Announcements

- Try to get help from me and tutors
- Reading assignment for this week: Chapter 3 of Downey
- Break around 10:15am
Functions
Functions that come with Python

- Some examples:
  - `type(45)`
  - `float(45)`  `float('45.62')`
  - `int(34.56)`  `int('1432')`
  - `len('apple')`
  - `str(3421)`
  - `round(2.32)`
  - `range(1, 5)`
  - `pow(2, 3)`
  - `abs(-5)`
  - `id(3)`  # memory address
  - `help(pow)`
  - `help(id)`
  - ...
Type conversion functions

int('327')
int('hello')  # error
int(-2.9)
float(32)
float('32.4312')
str(345)
str(32.123)
math, random functions

import math
print(math.log(10))
print(math.log10(10))
print(math.log10(1e6))
radians = 0.7
height = math.sin(radians)
print(math.sqrt(3))
...

import random
print(random.random())
print(random.randint(0,100))
**string functions**

s = ‘hello all’

s.count(‘ll’)  # count, index are different kind of
s.index(‘ll’)  # functions (called methods) that require s.

s.len()  # s. would not work in this case because
len(s)  # len is a regular function using s as a parameter
Function composition

- Can compose functions as we do in math, e.g., \( f(g(x, y)) \)

```python
import math
radians = 0.7
math.radians(math.degrees(radians))

radians = 0.3
math.acos(math.cos(radians))

pow(abs(-3), round(5.6))
```
Creating new functions

• What we know in **math**: a 2 step process
  1. Define a function
     \[ f(x, y) = x \times y + 1 \]
  2. Apply/Use/Invoke/CALL the function
     \[ f(2, 3) = 2 \times 3 + 1 = 7 \]

• We do the same in **programming**: again 2 step process
  1. Define a function
     ```python
     def f(x, y):
         return x * y + 1
     ```
  2. Apply/Use/Invoke/CALL the function
     
     \[ f(2, 3) \]

     (BTW, defining a function creates a variable with the same name)
Mechanics of defining/calling a function

required elements

function name
parameters

function header

function body

indentation required

\textbf{def} \textbf{f} (x, y):
\hspace{1cm} \textbf{return} \ x \ast \ y + 1

f(2, 3)

() after a function name means a function call, in this case with two arguments
Program: flow of execution

```python
def message():
    print '1'
message1()
print '2'

def message1():
    print 'a'
message2()
print 'b'

def message2():
    print 'middle'
message()```

Output:

```
1
a
middle
b
```

Note: three functions and a call to message on the left is a *program*!
Parameters and arguments

- Function can have **zero or more** parameters
  - Function may be defined with **formal parameters**
  - Then called with **actual arguments**

- Example:

```python
def announce(msg):
    print msg

announce('Welcome to the world of Python!')
announce(45)
announce(3 + 4.5)  # how many arguments here?
announce(abs(pow(2, 3)))
```
Another example

def languages(lang1, lang2):
        result = lang1 + ' is simpler than ' + lang2
        print result

lang1 = 'Python'
lang2 = 'Java'
languages(lang1, lang2)
Frames in Python

- Each function executes in a *frame* of its own
- `announce` executes in its own frame
- `languages` executes in its own frame
- They are inside an implicit *topmost* frame named `__main__`
- So, `languages` is called by `__main__` on previous slide

- When you create a variable outside of any function, it belongs to the `__main__` frame (e.g., `lang1` and `lang2` on previous slide)
def foo(a):
    a = bar(a)
    print a

def bar(n):
    a = n * n
    a = foobar(a)
    return a

def foobar(a):
    n = a * a
    return n  # Line 1

n = 2
foo(n)

A snapshot of the stack diagram at the moment the program is about to execute Line 1. Note that __main__ calls foo, foo calls bar, bar calls foobar in that order:

<table>
<thead>
<tr>
<th>Function</th>
<th>a</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>foobar</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>bar</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>main</strong></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

n: 2
Void functions and fruitful functions

- **announce** is an example of a *void function*
  - It does not *return* any useful value when it is called; it only prints a value
- **square** is an example of a *fruitful function*
  - It *returns* a value when it is called

```python
// fruitful function
def square(n):
    return n * n
print square(3)
```

```python
// void function
def announce(msg):
    print msg
Announce('hello!')
```

See what gets printed by the print statement in each case!
You shouldn’t really expect a void function to return any useful value and
announce(‘hello’) returns *None* to indicate that.

```python
print square(3)
```
Functions with multiple parameters

- Functions may have any number of parameters
- 1-to-1 mapping between actual arguments and formal parameters

```python
def distance(x1, y1, x2, y2):
dx = x2 - x1
dy = y2 - y1
return math.sqrt(square(dx) + square(dy))

d = distance(1, 2, 4, 6)
print d
```
Caution with return

- **return** means unconditional, immediate exit out of the function even if there is another line after a **return** statement in the function.
- If there is a line after a **return** statement, make sure it is useful to be there.
- Example: **Line 1 and Line 2** in this example will never be executed.

```python
def test():
    return 12
    x = 4  # Line 1
    print x  # Line 2

print test()
```
Why functions?

- **Abstraction**: give a name to a group of statements and use it
  - Makes your program easier to read and debug

- Make a program smaller by eliminating repetitive code
  - Later if you make a change, you only have to make it in one place

- Dividing a long program into a meaningful set of functions allows you to develop and debug the parts one at a time

- **Reuse**: well-designed functions are often useful for many programs
A complete example

def f2c(fahr):
    """ (int) -> float     # docstring: These multiline comments
    (float) -> float
    Returns the Celsius equivalent of a Fahrenheit temperature.
    Ex. f2c(212) -> 100.0; f2c(32.0) -> 0.0; f2c(-40) -> -40.0
    """
    return (fahr - 32) * 5.0 / 9.0

def main():
    fahr1 = 212
    print f2c(fahr1)
    print f2c(32)
    print f2c(-40)

if __name__ == '__main__':
    main()  # These 2 lines are executed when
    # this program is run in script mode.
    # The call to main is skipped when this
    # file is imported by another module.
See f2c.py on the course web
def languages(lang1, lang2):
    result = lang1 + ' is simpler than ' + lang2
    print result

lang1 = 'Python'
lang2 = 'Java'
languages(lang1, lang2)
result  # Can’t do this

- **Function** `languages` has 3 **local** variables: `lang1`, `lang2`, and `result`.
- The special frame (the *topmost* frame) named `__main__` has 2 **global** variables named `lang1` and `lang2`. They happened to have the same names as the local variables in `languages`, but there is no connection between them.
- **Trying to access** `result` outside of the function `languages` is not allowed! The variable `result` is *not in scope*!
- A variable can be accessed only when it is *in scope*!
Examples: Scoping rules

```
x = 3

def foo():
    print x

foo()
print x

Output:
3
3
```

```
x = 3

def foo():
    x = 4
    print x

foo()
print x

Output:
4
3
```
Examples: Scoping rules (cont’d)

```python
x = 3

def foo():
    x = x + 4
    print x

foo()
print x

UnboundLocalError

Since x is being declared inside foo, trying to access it is not allowed before a value is assigned to it.
```

```python
x = 3

def foo():
    global x
    x = x + 4
    print x

foo()
print x

Output:
7
7
```
Examples: Scoping rules (cont’d)

x = 3

def f(x):
    return x + y

y = 4
print f(2)

Output:
6
Importing functions from a module

- Two ways of importing functions, using `math` as an example:
  1. `import math` *(use `math.` in front of names imported)*
  2. `from math import *` *(skip `math.` in front of names imported)*

# To import everything from math
```python
>>> import math
>>> math
>>> math.pi
>>> pi  # error!
>>> math.cos(.7)
```

# To import only pi from math
```python
>>> from math import pi
>>> pi
```

# To import everything in math
```python
>>> from math import *
>>> cos(.7)
```
Example: modules we create (optional)

- Suppose we have two modules (.py files): test.py and test1.py in the same folder.
- Suppose you want to import test1.py into test.py

```python
# test1.py
x = 45
def foo():
    return x

# test.py
import test1
x = 20
def foo():
    return x
print x  # 20
print foo()  # 20
print test1.x  # 45
print test1.foo()  # 45
```

When a module is in a different folder, more work needs to be done. Not explained here.
Example: modules we create (optional)

- Suppose we have two modules (.py files): test.py and test1.py in the same folder.
- Suppose you want to import test1.py with from into test.py

```python
# test1.py
x = 45
def foo():
    return x
```

```python
# test.py
x = 20  # Line 1
def foo():  # Line 2
    return x
print x  # 20
print foo()  # 20
from test1 import *
print x  # 45
print foo()  # 45
```

With this import, Line 1 and Line 2 are hidden!
Python comes with many modules

- **io**: read/write from files
- **math**: mathematical functions
- **random**: generate random numbers
- **string**: string functions
- **sys**: information about the OS (operating system)

Complete list: [http://docs.python.org/library](http://docs.python.org/library)

**Library**: built-in modules
- May change in each release
- Why version #s are an issue
string functions (revisited)

- Some string functions come by default in the base language:
  ```python
  s = 'hello all'
  s.count('ll')  # count, index are different kind of functions (called methods) that require s.
  s.index('ll')  # functions (called methods) that require s.
  
  s.len()  # s. would not work in this case because len(s) # len is a regular function using s as a parameter
  ```

- Others need to be imported:
  ```python
  import string
  string.capwords(s)  # Use string. in this case
  # Note that string is a module name
  # capwords is a regular function in that module
  ```
Do this before next class

- Finish reading Chapter 3
  - Try the examples using your python installation

- Next time
  - Conditionals (Chapter 5)