Lecture 4: Higher-Order Functions

CS 61A - Summer 2024

Raymond Tan

Review: Boolean operators

Boolean Operators

- **not**
  - returns the opposite boolean value of an expression
  - will always return either **True** or **False**

- **and**
  - evaluates expressions in order
  - stops evaluating (short-circuits) at the first *falsey* value and returns it
  - if all values evaluate to a *truthy* value, the last value is returned

- **or**
  - evaluates expressions in order
  - stops evaluating (short-circuits) at the first *truthy* value and returns it
  - if all values evaluate to a *falsey* value, the last value is returned

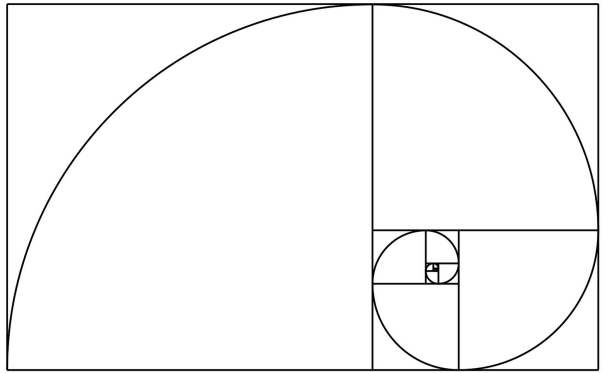
Demo: Short-circuiting

Iteration cont

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987…

Each term is defined as the sum of the previous two terms
(with the exception of term #0 and #1)



The Fibonacci Sequence - In Python!

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # 0th and 1st fibonacci number
    k = 1             # curr is the kth fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987…

Go bears!

Generalization

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

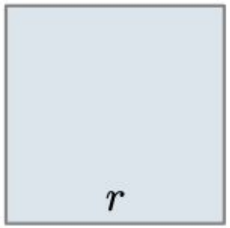
Give a function exactly one job, but have it apply to many related situations:

```
round(1.23)    # 1
round(1.23, 0) # 1
round(1.23, 1) # 1.2
round(1.23, 2) # 1.23
```

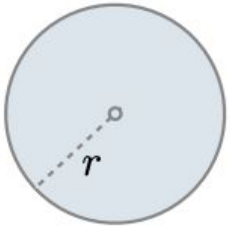
DRY (Don't Repeat Yourself) - Implement a process once, use it many times

Example: Area formulas

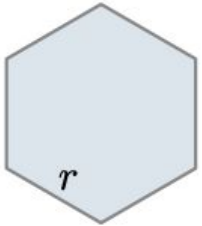
Many shapes have really similar formulas to calculate their areas:



$1 * r^2$



$\pi * r^2$



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

We can exploit patterns like this when we write functions!

Approach #1

```
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
```

Approach #2

```
from math import pi, sqrt

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

```
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)
```

Break

Higher-Order Functions

First Class Objects

- Functions are considered “First Class Objects” in Python
  - They can be manipulated in the same way as other objects, such as numbers or booleans
- A first class object is an entity that can be dynamically created, destroyed, passed to a function, returned as a value, and have all the rights as other variables in the programming language have
- Higher-Order Functions take advantage of the fact that functions are First Class Objects in Python

Higher-Order Functions

- **Definition**: A Higher-Order Function either:
  a. Takes in a function as an input
  b. Returns a function as an output
- Some Higher-Order Functions do both
- Higher-Order Functions allow us to expand on generalization in our programs

Generalizing over Computational Processes

- The common structure among functions may be a computational process, rather than a number

Example: Summations

$$\sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^{5} \frac{8}{(4k-3)\cdot(4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

Summations - Attempt #1

```
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
```

```
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
```

Summations - Attempt #2

```
def summation(n, term):
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

def natural(n):
    return n

def cube(n):
    return n
```

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

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

Example: make_adder

- We've seen how we can use the built-in function in Python, `add`, to add two numbers together
- What if we want an `add` function that always adds the same number to an argument?

make_adder

```
def make_adder(x):
    def adder(y):
        return x + y
    return adder
```

```
>>> three_adder = make_adder(3)
>>> three_adder(4) # 3 + 4
7
>>> three_adder(5) # 3 + 5
8
```

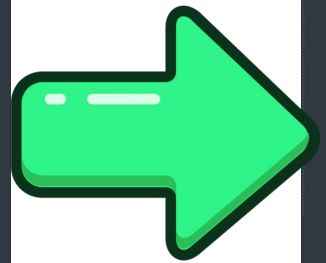
Currying

- As we've seen in the previous example, we can use higher-order functions to convert a function that takes multiple arguments into a chain of functions that each take a single argument
- In general, we can turn:
  - f(x, y) into g(x)(y)
  - f(x, y, z) into g(x)(y)(z)
  - etc…
- This transformation is called **currying**

Non-curried vs curried

```
def mul_three(x, y, z):
    return x * y * z

mul_three(1, 2, 3)
```



```
def f(x):
    def g(y):
        def h(z):
            return x * y * z
        return h
    return g

f(1)(2)(3)
```

Why do we curry?

- Allows you to “fix” a number of arguments while varying others
  - In `three_adder`, we fixed an argument to always be 3
- Some higher-order functions take in a function that takes exactly one argument
  - In `summation`, `summation` takes a `term` function that must take exactly one argument
- Some programming languages, such as Haskell, only allow functions that take a single argument, so the programmer must curry all multi-argument procedures

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>`

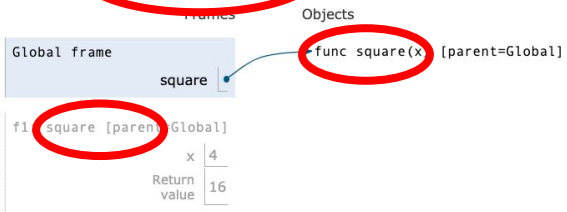
Here's an example of how we could rewrite the `square` function using a lambda expression:

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

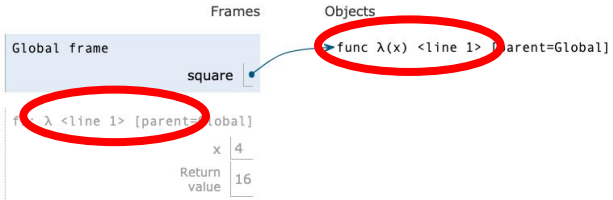
^^^This binds the name `square` to a function that takes in `x` as a parameter and returns `x * x`

Lambda vs Def

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




```
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

Summary

- When designing functions, we want to be as general as possible
  - Give a function exactly one job, but have it apply to many related situations
  - DRY principle: Don’t Repeat Yourself
- Higher-Order Functions are defined as functions that:
  - Take a function as input
  - Return a function as output
- Higher-Order Functions allow us to generalize over computational processes
  - Ex: `summation`, `make_adder`
- Currying transforms multi-argument functions into multiple single argument functions
- Lambda functions are functions defined in one line