MINDFUL LEARNING

python

A language is not worth knowing unless it teaches you to think differently

- Larry Wall, Randal Schwartz

Professional python

Object Oriented Programming

OOP Concepts and Terminology

- OOP in Python is optional! However, using the OOP paradigm help solving problems in an efficient way
- The following terms should be understood:
  - Class
  - Object or Instances
  - Inheritance
  - Polymorphism

Discussion

Mindful Learning

python

Why OO paradigm?

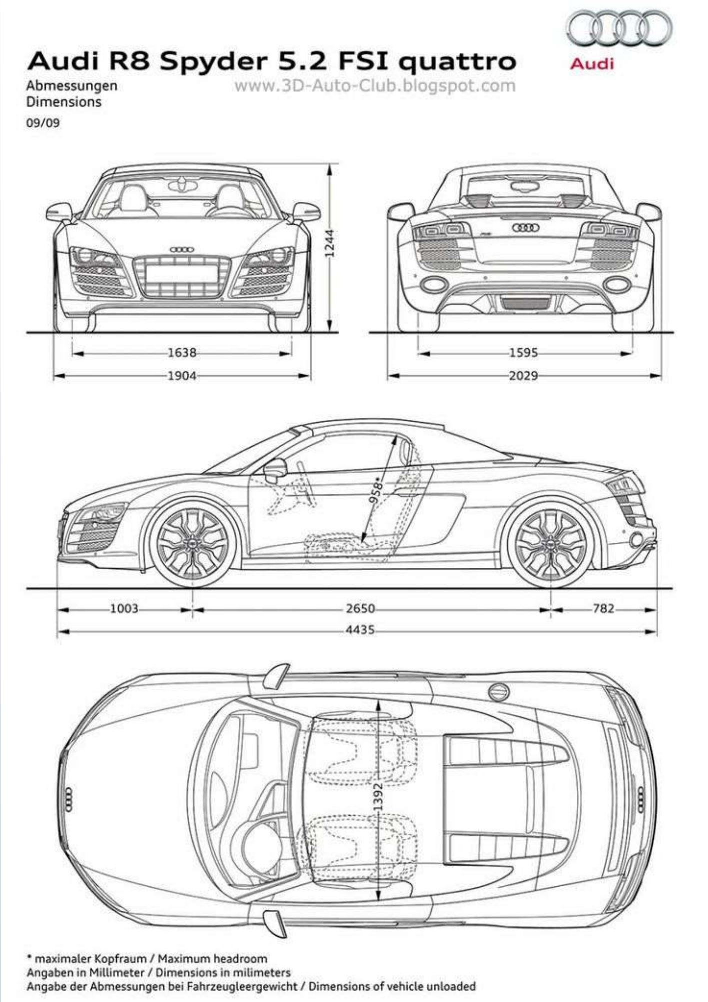
Conclusion

- OOP facilitates
  - Better organization on data
  - Better way of operating on data
  - Reuse and maintanance

Class

- Class defines an extensible template for creating objects
- It clubs variables and member functions under a single entity

Understanding a class



Mindful Learning

python

Object

- An object is a instance of a class and its an entity which has its own set of attributes and functions that were defined by a class

Public Interface

- The set of all methods provided by a class, together with the description of their behavior is called the public interface of the class

Encapsulation

- Encapsulation is the act of providing a public interface and hiding the implementation details
- Encapsulation enables changes in the implementation without affecting users of the class

You can drive a car by operating the steering wheel and pedals, without knowing how the engine works. Similarly, you use an object through its methods. The implementation is hidden.

Inheritance

- Inheritance is a way to form new classes and thereafter objects using classes that have already been defined.

Polymorphism

- Polymorphism means that meaning of operation depends on the object being operated on.

CAR

Let's discuss about driving and maintenance tasks for cars…

Classes in Python

- Everything in Python is an object, hence a class is considered as an object in Python
- Classes are essentially factories for generating multiple instances or objects
- When we run a class statement it creates a class object and assigns it a name
- Class objects provide a default behavior

Constructor

- A constructor creates and initializes the instance variables of an object
- It is automatically called when an object is created
- The constructor is defined using the special method name **__init__**

Class Variable

- It is a variable that is shared by all instances of a class.
- Class variables are defined within a class but outside any of the class's methods.
- Class variables are not used as frequently as instance variables are.

Instance Variable

- A variable that is defined inside a method and belongs only to the current instance of a class

Methods

- They provide the public interface for every object that is created
- These are defined inside a class
- A method can access the instance variables of the object on which it acts
- A **mutator/setter** method changes the object attributes on which it operates
- An **accessor/getter** method does not change the attributes but queries the object for some information

The **self** Argument

- You declare other class methods like normal functions with the exception that the first argument to each method is **self**.
- Python adds the **self** argument to the list for you; you do not need to include it when you call the methods.

Instance Objects

- An instance object or simply object is created by calling the class object
- Instance variables store data required for executing methods
- Each object/instance of a class has its own set of instance variables

Accessing Methods

- You access the object's attributes using the dot operator with object

Example

Mindful Learning python

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary


emp1 = Employee("Kumar", 2000)
emp2 = Employee("Abhinav", 5000)

emp1.displayEmployee()
emp2.displayEmployee()

print "Total Employee %d" % Employee.empCount
```

Class variable

Constructor

Instance variables

Methods

Object creation

Accessing methods

Special Functions to Access Attributes

- Instead of using the normal statements to access attributes, you can use the following functions –
  - **getattr(obj, name[, default])** : to access the attribute of object.
  - **hasattr(obj,name)** : to check if an attribute exists or not.
  - **setattr(obj,name,value)** : to set an attribute. If attribute does not exist, then it would be created.
  - **delattr(obj, name)** : to delete an attribute.

Example

```
hasattr(emp1, 'age')     # Returns true if 'age' attribute exists
getattr(emp1, 'age')     # Returns value of 'age' attribute
setattr(emp1, 'age', 8)  # Set attribute 'age' at 8
delattr(empl, 'age')     # Delete attribute 'age'
```

Special Attributes

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

  __dict__ : Dictionary containing the class's namespace.

  __doc__ : Class documentation string or none, if undefined.

  __name__ : Class name.

  __module__ : Module name in which the class is defined. This attribute is "__main__" in interactive mode.

  __bases__ : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Example

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

Testing a Class

- A unit test verifies that a class works correctly in isolation outside a complete program
- To test a class, use an environment for interactive testing or write a tester program to execute the test instructions
- Determining the expected result in advance is an important part of testing
- Test on the go

Patterns for Object Data

- When an object is designed, the needs of the programmers who use your class should be considered. Thus, you provide the public interface.
- Though the methods that constitute the public interface is not obvious in every type of class, there are few recurring patterns as below:
  - Keeping a total
  - Collecting values
  - Counting events
  - Managing properties of an object
  - Modeling object with distinct states
  - Describing the position of an object to model a moving object

Destroying Objects

- The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.
- Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero.
- The object's reference count decreases when it's deleted with **del**, its reference is reassigned, or its reference goes out of scope.
- A class can implement the special method **__del__()**, called a destructor, that is invoked when the instance is about to be destroyed.

Inheritance

- **Inheritance** is a mechanism of code customization and reuse.
- It is a way to form new classes (instances of which are called **objects**) using classes that have already been defined.
- It creates an 'is-a' relationship with the superclass

Superclass and Subclass

- In object-oriented design, inheritance is a relationship between a more general class called the superclass and a more specialized class called the subclass.
- The subclass inherits data and behavior from the superclass.
- You can always use a subclass object in place of a superclass object
- A subclass reference can be used when a superclass reference is expected

Constructor in subclass

- The superclass is responsible for defining its own instance variables
- The subclass constructor must explicitly call the superclass constructor
- Use the super function to call the superclass constructor

super()

- Returns a proxy object that delegates method calls to a parent

  super([*type*[, *object-or-type*]])

```
class Root(object):
    def draw(self):
        # the delegation chain stops here
        assert not hasattr(super(Root, self), 'draw')

class Shape(Root):
    def __init__(self, shapename, **kwds):
        self.shapename = shapename
        super(Shape, self).__init__(**kwds)
    def draw(self):
        print 'Drawing.  Setting shape to:', self.shapename
        super(Shape, self).draw()

class ColoredShape(Shape):
    def __init__(self, color, **kwds):
        self.color = color
        super(ColoredShape, self).__init__(**kwds)
    def draw(self):
        print 'Drawing.  Setting color to:', self.color
        super(ColoredShape, self).draw()

ColoredShape(color='blue', shapename='square').draw()
print '-' * 20
```

Keep in Mind

- When you are doing this kind of specialization, there are three ways that the parent and child classes can interact:
  - Actions on the child imply an action on the parent.
  - Actions on the child override the action on the parent.
  - Actions on the child alter the action on the parent.

Method Override

- The subclass inherits all methods from the superclass.
- A subclass can override a superclass method by providing a new implementation
- Overriding a method can extend or replace the functionality of the superclass method
- Use super to call a superclass method

Polymorphism

- Polymorphism refers to the ability of the object to adapt the code to the type of data it is processing
- It has two major applications in OOP:
  - It provides different implementations of the methods depending upon the input types
  - Code written for a given type of data may be used on data with a derived type, i.e. methods understand the class hierarchy of a type

Operator Overloading

- Operator overloading refers to intercepting built-in operations in class's methods – Python automatically invokes these methods when instances of the class appear in built-in operations
- Key ideas:
  - Operator overloading lets classes intercept normal python operations
  - Classes can overload all Python expression operators
  - Classes can also overload built-in operations
  - Overloading makes class instances act more like built-in types
  - Overloading is implemented by providing operator method names beginning and ending with double underscore

Check: https://docs.python.org/2/library/functions.html
Refer: Learning Python 5 Ed. Mark Lutz Chapter 30

Example: Operator Overloading

Let's overload the '+' operator or the add()

```
class Point:
        def __init__(self, x = 0, y = 0):
                self.x = x
                self.y = y

        def __str__(self):
                return "({0},{1})".format(self.x,self.y)

        def __add__(self,other):
                x = self.x + other.x
                y = self.y + other.y
                return Point(x,y)
```

```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> print(p1 + p2)
 (1,5)
```

Similarly many built-in functions and operators can be overloaded

Example: Override Built-Ins

- Say, you want to change the output produced by printing or viewing instances to something more sensible.

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
>>> import pair
>>> p = Pair(3,4)
>>> p
<__main__.Pair instance at 0x00AE9800>
>>> print(p)
<__main__.Pair instance at 0x00AE9800>
```

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)
    def __str__(self):
        return '({0.x!s}, {0.y!s})'.format(self)
```

```
# Restart the shell and repeat
>>> import pair
>>> p = Pair(3,4)
>>> p
Pair(3, 4)
>>> print(p)
(3, 4)
```

Application

- Say, you want an object to support customized formatting through the format() function and string method.

```
_formats = {
    'ymd' : '{d.year}-{d.month}-{d.day}',
    'mdy' : '{d.month}/{d.day}/{d.year}',
    'dmy' : '{d.day}/{d.month}/{d.year}'
    }

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

    def __format__(self, code):
        if code == '':
            code = 'ymd'
        fmt = _formats[code]
        return fmt.format(d=self)
```

```
>>> d = Date(2017, 1, 21)
>>> format(d)
'2017-1-21'
>>> format(d, 'dmy')
'21/1/2017'
```

__new__

- The magic method __new__ will be called when instance is being created.
- Using this method you can customize the instance creation.
- This is only the method which will be called before __init__ to initialize instance when you are creating instance

```
class AbstractClass(object):

    def __new__(cls, a, b):
        instance = super(AbstractClass, cls).__new__(cls)
        instance.__init__(a, b)
        return 3

    def __init__(self, a, b):
        print "Initializing Instance", a, b
```

```
>>> a = AbstractClass(2, 3)
Initializing Instance 2 3
>>> a
3
```

Protected Attributes

- By prefixing the name of a member in a class with a single underscore
- It actually changes nothing. It's only a convention to follow

```
class Cup:
    def __init__(self):
        self.color = None
        self._content = None # protected
variable

    def fill(self, beverage):
        self._content = beverage

    def empty(self):
        self._content = None

cup = Cup()
cup._content = "tea"
```

Protected attribute

Private Attributes

- In Python, we use double underscore (Or __) before the names of methods and attributes to make them invisible outside the class
- These methods and attributes will not be directly accessible outside.

PS: In Python, nothing is strictly private

Example: Private Attributes

```
class MyClass:

    # Hidden member of MyClass
    __hiddenVariable = 0

    # A member method that changes
    # __hiddenVariable
    def add(self, increment):
        self.__hiddenVariable += increment
        print (self.__hiddenVariable)

    def __test(self):
        print(seld.__hiddenVariable)

# Driver code
myObject = MyClass()
myObject.add(2)
myObject.add(5)

# These lines cause error
print (myObject.__hiddenVariable)
myObject.__test()
```

Hidden data

Hidden method

Accessing Private Attributes

- Python supports a technique called name mangling.
- This feature turns every member name prefixed with at least two underscores and suffixed with at most one underscore into _<className><memberName>

```
class MyClass:

    __hiddenVariable = 0

    def add(self, increment):
        self.__hiddenVariable += increment
        print (self.__hiddenVariable)

    def __test(self):
        print(self.__hiddenVariable)

# Driver code
myObject = MyClass()
myObject.add(2)

print (myObject._MyClass__hiddenVariable)
myObject._MyClass__test()
```

Accessing hidden attributes

Composition

- Composition is another way of extending a class by explicitly creating an instance of a class inside a sub-class
- It creates 'has-a' relationship
- Inheritance Vs Composition
  - In Inheritance, a class is inherited (extended) by a new sub-class that will add custom attributes and behavior to the inherited ones
  - In Composition, a class is utilized by creating an instance of it, and including that instance inside another larger object

Example: Composition


```
class math:
    def __init__(self, x, y):
        self.x = x;
        self.y = y;
    def add(self):
        return self.x + self.y
    def subtract(self):
        return seld.x - self.y


class math2:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def multiply(self):
        return self.x * self.y
    def divide(self):
        return self.x/self.y
```

```
class math3:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.m1 = math(x,y)
        self.m2 = math2(x,y)
    def power(self):
        return self.x ** self.y
    def add(self):
        return self.m1.add()
    def subtract(self):
        return self.m1.subtract()
    def multiply(self):
        return self.m2.multiply()
```

Composition

Create a unit test for this code.
How would you achieve the same through inheritance?

Delegation

- Delegate = entrusting the responsibility to another person
- Delegation is a special form of composition, with a single embedded object managed by a wrapper (sometimes called a proxy) class that retains most or all of the embedded object’s interface.

```
# Simple demonstration of the Proxy pattern.

class Implementation:
    def f(self):
        print("Implementation.f()")
    def g(self):
        print("Implementation.g()")
    def h(self):
        print("Implementation.h()")

class Proxy:
    def __init__(self):
        self.__implementation = Implementation()
    # Pass method calls to the implementation:
    def f(self): self.__implementation.f()
    def g(self): self.__implementation.g()
    def h(self): self.__implementation.h()

p = Proxy()
p.f(); p.g(); p.h()
```

Delegation

```
# Simple demonstration of the Proxy pattern.

class Implementation2:
    def f(self):
        print("Implementation.f()")
    def g(self):
        print("Implementation.g()")
    def h(self):
        print("Implementation.h()")

class Proxy2:
    def __init__(self):
        self.__implementation = Implementation2()
    def __getattr__(self, name):
        return getattr(self.__implementation, name)

p = Proxy2()
p.f(); p.g(); p.h();
```

Composition

Delegated

Mixins – Multiple Inheritance

- Mixin class is a class that has been inherited from multiple superclasses
- In a class statement, more than one superclass can be listed in parentheses in the header line.
- When you do this, you leverage multiple inheritance—the class and its instances inherit names from all the listed superclasses.

Example

- Say we have two classes: clock and calendar. Let’s create a clock_calendar class that inherits from clock and calender classes as shown:

Example

```
class Clock(object):

    def __init__(self, hours, minutes, seconds): pass
    def set_Clock(self, hours, minutes, seconds): pass
    def __str__(self): pass
    def tick(self): pass
```

```
class Calendar(object):

    months = (31,28,31,30,31,30,31,31,30,31,30,31)
    date_style = "British"
    @staticmethod
    def leapyear(year):
    def __init__(self, d, m, y):
    def set_Calendar(self, d, m, y):
    def __str__(self):
    def advance(self):
```

```
from clock import Clock
from calendar import Calendar

class CalendarClock(Clock, Calendar):

    def __init__(self,day, month, year, hour, minute, second):
    def tick(self):
    def __str__(self):
```

Multiple Inheritance

Factories

- Sometimes, class-based designs require objects to be created in response to conditions that can't be predicted when a program is written.
- The factory design pattern allows such a deferred approach
- Factory design pattern makes it easy to create specialized objects by instantiating a subclass chosen at runtime.

Example

Mindful Learning python

```
class Cup:
    color = ""

    # This is the factory method
    @staticmethod   #Check what happens if you comment this
    def getCup(cupColor):
        if (cupColor == "red"):
            return RedCup()
        elif (cupColor == "blue"):
            return BlueCup()
        else:
            return None

class RedCup(Cup):
    color = "red"

class BlueCup(Cup):
    color = "blue"

# A little testing
redCup = Cup.getCup("red")
print "%s(%s)" % (redCup.color, redCup.__class__.__name__)
blueCup = Cup.getCup("blue")
print "%s(%s)" % (blueCup.color, blueCup.__class__.__name__)
```

```
RESTART: E:/Python27/mindful_examples/Classes - OOP/factory_design_pattern_02.py
red(RedCup)
blue(BlueCup)
```

Study: `factory_design_pattern_01/03.py`

Special Types of Methods

- Sometimes we need to process data associated with classes instead of instances
  - Such as keeping track of instances created or instances currently in the memory
  - Simple function written outside the class can suffuce this requirement
  - However, the code will not be well associated with class and cannot be inherited and the name of the method is not localized
- Three types of methods are available:
  - Instance methods – discussed so far, default
  - Static methods
  - Class methods

Mindful Learning

python

Static Method

- **With static methods**, neither self (the object instance) nor cls (the class) is implicitly passed as the first argument.
- Nested inside a class
- Work on class attributes and not on the instance attributes
- They behave like plain functions except that you can call them from an instance or the class
- A built-in function staticmethod() is used to create them

Example

```
class example:
    numInstances = 0 # Use static method for class data
    def __init__(self):
        example.numInstances += 1
    def printNumInstances():
        print("Number of instances: %s" % example.numInstances)

    printNumInstances = staticmethod(printNumInstances)
```

```
==== RESTART: E:/Python27/mindful_examples/Classes - OOP/static_method.py ====
>>> import static_method
>>> a = example()
>>> b = example()
>>> c = example()
>>> example.printNumInstances()
Number of instances: 3
```

Benefits of Static Methods

- It localizes the function name in the class scope (so it won’t clash with other names in the module)
- It moves the function code closer to where it is used (inside the class statement)
- It allows the sub-classes to customize the static method with inheritance
- Classes can inherit the static method without redefining it

Class Method

- Method definitions that have first argument as class name
- Can be called through both class and instance
- These are created with classmethod() inbuilt function
- The lowest class is passed in whenever a class method is run, even for subclasses that have no class methods of their own (Example 2)
- Class methods may be better suited to processing data that may differ for each class in a hierarchy (Example 3)

Example

```
class example:
    numInstances = 0 # Use class method instead of static
    def __init__(self):
        example.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances: %s" % cls.numInstances)
    printNumInstances = classmethod(printNumInstances))
```

```
==== RESTART: E:/Python27/mindful_examples/Classes - OOP/class_method.py ====
>>> import class_method
>>> a = example()
>>> b = example()
>>> example.printNumInstances()
Number of instances: 2
```

Example 2

```
class example:
    numInstances = 0 # Use class method instead of static
    def __init__(self):
        example.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances: %s" % cls.numInstances)
    printNumInstances = classmethod(printNumInstances)

class sub(example):
    def printNumInstances(cls): # Override a class method
        print("Extra stuff...", cls) # But call back to original
        example.printNumInstances()
    printNumInstances = classmethod(printNumInstances)

class other(example): pass # Inherit class method verbatim
```

Change: class other(sub): pass
Re-run
\>>> z = other()
\>>> z.printNumInstances()
What do you infer?

Example 2

```
=== RESTART: E:/Python27/mindful_examples/Classes - OOP/class_method_03.py ===
>>> import class_method_03
>>> x = sub()
>>> y = example()
>>> x.printNumInstances()
('Extra stuff...', <class __main__.sub at 0x01225298>)
Number of instances: 2
>>> sub.printNumInstances()
('Extra stuff...', <class __main__.sub at 0x01225298>)
Number of instances: 2
>>> z = other()
>>> z.printNumInstances()
Number of instances: 3
```

Example 3

Mindful Learning python

```
class Spam:
    numInstances = 0
    def count(cls):            # Per-class instance counters
        cls.numInstances += 1 # cls is lowest class above instance
    def __init__(self):
        self.count() # Passes self.__class__ to count
    count = classmethod(count)

class Sub(Spam):
    numInstances = 0
    def __init__(self): # Redefines __init__
        Spam.__init__(self)

class Other(Spam): # Inherits __init__
    numInstances = 0
```

```
=== RESTART: E:/Python27/mindful_examples/Classes - OOP/class_method_01.py ===
>>> import class_method_01
>>> x = Spam()
>>> y = Sub()
>>> z = Sub()
>>> a = Other()
>>> b = Other()
>>> c = Other()
>>> Spam.numInstances, Sub.numInstances, Other.numInstances
(1, 2, 3)
```

Example using Decorators

```
class Circle:
  all_circles = [] # class variable

  @staticmethod
  def total_area():
      for c in Circle.all_circles: # hardcode class name
          # do somethig

  @classmethod
  def total_area(cls):
      for c in cls.all_circles: # no hardcode class name
          # do something
```

No self or cls

MINDFUL LEARNING

python

Abstract Base Class

- Abstract super/base class—a class that expects parts of its behavior to be provided by its subclasses.
  - If an expected method is not defined in a subclass, Python raises an undefined name exception when the inheritance search fails.
- ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by isinstance() and issubclass()
- By defining an abstract base class, you can define a common API for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations
- It can also aid you when working on a large team or with a large code-base where keeping all classes in your head at the same time is difficult or not possible.

Example

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!' # If this version is called
```

```
>>> import abstract_super_class
>>> x = Super()
>>> x.delegate()

Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    x.delegate()
  File "E:/Python27/mindful_examples/Classes - OOP/abstract_super_class.py", line 3, in
delegate
    self.action()
  File "E:/Python27/mindful_examples/Classes - OOP/abstract_super_class.py", line 5, in
action
    assert False, 'action must be defined!' # If this version is called
AssertionError: action must be defined!
>>> class Sub(Super):
       def action(self):
           print('Pythonic action')


>>> y = Sub()
>>> y.delegate()
Pythonic action
>>>
```

Decorators

- A decorator in Python is a callable Python object that is used to modify a function, method or class definition.
- The original object, the one which is going to be modified, is passed to a decorator as an argument.
- The decorator returns a modified object, e.g. a modified function, which is bound to the name used in the definition.

Example

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

def foo(x):
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo(42)
```

```
We call foo before decoration:
Hi, foo has been called with Hi
We now decorate foo with f:
We call foo after decoration:
Before calling foo
Hi, foo has been called with 42
After calling foo
```

Example Continued..

- The decoration occurrs in the line before the function header. The "@" is followed by the decorator function name.
- We will rewrite now our initial example. Instead of writing the statement

    foo = our_decorator(foo)   we can write

    @our_decorator

Example Continued..

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```

MINDFUL LEARNING python

Class as a Decorator

- The example shows how a class can be used as a decorator
- __call__ function holds an important role here

```
def decorator1(f):
    def helper():
        print("Decorating", f.__name__)
        f()
    return helper

@decorator1
def foo():
    print("inside foo()")

foo()
```

```
class decorator2(object):

    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Decorating", self.f.__name__)
        self.f()

@decorator2
def foo():
    print("inside foo()")

foo()
```