Welcome to lecture 3, everyone! Are there any questions? How did the first lab go?!
Let's start with our usual announcements and then we will proceed with Python. As you can see we have a busy day ahead :)

Lecture 3: Turtle and for loops

▸ Administrative
▸ print function
▸ Multiline strings and docstrings
▸ Turtle module
▸ For loops
This week

- All course handouts can be found on the course website
  - https://cs.pomona.edu/classes/cs51a/
- First assignment due this coming Sunday.
- If you have any questions, join our office hours and mentor sessions.
  - Schedule is posted on website.
  - Mentor sessions Zoom links on Slack.

I sent you an announcement on Monday to remind you that all course material, including slides, assignments, and labs, are posted on the course website https://cs.pomona.edu/classes/cs51a/ Please make sure to check it regularly. The first assignment (the handout includes the ethics reading) is due this coming Sunday. If you have any questions, please join our office hours and mentor sessions. The schedule is on the course website and the TAs have posted their Zoom links on Slack.
Lecture 3: Turtle and for loops

- Administrative
- `print` function
- Multiline strings and docstrings
- Turtle module
- For loops

If there are no questions about logistics, let's jump into some exciting new Python code.
PRINT FUNCTION

print function

- Use it when you want to “print” (i.e. display on the screen) certain expressions (e.g., numbers, strings, contents of variables, messages, etc.).

- Extremely useful for figuring out how our code works.

Let's start with seeing the print function, one of the most versatile built-in Python functions. Its name can be confusing, it doesn’t actually use a printer. Instead, we use we want to display on the screen certain expressions. For example, we might want to print numbers, strings, contents of variables, messages, and so on. We will use the print function a lot to figure out how our code works.

Let's look at bbq_cost function in the bbq-functions.py source code.
Using the print function to understand our code

```python
def bbq_cost(angle, jasmine, num_people):
    soda_cost = 9.5
    hotdog_cost = 8.75
    num_hotdogs = hotdogs(angle, jasmine)
    num_sodas = soda(num_people)
    print("hotdogs: " + str(num_hotdogs))
    print("sodas: " + str(num_sodas))
    return num_sodas * soda_cost + num_hotdogs * hotdog_cost
```

If you wanted to figure out why it was that high, you could temporarily add some print statements in the code.

Let's assume you call the function `bbq_cost(1,2,6)` which will calculate that our bbq will cost us $15.75. If you wanted to figure out why it was that high, you could temporarily add some print statements in the code. For example, you could add

```python
print("hotdogs: " + str(num_hotdogs))
print("sodas: " + str(num_sodas))
```

If you were to call again `bbq_cost(1,2,6)` you would see that the total cost was a consequence of needing to buy 13 hotdogs and 12 sodas.
Don’t forget to remove unnecessary print statements

▸ We can dig further if we’d like by adding more print statements.
  ▸ E.g., `print("cost of sodas" + str(num_sodas * soda_cost))`
  ▸ When you’re done, don’t forget to REMOVE ALL PRINT STATEMENTS!

▸ In most cases, we’re adding print statements to help us debug our program.
  ▸ debugging: the process of finding and removing programming errors.

We can dig further if we’d like by adding more print statements. E.g., `print("cost of sodas" + str(num_sodas * soda_cost))`. When you’re done, don’t forget to REMOVE ALL PRINT STATEMENTS! In most cases, we’re adding print statements to help us debug our program. debugging is the process of finding and removing programming errors.
PRINT FUNCTION

print vs return

- print
  - the print function displays the value to the screen/shell.

- return
  - a return statement has two parts, return [expression]
  - When the program gets to this line, it evaluates the expression.
  - Whatever value this expression evaluates to then is "returned" from that function and represents the value at where the function was called.

You might be wondering what is the difference about print and return statements. As we just discussed, the print function displays the value to the screen/shell. If we examine the syntax for the return statement, we will see that it has two parts: the keyword return and then some expression. When the program gets to this line, it evaluates the expression which will result to some value. This value will be then returned to whomever called the function.
print vs return.py

- Similar calculations but VERY different behavior.

Let’s take a look at the print_vs_return.py source code. Both print_square and return_square functions do similar things, but their behavior is VERY different. What would be printed out if the following was typed into the Python shell after defining these functions?

```
print_square(10)
return_square(10)
x = print_square(10)
x
y = return_square(10)
y
```

The results would be
100
100
100
100
Undefined
Undefined
100
print_vs_return.py

- `print_square(10)` and `return_square(10)` appear to do the same thing, but they are different.
  - `print_square(10)` is actually printing to the shell inside the function.
  - `return_square(10)` evaluates to 100, then that value is printed because the default behavior for the shell is to print the value.
- This difference is highlighted in the next 4 statements:
  - `x = print_square(10)` calls `print_square(10)` which prints but does NOT return a value. Therefore, `x` remains undefined.
  - `y = return_square(10)` calls `return_square(10)` which does NOT print out the value (100) but returns it, therefore `y` is assigned the value 100.

Let's break it down: `print_square(10)` and `return_square(10)` appear to do the same thing, but they are different.
`print_square(10)` is actually printing to the shell inside the function.
`return_square(10)` evaluates to 100, then that value is printed because the default behavior for the shell is to print the value.
This difference is highlighted in the next 4 statements:
`x = print_square(10)` calls `print_square(10)` which prints but does NOT return a value. Therefore, `x` remains undefined.
`y = return_square(10)` calls `return_square(10)` which does NOT print out the value (100) but returns it, therefore `y` is assigned the value 100.
If you hit Run (green triangle), you get:

```
25
#
##
25
None
###
25
####
```

Let's assume we comment out the last lines of the `print_vs_return.py` file and we hit the run button (green triangle), instead of “Run file in Python Console”. What we will get is

```
25
#
##
25
None
###
25
####
```
print_vs_return.py

- When you run a file, it starts at the top and executes each statement/line one at a time.
- `print_square(5)` prints 25.
- `print("#")` prints #
- `return_square(5)` does NOTHING. It returns a value, but then we don't do anything with it (just as if we'd typed 5*5 there) so the result of the calculation is lost.
- `print("##")` prints ##
- `print(print_square(5))` calls `print_square(5)` which again prints 25. Then, when we return, we try and print out the value that was returned from `print_square(5)`. Since `print_square` does not return a value, we get “None”.
- `print("###")` prints ###
- `print(return_square(5))` prints 25 because `return_square(5)` returned it!
- `print("####")` prints ###
When the interpreter reaches a return statement the program indicates a disruption in flow.

We have to leave that function.

Therefore any code in a function body that directly follows a return statement cannot be reached.
Let's switch gears now and start looking into some special versions of strings
Multiline strings

- So far we've seen double quotes and single quotes to enclose strings.
- If we want a string to span over multiple lines we have a few options
  - there is a special character '\n' that represents the end of the line. E.g.,

```python
print("This is a string
that spans over multiple
lines")
```

So far we've seen double quotes and single quotes to enclose strings. If we want a string to span over multiple lines we have a few options there is a special character '\n' that represents the end of the line. E.g., looking at the multiline_strings.py file we can see that the print statement `print("This is a string\nthat spans over multiple\nlines")` results to `print("This is a string\nthat spans over multiple\nlines")`
Multiline strings using triple quotes

- Previous approach has a few drawbacks:
  - hard to read as a human
  - hard to get formatting/alignment right
  - if it's a long string (e.g., a paragraph) it's going to go off the screen
  - pain to copy and paste multiline text from somewhere else
- Use triple quotes instead, e.g.,

```python
print('''This is a multiline string
I can continue to type
over many different lines
and it won't stop until
I close the strings''')
```

The previous approach has a few drawbacks: it's hard to read as a human, it's hard to get formatting and alignment right, if it's a long string (e.g., a paragraph) it's going to go off the screen, and in general it's a pain to copy and paste multiline text from somewhere else.

Luckily, Python has a different solution where we can use triple quotes instead.

If we look at the multiline_strings.py, we have a print statement with a long string that spans multiple lines and is enclosed within triple quotes.
Docstrings

- Docstring: a string immediately following a definition.
- Another form of commenting.

Let's take a look at the bbq-functions-commented.py file. Right after we define our function, you will notice a multiline string. A string that immediately follows a definition is recognized by the Python interpreter as a docstring, or documentation string. You can think of docstrings as another form of commenting that allows you and other developers to quickly understand the purpose of your functions and what parameters it expects.
Using the `help` function to read docstrings

- If you pass a method as an argument to the `help` function, you will get back the docstring of that method. E.g.,

```python
help(hotdogs)
Help on function hotdogs in module __main__:
hotdogs(angie, jasmine)
    Returns the number of hotdogs required for the party.

Parameters:
    angie -- the number of hotdogs angie will eat
    jasmine -- the number of hotdogs jasmine will eat
```

- This can be VERY useful when you're using code that you haven't written!

Another cool thing about docstrings is that we can pass a method as an argument to the built-in `help` method, and we will get back the docstring of that method. E.g., `help(hotdogs) will return`

```
Help on function hotdogs in module __main__:
hotdogs(angie, jasmine)
    Returns the number of hotdogs required for the party.

Parameters:
    angie -- the number of hotdogs angie will eat
    jasmine -- the number of hotdogs jasmine will eat
```

This can be VERY useful when you're using code that you haven't written!
Conventions

- We’re going to be defining docstrings for ALL functions we write from here on out.
- We’ll always use triple quotes for docstrings (even if they're just one line).
- For simple functions, a one line docstring is sufficient.
- For longer ones, first give a description of what it does, then describe what each of the parameters represents.

Let's agree on the following conventions.
- We’re going to be defining docstrings for ALL functions we write from here on out
- We’ll always use triple quotes for docstrings (even if they're just one line)
- For simple functions, a one line docstring is sufficient
- For longer ones, first give a description of what it does, then describe what each of the parameters represents.
Good style

- Use good variable/function names.
- Use whitespace (both vertical and horizontal) to make code more readable.
- Comment code, including both comments and docstrings.
- Try and write code as simply as possible (more on this as we go).

In general, every time you write code for this class I want you to follow the following good style guides:
Use good variable/function names,
Use whitespace (both vertical and horizontal) to make code more readable,
Comment code, including both comments and docstrings,
Try and write code as simply as possible (more on this as we go).
Lecture 3: Turtle and for loops

- Administrative
- print function
- Multiline strings and docstrings
- Turtle module
- For loops

We’ll now get in a very fun and creative part of working with Python.
To get there, we will need to first talk about modules. A module is a collection of functions and variables. Modules allow us house code that other people have written. For example, there is a module called math that has many of the math functions you might want. If you look up the official documentation of Python, you will discover that the math module offers you access to all sorts of functions, such as logarithmic ones, square root, trigonometric functions, and constants such as PI, e, etc.
If we want to use a module, we need to tell the program to include it with our program. To do this, we need to "import" it.

There are many ways of importing modules (some better than others).

For now, we’re going to import the functions and variables into our program as if they were local (i.e. just as if we’d written them in our program).

this is convenient for now, but in some situations there are better ways of doing it (more on this later)

```python
from math import *
```

This statement has multiple components:

- `from` is a keyword,
- `math` is the name of the module,
- `import` loads the module into our program,
- `*` means everything, i.e. load everything included in the math module.

Let’s see how we would load the math module into our program. We would type from math import *.

This statement has multiple components:

- `from` is a keyword,
- `math` is the name of the module,
- `import` loads the module into our program,
- `*` means everything, i.e. load everything included in the math module.
The turtle module implements a set of commands similar to the Logo programming language. The basic idea is that you control the movements of a turtle (in our case, it will be an arrow) through basic commands such as:

- `forward(distance)`: Move the turtle forward by the specified distance, in the direction the turtle is headed.
- `backward(distance)`: Move the turtle backward by distance, opposite to the direction the turtle is headed. Do not change the turtle's heading.
- `right(angle)`: Turn turtle right by angle units.
- `left(angle)`: Turn turtle left by angle units.

...and many others

As the turtle moves, it draws a line behind it, so by giving it different commands, we can draw things on the screen!

Check the documentation for the turtle class online.

You'll be getting more comfortable with this documentation as part of next week's lab.
turtle module

- The turtle module implements a set of commands similar to the Logo programming language.
- The basic idea is that you control the movements of a turtle (in our case, it will be an arrow) through basic commands such as:
  - `forward(distance)`: Move the turtle forward by the specified distance, in the direction the turtle is headed.
  - `backward(distance)`: Move the turtle backward by distance, opposite to the direction the turtle is headed. Do not change the turtle's heading.
  - `right(angle)`: Turn turtle right by angle units.
  - `left(angle)`: Turn turtle left by angle units.
  - ...and many others
- As the turtle moves, it draws a line behind it, so by giving it different commands, we can draw things on the screen!
- Check the [documentation](#) for the turtle class online.
- You’ll be getting more comfortable with this documentation as part of next week's lab.

Time to have fun with a cool module called turtle... The turtle module implements a set of commands similar to the Logo programming language.
The basic idea is that you control the movements of a turtle (in our case, it will be an arrow) through basic commands such as:
forward
backward
right
left
...and many others
As the turtle moves, it draws a line behind it, so by giving it different commands, we can draw things on the screen!
Check the documentation for the turtle class online.
You’ll be getting more comfortable with this documentation as part of next week’s lab.
Let’s move our turtle!

- How would you create a square?
- forward(some_length)
  right(90)
- forward(some_length)
  right(90)
- forward(some_length)
  right(90)
- forward(some_length)

Let’s say we want to create a square of a certain length. How we would do it?
forward(some_length)
  right(90)
forward(some_length)
  right(90)
forward(some_length)
  right(90)
forward(some_length)
Let’s move our turtle!

▶ This seems like a lot of repetitive typing. Let’s say we can tell the turtle to repeat some statements, would there be a better way of creating a square?

▶ go forward some length and then turn right, repeat this 4 times

Let’s put our code in a function to make it more reusable. If we run this code, we can call square, e.g., as square(100) and see the following turtle trail. This seems like a lot of repetitive typing. Let’s say we can tell the turtle to repeat some statements, would there be a better way of saying in English that we want to create a square? go forward some length and then turn right, repeat this 4 times
Lecture 3: Turtle and for loops

- Administrative
- print function
- Multiline strings and docstrings
- Turtle module
- For loops

Lucky for us, Python has a very nice way of repeating statements without having to write a ton of code.
Python for loops

- Python has a number of different "loop" structures that allow us to do repetition (computers are really good at doing repetitive tasks!)
- The for loop is one way of doing this
- There are a number of ways we can use the for loop, but for now the basic structure we'll use is:

```python
for some_variable in range(num_iterations):
    statement1
    statement2
    ...
```

Python has a number of different "loop" structures that allow us to do repetition (computers are really good at doing repetitive tasks!)
The for loop is one way of doing this
There are a number of ways we can use the for loop, but for now the basic structure we'll use is:
for some_variable in range(num_iterations):
    statement1
    statement2
    ...

Python for loops syntaxes

```python
for some_variable in range(num_iterations):
    statement1
    statement2
...
```

- `for` is a keyword
- `in` is a keyword
- `range` is a function that we'll use to tell Python how many repetitions we want
- `num_iterations` is the number of iterations that we want the loop to do
- `some_variable` is a local variable whose scope (where it can be referred to) is only within the `for` loop
- `some_variable` will take on the values from 0 to `num_iterations-1` as each iteration of the loop occurs
  - We're computer scientists so we start counting at zero :)
- for example, in the first iteration, it will be 0, the second time 1, the third time 2, etc. we're computer scientists so we start counting at zero :)
- Don't forget the `:` at the end!
- Like with defining functions, Python uses indenting to tell which statements belong in the `for` loop
What would this code do?

```python
for i in range(10):
    print(i)
```

What if I type
```python
for i in range(10):
    print(i)
```

This code will print the numbers 0 to 9
FOR LOOPS

An iterative square

```python
def iterative_square(length):
    for i in range(4):
        forward(length)
        right(90)
```

Take a look at the `iterative_square` method in the `turtle_examples.py` file.
ASSIGNED READINGS AND PRACTICE PROBLEMS

Resources

- Textbook: Continue reading Chapter 4.
- print_vs_return.txt
- multiline_strings.txt
- bbq-functions-commented.txt
- turtle-examples.txt

Practice Problems

- Practice 1 (solution)

Homework

- (Work in progress) - Assignment 1