# Object Oriented Programming

A Practical Course

# Michael Foord
# https://agileabstractions.com/

- Python trainer
- Python developer since 2002
- Core Python Developer
- Author of IronPython in Action
- Creator of “unittest.mock”
- Twitter: @voidspace

# Object Oriented Programming

- Why programming languages at all?
- Python
- Classes and objects in theory and practise
- Interfaces, encapsulation, operator overloading
- Inheritance, polymorphism and object theory

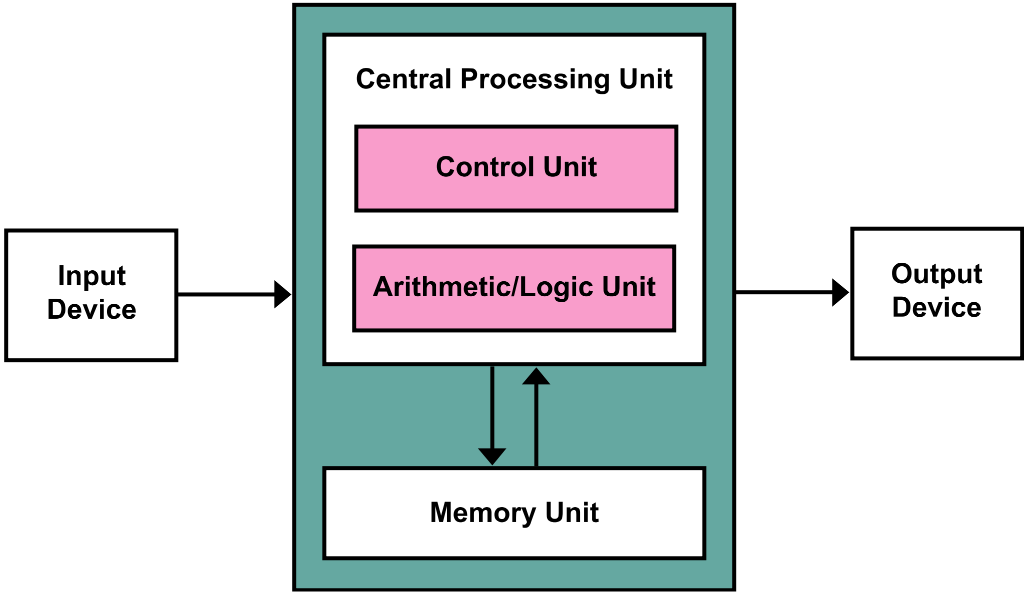
# And in French

- Programmation
- Orientée
- Objet
- Python

# Language

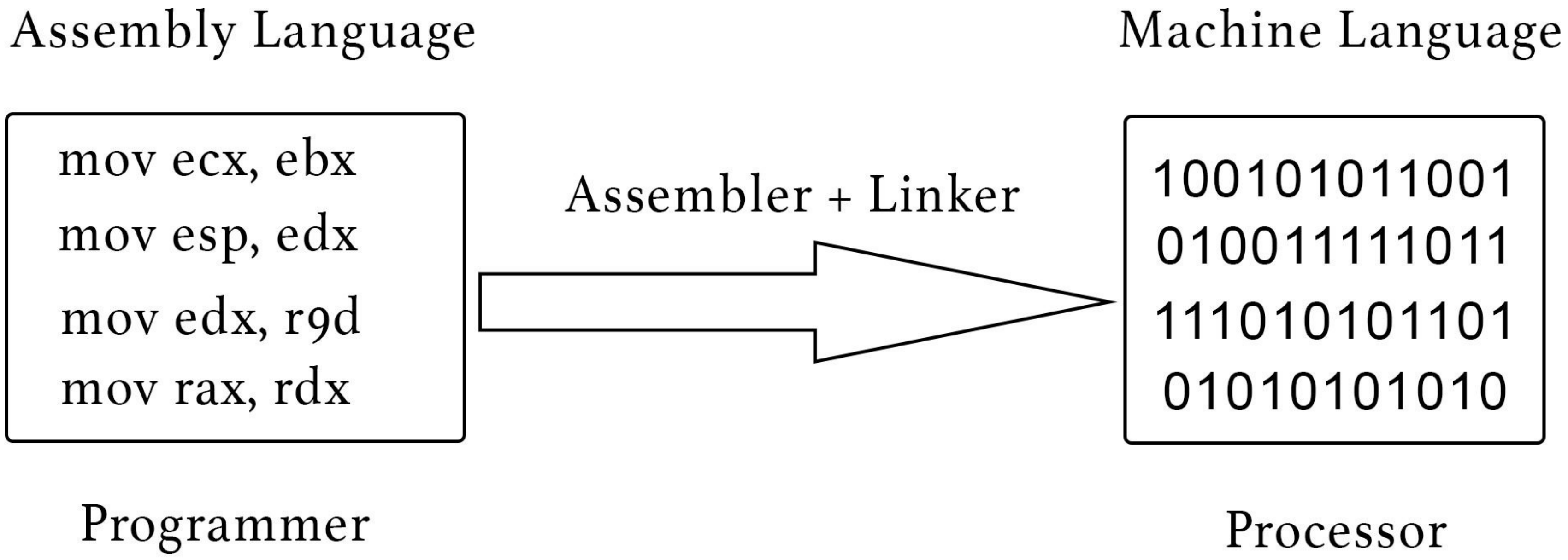
What language do computers speak?

# Von Neumann Architecture




Michael Foord 2018

# Assembly Language



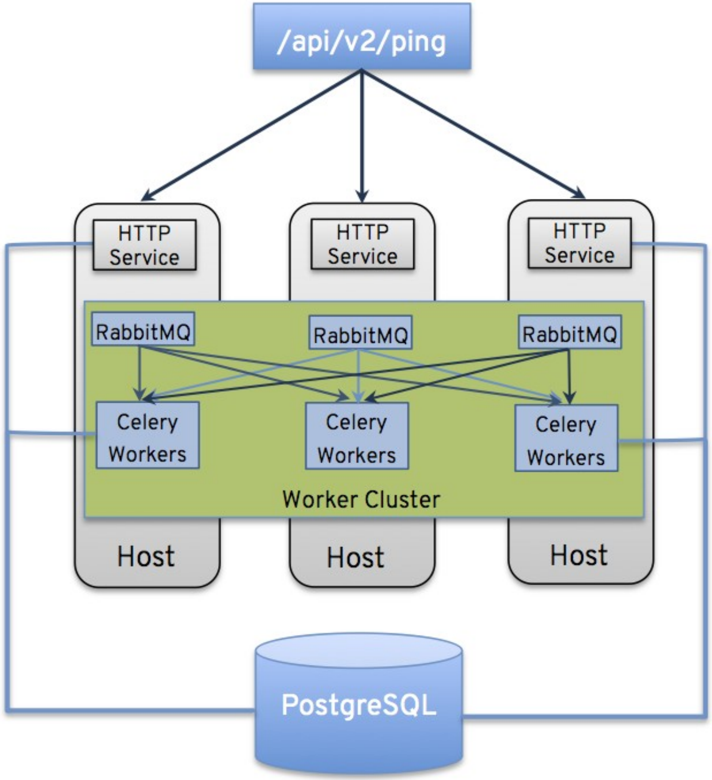
# Brief History of OOP

- First programming language with objects was Simula 67, from the sixties in Norway
- "Object Oriented Programming" was first used as a term for Smalltalk
- Became predominant in the eighties at least partly due to C++
- Now it's everywhere, but not uncontroversial...

# Why Objects?

- Objects are a concept, a way of representing and reasoning about system behaviour
- An object is code that represents an element or component of a program
- They easily correspond to how we think about the real world
  - Objects with properties and behaviour
  - So they provide convenient black boxes
  - An abstraction for thinking
- Reusable patterns for building things with!
- No single clear definition of "object oriented programming"

# Thinking about Systems

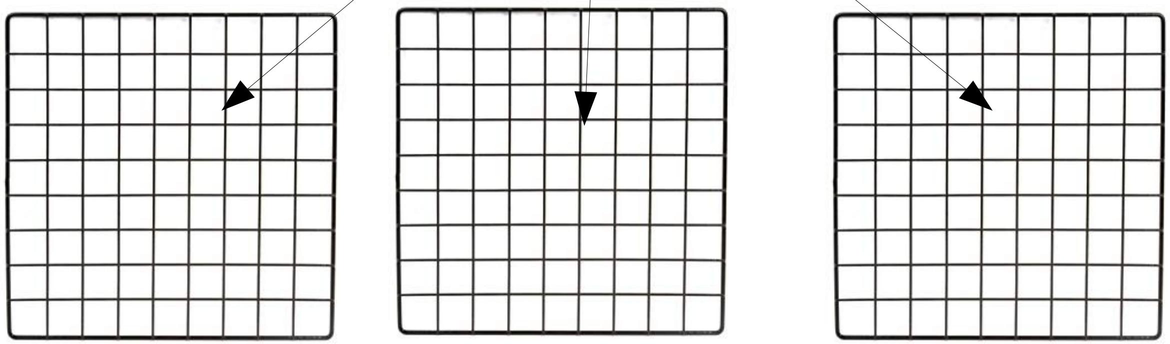



Ansible Tower clustering architecture.

# Objects for Structure

Spreadsheet class

One spreadsheet owns many worksheets
Worksheets could own rows, columns and cells



# No strict definition, but...

The three standard pillars of OOP

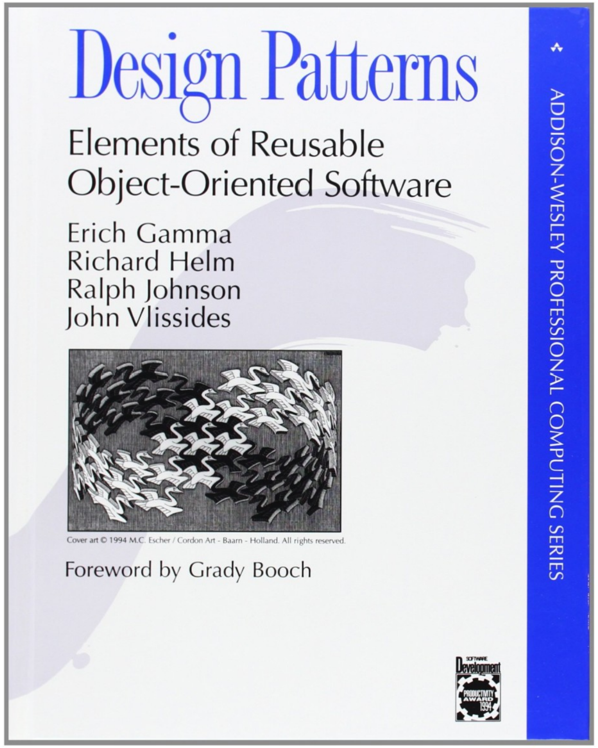
- Encapsulation
- Inheritance
- Polymorphism

The biggest practical advantage. Code reuse!

# Lots more terminology

- Instances, fields, members, attributes, methods
- Composition, delegation and duck typing
- Multiple inheritance and mixins
- Polymorphism, interfaces and APIs
- Encapsulation and data hiding

# Resuable Patterns




- Published 1994
- By the gang-of-four
- Enormously  influential
- Common patterns like "MVC" still the basis for many modern web applications
- But beware the singleton!

# Modern Languages

Object Oriented

- C++
- Java
- C#
- Python
- Ruby
- Javascript

Not Object Oriented

- C
- Go (sort of)
- PHP (sort of)

# Objects Inside and Out

- To really understand OOP you need to know how to use objects, what they represent
- But you also need to know what they are, how they work, from the inside
- It's just code
- The law of leaky abstractions

# The Class

- The core concept in OOP (we never mention prototypes)
- The blueprint for objects
- Every object has a type (synonym for class)

```
class Something:
    def say_something(self):
        print("something")
```

# Objects Themselves

- Created from the blueprint
- Usually called "instances". An object is an instance of its class
- Creating a new object is called "instantiation"

```
thing = Something()
```

# Python

## Just enough to be dangerous

Michael Foord 2018

# Why Python?

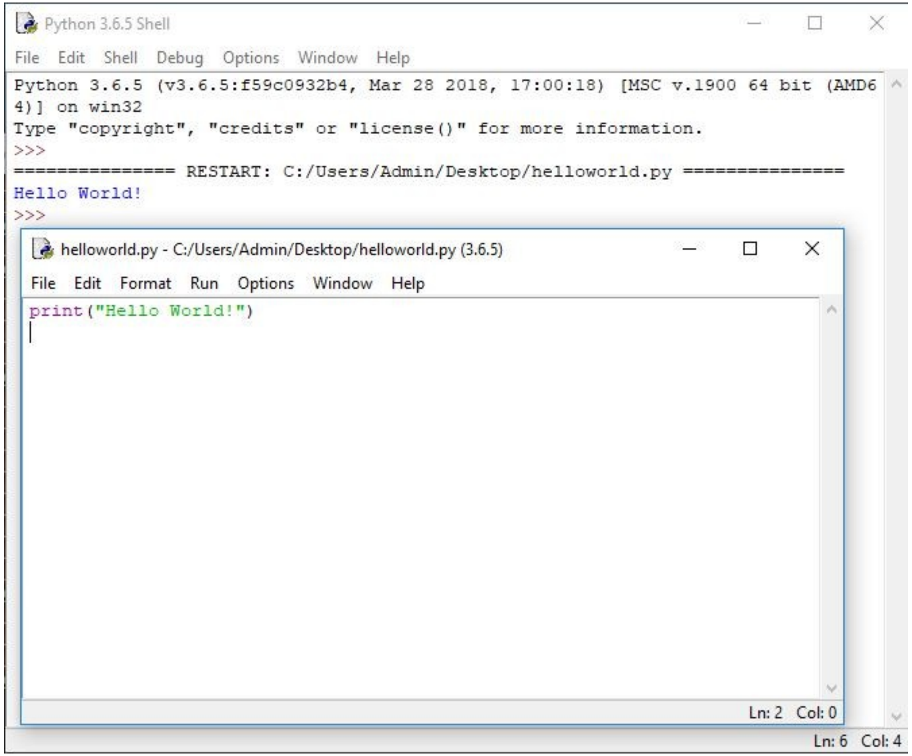
- Python is fully objected oriented (everything is an object)
- Widely used, one of the most popular programming languages commercially and in education
- Very easy to learn
- Named after Monty Python

# Dynamically Typed

- Everything happens at "runtime"
- You don't need to declare types
- No compile time checks
- Makes Python much easier to learn and more flexible
- But makes testing your code more important!

# Being IDLE

- IDLE is a code editor
- Bundled with Python
- Basic but functional
- Includes an interactive interpreter



# The Interactive Interpreter

Command Prompt - python

```
(webscraping) C:\Users\Admin\training-webscraping>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 1.7 * 42 - 284 + (37 * 12)
231.39999999999998
>>> print("Hello World!")
Hello World!
>>> for x in range(5):
...     string = "This is number {}".format(x)
...     print(string)
...
This is number 0
This is number 1
This is number 2
This is number 3
This is number 4
>>>
```

One of the most powerful features of the Python programming language! The REPL, or interactive interpreter.

# Code Examples

Many of the examples throughout this course, including the exercises, will be formatted using the same style as the interactive interpreter. That means you should be able to type them into the interpreter (either at the command line or in IDLE) and see the results.

```
>>> print(“Hello World!”)
Hello World!
>>> for x in range(5):
...   string = "This is number {}".format(x)
...   print(string)
...
This is number 0
This is number 1
This is number 2
This is number 3
This is number 4
>>>
```

# help and dir

There are two special built-in functions that are particularly useful for exploring inside the interactive interpreter. `help()` and `dir()`

```
>>> help(list)
Help on class list in module builtins:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's
items
 |
 |  Methods defined here:
>>> dir(list)
['append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

# A Comment on Comments

Python has comments. Any line that starts with a # is a comment. Or anything after a # is a comment.

```
# This is a comment
x = 3 # and so is this
```

# Variables

Perhaps the most basic element of a program is the variables. How we store data and do anything.

We create variables by assigning to a name. Having given something a name we can do things with it. Giving something a name is called assignment. Variables have a name and a value, and that value has a type.

```
>>> x = 3 # integer

>>> y = "Some text" # string

>>> z = 4.2 # float

>>> print(x, y, z)

3 Some text 4.2
```

# Case Sensitive

Programming languages are very sensitive. They’re unforgiving of spelling mistakes and typos. Python is also case sensitive. This applies to variables, function names, and just about everything else.

```
>>> x = 3
>>> print(X)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'X' is not defined
>>> PRINT(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'PRINT' is not defined
```

# Comparisons

One of the things we can do with variables is compare them. We use the “comparison operators” to do this. The basic ones are obvious. There are also some you might not have seen before for testing equality (are these things equal) or inequality.

```
>>> x = 3
>>> x < 5 # less than
>>> x > 2 # greater than
>>> x == 3 # equal to
>>> x != 3 # not equal to
>>> x <= 3 # less than or equal to
>>> x >= 3 # greater than or equal to
```

# The Booleans: True and False

When we try the comparison operations at the command line we see Python tells us whether the comparison is True or False.

```
>>> x = 3
>>> x < 5
True
```

True and False are special values, they're called the boolean types. We say an operation "evaluates" to either True or False. We can use them in our code too.

# Block Structure and Indentation

We’re now moving onto our first real language construct. So far we’ve been using expressions (other than assignment to create variables). Checking if a condition is true or not, a “conditional”, uses the “if statement”. That requires a block of code to execute if the condition is True. In Python blocks start with a line with a “:” and are then indented. It’s normal to use four spaces for indentation. Anything indented is inside the block.

```
x = 3

if x == 3:

    print(“x is 3”)

print(“This is not inside the if statement”)
```

# A Simple Program

```
total = 0
days = 0
distance_per_day = 10
distance_to_the_moon = 238900 # in miles
total_run = 0

while total_run < distance_to_the_moon:
    total_run = total_run + distance_per_day
    days = days + 1
    distance_per_day = distance_per_day * 1.1 # every day we run further

print(days)
print(total_run)
```

# Exercise 0

Time: 10 minutes

# Different "classes" of Objects

- Structural
- Behavioural (algorithms etc)
- Collections (of other objects)
- As data (primitives)

But objects wrapping up data with methods for working with that data is fundamental to object orientation.

# Primitive Datatypes

- Numbers (integers and floating point)
- Strings (text)
- Booleans (True and False)
- None (the "null" value)
- In Python all the primitives are objects, not true in all languages (where they're "values" not objects)

# Strings and String Methods

```
>>> string = "foo"
>>> string.upper()
'FOO'
>>> type(string)
<type 'str'>
>>> dir(string)
['__add__', '__class__', ...
```

# More Useful Methods

- `s.strip()`
- `s.lower() / s.upper()`
- `s.replace(“hello”, “goodbye”)`
- `s.startswith(‘foo’)`
- `‘, ‘.join([“foo”, “bar”, “baz”])`

# Escape Codes

- We use the “\”, the backslash, for escaping special characters
- A geek standard, comes from the old days of Unix.
- It means the “\” is special, so using Windows file paths in strings takes care!

`'\n'` A newline

`'\t'` A Tab

`'\''` A literal single quote

`'\"'` A literal double quote

`'\\'` A literal backslash

# Converting Strings

- Sometimes we need to create strings
- We use the type object "`str`"
- Works on any object

```
>>> x = 3.2
>>> str(x)
'3.2'
```

# String Formatting

- Sometimes we need to print or create strings containing bits of data from somewhere else
- We can do this with string formatting
- A big topic, uses the “`format`” method of strings

```
>>> string = "My name is {}. My age is {}."
>>> print(string.format("Michael", 44))
My name is Michael. My age is 44.
```

# Input, Output: print and input

- You’ve already seen it, but we output data to the screen with the `print` function
- We can also ask the user for input with the `input` function

```
>>> result = input("What is your name? ")
What is your name? Michael
>>> print(result)
Michael
```

# Integers

- One of the two basic number types
- They can be positive or negative
- Only whole numbers
- Any size (no maximum value)
- Written as you would expect:
  - `1, 2, 3, +6, -7, -888, 1987676477479432288888`

# Floating Point Numbers

- Represent any number
- An inexact representation
- Python uses the industry standard IEEE 754
- All the standard maths operations work
- Written in a couple of different ways (relevant for very big or very small numbers)

    `-4.7`, `7e8`, `-1.8256e-37`

# Exercise 1

Right, now you know enough to try some.

Time: 10 minutes

# Mutability

- A core computer science concept
- The primitive types in Python are immutable
- This makes them safe to pass around
- Collections (coming next) are generally mutable
- As are user created types

# Collections and Containers

- The basis of data-structures (i.e. just about everything useful a program does)
- You can write your own
- In Python the most common ones are the list and the dictionary

# Nested Data Structures

- Arbitrarily complicated
- Form an "object graph" or "object tree"

```
data = {
    "coordinates": [(0.5, 6), (-0.3, 3.2), (1.1, 0)],
    "people": {
        "james": {"address": ["26 Broadacre", "Newton", "PE2 
4XZ"]},
        "sally": {"address": ["37 Downtown", "Newton", "PE2 
4XZ"]},
    },
    "names": ["James", "Sally"],
}
```

# The List

- For working with more than one piece of data at a time we use containers
- The basic one is the list
- Very powerful and flexible
- Written with square brackets
- Can contain anything

```
values = [100, 478, 96, 32, 80]

words = ["fish", "eggs", "ham"]
```

# Adding to a list

- We add new items to a list with the `append` method
- We can insert lists at a specific position with `insert`
- We can extend a list with the `extend` method or adding another list

```
our_list = []
x = 0
while x < 100:
    if x % 2 != 0:
        our_list.append(x)
```

# Indexing

- We can pull out a value from a specific position by indexing
- We can use the same technique to change a value
- And also to delete one

```
my_list = ["James", "John", "Jack", "Jill"]
first_name = my_list[0]
my_list[0] = "Rupert"
del my_list[0]
```

# What’s my length? What’s in me?

- Often you need to know the length of a list
- For this we use the built-in function `len`
- To check if a list contains a value use the `in` operator

```
>>> my_list = ["Jack", "John", "Jill", "James"]
>>> len(my_list)
4
>>> "Jack" in my_list
True
>>> "Michael" in my_list
False
```

# Dictionaries

- Dictionaries are another type of container
- Most of Python is built on dictionaries!
- They store values with a key
- Anything “immutable” can be a key
- Use a list for anything you want to process one at a time
- Use a dictionary where you need to be able to fetch values out easily (a value is stored with a ‘name’)

# Basic Operations

The syntax is very similar to using a list except with keys instead of indices:

- Create a dictionary

```
>>> my_dict = {'key': 'value'}
```

- Fetch a value with the key:

```
>>> my_dict[‘key’]
‘value’
```

- Change a value or create a new one

```
>>> my_dict[‘key’] = ‘new value’
>>> my_dict[‘new_key’] = ‘a different value’
```

- Delete a value

```
>>> del my_dict[‘key’]
```

# Exercise 2

Time: 15 minutes

# Functions

To understand classes we need to understand methods, so we need to understand functions.

A function is a component of programming that "does something". One of the fundamental ways of structuring programs.

They allow code to be reused.

# The Basic Building Block

- Functions are resusable blocks of code
- They’re defined with the “`def`” keyword
- They can take arguments and return values
- A function is just a sequence of statements indented inside the def keyword
- They’re called with parentheses:

```
result = function(arg1, arg2)
```

# An Example

```
def build_list(maximum_value, step):
    new_list = []
    value = 0
    while value <= maximum_value:
        new_list.append(value)
        value += step
    return new_list
```

# Local and Global Variables

- Variables inside a function are “local”
- Variables outside a function are “global”
- Variables inside a function can’t be seen from the outside
- This is a computer science concept called “scope”

# Scope

```
>>> x = 3
>>> def function():
...     x = 4
...     print(x)
...
>>> print(x)
3
>>> function()
4
>>> print(x)
3
```

# Exercise 3

Time: 10 minutes

# Exceptions

- What happens when things go wrong?

```
>>> 375 + "this really isn't a number"
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# Handling Exceptions

- Catch any exception:

```
try:
    print(10 + "this isn't a number")
except:
    print("We're ignoring this error")
```

# Handling Exceptions

- Or a specific exception:

```
try:
    with open("foo.txt") as handle:
        data = handle.read()
except FileNotFoundError:
    print("Missing foo.txt")
    data = ""
```

# Raising Exceptions

- And we can raise them

```
if value > max_value:
    raise Exception("Value is too high")
```

# Catching All Errors

- Never catch all exceptions unless you report/ record the actual exception that occurred

```
try:
    # Some complicated operation
    ...
except Exception as e:
    print("Sorry, it didn't work.")
    print("Reason:", e)
    ...
```

- Not reporting actual exception information is the fastest way to create undebuggable code

# Ignoring Errors

- No! No! No!

```
try:
    # Some complicated operation
    ...
except Exception:
    pass
```

- Argh!!!! Boom!

```
try:
    # Some complicated operation
    ...
except Exception:
    # !! TODO
    pass
```

- Catastrophic failures are often a result of exception handling gone terribly wrong.

Ariane 5

# Reraising Exceptions

- Log/re-raise

```
try:
    # Some complicated operation
    ...
except Exception as e:
    print("Sorry, it didn't work.")
    print("Reason:", e)
    raise
```

- Useful if you want to do something with the exception, but allow it to propagate

# Exercise 4

Time: 10 minutes

# Procedural Programming

- Prior to OOP was procedural programming (still around in languages like C)
- Data and ways of working with them were separate

```
x = "Some string"
def to_upper(string):
    ...
    return new_string

>>> to_upper(x)
"SOME STRING"
```

# Types

- With objects behaviour can be bundled with data
- The behaviour is defined on the "type" of the object

```
>>> x = "Some string"
>>> type(x)
<type 'str'>
>>> str.upper
<method 'upper' of 'str' objects>
>>> x.upper()
"SOME STRING"
```

# User defined types: class

In Python new types, new classes are created with the `class` statement.

```
class SomeNewType:
    def say_hello(self):
        print("Hello")
>>> thing = SomeNewType()
>>> thing.say_hello()
Hello
```

# Classes and Instances

- The class defines the blueprint
- An instance is one particular object made from that blueprint
- The classic example is that "cow" is the class whereas "buttercup" and "daisy" are instances of the cow class...

# Methods

Methods are created as functions defined inside the body of the class. They use a special argument called "`self`" to refer to the current instance. Sometimes called "`this`" in other languages.

# Object Operations

- There are only three basic operations on an object
  - Getting an attribute
  - Setting an attribute
  - Deleting an attribute

# Attribute Access

- These functions may be used to manipulate attributes given an attribute name string

```
getattr(obj, 'name')            # Same as obj.name
setattr(obj, 'name', value)     # Same as obj.name = value
delattr(obj, 'name')            # Same as del obj.name
hasattr(obj, 'name')            # Tests if attribute exists
```

- Example: Probing for an optional attribute

```
if hasattr(obj, 'x'):
   x = getattr(obj, 'x'):
else:
   x = None
```

- Note: getattr() has a useful default value arg

```
x = getattr(obj, 'x', None)
```

# Method Lookup

Calling a method is a two step operation. First look-up the method with the "dot operator" and then call the method, the "()" operator. Methods follow the normal rules of attribute lookup.

```
class SomeNewType:
    def say_hello(self):
        print("Hello")
>>> thing = SomeNewType()
>>> thing.say_hello
<bound method SomeNewType.say_hello of
<__main__.SomeNewType object at ...>>
```

# Shared Attributes

- Class can have attributes as well as methods (fields or members in other languages)

```
class Thing:
    shared = "A class attribute"

>>> thing1 = Thing()
>>> thing1.shared
'A class attribute'
>>> thing2 = Thing()
>>> thing1 == thing2
False
>>> thing2.shared
'A class attribute'
```

# Exercise 5

Time: 10 minutes

# Core Concepts

- Primitive types, collections and user defined types
- Mutable and immutable objects
- Classes and instances
- Methods and instance attributes
- Class attributes (shared attributes)

# Encapsulation

- Two schools of thought
  - Encapsulation means data hiding
  - Encapsulation is wrapping up data and ways of working with it as a single object

# Private Data

```
class A{
    private int data=40;
    private void msg(){System.out.println("Hello java");}
}

public class Simple{
    public static void main(String args[]){
        A obj =new A();
       System.out.println(obj.data);    //Compile Time Error
       obj.msg();     //Compile Time Error
     }
}
```

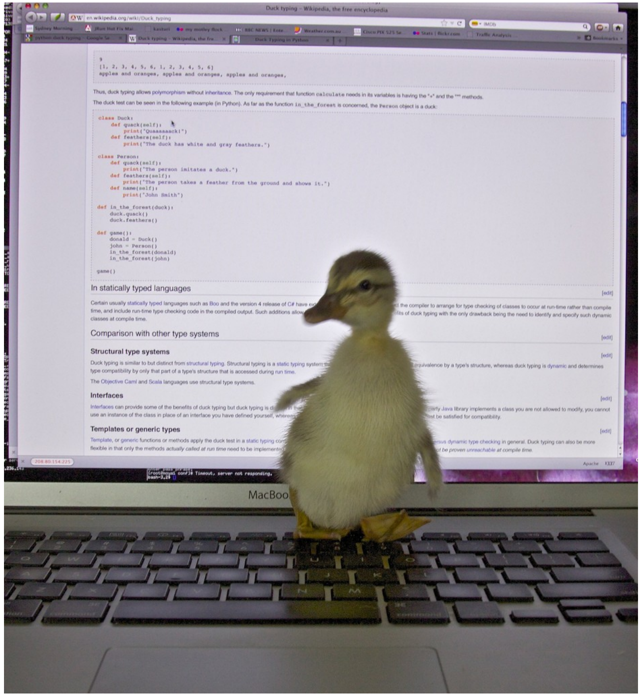
# Private by Convention

- In Python we can use "properties" to protect access
- But by default everything is open
- We use an underscore to mark data or methods as private
- A "translucent" encapsulation

# Exercise 6

Time: 15 minutes

# Duck Typing


In statically typed languages
Comparison with other type systems
Structural type systems
Interfaces
Templates or generic types

# Programming to an Interface

- Define the public API you expect
- Concerned with behaviour not type
- If it quacks like a duck and walks like a duck…
- Interface is an overloaded term (has a formal meaning in many languages)

# Program to Behaviour not type

- For example, `json.load` works with any “file like object”

```
with open(filename) as h:
    data = json.load(h)

data = json.load(StringIO(...))
```

# Interfaces in Go

```
type geometry interface {
    area() float64
    perim() float64
}
```

# Interfaces in Go

```
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}
```

# `twisted.internet.interfaces.ITransport(Interface)` interface documentation

Part of `twisted.internet.interfaces` View Source (View In Hierarchy)

Known subclasses: `twisted.conch.insults.insults.ITerminalTransport`, `twisted.conch.telnet.ITelnetTransport`, `twisted.internet.interfaces.IProcessTransport`, `twisted.internet.interfaces.ITCPTransport`, `twisted.internet.interfaces.IUNIXTransport`

Known implementations: `twisted.conch.ssh.channel.SSHChannel`, `twisted.conch.ssh.session.SSHSessionProcessProtocol`, `twisted.internet._win32stdio.StandardIO`, `twisted.internet.abstract.FileDescriptor`, `twisted.internet.iocpreactor.abstract.FileHandle`, `twisted.internet.protocol.FileWrapper`, `twisted.internet.stdio.StandardIO`, `twisted.protocols.loopback._LoopbackTransport`, `twisted.protocols.loopback.LoopbackRelay`, `twisted.test.proto_helpers.StringTransport`, `twisted.trial._dist.worker.LocalWorkerTransport`, `twisted.web._http2.H2Stream`, `twisted.web.http.HTTPChannel`

I am a transport for bytes.

I represent (and wrap) the physical connection and synchronicity of the framework which is talking to the network. I make no representations about whether calls to me will happen immediately or require returning to a control loop, or whether they will happen in the same or another thread. Consider methods of this class (aside from getPeer) to be 'thrown over the wall', to happen at some indeterminate time.

| | | |
|---|---|---|
| Method | `write` | Write some data to the physical connection, in sequence, in a non-blocking fashion. |
| Method | `writeSequence` | Write an iterable of byte strings to the physical connection. |
| Method | `loseConnection` | Close my connection, after writing all pending data. |
| Method | `getPeer` | Get the remote address of this connection. |
| Method | `getHost` | Similar to getPeer, but returns an address describing this side of the connection. |

# Python ABCs

- Python “recently” acquired formal interfaces
- The “abc” module, “Abstract Base Classes”
- Powerful but not widely used
- Extends the type system!
- More on these later...

# Exercise 7

Time: 15 minutes

# Operator Overloading

- Python protocols, the magic methods
- Effectively interfaces
- Examples include:
  - The numeric protocol
  - Comparisons
  - The container protocol

# Methods: Mathematics

- Mathematical operators

```
a + b        a.__add__(b)
a - b        a.__sub__(b)
a * b        a.__mul__(b)
a / b        a.__div__(b)
a // b       a.__floordiv__(b)
a % b        a.__mod__(b)
a << b       a.__lshift__(b)
a >> b       a.__rshift__(b)
a & b        a.__and__(b)
a | b        a.__or__(b)
a ^ b        a.__xor__(b)
a ** b       a.__pow__(b)
-a           a.__neg__()
~a           a.__invert__()
abs(a)       a.__abs__()
```

- Consult reference for further details

# String Representation

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __repr__(self):
        return 'Date({!r}, {!r}, {!r})'.format(self.year,
                         self.month, self.day)

    def __str__(self):
        return '{}/{}/{}'.format(self.day, self.month, self.year)
```

# The Container Protocol

```
class Container:
    def __init__(self):
        self._store = {}

    def __getitem__(self, name):
        return self._store[name]

    def __setitem__(self, name, value):
        self._store[name] = value

    def __delitem__(self, name):
        del self._store[name]
```

# Exercise 8

Time: 20 minutes

# Stepping Back: Understanding Assignment

- Many operations in Python are related to "assigning" or "storing" values

```
a = value                   # Assignment to a variable
s[n] = value                # Assignment to an list
s.append(value)             # Appending to a list
d['key'] = value            # Adding to a dictionary
```

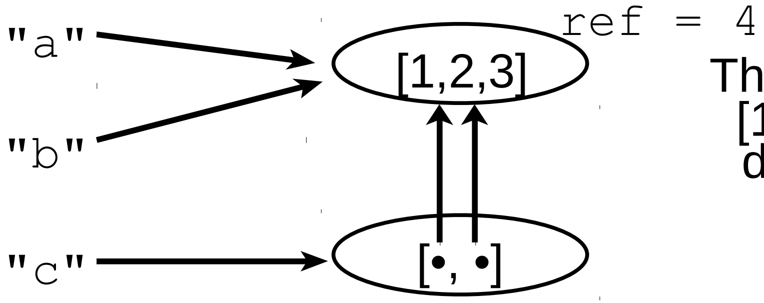
- A caution : assignment operations never make a copy of the value being assigned
- All assignments are merely reference Copies (or pointer copies if you prefer)

# Assignment Example

- Consider this code fragment:

```
a = [1,2,3]
b = a
c = [a,b]
```

- A picture of the underlying memory




There is only one list object [1,2,3], but there are four different references to it.

# Assignment Caution

- Modifying a value affects all references

```
>>> a.append(999)
>>> a
[1,2,3,999]
>>> b
[1,2,3,999]
>>> c
[[1,2,3,999], [1,2,3,999]]
>>>
```

- Notice how a change to the original list shows up everywhere else (yikes!)
- This is because no copies were made everything is pointing at the same thing

# Call by Object

- Objects are never copied on function call

```
def func(items):
    items.append(42)

>>> a = [1,2,3]
>>> func(a)
>>> a
[1, 2, 3, 42]
>>>
```

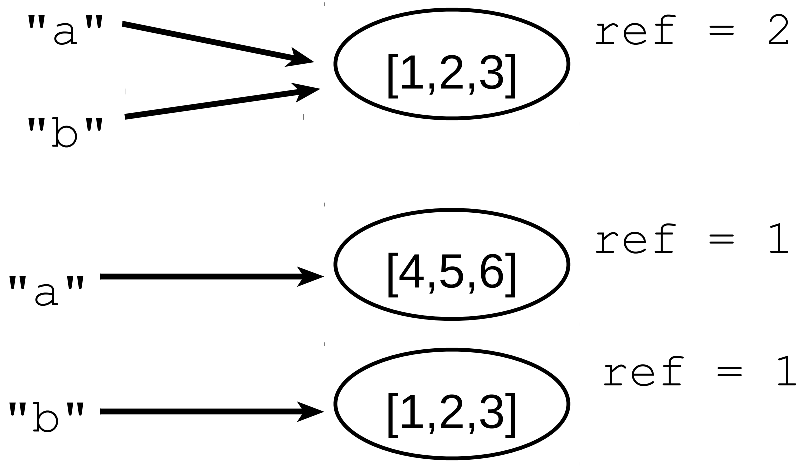
- Mutations affect the original object
- Reminder: many objects are immutable

# Reassigning Names

- Reassigning a name never overwrites the memory used by the previous value

```
a = [1,2,3]
b = a
```

```
a = [4,5,6]
```




- The name now refers to a different object

# Identity and References

- Use the "is" operator to check if two values are exactly the same in memory

```
>>> a = [1,2,3]
>>> b = a
>>> a is b
True
>>>
```

- Every object also has an integer identifier

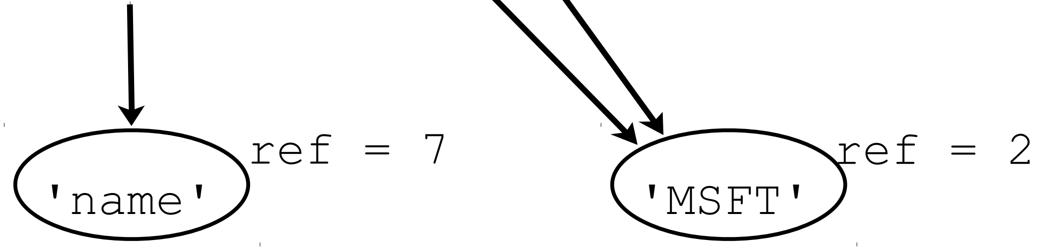
```
>>> id(a)
2774760
>>> id(b)
2774760
>>>
```

The object identifier is kind of like a pointer. If two names have the same id value, they're referring to the same object.

# Exploiting Immutability

- Immutable values can be safely shared

```
portfolio = [
   {'name': 'AA', 'price': 32.2, 'shares': 100},
   {'name': 'IBM', 'price': 91.1, 'shares': 50},
   {'name': 'CAT', 'price': 83.44, 'shares': 150},
   {'name': 'MSFT', 'price': 51.23, 'shares': 200},
   {'name': 'GE', 'price': 40.37, 'shares': 95},
   {'name': 'MSFT', 'price': 65.1, 'shares': 50},
   {'name': 'IBM', 'price': 70.44, 'shares': 100}
]
```




- Sharing can save significant memory

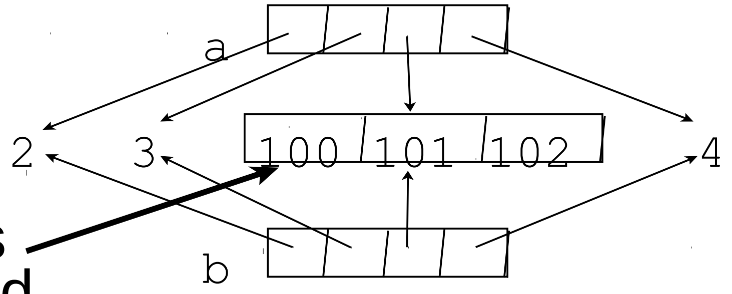
# Shallow Copies

- Containers have methods for copying

```
>>> a = [2,3,[100,101],4]
>>> b = list(a)                    # Make a copy
>>> a is b
False
```

- However, items are copied by reference

```
>>> a[2].append(102)
>>> b[2]
[100,101,102]
>>>
```




This inner list is
still being shared

- Known as a "shallow copy"

# Deep Copying

- Sometimes you need to makes a copy of an object and all objects it contains
- Use the copy module

```
>>> a = [2,3,[100,101],4]
>>> import copy
>>> b = copy.deepcopy(a)
>>> a[2].append(102)
>>> b[2]
[100,101]
>>>
```

- This is the only safe way to copy

# Everything is an object

- Numbers, strings, lists, functions, exceptions, classes, instances, etc...
- All objects are said to be "first-class"
- Meaning:  All objects that can be named Can be passed around as data, placed in containers, etc., without any restrictions.
- There are no "special" kinds of objects

# Example: Emulating Cases

A big conditional with many cases

```
if op == '+':
    r = add(x, y)
elif op == '-':
    r = sub(x, y):
elif op == '*':
    r = mul(x, y):
elif op == '/':
    r = div(x, y):
```

Reformulation using a dict of functions

```
ops = {
   '+' : add,
   '-' : sub,
   '*' : mul,
   '/' : div
}

r = ops[op](x,y)
```

- Key idea: Can make data structures from anything.

# Namespaces

- Assignment (often) creates names
- Those names are stored so they can be looked up
- The name "refers to" (is a reference to) the value
- Where they are stored is a "namespace"
- Namespaces are a core idea in Python

# Exercise 9

Time: 5 minutes

# Inheritance

- A tool for specializing existing objects

```
class Parent(object):
    ...

class Child(Parent):
    ...
```

- New class called a subclass
- Parent known as base class or superclass
- Parent is specified in () after class name

# Inheritance

- What do you mean by "specialize?"
- Take an existing class and ...
    - Add new methods
    - Redefine existing methods
    - Add new attributes to instances
- In a nutshell: Extending existing code

# Inheritance Example

- Adding a new method

```
class MyStock(Stock):
    def panic(self):
        self.sell(self.shares)

>>> s = MyStock('GOOG', 100, 490.1)
>>> s.sell(25)
>>> s.shares
75
>>> s.panic()
>>> s.shares
0
>>>
```

- You can give new capabilities to existing objects

# Inheritance Example

- Redefining a method

```
class MyStock(Stock):
    def cost(self):
        return 1.25 * self.shares * self.price

>>> s = MyStock('GOOG', 100, 490.1)
>>> s.cost()
61262.5
>>>
```

- The new method replaces the old one
- Other methods are unaffected

# Inheritance and Overriding

- Sometimes a class extends an existing method, but it has to use the original implementation

```
class Stock(object):
    ...
    def cost(self):
        return self.shares * self.price
    ...
class MyStock(Stock):
    def cost(self):
        actual_cost = super().cost()
        return 1.25 * actual_cost
```

- Use super() to call previous version
- Caution: Python 2 is different

```
actual_cost = super(MyStock, self).cost()
```

# Inheritance and __init__

- With inheritance, you must initialize parents

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

class MyStock(Stock):
    def __init__(self, name, shares, price, factor):
        super().__init__(name, shares, price)
        self.factor = factor
    def cost(self):
        return self.factor * super().cost()
```

- Again, you should use super() as shown

# "is a" relationship

- Inheritance establishes a type relationship

```
class Stock(object):
    ...

class MyStock(Stock):
    ...

>>> s = MyStock('ACME', 50, 91.1)
>>> isinstance(s, Stock)
True
>>>
```

- Important: objects defined via inheritance are a special version of the parent (same capabilities)

# Using Inheritance

- Sometimes used to organize related objects

```
class Shape(object):
    ...

class Circle(Shape):
    ...

class Rectangle(Shape):
    ...
```

- Think logical hierarchy or taxonomy

# Liskov Substitution Principle

*Anywhere you can use a parent class you should be able to substitute a derived class and everything should still work.*

The child class should provide the same API and guarantees (e.g. exceptions raised) as the parent class.

This is polymorphism. Python has polymorphism plus duck-typing.

*The Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called (strong) behavioral subtyping, that was initially introduced by Barbara Liskov in a 1987 conference keynote address titled Data abstraction and hierarchy.*

# "is a" in Practise

`unittest` test discovery inspects all the test files it can find and collects every subclass of `TestCase` and runs the tests in them.

```
class MyTestCase(unittest.TestCase):

    def test_something(self):
        self.assertEqual(2, 3)
```

# object base class

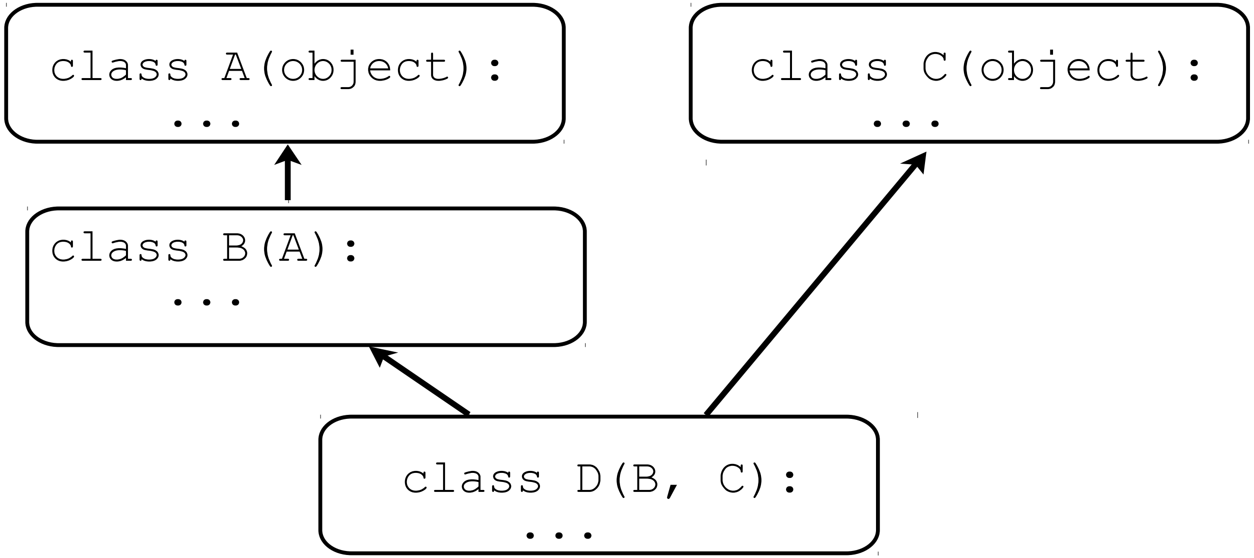
- If a class has no parent, use object as base

```
class Stock(object):
    ...
```

- object is the parent of all objects in Python (even if you don't specify it in Python 3)
- Note: There is some historical baggage with Python 2. Inheriting from object is required to get a "new-style" class

# Multiple Inheritance

- Classes can have multiple parents




- The child will inherit features from all parents
- But, it's a lot sneakier than this

# Cooperative Inheritance

- Python uses "cooperative multiple inheritance"
- Big idea: A child class can specifically arrange its parents to cooperate with each other

```
class Child(Parent1, Parent2, Parent3):
    ...
```

- The order of the parents has significance
- Attribute search may jump parent-to-parent

```
class Child(Parent1, Parent2, Parent3):
    ...
```

# Cooperative Inheritance

- Example: Consider this arrangement

```
class Parent(object):
    def spam(self):
        print('Parent')
```

```
class A(Parent):
    def spam(self):
        print('A')
        super().spam()
```

```
class B(Parent):
    def spam(self):
        print('B')
        super().spam()
```

- Now, this:

```
class Child(A,B):
    pass
```

```
>>> c = Child()
>>> c.spam()
A
B
Parent
>>>
```

it's gone sideways!

# Cooperative Inheritance

- There are applications

- You can make collections of classes that are meant to be stacked together to make more interesting things

# An Odd Code Reuse

```
class Dog(object):
    def noise(self):
        return 'Woof'

    def chase(self):
        return 'Chasing!'

class LoudDog(Dog):
    def noise(self):
        return super()\
              .noise().upper()
```

```
class Bike(object):
    def noise(self):
        return 'On Your Left'

    def pedal(self):
        return 'Pedaling!'

class LoudBike(Bike):
    def noise(self):
        return super()\
              .noise().upper()
```

- Completely unrelated objects
- But, there is a code commonality

# Mixin Classes

- A mixin is a class whose purpose is to add extra functionality to other class definitions
- Idea : If a user implements some basic features in their class, a mixin can be used to fill out the class with extra functionality
- Sometimes used as a technique for reducing the amount of code that must be written

# Mixin Example

- A class with a fragment of code

```
class Loud(object):
    def noise(self):
        return super().noise().upper()
```

- Not usable in isolation
- Mixes with other classes via inheritance

```
class LoudDog(Loud, Dog):
    pass

class LoudBike(Loud, Bike):
    pass
```

# How it works

- Example:

```
class LoudDog(Loud, Dog):
    pass
```

super()

```
>>> d = LoudDog()
>>> d.noise()
'WOOF'
>>>
```

- super() moves to the next class
- Allows mixins to combine with arbitrary classes. We’ll explain shortly!

# Real World Example

```
from app import db
from flask_login import UserMixin

class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))

    def __repr__(self):
        return '<User {}>'.format(self.username)
```

# First: Inside Python Objects

- A dictionary is a collection of named values

```
stock = {
          'name'   : 'GOOG',
          'shares' : 100,
          'price'  : 490.10
       }
```

- Dictionaries are commonly used for simple data structures (shown above)
- However, they are used for everywhere inside Python and may be the most important data type

# Dicts and Objects

- User-defined objects use dictionaries
  - Instance data
  - Class members
- In fact, the entire object system is mostly just an extra layer that's put on top of dictionaries
- Let's take a look...

# Dicts and Instances

- A dictionary holds instance data (__dict__)

```
>>> s = Stock('GOOG',100,490.10)
>>> s.__dict__
{'name' : 'GOOG','shares' : 100, 'price': 490.10 }
```

- You populate this dict when assigning to self

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

```
self.__dict__
```

```
{
   'name' : 'GOOG',
   'shares' : 100,
   'price' : 490.10
}
```

instance data

# Dicts and Instances

- Critical point : Each instance gets its own private dictionary

```
s = Stock('GOOG',100,490.10)
t = Stock('AAPL',50,123.45)
```

```
{
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10
}
```

```
{
    'name' : 'AAPL',
    'shares' : 50,
    'price' : 123.45
}
```

# Dicts and Classes

- A dictionary holds the members of a class

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
    def cost(self):
        return self.shares * self.price
    def sell(self, nshares):
        self.shares -= nshares
```

```
Stock.__dict__   ──────►   {
                            'cost' : <function>,
                            'sell' : <function>,
                            '__init__' : <function>,
                           }
```

methods

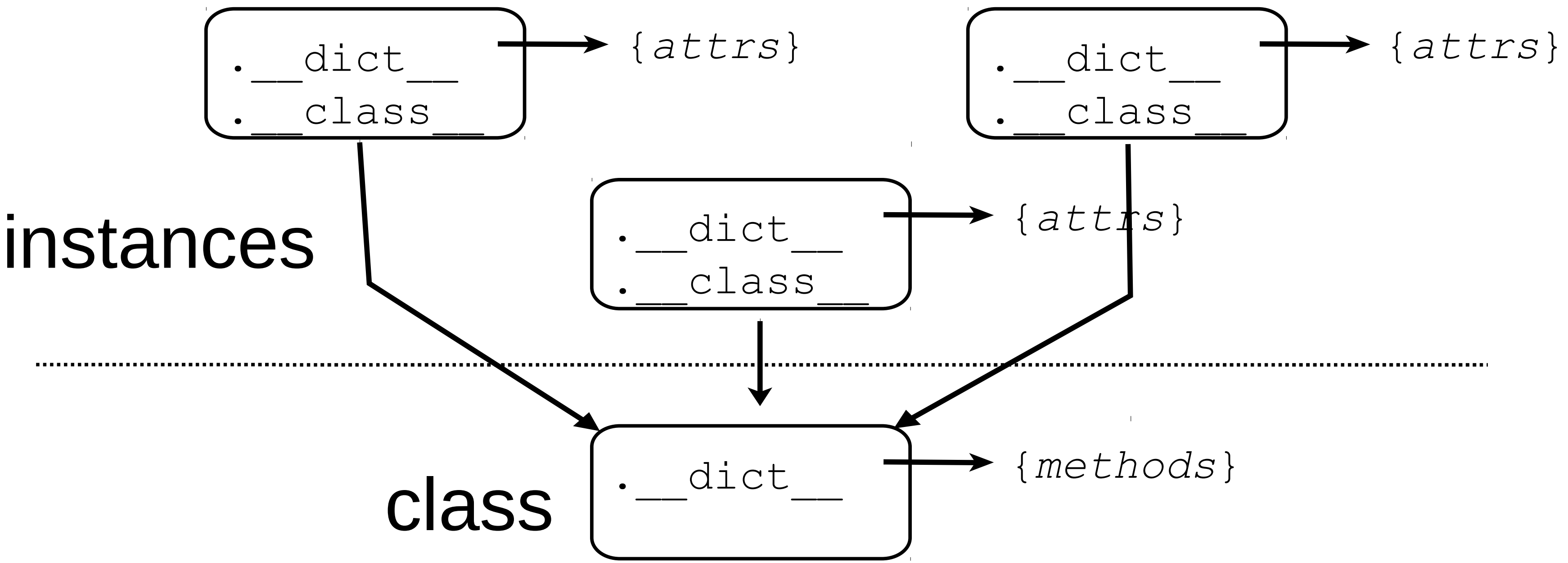
# Instances and Classes

- Instances and classes are linked together
- __class__ attribute refers back to the class

```
>>> s = Stock('GOOG', 100, 490.10)
>>> s.__dict__
{'name':'GOOG','shares':100,'price':490.10 }
>>> s.__class__
<class '__main__.Stock'>
>>>
```

- The instance dictionary holds data unique to each instance whereas the class dictionary holds data collectively shared by all instances

# Instances and Classes



# Attribute Access

- When you work with objects, you access data and methods using the (.) operator

```
x = obj.name          # Getting
obj.name = value      # Setting
del obj.name          # Deleting
```

- These operations are directly tied to the dictionaries sitting underneath the covers

# Modifying Instances

- Operations that modify an object always update the underlying dictionary

```
>>> s = Stock('GOOG',100,490.10)
>>> s.__dict__
{'name':'GOOG', 'shares':100, 'price':490.10 }
>>> s.shares = 50
>>> s.date = '6/7/2007'
>>> s.__dict__
{ 'name':'GOOG', 'shares':50, 'price':490.10,
  'date':'6/7/2007'}
>>> del s.shares
>>> s.__dict__
{ 'name':'GOOG', 'price':490.10,  'date':'6/7/2007'}
>>>
```

# Reading Attributes

- Suppose you read an attribute on an instance

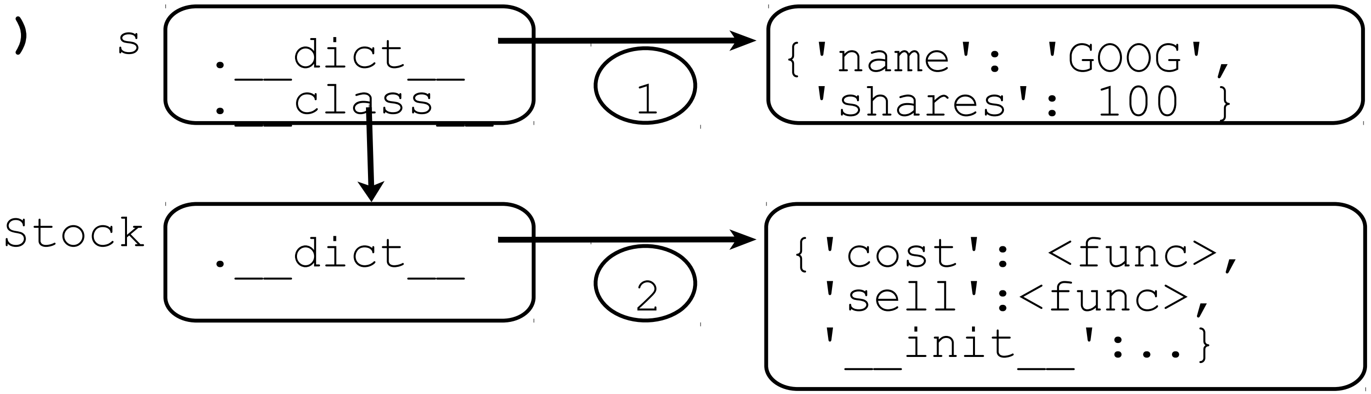
```
x = obj.name
```

- Attribute may exist in two places
  - Local instance dictionary
  - Class dictionary
- So, both dictionaries may be checked

# Reading Attributes

- First check in local __dict__
- If not found, look in __dict__ of class

```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
>>>
```




- This lookup scheme is how class attributes are shared by all instances

# Exercise 10

Approx 10 minutes

# How Inheritance Works

- Classes may inherit from other classes

```
class A(B,C):
    ...
```

- Bases are stored as a tuple in each class

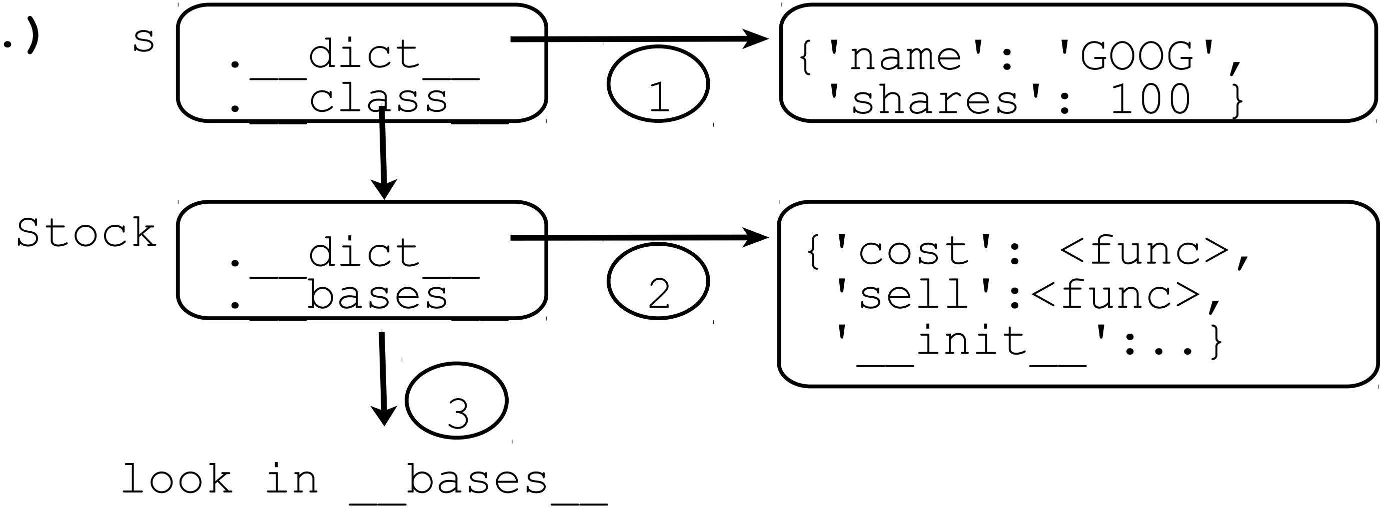
```
>>> A.__bases__
(<class '__main__.B'>,<class '__main__.C'>)
>>>
```

- This provides a link to parent classes
- This link simply extends the search process used to find attributes

# Reading Attributes

- First check in local __dict__
- If not found, look in __dict__ of class
- If not found in class, look in base classes

```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
>>>
```



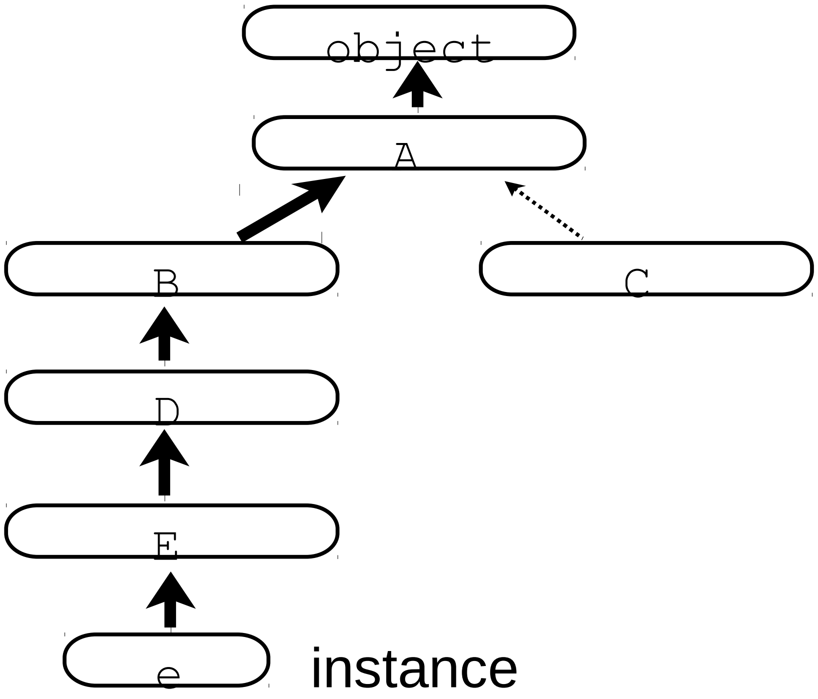
# Single Inheritance

- In inheritance hierarchies, attributes are found by walking up the inheritance tree

```
class A(object): pass
class B(A): pass
class C(A): pass
class D(B): pass
class E(D): pass
```

- With single inheritance, there is a single path to the top
- You stop with the first match

```
e = E()
e.attr
```



# The MRO

- The inheritance chain is precomputed and stored in an "MRO" attribute on the class

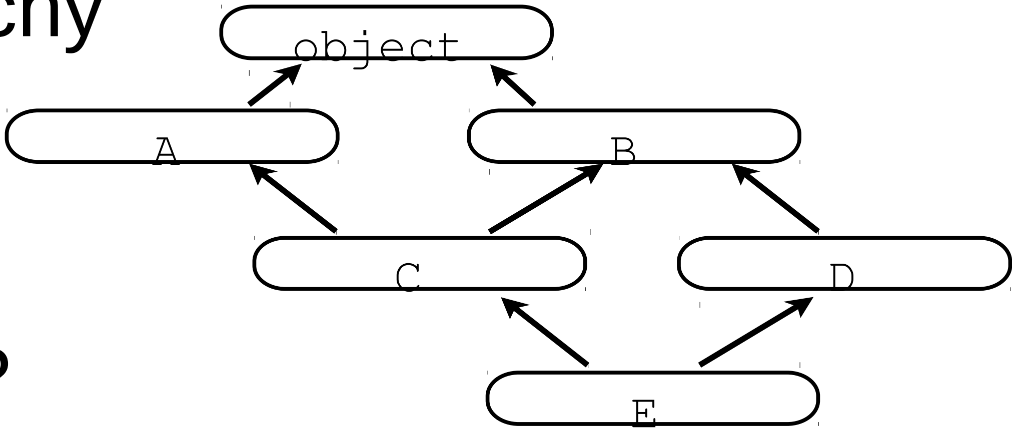
```
>>> E.__mro__
(<class '__main__.E'>,  <class '__main__.D'>,
 <class '__main__.B'>,  <class '__main__.A'>,
 <type 'object'>)
>>>
```

- "Method Resolution Order"
- To find attributes, Python walks the MRO
- First match wins

# Multiple Inheritance

- Consider this hierarchy

```
class A(object): pass
class B(object): pass
class C(A,B): pass
class D(B): pass
class E(C,D): pass
```




- What happens here?

```
e = E()
e.attr
```

- A similar search process is carried out, but there is an added complication in that there may be many possible search paths

# Multiple Inheritance

- Python uses "cooperative multiple inheritance"
- There are some ordering rules:

> Rule 1: Children before parents
> Rule 2: Parents go in order

- Inheritance works in two directions (up the hierarchy, across the list of parents)

rule 1

rule 2

```
class C(A, B):
    ...
```

# Multiple Inheritance

- Multiple inheritance hierarchy is flattened

```
>>> D.__mro__
(<class '__main__.D'>,  <class '__main__.B'>,
 <class '__main__.C'>,  <class '__main__.A'>,
 <type 'object'>)
>>>
```

- Using the C3 Linearization algorithm
- A constrained merge sort of parent MROs
- An ordering based on "the rules"

# Multiple Inheritance

- Consider classes with a common parent

```
class Base(object):
    ...
```

```
class A(Base):
    ...
```

```
class B(Base):
    ...
```

```
class C(Base):
    ...
```

- All children of a common parent go first

```
class D(A,B,C):
    ...
```

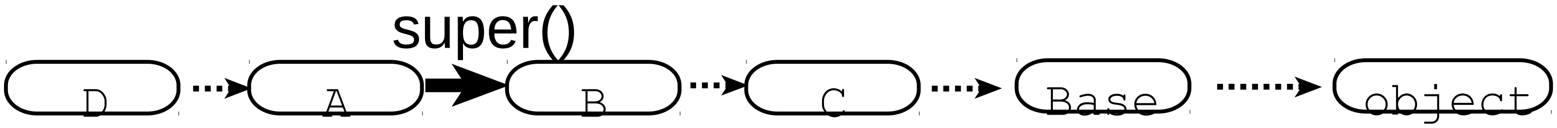
MRO D → A → B → C → Base

# Why super()?

- Always use super() when overriding methods

```
class A(Base):
    def spam(self):
        ...
        return super().spam()
```

- super() delegates to the next class on the MRO


- Tricky bit: You don't know what it is

# super() Explained

- super() is one of the most poorly understood Python features

```
class A(Base):
    def spam(self):
        Base.spam(self)
```

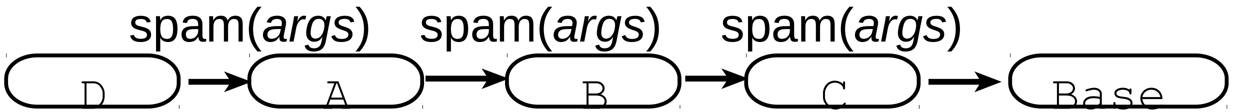
vs.

```
class A(Base):
    def spam(self):
        super().spam()
```

- These two classes are not the same
- super binds to the next implementation that is defined according to the instance's MRO
- It's not necessarily the immediate parent

# Designing for Inheritance

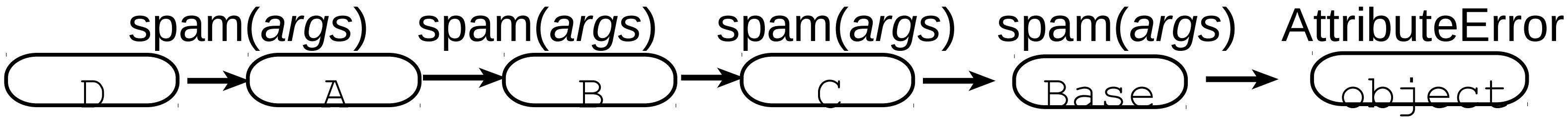
- Rule 1: Compatible Method Arguments




- Overridden methods must have a compatible signature across the entire hierarchy
- Remember: super() might not go to the immediate parent
- Tip: If there are varying method signatures, use keyword arguments

# Designing for Inheritance

- Rule 2: Method chains must terminate



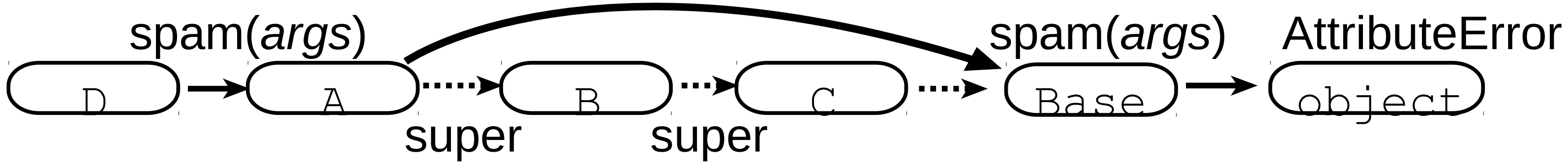

- You can't use super() forever--some class has to terminate the search chain

```
class Base(object):
    def spam(self):
        pass
```

- Typically the role of an abstract base class

# Designing for Inheritance

- Rule 3:  use super() everywhere




- Direct parent calls might explode heads

```
class A(Base):
    def spam(self):
        Base.spam(self)    # NO!
```

- If multiple inheritance is used, a direct parent call will probably violate the MRO

# Class Methods

- A method that operates on the class

```
class SomeClass(object):
    @classmethod
    def yow(cls):
        print('SomeClass.yow', cls)
```

- It's invoked on the class, not an instance

```
>>> SomeClass.yow()
SomeClass.yow <class '__main__.SomeClass'>
>>>
```

- The class is passed as the first argument

```
SomeClass.yow()
```

```
@classmethod
def yow(cls):
    ...
```

# Using Class Methods

- Class methods are often used as a tool for defining alternate initializers

```
class Date(object):
    def __init__(self, year, month, day):
        self.year  = year
        self.month = month
        self.day   = day
    @classmethod
    def today(cls):
        tm = time.localtime()
        return cls(tm.tm_year, tm.tm_mon, tm.tm_mday)
```

Notice how the class passed as an argument.

```
d = Date.today()
```

# Exercise 11

Approx 20 minutes

# Composition over Inheritance

- Composing objects creates a "has-a" relationship
- Favoured by many modern OOP enthusiasts
- Methods and attributes are delegated instead of inherited
- A very flexible approach
- Our `LocationStore` container was an example
- Dogmatism is always wrong!

# Composition and Delegation

- An employee has a job but they are not a job
- "has-a" is a better relationship than "is-a"

```
class Job:
    def __init__(self, title, salary):
        self.title = title
        self.salary = salary

class Employee:
    def __init__(self, job):
        self._job = job

    @property
    def salary(self):
        return self._job.salary
```

# Exercise 12

Approx 15 minutes

# Modules & Packages

- Objects are structural elements of software
- Functions are the basic re-usable element
- Classes and objects are the next level
- Modules and packages are the "top level" objects of application structure
- Useful for organising and distributing codes

# Modules

- Every Python source file is a module

```
# foo.py
def grok(a):
    ...
def spam(b):
    ...
```

- import statement loads and executes a module

```
import foo

a = foo.grok(2)
b = foo.spam('Hello')
...
```

# Module Objects

- Modules are objects

```
>>> import foo
>>> foo
<module 'foo' from 'foo.py'>
>>>
```

- A "namespace" for definitions inside

```
>>> foo.grok(2)
>>>
```

- Actually a layer on top of a dictionary (globals)

```
>>> foo.__dict__['grok']
<function grok at 0x1006b6c80>
>>>
```

# Special Variables

- A few special variables defined in a module

```
__file__        # Name of the source file
__name__        # Name of the module
__doc__         # Module documentation string
```

- Example: "main" check

```
if __name__ == '__main__':
   print('Running as the main program')
else:
   print('Imported as a module using import')
```

# Import Implementation

- Import in a nutshell (pseudocode)

```
import types

def import_module(name):
    # locate the module and get source code
    filename = find_module(name)
    code = open(filename).read()

    # Create the enclosing module object
    mod = types.ModuleType(name)

    # Run it
    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

- Source is exec'd in module dictionary
- Contents are whatever is left over

# import statement

- import executes the entire module

```
# bar.py
import foo
```

- It inserts a name reference to the module object into the dictionary used by the code that made the import

```
# bar.py
import foo

foo.grok(2)
```

```
bar.__dict__
{
    ...
    'foo' : <module 'foo'>
    ...
}
```

# Module Cache

- Each module is loaded only once
- Repeated imports just return a reference to the previously loaded module
- sys.modules is a dict of all loaded modules

```
>>> import sys
>>> list(sys.modules)
['copy_reg', '__main__', 'site', '__builtin__',
'encodings', 'encodings.encodings', 'posixpath', ...]
>>>
```

# Import Caching

- Import (pseudocode)

```
import types
import sys

def import_module(name):
    # Check for cached module
    if name in sys.modules:
        return sys.modules[name]

    filename = find_module(name)
    code = open(filename).read()
    mod = types.ModuleType(name)
    sys.modules[name] = mod

    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

- There is more, but this is basically it

# from module import

- Selected symbols can be imported locally

```
# bar.py
from foo import grok

grok(2)
```

- Useful for frequently used names

- Confusion: This does not change how import works. The entire module executes and is cached. This merely copies a name.

```
grok = sys.modules['foo'].grok
```

# from module import *

- Takes all symbols from a module and places them into the caller's namespace

```
# bar.py
from foo import *

grok(2)
spam('Hello')
...
```

- However, it only applies to names that don't start with an underscore (_)
- _*name* often used when defining non-imported values in a module.

# Module Reloading

- Modules can sometimes be reloaded

```
>>> import foo
...
>>> import importlib
>>> importlib.reload(foo)
<module 'foo' from 'foo.py'>
>>>
```

- It re-executes the module source on top of the already defined module dictionary

```
# pseudocode
def reload(mod):
    code = open(mod.__file__, 'r').read()
    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

# Module Reloading Danger

- Module reloading is not advised
- Problem: Existing instances of classes will continue to use old code after reload
- Problem: Doesn't update definitions loaded with 'from module import name'
- Problem: Likely breaks code that performs typechecks or uses super()

# Locating Modules

- When looking for modules, Python first looks in the same directory as the source file that's executing the import
- If a module can't be found there, an internal module search path is consulted

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python36.zip',
 '/usr/local/lib/python3.6',
 '/usr/local/lib/python3.6/plat-darwin',
 '/usr/local/lib/python3.6/lib-dynload',
 '/usr/local/lib/python3.6/site-packages']
```

# Module Search Path

- sys.path contains search path
- Can manually adjust if you need to

```
import sys
sys.path.append('/project/foo/pyfiles')
```

- Paths also added via environment variables

```
% env PYTHONPATH=/project/foo/pyfiles python3
Python 3.6.0 (default, Jan 12 2017, 13:20:23)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)]
>>> import sys
>>> sys.path
['', '/project/foo/pyfiles',
 '/usr/local/lib/python36.zip', ... ]
```

# Organizing Libraries

- It is standard practice for Python libraries to be organized as a hierarchical set of modules that sit under a top-level package name

```
packagename
packagename.foo
packagename.bar
packagename.utils
packagename.utils.spam
packagename.utils.grok
packagename.parsers
packagename.parsers.xml
packagename.parsers.json
...
```

- Other programming languages have a similar convention (e.g., Java)

# Creating a Package

- To create the module library hierarchy, organize files on the filesystem in a directory with the desired structure

```
packagename/
            foo.py
            bar.py
            utils/
                  spam.py
                  grok.py
            parsers/
                  xml.py
                  json.py
...
```

# Creating a Package

- Add __init__.py files to each directory

```
packagename/
    __init__.py
    foo.py
    bar.py
    utils/
        __init__.py
        spam.py
        grok.py
    parsers/
        __init__.py
        xml.py
        json.py
...
```

- These can be empty, but they should exist

# Using a Package

- Once you have the __init__.py files, the import statement should just "work"

```
import packagename.foo
import packagename.parsers.xml

from packagename.parsers import xml
```

- *Almost* everything should work the same way that it did before except that import statements now have multiple levels

# Fixing Relative Imports

- Relative imports of submodules don't work

```
spam/
      __init__.py
      foo.py
      bar.py
```

```
# bar.py
import foo    # Fails (not found)
```

- The issue: Resolving name clashes between top-level packages and submodules

```
spam/
      __init__.py
      os.py
      bar.py
```

```
# bar.py
import os    # ??? (uses stdlib)
```

- imports are always "absolute" (from top level)

# Absolute Imports

- One approach : use absolute imports

```
spam/
     __init__.py
     foo.py
     bar.py
```

- Example :

```
# bar.py

from spam import foo
```

- Notice use of top-level package name

# Package Relative Imports

- Consider a package

```
spam/
    __init__.py
    foo.py
    bar.py
    grok/
        __init__.py
        blah.py
```

- Package relative imports

```
# bar.py

from . import foo          # Imports ./foo.py
from .foo import name      # Load a specific name

from .grok import blah     # Imports ./grok/blah.py
```

# __init__.py Usage

- What are you supposed to do in those files?
- Main use: stitching together multiple source files into a "unified" top-level import

# Module Assembly

- Consider two submodules in a package

```
spam/
    foo.py
```

➤
```
# foo.py

class Foo(object):
    ...
    ...
```

```
    bar.py
```

➤
```
# bar.py

class Bar(object):
    ...
    ...
```

- Suppose you wanted to combine them

# Module Assembly

- Combine in __init__.py

```
spam/
    foo.py
```

➤
```
# foo.py

class Foo(object):
    ...
    ...
```

```
    bar.py
```

➤
```
# bar.py

class Bar(object):
    ...
    ...
```

```
__init__.py
```

➤
```
# __init__.py

from .foo import Foo
from .bar import Bar
```

# Module Assembly

- Users see a single unified top-level package

```
import spam

f = spam.Foo()
b = spam.Bar()
...
```

- Split across submodules is hidden

# Case Study

- The collections "module"
- It's actually a package with a few components

_collections.so

```
deque
defaultdict
```

_collections_abc.py

```
Container
Hashable
Mapping
...
```

collections/__init__.py

- ➤ **`from _collections import ( deque, defaultdict )`**

```
from _collections import (
                    deque, defaultdict )
```

- ➤ **`from _collections_abc import *`**

```
from _collections_abc import *

class OrdererDict(dict):
    ...

class Counter(dict):
    ...
```

# Exercise 13

Approx 10 minutes

# Abstract Base Classes

- Formal interfaces in Python
- Uses the “abc” module
- Very powerful but heavy on metaclasses and inheritance
- Allows us to extend `isinstance` and `issubclass` to support duck typing!

# Abstract Base Class

```
import abc
class PluginBase(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def load(self, input):
        """Retrieve data from the input source
        and return an object.
        """

    @abc.abstractmethod
    def save(self, output, data):
        """Save the data object to the output."""
```

# Registering a Concrete Class

```
@PluginBase.register
class RegisteredImplementation:

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)

print('Subclass:', issubclass(RegisteredImplementation, PluginBase))
print('Instance:', isinstance(RegisteredImplementation(), PluginBase))
```

# Exercise 14

Approx 15 minutes

# Functional Programming

- Programming style characterized by
  - Functions
  - No sides effects/mutability
  - Higher order functions

# Higher Order Functions

- Essential features...
  - Functions can accept functions as input
  - Functions can return functions as results
- Python supports both

# Functions as Input

- Consider these two functions

```
def sum_squares(nums):
    total = 0
    for n in nums:
        total += n * n
    return total

def sum_cubes(nums):
    total = 0
    for n in nums:
        total += n ** 3
    return total
```

only difference

- They're almost identical (one line differs)

# Functions as Input

- Recognizing commonality is part of abstraction

```
def sum_map(func, nums):
    total = 0
    for n in nums:
        total += func(n)
    return total

def square(x):
    return x * x

nums = [1, 2, 3, 4]
r = sum_map(square, nums)
```

- This version allows any function to be passed
- Sometimes referred to as a "callback function"

# Lambda Functions

- One-expression functions can use lambda

```
def sum_map(func, nums):
    total = 0
    for n in nums:
        total += func(n)
    return total

nums = [1, 2, 3, 4]
result = sum_map(lambda x: x*x, nums)
```

- Creates an anonymous function
- Can only contain a single expression
- No control flow, exceptions, etc.

# Partial Application

- Lambda often used to alter function args

```
def distance(x, y):
    return abs(x - y)

>>> distance(10, 20)
10
>>> dist_from10 = lambda y: distance(10, y)
>>> dist_from10(3)
7
>>> dist_from10(14)
4
>>>
```

- functools.partial

```
from functools import partial
dist_from10 = partial(distance, 10)
```

# Returning Functions

- Consider the following function

```
def add(x, y):
    def do_add():
        print(f'{x} + {y} -> {x+y}')
    return do_add
```

- A function that returns another function?

```
>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
3 + 4 -> 7
>>>
```

- Notice that it works, but ponder it...

# Nested Scopes

- Observe how the inner function refers to variables defined by the outer function

```
def add(x, y):
    def do_add():
        print(f'{x} + {y} -> {x+y}')
    return do_add
```

- Further observe that those variables are somehow kept alive after add() has finished

```
>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
3 + 4 -> 7
```

Where are the x,y values coming from?

# Closures

- If an inner function is returned as a result, the inner function is known as a "closure"

```
def add(x, y):
    def do_add():
        print(f'{x} + {y} -> {x+y}')
    return do_add
```

- Essential feature : A "closure" retains the values of all variables needed for the function to run properly later on

# Closures

- To make it work, references to the outer variables (bound variables) get carried along with the function

```
def add(x, y):
    def do_add():
        print(f'{x} + {y} -> {x+y}')
    return do_add

>>> a = add(3, 4)
>>> a.__closure__
(<cell at 0x54f30: int object at 0x54fe0>,
 <cell at 0x54fd0: int object at 0x54f60>)
>>> a.__closure__[0].cell_contents
3
>>> a.__closure__[1].cell_contents
4
```

# Closures

- Closures only capture used variables

```
def add(x, y):
    result = x + y
    def get_result():
        return result
    return get_result
```

```
>>> a = add(3, 4)
>>> a.__closure__
(<cell at 0x10bb52708: int object at 0x10b5d3610>,)
>>> a.__closure__[0].cell_contents
7
>>>
```

- Carefully observe: x and y are not included (not needed in the function body)

# Closures and Mutability

- Closure variables are mutable (nonlocal decl)

```
def counter(n):
    def incr():
        nonlocal n
        n += 1
        return n
    return incr

>>> c = counter(10)
>>> c()
11
>>> c()
12
>>>
```

- Can be used to hold mutable internal state, much like an object or class

# Using Closures

- Closures are an essential feature of Python
- Common applications:
  - Delayed evaluation
  - Callback functions
  - Code creation ("macros")

# Exercise 15

Approx 15 minutes

# The End

- OOP provides a way of thinking about software
- A variety of contradictory definitions
- Lots of jargon
- Still a powerful and useful approach to system design