Higher-Order Functions

Adapted from Materials from UC Berkeley CS61A course

Instructor: Paruj Ratanaworabhan

Designing Functions

```
def square(x):
    """Return X * X."""
```

- A function's **domain** is the set of all inputs it might possibly take as arguments

x is a number

- A function's **range** is the set of output values it might possibly return

square returns a non-negative real number

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

square returns the square of x

Designing Functions

- Give each function exactly one job, but make it apply to many related situations

```
>>> round(1.23) >>> round(1.23, 1) >>> round(1.23, 0) >>> round(1.23, 5)
    1                1.2                 1                   1.23
```

- Don't repeat yourself (DRY): Implement a process just once, but execute it many times

Generalization: Squaring Numbers

```
def square_1():
    return 1 * 1

def square_2():
    return 2 * 2

def square_3():
    return 3 * 3

...

def square_1024():
    return 1024 * 1024
```

There has to be a better way!

Generalization: Squaring Numbers

```
def square_1():
    return 1 * 1

def square_2():
    return 2 * 2

def square_3():
    return 3 * 3

...

def square_1024():
    return 1024 * 1024
```

DRY:
don't repeat yourself

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

Can You Make This Code DRY?

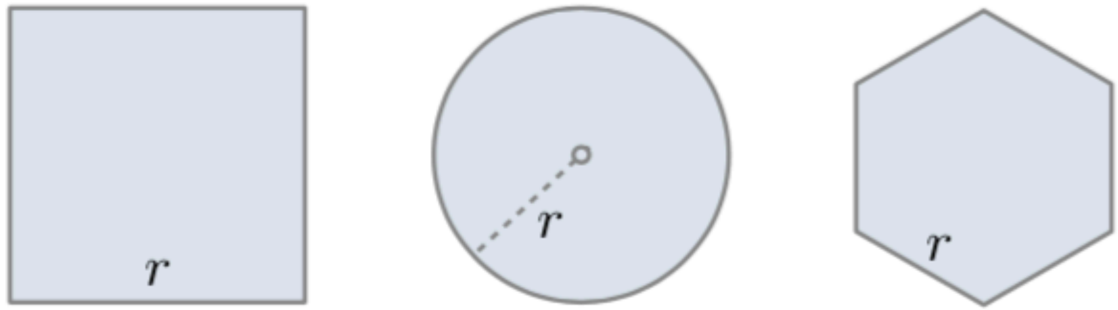
dry1.py

```
def same_length(a, b):
    """Return whether positive integers a and b have the same number of digits."""

    a_digits = 0
    while a > 0:
        a = a // 10
        a_digits = a_digits + 1
    b_digits = 0
    while b > 0:
        b = b // 10
        b_digits = b_digits + 1
    return a_digits == b_digits

print(same_length(50, 70))
print(same_length(50, 100))
print(same_length(10000, 12345))
```

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:



Area: $\boxed{1} \cdot r^2$ $\quad \boxed{\pi} \cdot r^2$ $\quad \boxed{\frac{3\sqrt{3}}{2}} \cdot r^2$

Finding common structure allows for shared implementation

Can You Make This Code DRY?

dry2.py

```
def same_length(a, b):
from math import pi, sqrt

def area_square(r):
    """Return the area of a square with side length R."""
    assert r > 0, 'A length must be positive'
    return r * r

def area_circle(r):
    """Return the area of a circle with radius R."""
    assert r > 0, 'A length must be positive'
    return r * r * pi

def area_hexagon(r):
    """Return the area of a regular hexagon with side length R."""
    assert r > 0, 'A length must be positive'
    return r * r * 3 * sqrt(3) / 2

print(area_square(8))
print(area_circle(8))
print(area_hexagon(8))
print(area_circle(-8))
```

Generalizing Over Computational Processes

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

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

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

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

Not So DRY Code

```
def sum_naturals(n):
    """Sum the first N natural numbers.
    """
    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.
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + pow(k, 3), k + 1
    return total

print(sum_naturals(5))
print(sum_cubes(5))
```

Using Higher-Order Functions

Making the code more DRY

```
def identity(k):
    return k

def cube(k):
    return pow(k, 3)

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

def sum_naturals(n):
    """Sum the first N natural numbers.
    """
    return summation(n, identity)

def sum_cubes(n):
    """Sum the first N cubes of natural numbers.
    """
    return summation(n, cube)

print(sum_naturals(5))
print(sum_cubes(5))
```

The Summation Example

```
def cube(k):
    return pow(k, 3)

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
    return total
```

Function of a single argument (*not called "term"*)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

0 + 1 + 8 + 27 + 64 + 125

The function bound to term gets called here

Exercise

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

Given that the above summation gives a closer approximation to the Pi value when the number of terms increases. Write a Python code based on the one using higher-order functions to print out an approximate Pi value for 1 000 000 terms.

Higher-Order Function

A function that:

- Takes functions as arguments
- Returns functions

Functions as Return Values

Functions defined within other function bodies are bound to names in a local frame

A function that returns a function

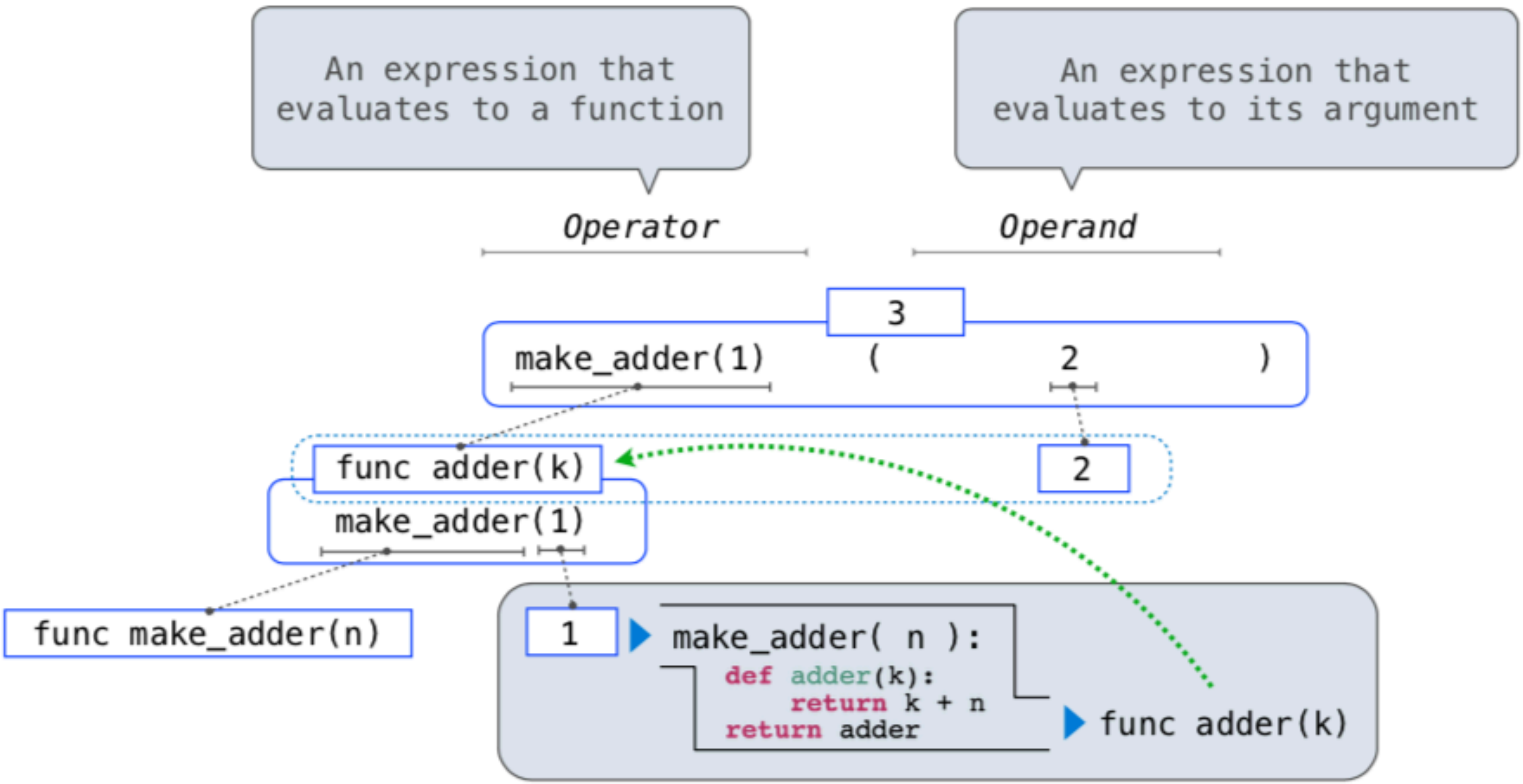
```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.

    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

The name add_three is bound to a function

A def statement within another def statement

Can refer to names in the enclosing function

Call Expressions as Operator Expressions



Why is This Useful?

Function composition

- Once many simple functions are defined, function composition is a natural method of combination to include in our programming language

- That is, given two functions f(x) and g(x), we might want to define h(x) = f(g(x))

```
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h
```

Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values 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

Higher-order functions:

- Express general methods of computation
- Remove repetition from programs
- Separate concerns among functions

Lambda Expressions

```
>>> x = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

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

Important: No "return" keyword!

A function
with formal parameter x
that returns the value of "x * x"

Must be a single expression

```
>>> square(4)
16
```

Lambda expressions are not common in Python, but important in general

Lambda expressions in Python cannot contain statements at all!

Lambda Expressions

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

VS

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

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).

Why Use Lambda Expressions?

- Lambda expressions are used when you need a function for a short period of time
- This is commonly used when you want to pass a function as an argument or return it in cases for higher-order functions

More Example

```
def search(f):
    """Return the smallest non-negative integer x for which f(x) is a true value."""
    x = 0
    while True:
        if f(x):
            return x
        x += 1

def square(x):
    return x * x

def inverse(f):
    """Return a function g(y) that returns x such that f(x) == y.
    """
    return lambda y: search(lambda x: f(x) == y)
```

What We Have Learned?

- Good code must be DRY
- Higher-order functions
  - take functions as arguments or return functions
- Lambda expressions
- Higher-order functions and Lambda expressions allow us to make DRY code