Decorators

Recall the typical fib method.

$$0, 1, 1, 2, 3, 5, 8, \ldots$$

$$\begin{aligned}\text{fib}(0) &= 0\\ \text{fib}(1) &= 1\end{aligned}$$

$$\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$$

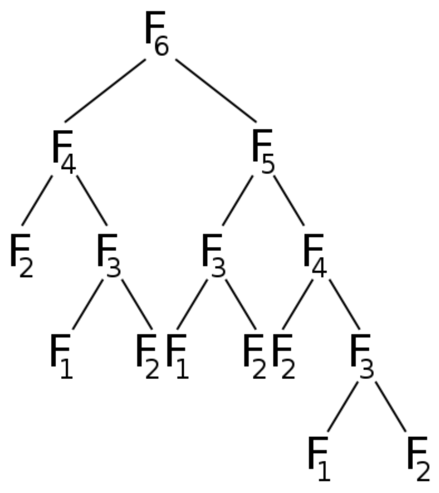
In [5]:
```
def fib(n):
    if n <= 1:
        return n

    else:
        return fib(n-2) + fib(n-1)
```

In [11]:
```
%time fib(35)
```

```
CPU times: user 6.17 s, sys: 141 ms, total: 6.31 s
Wall time: 6.54 s
```

Out[11]:
```
9227465
```

This will take a bit of time so let's see what's wrong.



We can use the concept of higher-order functions to tackle this issue. But let's take a step back for a minute.

In [12]:
```
def logger(f):

    # f will be "remembered" by the wrapper even after we exit logger.
    # This is the concept of a closure.

    def wrapper(n):
        print("I'm going to call a function.")
        v = f(n)
        print("The function returned: ", v)
        return v

    return wrapper
```

In [13]:
```
logged_fib = logger(fib)        # remember, fib is just a name!
```

In [14]:
```
logged_fib(4)
```

```
I'm going to call a function.
The function returned:  3
```

Out[14]:
```
3
```

Now that we can do stuff before the `fib` call, let's see if we can save some values that are repeatedly needed.

In [16]:
```
def memoize(f):                     # Essentially 'memorize values'
    mem = {}

    def memoized_function(n):       # typically called a wrapper
        if n not in mem:
            mem[n] = f(n)

        return mem[n]

    return memoized_function
```

In [17]:
```
fib = memoize(fib)                   # not calling fib at the moment!
```

In [20]:
```
%time fib(100)
```

```
CPU times: user 124 µs, sys: 66 µs, total: 190 µs
Wall time: 198 µs
```

Out[20]:
```
354224848179261915075
```

That's about 450,000 times speedup!

Syntactic Sugar

We can write this in another way.

In [21]:
```
def memoize(f):
    mem = {}

    def wrapper(x):
        if x not in mem:
            mem[x] = f(x)

        return mem[x]

    return wrapper
```

In [22]:
```
@memoize             # this is called a "decorator", EQUALS: fib = memoize(fib)
def fib(n):
    if n <= 1:
        return n

    else:
        return fib(n-1) + fib(n-2)
```

In [23]:
```
fib(50)
```

Out[23]:
```
12586269025
```

And now you can memoize (almost) any function with ease -- Just add the decorator to it.

Another example is: ensuring that a user is logged in before executing a function. See Django's login decorator here: https://docs.djangoproject.com/en/2.0/topics/auth/default/#the-login-required-decorator (https://docs.djangoproject.com/en/2.0/topics/auth/default/#the-login-required-decorator)

In [ ]:
```
```