Object-Oriented Python

February 5, 2019

Overview

Recap of FP

Classes

Instances

Inheritance

Magic Methods

Exceptions

Recap from Last Week

Why Functional Programming?

Why avoid objects and side effects?

Formal Provability Line-by-line invariants

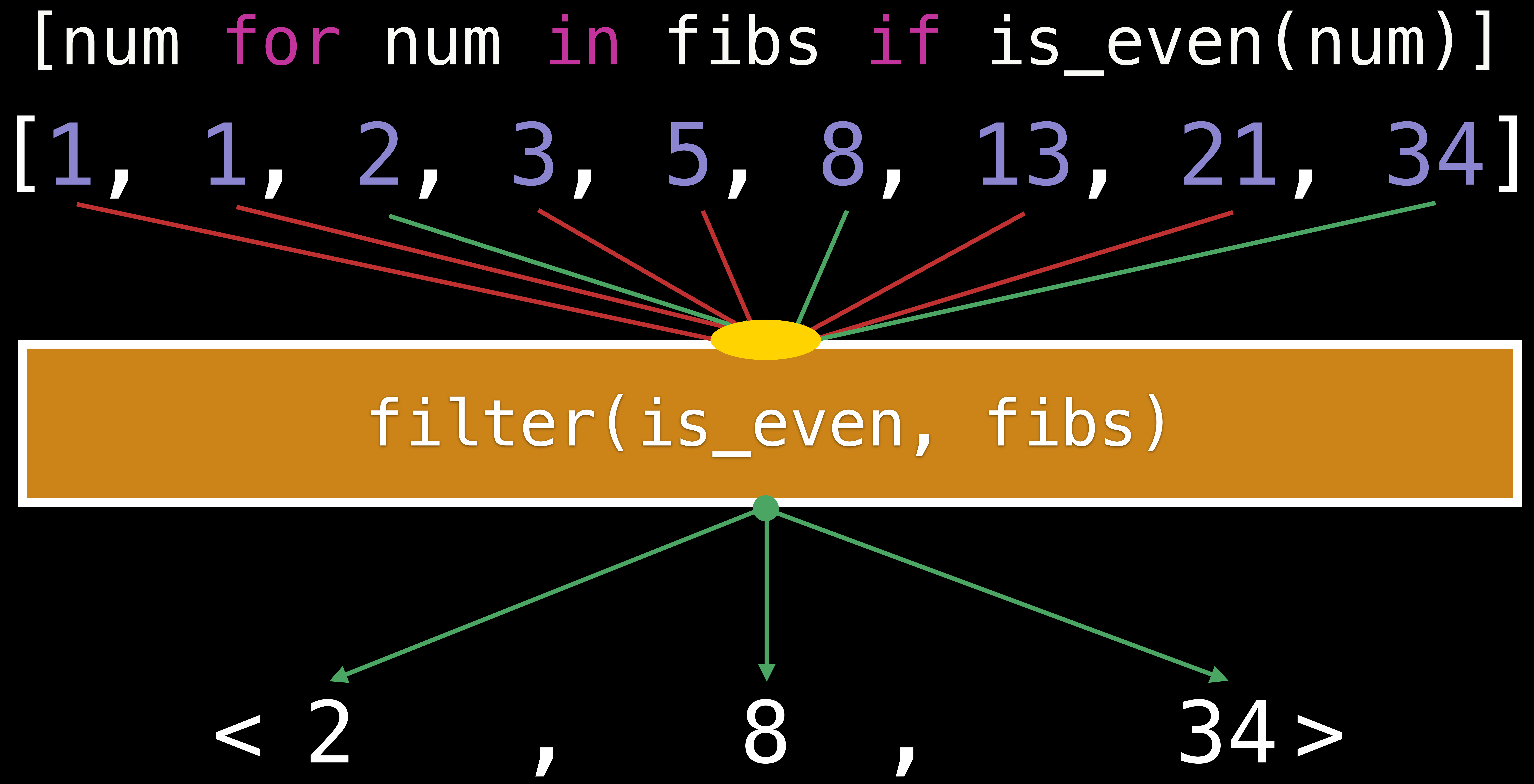
Modularity Encourages small independent functions

Composability Arrange existing functions for new goals

Easy Debugging Behavior depends only on input

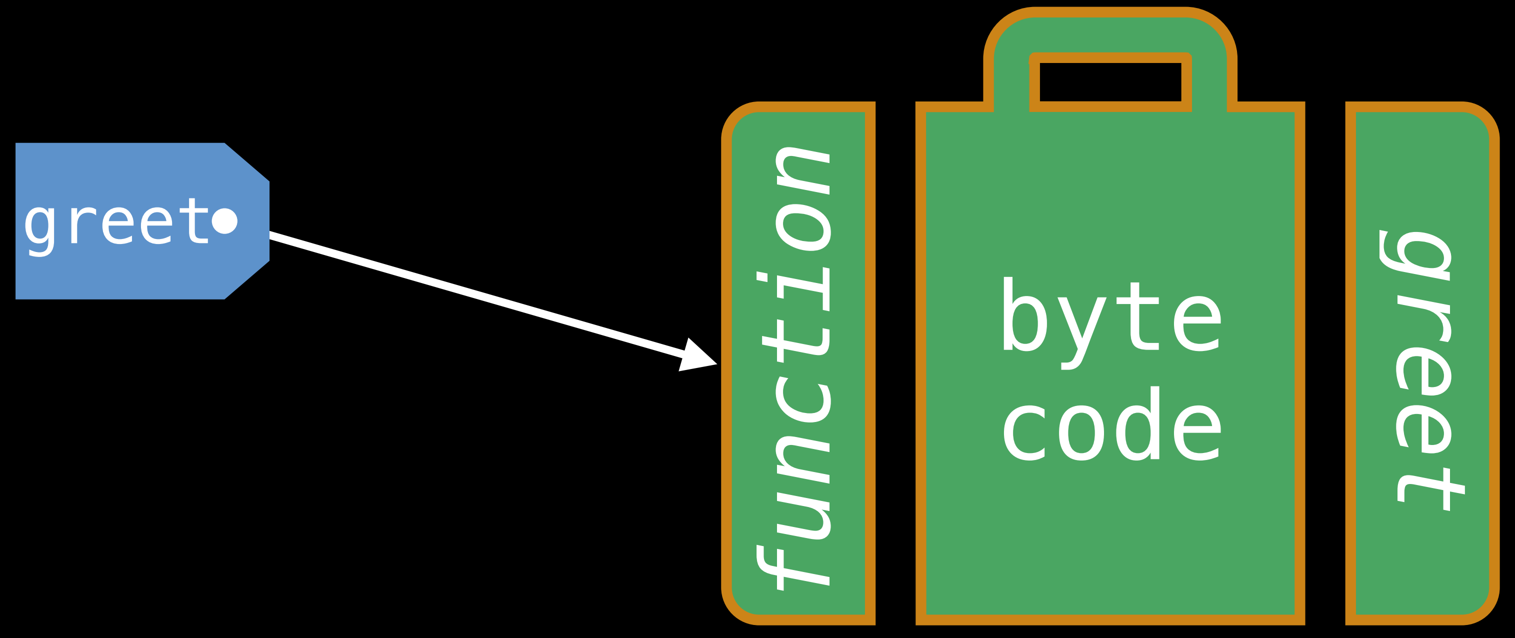
Let's Get Started!

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

map(len, languages)

< 6 , 4 , 4, 3>
```


[num for num in fibs if is_even(num)]
[1, 1, 2, 3, 5, 8, 13, 21, 34]
filter(is_even, fibs)
< 2 , 8 , 34>

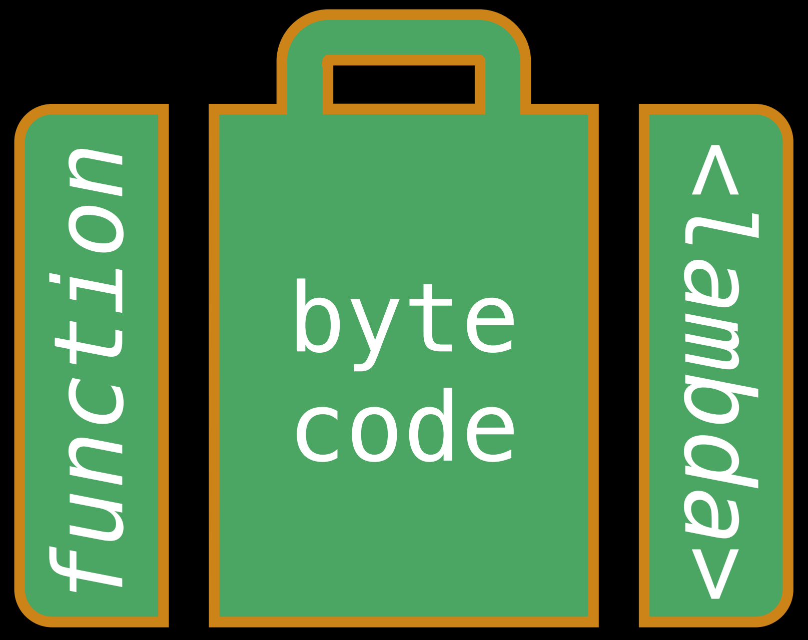
Defined Functions vs. Lambdas

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



def binds a name to a function object

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

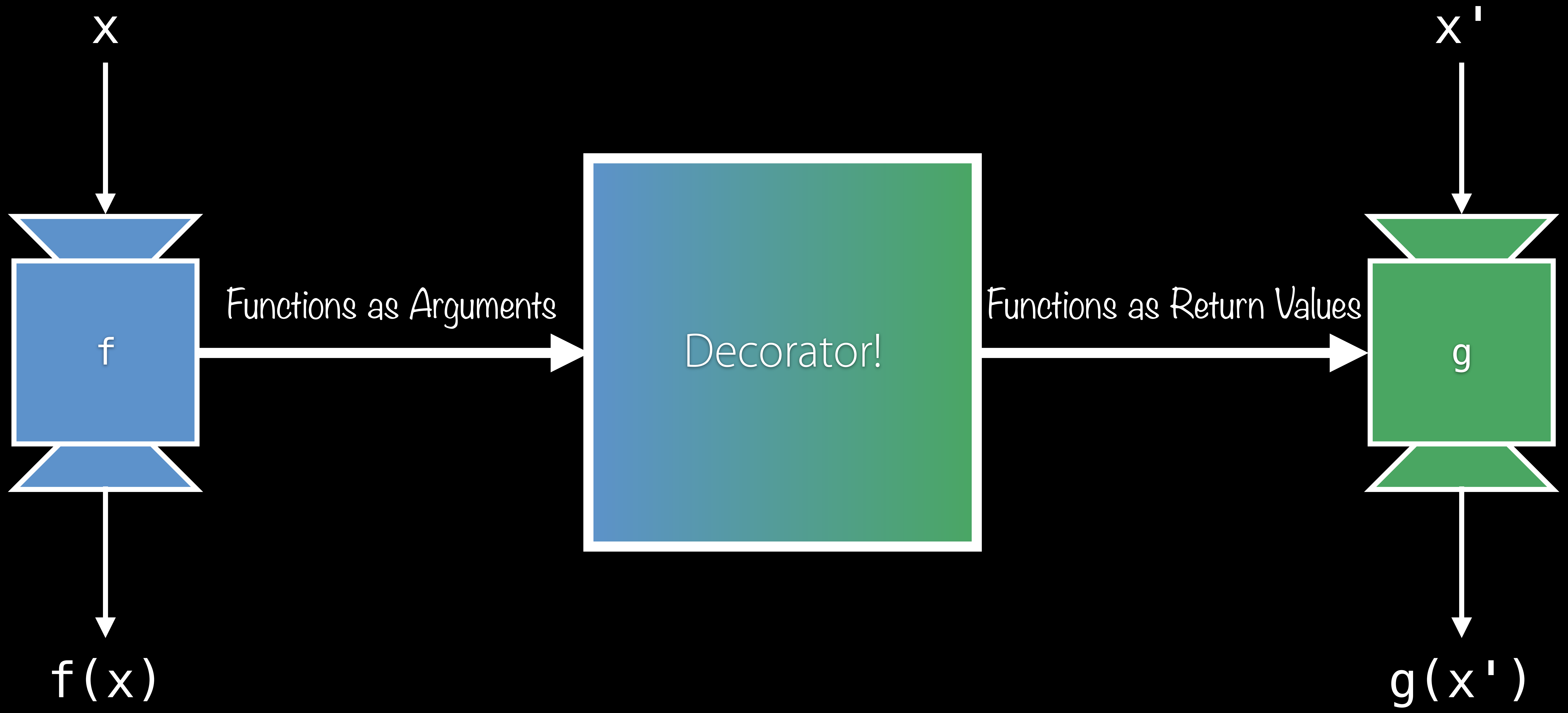


lambda only creates a function object

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

Creates a function object and immediately call it

Decorators



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

Our First Decorator

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

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

Object-Oriented Python

Recall: 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

Objects, Names, Attributes

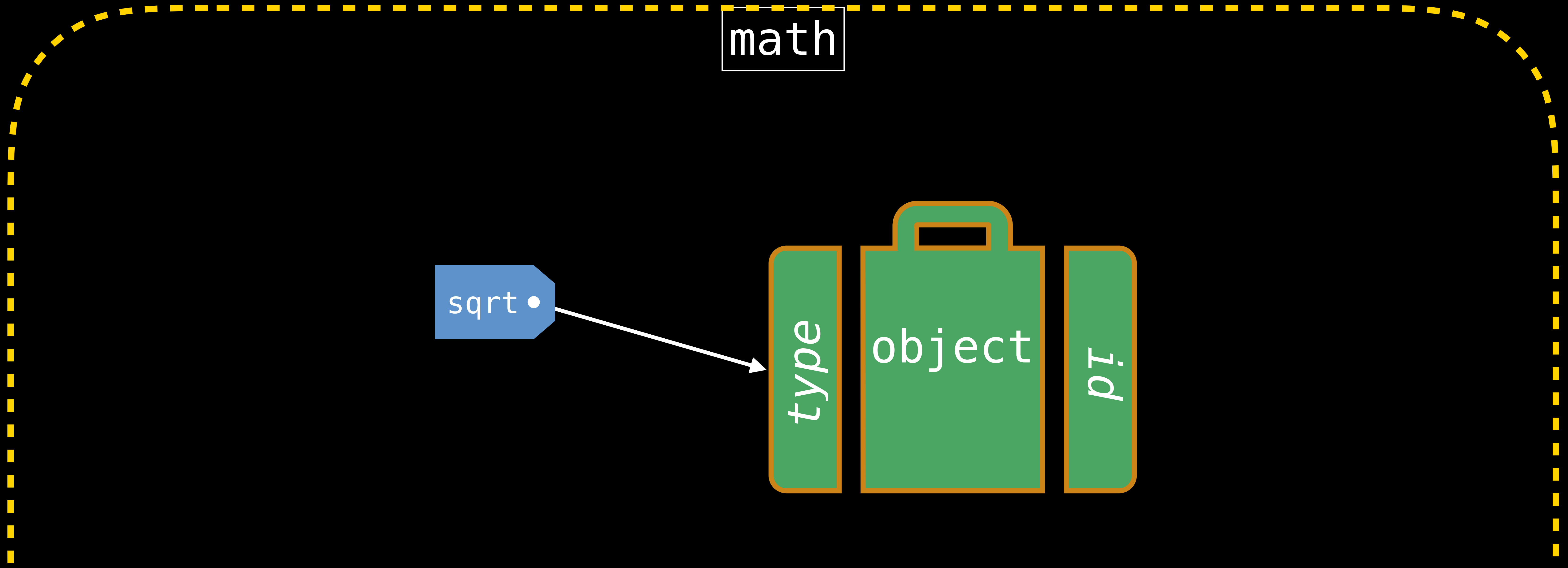
Recall: Some Definitions

An *object* has identity

A *name* is a reference to an object

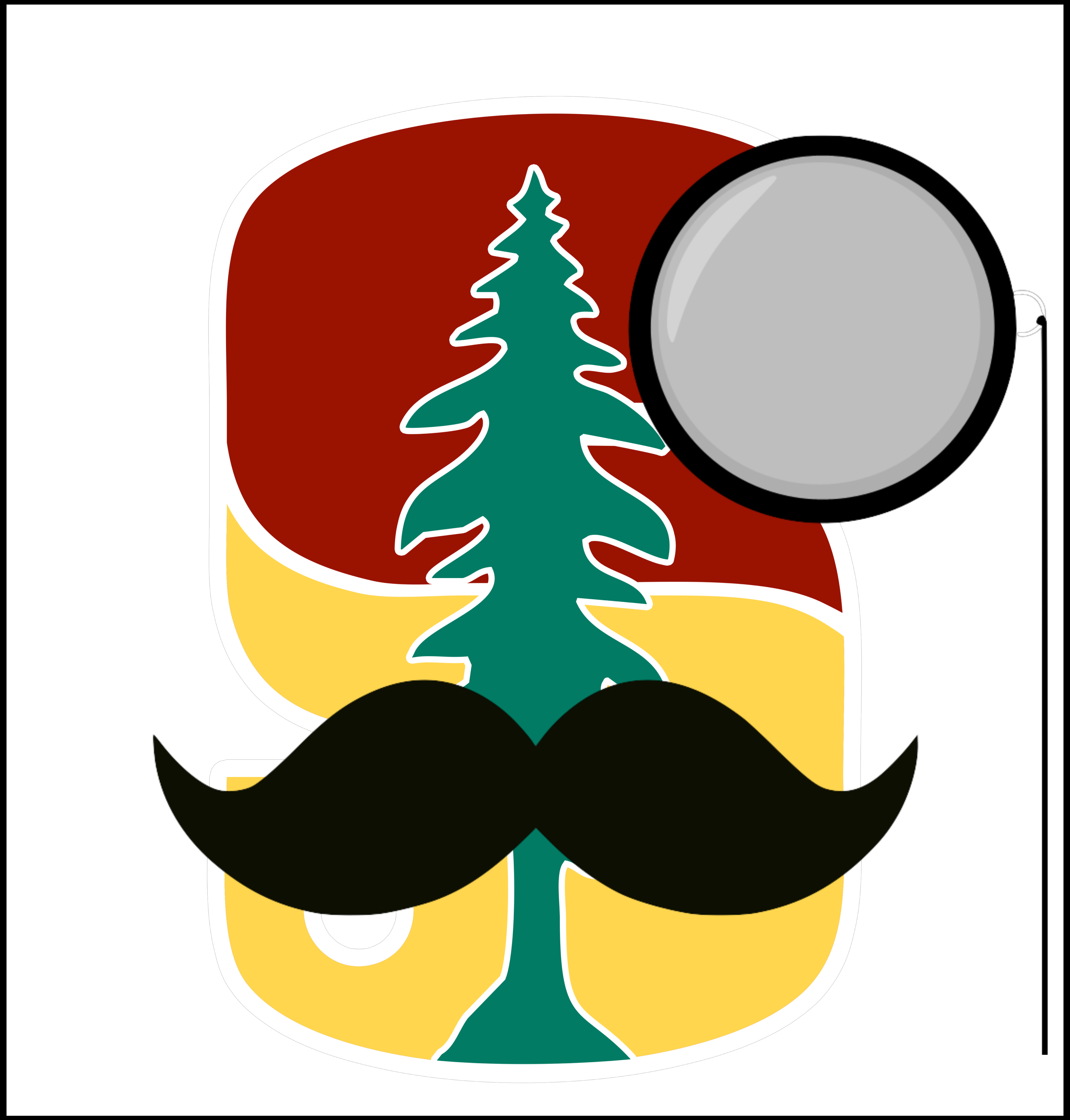
A *namespace* is an associative mapping from names to objects

An *attribute* is any name following a dot (`'.'`)



Classes

First Look at Classes



New Syntax

Class Objects

Instance Objects

Methods vs. Functions

Who says Python isn't classy?

Class Definition Syntax

The `class` keyword introduces a new class defintion

```
class ClassName:
    <statement>
    <statement>
    ...
```

Must be executed to have effect (like def)

Class Definitions

Statements are usually assignments or function definitions

Entering a class definition creates a new "namespace"-ish

- Really, a special `__dict__` attribute where attributes live

Exiting a class definition creates a *class object*

- Defining a class `==` creating a class object (like `int`, `str`)
- Defining a class `!=` instantiating a class

Wait, What?

Class Objects vs. Instance Objects

Defining a class creates a *class object*

- Supports attribute reference and instantiation

Instantiating a class object creates an *instance object*

- Only supports attribute reference

Class objects are not instance objects!

Class Objects

Support (1) attribute references and (2) instantation

Class Attribute References

Class Attribute References

```
class MyClass:
    """A simple example class."""
    num = 12345
    def greet(self):
        return "Hello world!"

# Attribute References
MyClass.num    # => 12345          (int object)
MyClass.greet  # => <function f> (function object)
```

Warning! Clients can write to (and override) class attributes.

Class Instantiation

Class Instantiation

No `new`

Classes are instantiated using parentheses and an optional argument list

```
x = MyClass(args)
```

"Instantiating" a class constructs an instance object of that class object.
In this case, `x` is an instance object of the `MyClass` class object

We've Seen Instantiation Before

```
# Remember these?
float('3.5')
int('101001', base=2)
str(41)

list('hap.py')
dict(a=1, b=2)
```

Custom Constructor using `__init__`

```
class Complex:
    def __init__(self, realpart=0, imagpart=0):
        self.real = realpart
        self.imag = imagpart
```

Class instantiation calls the special method `__init__` if it exists, supplying a freshly-minted instance object as the first parameter.

```
# Make an instance object!
c = Complex(3.0, -4.5)
c.real, c.imag  # => (3.0, -4.5)
```

You can't overload `__init__`!
Use keyword arguments
or factory methods

Instance Objects

Only support attribute references

Data Attributes

= "instance variables"
= "data members"

```
c = Complex(3.0, -4.5)

# Get attributes
c.real, c.imag  # => (3.0, -4.5)

# Set attributes
c.real = -9.2
c.imag = 4.1
```

Instance Attribute Reference Resolution

```
class MyOtherClass:
    num = 12345
    def __init__(self):
        self.num = 0

x = MyOtherClass()
print(x.num)  # 0 or 12345?
del x.num
print(x.num)  # 0 or 12345?
```

Attribute references first search the instance's `__dict__` attribute, then the class object's

Setting Data Attributes

```
# You can set attributes on instance (and class) objects
# on the fly (we used this in the constructor!)
c.counter = 1
while c.counter < 10:
    c.counter = x.counter * 2
    print(c.counter)
del c.counter  # Leaves no trace

# prints 1, 2, 4, 8
```

Setting attributes actually inserts into the instance object's `__dict__` attribute

Recall: A Sample Class

```
class MyClass:
    """A simple example class."""
    num = 12345
    def greet(self):
        return "Hello world!"

x = MyClass()
x.greet()  # 'Hello world!'
# Weird... doesn't `greet` accept an argument?

print(type(x.greet))         # method <bound method MyClass.greet of ...>
print(type(MyClass.greet))  # function <function MyClass.greet(self)>

print(x.num is MyClass.num)  # True
print(x.greet is MyClass.greet)  # False
```

Methods vs. Functions

Methods vs. Functions

A *method* is like a function attached to an object

```
method ≈ (object, function)
```

Methods calls invoke special semantics

```
object.method(arguments) = function(object, arguments)
```

Example: 🍕🍕🍕

Pizza

```
class Pizza:
    def __init__(self, radius, toppings, slices=8):
        self.radius = radius
        self.toppings = toppings
        self.slices_left = slices

    def eat_slice(self):
        if self.slices_left > 0:
            self.slices_left -= 1
        else:
            print("Oh no! Out of pizza")

    def __repr__(self):
        return '{}" pizza'.format(self.radius)
```

Pizza

```
p = Pizza(14, ("Pepperoni", "Olives"), slices=12)
print(Pizza.eat_slice)
# => <function Pizza.eat_slice>

print(p.eat_slice)
# => <bound method Pizza.eat_slice of 14" Pizza>

method = p.eat_slice
print(method.__self__)  # => 14" Pizza
print(method.__func__)  # => <function Pizza.eat_slice>

p.eat_slice()  # Implicitly calls Pizza.eat_slice(p)
```

Class and Instance Attributes

🐶🐶🐶

@buddypelu on IG

Class and Instance Variables

```python
class Dog:
    kind = 'Canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

a = Dog('Astro')
b = Dog('Buddy')

a.kind  # 'Canine' (shared by all dogs)
b.kind  # 'Canine' (shared by all dogs)
a.name  # 'Astro' (unique to a)
b.name  # 'Buddy' (unique to b)
```

Warning

```
class Dog:
    tricks = []

    def __init__(self, name):
        self.name = name

    def teach_trick(self, trick):
        self.tricks.append(trick)
```

What could go wrong?

Warning

```
d = Dog('Fido')
e = Dog('Buddy')
d.teach_trick('roll over')
e.teach_trick('come here')
d.tricks  # => ['roll over', 'come here'] (shared value)
```

Did we Solve It?

```
class Dog:
    # Let's try a default argument!
    def __init__(self, name='Mr. B', tricks=[]):
        self.name = name
        self.tricks = tricks

    def teach_trick(self, trick):
        self.tricks.append(trick)
```

Hmm...

```
d = Dog('Fido')
e = Dog('Buddy')
d.teach_trick('roll over')
e.teach_trick('come here')
d.tricks  # => ['roll over', 'come here'] (shared value)
```

Solution

```python
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []  # New list for each dog

    def teach_trick(self, trick):
        self.tricks.append(trick)
```

Solution

```
d = Dog('Fido')
e = Dog('Buddy')
d.teach_trick('roll over')
e.teach_trick('come here')
d.tricks  # => ['roll over']
e.tricks  # => ['come here']
```

Privacy and Style

Keep an Eye Out!

Nothing is truly private!

Clients can modify *anything*

"With great power…"

Stylistic Conventions

- A method's first parameter should always be **`self`**
  - Why? Explicitly differentiate instance and local variables
  - Calling a method ensures the caller is the first argument
- Prefix private attributes with an underscore (e.g. `_spam`)
  - Not enforced, but a standard hiding convention
  - Use double underscores (`__spam`) for more obfuscation
- Use verbs for methods and nouns for data attributes

Starting Class Template

```
class MyClass:
    CLASS_LEVEL_CONSTANT = 100

    def __init__(self, arg1, arg2=0):
        self.foo = arg1
        self._bar = arg2
        self.baz = []

    def my_first_method(self, args):
        do_something_with_self_and_args()

    def my_second_method(self, args):
        do_something_else_with_self_and_args()

    def __str__(self):
        return string_representation_of_self
```

Inheritance

Parentheses indicate inheritance

```
class DerivedClassName(BaseClassName):
    pass
```

Any expression is valid

Facts about Single Inheritance

- A class object 'remembers' its base class
- All class objects inherit from `object` (default in Python 3)
- All attribute references start from derived class
  - This includes methods, as attributes of the class object!
  - Proceeds up the chain of base classes
- Derived methods override (shadow) base methods
  - Similar to `virtual` in C++

Multiple Inheritance

"The Dreaded Diamond Pattern"

Multiple Inheritance

Base classes are separated by commas

```
class Derived(Base1, Base2, …, BaseN):
    pass
```

Order matters!

Attribute Resolution

All we need is an order to search for attributes.

Attribute lookup is (almost) breadth-first, left-to-right

- Officially called "C3 Superclass Linearization" (Wikipedia)

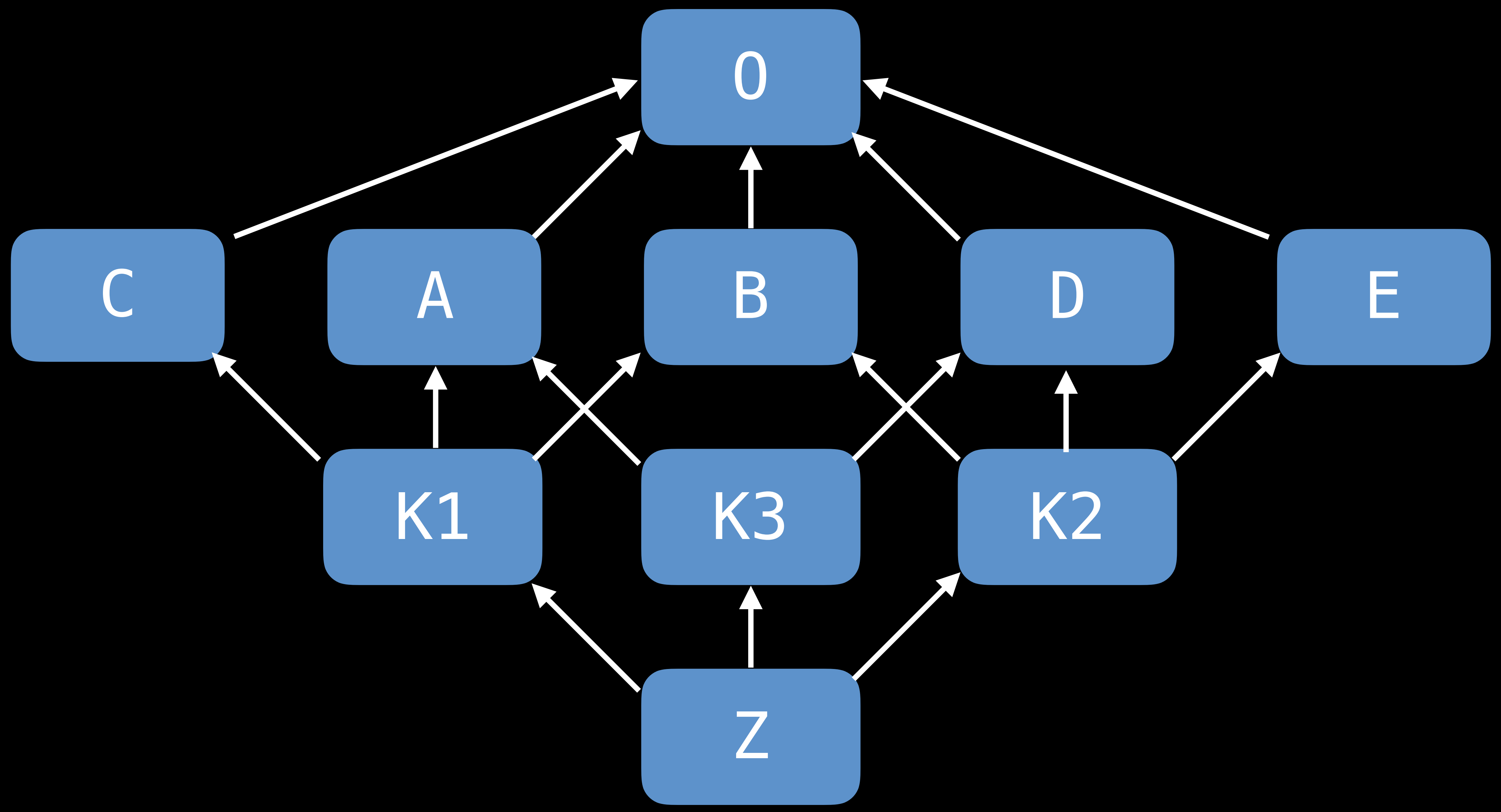
Class objects have a (hidden) function attribute `.mro()`

- Shows linearization of base classes

```
bool.mro()  # => [bool, int, object]
```

Attribute Resolution In Action

```python
class A: pass
class B: pass
class C: pass
class D: pass
class E: pass
class K1(A, B, C): pass
class K2(D, B, E): pass
class K3(D, A): pass
class Z(K1, K2, K3): pass

Z.mro()  # [Z, K1, K2, K3, D, A, B, C, E, object]
```



Magic Methods

__dunderbar_methods__

Magic Methods

- Python uses `__init__` to build classes
    - Overriding `__init__` lets us hook into the language
- What else can we do? Can we define classes that act like:
    - iterators? lists?
    - sets? dictionaries?
    - numbers?
    - comparables?

Implementing Magic Methods

```python
class MagicClass:
    def __init__(self): ...
    def __contains__(self, key): ...
    def __add__(self, other): ...
    def __iter__(self): ...
    def __next__(self): ...
    def __getitem__(self, key): ...
    def __len__(self): ...
    def __lt__(self, other): ...
    def __eq__(self, other): ...
    def __str__(self): ...
    def __repr__(self): ...  # And even more...
```

Python Uses Magic Methods

```
x = MagicClass()
y = MagicClass()
str(x)    # => x.__str__()
x == y    # => x.__eq__(y)

x < y     # => x.__lt__(y)
x + y     # => x.__add__(y)
iter(x)   # => x.__iter__()
next(x)   # => x.__next__()
len(x)    # => x.__len__()
el in x   # => x.__contains__(el)
```

Some builtins, like print and sort, implicitly use `__str__` and `__lt__`

Many, many more

Link 1

Link 2

Link 3

Example: Point

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def rotate_90_CC(self):
        self.x, self.y = -self.y, self.x

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return "Point({0}, {1})".format(self.x, self.y)
```

Objects

```
o = Point()
print(o)          # Point(0, 0)

p1 = Point(3, 5)
p2 = Point(9, -2)
print(p1, p2)     # Point(3, 5) Point(9, -2)

p1.rotate_90_CC()
print(p1)         # Point(-5, 3)

print(p1 + p2)    # Point(4, 1)
```

Now our point object works wherever a + was expected, such as in `sum`

OOP Case Study:
Errors and Exceptions

Syntax Errors

"Errors before execution"

```
>>> while True print("Hello world")
  File "<stdin>", line 1
    while True print("Hello world")
                   ^
SyntaxError: invalid syntax
```

Error is detected at the token preceding the arrow

Exceptions

"Errors during execution"

```
>>> 10 * (1/0)
Traceback (most recent call last):
    File "<stdin>", line 1
ZeroDivisionError: division by zero

>>> 4 + spam*3
Traceback (most recent call last):
    File "<stdin>", line 1
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
    File "<stdin>", line 1
TypeError: Can't convert 'int' object to str implicitly
```

And More

`KeyboardInterrupt`

`UnboundLocalError`

`SystemExit`

`StopIteration`

`SyntaxError`

`ZeroDivisionError`

`AttributeError`

`KeyError`

`IndexError`

`NotImplementedError`

`TypeError`

`OSError`

`NameError`

```
BaseException
 ├── SystemExit
 ├── KeyboardInterrupt
 ├── GeneratorExit
 └── Exception
      ├── StopIteration
      ├── StopAsyncIteration
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    └── RecursionError
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── DeprecationWarning
           ├── PendingDeprecationWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UserWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── UnicodeWarning
           ├── BytesWarning
           └── ResourceWarning
```

Inheritance in Action!

Handling Exceptions

What Might Go Wrong?

```python
def read_int():
    """Read an integer from the user."""
    return int(input("Please enter a number: "))
```

What happens if the user enters a nonnumeric input?

Solution

```
def read_int():
    """Read an integer from the user (better)."""
    while True:
        try:
            x = int(input("Please enter a number: "))
            break
        except ValueError:
            print("Oops! Invalid input. Try again...")
    return x
```

Handling Exceptions

```
try:
    dangerous_code()
except SomeError:
    handle_the_error()
```

How `try` works

1) Attempt to execute the try clause

2a) If no exception occurs, skip the except clause. Done!

2b) If an exception occurs, skip the rest of the try clause.

  2bi) If the type of the raised exception is a subclass of the named exception type, then execute the except clause.

  2bii) Otherwise, propagate the exception to the world.

Unhandled exceptions halt execution

Conveniences

```
try:
    distance = int(input("How far? "))
    time = car.speed / distance
    car.drive(time)
except ValueError as e:
    print(e)
except ZeroDivisionError:
    print("Division by zero!")
except (NameError, AttributeError):
    print("Bad Car")
except:
    print("Car unexpectedly crashed!")
```

Bind a name to the exception instance

Catch multiple exceptions

"Wildcard" catches everything

Solution?

```python
def read_int():
    """Read an integer from the user (fixed?)."""
    while True:
        try:
            x = int(input("Please enter a number: "))
            break
        except:
            print("Oops! Invalid input. Try again...")
    return x
```

"I'll just catch 'em all!"

Oops! Now we can't CTRL+C to escape

Raising Exceptions

The raise keyword

```
>>> raise NameError('Why hello there!')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError: Why hello there!

>>> raise NameError
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError
```

You can raise either instance objects or class objects

raise within except clause

```
try:
    raise NotImplementedError("TODO")
except NotImplementedError:
    print('Looks like an exception to me!')
    raise
# Looks like an exception to me!
# Traceback (most recent call last):
#   File "<stdin>", line 2, in <module>
# NotImplementedError: TODO
```

Re-raises the currently active exception

Used to acknowledge an exception
but also propagate to external handlers

Good Python: Using `else`

```
try:
    ...
except ...:
    ...
else:
    do_something()
```

Code that executes if the try clause does not raise an exception

Why? Avoid accidentally catching an exception raised by something other than the code being protected

Example: Database Transactions

```
try:
    update_the_database()
except TransactionError:
    rollback()
    raise
else:
    commit()
```

If the commit raises a TransactionException,
we might actually *want* to crash

Aside: Python Philosophy

Coding for the Common Case (Controversial)

"Easier to Ask for Forgiveness than Permission" (EAFP)

- vs. "Look Before You Leap" (LBYL)

Just open a file instead of checking that it exists first!

- Handle exceptional cases with an except clause (or two)

Just pop an element; don't check that a list is nonempty!

Helps combat race conditions

Often a source of bugs if exceptional cases are forgotten!

Good Python:
Custom Exceptions

Custom Exceptions

```
class Error(Exception):
    """Base class for errors in this module."""

class BadLoginError(Error):
    """A user attempted to login with
    an incorrect password."""
```

You can define an `__init__` method to be fancy

Remember, explicit is better than implicit!
BadLoginError is more descriptive than e.g. KeyError

Cleanup Actions

The finally clause

Executed upon leaving the try/except/else block

```
try:
    raise NotImplementedError
finally:
    print('Goodbye, world!')

# Goodbye, world!
# Traceback (most recent call last):
#   File "<stdin>", line 2, in <module>
# NotImplementedError
```

How `finally` works

Always executed before leaving the try statement.

Unhandled exceptions (not caught, or raised in except)

are re-raised after finally executes.

Also executed "on the way out" (break, continue, return)

Recall `with ... as ...`

```
# This is what enables us to use with ... as ...
with open(filename) as f:
    raw = f.read()
# is (almost) equivalent to
f = open(filename)
f.__enter__()
try:
    raw = f.read()
finally:
    f.__exit__()  # Closes the file
```

Surprisingly useful and flexible!