Special Topics: Python

Class #2

Jotto

NYU

- Simple logic word game
- Player 1 chooses a secret 5-letter word
- Player 2 must guess only real, 5-letter words
- For each guess, player 1 reports how many letters in the guessed word are shared with the secret word
- Player 2 wants to guess the secret word in as few guesses as possible
- Relies on good combinatorial logic

Sample lucky Jotto game

Guesses	Answers	Notes
1. slate	3	
2. blade	2	
3. brand	0	Has (L,E) + 1 of (S,T)
4. meats	3	Has 2 of (M,T,S)...
5. miles	5	but not both S & T
6. smile	**WIN!**	so must have (M,L,E)

Our project

NYU

- Build our own version of Jotto!
- Human and Computer player options
- Record guesses
- Record game history
- Variant rules
- Scoring
- Networking?

Let's see how cool we can make this game in 6 labs!

Objects

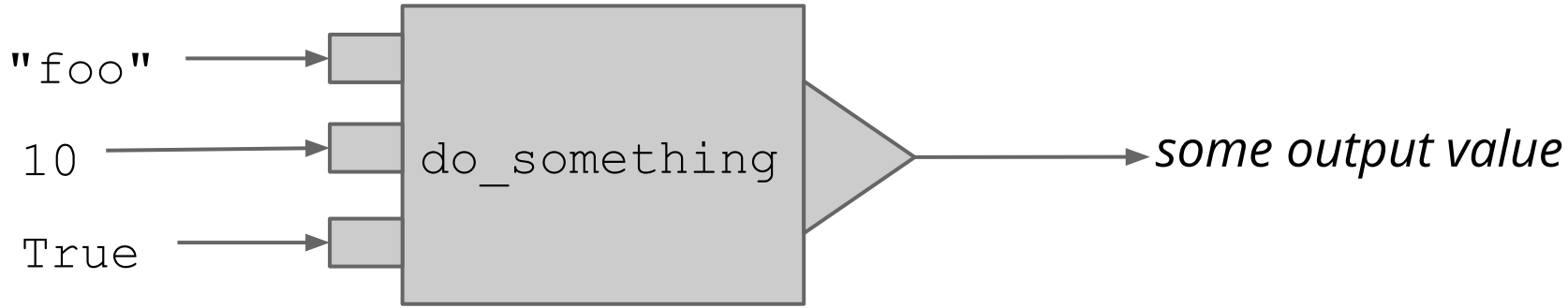
NYU

- Every value in Python is an Object!
- Every value has a Type
- Every value can be passed into a function, or have functions called on it

Objects & Functions

Think of a function like a little machine:

do_something("foo", 10, True)



Objects & Functions

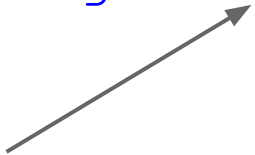
- We know something is being "called" as a function because of the parentheses
- Functions can also be "called on" objects
  - Sometimes, just for organization. One object may contain a bunch of similarly themed functions
  - Also often done to modify the state of an object
    - ➢ We'll see this when we get to objects whose state can be modified

Methods: Functions on Objects

NYU

- To call a function on an object, we use this syntax:

```
some_object.some_function(some_parameter)
```



The dot implies that `some_object` is the machine, and it is doing `some_function` on `some_parameter`.

A function called on an object is called a **method.**

Methods

NYU

Let's look at some methods on Types that we've already played with. Try out the following:

- `'hello'.capitalize()`
- `'hello'.upper()`
- `'hElLo'.swapcase()`
- `'hello'.replace('l', 'z')`
- `'  hello  '.strip()`
- `'123'.isdigit()`

Methods

NYU

What will the following resolve to?

- `'o'.swapcase() + 'k'.swapcase()`
  - `'O' + 'K'`
  - `'OK'`

- `int('    123    '.strip()) + 10`
  - `int('123') + 10`
  - `123 + 10`
  - `133`

Method Chaining

What will the following resolve to?

- `'heLLo'.capitalize().swapcase()`
  - `'HeLLo'.swapcase()`
  - `'hEllO'`

This is called "**method chaining**".

The output of the first method becomes the object for the second, and so on.

Lists

- List: Exactly what it sounds like.
  - A linear, list of values contained in one object.
  - A list can contain different Types of values
- Lists in Python are super powerful & easy to use
- "List" is another Type in Python
- Lists are **mutable**, so we can change their state by calling functions on them

Lists

NYU

A list is designated by square brackets: `[ ]`

```
x = [] # x is an empty list
```

List values are separated by commas

```
x = [1, 2, 3] # x is a list with 3 elements
```

Python's lists can mix types freely

```
x = [1, 'banana', True]
```

Lists

NYU

We can use the built-in `len` function to see the "length" of a list (i.e. how many elements it has)

```
len([1,2,3,4,5])   # returns 5
len([])   # returns 0
```

We can use the + operator to combine lists

```
[1,2,3]+[4,5,6] # resolves to [1,2,3,4,5,6]
```

Accessing Elements from Lists

NYU

How do we get an element out of a list? Try the following:

```
x = ['mon', 'tues', 'wed', 'thurs', 'fri']
x[0] # 'mon'
x[1] # 'tues'
x[2] # 'wed'
```

When reading `x[1]`, you would say "x sub 1", because the brackets denote a "subscript"

Accessing Elements from Lists

NYU

- Within the "subscript", you provide an "index".
- The index simply points to an element in the list

```
x = ['first', 'second', 'third', 'fourth']
```

```
x[0]    x[1]    x[2]    x[3]
```

Accessing Elements from Lists

- Lists are 0-indexed, meaning the first element is at index #0
- Therefore, the last element is always one less than the length
- Accessing an element beyond the length of the list is an "IndexError"

```
x = [1,2,3,4]

x[len(x) - 1] # always grabs the last elem
```

Strings and Lists

NYU

- Strings are a special type of List
- Most of the tricks we do with Lists work with Strings

```
len('hello') == len(['h','e','l','l','o'])


'hello'[1]  # what letter does this grab?
```

Strings and Lists

Likewise, the `in` operator that we learned for Strings works just as well for Lists (and it's super useful)

```
'foo' in 'foobar'   # True

'foo' in ['bar', 'foo', 'blah']   # True
```

Functions on Lists

Lists have some powerful functions. Let's examine a few:

- `append` adds the parameter to the end of the list
  - This is called an **in-place** function. That means:
    - ➢ This function mutates the list
    - ➢ This function does **not** return a value (it returns `None`)
    - ➢ This function is only useful on a variable

```
x = [1,2,3]

x.append(4)

print(x)
```

Functions on Lists

NYU

- `sort` rearranges the elements of the list to be in ascending order
  - This is an **in-place function**

```
x = [4,2,3,6,4,5]

x.sort()

print(x)
```

Functions on Lists

NYU

- `count` returns the number of times the parameter appears in the list

```
[1,2,3,2,2,1,2,3,4,2].count(2)
```

- `reverse` does exactly what you'd expect.
  - Also an **in-place function**

```
x = ['first', 'second', third']
x.reverse()
```

Functions on Lists

NYU

- `index` takes a value and returns the **first** index that value appears at in the list.

```
x = ['foo', 'bar', 'cat', 'bar', 'doo']
x.index('foo')  # returns 0
x.index('bar')  # returns 1
x.index('zap')  # ValueError, not in list
```

Functions on Lists

- There are still more functions we didn't cover
  - https://docs.python.org/3/tutorial/datastructures.html#more-on-lists
- Python has tons of power provided by its functions and methods. For example, check out how many methods Strings have:
  - https://docs.python.org/3.4/library/stdtypes.html#string-methods

Powerful Indexing

NYU

- Indexing can be used to mutate as well as access

```
x = [200,400,600]
x[1] = 5  # x is now [200, 5, 600]
```

- Referencing an index past the end of the list is always an error

```
x[3] = 5  # x has 3 elements (index 0,1,2)
          # so this causes an IndexError
x.append(5)  # This is the correct way to
             # add a new value to the end
```

Powerful Indexing

NYU

- We can also index a range of values

```
x = [2,4,6,8,10,12,14]

y = x[2:5]
```

The first value is the first index to grab from, **inclusive.**

The second value is the last index in the range, **exclusive**

```
print(y)   # So what will this print?
           # Answer: [6, 8, 10]
```

Powerful Indexing

NYU

- You can omit the first or second value:

```
x = [0,1,2,3,4,5,6,7]
x[2:]   # Grabs from 2nd index to the end
        # [2,3,4,5,6,7]
x[:4]   # Grabs from start to the 4th index
        # [0,1,2,3]
x[:]    # Grabs from the start to the end!
        # [0,1,2,3,4,5,6,7]
```

Powerful Indexing

NYU

You can even provide negative values!

```
x = [0,1,2,3,4,5,6,7]

x[-2]   # Grabs the second-to-last elem (6)

x[-3:] # Grabs last three elements [5,6,7]

x[-3:-1]   # Grabs [5,6] -- guess why?
```

- Remember, indexing without a colon gets a single value
- Indexing with a colon gets a list of values
- All these indexing tricks can also be done with strings!

Loops

NYU

- Loops and Lists go hand-in-hand.
- In Python there are two main types of loops:
  - **While** some condition is true
  - **For Each** element in a list
- Both will repeat the code in the block that follows a certain number of times before continuing to the next statement

While Loops

NYU

- While loops work just like conditionals, except they keep repeating the block over and over until the conditional is no longer true

```
x = 0
while x < 100:
    x = x + 10
    print(x)
print('Out of the loop')
```

For-Each Loops

NYU

- For each loops will *iterate* through any "iterable" object (i.e. a List) one value at a time
  - It stores the accessed value in a specified variable so you can do something with it

```
numbers = [1,2,3,4,5]
for num in numbers:
    print(num * 2)
```

Iterables

- So far we've seen two types of Iterables:
  - Lists
  - Strings

```
for letter in 'Hello, World!':
    print(letter)
```

The range Function

NYU

- `range` is a built-in function that takes 1, 2, or 3 args

```
range(10)   # Returns [0,1,2,3,4,5,6,7,8,9]

range(5,12) # Returns [5,6,7,8,9,10,11]

range(1,20,3) # Returns [1,4,7,10,13,16,19]
```

This function plays very nicely with for-each loops...

Looping over ranges


```
x = ['cat', 'dog', 'cow', 'rat', 'pig']
for index in range(len(x)):
    print('Index: ' + str(index))
    print('Element: ' + x[index])
```

This is a common pattern in Python.

It's equivalent to the basic for-loop in languages like Java

Example...


```
first_list = [1,4,10,12,16]
second_list = [2,12,0,4,13]
for first_list_elem in first_list:
  if first_list_elem in second_list:
    print('Found a shared value: ')
    print(str(first_list_elem))
```

Defining Functions

NYU

- What if we need to do that list comparison more than once throughout your program?
- Define a function!

```
def my_function(x, y):
  print('x is ' + str(x))
  print('y is ' + str(y))


my_function(10, 'fish')
```

Defining Functions

Advantages to defining functions:

- If you need to change the definition, or fix a bug, you only have to do it in one place
- *Much* easier to read, especially with well-named functions

When defining functions…

- Try to keep each function to one, easily-described, logical **task**

Functions & return

- How do we specify the output of our function?
- The `return` keyword returns a value as the output of a function!
- Important: as soon as a function hits a `return` statement, it is done!
  - i.e. It will not execute any more code in the function definition

Function Flow

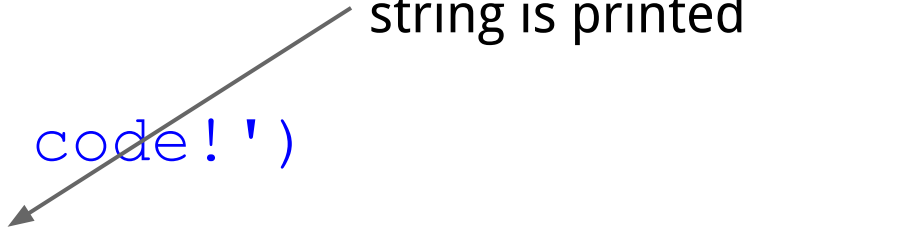
NYU

```
def print_greeting():
    print('Hello')
    return 5
    print('Unreachable code!')
print('Before call')
x = print_greeting()
print('Call returned ' + str(x))
```

Defining a function does not execute the code - it stores the instructions for when it is called

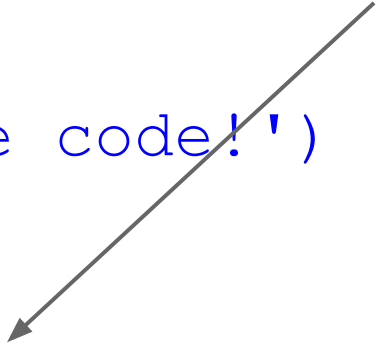
Function Flow

```
def print_greeting():
    print('Hello')
    return 5
    print('Unreachable code!')
print('Before call')
x = print_greeting()
print('Call returned ' + str(x))
```



Function Flow

```
def print_greeting():
    print('Hello')
    return 5
    print('Unreachable code!')
print('Before call')
x = print_greeting()
print('Call returned ' + str(x))
```



Next, we hit the function call, so we jump into the function's body

Function Flow

NYU

```
def print_greeting():
    print('Hello')
    return 5
    print('Unreachable code!')
print('Before call')
x = print_greeting()
print('Call returned ' + str(x))
```

We perform the first instruction in the function body, printing the string "Hello"

Function Flow

NYU

```
def print_greeting():
    print('Hello')
    return 5
    print('Unreachable code!')
print('Before call')
x = print_greeting()
print('Call returned ' + str(x))
```

Next we hit our return statement. The value 5 will be passed as output. We leave the function body and resume the flow we were in.

Function Flow

NYU

```
def print_greeting():
    print('Hello')
    return 5
    print('Unreachable code!')
print('Before call')
x = print_greeting()
print('Call returned ' + str(x))
```

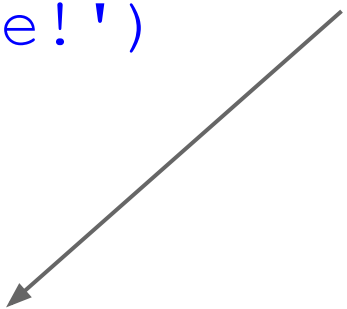
This is "dead code" - code that can **never** actually execute. It should be deleted.

x now stores the outputted value, 5

Function Flow

```
def print_greeting():
    print('Hello')
    return 5
    print('Unreachable code!')
print('Before call')
x = print_greeting()
print('Call returned ' + str(x))
```

Finally, we print the output that was stored in the variable x



Defining Functions

Let's define a function that takes in two lists.

It returns `True` if they share at least one element in common, and `False` otherwise.

lists_share_elem

NYU

```
def lists_share_elem(x, y):
    for elem in x:
        if elem in y:
            return True
    return False
```

Here we take advantage of the fact that return stops a method's execution.

Returning from Infinite Loops

One acceptable place to use an infinite loop is in a method where the quit condition is a return:

```
num_tries = 0
while True:
    success = attempt_connect()
    num_tries += 1
    if success:
      return num_tries
```

This syntax is called "augmented assignment". The statement is equivalent to:

```
num_tries = num_tries + 1
```

Returning None

NYU

- You may also use the `return` keyword without a value following it.
- You might do this if:
  - the function only prints something
  - the function is **in-place** and modifies its inputs
- This will simply return `None` and implies the function has no meaningful return value.

Recursion

NYU

- Recursion is when a function calls itself
  - (Usually on a reduced version of the problem)
- Many problems that can be solved "iteratively" (with a loop) can also be solved recursively
- Certain problems lend themselves better to one solution or the other

Fibonacci

NYU

- The Fibonacci sequence is an infinite series of numbers
- Begins with 1, 1
- Each following number is the sum of the last two

Defined in mathematics:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Fibonacci

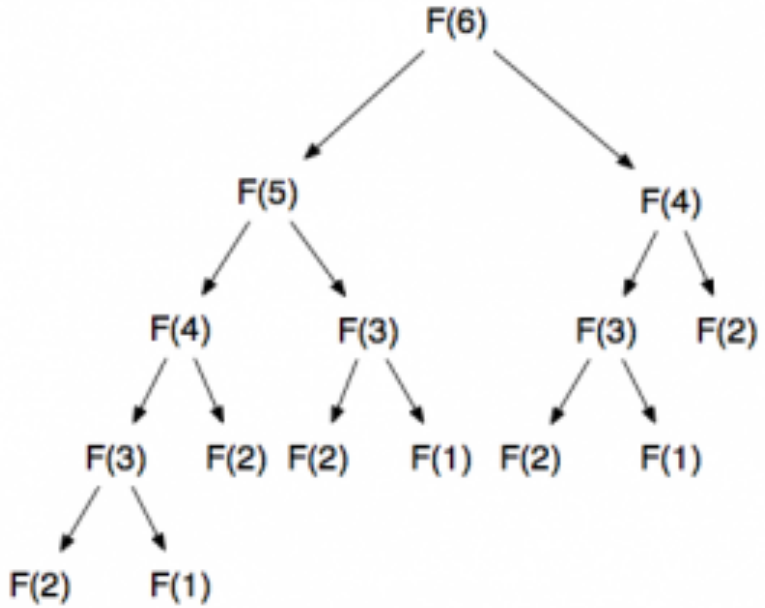
The mathematical definition of the sequence is easily translated to a recursive function

```
def get_fib_num(n):
  if n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    return get_fib_num(n-1) + get_fib_num(n-2)
```

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Fibonacci

- As a side note, the solution on the last slide is *horribly inefficient*.
- We'll discuss more later, but look at the calls performed:



Style

NYU

- Code follows certain style guidelines to keep consistency and readability (important for maintainability)
- Certain style rules are accepted by the Python language standard, and others are more subjective
- We will follow a few of the style rules specified in the Google Python Style Guide

https://google-styleguide.googlecode.com/svn/trunk/pyguide.html

Style Rules - Line Length

NYU

Rule 1: Max line length of 100 characters

- Lines can safely be split on commas in a list:

```
foo_bar(self, width, height, color='black', design=None, x='foo',
        emphasis=None, highlight=0)
```

- Wrap an expression in parentheses to make line breaks legal where they otherwise wouldn't be:

```
if (width == 0 and height == 0 and
    color == 'red' and emphasis == 'strong'):
```

- When adding a break, indent the wrapped line to align appropriately

Style Rules - Whitespace

NYU

- Horizontal Whitespace:
  - Always use 4 spaces for indenting blocks.
  - Never use tabs
- Vertical Whitespace:
  - 2 new-lines between each top-level definition
    - ➢ A top-level definition is a function or class defined *not* inside of another block
  - 1 new-line between each method definition
    - ➢ We haven't defined our own classes and methods yet…
  - Use new-lines in blocks for readability when necessary

Style Rules - Whitespace

NYU

- The style guide has a lot of additional rules around horizontal whitespace
  - https://google-styleguide.googlecode.com/svn/trunk/pyguide.html?showone=Whitespace#Whitespace
- I'm not going to worry about those little details for our purposes, but they're nice to follow.
- Most of them will be done automatically by PyCharm by going to Code → Reformat Code
  - Or, pressing Ctrl+Alt+L with the default key bindings

Style Rules - Naming

NYU

- Names of methods, functions, and variables should all be lower-case and separated by underscores

```
function_name(param_name)
```

- Avoid single-letter names, except in iterators

```
x = input(prompt)   # x is not descriptive
for i in range(10):   # This is fine
```

Naming and Privacy

NYU

- Python has no enforced privacy rules
  - Any value or function an object possesses can be accessed
- However, privacy is *suggested* through naming conventions common throughout Python
  - Adding a single underscore to the beginning of a name means it's **internal**
    - ➢ internal = should **not** be accessed by code outside the file
  - Adding two underscores to the beginning of a name means it's **private**
    - ➢ private = should **not** be accessed by code outside the block

Documentation

NYU

- Documentation is necessary for future users or maintainers of code to be able to understand the code
- Python has a built-in way of writing documentation in the code
- Python's documentation units are called **Docstrings**
- Unlike most languages where documentation is written in comments, Python uses a String that is formatted in a specific way.

Docstrings

NYU

- A Docstring **must**:
  - Be a multiline string using double-quotes
  - Be the first line of the method's block
  - Be formatted in a specific manner

- Docstrings are either:
  - A "one-line summary" for very simple or obvious functions
  - A "multiline docstring", which begins with a one-line summary

One-line Docstring

NYU

```
def isThreeDigitNumber(num):
    """Determine whether the provided number is three or more
       digits, not including decimal places."""
    return num >= 100 or num <= -100
```

Your one-line documentation should:

- Fit on a single, 100-character line
- Describe the general purpose of the function
- End with punctuation
- Have the quotes begin and end on the same line as the text.

Official policy: http://legacy.python.org/dev/peps/pep-0257/#one-line-docstrings

Multi-line Docstrings


```
def calcDigitOfPi(digit, timeout, useSpigot):
    """Calculate the specified digit in Pi.

    Calculates a particular digit of Pi, or simply
    retrieves it from a cache if calculated
    previously.

    Args:
        digit: The integer digit to retrieve
        timeout: Max time in milliseconds before giving up
        useSpigot: True to use the Spigot algorithm
    Returns:
        The numeric digit, or -1 if the function timed out.
    """
```

Multi-line Docstrings

- Official Policy:
  - http://legacy.python.org/dev/peps/pep-0257/#multi-line-docstrings
  - This is a very terse read… I don't recommend it
- Instead, refer to the Google Python Style Guide:
  - https://google-styleguide.googlecode.com/svn/trunk/pyguide.html?showone=Comments#Comments
  - Much easier to read

Viewing Docstrings in PyCharm

- You can quickly view the documentation of a function in PyCharm
  - Super useful for quickly understanding how a function works
  - Also very useful for seeing how your Docstring looks
- Click on the function name (or select it if you're in an autocomplete list) and press Ctrl+Q
  - For more info, check out the official PyCharm docs: http://www.jetbrains.com/pycharm/webhelp/viewing-inline-documentation.html

Tuples

NYU

- A Tuple is much like a List, but used in very different scenarios.
- The main difference is that tuples are **immutable**
- Tuples are defined by a comma separated list *not* in brackets
- We will always surround them in parentheses, even though it's only *sometimes* required, for consistency.

Tuples (example)

NYU

```
value = ('foobar', 25)
value[0]   # foobar
value[1]   # 25
value[1] = 100 # TypeError! Immutable!
               # The tuple's data cannot be modified!
value = 'foobar', # This is a valid Tuple, because of
                  # the trailing comma, but it's easy
                  # to overlook. That's why we will
                  # always use parentheses
```

Uses of Tuples

NYU

You generally use a tuple when:

- You know exactly what structure of data you are working with
  - For example, if your method always returns a list of two elements of certain meaning in certain order, it should be a tuple
- You want to return multiple values
- You really want your data to be immutable
  - We'll discuss this in a later session

String formatters

- From here on, let's never write this again:

```
'Your value was ' + str(val)
```

- There's an easier way, using String formatters!
- `%` is a special character in a String, like a backslash
  - `%` means the next character describes a formatting param
  - The character following the `%` describes how to format it
  - Therefore, if we want to type a literal percent sign, we need to escape it with a backslash

String formatter example


```
'Your value was %s' % val
```

- The `%` is also an operator that performs the formatting
- The value after the `%` operator is substituted into the string, where the `%s` is.
- `%s` means to format the value as a string. It does the type conversion for us!

String formatters

You can have multiple formatting values in a single String:

```
'Hello %s, I am %s!' % ('Bob', 200)
```

- Here, it expects a tuple to be provided.
- The value on the right of the `%` must provide **exactly** as many values as `%`'s in the string.
  - If `%s` appears three times in the string, the `%` must be followed by a 3-tuple (a tuple with 3 elements)

String formatting

NYU

- There is an entire **lecture's worth** of things you can do with string formatting…
  - Other formatters, like `%d` and `%x`
  - Positional arguments
  - Named arguments
  - Alignment operators
  - Formatters to specify exactly how many digits in a number to show before and after the decimal point…
- I won't bore you with them all here. `%s` is all you need for now
- If you really want to know, read this:
  - https://docs.python.org/3/library/string.html#formatspec

Some closing tips...

- If you're going to populate a list inside a loop, be sure to create it before entering the loop:

```
user_inputs = []
while len(user_inputs) < 5:
    next_input = input('Say something: ')
    user_inputs.append(next_input)
```

Some closing tips...

- To test that your function works in PyCharm, add a call to it at the end of the file.

```
def doSomething(param1, param2):
    ...


doSomething(10, 'foo')
```

- Try changing the inputs to make sure it works in all cases. Remove the call before submitting, unless it's for your main function.

Some closing tips...

NYU

- The order you define your methods in generally doesn't matter. Find a system that works well for you.
- Your call to the main method should be at the very end

```
def _foo():
  …
def _bar():
  …
def main():
  …
main()
```

Some closing tips...

NYU

- For full lab credit, be sure to…
  - Have a docstring for every function you define
  - **Run your code!** Make sure it actually *works*
    - ➢ Try out different inputs. Specifically, inputs that would cause your conditionals to have different results, to test all your code paths.
    - ➢ If you're not happy with how your code runs, odds are it still needs more improvements!
- Remember, each lab is incremental, so don't leave obvious bugs for yourself later