WELCOME BACK!
Questions?
### Where We Are

<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min()/max()</td>
<td></td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>int()/float()</td>
<td></td>
<td>if/elif/else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True/False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td>for</td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Randomness

Wednesday/Thursday we'll be talking about testing/debugging. Today we're going to detour into randomness.
Python's random module generates pseudorandom numbers for us (and has other functionality).

```python
import random

print( random.randint(0,10) )
print( random.randint(0,10) )
print( random.randint(0,10) )
```

generates a random int between 0 and 10 inclusive of both (unusual in python).
okay, pseudorandomness

images from http://boallen.com/random-numbers.html
The difference

Pseudorandom numbers are sequences mathematically generated to approximate a random distribution of numbers, but they are actually deterministic based on a specific starting seed. The seed is a number or vector used for initialization of the process. Enter the same seed you get the same sequence of numbers.

Ever play a computer game where if you take a turn some stuff happens, but if you save first and then take the turn multiple times the same random events happen every time? Welcome to pseudorandom.
Seeds

Speaking of seeds

```python
import random
random.seed(1)
print(random.randint(0,10))
random.seed(1)
print(random.randint(0,10))
```

Not exactly random.

seed() initializes the seed used by the pseudorandom number generator. If called with no argument it uses the current system time.
randrange() works very similar to randint() except the end is exclusive (i.e. below a 10 cannot be generated)

```python
import random

print( random.randrange(0,10) )
print( random.randrange(0,10) )
print( random.randrange(0,10) )
```

generates a random int between 0 and 10 inclusive of 0 but not 10.

Let’s test that…
(printing is sloooooooooooww)
choice

choice selects one member of a sequence pseudorandomly

```python
import random
str1 = "abcd"
print( random.choice(str1) )
print( random.choice(str1) )
print( random.choice(str1) )
print( random.choice(str1) )
```

each object has essentially the same chance of being picked.
similar to choice but can select multiple times from the sequence returning the result as a list

```python
import random
str1 = "abcd"
print(random.sample(str1, 3))
```

each object has *essentially* the same chance of being picked.
remember range() gave us a sequence of numbers…

```python
import random

print( random.choice( range(1000000000000) ) )
```

(you could do the same thing with randint() but I believe this method is faster)
Random string generator

Remember the string constants?
string.ascii_letters
string.digits
string.punctuation

We can use these now to generate random strings if we like…
WELCOME BACK!
<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min()/max()</td>
<td></td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>int()/float()</td>
<td></td>
<td>if/elif/else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True/False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td>for</td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;random functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Error types

Remember these?

<table>
<thead>
<tr>
<th>Type</th>
<th>Problem</th>
<th>Finding</th>
<th>Fixing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Code</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Logic</td>
<td>Thinking</td>
<td>Moderate</td>
<td>Hard</td>
</tr>
<tr>
<td>Runtime</td>
<td>Variable names</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td></td>
<td>Special cases</td>
<td>Hard</td>
<td>Moderate</td>
</tr>
</tbody>
</table>
You basically have two options when programming:
- you can spend time thinking *before* coding, or
- you can spend time rethinking *after* coding (i.e. debugging)
Begin rewriting your code before finishing original concept usually leading to a muddled mess.

(Not recommended)
functions (in the mathematical sense) map inputs to outputs (or solutions).

\[ f(y) = x^2 \]
Similarly...

Computer science functions also map inputs to solutions but we often have to give more thought to carefully evaluating a variety of inputs and how they might map to unexpected values or errors.

```python
def foo(x):
    y = x ** 3
    y /= x - y
    return y
```

what values might be an issue?

Can we test exhaustively?
If we can't test exhaustively (and we can't) what can we do?

Choose a good assortment of test cases. We'll discuss this more tomorrow.
with *integrated* testing we test the entire run of the program, or at least a major section of it, potentially testing dozens or hundreds of individual function calls at one time.
with *unit* testing we isolate individual functions and test them one at a time.
Which to use?

The answer is almost always **both**.

Unit testing lets you isolate problems *within* specific functions that integrated testing might notice but have trouble pinning down ("somewhere my program went bad but I don't know if it was in function 2, 4, or 5"). Unit testing is also good for testing for edge or corner cases that may cause problems.

Integrated testing lets you find problems *between* functions (i.e. mismatches of outputs to inputs) which unit testing is oblivious to. Example: function 1 returns a string as an output which is used as the input of function 2, but function 2 was written expecting an int input. Both functions work in isolation but when tested as an integrated whole they break down.
import turtle

def line_and_turn(length, angle):
    turtle.fd(length)
    turtle.rt(angle)

def poly(sides, size):
    for i in range(sides):
        line_and_turn(size, 360/sides)

def turtHouse():
    poly(4, 100)
    poly(3, 100)

how do we test this?
WELCOME BACK!
Questions?
<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Flow Control</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>print()</td>
<td>branching</td>
<td>def</td>
</tr>
<tr>
<td>float</td>
<td>type()</td>
<td>loops</td>
<td>return</td>
</tr>
<tr>
<td>string</td>
<td>help()</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>bool</td>
<td>min()/max()</td>
<td></td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>int()/float()</td>
<td></td>
<td>if/elif/else</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td></td>
<td>True/False</td>
</tr>
<tr>
<td></td>
<td>len()</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>Input()</td>
<td></td>
<td>for</td>
</tr>
<tr>
<td></td>
<td>&lt;user defined&gt;</td>
<td></td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>&lt;turtle functions&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;random functions&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Error types

<table>
<thead>
<tr>
<th>Type</th>
<th>Problem</th>
<th>Finding</th>
<th>Fixing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Code</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Logic</td>
<td>Thinking</td>
<td>Moderate</td>
<td>Hard</td>
</tr>
<tr>
<td>Runtime</td>
<td>Variable names</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td></td>
<td>Special cases</td>
<td>Hard</td>
<td>Moderate</td>
</tr>
</tbody>
</table>
def foo(x, y, bool1):
    if bool1:
        if x > y:
            out = x - y
        elif x == y:
            out = x / y
        elif x < y:
            out = y - x
    else:
        if x > y:
            out = True
        elif x == y:
            out = y / x
        elif x < y:
            out = False
    return out
There are generally 3 types of test cases:
  • Simple Cases
  • General Cases
  • Boundary Cases
Simple cases are the first you try because you use them to test the basic functionality of the code.

Simple cases are defined by being easy to calculate the results by hand.

For example the inputs 1, 2, and 3 for a function that returns the input cubed are simple cases. If 1 gives me an output of 1, 2 gives me an output of 8, and 3 gives me an output of 27 then I'm pretty sure the function is working fine for typical positive inputs.

The test cases I've been giving you so far have been examples of simple cases for the most part.
General cases are used to test all branches and paths of the code.

It's all too easy to test three out of four branches and assume the last one is good. Don't trust *anything* you haven't tested.

Trust no one.

Woof.

Except for this fat, fluffy baby duck who thinks he's a dog.

He's legit.

http://www.sebastienmillon.com/Trust-No-One-8-5x11-Art-Print-15
Boundary Cases

Boundary cases cover the edges of input regimes. Also called corner cases or edge cases.
example edge cases:
Int 0, 1, -1
Float 0.0, 1.0, -1.0
String empty string, single character strings, strings with only numbers, strings with only symbols

It's important to remember these are typical edge cases but they may not be the edge cases for your particular application.

For example:

```python
def divide_num1_by_num2_minus_two(x, y):
    return x / (y - 2)
```
Boundary Cases

For our purposes if we tell you a given variable is a string you can trust that, in other words we aren't asking you to check that you've been given a number when a number was specified in the docstring or problem statement.

In real life you do need to check your input and often sanitize it, particularly if the input in question is coming from an outside source (web form for instance).
def foo(x, y, bool1):
    if bool1:
        if x > y:
            out = x - y
        elif x == y:
            out = x / y
        elif x < y:
            out = y - x
    else:
        if x > y:
            out = True
        elif x == y:
            out = y / x
        elif x < y:
            out = False
    return out
## A Plan of Attack

<table>
<thead>
<tr>
<th>Type</th>
<th>x</th>
<th>y</th>
<th>bool1</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>1</td>
<td>2</td>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>General</td>
<td>3</td>
<td>2</td>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>False</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>2</td>
<td>True</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>-3</td>
<td>True</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>-1</td>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>2</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>-3</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>2</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Boundary</td>
<td>0</td>
<td>0</td>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>False</td>
<td>1</td>
</tr>
</tbody>
</table>
def testHarness(x, y, bool1, expected):
    out = foo(x, y, bool1)
    if out != expected:
        print(x, y, bool1, expected, out)
        return 0
    else:
        return 1

count = 0
for i in range(-30, 31):
    count += testHarness(i, i-2, True, 2)
    count += testHarness(i-3, i, True, 3)
    count += testHarness(i, i, True, 1)
    count += testHarness(i, i-2, False, True)
    count += testHarness(i-3, i, False, False)
    count += testHarness(i, i, False, 1)
print("tested 360 cases, " + str(count) +" passed.")
def count_xs(str1):
    count = 0
    for char in str1:
        if char == "x":
            count += 1
    return count

So what test cases would you choose for this?