Polymorphism

- The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

```
# Python program to demonstrate in-built poly-
# morphic functions

# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

```
class Animal:
    def Name(self):
        pass
    def Sleep(self):
        print('sleep')
    def makeNoise(self):
        pass
class Dog(Animal):
    def Name(self):
        print('I am a dog!')
    def makeNoise(self):
        print('Woof! Woof!')

class Cat(Animal):
    def Name(self):
        print('I am a cat!')
    def makeNoise(self):
        print('Meow! Meow!')

class Lion(Animal):
    def Name(self):
        print('I am a lion!')
    def makeNoise(self):
        print('Roar! Roar!')

class TestAnimals:
    def printName(self, animal):
        animal.Name()
    def goToSleep(self, animal):
        animal.Sleep()
    def makeNoise(self,animal):
        animal.makeNoise()
```

```
class TestAnimals:
    def printName(self, animal):
        animal.Name()
    def goToSleep(self, animal):
        animal.Sleep()
    def makeNoise(self,animal):
        animal.makeNoise()

TestAnimal=TestAnimals()
dog=Dog()
cat=Cat()
lion=Lion()

for animal in (dog, cat, lion):
    TestAnimal.printName(animal)
    TestAnimal.goToSleep(animal)
    TestAnimal.makeNoise(animal)

#output:
    I am a dog!
    sleep
    Woof! Woof!
    I am a cat!
    sleep
    Meow! Meow!
    I am a lion!
    sleep
    Roar! Roar!
```

Operator Overloading

- The concept of overloading is also a branch of polymorphism.
- In operator overloading different operators have different implementations depending on their arguments.
- Python operators work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings. This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

```
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('Data in first class: ', self.data)


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

class ThirdClass(SecondClass):                              # Inherit from SecondC
    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)", "s
        return '[ThirdClass: %s]' % self.data
    def mul(self, other):                                   # In-place change: nam
        self.data *= other


x = FirstClass()
z = SecondClass()
x.setdata(10)
z.setdata("Second King")
x.display()
z.display()

a = ThirdClass('abc')                # __init__ called
a.display()                          # Inherited method called
print(a)                             # __str__: returns display string
b = a + 'xyz'                        # __add__: makes a new instance
b.display()                          # b has all ThirdClass methods
print(b)                             # __str__: returns display string
a.mul(3)                             # mul: changes instance in place
print(a)
```

```
x = FirstClass()
z = SecondClass()
x.setdata(10)
z.setdata("Second King")
x.display()
z.display()

a = ThirdClass('abc')
a.display()
print(a)
b = a + 'xyz'
b.display()
print(b)
a.mul(3)
print(a)
```

```
class ThirdClass(SecondClass):
    def __init__(self, value):
        self.data = value
    def __add__(self, other):
        return ThirdClass(self.data + other)
    def __str__(self):
        return '[ThirdClass: %s]' % self.data
    def mul(self, other):
        self.data *= other
```

```
#output
   Data in first class:  10
   Current value = "Second King"
   Current value = "abc"
   [ThirdClass: abc]
   Current value = "abcxyz"
   [ThirdClass: abcxyz]
   [ThirdClass: abcabcabc]
```

An Example

```
class Person:
    def __init__(self, name, job=None, pay=0):
                                            # Constructor takes three arguments
        self.name = name                    # Fill out fields when created
        self.job  = job                     # self is the new instance object
        self.pay  = pay

bob = Person('Bob Smith')                           # Test the class
sue = Person('Sue Jones', job='dev', pay=100000)    # Runs __init__ automatically
print(bob.name, bob.pay)                            # Fetch attached attributes
print(sue.name, sue.pay)                            # sue's and bob's attrs differ

#output:

    Bob Smith 0
    Sue Jones 100000
```

Using Code Two Ways

```
class Person:
    def __init__(self, name, job=None, pay=0):
                                            # Constructor takes three arguments
        self.name = name                    # Fill out fields when created
        self.job  = job                     # self is the new instance object
        self.pay  = pay

if __name__ == '__main__':                  # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1])             # Extract object's last name
    sue.pay *= 1.10                         # Give this object a raise
    print('%.2f' % sue.pay)

#output:
    Bob Smith 0
    Sue Jones 100000
    Smith
    110000.00
```

Adding Behavior Methods

```
class Person:
    def __init__(self, name, job=None, pay=0):
                                            # Constructor takes three arguments
        self.name = name                    # Fill out fields when created
        self.job  = job                     # self is the new instance object
        self.pay  = pay
    def lastName(self):                                 # Behavior methods
        return self.name.split()[-1]                    # self is implied subject
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))        # Must change here only

if __name__ == '__main__':                  # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())               # Use the new methods
    sue.giveRaise(.10)                                  # instead of hardcoding
    print(sue.pay)


#output:
    Bob Smith 0
    Sue Jones 100000
    Smith Jones
    110000
```

Operator Overloading

```
class Person:
    def __init__(self, name, job=None, pay=0):
                                          # Constructor takes three arguments
        self.name = name                  # Fill out fields when created
        self.job  = job                   # self is the new instance object
        self.pay  = pay
    def lastName(self):                           # Behavior methods
        return self.name.split()[-1]              # self is implied subject
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))  # Must change here only
    def __repr__(self):                               # Added method
        return '[Person: %s, %s]' % (self.name, self.pay)  # String to print
if __name__ == '__main__':                    # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)

#output:
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 100000]
    Smith Jones
    [Person: Sue Jones, 110000]
```

Customizing Behavior by Subclassing

```
class Person:
    def __init__(self, name, job=None, pay=0):
                                          # Constructor takes three arguments
        self.name = name                  # Fill out fields when created
        self.job  = job                   # self is the new instance object
        self.pay  = pay
    def lastName(self):                           # Behavior methods
        return self.name.split()[-1]              # self is implied subject
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))  # Must change here only
    def __repr__(self):                                 # Added method
        return '[Person: %s, %s]' % (self.name, self.pay)   # String to print

class Manager(Person):                        # Define a subclass of Person
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)         # Good: augment original
```

Customizing Behavior by Subclassing

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)          # Make a Manager: __init__
    tom.giveRaise(.10)                                # Runs custom version
    print(tom.lastName())                             # Runs inherited method
    print(tom)                                        # Runs inherited __repr__

#output:
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 100000]
    Smith Jones
    [Person: Sue Jones, 110000]
    Jones
    [Person: Tom Jones, 60000]
```

Polymorphism in Action

```
class Person:
    def __init__(self, name, job=None, pay=0):       # Constructor takes
        self.name = name                             # Fill out fields when crea
        self.job  = job                              # self is the new instance
        self.pay  = pay
    def lastName(self):                                # Behavior methods
        return self.name.split()[-1]                   # self is implied
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))       # Must change here
    def __repr__(self):                                          # Added m
        return '[Person: %s, %s]' % (self.name, self.pay)        # String

class Manager(Person):                               # Define a subclass of P
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)                # Good: augme
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)            # Make
    tom.giveRaise(.10)                                  # Runs
    print(tom.lastName())                               # Runs
    print(tom)                                          # Runs
    print('--All three--')
    for obj in (bob, sue, tom):                 # Process objec
        obj.giveRaise(.10)                      # Run this obje
        print(obj)                              # Run the comm


#output:
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 100000]
    Smith Jones
    [Person: Sue Jones, 110000]
    Jones
    [Person: Tom Jones, 60000]
    --All three--
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 121000]
    [Person: Tom Jones, 72000]
```

Polymorphism in Action

```
class Person:
    def __init__(self, name, job=None, pay=0):       # Constructor
        self.name = name                             # Fill out fields whe
        self.job  = job                              # self is the new ins
        self.pay  = pay
    def lastName(self):                                # Behavior m
        return self.name.split()[-1]                   # self is im
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))       # Must chang
    def __repr__(self):                                          # A
        return '[Person: %s, %s]' % (self.name, self.pay)        # S

class Manager(Person):                               # Define a subclas
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)                # Good:
    def giveDown(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent- bonus))
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)            # Make a l
    tom.giveRaise(.10)                                  # Runs cu
    tom.giveDown(.10)
    print(tom.lastName())                               # Runs in
    print(tom)                                          # Runs in
    print('--All three--')
    for obj in (bob, sue, tom):                 # Process objects
        obj.giveRaise(.10)                      # Run this object'
        print(obj)                              # Run the common __

#output:
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 100000]
    Smith Jones
    [Person: Sue Jones, 110000]
    Jones
    [Person: Tom Jones, 60000]
    --All three--
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 121000]
    [Person: Tom Jones, 72000]
```

Customizing Constructors

```
class Person:
    def __init__(self, name, job=None, pay=0):       # Constructor t
        self.name = name                             # Fill out fields wher
        self.job  = job                              # self is the new inst
        self.pay  = pay
    def lastName(self):                                # Behavior me
        return self.name.split()[-1]                   # self is imp
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))       # Must change
    def __repr__(self):                                          # Ad
        return '[Person: %s, %s]' % (self.name, self.pay)        # St

class Manager(Person):                               # Define a subclass
    def __init__(self, name, pay):                         # Redefine c
        Person.__init__(self, name, 'mgr', pay)            # Run origin
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)              # Good:
    def giveDown(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent- bonus))
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)                    # Job na
    tom.giveRaise(.10)                                   # Runs cu
    print(tom.lastName())                                # Runs in
    print(tom)                                           # Runs in
    print('--All three--')
    for obj in (bob, sue, tom):                  # Process objects
        obj.giveRaise(.10)                       # Run this object
        print(obj)                               # Run the common

#output:
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 100000]
    Smith Jones
    [Person: Sue Jones, 110000]
    Jones
    [Person: Tom Jones, 60000]
    --All three--
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 121000]
    [Person: Tom Jones, 72000]
```

Other Ways to Combine Classes

```
class Person:
    def __init__(self, name, job=None, pay=0):      # Constructor takes th
        self.name = name                            # Fill out fields when create
        self.job  = job                             # self is the new instance ob
        self.pay  = pay
    def lastName(self):                               # Behavior methods
        return self.name.split()[-1]                  # self is implied su
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))      # Must change here o
    def __repr__(self):                                           # Added met
        return '[Person: %s, %s]' % (self.name, self.pay)      # String to

class Manager(Person):                              # Define a subclass of Per
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)      # Embed a Person objec
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)      # Intercept and delega
    def giveDown(self, percent, bonus=.10):
        self.person.giveDown = int(self.pay * (1 + percent- bonus))
    def __getattr__(self, attr):
        return getattr(self.person, attr)           # Delegate all other a
    def __repr__(self):
        return str(self.person)                     # Must overload again
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)                    # Job nam
    tom.giveRaise(.10)                                   # Runs cus
    print(tom.lastName())                                # Runs inh
    print(tom)                                           # Runs inh
    print('--All three--')
    for obj in (bob, sue, tom):                  # Process objects g
        obj.giveRaise(.10)                       # Run this object's
        print(obj)                               # Run the common __


#output:
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 100000]
    Smith Jones
    [Person: Sue Jones, 110000]
    Jones
    [Person: Tom Jones, 60000]
    --All three--
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 121000]
    [Person: Tom Jones, 72000]
```

```
class Person:
    def __init__(self, name, job=None, pay=0):       # Constructor
        self.name = name                             # Fill out fields whe
        self.job  = job                              # self is the new ins
        self.pay  = pay
    def lastName(self):                                # Behavior m
        return self.name.split()[-1]                   # self is im
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))       # Must chang
    def __repr__(self):                                              # A
        return '[Person: %s, %s]' % (self.name, self.pay)      # S

class Manager(Person):                               # Define a subclas
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)       # Embed a Pers
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)       # Intercept an
    def __getattr__(self, attr):
        return getattr(self.person, attr)            # Delegate all
    def __repr__(self):
        return str(self.person)                      # Must overloa

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)
```

Other Ways to Combine Classes

```
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)           # Embe
    development.addMember(tom)
    development.giveRaises(.10)                  # Runs
    development.showAll()                        # Runs

#output:
    [Person: Bob Smith, 0]
    [Person: Sue Jones, 110000]
    [Person: Tom Jones, 60000]
```

Special Class Attributes

```
class Person:
    def __init__(self, name, job=None, pay=0):        # Construct
        self.name = name                              # Fill out fields
        self.job  = job                               # self is the new
        self.pay  = pay
    def lastName(self):                                 # Behavio
        return self.name.split()[-1]                    # self is
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))        # Must ch
    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

bob = Person('Bob Smith')
print(bob)                                              # Show bob's
print(bob.__class__)                                    # Show bob's
print(bob.__class__.__name__)
print(list(bob.__dict__.keys()))                        # Attributes
for key in bob.__dict__:
    print(key, '=>', bob.__dict__[key])     # Index manually
for key in bob.__dict__:
        print(key, '=>', getattr(bob, key))     # obj.attr, but

#output:
    [Person: Bob Smith, 0]
    <class '__main__.Person'>
    Person
    ['pay', 'name', 'job']
    pay => 0
    name => Bob Smith
    job => None
    pay => 0
    name => Bob Smith
    job => None
```

A Generic Display Tool

```
"Assorted class utilities and tools"
class AttrDisplay:
    """
    Provides an inheritable display overload method that shows
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __repr__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())
if __name__ == '__main__':
    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2
    class SubTest(TopTest):
        pass

    X, Y = TopTest(), SubTest()       # Make two instances
    print(X)                          # Show all instance attrs
    print(Y)                          # Show lowest class name
```

```
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __repr__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())
if __name__ == '__main__':
    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2
    class SubTest(TopTest):
        pass

    X, Y = TopTest(), SubTest()      # Make two instances
    print(X)                         # Show all instance attrs
    print(Y)                         # Show lowest class name

#output:
    [TopTest: attr1=0, attr2=1]
    [SubTest: attr1=2, attr2=3]
```

Our Classes' Final Form

```
"Assorted class utilities and tools"
class AttrDisplay:
    """
    Provides an inheritable display overload method that shows
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __repr__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())


class Person(AttrDisplay):                              # Mix in a repr at t
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                                 # Assumes last is la
        return self.name.split()[-1]
    def giveRaise(self, percent):                       # Percent must be 0.
        self.pay = int(self.pay * (1 + percent))
```

```
class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)        # Job name i
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)
    tom.giveRaise(.10)
    print(tom.lastName())
    print(tom)


#output:
    [Person: job=None, name=Bob Smith, pay=0]
    [Person: job=dev, name=Sue Jones, pay=100000]
    Smith Jones
    [Person: job=dev, name=Sue Jones, pay=110000]
    Jones
    [Manager: job=mgr, name=Tom Jones, pay=60000]
```

Storing Objects in a Database

- At this point, our work is almost complete. We now have a *two-module system that not* only implements our original design goals for representing people, but also provides a general attribute display tool we can use in other programs in the future.
- By coding functions and classes in module files, we've ensured that they naturally support reuse.
- And by coding our software as classes, we've ensured that it naturally supports extension.

Pickles and Shelves

- Object persistence is implemented by three standard library modules, available in every Python:
- pickle
  - Serializes arbitrary Python objects to and from a string of bytes
- dbm
  - Implements an access-by-key filesystem for storing strings
- shelve
  - Uses the other two modules to store Python objects on a file by key

shelve

Although it's easy to use pickle by itself to store objects in simple flat files and load them from there later, the shelve module provides an extra layer of structure that allows you to store pickled objects by *key. shelve translates an object to its pickled string with* pickle and stores that string under a key in a dbm file; when later loading, shelve fetches the pickled string by key and re-creates the original object in memory with pickle. This is all quite a trick, but to your script a shelve of pickled objects looks just like a *dictionary—you index by key to fetch, assign to keys to store, and use dictionary tools* such as len, in, and dict.keys to get information. Shelves automatically map dictionary operations to objects stored in a file.

```
class AttrDisplay:
    """
    Provides an inheritable display overload method that shows
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __repr__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())


class Person(AttrDisplay):                              # Mix in a repr at
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                                 # Assumes last is l
        return self.name.split()[-1]
    def giveRaise(self, percent):                       # Percent must be 0
        self.pay = int(self.pay * (1 + percent))
```

```
class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)        # Job name
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

import shelve

if __name__ == '__main__':
    bob = Person('Bob Smith')                                  # Re-create
    sue = Person('Sue Jones', job='dev', pay=100000)
    tom = Manager('Tom Jones', 50000)
    db = shelve.open('persondb')                               # Filename
    for obj in (bob, sue, tom):                                # Use objec
        db[obj.name] = obj                                     # Store obj
    db.close()                                                 # Close aft
    db = shelve.open('persondb')
                                                          # keys is the i
    bob = db['Bob Smith']                                 # Fetch bob by
    print(bob)                                            # Runs __repr__
    print(bob.lastName())                                 # Runs lastName
    for key in db:                                        # Iterate, fetc
        print(key, '=>', db[key])
    for key in sorted(db):
        print(key, '=>', db[key])                         # Iterate by sc
    db.close()
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith')                                  # Re-create
    sue = Person('Sue Jones', job='dev', pay=100000)
    tom = Manager('Tom Jones', 50000)
    db = shelve.open('persondb')                               # Filename w
    for obj in (bob, sue, tom):                                # Use object
        db[obj.name] = obj                                     # Store obje
    db.close()                                                 # Close afte
    db = shelve.open('persondb')
                                                          # keys is the ir
    bob = db['Bob Smith']                                 # Fetch bob by k
    print(bob)                                            # Runs __repr__
    print(bob.lastName())                                 # Runs lastName
    for key in db:                                        # Iterate, fetch
        print(key, '=>', db[key])
    for key in sorted(db):
        print(key, '=>', db[key])                         # Iterate by sor
    db.close()



#output:
[Person: job=None, name=Bob Smith, pay=0]
Smith
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

Updating Objects on a Shelve

```
class AttrDisplay:
    """
    Provides an inheritable display overload method that shows
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __repr__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())


class Person(AttrDisplay):                              # Mix in a repr at t
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                                 # Assumes last is la
        return self.name.split()[-1]
    def giveRaise(self, percent):                       # Percent must be 0.
        self.pay = int(self.pay * (1 + percent))
```

Updating Objects on a Shelve

```
class Person(AttrDisplay):                              # Mix in a repr at
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                                 # Assumes last is l
        return self.name.split()[-1]
    def giveRaise(self, percent):                       # Percent must be 0
        self.pay = int(self.pay * (1 + percent))
class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)         # Job name is impli
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith')                          # Re-create obj
    sue = Person('Sue Jones', job='dev', pay=100000)
    tom = Manager('Tom Jones', 50000)
    db = shelve.open('persondb')                       # Filename wher
    for obj in (bob, sue, tom):                        # Use object's
        db[obj.name] = obj                             # Store object
    db.close()                                         # Close after m
    db = shelve.open('persondb')
                                                       # keys is the index
    bob = db['Bob Smith']                              # Fetch bob by key
    print(bob)                                         # Runs __repr__ fro
    print(bob.lastName())                              # Runs lastName fro
    for key in db:                                     # Iterate, fetch, p
        print(key, '=>', db[key])
    for key in sorted(db):
        print(key, '=>', db[key])                      # Iterate by sorted
    sue = db['Sue Jones']                            # Index by key to fet
    sue.giveRaise(.10)                               # Update in memory us
    db['Sue Jones'] = sue                            # Assign to key to up
    for key in db:                                     # Iterate, fetch, p
        print(key, '=>', db[key])
    for key in sorted(db):
        print(key, '=>', db[key])                      # Iterate by sorted
    db.close()                                       # Close after making
```

Updating Objects on a Shelve

```
#output:
[Person: job=None, name=Bob Smith, pay=0]
Smith
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=110000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=110000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

Updating Objects on a Shelve

```
class AttrDisplay:
    """
    Provides an inheritable display overload method that shows
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __repr__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())


class Person(AttrDisplay):                              # Mix in a repr at t
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                                 # Assumes last is la
        return self.name.split()[-1]
    def giveRaise(self, percent):                       # Percent must be 0.
        self.pay = int(self.pay * (1 + percent))
```

Updating Objects on a Shelve

```
class Person(AttrDisplay):                              # Mix in a repr at
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                                 # Assumes last is l
        return self.name.split()[-1]
    def giveRaise(self, percent):                       # Percent must be 0
        self.pay = int(self.pay * (1 + percent))
class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)         # Job name is impli
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith')                          # Re-create objects to
    sue = Person('Sue Jones', job='dev', pay=100000)
    tom = Manager('Tom Jones', 50000)
    db = shelve.open('persondb')                       # Filename where objec
    for obj in (bob, sue, tom):                        # Use object's name at
        db[obj.name] = obj                             # Store object on shel
    db.close()                                         # Close after making c
    db = shelve.open('persondb')

                                                       # keys is the index
    bob = db['Bob Smith']                              # Fetch bob by key
    print(bob)                                         # Runs __repr__ from AttrD
    print(bob.lastName())                              # Runs lastName from Perso
    for key in db:                                     # Iterate, fetch, print
        print(key, '=>', db[key])
    for key in sorted(db):
        print(key, '=>', db[key])                        # Iterate by sorted keys
    sue = db['Sue Jones']                              # Index by key to fetch
    sue.giveRaise(.10)                                 # Update in memory using cla
    db['Sue Jones'] = sue                              # Assign to key to update in
    for key in db:                                       # Iterate, fetch, print
        print(key, '=>', db[key])
    for key in sorted(db):
        print(key, '=>', db[key])                        # Iterate by sorted keys
    rec = db['Sue Jones']                            # Fetch object by key
    print(rec)
    print(rec.lastName())
    print(rec.pay)
    db.close()                                         # Close after making changes
```

Updating Objects on a Shelve

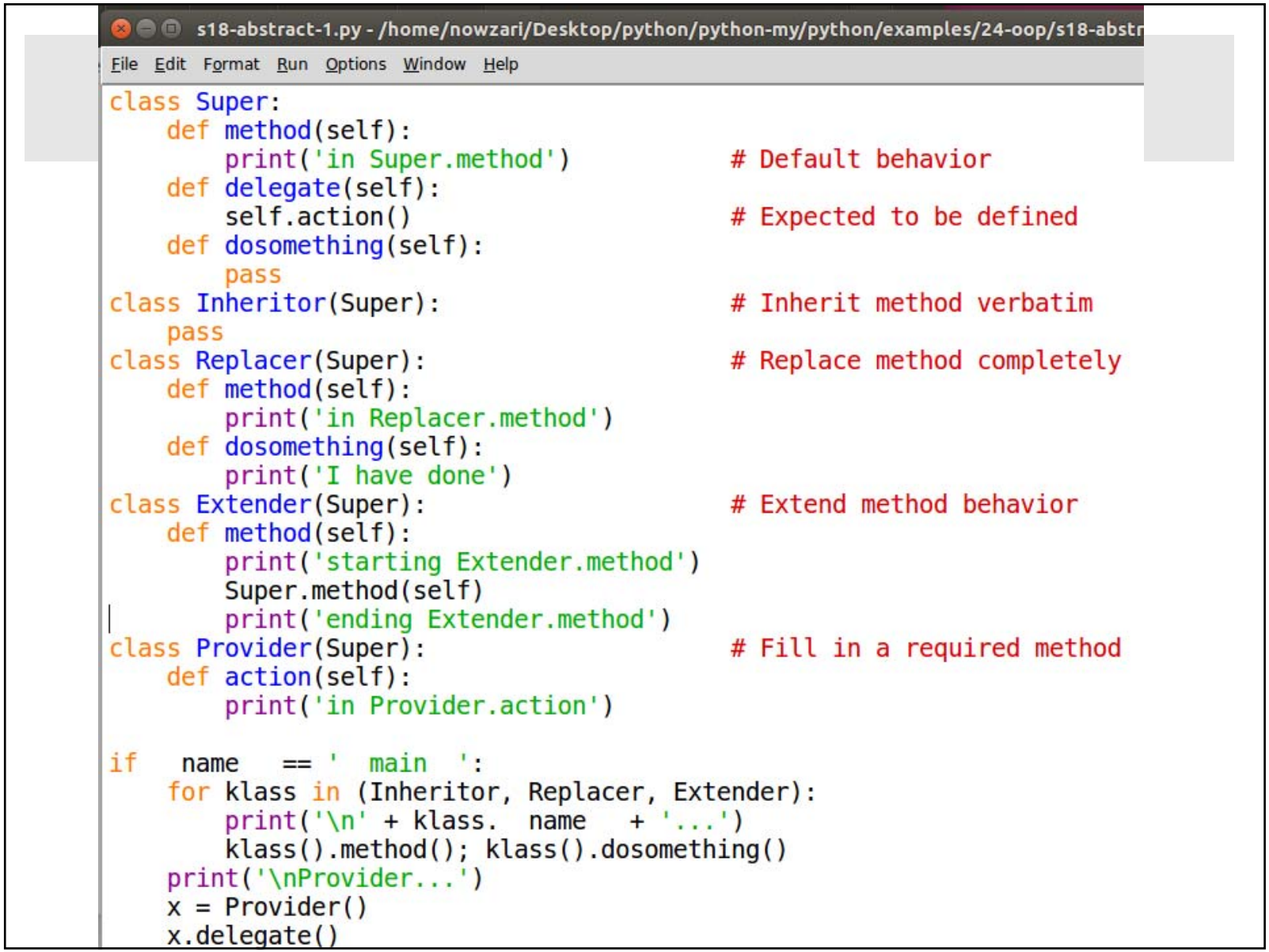
```
#output:
[Person: job=None, name=Bob Smith, pay=0]
Smith
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=110000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=110000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
[Person: job=dev, name=Sue Jones, pay=110000]
Jones
110000
```

Polymorphism and abstract class

- The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.
- The concept of **abstract classes** is also a branch of polymorphism.

Polymorphism and abstract class

- Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods. Subclasses of an abstract class in Python are not required to implement abstract methods of the parent class.


s18-abstract-1.py - /home/nowzari/Desktop/python/python-my/python/examples/24-oop/s18-abstr
File Edit Format Run Options Window Help
class Super:
def method(self):
print('in Super.method') # Default behavior
def delegate(self):
self.action() # Expected to be defined
def dosomething(self):
pass
class Inheritor(Super): # Inherit method verbatim
pass
class Replacer(Super): # Replace method completely
def method(self):
print('in Replacer.method')
def dosomething(self):
print('I have done')
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')
if __name__ == '__main__':
for klass in (Inheritor, Replacer, Extender):
print('\n' + klass.__name__ + '...')
klass().method(); klass().dosomething()
print('\nProvider...')
x = Provider()
x.delegate()

```
if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print('\n' + klass.__name__ + '...')
        klass().method(); klass().dosomething()
    print('\nProvider...')
    x = Provider()
    x.delegate()


#output:
```

```
Inheritor...
in Super.method

Replacer...
in Replacer.method
I have done

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def   init  (self, value):
        self.value = value
        super().  init  ()

    @abstractmethod
    def do something(self):
        pass

class DoAdd42(AbstractClassExample):
    pass

x = DoAdd42(4)
```

s18-abstract-2.py - /home/nowzari/Desktop/python/python-my/python/examples/2

File Edit Format Run Options Window Help

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def   init  (self, value):
        self.value = value
        super().  init  ()

    @abstractmethod
    def do something(self):
        pass

class DoAdd42(AbstractClassExample):

    def do something(self):
        return self.value + 42

class DoMul42(AbstractClassExample):

    def do something(self):
        return self.value * 42

x = DoAdd42(10)
y = DoMul42(10)
print(x.do something())
print(y.do something())

#output

    52
    420
```

isinstance

- Returns a Boolean stating whether the object is an instance or subclass of another object.
- Syntax:

  isinstance *(object, classinfo)*

isinstance

```
i=22
print(isinstance(i,int))
print(isinstance(1,type(55)))
print(isinstance(1, (int, float)))
print(isinstance('Ni', (int, float)))
print(isinstance(42, str))
print(isinstance('x', str))
print(isinstance(b'x', str))

#output
   True
   True
   True
   False
   False
   True
   False
```

isinstance

```
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('Data in first class: ', self.data)

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

x = FirstClass()
z = SecondClass()
print(isinstance(x,FirstClass))
print(isinstance(z,FirstClass))
print(isinstance(x,int))

#output
   True
   True
   False
```

Issubclass

- Return true if class is a subclass of other class. A class is considered a subclass of itself.

```
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('Data in first class: ', self.data)

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

x = FirstClass()
z = SecondClass()
print(issubclass(FirstClass,SecondClass))
print(issubclass(SecondClass,FirstClass))

#output

   False
   True
```

Namespaces and classes

- Now that we've examined class and instance objects, the Python namespace story is complete. For reference, I'll quickly summarize all the rules used to resolve names here. The first things you need to remember are that qualified and unqualified names are treated differently, and that some scopes serve to initialize object namespaces.

```
X = 11                              # Global (module) name/attribute
                                    #(X, or manynames.X)
def f():
    print('in f ', X)               # Access global X (11)
def g():
    X = 22                          # Local (function) variable
                                    # (X, hides module X)
    print('in g ', 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)

# manynames.py, continued
if __name__ == '__main__':
    print('x in main ', X)          # 11: module
                                    # (a.k.a. manynames.X outside file)
    f()                             # 11: global
    g()                             # 22: local
    print('x in main ', X)          # 11: module name unchanged
    obj = C()                       # Make instance
    print('x of object ', obj.X) # 33: class name inherited by instance
    obj.m()                         # Attach attribute name X to instance now
    print('x of object ', obj.X) # 55: instance
    print('x of class  ', C.X)    # 33: class
                                        # (a.k.a. obj.X if no X in instance)
    #print(C.m.X)                   # FAILS: only visible in method
    #print(g.X)                     # FAILS: only visible in function

#output:
    x in main  11
    in f  11
    in g  22
    x in main  11
    x of object  33
    x of object  55
    x of class   33
```

Namespaces and classes

```
import manynames

X = 66
print(X)                        # 66: the global here
print(manynames.X)              # 11: globals become attributes after imports
manynames.f()                   # 11: manynames's X, not the one here!
manynames.g()                   # 22: local in other file's function
print(manynames.C.X)            # 33: attribute of class in other module
I = manynames.C()
print(I.X)                      # 33: still from class here
I.m()
print(I.X)                      # 55: now from instance!


#output:
    66
    11
    in f  11
    in g  22
    33
    33
    55
```

destructor

- A destructor is a special member function that is called when the lifetime of an object ends. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.
- This function is __del__.
- If the del built-in function is called, this function will be run.
- Without the explicit call to del, __del__ is only called at the end of the program.

```
class Counter:
    Count = 0   # This represents the count of objects of this class
    def __init__(self, name):
        self.name = name
        print (name, 'created')
        Counter.Count += 1
    def __del__(self):
        print (self.name, 'deleted')
        Counter.Count -= 1
        if Counter.Count == 0:
            print ('Last Counter object deleted')
        else:
            print (Counter.Count, 'Counter objects remaining')

x = Counter("First")
y = Counter("Secon")
z = Counter("Third")
del x
del y

#output

First created
Secon created
Third created
First deleted
2 Counter objects remaining
Secon deleted
1 Counter objects remaining
```

Multiple Inheritance

- **Python supports a limited form of multiple inheritance.**
- **A class definition with multiple base classes looks as follows:**

```
class DerivedClass(Base1, Base2, Base3 …)
        <statement-1>
        <statement-2>
        …
```

Multiple Inheritance

```
class A:
    def A(self):
        print('I am A')

class B:
    def A(self):
        print('I am a')
    def B(self):
        print('I am B')
class C(A, B):
    def C(self):
        print('I am C')
c=C()
c.A()
c.B()
c.C()

#output
I am A
I am B
I am C
```

Multiple Inheritance

- C multiple inherit A and B, but since A is in the left of B, so C inherit A and invoke A.A() according to the left to right sequence.
- To implement C.B(), class A does not have B() method, so C inherit B for the second priority. So C.B() actually invokes B() in class B.

Iterators

- An iterator is an object that allows a programmer to traverse through all the elements of a collection regardless of its specific implementation.
- Technically speaking, Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.
- An object is called iterable if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables. The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

Iterators

```
L = [1, 2, 3]
I = __iter__(L)      # Obtain an iterator object from an iterable
print(I.__next__())  # Call iterator's next to advance to next item
print(I.__next__())
print(I.__next__())

#output
 1
 2
 3
```

```
L = [1, 2, 3]
I = iter(L)          # Obtain an iterator object from an iterable
print(next(I))       # Call iterator's next to advance to next item
print(next(I))
print(next(I))

#output
 1
 2
 3
```

Building Your Own Iterator

- Building an iterator from scratch is easy in Python. We just have to implement the methods __iter__() and __next__(). The __iter__() method returns the iterator object itself. If required, some initialization can be performed. The __next__() method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise StopIteration.

Building Your Own Iterator

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

Building Your Own Iterator

Now we can create an iterator and iterate through it as follows.

```
>>> a = PowTwo(4)
>>> i = iter(a)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
4
>>> next(i)
8
>>> next(i)
16
>>> next(i)
Traceback (most recent call last):
...
StopIteration
```

We can also use a **for** loop to iterate over our iterator class.

```
>>> for i in PowTwo(5):
...     print(i)
...
1
2
4
8
16
32
```

End