Lecture 3

Functions, Modules, and Randomness; String Basics; Input; Errors; `.format()`; `.format` Specifiers

0. Programming Assignment!!!!!

- Put in-line comments, describing variables and non-obvious statements.
- Make sure to read the specifications precisely!
- Read the "Grading and Submission" document for more info.
- Use my templates and my exact file names. So, `triangle.py`, not `triangles.py`, not `jon_smith_triangle.py`.
- List collaborators! Or else, specifically write "None." If you talked to the person next to you, but didn't catch their name, just write "the person who sits next to me in class."
- If you need help with fixing bugs, send me your .py file (or, if that's hard, cut and paste your code into an email). I generally can't fix your bug blind! I'm not clairvoyant -- I don't know if you have two extra spaces on line 10. Screenshots sometimes can also be helpful just because I can look at them on my phone when I'm out, and sometimes I can spot bugs by sight.
- Suggestion: don't wait until your program is complete to run it. Write a little bit, use liberal print statements, run what you have, and see if the output is what you expect it to be.
- START EARLY!

1. Functions, Modules, and Randomness

In general, a ***function*** in Python is any expression that has the general form

In [ ]:
```
FUNCTION SYNTAX:

<fn name>()

or

<fn name>(  <inputs>  )
```

So a function is something that you write with parentheses, that often (not always) has inputs, and which produces some action or value. (Cripes, what a horrible definition. Let's stick with it anyway.)

`print()` is a function: you give it a `str` or a number, and it displays it on the screen.

`len()` is a function: you give it a `str`, and it produces a value: the length of that `str`.

And of course, your favorite math functions are Python functions too! They look slightly different in Python than they do in Calculus class, but I'm sure you can get the gist easily. But: to use these functions, be sure to **import the `math` module**. See below. (For other math functions, use a textbook or Google!)

In [ ]:
```
# EXAMPLE 1a: The math module

# The following line IMPORTS A MODULE: the math module, which contains
# functions beyond basic arithmetic

import math

# Now you can use functions and variables in the math module; just write math.
# in front of whatever you need.

y = 25.0
z = (3.14159)/6 # This is a crappy way of writing PI/6 -- see below.

# math.exp() is the function e^x
print(math.exp(1))
# math.sqrt() is the square root function
print(math.sqrt(y))
# I think you can guess this one
print(math.sin(z))

# math.pi isn't a function -- it's an imported variable.
# Indeed, notice the lack of parentheses.
print(math.pi)
```

Computers are great at following directions, but they are suprisingly bad at picking random numbers. Fortunately, the creators of Python have made tools so that Python can do a pretty convincing job at faking it. To use random numbers, we'll have to import another module: the `random` module.

For now, we'll use two basic ways to generate random numbers:

- `random.random()` will return a random float between 0 and 1. Each float has a roughly equal chance of being chosen.
- `random.randrange(<int1>, <int2>)` will return a random integer that is $\geq$ `<int1>` but less than `<int2>`. Pay attention to the boundaries: `<int1>` can be chosen, but `<int2>` cannot!

In [ ]:
```
# EXAMPLE 1b: Random numbers

import random

# Every time we run this, we'll get different results, but all 3 numbers should be
between 0 and 1.
x = random.random()
y = random.random()
z = random.random()

print(x,y,z)

# Here are some random "dice rolls" -- numbers between 1 and 6 (inclusive).

roll_1 = random.randrange(1,7)
roll_2 = random.randrange(1,7)
roll_3 = random.randrange(1,7)

print(roll_1, roll_2, roll_3)
```

2. String Basics

Basic string operations include +, `len()`, and * as we discussed before, and also the *index operator*, represented by a pair of square brackets. It works as follows: suppose that we write the lines

```
x = "Hi!  Ho? "
print(x[0])
print(x[1])
print(x[2])
print(x[6])
```

`x` is a `str` containing 9 characters (note the 2 spaces in the middle and the one at the end). Then `x[0]` will produce the character `"H"`, while `x[1]` will produce the character `"i"`, and `x[2]` will produce the char `"!"`. And `x[6]` will produce the seventh character (counting spaces), which is `"o"`. Notice the **zero-based indexing**: the first character is labelled 0 rather than 1. This is for historical reasons, but most major programming languages use it, and Reddit will make fun of you if you try to start counting with 1.

Also be aware that if you see numbers in quotation marks, they are being treated by the program as sequences of symbols, rather than numbers. See the next example.

In [ ]:
```
# EXAMPLE 2a: Strings
a = "Hello"
b = "Goodbye"
print(a+b)
print(len(a+b))
print(3*a)

x = "Hi!  Ho? "
print(x[0], x[1], x[2], x[6])
print(x[-2]) # Here's a bonus tip -- if you put in a negative index, it counts from
the end (but not zero-based)

# Watch out here! It doesn't matter that us humans see numbers --
# if they are in quotes, they are treated as strings, NOT numbers.
num1 = "10"
num2 = "20"
print(num1 + num2)
print(num1 - num2)
```

Imagine that I wanted to store the following literal in a single string variable:

```
Evan said "3300 is fun" and we nodded politely.
```

Do you see why that could be tricky?

In [ ]:
```
# EXAMPLE 2b: A troublesome string literal

# Look at the coloring of the code below -- it gives a big hint
# about what the problem is.
sentence = "Evan said "3300 is fun" and we nodded politely."
print(sentence)
```

The quotes, which we want to be characters within the string, accidentally end the string!

There are a few characters that, for one reason or another, are tricky to type. To type these, Python (and many languages) have the so-called ***escape characters***. They are all typed as a backslash(`\`) and one more keystroke; and they are used to represent these tricky characters. Here are a few:

- `\n`: is the newline character. If this is included in a string literal, then when that literal is printed, the slash and n will not be displayed -- in their place, you will see your output with part of it on a new line (the part after the backslash n).
- `\"`: if you want to see the quote mark character, you can use \", and then that quote won't be interpreted as the end of the string.
- `\\`: what if you actually truly want to see the backslash character? You use two backslashes.

In [ ]:
```
# EXAMPLE 2c: Escape characters

sentence = "Evan said \"escape sequences are helpful\" and we nodded knowingly."
print(sentence)

forget = "\nHe also said... \n\n ...actually, I can't remember what else he said.\n
"
print(forget)

slash = "I guess I wasn't \\\\listening//."
print(slash)
```

If you surround your string literals with pairs of double-quotes (`"my usual"`), they have to be typed on a single line of code. But there is an alternative: you can surround them with **triple** double quotes (`"""Like this"""`) -- in this case, your literal is allowed to span many lines. (This is also a trick used to temporarily disable big chunks of code without deleting them.)

In [ ]:
```
# EXAMPLE 2d: Multi-line literals

# This works
multi_line = """This multi-line literal is legal.
(The syntax coloring seems to back that up.)
That's because I started the literal with a triple quote."""
print(multi_line)


# This one won't work.
bad = "Let me just
use single quotes
for a multi-line literal
and see what happens"

print(bad)
```

By the way -- we learned how the computer stores numbers, but how does it store strings? Simplified answer: there is a special, widely used system called ASCII (American Standard Code for Information Interchange), which associates a number to each printable character. For example, `a` is 97, `b` is 98, `A` is 65, `0` (the symbol) is 49, `!` is 33, etc. So, characters can be represented by integers, which we know how the computer stores. So strings can also be stored in binary -- and indeed, all data stored by a computer is encoded in 0's and 1's somehow.

3. Input

Starting now, this class will slowly begin to seem less silly: finally, we'll learn how to make our programs interactive. This means that you, the programmer, can write a program, and then have someone else (henceforth referred to as "the user") run that program, and put in their own input. You can design your programs to do interesting things with that input, even if you don't know what the user will enter in advance!

An ***input statement*** will be a statement of the following form:

In [ ]:
```
INPUT STATEMENT SYNTAX:

<variable> = input(<prompt message>)
```

Here, `<variable>` denotes any variable you like; `<prompt message>` is a string literal or variable which will get printed, which should probably be something like `"Please type something in here:"`; and the rest is typed in exactly as shown.

When your program gets to such a line, here is what happens:

1. The prompt message is printed to the screen.
2. The program's execution pauses, until the program's user types a few characters, followed by Enter.
3. After Enter is pressed, all that the user has typed out (with the exception of the Enter) gets stored to *{variable}*. (Or, more precisely, what the user types gets put into a `str` object, which *{variable}* is then associated to.)

In [ ]:
```
# EXAMPLE 3a: Echo!

message = input("Please give me a phrase to repeat: ")
print(message + " " + message + " " + message)

# What's up with this line yall?
print("Here's the first letter of the phrase: " + message[1])
```

Notice that if you enter `Hey\n` or `5+3` at the prompt, you get exactly what you typed in out: no newlines or evaluation takes place. *User input is interpreted as characters, not code!*

There is one big time when this is truly inadequate: when you want to do math with inputs. If inputs are automatically interpeted as `str`s, then how can you do math with them?

The answer is kind of simple: the functions

`int()` and `float()`.

These functions have a `str` as input, and (if it makes sense) will convert that `str` to the equivalent `int` or `float` value. So for example:

`int("125")` will be the `int` value 125
`float("125.34")` will be the `float` value 125.34
`float("1")` will be the `float` value 1.0
`float("12xu")` and `int("1.23")` are both examples of things that will cause errors.

Of course, use of these functions will only make your program work if the user actually enters a number. You can't force them to comply, so you can only hope the user actually uses your program properly (well, that's not reallllly true, you can do a little more than that, as we'll see later in the course).

With this knowledge, let's fix the following program.

In [ ]:
```
# EXAMPLE 3b: Doubler
# This is a program that asks the user to enter a number, and then print out the re
sult of doubling it several times.

number = input("Enter a number, which I will double several times: ")
print("Doubling once: ", end = "")
print(2 * number)
print("Doubling again: ", end = "")
print(4 * number)
print("Doubling once again: ", end = "")
print(8 * number)
print("Doubling one more time: ", end = "")
print(16 * number)
```

In [ ]:
```
# EXAMPLE 3b': Doubler
# This is a program that asks the user to enter a number, and then print out the re
sult of doubling it several times.

########### Here's the fix: surround input with float()
number = float(input("Enter a number, which I will double several times: "))
###########
print("Doubling once: ", end = "")
print(2 * number)
print("Doubling again: ", end = "")
print(4 * number)
print("Doubling once again: ", end = "")
print(8 * number)
print("Doubling one more time: ", end = "")
print(16 * number)
```

Should you use `int()` or `float()`? Again: if your program really only makes sense for integer inputs, and/or you care about the values input being *exact*, use `int()`; otherwise, use `float()`.

Now, we'll make our first non-silly program. It's still really basic, but I'm sure you could imagine all the same ideas being employed to make a program that could, say, help you compute your taxes.

This program should ask for: the user's name, and 3 scores for input. After getting that, it should print out

```
<name>'s average is  <average of the three scores>.
```

To write this program, you need to have a clear view of the steps that need to take place, and in what order:

- Ask for (and store) name
- Ask for (and store) score 1
- Do the last step twice more
- Compute average
- Print out result in desired format

Also, be aware that you are writing the *program*, *not* the inputs. In other words, once you've written your program, I should be able to walk up to your program, run it, enter my name and my scores and get the correct output.

Finally, why do you need variables here? Again, your program needs them to **remember**: it needs to remember the scores long enough to compute the average of all 3, and it needs to remember the name so it can print it at the end.

In [ ]:
```
# EXAMPLE 3c: Average
# Input name and three scores, print out "[name]'s average is [average of scores]."

name = input("What's your name: ")
score1 = input("First score: ")
score2 = input("Second score: ")
score3 = input("Third score: ")
average = (float(score1) + float(score2) + float(score3))/3
print(name + "'s average is " + str(average) + ".")
```

4. Errors

Let's talk about errors. There are three basic types: ***syntax***, ***runtime*** and ***semantic*** errors.

Syntax errors occur when your program can't be understood by the interpreter. This typically happens because: you've put symbols together in an order the language doesn't recognize: putting more than one variable on the left side of an equal sign, starting a variable name with a number, unbalanced parentheses(!!). Syntax errors occur when Python can't even break down your program into variables and operations without getting confused.

In [ ]:
```
# EXAMPLE 4a: Syntax Errors
# Any of these lines anywhere in your program will cause Python to
# be so confused that it won't even attempt to run your program

# The assignment operator needs a variable on the left side!
4 + 2 = x

# Python sees a 3 at the beginning, and it thinks it's dealing
# with a number; it then gets very confused when it then sees "x"
y = 3x # Try 3*x instead.

# GOLDEN RULE OF PARENTHESES: number of opens = number of closes
# (This is necessary, but not sufficient, for correctness)
y = 2 + ((4 + 5)/(6 - 7)
```

Runtime errors occur when Python can at least figure out what all your variables and operations are, but can't execute them for whatever reason. There are tons of different types of runtime errors; for example:

- `NameError`: when you've referenced a variable prior to definition (arises oftern from typos)
- `TypeError`: when you try to use an operation on an inappropriate type of data
- `ZeroDivisionError`: `/0`, `//0`, `%0` are all bad.

They're called "runtime" because most program checkers won't notice that this is a problem until the program is actually run.

In [ ]:
```
# EXAMPLE 4b: Runtime Errors
# Try removing some of these lines to see what the interpreter yells at you about f
or each one.

# We've seen this one before: types don't match what the + operator expects.
"Hello" + 3

# At the time that print(y) is encountered, there is no y variable yet
print(y)
y = 4

# Typos can be considered a special case of the above: you try to tell Python to pr
int the line, but
# the interpreter thinks you are introducing a new variable with the slightly-diffe
rent name "abce"
abcde = "Look out for the missing letter d on the next line"
print(abce)

# What happens when you divide by 0?
3/0
```

Semantic errors are errors that don't prevent the program from running, but which give undesired (that is to say, wrong) outputs. Here's a typical example.

In [ ]:
```
# EXAMPLE 4c: Temperature conversion
# This is supposed to convert 50F and 60F to Celsius.  It doesn't work -- why?

farentemp = 50
celtemp = (farentemp-32)/1.8
print(farentemp)
print("farenheit is the same as")
print(celtemp)
print("celsius.")

print("")

farentemp = 60
print(farentemp)
print("farenheit is the same as")
print(celtemp)
print("celsius.")
```

In [ ]:
```
# EXAMPLE 4c': Temperature conversion
# This is supposed to convert 50F and 60F to Celsius.  It doesn't work -- why?

farentemp = 50
celtemp = (farentemp-32)/1.8
print(farentemp)
print("farenheit is the same as")
print(celtemp)
print("celsius.")

print("")

farentemp = 60
##### FIX: You need to reassign celtemp, too
celtemp = (farentemp-32)/1.8
#####
print(farentemp)
print("farenheit is the same as")
print(celtemp)
print("celsius.")
```

How do you fix errors of any of these types? That's called ***debugging***, and it's a huge part of programming. There is no easy way to find bugs (often, once you *find* them, *fixing* them is relatively easy). But there are three main tricks we'll use:

1. Look at any errors the interpreter reports -- try your best to read them, and be aware that the error might be on an earlier line.
2. Do walkthroughs of your program, carefully executing lines as the interpreter should, and keeping track of variable values at all moments.
3. Print statements/ variable inspection: you should have an idea of what values all your variables hold at every point in your program, and you can confirm or deny your suspicions by inspecting their values at those points.

Roughly 100% of the code ever written by anybody has had bugs at some point in its development. Here is one of the keys to success in this class:

WHEN YOU ENCOUNTER A BUG, DON'T JUST FIX IT -- UNDERSTAND IT.

(This is useful for life in general, too.)

One more classic debug: suppose you want to swap the values of two variables. I.e., suppose that `x = 3` and `y = 4`, and you want to change that so `x = 4` and `y = 3`. The following code doesn't work. Why? How do you fix it?

In [ ]:
```
# EXAMPLE 4d: Swap

x = 3
y = 4

# Now I want to switch the values (without cheating and just writing x = 4 and y =
3
# directly).  So I try:

x = y
y = x

# What's going to happen when I print these values?
```

Before we go on, a riddle. Let's say you have a mug of hot coffee in your left hand, and a mug of hot tea in your right hand. How do you switch hands (put coffee in the right hand, tea in the left)?

Try to carefully walk through:

After the first two lines, `x` has the value `3` and `y` has the value `4`.

After the third line, `x` changes: so `x` now has the value `4`, and `y` still has the value `4`.

Pay attention to the fourth line! First, you evaluate `x`: as of the last line, that has the value `4`. Then you write that into `y`: so `y` has now "changed" to `4`. Thus, now `x` has `4` and `y` has `4`. Boooo.

We fix this the same we answer the riddle. `x` is holding a hot mug of `3`, `y` is holding a hot mug of `4`. To swap them, we need to put one down on the "table", by which I mean: we'll use a third variable, which we'll call `temp`.

In [ ]:
```
# EXAMPLE 4d': Swap

x = 3
y = 4

# Now I want to switch the values (without cheating and just writing x = 4 and y =
3
# directly).

# First, save the value of x to a new variable
temp = x

# Now, you can write over x, since its value has been saved
x = y

# Finally, you can rescue the old value of x from temp, and put that into y
y = temp

print(x,y)
```

5. Pretty Printing with `.format()`

Before we start learning about control structures, let's discuss a tool for careful printing: `.format()`. I'll start by just showing an example.

In [ ]:
```
# EXAMPLE 5a: Introducing .format()

n = input("Enter a name: ")

# .format() will replace any appearance of {0} in the string with n.
# Notice how you can have one uninterrupted string, which has "blanks" that can be
easily filled in.

hs = "Mr. {0}, that's my name, that name again is Mr. {0}!".format(n)

print(hs)

# Compare that with the more awkward....
print("Mr.", n, ", that's my name, that name again is Mr.", n, "!")
# Notice all the quotation marks, all the commas.
# Also, there are a couple of unintended space.
```

So, `.format()` is a device which helps us achieve what-you-see-is-what-you-get printing when you want to intersperse literal text with variable values. Here's the basics of how you use it.

- First, you type a string enclosed with quotes as usual. Presumably, this string will have some places where you want to fill in the value of some variable or expression. Where those places are, type `{0}` (this will get more elaborate in a moment).
- Directly after the closing quotation, type `.format()`. Inside the parentheses, put the variable or expression that you'd like to fill in. `.format()` is a function which will create a new string, by replacing all instances of `{0}` with the value of the variable/expression, which you can then store directly to a variable or print directly.

In [ ]:
```
# EXAMPLE 5b: Name tag

n = input("Enter a name: ")

# Let's use .format() to allow the program to print a name tag:
# "Hello, my name is [NAME], how are you?"

print( "Hello, my name is {0}, how are you?".format(n) )
```

You can also have multiple different fill ins. In this case, `{0}` will represent the first fill in (the first value within the parentheses of `.format()`), while `{1}` will represent the second, and `{2}` will represent the third, etc.

In [ ]:
```
# EXAMPLE 5c: Multiple fill-ins

x = """The wonderful thing about {1}
Is {1} are wonderful things!
Their tops are made out of rubber
Their bottoms are made out of springs!
They're b{0}, tr{0}, fl{0}, p{0}
{2}, {2}, {2}, {2}, {2}!
But the most wonderful thing about {1} is
I'm the only one""".format("ouncy", "tiggers", "fun")

print(x)
```

In [ ]:
```
# EXAMPLE 5d: Madlibs

pl_noun = input("Enter a plural noun: ")         # patrons
ing_verb = input("Enter a verb ending in ing: ")  # squawking

# Finish this line so that it reads "Sir, your squawking is annoying the other patr
ons.", using .format()
madlib =    "Sir, your {1} is annoying the other {0}.".format(pl_noun, ing_verb)

print(madlib)
```

6. `.format` Specifiers

The value of `.format()` becomes more apparent when we start using *format-specifiers* to make our output appear in a visually appealing manner. This can be done by replacing `{0}` and `{1}` and `{2}` with things like

`{0:6}` or `{1:^20}` or `{2:.4f}`

Each of the parts after a colon specifies something about exactly how the entry prints out. This is best shown by example, so see below. (This is just a sample of the options you have; these are the only ones I will test you on, though.)

In [ ]:
```
# EXAMPLE 6a: Let's play with some format specifiers.

name = "Evan"
number = 1/6

print("{0:10}'s number is {1}".format(name, number))
# Here's what the ":10" part does. It specifies that the first fill-in should print
10 characters in total.
# Since "Evan" is only 4 characters long, it fills in the remaining 6 spaces with b
lanks.

print("{0:^15}'s number is {1}".format(name, number))
# The ":^15" works the same way, except instead of making "Evan" the LEFT 4 charact
ers, it makes it the MIDDLE
# 4 chracters. (The text may not be perfectly centered.)

print("{0}'s number is {1:.5f}".format(name, number))
# The ":.5f" will only work for floating point numbers.  The ".5" means "5 digits",
the "f" means "after the
# decimal" ('f' stands for 'float').
```

Notice that the first two printed line have inappropriately rounded last digits, because of `float` inprecision at the lowest places. However, the last one, where we have demanded that the display is rounded to 5 places after the decimal, comes out rounded correctly!

Let's fix this code so that the output looks proper, like (for example, with 6 people)

`Each person pays: $16.92`

where the price is displayed with no space after the dollar sign, and rounded to two decimal places.

In [ ]:
```
# EXAMPLE 6b: Split the bill

n = int(input("Number of people: "))
bill = 101.53

print("Each person pays: $", bill/n)
```

In [ ]:
```
# EXAMPLE 6b': Split the bill

n = int(input("Number of people: "))
bill = 101.53

print("Each person pays: ${0:.2f}".format(bill/n))
```

Clearly, specifiers like `:.5f` are useful for when you want to display a certain number of significant digits. The other specifiers we mentioned can be really useful if you want Python to print pretty tables -- if you have columns, they can help you make sure that they line up properly.

In [ ]:
```
# EXAMPLE 6c: Some nicely formatted columns.

name1 = "Joe"
name2 = "Katherine"
name3 = "Larry"
score1 = 24.15818
score2 = 23.616
score3 = 17

# Since all three lines are given the same format specifiers, they will print tight
ly aligned columns, even though the data
# has different lengths.
print("Look at pretty columns!!!!")
print("{0:12}| Score = {1:.3f}".format(name1, score1))
print("{0:12}| Score = {1:.3f}".format(name2, score2))
print("{0:12}| Score = {1:.3f}".format(name3, score3))
```