Overview

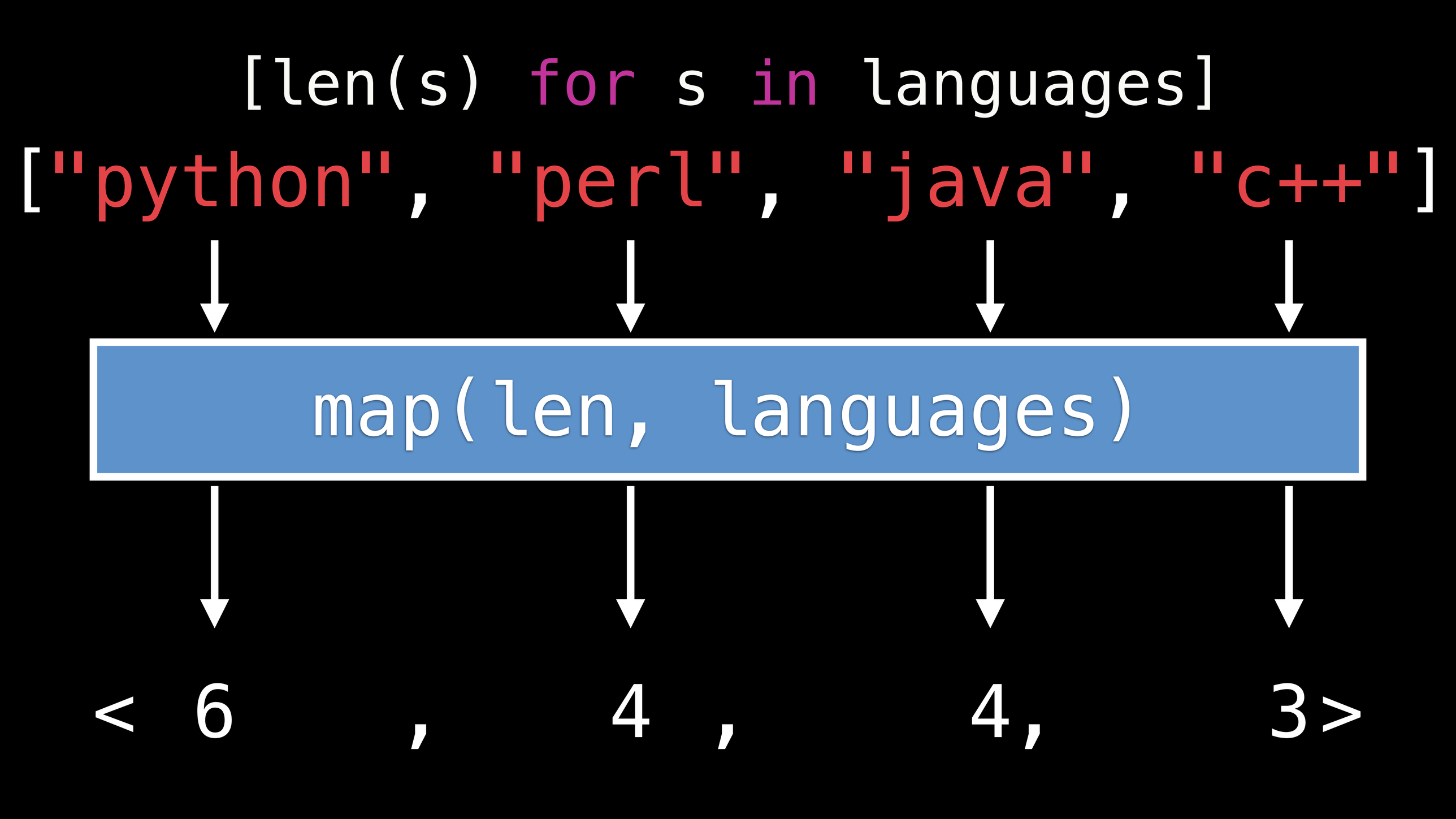
Recap of FP

Classes

Instances

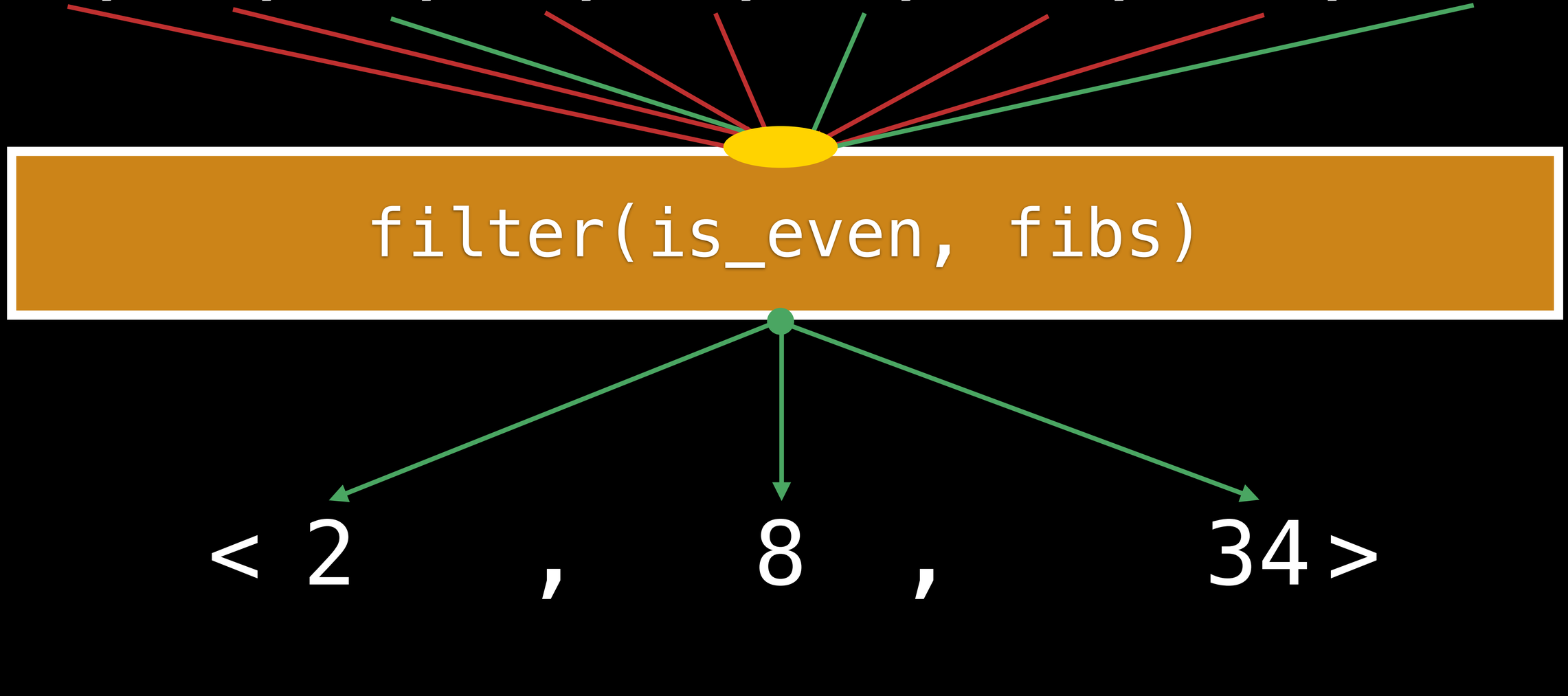
Inheritance

Exceptions


[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)]
```

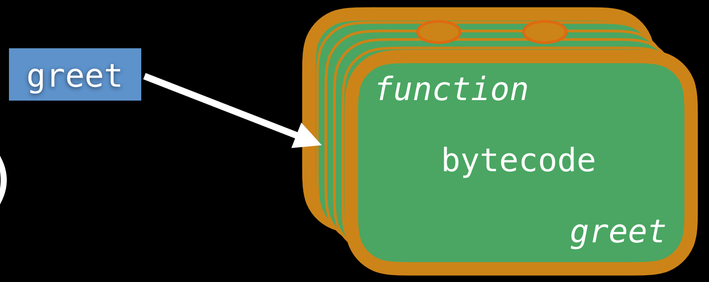
```
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```



Function Definitions vs. Lambdas

Function Definitions vs. Lambdas

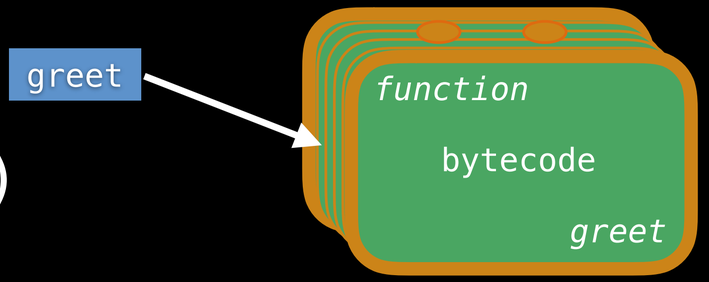
```
def greet():
    print("Hi!")
```



`def` binds a function object to a name

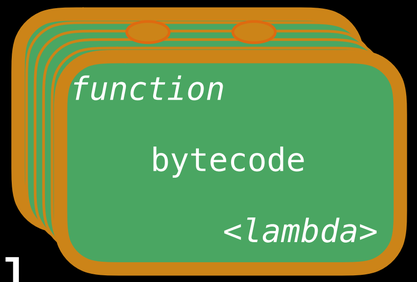
Function Definitions vs. Lambdas

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



`def` binds a function object to a name

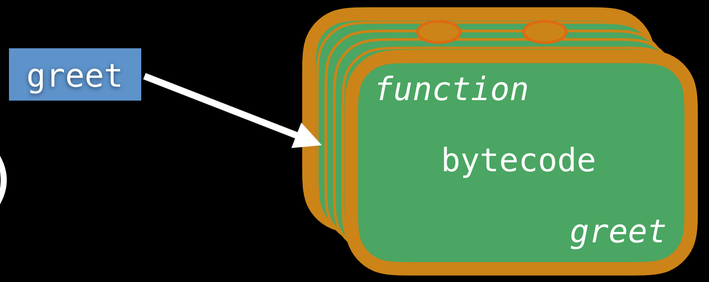
```
lambda val: val ** 2
lambda x, y: x * y
lambda pair: pair[0] * pair[1]
```



`lambda` only creates a function object

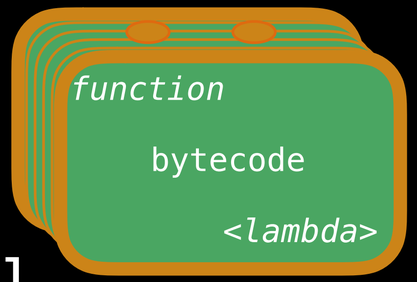
Function Definitions vs. Lambdas

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



`def` binds a function object to a name

```
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
```

Our First Decorator

```
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

Recall: Programming Paradigms

Procedural

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

Examples

C

Pascal

Unix (sh)

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

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

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

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

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

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

Some Definitions

Some Definitions

An o*bject* has identity

Some Definitions

An *object* has identity

A *name* is a reference to an object

Some Definitions

An *object* has identity

A *name* is a reference to an object

A n*amespace* is an associative mapping from names to objects

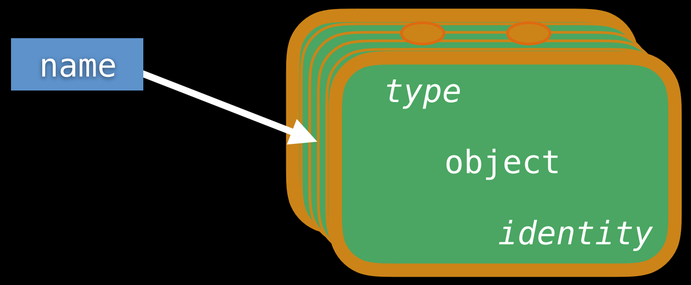
Some Definitions

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

Class Definition Syntax

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

The `class` keyword introduces a new class defintion

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

The `class` keyword introduces a new class defintion

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

Must be executed to have effect (like def)

Class Definitions

Class Definitions

Statements are usually assignments or function definitions

Class Definitions

Statements are usually assignments or function definitions

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

Class Definitions

Statements are usually assignments or function definitions

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

Exiting a class definition creates a class object

Class Definitions

Statements are usually assignments or function definitions

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

Exiting a class definition creates a class object

Defining a class == creating a class object (like int, str)

Class Definitions

Statements are usually assignments or function definitions

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

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

Support (1) attribute references
and (2) instantation

Class Attribute References

Class Attribute References

Class Attribute References

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

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)
```

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! Class attributes can be written to by the client

Class Instantiation

Class Instantiation

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

Class Instantiation

No new

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

Class Instantiation

No new

Classes are instantiated using parentheses and an argument list

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

Class Instantiation

No `new`

Classes are instantiated using parentheses and an 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

Custom Constructor using `__init__`

Custom Constructor using __init__

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

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

Custom Constructor using `__init__`

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

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

Class instantiation calls the special method `__init__` if it exists

Custom Constructor using `__init__`

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

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

Class instantiation calls the special method `__init__` if it exists

Custom Constructor using __init__

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

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

Class instantiation calls the special method __init__ if it exists

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

Instance Objects

Only support attribute references

Data Attributes

= "instance variables"

= "data members"

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

Data Attributes

= "instance variables"

= "data members"

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

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

Data Attributes

= "instance variables"

= "data members"

```
c = Complex(3.0, -4.5)
c.real, c.imag  # => (3.0, -4.5)
```

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

Data Attributes

= "instance variables"

= "data members"

```
c = Complex(3.0, -4.5)
c.real, c.imag  # => (3.0, -4.5)
```

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

Data Attributes

= "instance variables"
= "data members"

```
c = Complex(3.0, -4.5)
c.real, c.imag  # => (3.0, -4.5)

c.real = -9.2
```

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

Data Attributes

= "instance variables"
= "data members"

```
c = Complex(3.0, -4.5)
c.real, c.imag  # => (3.0, -4.5)

c.real = -9.2
c.imag = 4.1
```

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

Setting Data Attributes

Setting Data Attributes

```
# You can set attributes on instance (and class) objects
# on the fly (we used this in the constructor!)
```

Setting Data Attributes

```
# You can set attributes on instance (and class) objects
# on the fly (we used this in the constructor!)
c.counter = 1
```

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
```

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 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

A Sample Class

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

Calling Methods

Calling Methods

```
x = MyClass()
```

Calling Methods

```
x = MyClass()
x.greet()  # 'Hello world!'
```

Calling Methods

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

Calling Methods

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

Calling Methods

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

print(type(x.greet))        # method
```

Calling Methods

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

print(type(x.greet))        # method
print(type(MyClass.greet))  # function
```

Calling Methods

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

print(type(x.greet))        # method
print(type(MyClass.greet))  # function
```

Calling Methods

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

print(type(x.greet))        # method
print(type(MyClass.greet))  # function

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

Methods vs. Functions

Methods vs. Functions

A *method* is a function bound to an object

```
method ≈ (object, function)
```

Methods calls invoke special semantics

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

Pizza

Pizza

```
class Pizza:
```

Pizza

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

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")
```

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

Pizza

```
p = Pizza(14, ("Pepperoni", "Olives"), slices=12)
```

Pizza

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

Pizza

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

Pizza

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

Pizza

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

print(p.eat_slice)
```

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>
```

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>
```

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
```

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
method.__self__  # => 14" Pizza
```

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
method.__self__  # => 14" Pizza
method.__func__  # => <function Pizza.eat_slice>
```

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
method.__self__  # => 14" Pizza
method.__func__  # => <function Pizza.eat_slice>
```

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
method.__self__  # => 14" Pizza
method.__func__  # => <function Pizza.eat_slice>

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

Class and Instance Variables

Class and Instance Variables

Class and Instance Variables

```
class Dog:
```

Class and Instance Variables

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

Class and Instance Variables

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

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

Class and Instance Variables

```
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')
pb = Dog('Mr. Peanut Butter')
```

Class and Instance Variables

```
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')
pb = Dog('Mr. Peanut Butter')

a.kind   # 'Canine' (shared by all dogs)
pb.kind  # 'Canine' (shared by all dogs)
a.name   # 'Astro' (unique to a)
pb.name  # 'Mr. Peanut Butter' (unique to pb)
```

Warning

Warning

```
class Dog:
```

Warning

```
class Dog:
    tricks = []
```

Warning

```
class Dog:
    tricks = []

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

Warning

```
class Dog:
    tricks = []

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

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

Warning

```
class Dog:
    tricks = []

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

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

What could go wrong?

Warning

Warning

```
d = Dog('Fido')
e = Dog('Buddy')
```

Warning

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
```

Warning

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks  # => ['roll over', 'play dead'] (shared value)
```

Did we Solve It?

Did we Solve It?

```
class Dog:
```

Did we Solve It?

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

Did we Solve It?

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

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

Hmm…

Hmm...

```
d = Dog('Fido')
e = Dog('Buddy')
```

Hmm...

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
```

Hmm...

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks  # => ['roll over', 'play dead'] (shared value)
```

Solution

Solution

```
class Dog:
```

Solution

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

Solution

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

    def add_trick(self, trick):
```

Solution

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

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

Solution

Solution

```
d = Dog('Fido')
e = Dog('Buddy')
```

Solution

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
```

Solution

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks  # => ['roll over']
e.tricks  # => ['play dead']
```

Privacy and Style

Keep an Eye Out!

Keep an Eye Out!

Nothing is truly private!

Keep an Eye Out!

Nothing is truly private!

Clients can modify *anything*

Keep an Eye Out!

Nothing is truly private!

Clients can modify *anything*

"With great power…"

Stylistic Conventions

Stylistic Conventions

A method's first parameter should always be **`self`**

Stylistic Conventions

A method's first parameter should always be **`self`**

Why? Explicitly differentiate instance vars from local vars

Stylistic Conventions

- A method's first parameter should always be **`self`**
  - Why? Explicitly differentiate instance vars from local vars
  - Recall: method calls implicitly provide the calling object

Stylistic Conventions

A method's first parameter should always be **`self`**

Why? Explicitly differentiate instance vars from local vars

Recall: method calls implicitly provide the calling object

as the first argument to the class function

Stylistic Conventions

A method's first parameter should always be **`self`**

- Why? Explicitly differentiate instance vars from local vars
- Recall: method calls implicitly provide the calling object as the first argument to the class function

Attribute names prefixed with a leading underscore are

Stylistic Conventions

- A method's first parameter should always be **`self`**
  - Why? Explicitly differentiate instance vars from local vars
  - Recall: method calls implicitly provide the calling object as the first argument to the class function
- Attribute names prefixed with a leading underscore are intended to be private (e.g. _spam)

Stylistic Conventions

A method's first parameter should always be **`self`**

- Why? Explicitly differentiate instance vars from local vars
- Recall: method calls implicitly provide the calling object as the first argument to the class function

Attribute names prefixed with a leading underscore are intended to be private (e.g. _spam)

Use verbs for methods and nouns for data attributes

Inheritance

Note the parentheses

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

Any expression is fine

Single Inheritance

Single Inheritance

A class object 'remembers' its base class

Single Inheritance

A class object 'remembers' its base class

If you don't specify a base class, implicitly use object

Single Inheritance

A class object 'remembers' its base class

If you don't specify a base class, implicitly use object

Method and attribute lookup begins in the derived class

Single Inheritance

A class object 'remembers' its base class

If you don't specify a base class, implicitly use object

Method and attribute lookup begins in the derived class

- Proceeds down the chain of base classes

Single Inheritance

A class object 'remembers' its base class

If you don't specify a base class, implicitly use object

Method and attribute lookup begins in the derived class

    Proceeds down the chain of base classes

Derived methods override (shadow) base methods

Single Inheritance

A class object 'remembers' its base class

If you don't specify a base class, implicitly use object

Method and attribute lookup begins in the derived class

    Proceeds down the chain of base classes

Derived methods override (shadow) base methods

    Like `virtual` in C++

Multiple Inheritance

"The Dreaded Diamond Pattern"

Multiple Inheritance

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

Multiple Inheritance

Base classes are separated by commas

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

Multiple Inheritance

Base classes are separated by commas

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

Order matters!

Attribute Resolution

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

- Officially, "C3 superclass linearization"
  - More info on Wikipedia!

Rarely useful

Classes have a hidden method .mro()

- Shows linearization of base classes

Attribute Resolution In Action

```
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

Magic Methods

dunderbar

Magic Methods

Python uses __init__ to build classes

dunderbar

Magic Methods

Python uses __init__ to build classes

We can supply our own __init__ for customization

dunderbar

Magic Methods

Python uses __init__ to build classes

We can supply our own __init__ for customization

What else can we do? Can we make classes look like: dunderbar

Magic Methods

Python uses __init__ to build classes

We can supply our own __init__ for customization

What else can we do? Can we make classes look like:

dunderbar

iterators?

Magic Methods

Python uses __init__ to build classes

We can supply our own __init__ for customization

What else can we do? Can we make classes look like:

dunderbar

iterators?

sets? dictionaries?

Magic Methods

Python uses __init__ to build classes

We can supply our own __init__ for customization

What else can we do? Can we make classes look like:

dunderbar

iterators?

sets? dictionaries?

numbers?

Magic Methods

Python uses __init__ to build classes

We can supply our own __init__ for customization

What else can we do? Can we make classes look like:

dunderbar

iterators?

sets? dictionaries?

numbers?

comparables?

Some Magic Methods

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some Magic Methods

```
x = MagicClass()
```

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some Magic Methods

```
x = MagicClass()
y = MagicClass()
```

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some Magic Methods

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

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some Magic Methods

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

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some Magic Methods

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

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some Magic Methods

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

x < y    # => x.__lt__(y)
```

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some 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)
```

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some 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__()
```

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some 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__()
```

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some 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__()
```

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some 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)
```

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Some 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)
```

Suppose MagicClass implements all of these magic methods

And so many more

Link 1

Link 2

Link 3

Example

Example

```
class Point:
```

Example

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

Example

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

Example

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

Example

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

Example

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

    def rotate_90_CC(self):
```

Example

```
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
```

Example

```
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
```

Example

```
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):
```

Example

```
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)
```

Example

```
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)
```

Example

```
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):
```

Example

```
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)
```

Example

Example

```
o = Point()
```

Example

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

Example

```
o = Point()
print(o)          # Point(0, 0)
p1 = Point(3, 5)
```

Example

```
o = Point()
print(o)          # Point(0, 0)
p1 = Point(3, 5)
p2 = Point(4, 6)
```

Example

```
o = Point()
print(o)          # Point(0, 0)
p1 = Point(3, 5)
p2 = Point(4, 6)
print(p1, p2)     # Point(3, 5) Point(4, 6)
```

Example

```
o = Point()
print(o)          # Point(0, 0)
p1 = Point(3, 5)
p2 = Point(4, 6)
print(p1, p2)     # Point(3, 5) Point(4, 6)
p1.rotate_90_CC()
```

Example

```
o = Point()
print(o)          # Point(0, 0)
p1 = Point(3, 5)
p2 = Point(4, 6)
print(p1, p2)     # Point(3, 5) Point(4, 6)
p1.rotate_90_CC()
print(p1)         # Point(-5, 3)
```

Example

```
o = Point()
print(o)          # Point(0, 0)
p1 = Point(3, 5)
p2 = Point(4, 6)
print(p1, p2)     # Point(3, 5) Point(4, 6)
p1.rotate_90_CC()
print(p1)         # Point(-5, 3)
print(p1 + p2)    # Point(-1, 9)
```

Case Study:
Errors and Exceptions

Syntax Errors

"Errors before execution"

Syntax Errors

"Errors before execution"

```
>>>
```

Syntax Errors

"Errors before execution"

```
>>> while True print('Hello world')
```

Syntax Errors

"Errors before execution"

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

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"

Exceptions

"Errors during execution"

```
>>> 10 * (1/0)
```

Exceptions

"Errors during execution"

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

Exceptions

"Errors during execution"

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

>>> 4 + spam*3
```

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
```

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
```

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

And More

KeyboardInterrupt

UnboundLocalError

SystemExit

StopIteration

SyntaxError

ZeroDivisionError

AttributeError

KeyError

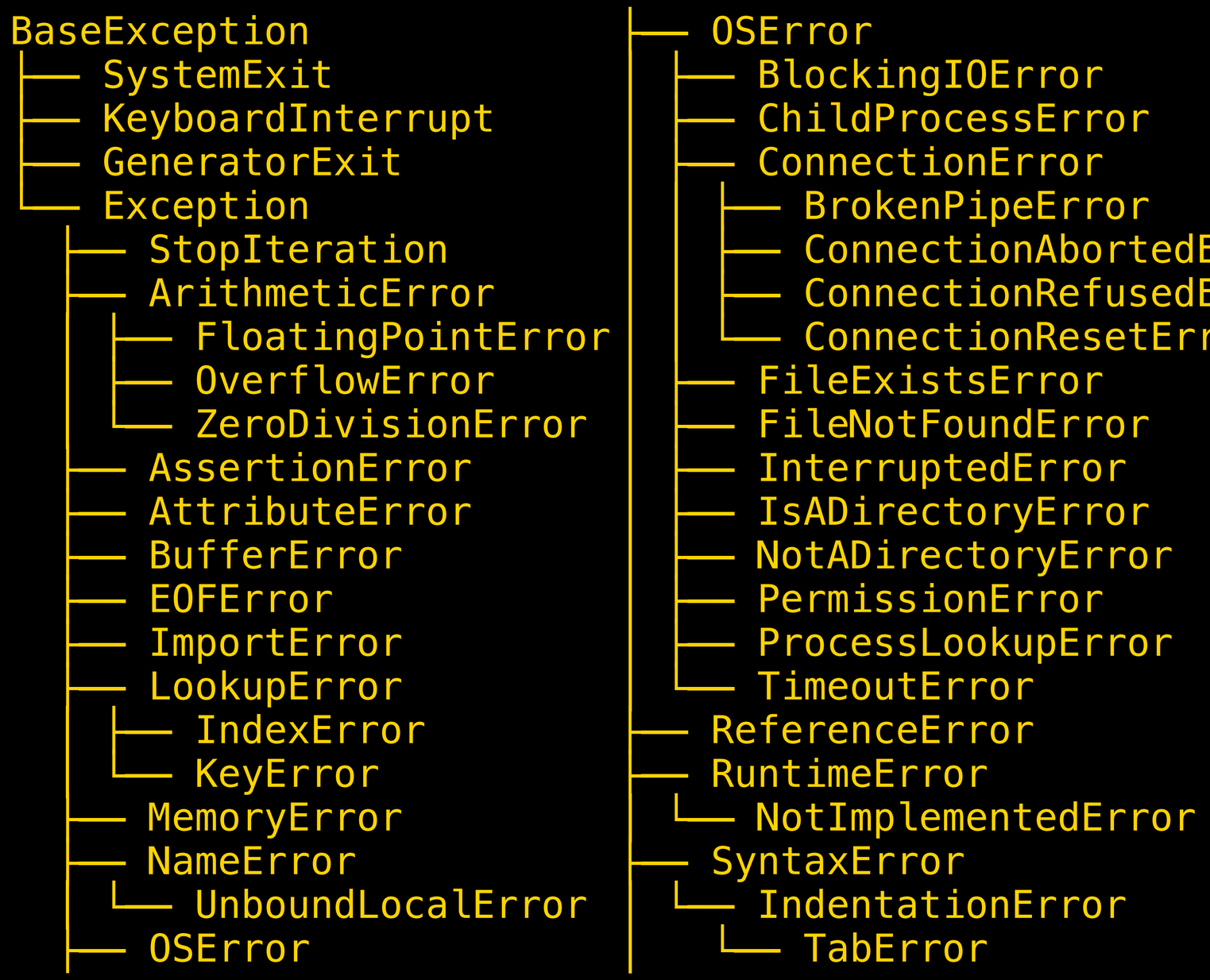
IndexError

NotImplementedError

TypeError

OSError

NameError



And Even More

Inheritance in Action!

Handling Exceptions

What's Wrong?

```
def read_int():
    """Reads an integer from the user (broken)"""
    return int(input("Please enter a number: "))
```

What's Wrong?

```
def read_int():
    """Reads an integer from the user (broken)"""
    return int(input("Please enter a number: "))
```

What happens if they enter a nonnumeric input?

Solution

Solution

```
def read_int():
```

Solution

```
def read_int():
    """Reads an integer from the user (fixed)"""
```

Solution

```
def read_int():
    """Reads an integer from the user (fixed)"""
    while True:
```

Solution

```
def read_int():
    """Reads an integer from the user (fixed)"""
    while True:
        try:
```

Solution

```
def read_int():
    """Reads an integer from the user (fixed)"""
    while True:
        try:
            x = int(input("Please enter a number: "))
```

Solution

```
def read_int():
    """Reads an integer from the user (fixed)"""
    while True:
        try:
            x = int(input("Please enter a number: "))
            break
```

Solution

```
def read_int():
    """Reads an integer from the user (fixed)"""
    while True:
        try:
            x = int(input("Please enter a number: "))
            break
        except ValueError:
```

Solution

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

Solution

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

Mechanics of `try` statement

Mechanics of `try` statement

1) Attempt to execute the try clause

Mechanics of `try` statement

1) Attempt to execute the try clause

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

Mechanics of `try` statement

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.

Mechanics of `try` statement

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 exception's type matches (/ is a subclass of) that named by except, then execute the except clause. Done!

Mechanics of `try` statement

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 exception's type matches (/ is a subclass of) that named by except, then execute the except clause. Done!

2bii) Otherwise, hand off the exception to any outer try statements. If unhandled, halt execution. Done!

Conveniences

Conveniences

```
try:
    distance = int(input("How far? "))
    time = car.speed / distance
    car.drive(time)
```

Conveniences

```
try:
    distance = int(input("How far? "))
    time = car.speed / distance
    car.drive(time)
except ValueError as e:
    print(e)
```

Bind a name to the exception instance

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!")
```

Bind a name to the exception instance

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")
```

Bind a name to the exception instance

Catch multiple exceptions

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?

```
def read_int():
    """Reads 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!"

Solution?

```
def read_int():
    """Reads 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

The `raise` keyword

```
>>> raise NameError('Why hello there!')
```

The `raise` keyword

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

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
```

The `raise` keyword

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

You can raise either instance objects or class objects

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

`raise` within `except` clause

`raise` within `except` clause

```
try:
    raise NotImplementedError("TODO")
except NotImplementedError:
    print('Looks like an exception to me!')
    raise
```

Re-raises the currently active exception

`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

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 an exception,
we might actually *want* to crash

Aside: Python Philosophy

Coding for the Common Case

Coding for the Common Case

Don't check if a file exists, then open it.

Coding for the Common Case

Don't check if a file exists, then open it.

Just try to open it!

Coding for the Common Case

Don't check if a file exists, then open it.

Just try to open it!

Handle exceptional cases with an except clause (or two)

Coding for the Common Case

Don't check if a file exists, then open it.

Just try to open it!

Handle exceptional cases with an except clause (or two)

(avoids race conditions too)

Coding for the Common Case

Don't check if a file exists, then open it.

Just try to open it!

Handle exceptional cases with an except clause (or two)

(avoids race conditions too)

Don't check if a queue is nonempty before popping

Coding for the Common Case

Don't check if a file exists, then open it.

- Just try to open it!
- Handle exceptional cases with an except clause (or two)
- *(avoids race conditions too)*

Don't check if a queue is nonempty before popping

- Just try to pop the element!

Coding for the Common Case

Don't check if a file exists, then open it.

- Just try to open it!
- Handle exceptional cases with an except clause (or two)
- *(avoids race conditions too)*

Don't check if a queue is nonempty before popping

- Just try to pop the element!

Good Python:
Custom Exceptions

Custom Exceptions

```
class BadLoginError(Exception):
    """Raised when a user attempts to
        login with an incorrect password"""
    pass
```

Don't misuse existing exceptions when the real error is something else!

You can also override the default behavior of __init__, which binds all positional arguments to an args attribute

Clean-Up Actions

The `finally` clause

The finally clause is always executed before leaving the try/... block

The `finally` clause

```
try:
```

The finally clause is always executed before leaving the try/... block

The `finally` clause

```
try:
    raise NotImplementedError
```

The finally clause is always executed before leaving the try/... block

The `finally` clause

```
try:
    raise NotImplementedError
finally:
```

The finally clause is always executed before leaving the try/... block

The `finally` clause

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

The finally clause is always executed before leaving the try/... block

The `finally` clause

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

The finally clause is always executed before leaving the try/... block

The `finally` clause

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

# Goodbye, world!
```

The finally clause is always executed before leaving the try/... block

The `finally` clause

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


# Goodbye, world!
# Traceback (most recent call last):
```

The finally clause is always executed before leaving the try/... block

The `finally` clause

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

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

The finally clause is always executed before leaving the try/... block

The `finally` clause

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


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

The finally clause is always executed before leaving the try/... block

How `finally` works

How `finally` works

Always executed before leaving the try statement.

How `finally` works

Always executed before leaving the try statement.

How `finally` works

Always executed before leaving the try statement.

Unhandled exceptions (not caught, or raised in except)

How `finally` works

Always executed before leaving the try statement.

Unhandled exceptions (not caught, or raised in except)

are re-raised after finally executes.

How `finally` works

Always executed before leaving the try statement.

Unhandled exceptions (not caught, or raised in except)

are re-raised after finally executes.

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)

Note: `with ... as ...`

Surprisingly useful and flexible!

Note: `with ... as ...`

```
# This is what enables us to use with ... as ...
with open(filename) as f:
    raw = f.read()
```

Surprisingly useful and flexible!

Note: `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!

The Road Ahead

Behind Us - The Python Language

Behind Us - The Python Language

Week 1 Python Fundamentals

Week 2 Data Structures

Week 3 Functions

Week 4 Functional Programming

Week 5 Object-Oriented Python

The Road Ahead - Python Tools

The Road Ahead - Python Tools

Week 6 Standard Library

Week 7 Third-Party Tools

Week 8 Ecosystem

Week 9 Advanced Topics

Week 10 Projects!

Next Time

Lab Time

Lab Time

Building Basic Classes

Lab Time

Building Basic Classes

Fun with Inheritance

Lab Time

Building Basic Classes

Fun with Inheritance

Magic Methods a.k.a. 007