

Functional Programming

You are not logged in.

Please [Log In](#) for full access to the web site.

Note that this link will take you to an external site (<https://shimmer.mit.edu>) to authenticate, and then you will be redirected back to this page.

This reading is relatively new, and your feedback will help us improve it! If you notice mistakes (big or small), if you have questions, if anything is unclear, if there are things not covered here that you'd like to see covered, or if you have any other suggestions; please get in touch during office hours or open lab hours, or via e-mail at 6.101-help@mit.edu.

Table of Contents

- [1\) Introduction](#)
- [2\) More Iteration to Recursion](#)
- [3\) Classes to Functions](#)
 - [3.1\) Aside: nonlocal](#)
- [4\) Memoization](#)

1) Introduction

Functions were one of the first Python features that we [studied](#) this semester, but they are a remarkably versatile feature that deserves more attention. *Functional programming* is a classic programming style heavily focused on functions. As recently as the mid-2000s, functional programming was seen by industry as a niche academic idea, but, in the mean time, it went quite mainstream. While it was originally associated with distinct programming languages like [Haskell](#), many of its best ideas were adopted by better-known languages like Python. We've been using those features (like `lambda`) all along this semester, and this reading will show you even more handy patterns in functional programming.

As it happens, one of the first programming languages that worked on multiple different kinds of computers was [LISP](#), and it arguably introduced functional programming as a practical style, way back in 1958. This week's lab will lead you through implementing your own subset of LISP. Getting a better understanding of functions and their uses is also directly relevant to that experience.

2) More Iteration to Recursion

One interesting feature of LISP is that it does not feature any looping structures (there are no `for` or `while` loops). But it does have functions, and, [as we've seen before](#), it's possible to use functions to implement loops. One of the first examples we looked at, factorial, could always be written with a loop like so:

```
def factorial(n):  
    result = 1  
    for i in range(1, n+1):
```

```
    result *= i
    return result
```

But we also learned that it can be written recursively as a one-liner.

```
def factorial(n):
    return 1 if n <= 1 else n * factorial(n-1)
```

In the LISP, every function needs to be written in this style, effectively as a single `return` with an expression that does all the heavy lifting!

Consider this example, which capitalizes all words in a list:

```
def all_caps_list(words):
    return [w.upper() for w in words]
```

We can apply another of our recursion patterns, the **first/rest** decomposition, to write a recursive version.

```
def all_caps_list(words):
    if len(words) == 1:
        return words[0].upper()
    else:
        return [words[0].upper()] + all_caps_list(words[1:])
```

How does this recursive version behave on lists of strings?

-- 


It may be surprising how many loops can be turned into instances of recursion. Consider this basic one.

```
for i in range(n):
    print('hello', i)
```

This structure might seem a little bit trickier to implement using recursion, but we'll see that we can indeed do it, using a **higher-order function** (meaning a function that takes a function as input). Here, we encode what was the body of our loop (i.e., the thing we want to repeat multiple times) as a function; then we call that function repeatedly, using recursion to keep track of how many times we still need to run that code.

```
def repeat_n_times(n, func):
    if n == 0:
        return
    func(n)
    repeat_n_times(n-1, func)
```

What happens when we call `repeat_n_times(20, lambda i: print('hello', i))`?

-- 

3) Classes to Functions

Object-oriented programming, e.g. with classes in Python, is commonly used in industry as a way to organize complex code bases. However, essentially any use of classes can be reimplemented with functions in Python. Let's look at two examples.

We can build a simple "bank" class that helps us maintain multiple named numeric balances.

```
class Bank:
    def __init__(self):
        self.accounts = {}

    def balance_of(self, account):
        return self.accounts.get(account, 0)

    def deposit(self, account, amount):
        self.accounts[account] = self.balance_of(account) + amount

>>> b = Bank()
>>> b.deposit('Steve', 3)
>>> b.deposit('Magdalena', 4)
>>> b.deposit('Steve', 5)
>>> b.balance_of('Steve')
8
>>> b.balance_of('Magdalena')
4
```

The chance to *encapsulate* the details of balance maintenance within a class pays off when it comes to writing and maintaining programs. However, functions also provide us all the ingredients we need to the same end. Consider the following code:

```
def bank():
    accounts = {}

    def balance_of(account):
        return accounts.get(account, 0)

    def deposit(account, amount):
        accounts[account] = balance_of(account) + amount

    return balance_of, deposit

>>> balance_of, deposit = bank()
>>> deposit('Steve', 3)
>>> deposit('Magdalena', 4)
>>> deposit('Steve', 5)
>>> balance_of('Steve')
8
>>> balance_of('Magdalena')
4
```

It's important that the two functions that `bank()` returns, `balance_of` and `deposit`, share a freshly created enclosing frame with a new `accounts` dictionary. This frame is equivalent to the object in the object-oriented version. If the details there are not clear, try drawing an environment diagram.

3.1) Aside: nonlocal

So far it looks like we'll have smooth sailing converting classes into functions. Here's a simpler example than the bank -- a counter that we can keep incrementing by one. But it reveals a subtle issue in Python with using variables from an enclosing frame to store information.

Show/Hide Line Numbers

```
1 def counter():
2     tally = 0
3
4     def increment():
5         tally += 1
6         return tally
7
8     return increment
```

Consider running `inc = counter()` and then repeatedly calling `inc()`. What happens when we do this?

--

This code exhibits a common pitfall in Python. Namely, variable `tally` in `increment` is being treated as *local* to that function rather than inherited from the enclosing frame. Why?

Because when a function *modifies* a variable, Python forces the variable to be present directly in the function's frame... but here `tally` was not initialized locally, so Python complains that `tally` does not exist.

But Python gives us a way to control this. In particular, if we *want* to use the variable `tally` from the enclosing frame inside of `increment`, we can accomplish this using the `nonlocal` keyword:

```
def counter():
    tally = 0

    def increment():
        nonlocal tally
        tally += 1
        return tally

    return increment
```

The line `nonlocal tally` tells Python that whenever we refer to the name `tally` inside of `increment` (whether we're just looking up the value *or* setting/changing it), we should always be looking for the variable in one of our parent frames, instead of in the local frame for calling `increment`.

Check Yourself:

Look back at the `bank` example from above:

```
def bank():  
    accounts = {}  
  
    def balance_of(account):  
        return accounts.get(account, 0)  
  
    def deposit(account, amount):  
        accounts[account] = balance_of(account) + amount  
  
    return balance_of, deposit
```

The functions `balance_of` and `deposit` are using `accounts` from their enclosing frame to update the balances of bank accounts.

What's the difference between the way `bank` uses a variable from an enclosing frame and the way that `counter` does? Why does `counter` need `nonlocal tally`, but `bank` doesn't need to say `nonlocal accounts`?

[Show/Hide](#)

4) Memoization

We can also use higher-order functions to improve the performance of naive code. Here's a classic of recursion, a calculator for [Fibonacci numbers](#).

```
def fib(n):  
    if n < 2:  
        return n  
    return fib(n-2) + fib(n-1)
```

Check Yourself:

Roughly how long will it take to compute `fib(100)`?

[Show/Hide](#)

We can start to see why this is so slow by looking at a tree showing the recursive calls that are made when computing `fib(100)`. We start with a single top-level call:

`fib(100)`

That then makes two recursive calls:

`fib(100)`
`fib(99)` `fib(98)`

Each of those makes two recursive calls:

`fib(100)`
`fib(99)` `fib(98)`
`fib(98)` `fib(97)` `fib(97)` `fib(96)`

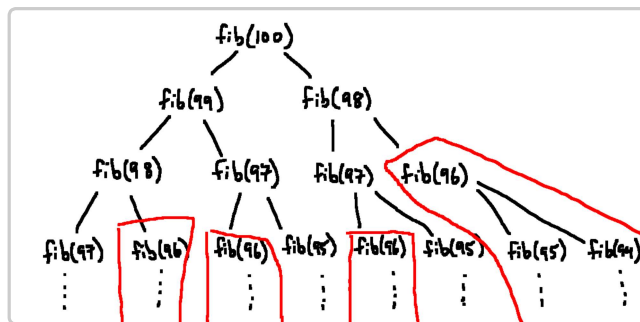
And this pattern continues:

`fib(100)`
`fib(99)` `fib(98)`
`fib(98)` `fib(97)` `fib(97)` `fib(96)`
`fib(97)` `fib(96)` `fib(96)` `fib(95)` `fib(96)` `fib(95)` `fib(95)` `fib(94)`
 \vdots \vdots \vdots \vdots \vdots \vdots \vdots \vdots

How many times will we ultimately make a call computing the 94th Fibonacci number, like `fib(94)`, in the structure above? Enter your answer as an integer below:

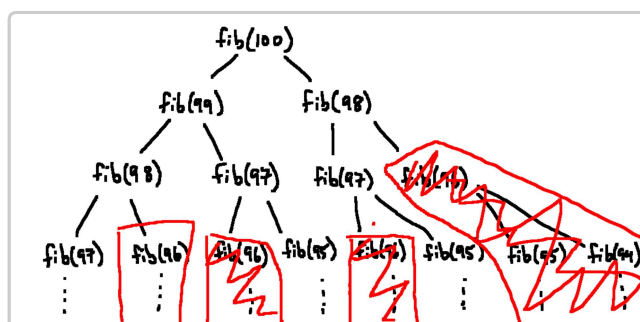
But there's something interesting here, which we can exploit: even though we're calling `fib(94)` multiple times, the result is always the same. But in order to find that result (as things are currently written), Python needs to explore the *entire* subtree of the drawing above that's rooted at 94, which takes a *long time*.

We can see some of this repeated structure by highlighting the subtrees associated with any given number, in this case 96:

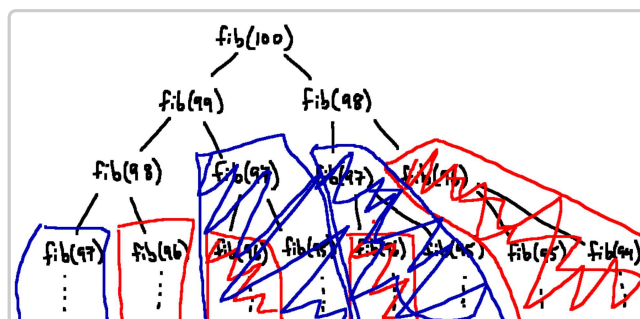


Any time we call `fib(96)` (multiple times shown here), Python will need to explore an entire subtree to find that result, and the result will always be the same.

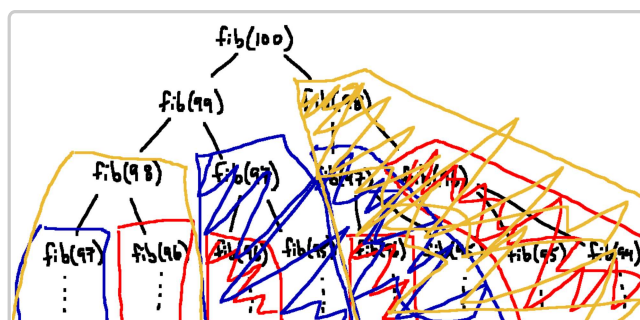
What we'll try next is exploiting this repeated structure in the tree above by *remembering* Fibonacci numbers as we compute them, so that we can reuse those results later. So, for example, when we call `fib(96)`, we'll first ask: have I already computed this result? If so, let's avoid repeating the work that took and just return the result I've remembered. Otherwise, let's do the computation normally. In that way, we can avoid exploring large swathes of the tree. For example, just remembering the result of `fib(96)` lets us cut out big parts of the tree:



But then we can do the same of `fib(97)`:



and `fib(98)`:



In fact, when we cross out all subtrees that are redundant in this way, we are left with a remaining tree that, in general for `fib(n)`, only has about n nodes that aren't crossed out. Now *that* is much more tractable to compute.

Here's a Python realization of the idea:

```

cache = {}

def fib(n):
    if n not in cache:
        if n < 2:
            cache[n] = n
        else:
            cache[n] = fib(n-2) + fib(n-1)
    return cache[n]

```

What's up with the word "cache"? The English word referred originally to something like the hiding place where a pirate stashed his treasure, but in computer science, a [cache](#) stores values that we think we will be consulting again soon.

We call the new `fib` **memoized**. No, that's not a typo for "memorized." "[Memoization](#)" is a computer-science term for saving and reusing intermediate results, particularly in problems with a lot of repeated structure like this one.

This code runs much more quickly, but it has a downside: it uses a global variable. All sorts of things can go wrong in larger code bases that include global variables. For instance, what if we wanted to use the same trick for a factorial function?

```

cache = {}

def factorial(n):
    if n not in cache:
        if n < 2:
            cache[n] = 1
        else:
            cache[n] = n * factorial(n-1)
    return cache[n]

```

Whoops, now both functions are sharing the same `cache` variable, and havoc ensues! (The initial assignment overwrites the existing global variable rather than creating a new one.) We would rather not force the authors of the two functions to coordinate on variable-name selection.

Here's a version that avoids that issue by putting `cache` inside of an enclosing frame:

```

def fib(n):
    cache = {}

    def _actual_fib(n):
        if n not in cache:
            if n < 2:
                cache[n] = n
            else:
                cache[n] = _actual_fib(n-2) + _actual_fib(n-1)
        return cache[n]

    return _actual_fib(n)

```

How does this version behave?

-- 

OK, one major complaint handled. However, you probably also noticed strong similarities between the memoized Fibonacci and factorial functions. Can we factor the common parts out into some reusable ingredient?

One way to implement this would be to use a class, like so:

```
class MemoizedFunction:
    def __init__(self, func):
        self.func = func
        self.cache = {}

    def __call__(self, *args):
        if args not in self.cache:
            self.cache[args] = self.func(*args)
        return self.cache[args]
```

We've already seen most of the ingredients used here, though one new piece of magic is the `__call__` method. Whenever code tries to call some class instance like a function, Python calls that instance's `__call__` method. In other words, we can define our own new function-like things by explaining what it means to call them.

Another new piece of magic is the `*` syntax applied to a formal parameter, here called `*args`. This allows the function to be called with any number of actual arguments, which are bundled up together into a *tuple* and bound to the formal parameter `args`. (Note that there is nothing special about the name `args`. It's just conventionally used for variable-length arguments like this, just as `self` is conventionally used for the `self` parameter. And there's nothing special about using it in the `__call__` dunder method. You can use `*` to handle variable-length arguments in any function you define.)

Finally, the code also uses `*args` in its body, but we've seen that before -- it's the [unpacking operator](#), so it unpacks the tuple back into individual arguments for the wrapped function `func()`.

Now we can add memoization to the original, naive Fibonacci function, like so.

```
def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)
fib = MemoizedFunction(fib)
```

But, as we saw earlier on, it turns out we don't need classes to implement generic memoization. Functions are more than capable, too!

```
def memoize(func):
    cache = {}

    def _mfunc(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]

    return _mfunc
```

Again, naive Fibonacci is easily memoized.

```
def fib(n):
    if n < 2:
        return n
```

```
    return fib(n-2) + fib(n-1)
fib = memoize(fib)
```

This pattern is so common, of using a library function to augment a function definition, that Python has special syntax for it. We indicate what's called a **decorator** by putting an "at" sign and a function name before another function's definition.

```
@memoize
def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)
```

You may remember that we used the same trick in recitation, to instrument functions to log all recursive calls to the screen. Now you know that it's not magic, and you are free to use the same trick in your own code!