed by computing `fibonacci(1)` and `fibonacci(0)`. Notice that `fibonacci(1)` had to be computed twice in this example. In general, this implementation of computing  $F_n$  is quite slow since we may have to compute `fibonacci(i)` many times over for different values of  $i$ .

Here we also introduced *f-strings* on line 11, a more convenient way to output text. F-strings require prepending `f` in front of the string. Inside an f-string the contents of curly braces are not printed out verbatim, but instead evaluated as code and the result of that evaluation is printed out. Thus on line 11, `f"F_{i} = {fibonacci(i)}"` evaluates the value of `i` and the value `fibonacci(i)` and prints that.

Listing 6 shows a slight modification that saves the previous values computed to avoid having to recompute them. This technique of speeding up code is known as *memoization*. The memo variable is a data structure called a *dictionary* in Python (often called a *map* in other languages such as C++ and Java) that can store values based on other values called keys. We will discuss this in more depth next time, but you can think of it as a single variable that acts as a box, holding many values. Here we use it to store the previous Fibonacci numbers we've calculated so far. Try modifying Listing 5 to compute the first 100 Fibonacci numbers and observe how it slows down past  $F_{25}$  or so compared to the memoized version.

```

1 memo = {}
2 def fibonacci(n):
3     if n not in memo:
4         if n == 0 or n == 1:
5             memo[n] = 1
6         else:
7             memo[n] = fibonacci(n-1) + fibonacci(n-2)
8     return memo[n]
9
10 for i in range(100):
11     print(f"F_{i} = {fibonacci(i)}")

```

Listing 6: Using Memoization in Computing the First 100 Fibonacci Numbers.

We will cover one more example of using recursion by solving the “Towers of Hanoi” game. In this game, there are 3 pegs and  $n$  disks of radii 1 through  $n$ . Initially, the  $n$  disks are stacked in order of decreasing radii on peg 1. The goal is to move all  $n$  disks to peg 3. At each step, you can move the topmost disk of a peg to another peg, but you cannot place a larger disk on top of a smaller disk. Fig. 2 illustrates a solution for when  $n = 3$ .

There is a simple recursive solution for the Towers of Hanoi. We will let peg 1 be called the *source* peg, let peg 3 be called the *target* peg, and let peg 2 be called the *spare* peg. If  $n = 1$ , then we move the sole disk from the source to the target and are done. Otherwise, we recursively move the first  $n - 1$  disks from the source peg to the spare peg, using the target peg as a new spare peg in this recursion. Then we move the largest disk from the source to the target. Finally, we recursively move the first  $n - 1$  disks from the spare peg to the target peg, using the source peg now as a spare peg in the recursion.

Listing 7 is an implementation of this recursive solution. Ignore lines 16 through 26 for now. Those lines of code print out the disks and pegs in a nice manner. We define a function `solve_towers_of_hanoi()` that takes 5 arguments. The first argument `pegs` is a data structure called a *list*, which as the name suggests, is a list of values. The values have a specific order to them and there be any number of values in a list. We will also discuss this data structure in more detail next lecture.

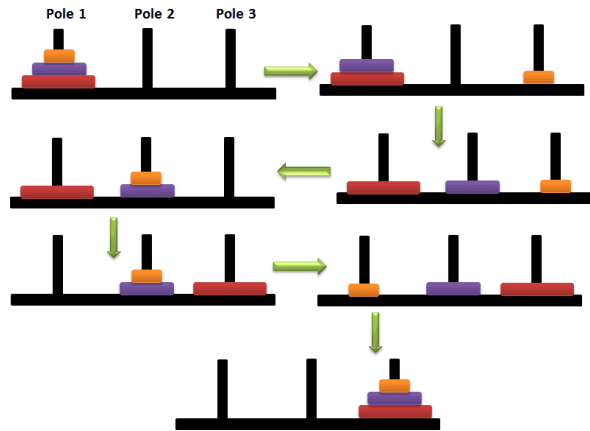


Figure 2: Solving the Towers of Hanoi for  $n = 3$ .

Here, `pegs` is in fact a list of 3 lists. That is, there are 3 values in `pegs` and each value is itself a list. The sublists contain the numbers 1 through  $n$ , representing the disk radii. The next three arguments specify which peg numbers are the source, spare, and target pegs respectively. The first time the function gets called on line 32, these values are 0, 1, and 2. In Python, lists are what are called 0-indexed, so the first value in the list is the 0th one, the second value is the 1st one, and so on. Finally, `depth` is the value of  $n$  used at that particular recursion step. Notice that the recursive calls on lines 10 and 14 call `solve_towers_of_hanoi()` with `depth-1`.

Let us briefly analyze the recursive implementation now. Lines 5 to 8 are the base case where  $n = 1$ . In this case, we remove the top most disk from the source peg and assign it to `x` by calling `x = source_peg.pop(0)` and moving it on top of the target peg by calling `target_peg.insert(0,x)`. Some data structures or values in Python have functions attached to them and calling those functions operates on that specific value. Functions attached to a specific value or object are called *methods*, and the general syntax for calling a method in Python is `<object>.<function>(<arguments>)`. Here, `source_peg` and `target_peg` are lists and we are calling their `pop(<position>)` and `insert(<position>, <value>)` methods which remove an element and insert an element into the list respectively.

Lines 10 through 14 are similar, except they involve the recursive steps mentioned earlier. Observe the order in which we pass in `source`, `target` and `spare` when making the recursive calls.

We will discuss list comprehensions next time and be able to understand how `print_hanoi()` works. For the time being, you can observe lines 23 through 26 and see how we make use of lists, for-loops, and f-strings.

**Work through exercise 2, computing the  $n$ th prime in a memoized fashion and concatenating two lists.**

```
1 def solve_towers_of_hanoi(pegs, source, spare, target, depth):
2     source_peg = pegs[source]
3     target_peg = pegs[target]
4
5     if depth == 1:
6         x = source_peg.pop(0)
7         target_peg.insert(0, x)
8         print_hanoi(pegs)
9     else:
10        solve_towers_of_hanoi(pegs, source, target, spare, depth-1)
11        x = source_peg.pop(0)
12        target_peg.insert(0, x)
13        print_hanoi(pegs)
14        solve_towers_of_hanoi(pegs, spare, source, target, depth-1)
15
16 def print_hanoi(pegs):
17     # Compute height as the total number of disks
18     height = sum([len(p) for p in pegs])
19
20     # Pad shorter pegs with | before printing.
21     padded_pegs = [{"|"} * (height - len(p)) + p for p in pegs]
22
23     l, c, r = padded_pegs[0], padded_pegs[1], padded_pegs[2]
24     for i in range(height):
25         print(f"{l[i]} {c[i]} {r[i]}")
26     print("-----\n")
27
28     depth = 4
29     pegs = [list(range(depth)), [], []]
30
31     print_hanoi(pegs)
32     solve_towers_of_hanoi(pegs, 0, 1, 2, depth)
```

Listing 7: Solving the Towers of Hanoi Game.