Decorators

Python is a live language. And function definition in python is actually a live thing.

In [1]:

```
def add(x, y=10):
    return x + y

print(add)
```

```
<function add at 0x7f1ad855c8c8>
```

Python interpreter tells where actually in memory this function exists. And this function is an object, so we can ask all sorts of things like what is name of it.

In [2]:

```
print(add.__name__)
```

```
add
```

In [3]:

```
print(add.__module__)
```

```
__main__
```

In [4]:

```
print(add.__defaults__)
```

```
(10,)
```

In [5]:

```
# ouputs the actual byte code of our add function
print(add.__code__.co_code)
```

```
b'|\x00|\x01\x17\x00S\x00'
```

In [6]:

```
# ouputs how many local variables does it have
print(add.__code__.co_nlocals)
```

```
2
```

In [7]:

```
# outputs what are the variable names in this function
print(add.__code__.co_varnames)
```

```
('x', 'y')
```

Every python structure that you interact with (whether it is an object or a function or a generator) has some runtime life. We can see in in memory and ask questions.

In [9]:

```
#inspect function can even give us a source code
from inspect import getsource
print(getsource(add))

#and we there are many more useful functions
```

```
def add(x, y=10):
    return x + y
```

How do we measure the execution time of a function?

In [11]:

```
from time import time

def add (x, y=10):
    return x + y

before = time()
print('add(10):', add(10))
after = time()
print('time_take:', after - before)
```

```
add(10): 20
time_take: 0.00026893615722265625
```

In [12]:

```
from time import time

def timer(func):
    def f(*args, **kwargs):
        before = time()
        rv = func(*args, **kwargs)
        after = time()
        print('elapsed', after - before)
        return rv
    return f

@timer
def add(x, y=10):
    return x + y

print(add(20))
```

```
elapsed 7.152557373046875e-07
30
```

A decorator is merely a syntax that is equivalent to the syntax *sub = timer(sub).*

Its syntax fits into the ability to dynamically construct a function to wrap this behaviour. So we were able to slip in the extra functionality we want withour rewriting the code. With *args and **kwargs our decorator works on functions with any parameters.

In [13]:

```
def ntimes(n):
    def inner(f):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                print('running {.__name__}'.format(f))
                rv = f(*args, **kwargs)
            return rv
        return wrapper
    return inner

@ntimes(2)
def add(x, y=10):
    return x + y

print(add(5))
```

```
running add
running add
15
```

This is called **higher-order decorators**. It is a fairly simplistic extension of the idea "a function can return a function". There is a concept "closure object duality".