Object-Oriented Programming

Functions vs. Classes

```
data = None
def setdata(value):
    global data
    data = value

def display():
    global data
    print data

setdata("King Arthur")
display()

setdata(3.1419)
display()

data = "New value"
display()
```

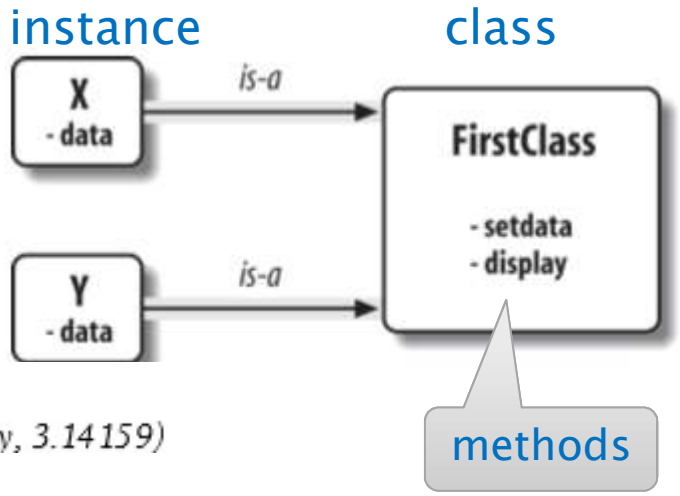
```
class MyClass:
    data = None
    @staticmethod
    def setdata(value):
        MyClass.data = value
    @staticmethod
    def display():
        print MyClass.data

MyClass.setdata("King Arthur")
MyClass.display()

MyClass.setdata(3.1419)
MyClass.display()

MyClass.data = "New value"
MyClass.display()
```

Encapsulation

`object.attribute`

Classes, Instances, and Attributes

```
>>> class FirstClass:                    # Define a class object
        def setdata(self, value):        # Define class's methods
            self.data = value            # self is the instance
        def display(self):
            print(self.data)             # self.data: per instance

>>> x = FirstClass()                     # Make two instances
>>> y = FirstClass()                     # Each is a new namespace

>>> x.setdata("King Arthur")             # Call methods: self is x
>>> y.setdata(3.14159)                   # Runs: FirstClass.setdata(y, 3.14159)

>>> x.display()                          # self.data differs in each instance
King Arthur
>>> y.display()                          # Runs: FirstClass.display(y)
3.14159

>>> x.data = "New value"                 # Can get/set attributes
>>> x.display()                          # Outside the class too
New value
```



equivalent to:
`FirstClass.display(y)`

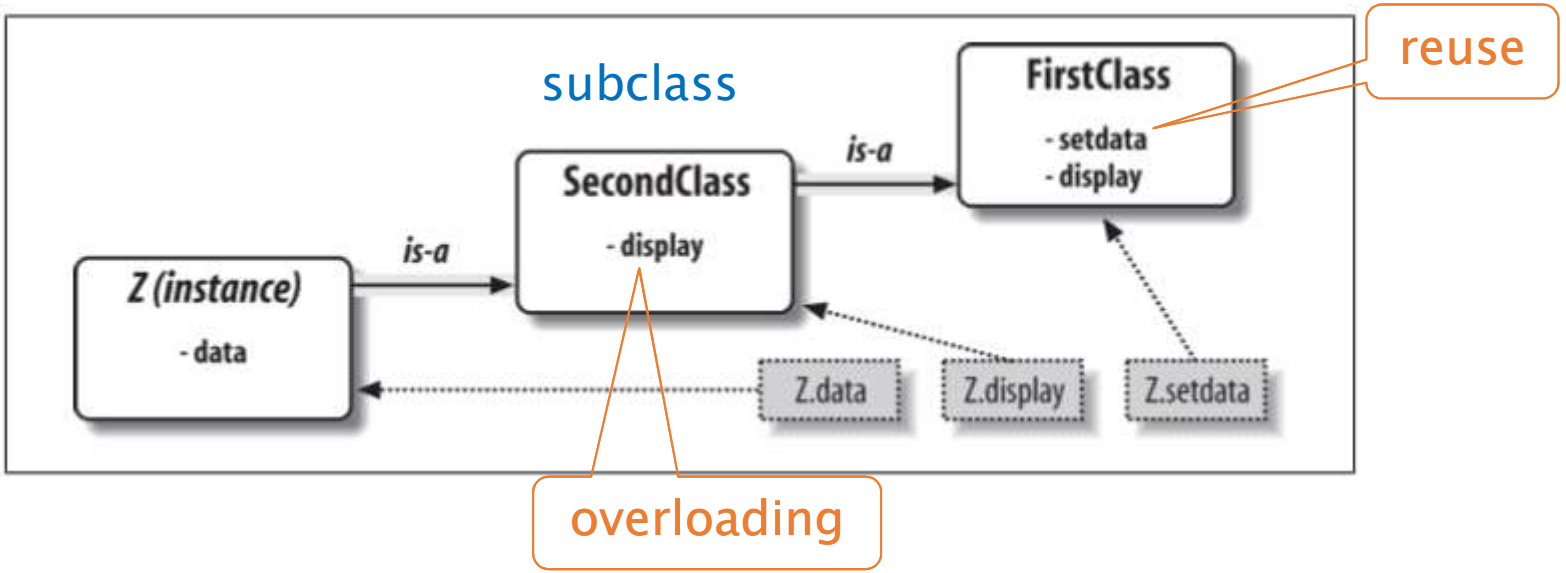
- Class objects provide default behavior and serve as factories for instance objects.
- Instance objects are the real objects your programs process—each is a namespace in its own right, but inherits

Classes Are Customized by Inheritance

```
>>> class SecondClass(FirstClass):                    # Inherits setdata
        def display(self):                            # Changes display
            print('Current value = "%s"' % self.data)

>>> z = SecondClass()
>>> z.setdata(42)                  # Finds setdata in FirstClass
>>> z.display()                    # Finds overridden method in SecondClass
Current value = "42"


>>> x.display()
New value
```

OOP is about code reuse!!

```
class name(superclass,...):                # Assign to name
    attr = value                           # Shared class data
    def method(self,...):                  # Methods
        self.attr = value                  # Per-instance data
```



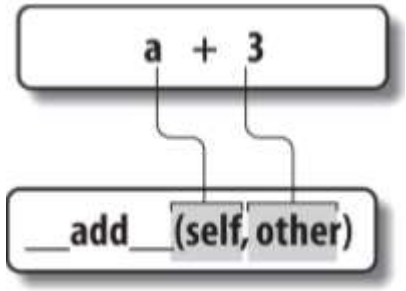
Operator Overloading

- Classes can intercept Python operators

```
>>> class ThirdClass(SecondClass):                    # Inherit from SecondClass
        def __init__(self, value):                    # On "ThirdClass(value)"
            self.data = value
        def __add__(self, other):                     # On "self + other"
            return ThirdClass(self.data + other)
        def __str__(self):                            # On "print(self)", "str()"
            return '[ThirdClass: %s]' % self.data

        def mul(self, other):                         # In-place change: named
            self.data *= other
```

```
>>> a = ThirdClass('abc')                             # __init__ called
>>> a.display()                                       # Inherited method called
Current value = "abc"
>>> print(a)                                          # __str__: returns display string
[ThirdClass: abc]

>>> b = a + 'xyz'                                     # __add__: makes a new instance
>>> b.display()                                       # b has all ThirdClass methods
Current value = "abcxyz"
>>> print(b)                                          # __str__: returns display string
[ThirdClass: abcxyz]

>>> a.mul(3)                                          # mul: changes instance in place
>>> print(a)
[ThirdClass: abcabcabc]
```

```
>>> rec = ('Bob', 40.5, ['dev', 'mgr'])
```



Classes vs. Dictionaries

tuple-based record

```
>>> rec = ('Bob', 40.5, ['dev', 'mgr'])
```

dict-based record

```
>>> rec = {}
>>> rec['name'] = 'Bob'
>>> rec['age']  = 40.5
>>> rec['jobs'] = ['dev', 'mgr']
```

instance-based records

```
>>> class rec: pass

>>> pers1 = rec()
>>> pers1.name = 'Bob'
>>> pers1.jobs = ['dev', 'mgr']
>>> pers1.age  = 40.5
>>>
>>> pers2 = rec()
>>> pers2.name = 'Sue'
>>> pers2.jobs = ['dev', 'cto']
>>>
>>> pers1.name, pers2.name
('Bob', 'Sue')
```

class-based record

```
>>> class rec: pass

>>> rec.name = 'Bob'
>>> rec.age  = 40.5
>>> rec.jobs = ['dev', 'mgr']
```

OO records

```
>>> class Person:
        def __init__(self, name, jobs, age=None):
            self.name = name
            self.jobs = jobs
            self.age  = age
        def info(self):
            return (self.name, self.jobs)

>>> rec1 = Person('Bob', ['dev', 'mgr'], 40.5)
>>> rec2 = Person('Sue', ['dev', 'cto'])
>>>
>>> rec1.jobs, rec2.info()
(['dev', 'mgr'], ('Sue', ['dev', 'cto']))
```

More Realistic Examples

person.py:

```
class Person:
    def __init__(self, name, pay=0, job=None):
        self.name = name
        self.pay  = pay
        self.job  = job
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def __init__(self, name, pay):
        Person.__init__(self, name, pay, 'manager')
    def giveRaise(self, percent, bonus=0.1):
        Person.giveRaise(self, percent + bonus)
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', 40000, 'hardware')
    print bob
    print sue
    print bob.lastName(), sue.lastName()
    sue.giveRaise(0.1)
    print sue
    tom = Manager(name='Tom Doe', pay=50000)
    tom.giveRaise(0.1)
    print tom.lastName()
    print tom
```

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 40000]
Smith Jones
[Person: Sue Jones, 44000.0
Doe
[Person: Tom Doe, 60000.0]
>>>
```

Composite Objects: embedding objs

```
class Person:
    ...same...

class Manager(Person):
    ...same...

class Department:
    def __init__(self, *args):
        self.members = list(args)
    def addMember(self, person):
        self.members.append(person)
    def giveRaises(self, percent):
        for person in self.members:
            person.giveRaise(percent)
    def showAll(self):
        for person in self.members:
            print(person)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    tom = Manager('Tom Jones', 50000)

    development = Department(bob, sue)          # Embed objects in a composite
    development.addMember(tom)
    development.giveRaises(.10)                 # Runs embedded objects' giveRaise
    development.showAll()                       # Runs embedded objects' __repr__
```

Special Class Attributes

```
>>> from person import Person
>>> bob = Person('Bob Smith')
>>> bob                                  # Show bob's __repr__ (not __str__)
[Person: Bob Smith, 0]
>>> print(bob)                           # Ditto: print => __str__ or __repr__
[Person: Bob Smith, 0]

>>> bob.__class__                        # Show bob's class and its name
<class 'person.Person'>
>>> bob.__class__.__name__
'Person'

>>> list(bob.__dict__.keys())            # Attributes are really dict keys
['pay', 'job', 'name']                   # Use list to force list in 3.X

>>> for key in bob.__dict__:
        print(key, '=>', bob.__dict__[key])    # Index manually

pay => 0
job => None
name => Bob Smith

>>> for key in bob.__dict__:
        print(key, '=>', getattr(bob, key))    # obj.attr, but attr is a var

pay => 0
job => None
name => Bob Smith
```

```
>>> dir(bob)                                   # Plus inherited attrs in classes
['__doc__', '__init__', '__module__', '__repr__', 'giveRaise', 'job', 'lastName',
'name', 'pay']
```

Class Interface Techniques

```
class Super:
    def method(self):
        print('in Super.method')            # Default behavior
    def delegate(self):
        self.action()                       # Expected to be defined

class Inheritor(Super):                     # Inherit method verbatim
    pass

class Replacer(Super):                      # Replace method completely
    def method(self):
        print('in Replacer.method')

class Extender(Super):                      # Extend method behavior
    def method(self):
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')

class Provider(Super):                      # Fill in a required method
    def action(self):
        print('in Provider.action')
```

abstract super class

implementation of action()

```
x = Provider()
x.delegate()
```

??

Namespaces: Assignments classify names

```
X = 11                          # Global (module) name/attribute (X, or manynames.X)

def f():
    print(X)                    # Access global X (11)

def g():
    X = 22                      # Local (function) variable (X, hides module X)
    print(X)

class C:
    X = 33                      # Class attribute (C.X)
    def m(self):
        X = 44                  # Local variable in method (X)
        self.X = 55             # Instance attribute (instance.X)
```

```
X = 11                          # Global in module

def g1():
    print(X)                    # Reference global in module (11)

def g2():
    global X
    X = 22                      # Change global in module

def h1():
    X = 33                      # Local in function
    def nested():
        print(X)                # Reference local in enclosing scope (33)

def h2():
    X = 33                      # Local in function
    def nested():
        nonlocal X              # Python 3.X statement
        X = 44                  # Change local in enclosing scope
```

```
X = 1

def nester():
   print(X)                     # Global: 1
   class C:
       print(X)                 # Global: 1
       def method1(self):
           print(X)             # Global: 1
       def method2(self):
           X = 3                # Hides global
           print(X)             # Local: 3
   I = C()
   I.method1()
   I.method2()

print(X)                        # Global: 1
nester()                        # Rest: 1, 1, 1, 3
print('-'*40)
```

Common Operator Overloading Methods

Method	Implements	Called for
`__init__`	Constructor	Object creation: `X = Class(args)`
`__del__`	Destructor	Object reclamation of X
`__add__`	Operator +	`X + Y`, `X += Y` if no `__iadd__`
`__or__`	Operator | (bitwise OR)	`X | Y`, `X |= Y` if no `__ior__`
`__repr__`, `__str__`	Printing, conversions	`print(X)`, `repr(X)`, `str(X)`
`__call__`	Function calls	`X(*args, **kargs)`
`__getattr__`	Attribute fetch	`X.undefined`
`__setattr__`	Attribute assignment	`X.any = value`
`__delattr__`	Attribute deletion	`del X.any`
`__getattribute__`	Attribute fetch	`X.any`
`__getitem__`	Indexing, slicing, iteration	`X[key]`, `X[i:j]`, `for` loops and other iterations if no `__iter__`
`__setitem__`	Index and slice assignment	`X[key] = value`, `X[i:j] = iterable`
`__delitem__`	Index and slice deletion	`del X[key]`, `del X[i:j]`

Common Operator Overloading Methods

`__len__`	Length	`len(X)`, truth tests if no `__bool__`
`__bool__`	Boolean tests	`bool(X)`, truth tests (named `__nonzero__` in 2.X)
`__lt__, __gt__, __le__, __ge__, __eq__, __ne__`	Comparisons	`X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y` (or else `__cmp__` in 2.X only)
`__radd__`	Right-side operators	`Other + X`
`__iadd__`	In-place augmented operators	`X += Y` (or else `__add__`)
`__iter__, __next__`	Iteration contexts	`I=iter(X)`, `next(I)`; `for` loops, `in` if no `__contains__`, all comprehensions, `map(F,X)`, others (`__next__` is named `next` in 2.X)
`__contains__`	Membership test	`item in X` (any iterable)
`__index__`	Integer value	`hex(X)`, `bin(X)`, `oct(X)`, `O[X]`, `O[X:]` (replaces 2.X `__oct__`, `__hex__`)
`__enter__, __exit__`	Context manager (Chapter 34)	`with obj as var:`
`__get__, __set__, __delete__`	Descriptor attributes (Chapter 38)	`X.attr, X.attr = value, del X.attr`
`__new__`	Creation (Chapter 40)	Object creation, before `__init__`

Comparisons: __lt__, __gt__, __le__, __ge__, __eq__, __ne__

```
class C:
    data = 'spam'
    def __gt__(self, other):                # 3.X and 2.X version
        return self.data > other
    def __lt__(self, other):
        return self.data < other

X = C()
print(X > 'ham')                            # True  (runs __gt__)
print(X < 'ham')                            # False (runs __lt__)
```

In Python 2.X, a __cmp__ method is used if above methods are not defined

- cmp, __cmp__ are removed in Python 3.X

```
class C:
    data = 'spam'                           # 2.X only
    def __cmp__(self, other):               # __cmp__ not used in 3.X
        return cmp(self.data, other)        # cmp not defined in 3.X

X = C()
print(X > 'ham')                            # True  (runs __cmp__)
print(X < 'ham')                            # False (runs __cmp__)
```

Indexing and Slicing: __getitem__ and __setitem__

```
>>> class Indexer:
...     def __init__(self):
...         self.data = [5, 6, 7, 8, 9]
...     def __repr__(self):
...         return repr(self.data)
...     def __getitem__(self, index):   # Called for index or slice
...         return self.data[index]     # Perform index or slice
...     def __setitem__(self, index, value):
...         self.data[index] = value
...
>>> x = Indexer()
>>> y = Indexer()
>>> x
[5, 6, 7, 8, 9]
>>> print x
[5, 6, 7, 8, 9]
>>> x[0]
5
>>> x[2:4]
[7, 8]
>>> x[0] = 0
>>> x
[0, 6, 7, 8, 9]
>>> y
[5, 6, 7, 8, 9]
```

```
>>> for item in x:
...     print item,
...
0 6 7 8 9
>>> [n for n in x]
[0, 6, 7, 8, 9]
>>> sum(x)
30
```

Iterable Objects: __iter__ and next (__next__ in Python 3)

```
>>> class Squares:
...     def __init__(self, start, stop):
...         self.value = start - 1
...         self.stop  = stop
...     def __iter__(self):
...         return self
...     def next(self):
...         if self.value == self.stop:
...             raise StopIteration
...         self.value += 1
...         return self.value ** 2
...
>>> for i in Squares(1,5):
...     print i,
...
1 4 9 16 25
>>> x = Squares(1,5)
>>> [n for n in x]
[1, 4, 9, 16, 25]
>>> 36 in Squares(1,10)
True
>>> x[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Squares instance has no attribute '__getitem__'
```

Only if no such __iter__ method is found, Python falls back on the __getitem__ scheme

generator functions and expressions

```
>>> for i in gsquares(1,5):
...     print i,
...
1 4 9 16 25
>>> for i in (x ** 2 for x in range(1, 6)):
...     print i,
...
1 4 9 16 25
```

Attribute Access: __getattr__ and __setattr__

```
>>> class Empty:
        def __getattr__(self, attrname):
            if attrname == 'age':
                return 40
            else:
                raise AttributeError(attrname)

>>> X = Empty()
>>> X.age
40
>>> X.name
...error text omitted...
AttributeError: name
```

__getattr__ is called whenever attr is undefined

__setattr__ always intercepts X.attr = value

```
>>> class Accesscontrol:
        def __setattr__(self, attr, value):
            if attr == 'age':
                self.__dict__[attr] = value + 10        # Not self.name=val or setattr
            else:
                raise AttributeError(attr + ' not allowed')

>>> X = Accesscontrol()
>>> X.age = 40                                          # Calls __setattr__
>>> X.age
50
>>> X.name = 'Bob'
...text omitted...
AttributeError: name not allowed
```

Function Interfaces and Callback-Based Code

The __call__ method is called when your instance is called.

```
>>> class Callee:
        def __call__(self, *pargs, **kargs):        # Intercept instance calls
            print('Called:', pargs, kargs)          # Accept arbitrary arguments

>>> C = Callee()
>>> C(1, 2, 3)                                      # C is a callable object
Called: (1, 2, 3) {}
>>> C(1, 2, 3, x=4, y=5)
Called: (1, 2, 3) {'y': 5, 'x': 4}
```

As an example, the tkinter GUI toolkit (named Tkinter in Python 2.X) allows you to register functions as event handlers (a.k.a. callbacks)

```
class Callback:
    def __init__(self, color):                     # Function + state information
        self.color = color
    def __call__(self):                             # Support calls with no arguments
        print('turn', self.color)

# Handlers
cb1 = Callback('blue')                              # Remember blue
cb2 = Callback('green')                             # Remember green

B1 = Button(command=cb1)                            # Register handlers
B2 = Button(command=cb2)
```

Polymorphism Means Interfaces, Not Call Signatures

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

Do not use C++ style.

```
class C:
    def meth(self, *args):
        if len(args) == 1:
            ...
        elif type(arg[0]) == int:
            ...
```

var. arguments and type-testing

```
class C:
    def meth(self, x):
        x.operation()
```

write your code to expect only an object interface, not a specific data type.

OOP and Inheritance: "Is-a" Relationships

employees.py:

```
from __future__ import print_function

class Employee:
    def __init__(self, name, salary=0):
        self.name   = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print(self.name, "does stuff")
    def __repr__(self):
        return "<Employee: name=%s, salary=%s>" % (self.name, self.salary)

class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print(self.name, "makes food")

class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print(self.name, "interfaces with customer")

class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print(self.name, "makes pizza")
```

Classes and Persistency

Instances of classes can be stored away on disk using Python's pickle or shelve modules

```
>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.server, shop.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)
>>> import pickle
>>> pickle.dump(shop, open('shopfile.pkl', 'wb'))
```

```
>>> import pickle
>>> obj = pickle.load(open('shopfile.pkl', 'rb'))
>>> obj.server, obj.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)

>>> obj.order('LSP')
LSP orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
LSP pays for item to <Employee: name=Pat, salary=40000>
```

OOP and Composition: "Has-a" Relationships

pizzashop.py:

```
from __future__ import print_function
from employees import PizzaRobot, Server

class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print(self.name, "orders from", server)
    def pay(self, server):
        print(self.name, "pays for item to", server)

class Oven:
    def bake(self):
        print("oven bakes")

class PizzaShop:
    def __init__(self):
        self.server = Server('Pat')          # Embed other objects
        self.chef   = PizzaRobot('Bob')      # A robot named bob
        self.oven   = Oven()

    def order(self, name):
        customer = Customer(name)            # Activate other objects
        customer.order(self.server)          # Customer orders from server
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)
```

```
if __name__ == "__main__":
    scene = PizzaShop()
    scene.order('Homer')
    print('...')
    scene.order('Shaggy')
```

PizzaShop is:

a container

controller

Example: stream processor

stream.py:

```
class Processor:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer

    def process(self):
        while True:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)

    def converter(self, data):
        assert False, 'converter must be defined'
```

abstract super class

delegates implementation

```
from streams import Processor

class Uppercase(Processor):
    def converter(self, data):
        return data.upper()

if __name__ == '__main__':
    import sys
    obj = Uppercase(open('trispam.txt'), sys.stdout)
    obj.process()
```

implementation

OOP and Delegation: "Wrapper" Proxy Objects

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object
    def __getattr__(self, attrname):
        print('Trace: ' + attrname)
        return getattr(self.wrapped, attrname)
```

like self.attrname
if attrname is undefined

getattr(X,N) is like X.N,
except that N is an expression that evaluates to a string at runtime, not a variable.

```
>>> from trace import Wrapper
>>> x = Wrapper([1, 2, 3])                  # Wrap a list
>>> x.append(4)                             # Delegate to list method
Trace: append
>>> x.wrapped                               # Print my member
[1, 2, 3, 4]

>>> x = Wrapper({'a': 1, 'b': 2})           # Wrap a dictionary
>>> list(x.keys())                          # Delegate to dictionary method
Trace: keys
['a', 'b']
```

Classes Are Objects: Generic Object Factories

```
def factory(aClass, *pargs, **kargs):        # Varargs tuple, dict
    return aClass(*pargs, **kargs)           # Call aClass (or apply in 2.X only)

class Spam:
    def doit(self, message):
        print(message)

class Person:
    def __init__(self, name, job=None):
        self.name = name
        self.job  = job

object1 = factory(Spam)                      # Make a Spam object
object2 = factory(Person, "Arthur", "King")  # Make a Person object
object3 = factory(Person, name='Brian')      # Ditto, with keywords and default
```

```
>>> object1.doit(99)
99
>>> object2.name, object2.job
('Arthur', 'King')
>>> object3.name, object3.job
('Brian', None)
```

Extending Types by Embedding

```
class Set:
    def __init__(self, value = []):    # Constructor
        self.data = []                  # Manages a list
        self.concat(value)

    def intersect(self, other):         # other is any sequence
        res = []                        # self is the subject
        for x in self.data:
            if x in other:              # Pick common items
                res.append(x)
        return Set(res)                 # Return a new Set

    def union(self, other):             # other is any sequence
        res = self.data[:]              # Copy of my list
        for x in other:                 # Add items in other
            if not x in res:
                res.append(x)
        return Set(res)

    def concat(self, value):            # value: list, Set...
        for x in value:                 # Removes duplicates
           if not x in self.data:
                self.data.append(x)

    def __len__(self):          return len(self.data)
    def __getitem__(self, key): return self.data[key]
    def __and__(self, other):   return self.intersect(other)
    def __or__(self, other):    return self.union(other)
    def __repr__(self):         return 'Set:' + repr(self.data)
    def __iter__(self):         return iter(self.data)
```

Extending Types by Subclassing

```
# Subclass built-in list type/class
# Map 1..N to 0..N-1; call back to built-in version.

class MyList(list):
    def __getitem__(self, offset):
        print('(indexing %s at %s)' % (self, offset))
        return list.__getitem__(self, offset - 1)

if __name__ == '__main__':
    print(list('abc'))
    x = MyList('abc')              # __init__ inherited from list
    print(x)                       # __repr__ inherited from list

    print(x[1])                    # MyList.__getitem__
    print(x[3])                    # Customizes list superclass method

    x.append('spam'); print(x)     # Attributes from list superclass
    x.reverse();      print(x)
```

Extending Types by Subclassing

```
from __future__ import print_function    # 2.X compatibility

class Set(list):
    def __init__(self, value = []):     # Constructor
        list.__init__([])                # Customizes list
        self.concat(value)               # Copies mutable defa

    def intersect(self, other):          # other is any sequence
        res = []                         # self is the subject
        for x in self:
            if x in other:               # Pick common items
                res.append(x)
        return Set(res)                  # Return a new Set

    def union(self, other):              # other is any sequence
        res = Set(self)                  # Copy me and my list
        res.concat(other)
        return res

    def concat(self, value):             # value: list, Set, etc.
        for x in value:                  # Removes duplicates
            if not x in self:
                self.append(x)

    def __and__(self, other): return self.intersect(other)
    def __or__(self, other):  return self.union(other)
    def __repr__(self):       return 'Set:' + list.__repr__(self)
```

```
if __name__ == '__main__':
    x = Set([1,3,5,7])
    y = Set([2,1,4,5,6])
    print(x, y, len(x))
    print(x.intersect(y), y.union(x))
    print(x & y, x | y)
    x.reverse(); print(x)
```

```
% python setsubclass.py
Set:[1, 3, 5, 7] Set:[2, 1, 4, 5, 6] 4
Set:[1, 5] Set:[2, 1, 4, 5, 6, 3, 7]
Set:[1, 5] Set:[1, 3, 5, 7, 2, 4, 6]
Set:[7, 5, 3, 1]
```

Static and class method

```
class Methods(object):
    def imeth(self, x):
        print([self, x])

    @staticmethod
    def smeth(x):
        print([x])

    @classmethod
    def cmeth(cls, x):
        print([cls, x])
```

Counting instances

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

    @staticmethod
    def printNumInstances():
        print("Number of instances created: %s" % Spam.numInstances)
```

```
>>> from spam_static_deco import Spam
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()          # Calls from classes and instances work
Number of instances created: 3
>>> a.printNumInstances()
Number of instances created: 3
```

Why OOP?

- Code reuse
  - by supporting inheritance
- Encapsulation
  - Wrapping up implementation details behind object interfaces
- Structure
  - Classes provide new local scopes, which minimizes name clashes
- Maintenance
  - usually only one copy of the code needs to be changed
- Consistency
  - Classes and inheritance allow you to implement common interfaces
- polymorphism
  - makes code more flexible and widely applicable, and hence more reusable