Welcome to Data C88C!

Lecture 04: Higher-Order Functions
Thursday, June 26th, 2025
Week 1
Summer 2025
Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

- Lab01 + Lab02 released!
  - Due: Tues July 1st, 11:59 PM PST
- HW02 released!
  - Due: Tues July 1st, 11:59 PM PST
- Lab00
  - Due: Sun June 29th, 11:59 PM PST
- HW01
  - Due: Sun June 29th, 11:59 PM PST
- **Reminder**: watch YouTube video BEFORE lecture!
  - See course website for video link

Important: watch these videos before lecture to maximize learning!

Calendar

Week	Date	Lecture	Textbook	Lab & Discussion Links	Homework & Project
	Mon 6/23	Welcome		Disc 00: Getting Started	
	T[illegible]	Functions Videos	Ch. 1.1 Ch. 1.2 Ch. 1.3	Disc 01: Functions Lab 00: Getting Started Due Sun 6/29	HW 01: Functions Due Sun 6/29
1	Wed 6/25	Control Videos	Ch. 1.4 Ch. 1.5	Disc 02: Control, Environment Diagrams	

Lecture Overview

- Designing functions
  - Case study: `same_length()`
- Higher-order functions
- Case studies:
  - Summation
  - Twenty-one game ("Nim")

Designing Functions

Describing 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

A Guide to Designing Function

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

```
>>> round(1.23)
1
```

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

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

```
>>> round(1.23, 5)
1.23
```

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

Case study: same_length()

`same_length(a, b)` should return True if integers `a, b` contain the same number of digits:

```
>>> same_length(50, 70)
True
>>> same_length(50, 100)
False
>>> same_length(1000, 100000)
False
```

Question: what should our approach to coding this up be?

Answer: here's one way:

(1) Count the number of digits of the first input
(2) Count the number of digits of the second input
(3) Return True if the number of digits is the same, False otherwise.

Question: how to count the number of digits of an integer?

Answer: use the `%, //` operators with a while-loop

```
# Use % to fetch the right-most digit
>>> 125 % 10
5
# Use // to discard the right-most digit
>>> 125 // 10
12
```

Question: are there any opportunities to define any useful helper functions here?

Answer: (1) and (2) do the same exact thing, let's define a helper function that counts number of digits!

(Demo: 04.py:Demo00)

Higher-Order Functions

Definition: Higher-order function

- A **higher-order function** ("HOF") is a function that either accepts a function(s) as **input**
  - OR: returns a function(s) as an **output**
- Not all programming languages support HOF's
  - Languages that do are said to treat functions as "first class objects"

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

def call_fn(fn, x):
    return fn(x)

>>> call_fn(square, 2)
4
```

`call_fn()` accepts a function `fn` as input!

```
def create_square_fn():
    return square

>>> square_fn = create_square_fn()
>>> square_fn(3)
9
```

`create_square_fn()` returns a function as output!

Case study: summation + HOF's

- **Recall**: the summation function applies a function `f(x)` to a sequence of integers (1, 2, ..., N), and returns the sums of all the values

		f(x)
$\sum_{k=1}^{5} k = 1+2+3+4+5$	$= 15$	Identity function
$\sum_{k=1}^{5} k^3 = 1^3+2^3+3^3+4^3+5^3$	$= 225$	Cubed function
$\sum_{k=1}^{5} \frac{8}{(4k-3)\cdot(4k-1)} = \frac{8}{3}+\frac{8}{35}+\frac{8}{99}+\frac{8}{195}+\frac{8}{323}$	$= 3.04$	(some math expression)

Case study: summation + HOF's

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

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

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

One way to express these as Python functions is to define a separate function for each.

Downside: lots of repeated code. There's an abstraction opportunity here!

```
def summation_a_identity(k_end):
    k = 1
    out_sum = 0
    while k <= k_end:
        out_sum += k
        k += 1
    return out_sum

def summation_b_cubed(k_end):
    k = 1
    out_sum = 0
    while k <= k_end:
        out_sum += k ** 3
        k += 1
    return out_sum

def summation_c_mystery_math(k_end):
    k = 1
    out_sum = 0
    while k <= k_end:
        out_sum += (8 / ((4 * k - 3) * (4 * k - 1)))
        k += 1
    return out_sum
```

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

Program Design

Modularity

Abstraction

Separation of Concerns

Twenty-One (aka "Nim") Rules

Two players alternate turns, on which they can add 1, 2, or 3 to the current total

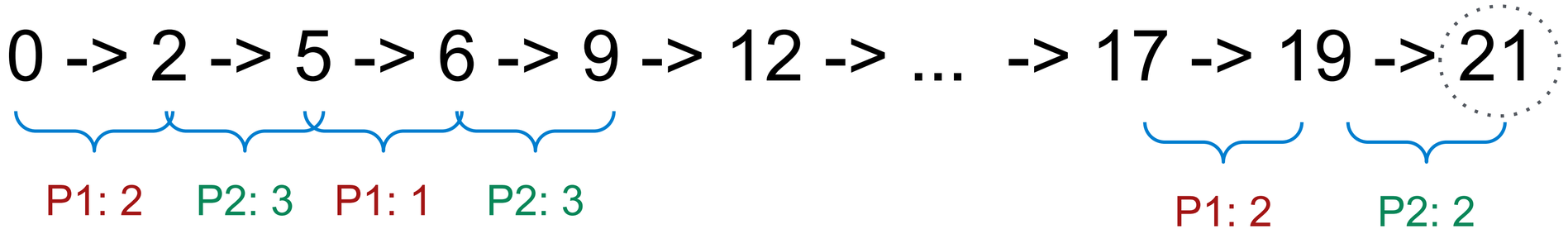
The total starts at 0

The game end whenever the total is 21 or more

The last player to add to the total loses

P2 wins!

Example game:



Today, we'll explore an implementation of this game that uses HOF's to represent strategies!

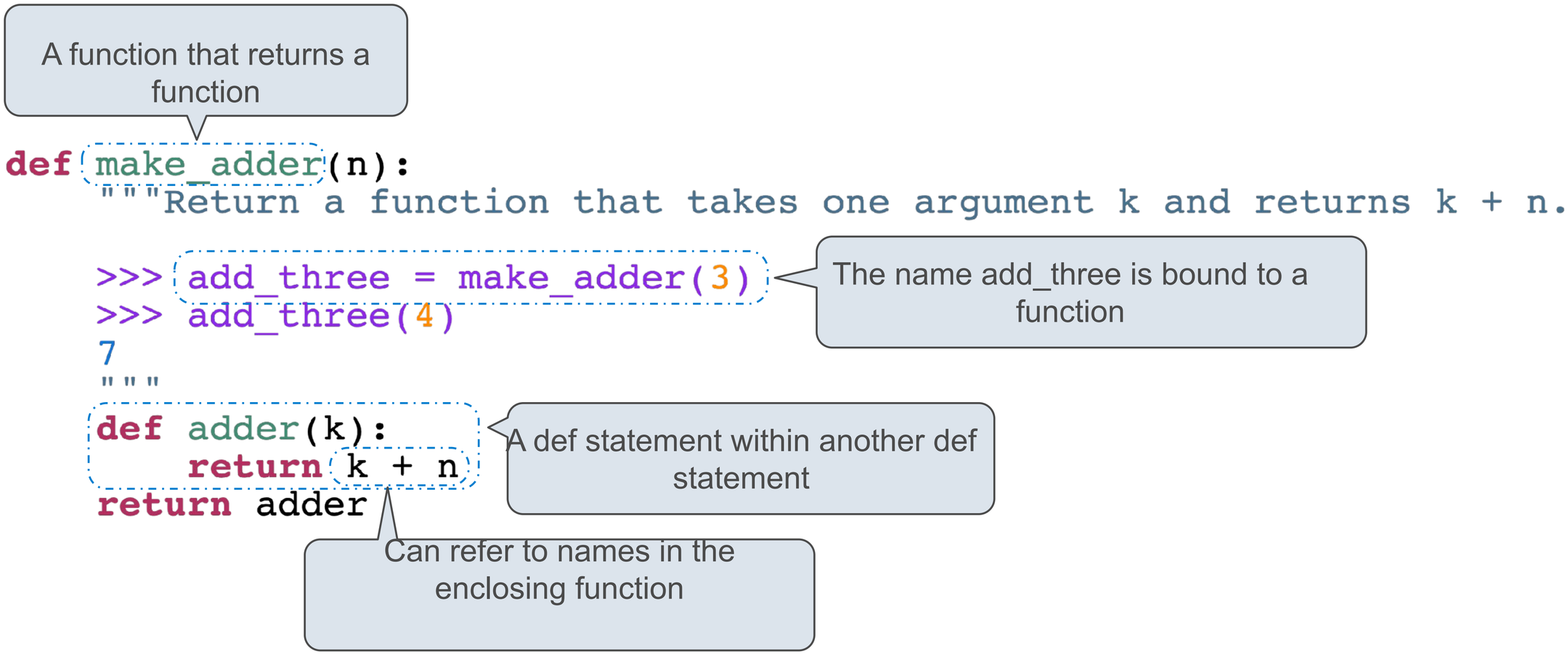
(Demo: 04.py:Demo01)

Functions as Return Values

(Demo: 04.py:Demo02)

Locally Defined Functions

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



Twenty-One Strategies

(Demo: 04.py:Demo03)