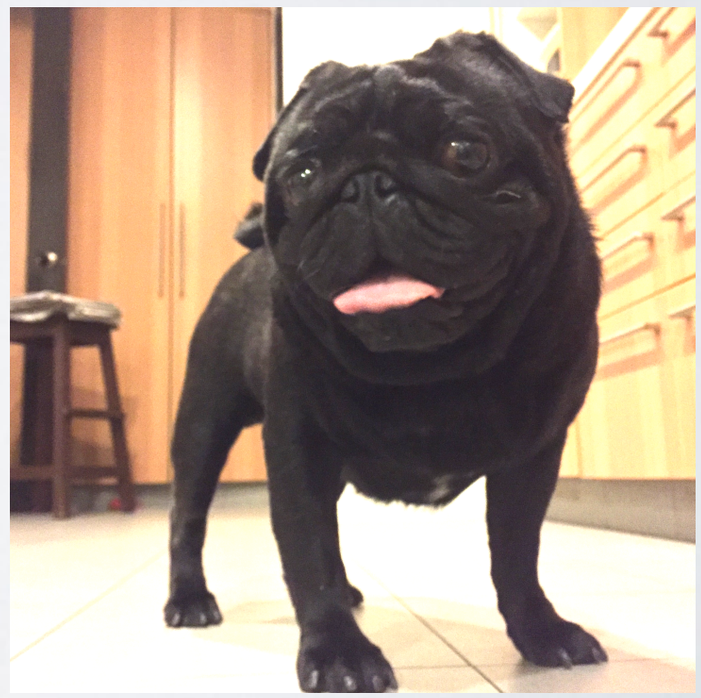
# Introduction to Concepts in Functional Programming

CS16: Introduction to Data Structures & Algorithms

Spring 2019

# Outline

- Functions
- State
- Functions as building blocks
- Higher order functions
  - Map
  - Reduce



# Functional Programming Paradigm

- A style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions.
- Programs written in this paradigm rely on smaller methods that do one part of a larger task. The results of these methods are combined using function compositions to accomplish the overall task.

# What are functions ?

- Takes in an input and always returns an output
- A given input maps to exactly one output
- Examples:
  - In math: `f(x) = x + 1`
  - In Python:

```
def f(x):
    return x + 1
```

# What is State?

- All stored information to which a program has access to at any given point in time
- How can a program's state change ?
  - ***Mutable data*** is data that can be changed after creation. Mutating data changes program state.
    - Mutator methods (i.e. setters…)
    - Loop constructs that mutate local variables
  - ***Immutable data*** is data that cannot be changed after creation. In a language with only immutable data, the program state can never change.

# State changes

- In a ***stateful*** program, the same method could behave differently depending upon the ***state*** of the program when it was called
- Let's look at an example of a ***stateful*** program.
- Our example is a short program that simulates driving behavior based on the color of the stoplight.

```
light_color = “RED”

def change_light():
  if light_color == “RED”:
    light_color = “GREEN”
  elif light_color == “GREEN”:
    light_color == “YELLOW”
  elif light_color == “YELLOW”:
    light_color = “RED”

def drive(car):
  if light_color == “RED”:
    car.stop()
  elif light_color == “YELLOW”:
    car.slow_down()
  elif light_color == “GREEN”:
    car.move_forward()
```

# State in Functional Programs

- In pure functional languages, all data is ***immutable*** and the program state cannot change.
- What are the implications of this property ?
  - Functions are ***deterministic*** i.e. the same input will always yield the same output. This makes it easier to re-use functions elsewhere.
- The order of execution of multiple functions does not affect the final outcome of the program.
- Programs don't contain any ***side effects.***

# Mutable vs Immutable State

- Consider these two programs

| **Program 1** | **Program 2** |
|---|---|
| `a = f(x)`<br>`b = g(y)`<br>`return h(a,b)` | `b = g(y)`<br>`a = f(x)`<br>`return h(a,b)` |

- Will they return the same result if the state is mutable ?
- What about when the state is immutable ?

# Mutable vs Immutable State

| Program 1 | Program 2 |
|---|---|
| `a = f(x)`<br>`b = g(y)`<br>`return h(a,b)` | `b = g(y)`<br>`a = f(x)`<br>`return h(a,b)` |

- ***Mutable State:*** Not guaranteed to give the same results because:
  - The first function call might have changed state.
  - Thus, the second function call might behave differently.
  - This is an example of a ***side effect.***
- ***Immutable State:*** Guaranteed to output the same result for the same inputs!

# State and Loops

```
for i = 0; i < len(L); i++:
    print L[i]
```

The local variable **i** is being mutated!

- If we can't mutate state - we can't use our usual **for** and **while** loop constructs!
- Instead, functional languages make use of recursion

# State and Loops (cont'd)

- What variables are being mutated in this example ?

```
def max(L):
    max_val = -infinity
    for i in range(0, len(L)):
        if L[i] > max_val:
            max_val = L[i]
    return max_val
```

*30 seconds*

# State and Loops (cont'd)

- What variables are being mutated in this example ?

```
def max(L):
    max_val = -infinity
    for i in range(0, len(L):
        if L[i] > max_val:
            max_val = L[i]
    return max_val
```

**i** is being mutated!

**max_val** is being mutated!

- How do we write this function without mutation … ?

# Replacing Iteration with Recursion

**Iterative Max**

```
def max(L):
  max_val = -infinity
  for i in range(0, len(L)):
    if L[i] > max_val:
      max_val = L[i]
  return max_val
```

**Recursive Max**

```
def max(L):
  return max_helper(L, -infinity)

def max_helper(L, max_val):
  if len(L) == 0:
    return max_val
  if L[0] > max_val:
    return max_helper(L[1:], L[0])
  else:
    return max_helper(L[1:], max_val)
```

The recursive version would work in a pure functional language, the iterative version would not.

# Replacing Iteration with Recursion

**Recursive Max**

```
def max(L):
  return max_helper(L, -infinity)


def max_helper(L, max_val):
  if len(L) == 0:
    return max_val
  if L[0] > max_val:
    return max_helper(L[1:], L[0])
  else:
    return max_helper(L[1:], max_val)
```

Example: [3, 1, 2]

```
max([3,1,2])
↓
max_helper([3,1,2], -inf)
↓
max_helper([1,2], 3)
↓
max_helper([2], 3)
↓
max_helper([], 3)
↓
3
```

# First Class Functions

- In the functional paradigm, functions are treated as ***first-class citizens*** i.e. they can be:
  - Passed as arguments to other functions
  - Returning them as values from other functions
  - Storing them in variables just like other data-types

```
def add_one(x):
   return x + 1

def apply_func_to_five(f):
   return f(5)

print apply_func_to_five(add_one)
>>> 6
```

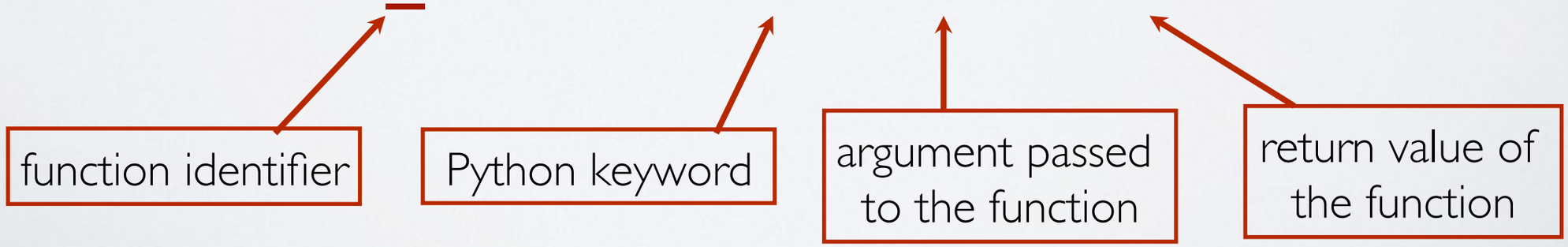
# First Class Functions (cont'd)

- What's actually happening in our definition of the **add_one** function ?

```
def add_one(x):
    return x + 1
```

- We're binding a function to the identifier **add_one**
- In Python, this is equivalent to

```
add_one = lambda x: x + 1
```



# Anonymous Functions

- Data types such as numbers, strings, booleans etc. don't need to be bound to a variable. Similarly, neither do functions!
- An ***anonymous function*** is a function that is not bound to an identifier.
- A Python example of an anonymous function is **lambda x: x + 1**
- An example of a function that returns an anonymous function:

```
# Input: A number k
# Output: A function that increments k by the number passed into it
def increment_by(k):
   return lambda x: x + k


add_three = increment_by(3)
```

# Function Syntax Overview

| Math | Bound Python Function | Anonymous Python Function |
|---|---|---|
| $f(x) = x + 1$ | `def f(x):`<br>`  return x + 1`<br>**or**<br>`f = lambda x: x + 1` | `lambda x: x + 1` |

# Higher Order Functions

- A function is a ***higher-order function*** if it either takes in one or more functions as parameters and/or returns a function.
- You've already seen examples of higher-order functions in the previous slides!

```
print apply_func_to_five(add_one)
>>> 6
```

```
# Input: A number k
# Output: A function that increments k by the number passed into it
def increment_by(k):
   return lambda x: x + k
```

# Using Higher Order Functions

```
# Input: A number x
# Output: A function that adds the number passed in to x
def increment_by(k):
   return lambda x: x + k


# we pass in 1 as the value of 'k'
>>> add_one = increment_by(1)

# add_one holds the function object returned by calling increment_by
>>> print add_one
<function <lambda> at 0x123e410>

# '5' is the value of the parameter 'x' in the function
# add_one which is lambda x: x + 1
>>> print add_one(5)
6
```

# Map

- ***Map*** is a higher order function with the following specifications:
  - Inputs
    - `f` - a function that takes in an element
    - `L` - a list of elements
  - Output
    - A new list of elements, with `f` applied to each of the elements of `L`

# Map

- Map is roughly equivalent in functionality to this Python function:

```
def map(func, list):
   for element in list:
       element = func(element)
   return list
```

# Map

```
map(lambda x: x-2, [11,9,24,-5,34,4])
```

# Map

```
print map(lambda x: x-2, [11,9,24,-5,34,4])
>>> [9,7,22,-7,32,2]
```

# Reduce

- ***Reduce*** is also a higher-order function.
- It ***reduces*** a list of elements to one element using a binary function to successively combine the elements.
  - Inputs
    - `f` - a binary function
    - `L` - list of elements
    - **`acc`** - accumulator, the parameter that collects the return value
  - Output
    - The value of `f` sequentially applied and tracked in **`acc`**

# Reduce

- Reduce is roughly equivalent in functionality to this Python function:

```
def reduce(binary_func, list, acc):
    for element in list:
        acc = binary_func(acc, element)
    return acc
```

# Reduce Example

```
# binary function 'add'
add = lambda x, y: x + y

# use 'reduce' to sum a list of numbers
>>> print reduce(add, [1,2,3], 0)
6
```

binary function

collection of elements

accumulator

# Reduce Example

```
add = lambda x, y: x + y
reduce(add, [1,2,3], 0)
```

| Math | Python |
|---|---|
| ((0 + 1) + 2) + 3) = ? | reduce(add, [1,2,3], 0) = ? |
| current accumulator | current accumulator |

# Reduce Example (cont'd)

```
add = lambda x, y: x + y
reduce(add, [1,2,3], 0)
```

| Math | Python |
|---|---|
| ((0 + 1) + 2) + 3) = ? | reduce(add, [1,2,3], 0) = ? |
| ((1 + 2) + 3) = ? | reduce(add, [2,3], 1) = ? |
| current accumulator | current accumulator |

# Reduce Example (cont'd)

```
add = lambda x, y: x + y
reduce(add, [1,2,3], 0)
```

| Math | Python |
|---|---|
| ((0 + 1) + 2) + 3) = ? | reduce(add, [1,2,3], 0) = ? |
| ((1 + 2) + 3) = ? | reduce(add, [2,3], 1) = ? |
| (3 + 3) = ? | reduce(add, [3], 3) = ? |

current accumulator

current accumulator

# Reduce Example (cont'd)

```
add = lambda x, y: x + y
reduce(add, [1,2,3], 0)
```

| Math | Python |
|---|---|
| ((0 + 1) + 2) + 3) = ? | reduce(add, [1,2,3], 0) = ? |
| ((1 + 2) + 3) = ? | reduce(add, [2,3], 1) = ? |
| (3 + 3) = ? | reduce(add, [3], 3) = ? |
| 6 | 6 |

final accumulator/ return value

final accumulator/ return value

# Reduce Activity

```
multiply = lambda x, y: x*y
reduce(multiply, [1,2,3,4,5], 1)
```

*2 mins*

# Reduce Activity

```
multiply = lambda x, y: x*y
reduce(multiply, [1,2,3,4,5], 1)
```

| Math | Python |
|---|---|
| ((1*1)*2)*3)*4)*5) = ? | reduce(multiply, [1,2,3,4,5], 1) = ? |
| ((1*2)*3)*4)*5) = ? | reduce(multiply, [2,3,4,5], 1) = ? |
| ((2*3)*4)*5) = ? | reduce(multiply, [3,4,5], 2) = ? |
| ((6*4)*5) = ? | reduce(multiply, [4,5], 6) = ? |
| (24*5) = ? | reduce(multiply, [5], 24) = ? |
| 120 | 120 |

# Reduce

- The accumulator doesn't always have to be an integer and reduce doesn't have to necessarily reduce the list to a single number.
- Another example is removing consecutive duplicates from a list.
  - The accumulator is also a list!

```
def compress(acc, e):
   if acc[len(acc)-1] != e:
      return acc + [e]
   else:
      return acc

def remove_consecutive_dups(L):
   return reduce(L, compress, [L[0]])
```

# Using Higher Order Functions

- With higher-order functions, we can make our programs much more concise!
- Let's look at the recursive **max** function we defined earlier:

| Original Recursive Example | Revised with Higher Order Functions |
|---|---|
| `def max(L):`<br>`  return max_helper(L, -infinity)`<br><br>`def max_helper(L, max_val):`<br>`  if len(L) == 0:`<br>`  return max_val`<br>`  if L[0] > max_val:`<br>`  return max_helper(L[1:], L[0])`<br>`  return max_helper(L[1:], max_val)` | `def max_of_two(acc, e):`<br>`  if acc > e:`<br>`  return acc`<br>`  return e`<br><br>`def max(L):`<br>`  return reduce(max_of_two, L, -infinity)` |

# Building Functional Programs

- With the power to use higher-order functions and using functions as first-class citizens, we can now build programs in a functional style!
- A functional program can be thought of as one giant function composition, such as **f(g(h(x))).**

# Advantages and Disadvantages of Functional Programming

| Advantages | Disadvantages |
| --- | --- |
| • Programs are deterministic.<br>• Code is elegant and concise because of higher order function abstractions.<br>• It's easy to run code concurrently because state is immutable.<br>• Learning functional programming teaches you as a programmer to think about problems very differently than imperative programming which makes you a better problem-solver! | • Programs are deterministic.<br>• Potential performance losses because of the amount of garbage-collection that needs to happen when we end up creating new variables as we can't mutate existing ones.<br>• File I/O is difficult because it typically requires interaction with state.<br>• Programmers who're used to imperative programming could find this paradigm harder to grasp. |

# Functional Programming Practice

*6 mins*

# Problem # 1

Write an anonymous function that raises a single argument 'n' to the $n^{th}$ power

# Problem #1

Write an anonymous function that raises a single argument 'n' to the $n^{th}$ power

***Solution:***

```
lambda n: n**n
```

# Problem #2

Write a line of code that applies the function you wrote in problem #1 to every element in an input list.

# Problem #2

Write a line of code that applies the function you wrote in problem #1 to every element in an input list.

***Solution:***

```
map(lambda n: n**n, list)
```

# Problem #3

Write an anonymous function that takes in a single argument 'n' and returns a function that takes in no arguments and returns n.

# Problem #3

Write an anonymous function that takes in a single argument 'n' and returns a function that takes in no arguments and returns n.

***Solution:***

```
lambda n: lambda: n
```

# Problem #4

Write a line of code that applies the function you wrote in problem #3 to an input list. This should give you a list of functions. Write another line of code that takes in the list of functions and outputs the original list again.

# Problem #4

Write a line of code that applies the function you wrote in problem #3 to an input list. This should give you a list of functions. Write another line of code that takes in the list of functions and outputs the original list again.

***Solution:***

```
function_list = map(lambda n: lambda: n, input_list)
map(lambda f: f(), function_list)
```

# Problem #5

Remove odd numbers from an input list of numbers using **reduce**.

# Problem #5

Remove odd numbers from an input list of numbers using **reduce**.

***Solution:***

```
reduce(lambda acc,x: acc+[x] if x%2 == 0 else acc, L, [])
```

# More Higher Order Functions

- There are many more commonly-used higher order functions besides **map** and **reduce.**
- **`filter(f,x)`**
  - Returns a list of those elements in list **`x`** that make **`f`** true, assuming that **`f`** is a function that evaluates to true or false based on the input.
- **`zipWith(f,x,y):`**
  - Takes in 2 lists of the same length and passes elements at the same index into a binary function **`f`** to return a single list that contains elements of the form **`f(x[i], y[i])`**.
- **`find(f,x):`**
  - Returns the first element of list **`x`** for which **`f`** evaluates to true.

# Main Takeaways

- Functional Programming is a way of structuring programs using mathematical functions that take in inputs and return an output.
- Functions written in this paradigm:
  - Don't mutate any data (stateless)
  - Are deterministic (always have the same output)
  - Are first-class citizens
- The functional approach allows us to write programs very concisely using higher-order function abstractions.
- Testing and debugging is easier when the overall program is split into functions that are used as a big function composition.

# Further Reading

- Examples of functional languages include Haskell, Common Lisp, Closure, OCaml
- Learning the functional paradigm through Haskell: http://learnyouahaskell.com/
- How To Design Programs (in a functional style): http://www.ccs.neu.edu/home/matthias/HtDP2e/

# Questions?