Classes**¶**

In [ ]:

The class statement**¶**

In [ ]:

class Account(object):  
 num\_accounts = 0 # class data  
 def \_\_init\_\_(self, name, balance):  
 self.name = name # public data  
 self.\_balance = balance # private data  
 Account.num\_accounts += 1  
 def \_\_del\_\_(self):  
 Account.num\_accounts -= 1  
 def deposit(self, amt):  
 self.\_balance += amt  
 def withdraw(self, amt):  
 self.\_balance -= amt  
 def enquire(self):  
 return self.\_balance

* \_\_init\_\_ : constructor
* \_\_del\_\_ : destructor
  + not always defined (garbage-collection)
  + used for closing internal non-reference-counted objects, like open file handles, sockets, etc

Instances**¶**

In [ ]:

a = Account('Jay', 1000.00) # invokes Account.\_\_init(a, 'Jay', 1000.00)  
b = Account('Veeru', 500.00)  
who = a.name  
a.deposit(200.0)  
a.enquire()

self**¶**

* very similar to this
* First argument to all instance methods
* inst.method(args) ==> class.method(inst, args)
* Also defines instance data : self.name, self.\_balance
* instance data is added by using it inside an instance method
* **NOT** a keyword; just a (very well-followed) convention

Encapsulation**¶**

* No language support for encapsulation
* Convention: \_names are taken as class private
* \_\_name taken as private for subclassing
  + Some versions support name-mangling, but not uniformly supported, can be worked around
* Respecting Data-hiding conventions programmers responsibilityWe're all adults here.

Inheritance**¶**

* Class specialization
* derived/sub class specializes base/super class
* derived class inherits all attributes of base class
  + may redifine any of these
  + may define attributes of its own

In [ ]:

class TaxedAccount(Account):  
 def \_\_init\_\_(self, name, balance):  
 super(TaxedAccount, self).\_\_init\_\_(name, balance) # Call base class ctr  
 def \_tax(self): # New private method  
 return self.\_balance \* 0.1 if self.\_balance>1000.0 else 0  
 def enquire(self):  
 return self.\_balance - self.\_tax()

In [ ]:

a = TaxedAccount('Jay', 1000.00)  
b = TaxedAccount('Veeru', 500.00)  
who = a.name  
a.deposit(200.0)  
a.enquire()

Multiple inheritance**¶**

In [ ]:

class DepositCharge(object):  
 fee = 5.00  
 def deposit\_fee(self):   
 self.withdraw(self.fee)  
  
class WithdrawlCharge(object):  
 fee = 2.50  
 def withdrawl\_fee(self):  
 self.withdraw(self.fee)

In [ ]:

class EvilAccount(TaxedAccount, DepositCharge, WithdrawlCharge):  
 def deposit(self,amt):  
 self.deposit\_fee()  
 super(EvilAccount,self).deposit(amt)  
   
 def withdraw(self,amt):  
 self.withdrawl\_fee()  
 super(EvilAccount,self).withdraw(amt)

In [ ]:

a = EvilAccount('Jay', 1000.00)  
a.deposit(200.0)  
print a.enquire()  
a.withdraw(10.0)  
print a.enquire()

In [ ]:

Method Resolution Order (MRO) and Duck Typing**¶**

* Order in which Methods are resolved
  + Depth-first, left to right
* Most specialized class should be left-most, most base right-most
* Method binding independent of object type
  + obj.name will work for any object that has an attribute name
* This is called **Duck Typing**

**"If it looks like a duck, walks like a duck, quacks like a duck .. I'd call it a duck!**

In [ ]:

EvilAccount.\_\_mro\_\_

In [ ]:

import inspect  
inspect.getmro(EvilAccount)

Static methods**¶**

In [ ]:

class Foo(object):  
 @staticmethod  
 def add(x,y):  
 return x + y

In [ ]:

x = Foo.add(3,4) # x = 7

* No self
* decorated with @staticmethod
* Commonly used for defining custom constructors

In [ ]:

import time  
class Date(object):  
 def \_\_init\_\_(self,year,month,day):  
 self.year = year   
 self.month = month  
 self.day = day   
   
 @staticmethod  
 def now():  
 t = time.localtime()  
 return Date(t.tm\_year, t.tm\_mon, t.tm\_mday)   
   
 @staticmethod  
 def tomorrow():  
 t = time.localtime(time.time()+86400)   
 return Date(t.tm\_year, t.tm\_mon, t.tm\_mday)

In [ ]:

# Example of creating some dates  
a = Date(1967, 4, 9)  
b = Date.now() # Calls static method now()  
c = Date.tomorrow() # Calls static method tomorrow()

Class methods**¶**

* Operate on the class itself
* Defined using @classmethod decorator
* Take the class object as the first argument

In [ ]:

class EuroDate(Date):  
 pass  
d = EuroDate.now()  
type(d)

In [ ]:

class Date(object):  
 def \_\_init\_\_(self,year,month,day):  
 self.year = year   
 self.month = month  
 self.day = day   
   
 @classmethod  
 def now(cls):  
 t = time.localtime(time.time())  
 return cls(t.tm\_year, t.tm\_mon, t.tm\_mday)

In [ ]:

class EuroDate(Date):  
 pass  
d = EuroDate.now()  
type(d)

In [ ]:

Magic Methods & Operator Overloading**¶**

* Methods with **d**ouble-**under**scores are special
  + **dunder** or magic methods
* Implementing classes gain support for a variety of constructs
  + Arithmetic operators overloading
  + Comparison operators overloading
  + Logical operators overloading
  + Contexts
  + Object representations
  + Type casting

In [ ]:

class Complex(object):  
 def \_\_init\_\_(self,real,imag=0):  
 self.real = float(real)  
 self.imag = float(imag)  
 def \_\_repr\_\_(self):  
 return "Complex(%s,%s)" % (self.real, self.imag)   
 def \_\_str\_\_(self):   
 return "(%g+%gj)" % (self.real, self.imag)  
 def \_\_add\_\_(self,other): # self + other   
 return Complex(self.real + other.real, self.imag + other.imag)   
 def \_\_sub\_\_(self,other): # self - other   
 return Complex(self.real - other.real, self.imag - other.imag)

In [ ]:

c1 = Complex(1.0,2.0)  
c2 = Complex(2.0,5.0)  
c3 = c1+c2 # Complex.\_\_add\_\_ called  
c3 # Complex.\_\_repr\_\_ called

In [ ]:

print c1+c2 # Complex.\_\_str\_\_

Dunders for comparison and logic ops**¶**

* \_\_lt\_\_, \_\_le\_\_, \_\_gt\_\_, \_\_ge\_\_, \_\_eq\_\_, \_\_ne\_\_
* Defining any two defines the rest
* \_\_and\_\_, \_\_or\_\_, \_\_not\_\_

Properties and Descriptors**¶**

* Normal (public) data attributes readable/writable directly
* Many classes define getters/setters to hide data
* Property is a special attribute that calculates its value when accessed
* Has a setter equivalent too

In [ ]:

class Foo(object):  
 def \_\_init\_\_(self,name):   
 self.\_\_name = name   
 @property  
 def name(self):   
 return self.\_\_name   
 @name.setter   
 def name(self,value):   
 if not isinstance(value,str):   
 raise TypeError("Must be a string")   
 self.\_\_name = value   
 @name.deleter   
 def name(self):  
 raise TypeError("Can't delete name")

In [ ]:

g = Foo('Guido')  
print g.name  
g.name = 'Monty'  
print g.name  
g.name = 45  
del g.name

* Any object that defines \_\_get\_\_, \_\_set\_\_ and \_\_dell\_\_ is acceptable as a property
* This is the Descriptor interface
* Descriptors instantiable only at the class level, not instance level

In [ ]:

class TypedProperty(object):  
 def \_\_init\_\_(self,name,type,default=None):  
 self.name = "\_" + name  
 self.type = type  
 self.default = default if default else type()  
 def \_\_get\_\_(self,instance,cls):   
 return getattr(instance,self.name,self.default)   
 def \_\_set\_\_(self,instance,value):  
 if not isinstance(value,self.type):  
 raise TypeError("Must be a %s" % self.type)   
 setattr(instance,self.name,value)   
 def \_\_delete\_\_(self,instance):  
 raise AttributeError("Can't delete attribute")  
   
class Foo(object):  
 name = TypedProperty("name",str)   
 num = TypedProperty("num",int,42)

In [ ]:

f = Foo()  
a = f.name   
f.name = "Guido"   
del f.name

In [ ]: