Functions

We want to encapsulate code, that is combine code together so that we can reuse it at multiple places. This is a **function**.

```
from math import pi # importing python builtins
def circle_area(radius):
    area = pi*radius*radius # calculate area
    return area # return the area
```

A function takes 0 or more arguments and returns a value. If `return` is not specified, python sneaks in one for you, its the special value `None`.

Functions can take multiple arguments and return multiple things.

```
def f(a,b):
    return a+b, a-b
print(f(1,2))
```

Returns a tuple: `(3, -1)`

Functions can have default arguments, which can be named. See on right side.

```
def f(a, b=5):
    return a+b, a-b
```

- `f(1,2)` returns `(3, -1)`
- `f(1,b=2)` returns `(3, -1)`: this form is for self-documentation
- `f(1)` returns `(6, -4)`, the default argument has applied.

Where are functions defined? lambda functions

Besides functions using the `def` syntax, you can define your own **anonymous** functions and assign them to variables. Then you can call them with the variable name. These are great for math functions.

Anonymous function: `lambda x: 5*x + 4`

```
square = lambda x: x*x
square(2)
```

gives 4

```
sos = lambda x, y : x*x + y*y
sos(3, 4)
```

gives 25

Where are functions defined? Built-in functions.

Python defines a lot of built-in functions. Examples you have seen are `set`, `list`, `dict`, and `id`. Here's two useful examples.

```
days = ['M', 'T']
for i, day in enumerate(days):
    print(i, day)
```

gives

```
0 M
1 T
```

```
days = ['M', 'T']
letters = ['A', 'B']
for letter, day in zip(letters, days):
    print(letter, day)
```

gives

```
A M
B T
```

Where are functions defined? Importing from modules

A module is a collection of values like π and functions like `sqrt` which have a common purposes. We saw earlier the symbol `pi` being imported. We can also do:

```
from math import sqrt
sqrt(4)
```

returns 2.0 . Or:

```
import math
math.sqrt(4)
```

We can import functions from modules to do our work:

```
from math import sqrt
hypot = lambda x, y : sqrt(x*x + y*y)
hypot(3, 4) # returns 5
```

Variable Scope

In python, variables are visible in the *scope* they are defined in, and all scopes inside.

The scope of the jupyter notebook is the **global scope**.

Functions can use variables defined in the global scope.

```
c = 1
def f(a, b):
    return a + b + c, a - b + c
```

`f(1,2)` returns `(4, 0)`

Variables defined *locally* will shadow globals.

```
c = 1
def f(a, b):
    c = 2
    return a + b + c, a - b + c
```

`f(1,2)` returns `(5, 1)`

Variables defined *locally* are not available outside.

```
def f(a, b):
    return a + b, a - b
f(1,2)
print(a)
```

Output:

```
NameError: name 'a' is not defined
```

Variables defined in loops (including loop index) are available after.

```
for m in range(2):
    print(m)
print(m)
```

gives us:

```
0
1
1
```

Functions are first class objects

Functions can be assigned to variables: `hypot = lambda x, y : sqrt(x*x + y*y)`.

Functions can also be passed to functions and returned from functions.

Functions passed:

```
def mapit(aseq, func):
    return [func(e) for e in aseq]
mapit(range(3), lambda x : x*x) # [0, 1, 4]
```

`map` and `reduce` are famous built-in functions.

Functions returned:

```
def soa(f): # sum anything
    def h(x, y):
        return f(x) + f(y)
    return h
sos = soa(lambda x: x*x)
sos(3, 4) # returns 25 like before
```

Functions and State

Sometimes we want to capture some state:

```
def soaplusbias(f, bias): # sum anything
    def h(x, y):
        return f(x) + f(y) + bias
    return h
sosplusbias = soaplusbias(lambda x: x*x, 5)
sosplusbias(3, 4)
```

Returns 30

The last line is indentical to before, but we have additionally captured a `bias` from the enclosing scope. The bias is captured when we define `sosplusbias`, but gets used when we call it. This is called a **closure**. It is useful at all sorts of places where state must be captured and used later, as in deep learning callbacks, GUIs, etc

Function decorators

Decorators use the @ syntax and are a shortcut for a function wrapping another.

```
def factorial(n):
    return n*factorial(n-1)

def check_posint(f):
    def checker(n):
        if n > 0:
            return f(n)
        elif n == 0:
            return 1
        else:
            raise ValueError("Not a positive int")
    return checker

factorial = check_posint(factorial)
print(factorial(4)) # returns 24
print(factorial(-1)) # raises a ValueError
```

```
def check_posint(f):
    def checker(n):
        if n > 0:
            return f(n)
        elif n == 0:
            return 1
        else:
            raise ValueError("Not a positive int")
    return checker

@check_posint
def factorial(n):
    return n*factorial(n-1)

print(factorial(4)) # returns 24
print(factorial(-1)) # raises a ValueError
```

Refactoring the decorator to use a cache

```
def check_posint_and_cache(cache):
    def check_posint(f):
        def checker(n):
            if n > 0:
                if n in cache:
                    return cache[n]
                else:
                    val = f(n)
                    cache[n] = val
                    return val
            elif n == 0:
                return 1
            else:
                raise ValueError("Not a positive int")
        return checker
    return check_posint

global_cache3 = {}
@check_posint_and_cache(global_cache)
def factorial(n):
    return n*factorial(n-1)
```

The syntax on the left implements the following python code:

```
global_cache = {}
def factorial(n):
    return n*factorial(n-1)
factorial3 = check_posint_and_cache(global_cache)(factorial)
```

This code runs almost a factor of 1000 faster as it caches factorials
already computed into a global cache. So `factorial(20)` will be faster if `factorial(15)` was already run, since the firat 15 factorials are already completed.

The "state" captured is the results from previous runs. We needed two functions rather than one so that we can make it work with the python decorator syntax.