Lecture 4: Higher Order Functions

June 24, 2022

Cooper Bedin

Announcements

- There's still a pandemic! This is a big lecture hall, and while by university policy we can't enforce mask-wearing, we do encourage it
    - If you have symptoms (or test positive) **please stay home**—that's why these lectures are recorded
    - If you need to attend a remote section, go to sections.cs61a.org and look for Rachel and Charlotte's sections—their zoom links should be right there (look for the links with "go" in them), so you don't need to switch your enrolled section
- By popular demand, we are opening a new section 9:30-11am! Go to sections.cs61a.org and look for Elisa's section (sign-ups are open now)
- Lab 0 due tonight—technical OH today if you still need help
- New resources starting next week
    - Small group tutoring (sign-ups open Saturday at noon)
    - Study hall Wednesday 4-5 (see the oh page for more details)
- Next week we kick into a higher pace so rest up and make sure you feel good about this week's material!

Now: More Iteration!

Virahanka-Fibonacci numbers

Famous mathematical sequence! You may have heard of it

Discovered by Virahanka in India, 600-800 AD, later re-discovered in Western mathematics—also commonly known as just “Fibonacci numbers”

We like to use it a lot in CS classes—you’ll see it again in this class!

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

```
2 + 3 = 5
```

Virahanka-Fibonacci numbers

A lot of the time, we'll describe Virahanka-Fibonacci numbers by their indices in the sequence:

```
0,   1,   1,   2,   3,   5,   8,   13,  21,  34
0th  1st  2nd  3rd  4th  5th  6th  7th  8th  9th
```

Let's write a function that calculates the *n*th Virahanka-Fibonacci number in the sequence!

Because we need to know the previous two numbers to calculate the *n*th one, we'll have to use iteration

Virahanka-Fibonacci numbers

```
def vir_fib(n):
    """Calculate the Nth Virahanka-Fibonacci number"""
    pred, curr = 0, 1
    k = 1
    while k <= n:
        pred, curr = curr, pred + curr
        k += 1
    return curr
```

Let's see what happens!

PythonTutor

Digit manipulation

Digit manipulation is the practice of treating a number as a string of digits, mostly by using `<n> % 10` and `<n> // 10`

It's not something you're likely to see a lot in real-world programming. However, it's a great way to practice iteration without having to learn data structures like lists (that's for next week)

Let's try a digit manipulation problem—writing a function that finds the largest digit in a number

Largest digit

```python
def largest_digit(n):
    """
    >>> largest_digit(15342)
    5
    """
    current_largest = 0
    while n: # or, while n > 0
        current_digit = n % 10
        if current_digit > current_largest:
            current_largest = current_digit
        n = n // 10
    return current_largest
```

PythonTutor

Designing Functions

How do we talk about functions?

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

A function's **domain** is the set of all possible inputs it can take

`square` can take in any single number for x

A function's **range** is the set of all possible outputs it can give

`square` returns a non-negative (real) number

A function's **behavior** is the relationship between inputs and outputs

`square` returns the square of **x**

Designing functions

Give a function exactly one job, but have it apply to many related situations:

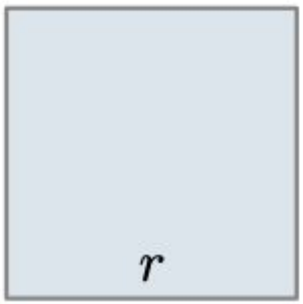
```
round(1.23)     # 1
round(1.23, 0) # 1
round(1.23, 1) # 1.2
round(1.23, 5) # 1.25
```

DRY (Don't Repeat Yourself) - Implement a process once, use it many times

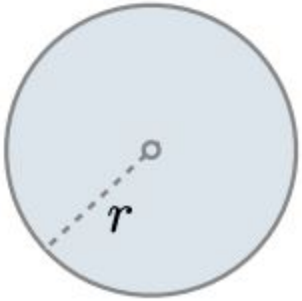
Generalization

Example: Area formulas

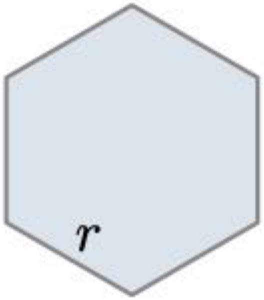
Many shapes have really similar formulas to calculate their areas:



$1 * r^2$



$\pi * r^2$



$3\sqrt{3} / 2 * r^2$

We can exploit patterns like this when we write functions!

Non-generalized :(

```
from math import pi, sqrt

def area_square(r):
    return r * r

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

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

All these functions look similar—how can we use that?

Generalized :')

```python
from math import pi, sqrt

def area(r, shape_constant):
    """Return the area of a shape from
    length measurement R"""
    return r * r * shape_constant

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)
```

Now: Higher-Order Functions

What are higher order functions?

Functions are "first class" in Python—this means that they can be manipulated in the same ways as other pieces of data such as numbers and booleans (this is not true of all programming languages!)

Higher-order functions take advantage of this

Any function is considered "higher-order" if it either:

- Takes in a function as an input
- Returns a function as an output

This lets us write code that's even more generalized

Even more generalized :"""""""")

```
from math import pi, sqrt

def make_area_function(shape_constant):
    """Returns a function that calculates the area
    of a shape, depending on SHAPE_CONSTANT"""
    def area_function(r):
        return r * r * shape_constant
    return area_function

area_square = make_area_function(1)

area_circle = make_area_function(pi)

area_hexagon = make_area_function(3 * sqrt(3) / 2)
```

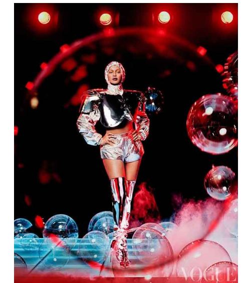
Functions as return values

Locally defined functions

```
>>> from math import pi
>>> def make_area_function(shape_constant):
...     def area_function(r):
...         return r * r * shape_constant
...     return area_function
...
>>> area_square = make_area_function(1)
>>> area_square(10)
100
>>> make_area_function(pi)(10) # :0
314.159265358979
```

Functions defined within the bodies of other functions are bound to names in a local frame

What's happening?



314.159265358

`make_area_function(pi)(10)`

Operator

Operand

func area_function

`make_area_function(pi)`

Operator

Operand

`func make_area_function...`

`3.1415926...`

```
def make_area_function(shape_constant):
    def area_function(r):
        return r * r * shape_constant
    return area_function
```

func area_function

Functions as inputs

Function composition

```
from math import pi

def compose(f, g, x):
    return f(g(x))

def mul_by_pi(n):
    return pi * n

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

PythonTutor

Lambda functions

Lambda syntax

A lambda expression is a simple function definition—an expression that fits in a single line and evaluate to functions. They look like this:

```
lambda <parameters>: <expression>
```

^^^Python will evaluate this to a function that takes in `<parameters>` and returns `<expression>`

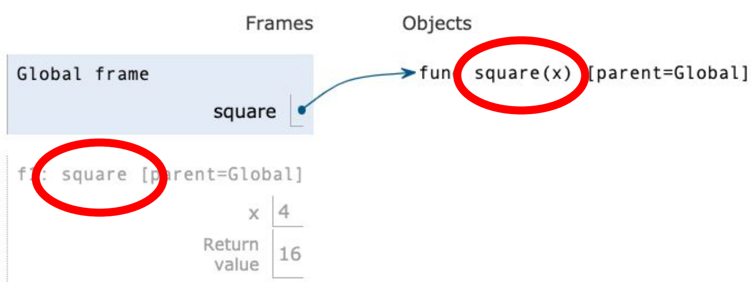
Here's an example of how we could rewrite the `square` function using a lambda expression:

```
square = lambda x: x * x
```

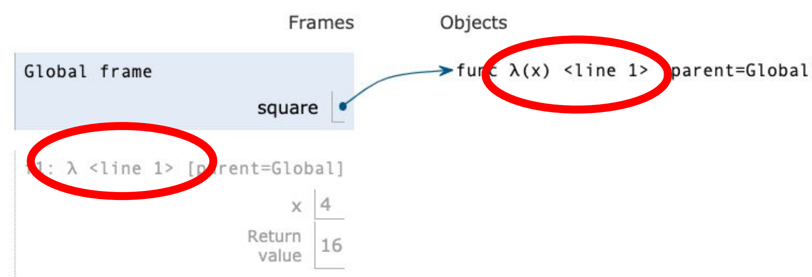
^^^This binds the name `square` to a function that takes in `x` as a parameter and returns `x * x`

Lambda vs Def

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




```python
square = lambda x: x * x
```



Both create functions with exactly the same domain, range, and behavior

Both bind that function to the name, square

The main difference is that the def statement will give the function an **intrinsic** name, while the lambda expression will create an **anonymous** function—this matters in environment diagrams, but won't affect execution

A particularly contrived example

```
from math import pi

make_composed = lambda f, g: lambda x: f(g(x)

mul_by_pi = lambda x: pi * x

square = lambda x: x * x

area_of_circle = make_composed(mul_by_pi, square)

area_of_circle(10)
```

PythonTutor

Summary

- As we start to design functions ourselves, we want to think about giving them well-defined jobs that can apply to many situations
- Well defined functions can help reduce redundancy in our code, which makes it more readable and adaptable
- Higher-order functions are functions that can take other functions as input, or produce other functions as output—they can help us further reduce redundancy in our code
- Lambda expressions are a quick way to define simple functions within a single line