SPISE 2019: Computer Science: Weeks 1 and 2

Matthew Ellison

July 15, 2019

Course Information: First Two Weeks

The first two weeks will be focused on building general skills in the Python language and you'll spend the second two weeks working in small teams (of up to 4) to build a game.

Class Time

Each class (except the first) will begin with a 5-10 min quiz on the material of the previous classes. Then there will be a lecture portion to explain the days topics. For the remaining class time, you'll work on a project set where you'll write programs to practice the day's material.

Homework

You're not expected to have time to complete the project sets in class—the completion of the project sets is the course homework. Your code to complete the project sets should be neat and split into separate python files for each problems. You can work together in small groups to complete the homework, but can't copy another student's code.

The week's project sets will all be due together, and must be submitted by email to mvelliso@mit.edu by 10pm Sunday [1]. The email should have subject line "Project Sets Week [Week #]: [Your Name]" and include a .zip file of your project set code as an attachment. The body of your email should list anyone you worked with on the homework.

Class Materials

Lecture notes, project sets and accompanying files, and project set solutions (after they're due), are posted at the class github, at
`https://github.com/SPISE-CS/SPISE-2019`

Date	Topic	Assignment	Quiz
July 15	Arithmetic, Variables, Strings, Booleans, If/Else	Project Set 1	
July 16	Lists, Tuples, Dictionaries, .txt Files, Loops	Project Set 2	Quiz 1
July 18	Functions, Libraries	Project Set 3	Quiz 2
July 22	Classes	Project Set 4	Quiz 3
July 23	Simulation	Project Set 5	Quiz 4
July 25	Debugging and Error Handling	Project Set 6	Quiz 5

[1] disregard the computer science due dates on the SPISE schedule

Date	Time	Assignment Due
July 21st	10pm	Project Sets 1,2,3
July 28th	10pm	Project Sets 4,5,6

Lecture 1: Arithmetic, Variables, Strings, Booleans, If/Else Statements

What is computer science about?

In this course you're going to learn how to write useful programs and think like a computer scientist.

Since you'll be learning to control computers, let's get oriented with a brief look at a typical computer system.

At the center of our picture we have the central processing unit, or CPU, which is like the brain. The CPU can store and process its own data — or exchange data with connected devices. If you browse the internet using wifi, for example, the CPU is sending data to your wireless port to request webpages, receiving website data from your wireless port, processing that data, and sending images to your display. Of course, there's also the data received from the keyboard as you type and the data received from the mouse as it moves.

At the hardware level, all this data consists of 0s and 1s (*bits*), which are stored in chunks called words (32 and 64-bit words are typical of modern devices). The CPU has many addresses where words may be stored and the CPU hardware supports a *machine language* of commands to manipulate words. For example, a machine language command might be used to move a word at one address to different address; or to add one word to another. It turns out that machine language instructions are themselves stored as words with addresses, and machine languages typically support commands like "execute the command word at address 57 if the word at address 2 is all 0s".

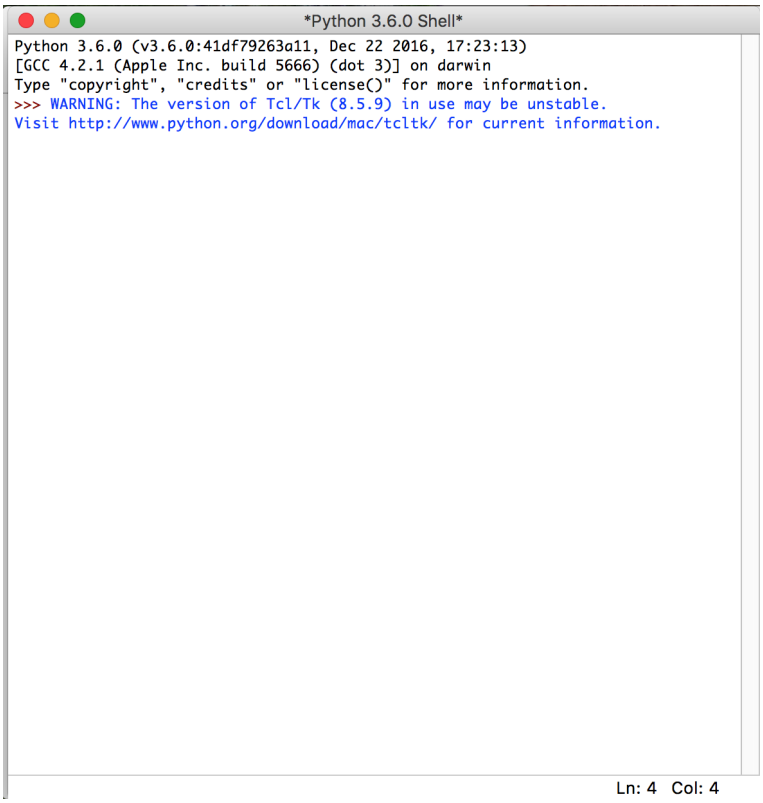
If you're thinking it would be a nightmare to get a computer to do anything useful with machine language, you're not alone. It would be much more convenient for a programmer to be able to write a program in a *higher-level* language, one closer to human language. In the 1950s, computer scientists began to make this possible, developing programs, which, given a program written in a higher-level language, would carry out the program's intent by systematically translating it to machine language — which is all the CPU really understands.

In this course you'll learn computer science with Python, a high level language which was first released by Dutch Programmer Guido van Rossum in 1990. Python is now on version 3.7.4 and is widely used both in university courses and industry. Once you learn to express yourself in one programming language it's much easier to pick up another, so the Python you learn will be useful to you in any future programming—from using C to control embedded systems, to writing apps in Swift, to doing engineering programming in MatLab.

But let's get to learning Python!

First, you'll need to install Python on your computer. Go to `https://www.python.org/downloads/`. If you're on a Mac, you just press the "download Python 3.7.4" button. If you're on Windows, click on the Windows link below the button and scroll down to download the "Windows x86 executable installer" below the Python 3.7.4 section.

We'll be using the IDLE development environment, which comes with the Python package you installed. When you open the IDLE application you find yourself in a window like this:



This is the IDLE Python Shell — kind of like the Terminal on Macs if you've used that. You type in an instruction and press enter to run it. If you've phrased your command properly, the shell will print out the result — otherwise the shell will give you an error message. The shell acts as a test area where you can test your ideas one command at a time (as opposed to writing a full program, where you conveniently arrange all your commands in a file before running them). We'll get to writing and executing full python files in a bit. In the meantime let's take the opportunity to learn a bit of the Python language and practice using the shell.

Arithmetic

We'll start with some arithmetic. Perhaps unsurprisingly, Python can do anything your standard calculator can do. Typing in `4+5`, for example, and pressing enter will cause the shell to print `9`, the result.

```
>>> 4 + 5
9
```

You can use all the standard math operations: `+`, `-`, `*` (multiplication), `/` (division), and `**` (exponentiation).

```
>>> 4 + 5 / 4 * 7 - 9
3.75
```

Python uses the standard order of operations, but if order of operations is a concern, you can use parentheses to be explicit.

```
>>> 4 + 5 / (4 * 7) - 9
-4.821428571428571
```

Python also supports some arithmetic functions you might not find on a calculator, such as the modulo operator (python uses the symbol `%`). `x % y` gives the remainder when x is divided by y.

```
>>> 5 % 3
2
```

Incidentally, some operations on your scientific calculator (for example cos, sin, tan) won't run if you type them in.

```
>>> cos(90)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    cos(90)
NameError: name 'cos' is not defined
```

This is because the function "cos" is not part of the Python language—so I lied by saying that python can do everything your calculator can. But it's only a little lie. People have written a Python library, the math library, which you can use to make the cosine function available (as well as sine, tangent, and many others).

```
>>> from math import cos
>>> cos(90)
-0.4480736161291701
```

Looks like the math library uses radians... The math library is always available with Python, so it's almost as good as having cosine be part of the Python language itself. We'll talk more about libraries—how they work and how you can write your own—in Lecture 3.

Variables and Assignment, Types

Most of the time, after doing some arithmetic, you'll want the computer to keep track of the computed value so you can work with it further. To do this you can *assign* the value to a *variable*. (recall `**` is Python's exponentiation operator)

```
>>> really_big_number = 2 ** 100
```

When you assign a variable, the shell won't print out the value of the variable, but, if you want to see it, you can enter the variable name into the shell.

```
>>> really_big_number
1267650600228229401496703205376
```

You can then use the variable in further commands and the shell will treat it just as if you put in the variable's value (126...76) instead of its name.

```
>>> really_big_number * 10 + 2
12676506002282294014967032053762
```

We can even define one variable in terms of another.

```
>>> tshirts_at_home = 14
>>> tshirts_thrown_out = 2
>>> tshirts_remaining = tshirts_at_home - tshirts_thrown_out
>>> tshirts_remaining
12
```

You can name a variable almost anything you want, as long as it follows the following two rules:

1. The variable name cannot be a reserved word in the Python language (such as `for`, `while`, `if`, `True`, and a few others). Luckily, IDLE turns these words different colors (try it!) so it's not hard to avoid this mistake.

2. The variable name must begin in a letter or underscore (`_`) and continue with some combination of letters, numbers, and underscores. (so I couldn't use the name `t-shirts_thrown_out`, for example, since it contains a hyphen.

In this course, we'll use the typical convention of naming variables as single lower case words, or a number of lower-case words separated by underscores.

So what is `tshirts_remaining`? It's a variable. But what kind of variable, what sort of value does it have? Let's ask Python, using the `type` function.

```
>>> type(tshirts_remaining)
<class 'int'>
```

We'll get to Python's class system later in the course, but the shell is telling us that the value of the variable `tshirts_remaining` is of type `'int'` — an integer. This is because 12 is an integer (a fancy word for whole number). Python has a number of different types. We've already seen numbers that aren't integers, let's find out their type...

```
>>> non_whole_number = .34758
>>> type(non_whole_number)
<class 'float'>
```

Numbers which aren't integers (whole numbers) are of type `'float'`. This is short for 'floating point number', because the number has a decimal point.

Python's type system distinguishes the different types of data you can work with when programming. So far we have whole numbers (`'int'`) and decimals (`'float'`). There are many others types in Python (you'll even learn to build your own types in lecture 4). Today we'll learn two more you can use in your programs — `'str'` (strings of text) and `'bool'` (can either be `True` or `False`).

Strings, Print, and Input

Strings are chunks of text, like `'Eat healthy'`, or `'clean your room!'`, or `'...;a94302'`. More formally, a string is a sequence of characters. `'Eat healthy'`, for example, begins with the character `'E'`, and continues with `'a'`, `'t'`, `' '`, `'h'`, and so on. Note that a space is just another character.

You can assign a variable to have a string value just like we assigned a variable to have an int or float value:

```
>>> current_location = 'Barbados'
```

If you want to check the value of `current_location`, you can just type it into the shell and press enter, like with numbers.

```
>>> current_location
'Barbados'
```

This is a good way to see the value of the `current_location` variable, but the problem is it only works in the shell (not in full programs). Let's learn a way to display a string variable that always works — the **`print`**`()` function.

```
>>> print(current_location)
Barbados
```

Pretty straightforward. In fact, the print function can be used to display the value of most any variable.

```
>>> print(tshirts_remaining)
12
```

I've made it seem like you can only print variables, not strings or numbers themselves; but if it works with the variable, it will work with the value. That's a pretty universal fact in Python.

```
>>> print('encyclopedia')
encyclopedia
```

Really, the print function only knows how to print strings (objects of type `'str'`). When asked to print an `'int'` like 12, it performs a *type conversion* behind the scenes to turn the integer into a string. How does it do that? Well, it makes a string out of the *digits* of the number, concatenating `'1'` and `'2'` to make `'12'`. `'12'` is a string, and the print function knows how to display strings. This might seem like I'm making a mountain out of a molehill, but type conversion is a very useful concept — take the time to understand this paragraph. There really is a difference between the integer `12` and the string `'12'`!

Python also supports a number of operations on strings. One is concatenation, which forms a new string by putting one directly after another. Python uses the `+` symbol to concatenate strings. The syntax (command format) is just like adding numbers:

```
>>> mystery_animal = 'mountain' + 'lion'
>>> print(mystery_animal)
mountainlion
```

Why didn't python add a space between the two words? Because we didn't tell it to add one. A space is a character like any other in a Python string and you must add it explicitly when concatenating strings.

```
>>> mystery_animal_fixed = 'mountain' + ' ' + 'lion'
>>> print(mystery_animal_fixed)
mountain lion
```

Python also supports string *slicing*, which means picking out characters in a string. For example, if we wanted to print the `'o'` in the string `'Barbados'`, we could do the following.

```
>>> country = 'Barbados'
>>> print(country[6])
o
```

As the above example shows, the syntax for selecting a single character from a string is to follow the string variable with brackets containing the position of the chosen character. This position is called the index of the character. But hold on a second! `'o'` isn't the 6th letter of Barbados. Is there a typo? The reason is that Python begins counting at 0, not 1. So the character at index `0` is `'B'`, the character at index `1` is `'a'`, the character at index `2` is `'r'`, and so on.

One final note on strings. When writing programs, you might want to obtain information from a user — such as their name — and store it as a variable. To do this you can use the input function:

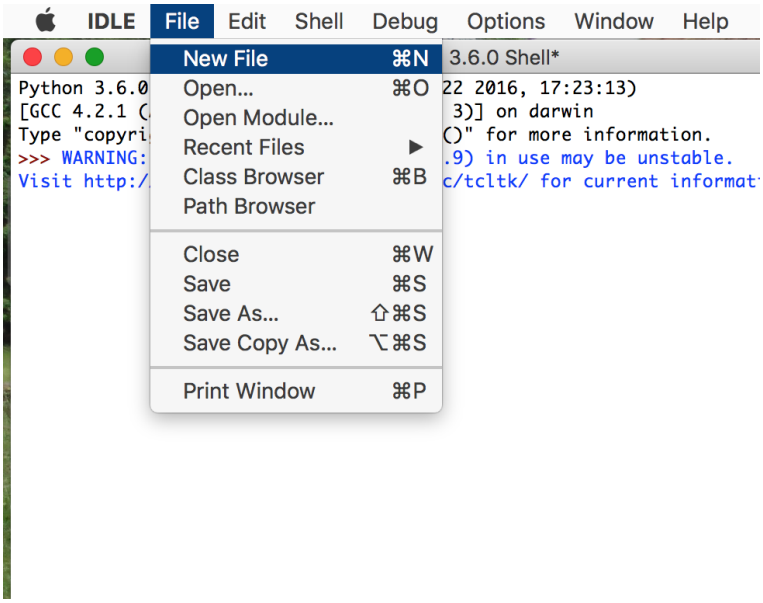
```
>>> name = input('Please enter your name: ')
Please enter your name: Matt             <--- I typed in my name here
>>> print('Welcome ' + name)
Welcome Matt
```

Give it a try. It will prompt you for your name, which you type in and press enter. After that, the entered text will be assigned to the variable `name`, which you can use later in your program.

Interlude: Break Out of the Shell!

As you begin to write larger programs, it will feel awkward executing them one line at a time. The shell is useful for testing that commands work the way you want—but if you're writing a Python program of any real complexity, you'll write it as its own .py file (or even as multiple .py files, but we're not there yet).

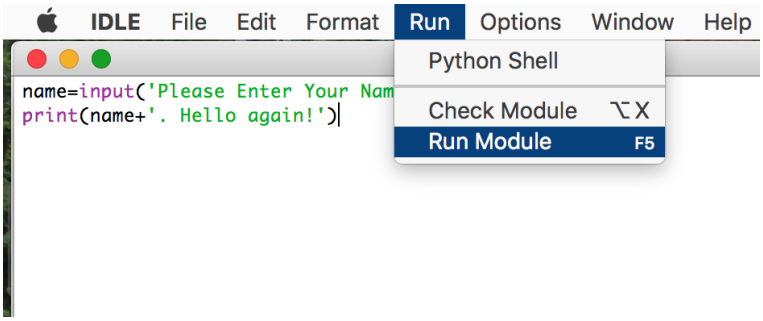
First, open up a new python file.



This will pull up a window where you can write your program. As an example, feel free to type in the following

```
1 name = input('Please Enter Your Name: ')
2 print(name + '. Hello again!')
```

To run your program, select Run —> Run Module



Idle will then prompt you to save your file as a .py file, and once you do so, it will run it in the shell. If Python can't understand your program (for example, if you mistype a command like print), your program won't run and IDLE will try to explain what went wrong in an error message.

This will be our program-writing process for the rest of the course, but don't feel shy about using the shell to test or double-check small ideas! For example, if you're wondering whether the character at index `2` of `'music'` is `'u'` or `'s'`, it's easiest to just plug the following command into the shell.

```
>>> 'music'[2]
's'
```

Booleans and if/else

You've learned a lot already, and we'll finish up today's lecture with one more piece of the Python language — one that allows your programs to make decisions. For example, you might write a program to ask for the user's password, and, *if* it's correct, admit the user. *Else*, the program can print out `'incorrect password'`.

The standard way to make these True/False decisions is with an `if/else` clause (run this as a .py file):

```
1 secret_password = 'bluetoad'
2 entered_password = input('Please Enter The Password to Continue: ')
3 if entered_password == secret_password:
4     print('Correct. Welcome!')
5 else:
6     print('Incorrect Password.')
```

This piece of code begins by asking the user to enter a password with the input function. The entered password is stored as the variable `user_password`. Then, the program compares the entered password to the real password.

This is done by the comparison operator, `==`, which tests whether the two strings are same.

If the comparison operator gives `True` (meaning the strings are the same), the program moves on to the part after `if`. Otherwise, the program moves on to the part after `else`.

The general form of an `if/else` clause is as follows:

```
if [an expression --- either true or false]:
    [commands to follow if the expression is true]
else:
    [commands to follow if the expression is false].
```

Let's dig deeper into the 'an expression—either true or false' bit. The use of `==` to compare two strings provides one example. Another example might be the following:

```
my_number = 10
if my_number % 2 == 0:
    print('your number is even')
else:
    print('your number is odd')
```

The expression `my_number % 2 == 0`, divides `my_number` by `2` and takes the remainder (the `my_number % 2` part does this), then compares the result (either `0` or `1`) to `0`. If they are equal, this means `my_number` is even, and the program says so. Otherwise, the program prints that the number is odd.

So the expression `my_number % 2 == 0` is either `True` or `False`, depending on the value of `my_number`. Such expressions which may be `True` or `False` are known as *booleans* (of type `'bool'`). Python will tell you this, too, if you ask.

```
>>> my_number = 10
>>> type(my_number % 2 == 0)
<class 'bool'>
```

Once the computer evaluates a boolean, it will give either `True` or `False`, and you can assign the value to a variable. If you already know which it should be, you can simply define a variable as `True` or `False` directly (python insists on uppercasing `True` and `False` in this case).

```
>>> its_raining = False
>>> i_have_no_umbrella = True
```

Such true/false expressions may also be built together using `or` and `and` in much the same way as english:

```
>>> my_clothes_are_soaked = its_raining and i_have_no_umbrella
>>> print(my_clothes_are_soaked)
False
>>>
>>> i_have_a_dog = False
>>> i_have_a_parrot = True
>>> i_have_a_pet = i_have_a_dog  or i_have_a_parrot
>>> print(i_have_a_pet)
True
```

To be more explicit, the expression `X and Y` is `True` only if both `X` and `Y` are `True` — and `False` otherwise. The boolean `X or Y` is `True` unless both `X` and `Y` are `False`—that is if `X` or `Y` is `True`, or they're both `True`.

Booleans can also be modified with the `not` keyword. If you put `not` before a boolean `X` it gives a new boolean which is the reverse. So `not X` is `True` if `X` is `False`, and `False` if `X` is `True`.

```
>>> sent_delivery = True
>>> didnt_send_delivery = not sent_delivery
>>> print(didnt_send_delivery)
False
```

These compound booleans, using `and`, `or`, `not` can be put into `if`/`else` expressions in the same way:

```
on_beach = True
have_work = False
if on_beach and (not have_work):
    print('on vacation!')
else:
    print('not on vacation...')
```

Alright, if you've followed all this give yourself a pat on the back! You're well on the way to writing interesting programs. The best way to develop skill as a programmer is to write programs—so let's go onto the first project set.

Lecture 2: Handling More Complex Data

Last time you got a feel for many of Python's basic features: doing arithmetic with integers and floating point numbers, assigning values to variables for later use, working with strings (text), and doing a bit of logic with Boolean True/False expressions.

So far, though, we've only covered how to store one number or chunk of text at a time. Imagine you wanted to write a program to analyze the weather in Barbados. If you had hourly temperature measurements over a range of even a few weeks, this would amount to hundreds of numbers! Or imagine you're writing a program to generate short stories and want your program to have access to an English dictionary consisting of tens of thousands of words. Both these problems would require working with many stored numbers or strings–and it's definitely not feasible to assign each to its own variable. Let's get acquainted with some new Python types which are designed to handle such complexity.

Lists

First we'll look at the list type, which allows you to form a sequence of Python objects (a Python object is anything with a type, such as an integer or string). Take the following example:

```
>>> small_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

The above statement assigns the list with elements `2`, `3`, ..., `29` to the variable `small_primes`. The items in a list are called its *elements*, and you can access the elements in a list by sllicing with their *index* (like with string slicing, remember counting starts at 0!). For example, if we wanted to print the `7` (at index 3), we could perform the following command:

```
>>> print(small_primes[3])
7
```

You might recognize the slicing notation as the same syntax used to pick out a certain character in a string. This is because the writer of Python chose similar syntax to make the language easy to remember. You can also pick out larger segments of a list. For example, the following command constructs a list of the items at index 2 up to (but not including) index 5, and assigns it to a new variable.

```
>>> a_few_small_primes = small_primes[2:5]
>>> print(a_few_small_primes)
[5, 7, 11]
```

It's also frequently useful to add an element to the end of a list, which you can do with the `append` command.

```
>>> common_english_names = ['Mary', 'John', 'Emily', 'Ethan']
>>> common_english_names.append('Susan')
>>> print(common_english_names)
['Mary', 'John', 'Emily', 'Ethan', 'Susan']
```

The two examples of lists so far have contained elements of all the same type (ints in `small_primes`, and strings in `common_english_names`), but there are no rules for what types of objects a list can contain:

```
>>> funky_list = ['cat', 17, ['rutabaga', 'cabbage']]
```

The above list is certainly unusual, containing a string, integer, and even another list—but it's perfectly legal to construct a list like this.

You can even create a list without any elements!

```
>>> empty_list = []
```

Does this list really have nothing in it? We can check with the **`len`** command–which works on all lists, not just empty ones, to give the number of elements.

```
>>> len(empty_list)
0
```

Why would we want to make an empty list? It's actually quite common. The typical reason is that you want your program to build the list for you using `append` — either because you don't yet know what should be in it, or it would be too tedious to type in all the elements by hand. You'll see an example of this later today.

For Loops

Let's move on to `for` loops by way of an example. Suppose we'd like to find the average of the numbers in `small_primes`. One way would be to add them all up and then divide by the number of numbers in the list. Let's put this idea in code.

```
1 small_primes = [2 ,3, 5, 7, 11, 13, 17, 19, 23, 29]
2 running_total = 0
3 for number in small_primes:
4     running_total = running_total + number
5 average = running_total / len(small_primes)
6 print(average)
```

Running this program will then print the average value, `12.9`. So what's happening in the `for` loop? It's running through the `small_primes` list, in order, each time running the code in the loop (in this case just one line), with the variable `number` set to the current element in the list.

We can also use a for loop to build a new list based on an old one. Suppose we wanted to select out small primes ending in 3:

```
small_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
primes_ending_in_3 = []
for number in small_primes:
    if number % 10 == 3:
        primes_ending_in_3.append(number)
print(primes_ending_in_3)
```

For loops aren't limited to just stepping through lists. In fact, any Python object which can be stepped through one at a time can be used in place of a list in a for loop. For example, a string can be stepped through one character at a time. The following code prints the letters of `'hippopatamus'` one at a time.

```
for letter in 'hippopotamus':
    print(letter)
```

Another use of `for` loops is to repeat the code inside a fixed number of times. This is accomplished as shown, using the **`range`** command:

```
for i in range(10):
    print('Running the for loop: ' + 'time ' + str(i))
```

The above code will run 10 times, with `i` set to `0`, `1`, ..., `9`.

While Loops

While we're on the subject of loops, let's take a look at the other type of loop in Python—the `while` loop. The `while` loop keeps on running the block of code inside while a given Boolean expression is `True`.

Take the following piece of code, which repeats a certain arithmetic procedure until `current_number` becomes `1`, after which the program ends.

```
current_number = 10
while current_number != 1:
    # != means not equal to
    #you can add notes like these to your program by using a # at the
    # and then typing the note after. The computer
    #ignores the notes when running the program.
    print(current_number)
    if current_number % 2 == 0:
        current_number = current_number / 2
    else:
        current_number = current_number * 3 + 1
```

If the Boolean expression is always `True`, the while loop will keep repeating forever–an infinite loop! If your program doesn't seem to be stopping, while loops are a good thing to check for errors.

Reading and Writing Text Files

Often times, when writing a program, you'll be working with data from an external source (such as temperature data from a weather station, or a dictionary of words in

the English language). This external data can come in a variety of formats, such as .txt or .csv, and will need to be read in to your program and stored. Later, once your program has completed its work, you might also want to create an output file to give to someone else. Python makes it easy to read and write .txt files — let's get to it!

Suppose we'd like to load a dictionary into one of our programs and are given a .txt file like this (a section of the A's shown below):

```
...
abase
abased
abasedly
abasedness
abasement
abasements
abaser
...
```

First, we'll need to open the .txt file as a file object, which can be done by the **open** command.

```
>>> dictionary_file = open('words.txt' , 'r')
>>> # words.txt is the filename and the 'r' means open the file to read.
```

Next, we can use the read command to scan the file, turning it's contents into a really big string:

```
>>> dictionary_raw_text = dictionary_file.read()
```

Now that we've assigned the raw text to the variable `dictionary_raw_text`, we should close the file to save memory:

```
>>> dictionary_file.close()
```

The string stored to `dictonary_raw_text` takes the form of many words separated by line breaks (a line break is just a character too, `\n` [2]). We can put all the items between the line breaks (the words) into a big list using the `split` command.

```
>>> dictionary_word_list = dictionary_raw_text.split()
```

Now the words of the dictionary have been successfully been brought into the program — stored in a handy list. That's about all there is to reading in a text file. To learn how to write a text file, let's create a .txt file of all the words which are 12 letters long.

First we'll create an output file:

```
>>> long_words = open('twelve_letter_words.txt', 'w')
>>> # twelve_letter_words.txt is the name for the new file,
>>> # 'w' since we'll be writing the file
```

[2] I know it looks like `\n` is two characters, but Python treats it as one.

Next we'll create an empty string which we'll build up with the twelve-letter words we find (and eventually write to the file):

```
to_write = ''
```

Now we'll run through the words in the dictionary to form the `to_write` string:

```
>>> for word in dictionary_word_list:
...     if len(word) == 12:
...         to_write = to_write + word + '\n'
...         # the '\n' is so we have line breaks between the words
```

Now we write the `to_write` string to the output file and close it up:

```
>>> long_words.write(to_write)
>>> long_words.close()
```

That's all there is to it! If you look in the folder where your proram is running, you'll see the .txt file you just created.

Tuples and Dictionaries

We'll close the lecture portion of today's class with two more Python types to handle more complex data. The first is the tuple type. Since you know lists, tuples are really easy. They're just lists which, after you create them, cannot change. Here's an example:

```
>>> my_name_and_age = ('Matt', 21).
```

If I were to then try to modify the element at index 1 (the integer `21`) with the following:

```
>>> my_name_and_age[1] = 22 # happy birthday
```

Python would give me an error since tuples cannot be modified. The formal way to say that tuples cannot be modified is that they are *immutable*. Lists on the other hand are *mutable* — for example, the following works just fine: (incidentally, strings are also immutable)

```
>>> my_name_and_age_list = ['Matt', 21]
>>> my_name_and_age_list[1] = 22
>>> print(my_name_and_age_list)
['Matt', 22]
```

You can also turn a list into a **`tuple`** with the tuple command:

```
>>> my_name_and_age_tuple = tuple(my_name_and_age_list)
```

or vice versa with the **`list`** command:

```
>>> my_name_and_age_list_for_real = list(my_name_and_age)
```

Why would you want to use a tuple instead of a list? For one, they're a bit more memory efficient since they don't reserve space for possible append commands — though this is rarely a concern. The main reason is that they can be used as "keys in a dictionary". Of course this doesn't make any sense without knowing what a dictionary is... Let's talk about dictionaries.

A dictionary is a Python type which can be used to associate pairs of objects. For example, suppose that we're creating a game which has various animals at different points in the x-y plane. We could store the animal positions using a dictionary to associate the animal names with their positions:

```
>>> animal_location_dictionary = {'cat': (3 ,2), 'gerbil': (-2, 3),
                                  'aardvark': (0, 0)}
```

Generally, the dictionary syntax is {`<key1> : <value1>, <key1> : <value1>, ...`}. If you'd like to access the value associated with a key, you can use the slicing syntax as shown:

```
>>> animal_location_dictionary['gerbil']
(-2,3)
```

In a way, dictionaries are a more flexible version of lists. In lists, items are associated with the indices 0,1,..., but in dictionaries the 'indices' (keys) can be any Python object — *with the rule that the keys must be immutable objects*, like integers, strings, floats, or tuples.

You can add a new key-value pair to a dictionary like this:

```
>>> animal_location_dictionary['zebra'] = (14, 20)
>>> print(animal_location_dictionary)
{'cat': (3, 2), 'gerbil': (-2, 3), 'aardvark': (0, 0),'zebra': (14, 20)}
```

As with lists, you can also build an empty dictionary which you can add to later using the **`dict`** command:

```
>>> empty_dict = dict()
```

As a final note on dictionaries, they can also be looped through in for loops (the `for` loop will loop through the keys). Python doesn't provide any guarantee, however, for the order the keys will be looped through (dictionaries said to be *unordered* types).

As an example, take the following code to make a list of all the positions of the animals:

```
1 animal_positions = []
2 for animal in animal_location_dictionary:
3     animal_positions.append(animal_location_dictionary[animal])
```

Lecture 3: Functions and Good Programming Practice

Functions

Suppose you're writing a program where you'd like to print out all text surrounded by hashtags,

```
############
#Like this.#
############
```

It would be a real pain to code in all the hashtags each time, but suppose we could extend the Python language to include the command `print_with_hashtags`, which given a string as input would print it with hashtags, like this:

```
>>> print_with_hashtags('asparagus')
##########
#asparagus#
##########
```

Now of course, `print_with_hashtags` isn't part of the Python language, and your computer won't recognize it — but you can add it in by *defining a function*. Here's code to do it.

```
1 def print_with_hashtags(string_input):
2     line_1 = (len(string_input) + 2) * '#'
3     line_2 = '#' + string_input + '#'
4     line_3 = line_1
5     print(line_1)
6     print(line_2)
7     print(line_3)
```

with this new function defined in your program, you'd be free to use the `print_with_hashtags` command wherever you'd like.

Generally, a function has a name (following the same rules as variables) and takes in some number of inputs (called *parameters*). You might just use a function for its effect (as in the printing effect of `print_with_hashtags`), but you can also have it *return* a value, which can then be assigned to a variable.

For example, the following function takes in a number, doubles it and adds two, and then returns the result:

```
1 def double_and_add_two(x):
2     return 2 * x + 2
```

we could then use the function later to assign the returned value to a variable as shown

```
>>> mystery_number = double_and_add_two(15)
```

`mystery_number` is then assigned the value returned by `double_and_add_two(15)`, which is of course `32`.

Here are two more examples of functions. In this case, the first function is used as part of the second function. Note that once a function returns a value its job is done and it doesn't run any more of its code.

```
def check_prime(x):  # Checks if number x is prime. Returns True or False.
    if x < 1:
        return False
    else:
        for i in range(2, x):
            if x % i== 0:
                return False
        return True

def print_first_n_primes(n):
    primes_found = 0
    number_to_check = 2
    while primes_found < n:
        if check_prime(number_to_check) == True:
            print(number_to_check)
            primes_found += 1 #shorthand for primes_found=primes_found+1
        number_to_check += 1
```

```
>>> print_first_n_primes(5)
2
3
5
7
11
```

As you can see, things are getting a bit more complicated. But notice that functions help to manage that complexity. Defining `check_prime` as a separate function helps to make `print_first_n_primes` easier to understand. Additionally, if you thought of a new way to check that if a number were prime, you could just modify the `check_prime` function without having to touch `print_first_n_primes`.

Importing Functions: Libraries

Writing your own functions is one way to expand the language at your disposal. Another is by importing functions which *other* people have written. You saw one example in Lecture 1 where we imported the cosine function from the math library.

```
>>> from math import cos
```

There are many other standard libraries besides `math`: such as `os` (the operating system library) and `random` (the randomization library). You can even write your own library—it's just a Python file. Here's one (don't worry if you don't understand all the function definitions):

```
# Matt's String Library
# Provides two commands to do fun things with strings
def rotate_string(string_input,rotation_amount):
    # For example, a rotation of 'butterfly' by 2 gives 'lybutterfl',
    # and a rotation of 8 gives 'utterflyb'
    string_output = ''
    current_letter_index = (len(string_input) - rotation_amount)
                           % len(string_input)
    for i in range(len(string_input)):
        string_output += string_input[current_letter_index]
        current_letter_index = (current_letter_index + 1) % len(string_input)
    return string_output
def print_word_box(string_input):
    for i in range(len(string_input)):
        print(rotate_string(string_input,i))
```

Suppose we save this library as `matt_string_lib.py`. Then in a new program *(if the library is in the current folder)*, we could `import` the `print_word_box` function and use it:

```
>>> from matt_string_lib import print_word_box
>>> print_word_box('cat')
cat
tca
atc
```

Functions: Philosophical Aside

What's the use of functions? If you read any academic paper or textbook, you'll typically find yourself confronted with technical language: take the beginning of the paper in which scientists reported their observation of gravitational waves:

"In 1916, the year after the final formulation of the field equations of general relativity, Albert Einstein predicted the existence of gravitational waves. He found that the linearized weak-field equations had wave solutions: transverse waves of spatial strain that travel at the speed of light, generated by time variations of the mass quadrupole moment of the source. Einstein understood that gravitational-wave amplitudes would be remarkably small; moreover, until the Chapel Hill conference in 1957 there was significant debate about the physical reality of gravitational waves." (Abbot et al., 2016)

What is the purpose of technical words such as 'mass quadrupole moment' or 'spatial strain'. Are they just there to confuse the reader at home? Couldn't the author have replaced such terms with their definitions in simpler language? The author *could* have done that, but it would make it harder for the author to express themself and the constant definitions would make it more confusing to grasp the idea and purpose of the text. It would be like writing a complex program without functions.

Just as technical language helps scientists to express and reason about complex ideas, carefully chosen functions make it simpler to express complex operations and make your code more concise and understandable. The power of functions is to develop a specialized language to solve your programming challenges.

Good Programming Practice

Functions are your key to designing and building more complex programs, but large programs come with all sorts of problems that are hard to appreciate without experience. Here are the big ones.

1. You might forget how part of your program works (or even what it's supposed to do)
2. If running the program gives an error, it's hard to track down what caused the issue
3. If someone else is trying to fix your program, they might be hopelessly lost

So far our programs have been pretty small and it hasn't mattered too much if you always call your variables `var_1` ,`var_2`, `var_3`, ..., but now is the time to start to think about writing programs which will stand the test of time (or at least be practice for programs which will stand the test of time). Different people have different ideas about good programming practice, here are a few that most everyone would agree with:

1. Begin by planning out your code. Are there certain functions which would especially useful? Does your problem break into smaller steps which you can implement as separate functions?
2. Use meaningful names for variables and functions
3. If a function is longer than a few lines, write a comment to describe what parameters it takes in and what result it outputs
4. If you write a clever piece of code which someone else might not understand, write a comment to explain it

Lecture 4: Classes

Last time we learned functions — which enable you to expand your language to express more complex *processes*. Today we'll learn another tool, classes, which will also help to manage complexity in your programs — by expanding your language to express more complex *data*.

First, a little vocabulary. In Python, every variable or expression belongs to a type — such as a boolean, integer, or float. A particular example of a type, say the integer `7`, is said to be an instance of the class `Int`. Similarly, the expression `True` is an instance of the class `Bool`.

Now suppose you were interested in writing a program to model the basic genetics of breeding pea plants.

Genetics Background

We'll be considering two traits: flower color and stem length. Each plant has two versions (alleles) of the gene for each of these traits. The possible alleles for flower color are 'white' and 'purple' and the possible alleles for stem length are 'long' and 'short'. The genotype of a plant is the allele pairs it carries (one possible genotype is ('white', 'white') for flower color and ('long', 'short') for stem length, for example). The genotype in turn determines the phenotype of the plant — what it actually looks like. For flower color, 'purple' is dominant, so if the plant has even one 'purple' allele it will be purple. Similarly, if a plant has even one 'long' allele, it will have a long stem. When two plants breed, each passes on one of its alleles to its offspring.

Back to Python

Wouldn't it be nice if you had a new type, not `Int` or `Bool`, but PeaPlant? And the PeaPlant type supported operations such as `get_genotype`, `get_phenotype`, and a breed operator which would take two instances of PeaPlant and give a new instance which was a genetic combination?

All of this can be done in a straightforward way by defining a new Python class, and endowing it with *attributes* (private variables each instance will have, such as the plant's genotype), and *methods* (special functions like `get_genotype` or `breed`, which only apply to the PeaPlant class). With the PeaPlant class defined, we could easily write clear and elegant programs to model pea plants.

The complete code to define the `PeaPlant` class follows. Don't be overwhelmed — it will make sense when you read the explanation after.

```
import random

class PeaPlant():
    def __init__(self, flower_allele_1, flower_allele_2, stem_allele_1,
                 stem_allele_2):
        self.flower_genotype = (flower_allele_1, flower_allele_2)
        self.stem_genotype = (stem_allele_1, stem_allele_2)
    def get_genotype(self):
        return (self.flower_genotype, self.stem_genotype)
    def get_phenotype(self):
        if self.flower_genotype == ('white', 'white'):
            flower_phenotype = 'white'
        else:
            flower_phenotype = 'purple'
        if self.stem_genotype == ('short', 'short'):
            stem_phenotype = 'short'
        else:
            stem_phenotype = 'long'
        return (flower_phenotype, stem_phenotype)
    def __add__(self, other_plant): # Breeding operation
        # Randomly chooses an element of the flower_genotype attribute
        flower_allele_1 = random.choice(self.flower_genotype)
        flower_allele_2 = random.choice(other_plant.flower_genotype)
        stem_allele_1 = random.choice(self.stem_genotype)
        stem_allele_2 = random.choice(other_plant.stem_genotype)
        return PeaPlant(flower_allele_1, flower_allele_2, stem_allele_1,
                        stem_allele_2)

def build_random_pea_plant():
    flower_alleles = ['white', 'purple']
    stem_alleles = ['short', 'long']
    return PeaPlant(random.choice(flower_alleles),
                    random.choice(flower_alleles),
                    random.choice(stem_alleles),
                    ransadom.choice(stem_alleles))
```

OK, so what's all this doing? We begin by importing the `random` library, so we can use its functions to build randomization into the breeding process. Next we define the `PeaPlant` **`class`**. The **`class`** definition begins with the class declaration **`class`** `PeaPlant()`, which lets python know that the following indented code will define a new **`class`**, with the name `PeaPlant`.

Next, we define the `__init__` method (remember method is the term for a *private* function for a class). This method must be defined any time you define a new **`class`**, and it describes how a new instance of a **`class`** is built. In the case of `PeaPlant`, you must create an instance of `PeaPlant` by providing the plant's genetic information as described by the parameters.

For example, you could create a plant like this:

```
>>> sample_plant = PeaPlant('white', 'purple', 'long', 'long')
```

It's just a fact of the Python language that the first parameter of any method must be self, but you pretend that variable spot doesn't exist when you call the function.

The `__init__` definition then specifies that the genetic data is stored in two attributes *(private variables)*, `self.flower_genotype` and `self.stem_genotype`,

which bundle the information in tuples. That's all for initialization.

The remaining methods describe operations that can be done on instances of the `PeaPlant` class. The first is `get_genotype()`. Here's how it would be called on `sample_plant`:

```
>>> sample_plant.get_genotype()
(('white', 'purple'), ('long', 'long'))
```

Notice that there's a new syntax for calling methods of a class, with the instance followed by a '`.`' and then the method. The intention is to reinforce the idea that methods are private to the class and this format must be used.

The `get_phenotype` method works similarly:

```
>>> sample_plant.get_phenotype()
('purple', 'long')
```

The final method defined, `__add__`, is used to define the breeding process. It takes one other plant as a parameter, and returns a newly constructed plant as a genetic mixture of the two plants. Note that this process is where we use the `random` library via the `random.choice` function. `random.choice()` takes in a list or tuple and returns a random item from it. Here's an example use.

```
>>> plant_2 = PeaPlant('purple', 'white', 'short', 'short')
>>> child_plant = sample_plant.__add__(plant_2)
```

How come I gave it an obscure name like `__add__` instead of something more descriptive — like 'breed'? The reason is that I'm defining how the built in Python addition function, '+', will interact with the `PeaPlant` **`class`**. `__add__` is the reserved method name which allows you to do this. So we can use the `__add__` method more conveniently like this:

```
>>> second_child = sample_plant + plant_2
```

There are a variety of special method names like '`__add__`' which enable your classes to interact with built in python functions. These methods, termed 'magic methods' in the Python documentation, also include '`__cmp__`' (so you can compare instances with ==), '`__len__`' (so you can take the 'length' of an instance), and many others. Most magic methods, however, are only occasionally used in practice.

Lecture 5: Simulation

Suppose you flip a coin 100 times. The result of this process will be a sequence of 100 heads and tails, such as HTTHTHHH...HT. Let's define a 'run of size n' to be a continuous sequence of n heads or n tails in a row. The example sequence above, for instance, begins with a run of size 1 (H), followed by one of size 2 (TT), followed by one of size 1 (H), and so on.

What do you think the longest run will be? What's the probability it's more than 8? This is the kind of problem that you wouldn't see in a math class, because — like most real-world problems — it's too hard to work out by hand.

With a computer, though, you can easily find a practical answer. Here's the idea. Using a random number generator (we'll use Python's `random` library), have the computer flip 100 coins in a row and compute the length of the longest run. Then have the computer do it a few more times— say a hundred thousand times total. Then, if we wanted to find the probability that the longest run is longer than 8, we can, to excellent approximation, just have the computer return the fraction of times the longest run was longer than 8. Here's code to do this below:

```
import random

def flip_coins(n):
    # Returns a string of n coin flips, like 'HTTHTHHH...HT'
    coin_tosses = ''
    for i in range(n):
        # random.choice() takes in a list and returns a random element from it
        coin_tosses += random.choice(['H', 'T'])
    return coin_tosses

def compute_longest_run(coin_tosses):
    # coin_tosses should be a string as output by flip_coins().
    # Returns the length of the longest run

    # The procedure is to scan through the string, counting each run length
    longest_run_length = 1
    current_index = 0
    current_run_length = 1
    while current_index <= len(coin_tosses) - 2:
        # If next letter is the same, increment current run length
        if coin_tosses[current_index + 1] == coin_tosses[current_index]:
            current_run_length += 1
            if current_run_length > longest_run_length:
                longest_run_length = current_run_length
        else:
            current_run_length = 1
        current_index += 1
    return longest_run_length

def collect_longest_run_data(coin_flips_per_trial,num_trials):
    '''
    returns an array of length coin_flips_per_trial, where the element
```

```
    at index i is the number of times the longest run was of length i+1.
    '''
    output_array = coin_flips_per_trial * [0]
    for i in range(num_trials):
        coin_toss_sequence = flip_coins(coin_flips_per_trial)
        longest_run = compute_longest_run(coss_toss_sequence)
        output_array[longest_run - 1] += 1
    return output_array
```

Let's save the above as `coin_toss.py`. We'll now use it as a library to find the probability that the longest run will be at least 8.

```
import coin_toss

TRIALS = 100000 # One hundred thousand trials!
longest_run_data = coin_toss.collect_longest_run_data(100, TRIALS)
# And count up the number of times it was more than 8...
trials_more_than_8 = 0
for i in range(7,100):
        trials_more_than_8 += longest_run_data[i]
fraction_more_than_8 = trials_more_than_8 / TRIALS
print(fraction_more_than_8)
```

Running the program, we see the probability of having a run of at least 8 is $\sim$`.31`!

Maybe you're more interested in knowing the probabilities that the longest run will be exactly 1, 2, 3, 4, ..., 100. Let's do that.

```
import coin_toss

TRIALS = 100000
longest_run_data = coin_toss.collect_longest_run_data(100, TRIALS)
for i in range(100):
        run_length = i + 1
        fraction = longest_run_data[i] / TRIALS
        print('Chance longest run is of length ' + str(run_length) +
              ': ' + str(fraction))
```

Running this will then print the desired probabilities:

```
Chance longest run is of length 1: 0.0
Chance longest run is of length 2: 0.0
Chance longest run is of length 3: 0.00031
Chance longest run is of length 4: 0.02722
Chance longest run is of length 5: 0.16408
Chance longest run is of length 6: 0.26483
Chance longest run is of length 7: 0.22826
Chance longest run is of length 8: 0.14805
Chance longest run is of length 9: 0.08138
...
Chance longest run is of length 100: 0.0
```

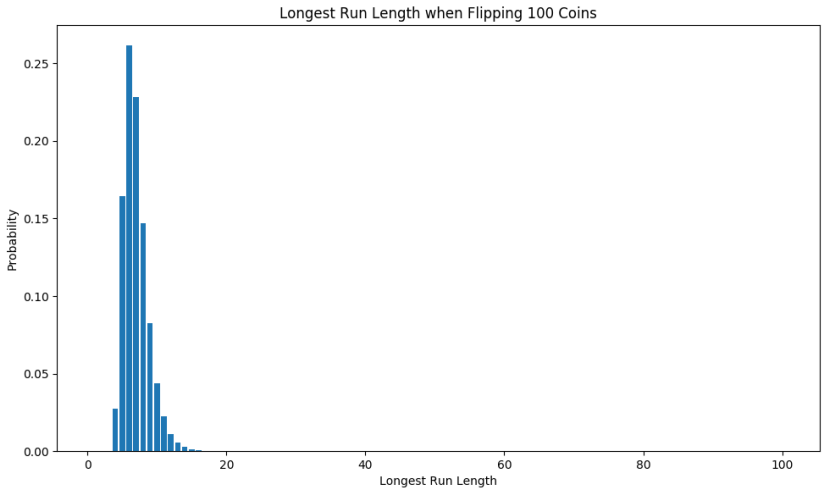
Seeing a big list of numbers isn't the best way to show the results to someone else, though. Wouldn't it be nicer to have a bar graph with run length on the x-axis and the probabilities represented by bar heights?

There's a very popular library to build visualizations like this, `matplotlib`. Let's use it to build the bar plot we want:

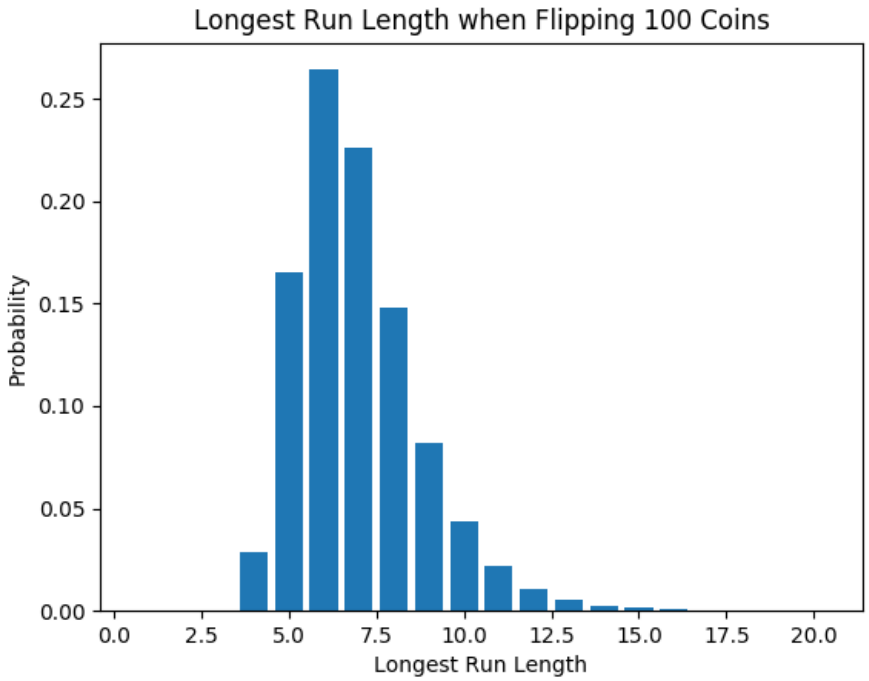
```
import coin_toss
import matplotlib.pyplot as plt
# This just means we can use the shorter string plt instead
# of matplotlib to refer to the library later in the program

TRIALS = 100000
longest_run_data = coin_toss.collect_longest_run_data(100, TRIALS)
fractions = []
for item in longest_run_data:
        fractions.append(item / TRIALS)

# Now we're going to build a plot using matplotlib's bar plot function.
# The first parameter is the list of x-values,
# and the second parameter is the list of heights.
plt.bar(list(range(1, 21)), fractions[:20])
# matplotlib plots also have functions to add titles and labels to make
# your plot nice, let's use a few of those.
plt.xlabel('Longest Run Length')
plt.ylabel('Probability')
plt.title('Longest Run Length when Flipping 100 Coins')
plt.show()
```



We can also focus in on the interesting part of the plot by limiting the input to `plt.bar` to the first 20 elements of each array. This yields the following:



Matplotlib provides the tools to build most any kind of plot you might be interested in. It also has many more functions to give you careful control over the resulting plot (for example, to fix the x-values in the above plot to be whole numbers). You can learn more and read the documentation at their website, matplotlib.org.

Lecture 6: Debugging and Error Handling

As I'm sure you've discovered, programmers spend lots of time fixing issues in their programs.

In this lecture we'll look at a few features Python provides to make your programs more reliable, and some strategies you can use to make debugging easier.

1 Try/Except

Sometimes you'll write procedures which, in some cases, won't work. Suppose for example, you're writing a function `compute_slope`, which takes in an `x` and `y` value and returns the slope of the line through the origin and the `(x, y)`. Here's the obvious implementation:

```
1 def compute_slope(x, y):
2     return y/x
```

What could go wrong with this function? One issue is that it could be called with $x = 0$. In this case, you'll find that Python gives the error `ZeroDivisionError`. What should the function to do in this case? Perhaps you'd like to return the string `'infinity'` if $y > 0$, `'-infinity'` if $y < 0$, and `'undefined'` if $y = 0$. We could modify the function to use an initial `if x == 0`, but let's use `try` and `except` instead:

```
1  def compute_slope(x, y):
2      try:
3          return y/x
4      except:
5          if y>0:
6              return 'infinity'
7          elif y<0:
8              return '-infinity'
9          else:
10             return 'undefined'
```

Python interprets the `try/except` structure by first running the code in the `try` block and then— if any error occurs— stopping without completing the error-causing line and running the except block. Is the function fixed? Certainly it's good enough for any normal use. But suppose some user tries to use the function with nonsensical arguments as in `compute_slope('plant',True)`. What will happen? `True/'plant'` will cause an error, so we'll go into the `except` portion of the code, where the code will compare `True` to `0`. It turns out (because Python defines order relations between seemingly incomparable types) that `True > 0` evaluates to `True` and so the function returns `'infinity'`. This doesn't seem like quite what we want, here's a modification which would be more appropriate if we were concerned with such nonsensical inputs:

```
def compute_slope(x, y):
    try:
        return y/x
    except ZeroDivisionError:
        if y>0:
            return 'infinity'
        elif y<0:
            return '-infinity'
        else:
            return 'undefined'
    except:
        print('Invalid arguments provided to compute_slope')
        return 'error'
```

Here we've used multiple `except` blocks to execute different blocks depending on the error Python might encounter in the `try` block. If it encounters a division by zero error, it will execute the first `except` block, and if it encounters any other issue it will print an error message and return `'error'`.

Is the function fixed now? Once again, it depends on the context the program is being written in. Suppose the function was part of a mission-critical system which enemy hackers were trying to break something by causing the program to throw an error. Could some malicious person cause it to throw an error?

It turns out the answer is still yes—though it would take some ingenuity on the part of the hackers: `compute_slope(0, complex(3, 4))` does it. The program will try to divide the complex number (an obscure Python type) $3 + 4i$ by 0, go to the `ZeroDivisionError` block, and then give an error because $3 + 4i > 0$ cannot be evaluated because $>$ has no meaning with complex numbers (and Python has not defined one).

We could keep adding in more `except` blocks to fix this and other possible issues, but let's stop here. The moral of the story is that it's easy to get carried away trying to make your functions work in every case, and you should match how careful you're being to the context the program and function is being used in. For example, if you know that `compute_slope` will only be called with two numbers, and that x will never be zero, the first implementation is good enough—and if the function were under attack by hackers, we'd need to improve on even the last version.

2 Writing Tests

Writing tests is common practice when developing complex programs. The idea is to make a separate .py file which will import the file your testing, and then use it's functions and classes to run tests. The tests should be written to cover the range of situations which the program will encounter. Here's a simple example with the slope function above, which I'll suppose was saved as `slope.py` (and is known to only receive numbers as inputs):

```
import slope.py

#tries a standard integer usage
print("testing compute_slope(4,5)")
if slope.compute_slope(4,5)==5/4:
    print('passed')
else:
    print('failed')

#tries a negative number and float
print("testing compute_slope(-4,19.23)"):
if slope.compute_slope(-4,19)==-19.23/4:
    print('passed')
else:
    print('failed')

#tests positive infinity case
print("testing compute_slope(0,5)")
if slope.compute_slope(0,5)=='infinity':
    print('passed')
else:
    print('failed')

#tests negative infinity case
print("testing compute_slope(0,-17)")
if slope.compute_slope(0,-17)=='-infinity':
    print('passed')
else:
    print('failed')

#tests undefined case
print("testing compute_slope(0,0)")
if slope.compute_slope(0,0)=='undefined':
    print('passed')
else:
    print('failed')
```

Running the test file and seeing it pass the tests will then give you confidence that the `compute_slope` function is working as desired. If it fails, seeing which particular test case failed can give you insight into how you need to make a fix. Many programmers will actually write their test cases before they've finished writing the actual program, as a sort of testable outline for what the program should do once complete. The programming process in this case looks something like this:

1. Decide on precise specifications for functions and classes.
2. Write test cases to test specifications are met.
3. Actually write the code to implement the functions and classes.
4. Test the code.