CIS 122

Functions Under the Surface
Functions Revisited

● We now have the power to write our own functions

```python
def plusOne(x):
    # Adds one to x
    return x+1
```

● Who cares?
  ○ We could just write the same code outside a function...
    ■ y = plusOne(x)
    ■ y = x+1
  ○ Why do we need functions?
Functions Revisited

- Functions simplify coding
  - Easier to solve small problems
  - Construct building blocks

- Reduce redundancy
  - Don't write the same 5 lines of code over and over
  - Write one function and call it 5 times

- Explain code
  - Descriptive function names
Functions Revisited

● So what are functions exactly?
  ○ In Python, functions are another type of object
  ○ Just like ints, strings, ...

● def is just a fancy way of defining a function object

```python
def addOne(x):
    return x+1
```

```python
>>> foo = addOne
>>> foo(1)
2
```
Functions Revisited

- What can we do with functions?
  - We can **add** ints...
  - We can **slice** strings...
  - We can **call** functions

- Also, anything we can do with a normal value
  - Print out
  - Assign to a variable
  - Give as argument to a function
def foo(x):
    y = x+1
    z = x+y
    return z

a = 5
b = foo(a)
c = a+b
def foo(x):
    y = x+1
    z = x+y
    return z

a = 5
b = foo(a)
c = a+b
def foo(x):
    y = x + 1
    z = x + y
    return z

__main__
foo → <function object>

a = 5
b = foo(a)
c = a + b
Stack Diagrams

def foo(x):
    y = x+1
    z = x+y
    return z

__main__
    foo → <function object>
    a → 5

a = 5
b = foo(a)
c = a+b
def foo(x):
    y = x+1
    z = x+y
    return z

__main__
    foo → <function object>
    a → 5
    b → ???

a = 5
b = foo(a)
c = a+b
def foo(x):
    y = x+1
    z = x+y
    return z

__main__
    foo → <function object>
    a → 5
    b → ???

a = 5
b = foo(a)
c = a+b
def foo(x):
    y = x+1
    z = x+y
    return z

__main__
    foo → <function object>
    a → 5
    b → ???

a = 5
b = foo(a)
c = a+b

foo
    x → 5
def foo(x):
    y = x+1
    z = x+y
    return z

__main__
foo → <function object>
a → 5
b → ???

a = 5
b = foo(a)
c = a+b

foo
x → 5
y → 6
```python
def foo(x):
    y = x + 1
    z = x + y
    return z

__main__
    foo → <function object>
    a → 5
    b → ???

foo
    x → 5
    y → 6
    z → 11
```

```
def foo(x):
    y = x + 1
    z = x + y
    return z

__main__
foo → <function object>
a → 5
b → ???

foo
x → 5
y → 6
z → 11

a = 5
b = foo(a)
c = a + b
def foo(x):
    y = x+1
    z = x+y
    return z

a = 5
b = foo(a)
c = a+b

__main__
foo → <function object>
a → 5
b → 11

foo
x → 5
y → 6
z → 11
def foo(x):
    y = x+1
    z = x+y
    return z

__main__

foo → <function object>
a → 5
b → 11
c → 16

c = a+b

foo

x → 5
y → 6
z → 11
Keeping track of your code

- Code doesn't always run linearly
  - During function calls, other code is put on hold
  - Python creates a new **stack frame** in memory
  - These stack frames can nest

- Who's seen the movie Inception?
More Fun with Functions

● Functions can take more than one argument
  ○ Just put more arguments in the header
    
    ```python
    def sum(a, b):
        """Adds two numbers together""
        return a + b
    ```

● Functions can take no arguments
  ○ Maybe you want to wrap up some computation...
    
    ```python
    def returnFive():
        """Returns five""
        return 5
    ```

● How would we write a power function?
More Fun with Functions

• Functions can call other functions
  ○ Good for breaking problems down
    def countRedSkittles():
        <skittle counting code>

    def countBlueSkittles():
        <skittle counting code>

    def countAllSkittles():
        """Returns a total skittle count""
        red = countRedSkittles()
        blue = countBlueSkittles()
        return red + blue
Variable Scoping

- Variables exist within a specific scope
  - Only make sense within a certain context

- Variables within a function cannot be seen from outside
  - Don't overwrite outside variables
  - Deleted when function ends
def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)
def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)
def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)
Variable Scoping

def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)
def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)
Variable Scoping

```python
def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)
```

```
__main__
foo → <function object>
x → 5
y → ???
```

```
x = 5
y = foo(6)
```

```
foo
```
Variable Scoping

def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)

__main__

foo → <function object>
x → 5
y → ???

foo

x → 6
```python
def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)

__main__
foo → <function object>
x → 5
y → ???

foo
x → 6
z → 7
```
def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)

__main__
foo → <function object>
x → 5
y → ???

foo
x → 6
z → 7
def foo(x):
    z = x + 1
    return z

x = 5
y = foo(6)

___main___
foo → <function object>
x → 5
y → 7

foo
x → 6
z → 7
Variable Scoping

- Why is variable scoping important?
  - Lots of built in functions in Python
  - We don't know (or care) how they're written
  - My code shouldn't depend on someone else's variable names!
def foo(x):
    y = x + 5
    z = bar(x, y)
    return z

def bar(a, b):
    c = a * b
    return c

a = 2
b = foo(a)
Function Quiz

```python
__main__

def foo(x):
y = x + 5
z = bar(x, y)
return z

def bar(a, b):
c = a * b
return c

a = 2
b = foo(a)
```

---

```
<function object>
<function object>

a → 2
b → 14

c

x → 2
y → 7
z → 14
```

---

```
foo

x → 2
y → 7
z → 14
```

---

```
bar

a → 2
b → 7
c → 14
```