Functional Programming

April 25, 2017

Agenda

f(x)

Map and Filter

Lambdas

Iterators/Generators

Decorators!

Preface

Programming Paradigms

Procedural

Sequence of instructions that inform the computer what to do with the program's input

Examples

C
Pascal
Unix (sh)

Declarative

Specification describes the problem to be solved, and language implementation figures out the details

Examples

SQL
Prolog

Multi-Paradigm

Supports several different paradigms, to be combined freely

Examples

Scala
C++
Python

Object-Oriented

Deal with collections of objects which maintain internal state and support methods that query or modify this internal state in some way.

Examples

Java
Smalltalk

Functional

Decomposes into a set of functions, each of which solely takes inputs and produces outputs with no internal state.

Examples

Haskell
OCaml
ML

Functional Programming Concepts

- Primary entity is a "function"
- "Pure" functions are mathematical
  - Output depends only on input
  - No side effects that modify internal state
    - `print()` and `file.write()` are side effects
- Strict (Haskell): no assignments, variables, or state
- Flexible (Python): encourage low-interference functions
  - Functional-looking interface but use variables, state internally

A Difference

Pseudocode

```java
public static class NameEntry {
    private String name;
    private int[] ranks;
    public NameEntry() {
        /* initialization */
    }
    public setName(String name) {
        this.name = name;
    }
    public int getRank(int decade) {
        return ranks[decade];
    }
}
```

```haskell
add :: Integer, Integer -> Integer
add x y = x + y

cube :: Integer -> Integer
cube x = x * x * x

concat :: String, String -> String
concat x [] = x
concat x [y:ys] = concat [x:y] ys
```

Why Functional Programming?

Why avoid objects and side effects?

Formal Provability Line-by-line invariants

Modularity Encourages small independent functions

Composability Arrange existing functions for new goals

Easy Debugging Behavior depends only on input

Let's Get Started!

Map / Filter

Common Pattern

```
output = []
for element in iterable:
    val = function(element)
    output.append(val)
return output
```

```
return [function(element)
    for element in iterable]
```

```
[len(s) for s in languages]
["python", "perl", "java", "c++"]
```

len

Apply some function to every element of a sequence

```
[ 6 , 4 , 4, 3]
```

```
map(fn, iter)
```

```
map :: (a -> b) x [a] -> [b]
```

No discussion of elements!

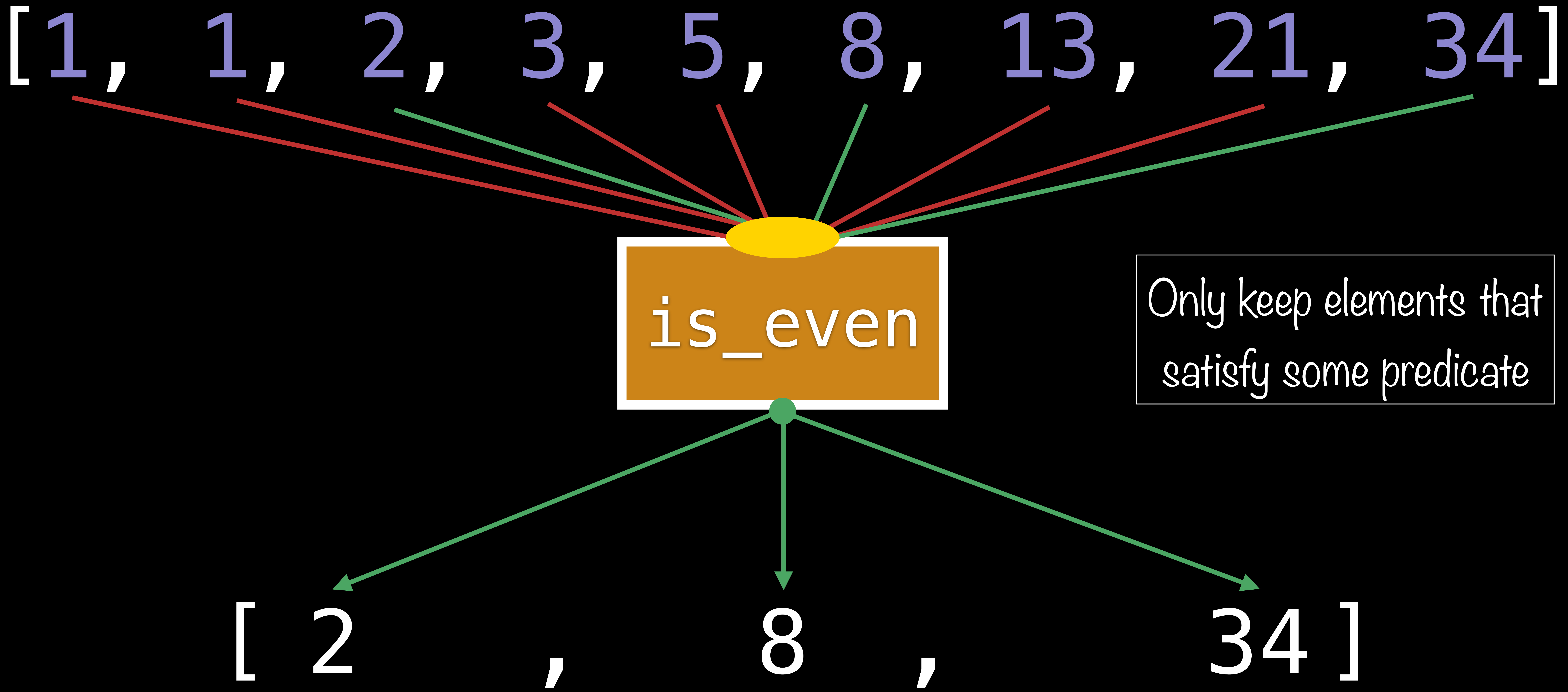

[len(s) for s in languages]
["python", "perl", "java", "c++"]
map(len, languages)
< 6 , 4 , 4, 3>

Another Common Pattern

```python
output = []
for element in iterable:
    if predicate(element):
        output.append(element)
return output
```

```python
[element for element in iterable
         if predicate(element)]
```

```python
[num for num in fibs if is_even(num)]
```



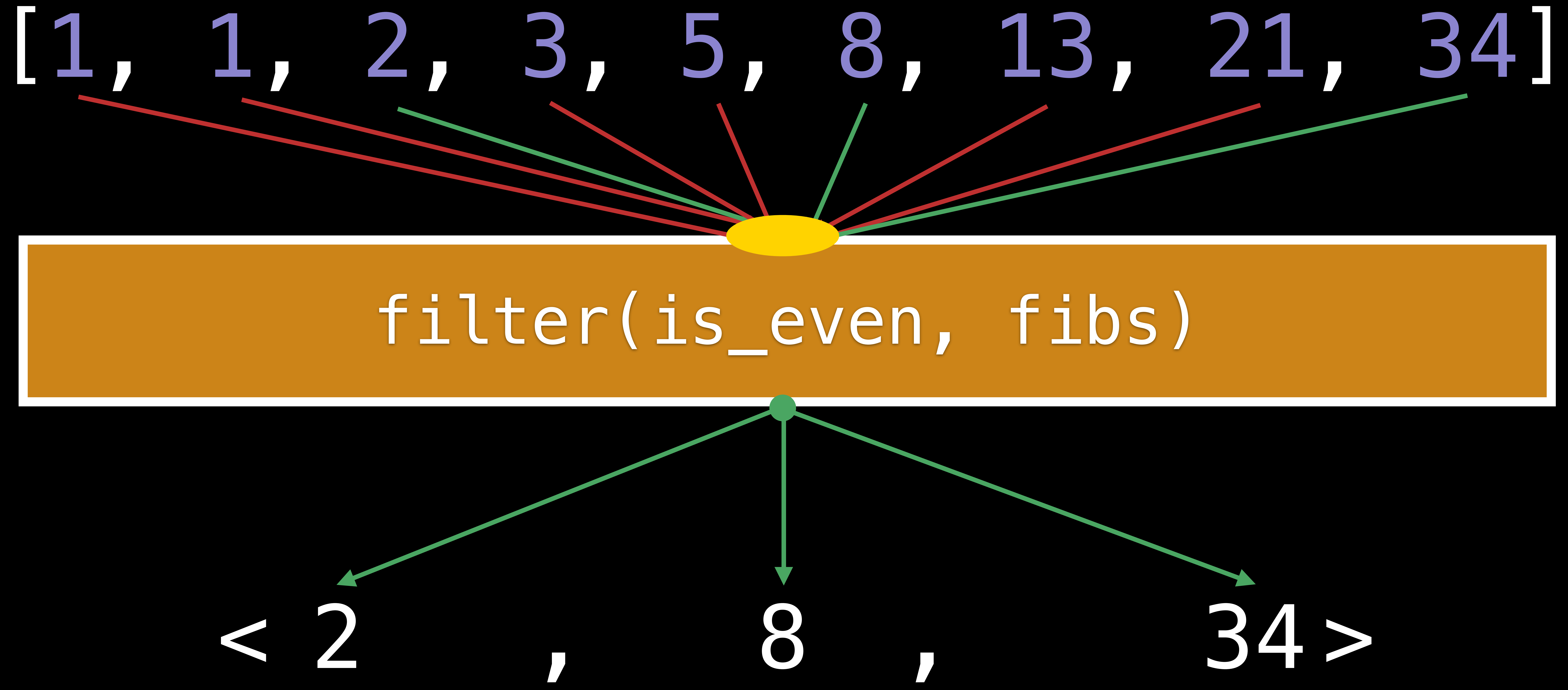
Only keep elements that satisfy some predicate

```
filter(pred, iter)
```

```
filter :: (a -> bool) x [a] -> [a]
```

No discussion of elements!

```
[num for num in fibs if is_even(num)]
```



More Examples

```
# What will the output be?
map(float, ['1.0', '3.3', '-4.2'])
filter(is_prime, range(100))

# Convert the LHS to the RHS using map/filter
[[1, 3], [4, 2, -5]] # <4, 1> (sum)
[1, True, [2, 3]] # => '1 : True : [2, 3]'
[0, 1, 0, 6, 'A', 1, 0, 7] # => <1, 6, 1, 7>
```

List Comprehensions vs. `map` + `filter`

- Memory
  - List Comprehensions: buffer all computed results
  - Map/Filter: only compute output elements when asked
- Speed
  - LCs: no function call overhead, slightly faster usually
  - Map/Filter: function calls, faster in some cases

Lambda Functions

Anonymous, on-the-fly, unnamed functions

Lambda Functions

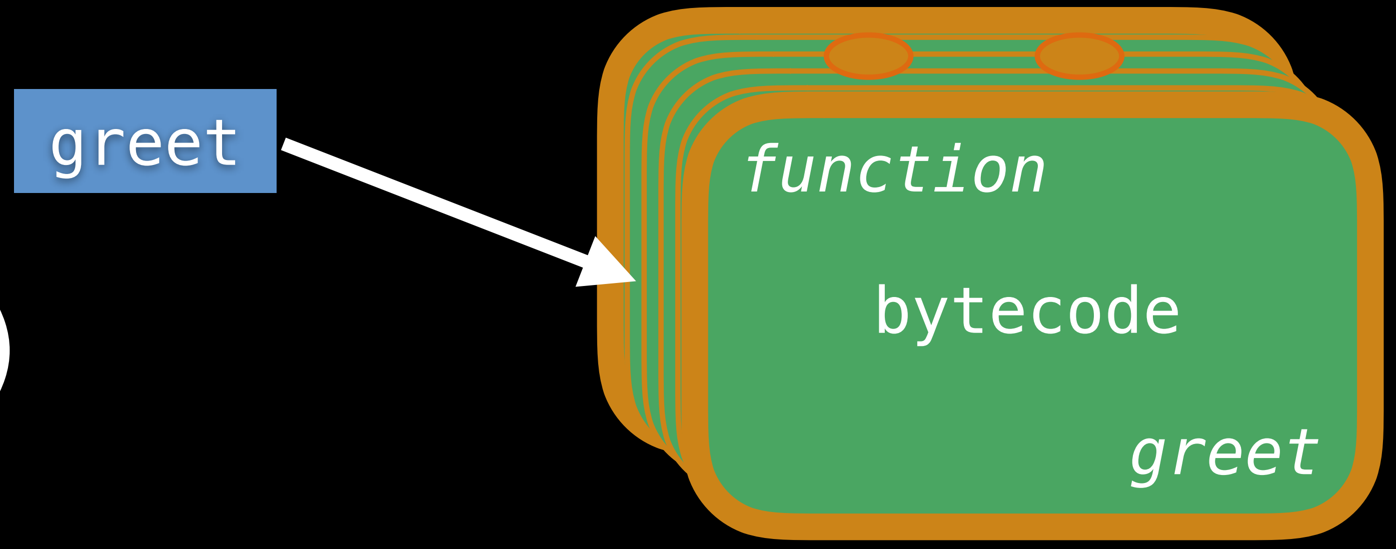
Keyword `lambda` creates an anonymous function

```
lambda params: expr(params)
```

Returns an expression

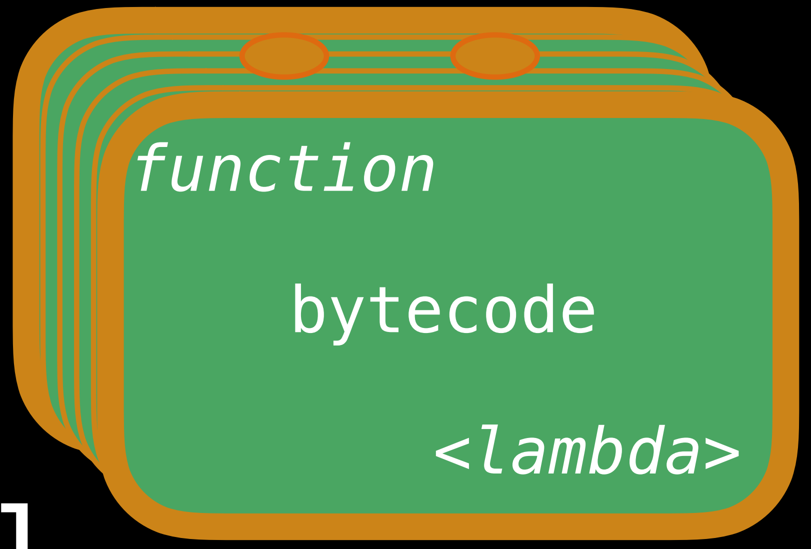
Defined Functions vs. Lambdas

```python
def greet():
    print("Hi!")
```



def binds a name to a function object

```python
lambda val: val ** 2
lambda x, y: x * y
lambda pair: pair[0] * pair[1]
```



lambda only creates a function object

```python
(lambda x: x > 3)(4)  # => True
```

Using Lambdas

```
triple = lambda x: x * 3  # NEVER EVER DO THIS

# Squares from 0**2 to 9**2
map(lambda val: val ** 2, range(10))

# Tuples with positive second elements
filter(lambda pair: pair[1] > 0, [(4,1), (3, -2), (8,0)]
```

Iterators and Generators

Stream of data,
returned one element at a time

Iterators

Iterators

Iterators are objects, like (almost) everything in Python

Represent finite or infinite data streams

Use `next(iterator)` to yield successive values

- Raises `StopIteration` error upon termination

Use `iter(data)` to build an iterator for a data structure

Iterable

```
# Build an iterator over [1,2,3]
it = iter([1,2,3])

next(it)  # => 1
next(it)  # => 2
next(it)  # => 3
next(it)  # raises StopIteration error
```

For Loops use Iterators

```
for data in data_source:
    process(data)

# is really
for data in iter(data_source):
    process(data)
```

Iterator sees changes to the underlying data structure

Builtins use Iterators

```
# Return a value
max(iterable)
val in iterable
all(iterable)
```

Consume iterable until return value is known

```
min(iterable)
val not in iterable
any(iterable)
```

What happens for infinite iterators?

```
# Return values are iterable
enumerate(iterable)
map(fn, iterable)
```

```
zip(*iterables)
filter(pred, iterable)
```

To convert to list, use `list(iterable)`

Generator Expressions

"Lazy List Comprehensions"

Generator Expressions

```
(expensive_function(data)
 for data in iterable)
```

For when you just need a stream of data, not all of it

```python
needle in
(expensive_fn(item)
          for item in haystack)

needle in
[expensive_fn(item)
          for item in haystack]
```

What's the difference?

Generators

"Resumable Functions"

Generators

Regular Functions vs. Generator Functions

Regular Functions

Return a single, computed value

Each call generates a new private namespace and new local variables, then variables are thrown away

Generators

Return an iterator that *generates* a stream of values

Local variables aren't thrown away when exiting a function – you can resume where you left off!

Simple Generator

```python
def generate_ints(n):
    for i in range(n):
        yield i

g = generate_ints(3)
type(g)  # => <class 'generator'>
next(g)  # => 0
next(g)  # => 1
next(g)  # => 2
next(g)  # raises StopIteration
```

The `yield` keyword tells Python to convert the function into a generator

Another Generator

```
def generate_fibs():
    a, b = 0, 1
    while True:
        a, b = b, a + b
        yield a
```

Infinite data stream of Fibonacci numbers

Using Our Generator

```
g = generate_fibs()

next(g)  # => 1
next(g)  # => 1
next(g)  # => 2
next(g)  # => 3
next(g)  # => 5
max(g)   # Oh no! What happens?
```

Lazy Generation

```
def fibs_under(n):
    for fib in generate_fibs():  # Loops over 1, 1, 2, …
        if fib > n:
            break
        print(fib)
```

Summary: Why Use Iterators and Generators?

- Compute data on demand
  - Reduces in-memory buffering
  - Avoid expensive function calls
  - Describe (finite or infinite) streams of data
- `range`, `map`, `filter` and others are or use iterables
- Asynchronous programming (network/web)

Time-Out for Announcements

Logistics

Assignment 1 Due Thursday @midnight

Feedback Optional 1-on-1 sessions about PS1 or general

Office Hours Sam (after class), course staff (appointment)

Assignment 2 Quest for the Holy Grail (!)

Back to Python!

Decorators

Functions as Arguments

```python
# map(fn, iterable)
# filter(pred, iterable)

def perform_twice(fn, *args, **kwargs):
    fn(*args, **kwargs)
    fn(*args, **kwargs)

perform_twice(print, 5, 10, sep='&', end='...')
# => 5&10...5&10...
```

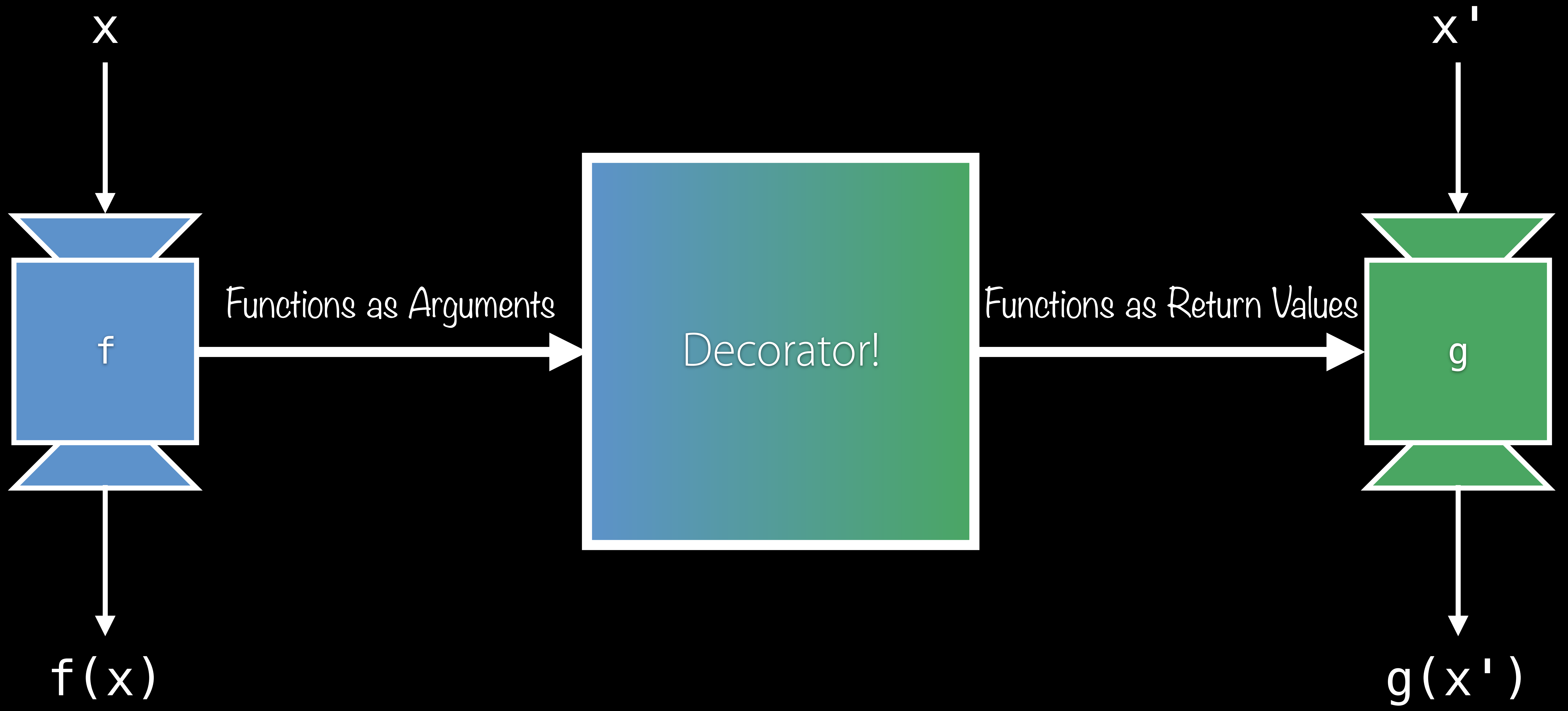
Functions as Return Values

```
def make_divisibility_test(n):
    def divisible_by_n(m):
        return m % n == 0
    return divisible_by_n

div_by_3 = make_divisibility_test(3)
filter(div_by_3, range(10))  # generates 0, 3, 6, 9
make_divisibility_test(5)(10)  # => True
```

Something Cool (but Complex)

```
def primes_under(n):
    tests = []
    # will hold [div_by_2, div_by_3, div_by_5, …]

    for i in range(2, n):
        # implement is_prime using our divis. tests
        if not any(map(lambda test: test(i), tests)):
            tests.append(make_divisibility_test(i))
            yield i
```

Why not both?


x
x'
f
Functions as Arguments
Decorator!
Functions as Return Values
g
f(x)
g(x')

Writing Our First Decorator

```
def debug(function):
    def wrapper(*args, **kwargs):
        print("Arguments:", args, kwargs)
        return function(*args, **kwargs)
    return wrapper
```

Woah <('_'<) – Pause for questions

Using our **debug** decorator

```
def foo(a, b, c=1):
    return (a + b) * c

foo = debug(foo)
foo(2, 3)  # prints "Arguments: (2, 3) {}
# => returns 5
foo(2, 1, c=3) # prints "Arguments: (2, 1) {'c': 3}"
# => returns 9
print(foo)  # <function debug.<locals>.wrapper at 0x...>
```

It seems like overkill to say foo twice here

Using our **debug** decorator

```python
@debug
def foo(a, b, c=1):
    return (a + b) * c

foo(5, 3, c=2)  # prints "Arguments: (5, 3) {'c': 2}"
# => returns 16
```

@decorator applies a decorator to the following function

Other Uses of Decorators

Cache function return value (memoization)

Set timeout for blocking function

Mark class properties as readonly

Mark methods as static methods or class methods

Handle administrative logic (authorization, routing, etc)

Next Time

Next Time

- More `map` and `filter`
- Investigate iterators/generators
- Explore function closures
- Build some decorators
    - Including a type checker!

Credit

Python Documentation, of course

Guide to Functional Programming

A few other sites, which I've unfortunately forgotten.