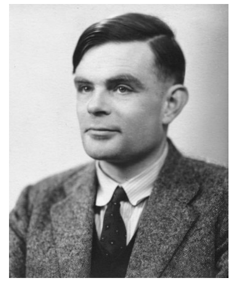
CS61A Lecture 5
Applications of Higher Order Functions

Jon Kotker and Tom Magrino
UC Berkeley EECS
June 25, 2012

Computer Science in the News
Happy 100th Birthday, Alan Turing!



http://upload.wikimedia.org/wikipedia/en/c/c8/Alan_Turing_photo.jpg

- Born on June 23, 1912.
- Father of computer science and artificial intelligence.
- Described hypothetical computational machines, now called *Turing machines*, before the first mechanical computer existed!
- Helped break ciphers during World War II.
- Also contributed to mathematical biology.

Today

- Practice with higher order functions and anonymous functions, and
- Applications of higher order functions:
  - Project 1 demonstration.
  - Iterative improvement.

Recap: Higher Order Functions

Higher order functions are functions that can either ***take functions as arguments*** or ***return a function***.

Aside: First-Class Citizens

In a programming language, an entity is a ***first-class citizen*** if:

1. It can be named by variables.
2. It can be passed as arguments to functions.
3. It can be returned from functions.
4. It can be included in data structures.

(We will see a few data structures in module 2.)

In Python, data and functions are *both* first-class citizens.

This may not be true in other programming languages.

λ Review: Anonymous Functions

Lambda expressions and defined functions

```
<name> = lambda <arguments>: <expression>

def <name>(<arguments>):
    return <expression>
```

λ Review: Anonymous Functions

Lambda expressions and defined functions

```
square = lambda x: x * x

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

7

Review: Anonymous Functions

What will the following expression return?

`(lambda x: x*5)(3+7)`

8

Review: Anonymous Functions

What will the following expression return?

`(lambda x: x*5)(3+7)`

50

9

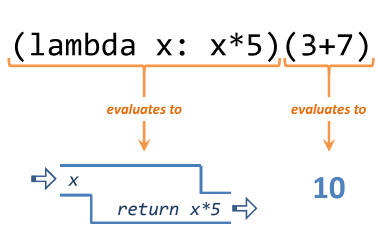
Review: Applicative Order

To evaluate a compound expression:
Evaluate the operator and then the operands, and
Apply the operator on the operands.

This is the ***applicative*** order of evaluation.

10

Applicative Order: Evaluate



11

Applicative Order: Apply

12

Review: Anonymous Functions

What will the following expression return?

```
(lambda x: x*5)((lambda y: y+5)(3))
```


Review: Anonymous Functions

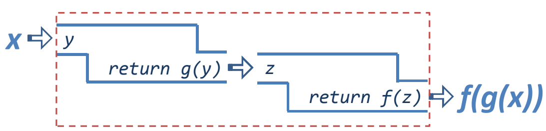
What will the following expression return?

```
(lambda x: x*5)((lambda y: y+5)(3))
```

40

Anonymous and Higher Order Functions

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



compose(f, g)

Anonymous and Higher Order Functions

```
increment = lambda num: num+1
square = lambda num: num*num
identity = lambda num: num
```

Which of the following expressions are valid, and if so, what do they evaluate to?

```
compose(increment, square)
compose(increment, square)(2)
compose(square, increment)(2)
compose(square, square(2))(3)
```


Anonymous and Higher Order Functions

```
increment = lambda num: num+1
square = lambda num: num*num
identity = lambda num: num
```

Which of the following expressions are valid, and if so, what do they evaluate to?

```
compose(increment, square)
Function that squares a number and then adds 1.
compose(increment, square)(2)
5
compose(square, increment)(2)
9
compose(square, square(2))(3)
Error
```


Anonymous and Higher Order Functions

```
twice = lambda f: compose(f, f)
```

Which of the following expressions are valid, and if so, what do they evaluate to?

```
twice(increment)(1)
twice(twice(increment))(1)
twice(square(3))(1)
twice(lambda: square(3))(1)
```

Anonymous and Higher Order Functions

```
twice = lambda f: compose(f, f)
```

Which of the following expressions are valid, and if so, what do they evaluate to?

```
twice(increment)(1)
3
twice(twice(increment))(1)
5
twice(square(3))(1)
Error
twice(lambda: square(3))(1)
Error
```

19

Anonymous and Higher Order Functions

```
def twice(f):
    return lambda x: f(f(x))
```

Which of the following expressions are valid, and if so, what do they evaluate to?

```
twice(lambda x: square(3))(1)
twice(identity)()
(twice(twice))(increment)(1)
```

20

Anonymous and Higher Order Functions

```
def twice(f):
    return lambda x: f(f(x))
```

Which of the following expressions are valid, and if so, what do they evaluate to?

```
twice(lambda x: square(3))(1)
9
twice(identity)()
Error
(twice(twice))(increment)(1)
5
```

21

Anonymous and Higher Order Functions

```
def twice(f):
    return lambda x: f(f(x))
```

Which of the following expressions are valid, and if so, what do they evaluate to?

```
twice(twice(twice))(increment)(1)
(twice(twice))(twice(increment))(1)
```

22

Anonymous and Higher Order Functions

```
def twice(f):
    return lambda x: f(f(x))
```

Which of the following expressions are valid, and if so, what do they evaluate to?

```
twice(twice(twice))(increment)(1)
17
(twice(twice))(twice(increment))(1)
9
```

23

Announcements

- Homework 3 is released and due **June 29**.
- Project 1 is also due **June 29**.
- The homework 0 for the staff will be available tonight.

24

PROJECT 1: THE GAME OF PIG

Dice game described by John Scarne in 1945.

Number of players: Two.

Goal: To reach a score of 100.

Played with: Six-sided die and four-sided die.

Rating: ★★★★½

http://www.amazon.com/Winning-Moves-1046-Pass-Pigs/dp/B00005JG3Y

Image from http://en.wikipedia.org/wiki/File:Pass_pigs_dice.jpg

25

PROJECT 1: THE GAME OF PIG

GAMEPLAY

One player keeps **rolling** a die, remembering the sum of all rolls (the **turn total**), *until*:

1. Player **holds**, adding the turn total (now the **turn score**) to the total score, *or*
2. Player **rolls a 1**, adding only 1 (the turn score) to the total score.

26

PROJECT 1: THE GAME OF PIG

RISK

Player can either pig out and *keep rolling*, or *hold* and keep the turn total.

27

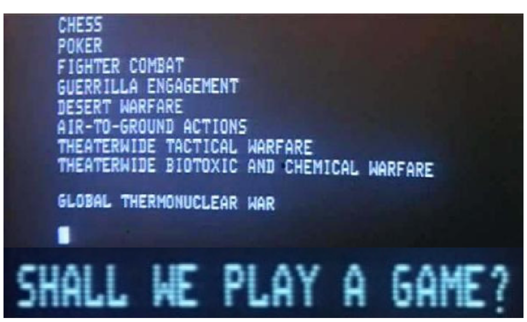
PROJECT 1: THE GAME OF PIG

DIE RULE

At the beginning of a player's turn, if the sum of the two scores is *a multiple of 7*, the player uses the *four*-sided die, not the *six*-sided die.

28

PROJECT 1: THE GAME OF PIG



http://cdn0.hark.com/images/000/001/157/1157/original.jpg
http://www.lazylaces.com/pics/01/center_shall_we_play_a_game.jpg

29

PROJECT 1: THE GAME OF PIG

HIGHER ORDER FUNCTIONS

Every player has a ***strategy***, or a game plan.

A strategy determines the player's ***tactics***.

A tactic supplies an ***action*** on each roll.

A player can either *keep rolling* or *hold*.

30

Project 1: The Game of Pig
Higher Order Functions

What is a ***strategy***?

A *higher order function* that uses the *player's current score* and the *opponent's score* to return a *tactic* for a particular turn.

What is a ***tactic***?

A *higher order function* that uses the *turn total* to determine the next *action*.

31

Project 1: The Game of Pig
Higher Order Functions

What is an ***action***?

Two *functions* (**roll** and **hold**) that use the *turn total* and the *die roll* to calculate:

the turn score, the new turn total, and if the player's turn is over.

32

Project 1: The Game of Pig
Phases

Phase 1: *Simulator* – Simulate gameplay between two players.

Phase 2: *Strategies* – Test a family of strategies and devise your own strategy.

33

Project 1: The Game of Pig
Tips

- Keep track of the *domain and range* of your functions.
- Remember: a *strategy* returns a *tactic*, which returns an *action*, which returns useful values.
- Start early.
- DBC: Ask questions!

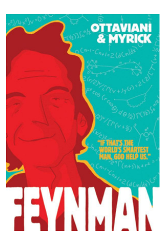
34

Problem Solving
The Richard Feynman Approach

Step 1: Write down the problem.

Step 2: Think real hard.

Step 3: Write down the solution.



(suggested by Murray Gell-Mann, a colleague of Feynman's)

35

Problem Solving
Iterative Improvement

Iteration refers to repeating something to reach a goal.

Step 1: Write down the problem.

Step 2: ***Guess*** an answer to the problem.

Step 3: If the guess is ***approximately correct***, it is the solution. Otherwise, ***update*** the guess, go back to step 2 and *repeat*.

36

Problem Solving
Iterative Improvement

```
def iter_improve(update, isclose, guess=1):
    while not isclose(guess):
        guess = update(guess)
    return guess
```

isclose and update must be functions!

iter_improve is thus a higher order function.

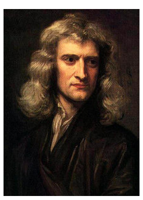
Problem Solving
Iterative Improvement

```
def iter_improve(update, isclose, guess=1):
    while not isclose(guess):
        guess = update(guess)
    return guess
```

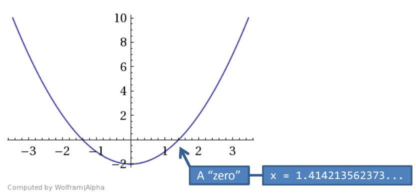
Need to choose the initial guess, the check for approximate correctness, and the update function.

Newton's Method

Used to find the roots (or *zeros*) of a function *f*, where the function evaluates to zero.



http://en.wikipedia.org/wiki/File:GodfreyKneller-IsaacNewton-1689.jpg



$f(x) = x^2 - 2$

Newton's Method

Many mathematical problems are equivalent to finding roots of specific functions.

"Equivalent to"

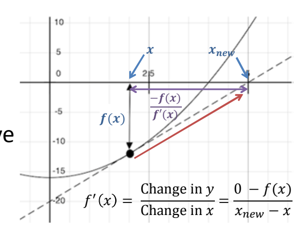
Square root of 2 is x, $x^2 = 2 \equiv$ Root of $x^2 - 2$

Power of 2 that is 1024 $\equiv$ Root of $2^x - 1024$

Number x that is one less than its square, or $x = x^2 - 1 \equiv$ Root of $x^2 - x - 1$

Newton's Method

1. Start with a function f and a guess x.
2. Compute the value of the function f at x.
3. Compute the derivative of f at x, $f'(x)$.
4. Update guess to be $x - \frac{f(x)}{f'(x)}$.



Animation: http://en.wikipedia.org/wiki/File:NewtonIteration_Ani.gif

Newton's Method

```
def approx_deriv(f, x, dx=0.000001):
    return (f(x + dx) - f(x))/dx

def approx_eq(x, y, tolerance=1e-5):
    return abs(x - y) < tolerance

def approx_zero(x):
    return approx_eq(x, 0)
```

$1 \times 10^{-5} = 0.00001$

Newton's Method

```
def newton_update(f):
    return lambda x: x - f(x)/approx_deriv(f, x)
```

We do not need to make a new update function for every function, thanks to higher order functions.

Generalization is a powerful theme.

Newton's Method

```
def iter_improve(update, isclose, guess=1):
    while not isclose(guess):
        guess = update(guess)
    return guess

def find_root(f, initial_guess=10):
    return iter_improve(newton_update(f),
                        lambda x: approx_zero(f(x)),
                        initial_guess)
```

Why must this be a function?

Newton's Method

Square root of 2 is x, $x^2 = 2$ ≡ Root of $x^2 - 2$

`find_root(lambda x:x**2 - 2)`

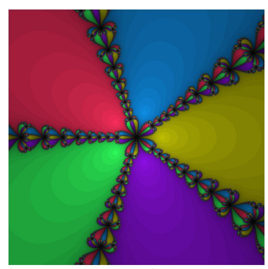
Power of 2 that is 1024 ≡ Root of $2^x - 1024$

`find_root(lambda x:2**x - 1024)`

Number x that is one less than its square, or $x = x^2 - 1$ ≡ Root of $x^2 - x - 1$

`find_root(lambda x:x**2 - x - 1)`

Aside: Newton Fractal



$f(x) = x^5 - 1$

http://upload.wikimedia.org/wikipedia/commons/b/bf/Newtroot_1_0_0_0_0_m1.png

Each colored region refers to one root, and the initial guesses that will eventually *converge* to that root.

Newton's Method

Incredibly powerful, but does not always work!

Certain conditions need to be satisfied: for example, the function needs to be differentiable.

The method can fail in many ways, including:

1. Infinite loop among a set of guesses. (Try $f(x) = x^3 - 2x + 2$.)
2. Guesses may never fall within the tolerance for approximate equality.
3. Guesses converge to the answer *very slowly*.

Problem Solving

Iterative Improvement (with one fix)

We can add a limit on the number of *iterations*.

```
def iter_improve(update, isclose, guess=1, max_iter=5000):
    counter = 1
    while not isclose(guess) and counter <= max_iter:
        guess = update(guess)
        counter += 1
    return guess
```

Problem Solving
Iterative Improvement (Other Examples)

Finding the square root of a number a
(*Babylonian method or Heron's method*)

Update: $x \to {}^{1}/_{2}\left(x + \frac{a}{x}\right)$
Implementation:
Write `my_sqrt(a)` using `iter_improve`.
(What function should we use as the `update` function?)

49

Problem Solving
Iterative Improvement (Other Examples)

Software development

Write a first draft of code and tests that prove the code works as specified. Update
Then, consider making the code more efficient, each time ensuring the tests pass.
Check for correctness

50

Conclusion

- *Iterative improvement* is a general problem-solving strategy, where a guess at the answer is successively improved towards the solution.
- Higher order functions allow us to express the *update* and the *check for correctness* within the framework of this strategy.
- ***Preview***: Can a function call itself?

51

Extras: Asterisk Notation

Variadic functions can take a variable number of arguments.

```
>>> def fn(*args):
...     return args
>>> foo = fn(3, 4, 5)
>>> foo
(3, 4, 5)
>>> a, b, c = foo
>>> a
3
>>> bar = fn(3, 4)
>>> bar
(3, 4)
```

Here, the asterisk signifies that the function can take a *variable* number of arguments.

The arguments provided are stored in the variable args.

52

Extras: Asterisk Notation

```
>>> def bar(a, b, c):
...     print a, b, c
>>> args = 2, 3, 4
>>> bar(*args)
2 3 4
```

Here, the asterisk "expands" the values in the variable args to "fill up" the arguments to bar.

53

Extras: Currying

`(lambda x: lambda y: (x*5)+y)(3)(4)`
is equivalent to
`(lambda x, y: (x*5)+y)(3, 4)`

Currying allows us to represent multiple variable functions with single-variable functions.
It is named after Haskell Curry, who rediscovered it after Moses Schönfinkel.
Intuition: When evaluating a function from left to right, we are temporarily making several functions with fewer arguments along the way.

54