CS61A Lecture 4

Wednesday, September 4th 2019

Iteration

The Fibonacci sequence was popularized by Fibonacci, a mathematician from long ago. It starts with 0 and then 1, and every element after that is the sum of the previous two elements.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

Every Fibonacci number is associated with its index, its position in the sequence. It begins with the 0th number, the 1st, the 2nd and so on.

We can create interesting patterns with the Fibonacci sequence. By tiling together squares whose side lengths are Fibonacci numbers, we can create the golden spiral.

It's a spiral that looks well-balanced to the human eye, and it is something people look for in nature.

Here is how to calculate the nth fibonacci number using a while loop, doing so by keeping track of various values and then executing a while statement.

While designing an iterative function, one of the most important things to think about is what information to keep track of in order to perform the iteration.

```
def fib(n):
    pred, curr = 0, 1   # 0th and 1st Fibonacci number
    k = 1               # k keeps track of the index
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1       # this keeps track of the index. When the current index reaches
    return curr
```

The following definition of `fib(n)` is better than our previous definition, because it properly returns `n=0`:

```
def fib(n):
    pred, curr = 1, 0
    k = 0
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

Designing Functions

There are lots of different functions that do the same thing, but the best ones are the ones that are easier for people to read and understand, and they are applicable in a wide range of situations.

Functions didn't use to exist, where programs used to just be a collection of statements. But functions are now widely regarded as the right way to write large programs.

- A function's domain is the set of all inputs it might possibly take as arguments.
- A function's range is the set of output values it might possibly return.

Knowing the domain and range of a function tells you where it can be used, because it tells you what goes in and out.

A pure function's behavior is the relationship it creates between the input and the output.

Let's take a look at two previous examples:

```
def square(x):
```

- Domain: real numbers `x`
- Range: non-negative real number
- Behavior: return value is square of its input

```
def fib(n):
```

- Domain: integer `n` greater than or equal to 1
- Range: Fibonacci number
- Behavior: return value is the nth Fibonacci number

Guide to Designing Functions

Heuristics, based on experience, but hold up in lots of situations.

- Give each function exactly one job.
- Don't repeat yourself (DRY). Implement a process just once, but execute it many times.
- Define functions generally.

Model your functions after scirssors, which do exactly one thing, not like a Swiss army kinfe, which do many things. They should probably be in different functions.

Higher Order Functions

- Feature of a programming language, that allow us to design functions by expressing very general methods of computation.

Generalizing Patterns with Arguments

What we want to do is to generalize patterns by defining functions that take arguments that give us back the specific instances of those arguments.

For example, the area of a geometric shape is something we could generalize.

- Square of length `r` has area of `r**2`
- Circle of radius `r` has area of `pi*r**2`
- Hexagon of radius `r` has area of `3/2*\sqrt(3)*r**2`

What these areas share in common is the constants in front of `r**2`. Finding common structure allows for shared implementation.

```
from math import pi, sqrt

def area_square(r):
    return r * r

def area_circle(r):
    return r * r * pi

def area_hexagon(r):
    return r * r * 3 * sqrt(3) / 2
```

The problem with this code is that `area_hexagon(10)` and `area_hexagon(-10)` return the same value, which is not quite right. We can fix this with what we call an `assert` statement.

`assert` statements are its keyword, a Boolean expression, and an error message. If the result of the expression is True, and the statement is run, nothing happens. Meanwhile, if the result is False, then the error message appears.

We can put an assert statement in each of the above codes, but then we'd be repeating ourself. Instead, we can generalize:

```
def area(r, shape_constant):
    assert r > 0, 'Length must be positive.'
    return r * r * shape_constant
```

This function isn't that intuitive and requires documentation, but what this allows us to do is take advantage of a common implementation in defining the other areas.

```
def area_square(r):
    return area(r, 1)

def area_circle(r):
    return area(r, pi)

def area_hexagon(r):
    return area(r, 3 * sqrt(3) / 2)
```

If we now try to execute `area_hexagon(10)`, it will work just as before, but `area_hexagon(-10)` returns an assertion error that the length must be positive.

Generalizing Over Computational Process

- The common structure among functions may not be a number, but a computational process, which is more complicated.
- Examine the following summations:
  -
  -
  -

Each of these have a common computational process, but also something specific about them, which is an expression and not just a number.

Let's write code to generalize this as well:

```
def sum_naturals(n):
    """Sum the first N natural numbers.

    >>> sum_naturals(5)
    15
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + k, k + 1
    return total
```

So the function above sums natural numbers. Now, let's write a function that sums cubes.

```
def sum_cubes(n):
    """Sum the first N cubes of natural numbers.
    >>> sum_cubes(5)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + pow(k, 3), k + 1
    return total
```

This function works, but we had to repeat ourselves to get here. Maybe we can do better. First, let's write an identity function that takes a number `n`, then returns it right back.

```
def identity(k):
    return k
```

Next, we need to write how cubes are calculated.

```
def cube(k):
```

```
    return pow(k, 3)
```

Now, let's solve for the generalization over `sum_naturals` and `sum_cubes`, which takes in the number of natural numbers we are going to sum over. We are also going to take in a function that tells us how to compute each term of the summation. We will repeat the previous logic in its most general form.

```
def summation(n, term):
    """Sum the first N terms of a sequence.
    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
```

From this code, we need `term` to be a function that returns how to compute the kth term of this sequence.

Now, we can replace our previous definitions.

```
def sum_naturals(n):
    return summation(n, identity)

def sum_cubes(n):
    return summation(n, cube)
```

This does exactly the same thing,

> **What happened?**
> We generalized over all these things, by for example, defining a cube function, then by defining a function, which we'll call term, which does the summing.

`cube` is a function of a single argument, and term is a formal parameter in `summation` that will be found to a function.

We do that so we can call refer to whatever computes each term within the body of `summation`.The function is passed in as an argument value, and the function bound to term gets called in the iteration.

That is a higher-order function, a function that takes another function as an argument.

Returning Functions and Locally Defined Functions

So we went through all this work to come up with a general version of summation, and we used it to sum natural numbers of cubes, but what about the summation of those terms that converge to 3.04 (close to pi).

Then, we can just define a new function called `pi_term`:

```
def pi_term(k):
    return 8/((4k-3)*(4k-1))
```

Now, let's see how we can define a function that returns a function as a value.

```
def make_adder(n): #Takes a numerical argument called N:
    """Return a function that takes one argument K and return K + N.
    >>> add_three = make_adder(3)
    >>> add_thrree(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

Now let us explain the structure of this. This is a def statement with another def statement within it. There are two return functions, one part of the inner function that returns a value, and the other part of the outer function that returns a function.

`adder` can use names that are its formal parameter and the formal parameter of `make_adder`.

Locally Defined Functions

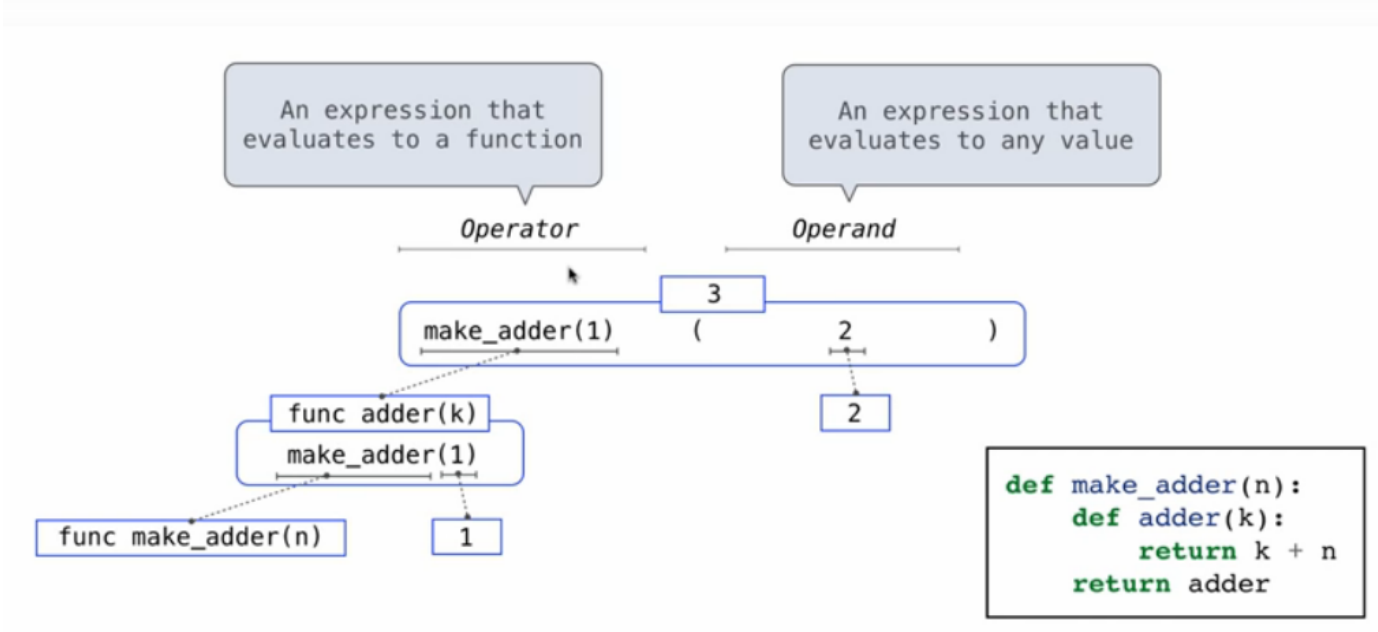
Functions defined within other functions are bound to names in a local frame. Both `k` and `n` are available to use for the `adder` function.

Since `make_adder` returns a function, this means we can also do things like:

```
make_adder(1)(2)
```

In this case, `make_adder(1)` becomes the operator (a call expression), and `2` is the operand.

An operator is any expression that evaluates to a function, which is what we get back from `make_adder`, while the operand is an expression that evaluates to any value.



The Purpose of Higher-Order Functions

- Functions are first-class values: functions can be manipulated as values in our programming language, meaning they can be passed ass arguments and returned as argumnets, just like any value in our programming language.
- Higher-order function: a function that takes a function as an argument value or returns a function as a return value
    - They express general methods of computation, and not having to worry about low-level details.
    - Remove repetition from programs, because we only need to define the general method once.
    - Separate concerns among functions, so that each function has exactly one job, and those jobs are either general methods of computation, or specific, if they are taken in as arguments or returned as values.

Lambda Expressions

- Expressions to evaluate to functions.
- We already know we can bind some value to a name.
- Would it be nice if we could bind a new function to a name, using the same syntax and assignment statement?
- Turns out, you can.
- We could just write what would be in the body of the function, for example:

```
>>> x = 10
>>> square = x * x # This is an expression that evaluates to a number
>>> x
10
>>> square
100
```

But this isn't a function at all. It's just the number 100 in this case, because names are bound to values.

Lambda expressions allow us to do what we wanted in the first place. They allow us to bind to the name square, using an assignment statement, that takes in some argument `x`, and computes `x * x` as its return value.

```
>>> square = lambda x: x * x # This is also an expression, but it evaluates to a functio
>>> square
<function ...>
>>> square(4)
16
>>> square(10)
100
```

A lambda expression looks like the above, and it's used to evaluate to functions, or we could even use it as part of a call expression.

```
>>> (lambda x: x * x)(3)
9
```

Reading a lambda expression is easy.

```
lambda      # A function
x:          # with formal parameter x
x * x       # that returns the value of "x * x"
```

Eventually, people will start to use the word lambda in casual conversation. Lambda functions do not have return keyword, where you just write down the return value directly after the colon.

Lambda expressions can only have a single expression as the body of the function you create. These expressions are thus always simple functions that do nothing but evaluate a single expression.

Lambda expressions are not common in Python, but they are really important in general, and in some programming languages, they are totally fundamental, and some languages that never had them are now starting to have them.

Lambda expressions in Python cannot contain statements like if or while at all, these must be in a def statement.

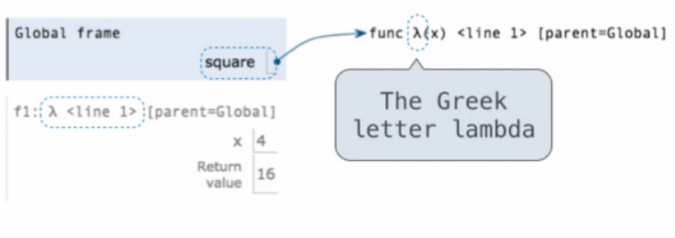
Lambda expressions versus def statements

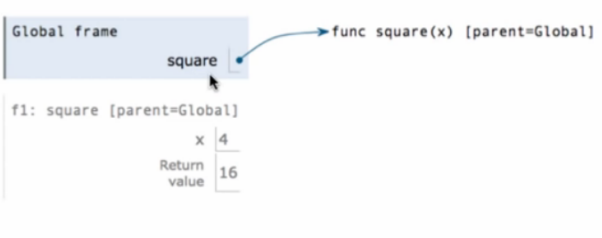
How are these different?

```
square = lambda x: x * x

def square(x):
    return x * x
```

- They’re almost exactly the same, as both create a function with the same domain, range and behavior.
- Just like a def statement, they are not evaluated at definition, but at the time they are called.
- Both functions have as their parent the frame in which they were defined, so the rules of creating functions remain the same.
- Both bind the function to the name square. However,
  - In a lambda expression, the function is created with no name at all, and the assignment statement gives it a name.
  - In a def statement, both of those things happen automatically.
- Only the def statement gives the function an intrinsic name.
  - What’s an intrinsic name? It is what happens when you type the function only in the interpeeter.
  - `>>> square`
    `function <lambda>...`
  - `>>> square`
    `function square`
- This difference changes how we draw environment diagrams:
- Def statements have names when they were created, while lambda expressions don’t until the assignment statement gave it one.


Global frame
square
func λ(x) <line 1> [parent=Global]
f1: λ <line 1> [parent=Global]
x 4
Return value 16
The Greek letter lambda


Global frame
square
func square(x) [parent=Global]
f1: square [parent=Global]
x 4
Return value 16