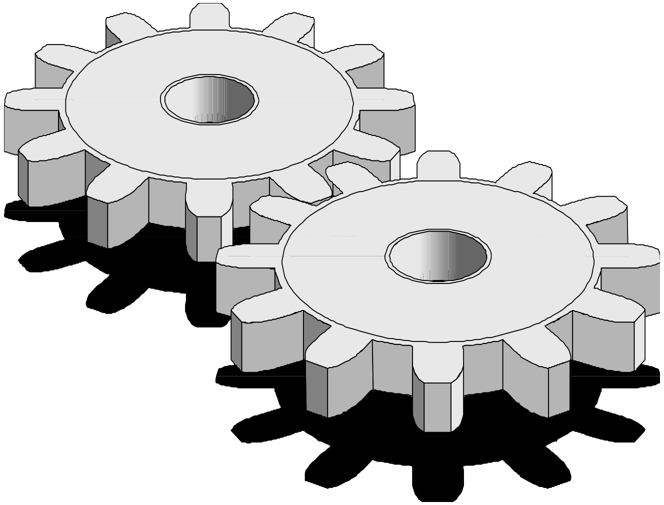
python™ 2 programming

Classes and Object Oriented Programming

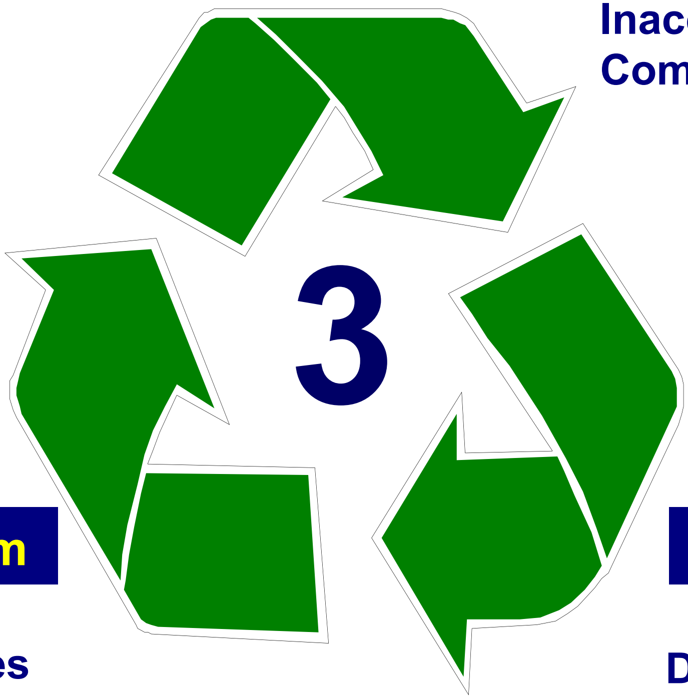
Classes and OOP

- Contents
  - Object Oriented Programming
  - Using objects
  - A little Python OO
  - A simple class
  - Defining classes
  - Defining methods
  - Constructing an object
  - Special methods
    - Operator overloading
  - Inheritance
  - New-style classes
    - Properties and decorators
  - Meta data
  - Design Patterns
- Summary



Object-Oriented terminology

- OO Languages support the *Big Three*



Encapsulation

Accessible behaviour (functions)
Inaccessible state (data)
Combined in a single unit

Inheritance

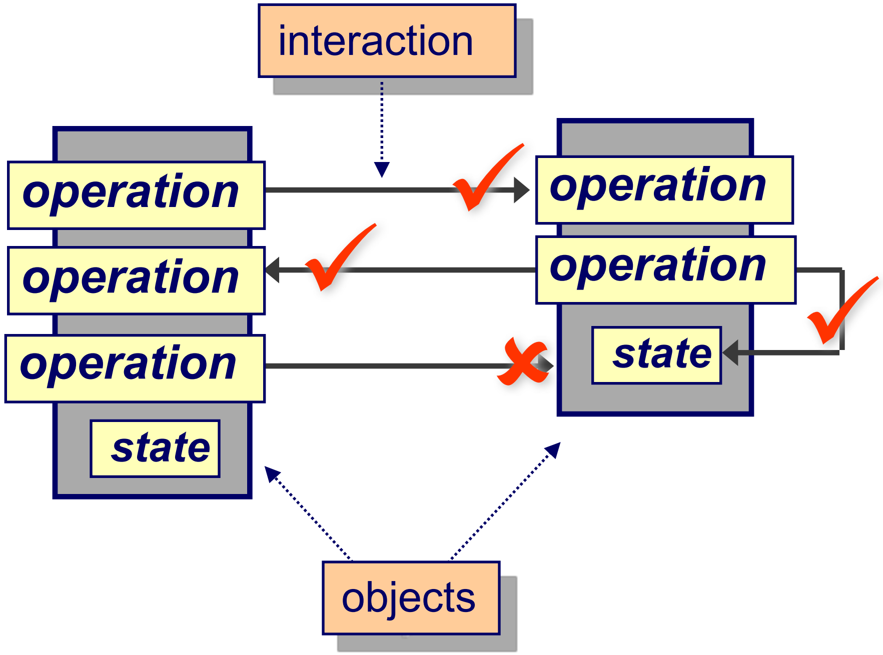
Define new classes in terms of existing classes

Polymorphism

Inheriting classes specialise behaviour

Object-Oriented Programming

- Offers an approach to structuring software
  - A model based on the behaviour of interacting objects
  - Key concept is abstraction, based around the *Big Three*
- Python was designed to be Object Oriented
  - Unlike *some* languages we could mention



Using objects

- Calling a class creates a new *instance object*
  - Invokes the constructor

```
from Account import Account

some_account = Account(1000.00)
some_account.deposit(550.23)
some_account.deposit(100)
some_account.withdraw(50)
print some_account.getbalance()

another = Account(0)
print Account.numCreated
print "object another is class",\
       another.__class__.__name__
```

```
1600.23
2
object another is class Account
```

A little Python OO

- A class is declared using `class`
  - Membership is by *indentation`*`
- Methods are declared as functions within that class
  - First argument passed is the object
  - The constructor is called `__init__`
  - The destructor is called `__del__`
    - Rarely required and unreliable
- Classes are usually declared in a module
  - File usually has same name as the class, with .py appended
  - After it has been used once, a pyc (compiled) form will be generated
  - Simple example over…

Defining classes

- **The class statement**
  - **Defines a class object**
    - Public attributes are referenced by *Class.attribute*
  - **Usually in a module with the same name as the class**

account.py

```
class Account:
    numCreated = 0
    def __init__(self, initial):
        self.__balance = initial
        Account.numCreated += 1
    def deposit(self, amt):
        self.__balance = self.__balance + amt
    def withdraw(self,amt):
        self.__balance = self.__balance - amt
    def getbalance(self):
        return self.__balance
```

Public class variable

Methods

Defining methods

- **Methods are functions defined within a class**
- **Conventions with underscores - reminder**
  - **Names beginning with one underscore are private to a *module***
  - **Names beginning with two underscores are private to a *class***
  - **Names beginning and ending with two underscores have a special meaning**
- **Object methods**
  - **First argument passed to a method is the object**
    - Usually called 'self', but can be anything
- **Class methods and attributes**
  - **Defined within the class**
  - **Can be called on a class or object**

Constructing an object

- **Python has two methods**
  - **__new__**
    - Called when an object is created
    - First parameter is the class name
    - Return the constructed object
  - **__init__**
    - Called when an object is initialised
    - First parameter is the object
    - An implicit return of the current object
  - **__new__ is called in preference, and then calls __init__**

If in doubt,
use __init__

- **Which to use?**
  - **Use __new__ only if constructing an object of a different class**
  - **In most cases use `__init__`**

Factory Design Pattern

```
_isSecureMode = True

class Account(object):
    __numCreated = 0
    def __new__(cls, *args, **kwargs):
        if not _isSecureMode:
            return super(Account, cls).__new__(cls, *args, **kwargs)
        else:
            return SecureAccount(*args,**kwargs)
    def __init__(self, initial):
         self.__balance = initial
        Account.__numCreated += 1
    def deposit(self, amt):
        self.__balance = self.__balance + amt
    def withdraw(self,amt):
         self.__balance = self.__balance - amt
    def getbalance(self):
        return self.__balance
```

```
#client code
a = Account(100)
a.deposit(50)
print
a.getbalance()
```

```
class SecureAccount(object):

   def __init__(self, initial):
        …
   def deposit(self, amt):
        …
   def withdraw(self, amt):
        …
   def getbalance(self):
        …
```

Special methods

- **A mechanism for operator and special function overloading**
- **Function names start and end with two underscores**

`__bool__(self)`	Return True or False
`__del__(self)`	Called when an object is destroyed
`__format__(self,`*spec*`)`	`str.format` support
`__hash__(self)`	Return a suitable key for dictionary or set
`__init__(self,` *args*`)`	Initialize an object
`__len__(self)`	Implement the `len()` function
`__new__(class,` *args*`)`	Create an object
`__repr__(self)`	Return a string representation
`__str__ (self)`	Return a human readable representation

Operator overload special methods

- All operators may be overloaded
  - See the online documentation for a complete list

- Return types vary
  - Can return a `NotImplemented` object
  - Examples:

`__add__`	+
`__sub__`	-
`__eq__`	==
`__lt__`	<
`__invert__`	~ (logical NOT)
`__getitem__(self,key)`	container element evaluation
`__setitem__(self, key, value)`	container element assignment

Special methods - example

```
import copy
class Date:

    def __init__(self, day=0, month=0, year=0):
        self.__day   = day
        self.__month = month
        self.__year  = year

    def __str__(self):
        return str(self.__day)   + '/' + \
               str(self.__month) + '/' + \
               str(self.__year)

    def __add__ (self, value):
        retn = copy.deepcopy(self)
        retn.__day = retn.__day + value
        retn.__validate_date()
        return retn
```

Note private variable and method names starting with two underscores

```
from date import Date
today = Date(13,12,1949)
print(today)
today += 1
```

'this' and 'self' mixture is for demonstration purposes only!

Custom container using special methods

```
class Bank(object):
    def __init__(self):
        self._accounts = []
    def __len__(self):
        return len(self._accounts)
    def __getitem__(self,key):
        return self._accounts[key]
    def __setitem__(self,key,value):
         self._accounts[key] = value
    def __delitem__(self,key):
        del self._accounts[key]
    def __contains__(self,item):
        return item in self._accounts
    def __iter__(self):
        for a in self._accounts:
            yield a.getbalance()
    def Add(self,account):
        self._accounts.append(account)
```

```
d = Checking()
b = Bank()
b.Add(Account(10))
b.Add(d)
b.Add(Account(2))
print len(b)
print b[0].getbalance()
b[0].deposit(1000)
print b[0].getbalance()
del b[0]
print len(b)
print b[1].getbalance()
for a in b:
    print a
print d in b
```

Observer Design Pattern

- Define a one-to-many dependency between objects
- When one object changes state, all its dependents are notified and updated automatically.

Invoker:

```python
class Invoker(object):
    def __init__(self):
        self._subscribers = []
    def __iadd__(self,subscriber):
        self._subscribers.append(subscriber)
        return self
    def __isub__(self,subscriber):
        if(subscriber in self._subscribers):
            self._subscribers.remove(subscriber)
        return self
    def Invoke(self,message,exclude=None):
        for s in self._subscribers:
            if(s!=exclude):
                s(message)
```

Observers

```
def handler1(message):
    print "handler1 got:" + message

def handler2(message):
    print "handler2 got:" + message

def handler3(message):
    print "handler3 got:" + message

def handler4(message):
    print "handler4 got:" + message

class A(object):
    def __init__(self,val):
        self.__val=val
    def handler5(self,message):
        print "handler1 got:" + message + " with " + 
str(self.__val)
```

Link all together

- Inspired by C# delegates

```
inv = Invoker()

a1 = A(10)
a2 = A(20)

inv+=a1.handler5
inv+=a2.handler5
inv+=handler1
inv+=handler2
inv+=handler3
inv+=handler1
inv+=handler4

inv.Invoke("Hi all")

inv-=(handler1)
inv-=(handler1)

inv.Invoke("Hi all",handler1)
```

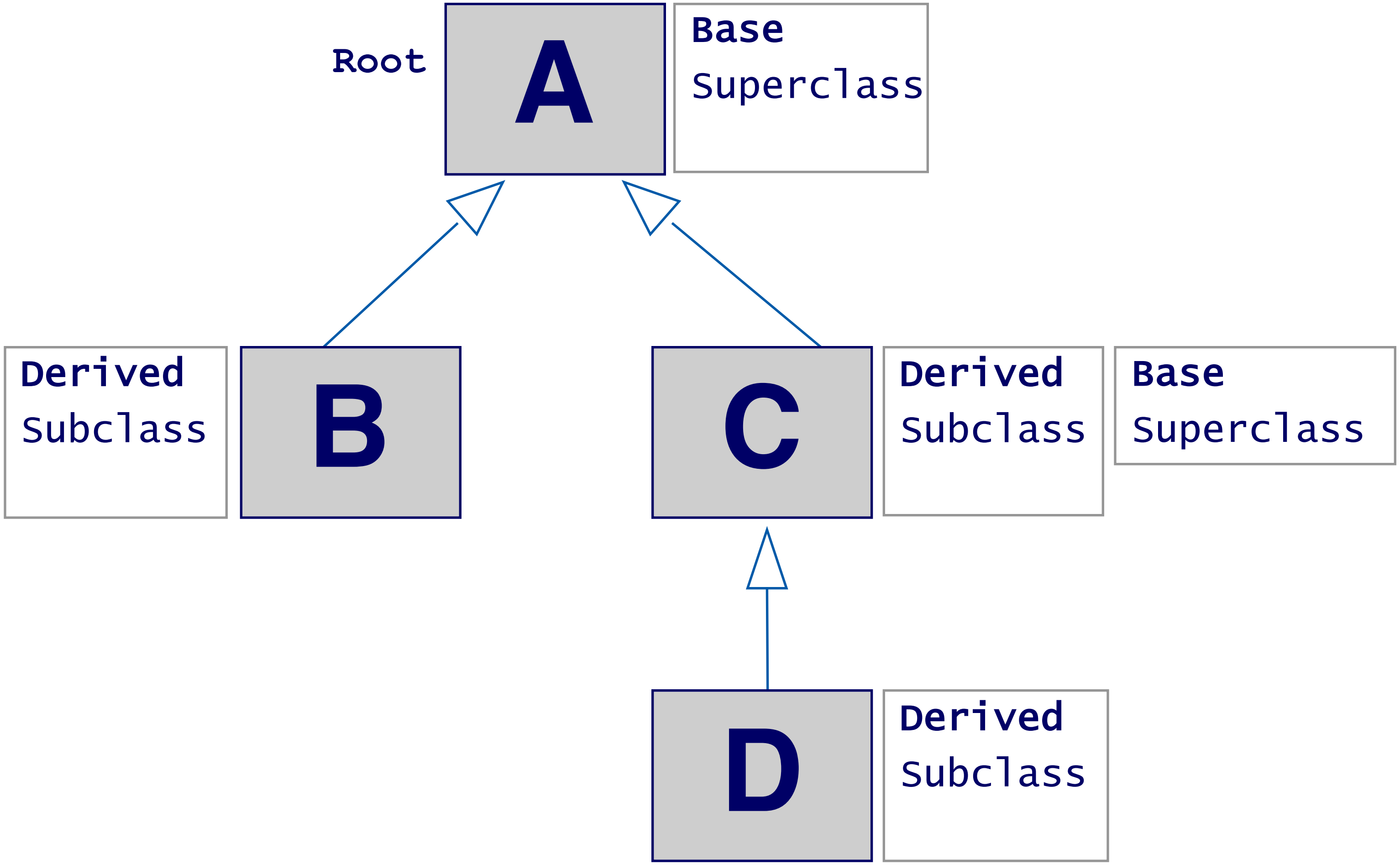
Inheritance

- **Use attributes and methods from a parent class**
  - **Important OO concept**
    - Python supports multiple inheritance - not often needed
  - **Attributes and methods not supplied in the derived class will be inherited from the base class**
  - **Common to derive our own classes from Python's own**
    - Multithreading
    - Exceptions
    - etc.

```
class DerivedClassName(base_classes):
    def __init__(self, arguments):
        base_class.__init__(self, arguments)

Other methods…
```

Inheritance terminology



New-style classes

- Classes which derive from *object*
  - Can be a built-in type like *dict*, *list*, or others
  - The aim is to unify Python classes
    - Whether built-in or user written
    - The only type of class at Python 3
  - Introduced at Python 2.2

- Some features only work with new-style classes

```
class Account(object):
    numCreated = 0
    def __init__(self, initial):
        self.balance = initial
        Account.numCreated += 1
    ...
```

This derives from the built-in type 'object'

Inheritance example

```
class Person(object):
   def __init__(self, name, gender):
       self.__name = name
       self.__gender = gender.upper()

   def __str__(self):
       return "Name: " + self.__name+ \
              " Gender: " + self.__gender
```

new-style class

User's view

```
from employee import Employee

me = Employee ("Fred Bloggs",
               'm', 'IT')
print me
```

This calls the parent class special method

```
from person import Person

class Employee(Person):

    def __init__(self, name, gender, dept):
        super(Employee, self).__init__(name, gender)
        self.__dept = dept
...
```

Multiple Inheritance – Old Style

```
class A:
    def __init__(self):
        print('Running A.__init__')
class B(A):
    def __init__(self):
        print('Running B.__init__')
        A.__init__(self)
 class C(A):
    def __init__(self):
        print('Running C.__init__')
        A.__init__(self)
 class D(B,C):
    def __init__(self):
        print('Running D.__init__')
        B.__init__(self)
        C.__init__(self)
```

```
>> d = D()
Running D.__init__
Running B.__init__
Running A.__init__
Running C.__init__
Running A.__init__
```

Multiple Inheritance – New Style

```
class Base(object):
  def __init__(self):
    super(Base, self).__init__()
    print "base“
class First(Base):
 def __init__(self):
  super(First, self).__init__()
  print "first“
class Second(Base):
 def __init__(self):
  super(Second, self).__init__()
  print "second“
class Third(First, Second):
 def __init__(self):
  super(Third, self).__init__()
  print "that's it“
# Third --> First --> Base --> object --> Second --> Base -->
object
# Third --> First --> Base  --> Second --> object
```

Properties

- Built-in property() creates an attribute
  - Only works correctly with new-style classes
  - property() has getter, setter, deleter, and docstring
  - The appropriate method is called depending on the way the attribute is used

```
class Date(object):
    ...
    def mget(self):
        return self.__day

    def mset(self, day):
        self.__day = day


    mday = property(mget, mset)
```

Call the (default) getter method

```
day = today.mday
```

Call the setter method

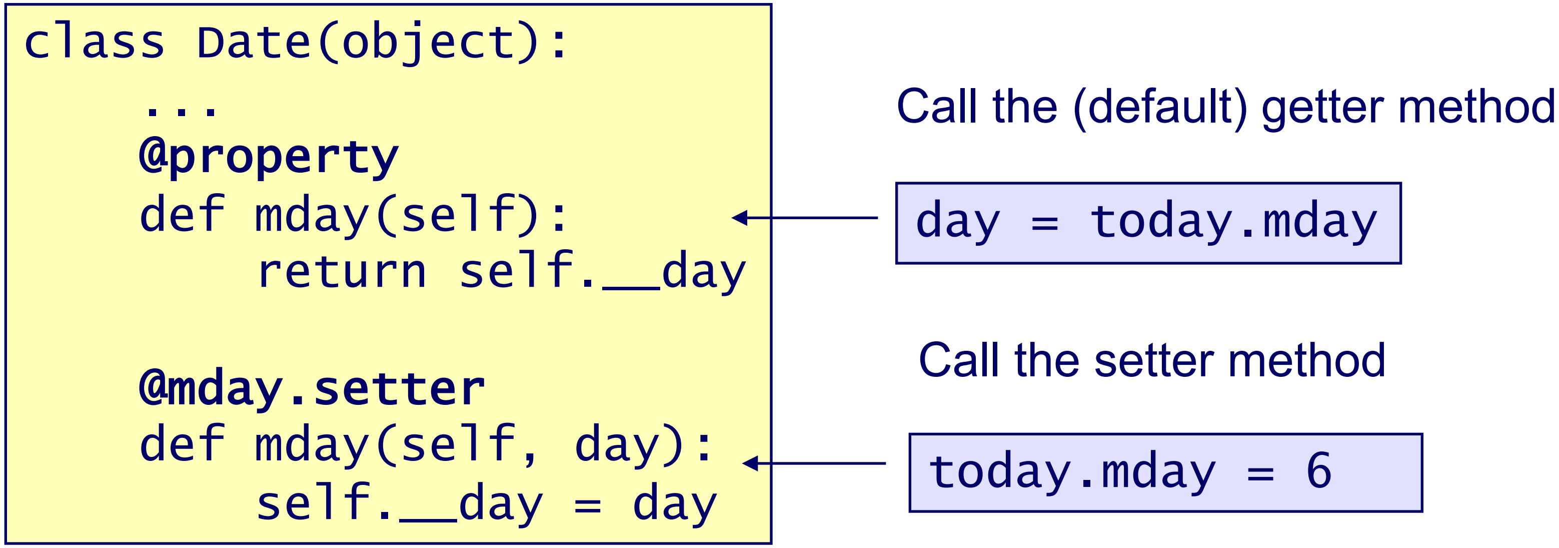
```
today.mday = 6
```

Omitting the setter method means that the attribute is *read-only*

Properties and decorators

- A decorator is a function name prefixed @
  - The function will normally return another function
  - The decorator is followed by the function to be returned

- Decorators are syntactic sugar, but commonly used
  - Built-in property() is usually called using a decorator



```
class Date(object):
    ...
    @property
    def mday(self):
        return self.__day

    @mday.setter
    def mday(self, day):
        self.__day = day
```

Call the (default) getter method

```
day = today.mday
```

Call the setter method

```
today.mday = 6
```

Property decorators only work with new-style classes

Class methods

- There are several ways to achieve this
  - Using a dummy class wrapper
  - Using the classmethod built-in as a decorator (preferred)
    - The class method itself

```
    __count = 0
...
    @classmethod
    def get_count(cls):
        return Date.__count
```

The class name is passed implicitly

- The user of the class

```
from date import Date
...
cnt = Date.get_count()
```

Functions or Bound/Unbound methods

```
class C(object):
   def foo(self):
      pass
```

```
>>> C.foo
<unbound method C.foo>

>>> c = C()
>>> c.foo
<bound method C.foo of <__main__.C object at 0x10d33aed0>>
```

But

```
>>> C.__dict__['foo']
<function foo at 0x10d33db18>
```

- **So a function or a method?**
  - **Class of C class implements a __getattribute__ that resolves descriptors**
  - **Functions have a __get__ method which makes them descriptors**

```
>>> C.__dict__['foo'].__get__(None, C)
<unbound method C.foo>
```

```
>>> c = C()
>>> C.__dict__['foo'].__get__(c, C)
<bound method C.foo of <__main__.C object at 0x17bd4d0>>
```

- **This way the method object binds the first parameter of a function to the instance of the class**

Class functions/methods

```
class D(object):
@staticmethod
  def foo():
   pass
```

- **staticmethod decorator implements a dummy __get__ that returns the wrapped function as function and not as a method:**

```
>>> C.__dict__['foo'].__get__(None, C)
<function foo at 0x17d0c30>
```

```
class E(object):
 @classmethod
 def foo(cls):
   pass
```

- **classmethod decorator implements a __get__ that returns the wrapped function as bound method where the class is the bounded instance:**

```
>>> E.foo
<bound method type.foo of <class '__main__.E'>>
```

Double Dispatch

- Python performs only single dispatching
- If you are performing an operation on more than one object whose type is unknown
  - Python can invoke the dynamic binding mechanism on only one of those types
- There must be two member function calls:
  - First to determine the first unknown type
  - Second to determine the second unknown type

Double Dispatch Example

```
# An enumeration type:
class Outcome:
    def __init__(self, value, name):
        self.value = value
        self.name = name
    def __str__(self):
 return self.name
    def __eq__(self, other):
        return self.value == other.value

Outcome.WIN = Outcome(0, "win")
Outcome.LOSE = Outcome(1, "lose")
Outcome.DRAW = Outcome(2, "draw")
```

Items

```
class Item(object):
    def __str__(self):
        return self.__class__.__name__
```

```
class Paper(Item):
    def compete(self, item):
        # First dispatch: self was Paper
        return item.evalPaper(self)
    def evalPaper(self, item):
        # Item was Paper, we're
        return Outcome.DRAW
    def evalScissors(self, item)
        # Item was Scissors, we
        return Outcome.WIN
    def evalRock(self, item):
        # Item was Rock, we're i
        return Outcome.LOSE
```

```
class Scissors(Item):
    def compete(self, item):
        # First dispatch: self was Scissors
        return item.evalScissors(self)
    def evalPaper(self, item):
        # Item was Paper, we're in Scissors
        return Outcome.LOSE
    def evalScissors(self, item):
        # Item was Scissors, we're in Scissors
        return Outcome.DRAW
    def evalRock(self, item):
        # Item was Rock, we're in Scissors
        return Outcome.WIN
```

```
class Rock(Item):
….
```

Double Dispatch in Action

```python
def match(item1, item2):
    print("%s <--> %s : %s" % (
      item1, item2, item1.compete(item2)))
```

```python
# Generate the items:
def itemPairGen(n):
    # Create a list of instances of all Items:
    Items = Item.__subclasses__()
    for i in range(n):
        yield (random.choice(Items)(),
               random.choice(Items)())

for item1, item2 in itemPairGen(20):
    match(item1, item2)
```

Metadata in Python

- Builtin functions
  - getattr
  - setattr
  - hasattr
  - globas , locals
  - eval
  - isinstance
  - issubclass
- Inspect module
  - isclass, ismethod and more...
  - getsource, getfile, getargspec and more…

Helper built-in functions example

- **isinstance(***object*, *classinfo***)**
  - **Returns True if** *object* **is of class** *classinfo*

- **issubclass(***class*, *classinfo***)**
  - **Returns True if** *class* **is a derived class of** *classinfo*

```
from employee import Employee
from person import Person

me = Employee("Fred Bloggs", 'm', 'IT')

if isinstance(me,Employee):
    print me,"isa Employee!"

if isinstance(me,Person):
    print me,"isa Person!"

if issubclass(Employee,Person):
    print "Employee is a subclass of Person"
```

All these conditions return True
(based on the inheritance example)

Delegation

- **Objects that call methods on other objects**
- **Implemented in Python using __getattr__**
    - **Delegates missing attributes to another object**
    - **Supported by the Python built-in getattr**
        - Returns an object's attribute by name (or a default value)
    - **In Python, a method is an attribute**
- **Used to build *wrapper* or *proxy* classes**

```
class proxy:
    def __init__(self, obj):
        self._wrapped = obj

    def __getattr__(self, aname):
        return getattr(self._wrapped, aname)
```

Returns the attribute (method) of the wrapped object

Proxy class example

- Using the Date class from the previous chapter

```
class Date:
...
def some_method(self):
        raise ValueError("False argument")
```

```
class Proxy:
    def __init__(self, obj):
        self._wrapped = obj

    def __getattr__(self, aname):
        return getattr(self._wrapped, aname)
    def other_method(self):

        raise ValueError("False argument")
```

```
today = Date(13,12,1949)
stuff = Proxy(today)
print stuff
stuff.some_method()
stuff.other_method()
```

Construct a Date object

Construct a Proxy object

Call Date.__str__

Call Date.some_method

Call Proxy.other_method

Metaclasses and ABC

- A metaclass is a class for creating other classes
  - The syntax for metaclasses changed at Python 3

- Abstract Base Classes
  - Classes that cannot be directly instantiated
  - Created metaclass ABCMeta and decorator abstractmethod

Why use Abstract Base Classes?

- Abstract base classes are a form of interface checking more strict than individual hasattr() checks for particular methods
- By defining an abstract base class, you can define a common API for a set of subclasses

How ABCs Work

- **abc** works by marking methods of the base class as abstract, and then registering concrete classes as implementations of the abstract base

- If your code requires a particular API, you can use issubclass() or isinstance() to check an object against the abstract class

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

Implements an Abstract class

- ## Registering a Concrete Class

```
class RegisteredImplementation(object):
    def load(self, input):
        return input.read()
    def save(self, output, data):
        return output.write(data)
PluginBase.register(RegisteredImplementation)
```

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

- ## Implementation Through Subclassing

```
class SubclassImplementation(PluginBase):
    def load(self, input):
        return input.read()
    def save(self, output, data):
        return output.write(data)
```

```
print 'Subclass:', issubclass(SubclassImplementation, PluginBase)
print 'Instance:', isinstance(SubclassImplementation(), PluginBase)
print 'PluginBase:‘
for sc in PluginBase.__subclasses__():
    print sc.__name__
```

Singleton with Meta Class

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton,
cls).__call__(*args, **kwargs)
        return cls._instances[cls]

#Python2
class SingleAccount(Account):
    __metaclass__ = Singleton
#Python3
class SingleAccount(Account, metaclass=Singleton):
    pass
```

```
s1 = SingleAccount(50)
s2 = SingleAccount(30)
```

Inspect module (python >= 2.1)

- **API** for learning about live objects, including modules, classes, instances, functions, and methods.
- Retrieve the original source code for a function
- Look at the arguments to a method on the stack
- Extract the sort of information useful for producing library documentation for your source code

Module Information

```
for name, data in inspect.getmembers(inspect_module):
    if name == '__builtins__':
        continue
        print '%s :' % name, repr(data)
for name, data in inspect.getmembers(inspect_module,
inspect.isclass):
    print '%s :' % name, repr(data)
from pprint import pprint
pprint(inspect.getmembers(inspect_module.A))
pprint(inspect.getmembers(inspect_module.A, inspect.ismethod))
pprint(inspect.getmembers(inspect_module.B, inspect.ismethod))
```

More services

- ## Retrieving Source

```
print inspect.getsource(inspect_module.A.get_name)
print inspect.getsource(inspect_module.A)
pprint(inspect.getsourcelines(inspect_module.A.get_name))
```

- ## Method and Function Arguments

```
arg_spec = inspect.getargspec(inspect_module.module_level_function)
print 'NAMES    :', arg_spec[0]
print '*        :', arg_spec[1]
print '**       :', arg_spec[2]
print 'defaults:', arg_spec[3]args_with_defaults = arg_spec[0][-
len(arg_spec[3]):]
print 'args & defaults:', zip(args_with_defaults, arg_spec[3])
```

Dynamic Class using namedtuple

```
import json
from collections import namedtuple

jstr = '{"name":"eli","age":20}‘
p = json.loads(jstr, object_hook=lambda d:
namedtuple('Person', d.keys())(*d.values()))

print p.name,p.age

p = p.__class__('ff',20)
```

Summary

- Classes vs. objects
  - A class is a user defined data type
  - An object is an instance of a class
  - Objects have identity
  - To achieve behaviour we call an operation on an object
  - The operations on an object are defined by its class
- Encapsulation
  - Separates interface from implementation
  - Publicly accessible operations
  - Privately maintained state
- Python supports the "Big Three"
  - Offers an approach to structuring software based on the behaviour of interacting objects
  - Supports abstraction and reuse