Classes

All Versions

Python offers itself not only as a popular scripting language, but also supports the object-oriented programming paradigm. Classes describe data and provide methods to manipulate that data, all encompassed under a single object. Furthermore, classes allow for abstraction by separating concrete implementation details from abstract representations of data.

Code utilizing classes is generally easier to read, understand, and maintain.

Examples

Improvements requested:

Introduction to classes

A class, functions as a template that defines the basic characteristics of a particular object. Here's an example:

```
class Person(object):
    """A simple class."""                           # docstring
    species = "Homo Sapiens"                        # class attribute

    def __init__(self, name):                       # special method
        """This is the initializer. It's a special
        function (see below).
        """
        self.name = name                            # instance attribute

    def __str__(self):                              # special method
        """This function is run when Python tries
        to cast the object to a string. Return
        this string when using print(), etc.
        """
        return self.name

    def rename(self, renamed):                      # regular method
        """Reassign and print the name attribute."""
        self.name = renamed
        print("Now my name is " + self.name)
```

There are a few things to note when looking at the above example.

1. The class is made up of *attributes* (data) and *methods* (functions).
2. Attributes and methods are simply defined as normal variables and functions.
3. As noted in the corresponding docstring, the `__init__()` function is called the *initializer* . It's equivalent to the constructor in other object oriented languages, and is the function that is first run when you create a new object, or new instance of the class. *Every Python class must have an initializer.*
4. Attributes that apply to the whole class are defined first, and are called *class attributes* .
5. Attributes that apply to a specific instance of a class (an object) are called *instance attributes* . They are generally defined inside `__init__()` ; this is not necessary, but it is recommended (since attributes defined outside of `__init__()` run the risk of being accessed before they are defined).
6. Every function, or method, included in the class definition passes the object in question as its first parameter. The word `self` is used for this parameter (usage of `self` is actually by convention, as the word `self` has no inherent meaning in Python, but this is one of Python's most respected conventions, and you should always follow it).
7. Those used to object-oriented programming in other languages may be surprised by a few things. One is that Python has no real concept of `private` elements, so everything, by default, imitates the behavior of the C++/Java `public` keyword. For more information, see the "Private Class Members" example on this page.
8. Some of the class's methods have the following form: `__functionname__(self, other_stuff)` . All such functions are called "magic methods" and are an important part of classes in Python. For instance, operator overloading in Python is implemented with magic methods. For more information, see the relevant documentation .

Now let's make a few instances of our `Person` class!

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

We currently have three `Person` objects, `kelly` , `joseph` , and `john_doe` .

We can access the attributes of the class from each instance using the dot operator . Note again the difference between class and instance attributes:

```
>>> # Attributes
>>> kelly.species
```

```
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

We can execute the methods of the class using the same dot operator `.` :

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>>  print(john_doe)
'John Doe'
>>>  john_doe.rename("John")
'Now my name is John'
```

Bound, unbound, and static methods

The idea of bound and unbound methods was removed in Python 3 . In Python 3 when you declare a method within a class, you are using a `def` keyword, thus creating a function object. This is a regular function, and the surrounding class works as its namespace. In the following example we declare method `f` within class `A` , and it becomes a function `A.f` :

Python 3.x ≥3.0

```
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...>  (in Python 3.x)
```

In Python 2 the behavior was different: function objects within the class were implicitly replaced with objects of type `instancemethod` , which were called *unbound methods* because they were not bound to any particular class instance. It was possible to access the underlying function using `.__func__` property.

Python 2.x ≥2.3

```
A.f
# <unbound method A.f>   (in Python 2.x)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

The latter behaviors are confirmed by inspection - methods are recognized as functions in Python 3, while the distinction is upheld in Python 2.

Python 3.x ≥3.0

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Python 2.x ≥2.3

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

In both versions of Python function/method `A.f` can be called directly, provided that you pass an instance of class `A` as the first argument.

```
A.f(1, 7)
# Python 2: TypeError: unbound method f() must be called with
#                      A instance as first argument (got int instance instead)
# Python 3: 14
a = A()
A.f(a, 20)
```

```
A.f(a, 20)
# Python 2 & 3: 40
```

Now suppose `a` is an instance of class `A`, what is `a.f` then? Well, intuitively this should be the same method f of class `A`, only it should somehow "know" that it was applied to the object `a` – in Python this is called method *bound* to `a`.

The nitty-gritty details are as follows: writing `a.f` invokes the magic `__getattribute__` method of `a`, which first checks whether `a` has an attribute named f (it doesn't), then checks the class `A` whether it contains a method with such a name (it does), and creates a new object `m` of type `method` which has the reference to the original `A.f` in `m.__func__`, and a reference to the object `a` in `m.__self__`. When this object is called as a function, it simply does the following: `m(...) => m.__func__(m.__self__, ...)`. Thus this object is called a **bound method** because when invoked it knows to supply the object it was bound to as the first argument. (These things work same way in Python 2 and 3).

```
a = A()
a.f
# <bound method A.f of <__main__.A object at ...>>
a.f(2)
# 4

# Note: the bound method object a.f is recreated *every time* you call it:
a.f is a.f  # False
# As a performance optimization you can store the bound method in the object's
# __dict__, in which case the method object will remain fixed:
a.f = a.f
a.f is a.f  # True
```

Finally, Python has **class methods** and **static methods** – special kinds of methods. Class methods work same way as regular methods, except that when invoked on an object they bind to the *class* of the object instead of to the object. Thus `m.__self__ = type(a)`. When you call such bound method, it passes the class of `a` as the first argument. Static methods are even simpler: they don't bind anything at all, and simply return the underlying function without any transformations.

```
class D(object):
    multiplier = 2

    @classmethod
    def f(cls, x):
        return cls.multiplier * x

    @staticmethod
    def g(name):
        print("Hello, %s" % name)

D.f
# <bound method type.f of <class '__main__.D'>>
D.f(12)
# 24
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

Note that class methods are bound to the class even when accessed on the instance:

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
# (2, 1337)
d.f
# <bound method D.f of <class '__main__.D'>>
d.f(10)
# 20
```

It is worth noting that at the lowest level, functions, methods, staticmethods, etc. are actually descriptors that invoke `__get__`, `__set__` and optionally `__del__` special methods. For more details on classmethods and staticmethods:

- What is the difference between @staticmethod and @classmethod in Python?
- Python @classmethod and @staticmethod for beginner?

Basic inheritance

Inheritance in Python is based on similar ideas used in other object oriented languages like Java, C++ etc. A new class can be derived from an existing class as follows.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

The BaseClass is the already existing (*parent*) class, and the DerivedClass is the new (*child*) class that inherits (or *subclasses*) attributes from BaseClass . **Note** : As of Python 2.2, all classes implicitly inherit from the object class , which is the base class for all built-in types.

We define a parent Rectangle class in the example below, which implicitly inherits from object :

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

The Rectangle class can be used as a base class for defining a Square class, as a square is a special case of rectangle.

```
class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
        super(Square, self).__init__(s, s)
        self.s = s
```

The Square class will automatically inherit all attributes of the Rectangle class as well as the object class. super() is used to call the __init__() method of Rectangle class, essentially calling any overridden method of the base class. **Note** : in Python 3, super() does not require arguments.

Derived class objects can access and modify the attributes of its base classes:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

Built-in functions that work with inheritance

issubclass(DerivedClass, BaseClass) : returns True if DerivedClass is a subclass of the BaseClass

isinstance(s, Class) : returns True if s is an instance of Class or any of the derived classes of Class

```
# subclass check
issubclass(Square, Rectangle)
# Output: True

# instantiate
r = Rectangle(3, 4)
s = Square(2)

isinstance(r, Rectangle)
# Output: True
isinstance(r, Square)
# Output: False
# A rectangle is not a square

isinstance(s, Rectangle)
# Output: True
# A square is a rectangle
isinstance(s, Square)
# Output: True
```

Monkey Patching

In this case, "monkey patching" means adding a new variable or method to a class after it's been defined. For instance, say we defined class A as

```
class A(object):
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

But now we want to add another function later in the code. Suppose this function is as follows

But now we want to add another function later in the code. Suppose this function is as follows:

```
def get_num(self):
    return self.num
```

But how do we add this as a method in `A`? That's simple we just essentially place that function into `A` with an assignment statement.

```
A.get_num = get_num
```

Why does this work? Because functions are objects just like any other object, and methods are functions that belong to the class.

The function `get_num` shall be available to all existing (already created) as well to the new instances of `A`

These additions are available on all instances of that class (or its subclasses) automatically. For example:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42

bar.get_num() # 6
```

Note that, unlike some other languages, this technique does not work for certain built-in types, and it is not considered good style.

New-style vs. old-style classes

Python 2.x ≥2.2.0

New-style classes were introduced in Python 2.2 to unify *classes* and *types*. They inherit from the top-level `object` type. *A new-style class is a user-defined type*, and is very similar to built-in types.

```
# new-style class
class New(object):
    pass

# new-style instance
new = New()

new.__class__
# <class '__main__.New'>
type(new)
# <class '__main__.New'>
issubclass(New, object)
# True
```

Old-style classes do **not** inherit from `object`. Old-style instances are always implemented with a built-in `instance` type.

```
# old-style class
class Old:
    pass

# old-style instance
old = Old()

old.__class__
# <class __main__.Old at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Python 3.x ≥3.0.0

In Python 3, old-style classes were removed.

New-style classes in Python 3 implicitly inherit from `object`, so there is no need to specify `MyClass(object)` anymore.

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class
```

```
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

Class methods: alternate initializers

Class methods present alternate ways to build instances of classes. To illustrate, let's look at an example.

Let's suppose we have a relatively simple `Person` class:

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

It might be handy to have a way to build instances of this class specifying a full name instead of first and last name separately. One way to do this would be to have `last_name` be an optional parameter, and assuming that if it isn't given, we passed the full name in:

```
class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

However, there are two main problems with this bit of code:

1. The parameters `first_name` and `last_name` are now misleading, since you can enter a full name for `first_name` . Also, if there are more cases and/or more parameters that have this kind of flexibility, the if/elif/else branching can get annoying fast.
2. Not quite as important, but still worth pointing out: what if `last_name` is `None` , but `first_name` doesn't split into two or more things via spaces? We have yet another layer of input validation and/or exception handling...

Enter class methods. Rather than having a single initializer, we will create a separate initializer, called `from_full_name` , and decorate it with the (built-in) `classmethod` decorator.

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    @classmethod
    def from_full_name(cls, name, age):
        if " " not in name:
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        return cls(first_name, last_name, age)

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Notice `cls` instead of `self` as the first argument to `from_full_name` . Class methods are applied to the overall class, *not* an instance of a given class (which is what `self` usually denotes). So, if `cls` is our `Person` class, then the returned value from the `from_full_name` class method is `Person(first_name, last_name, age)` , which uses `Person` 's `__init__` to create an instance of the `Person` class. In particular, if we were to make a subclass `Employee` of `Person` , then `from_full_name` would work in the `Employee` class as well.

To show that this works as expected, let's create instances of `Person` in more than one way without the branching in `__init__` :

```
In [2]: bob = Person("Bob", "Bobberson", 42)
```

```
In [3]: alice = Person.from_full_name("Alice Henderson", 31)

In [4]: bob.greet()
Hello, my name is Bob Bobberson.

In [5]: alice.greet()
Hello, my name is Alice Henderson.
```

Other references:

- Python @classmethod and @staticmethod for beginner?
- https://docs.python.org/2/library/functions.html#classmethod
- https://docs.python.org/3.5/library/functions.html#classmethod

---

Multiple Inheritance

Python uses the C3 linearization algorithm to determine the order in which to resolve class attributes, including methods. This is known as the Method Resolution Order (MRO).

Here's a simple example:

```
class Foo(object):
    foo = 'attr foo of Foo'


class Bar(object):
    foo = 'attr foo of Bar' # we won't see this.
    bar = 'attr bar of Bar'

class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'
```

Now if we instantiate FooBar, if we look up the foo attribute, we see that Foo's attribute is found first

```
fb = FooBar()
```

and

```
>>> fb.foo
'attr foo of Foo'
```

Here's the MRO of FooBar:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

It can be simply stated that Python's MRO algorithm is

1. Depth first (e.g. `FooBar` then `Foo`) unless
2. a shared parent (`object`) is blocked by a child (`Bar`) and
3. no circular relationships allowed.

That is, for example, Bar cannot inherit from FooBar while FooBar inherits from Bar.

For a comprehensive example in Python, see the wikipedia entry .

Another powerful feature in inheritance is `super` . super can fetch parent classes features.

```
class Foo(object):
    def foo_method(self):
        print "foo Method"

class Bar(object):
    def bar_method(self):
        print "bar Method"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

Class and instance variables

Instance variables are unique for each instance, while class variables are shared by all instances.

```
class C:
```

```
class C:
    x = 2  # class variable

    def __init__(self, y):
        self.y = y  # instance variable

C.x
# 2
C.y
# AttributeError: type object 'C' has no attribute 'y'

c1 = C(3)
c1.x
# 2
c1.y
# 3

c2 = C(4)
c2.x
# 2
c2.y
# 4
```

Class variables can be accessed on instances of this class, but assigning to the instance attribute will create an instance variable which shadows the class variable

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Note that *mutating* class variables from instances can lead to some unexpected consequences.

```
class D:
    x = []
    def __init__(self, item):
        self.x.append(item)  # note that this is not an assigment!

d1 = D(1)
d2 = D(2)

d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

Class composition

Class composition allows explicit relations between objects. In this example, people live in cities that belong to countries. Composition allows people to access the number of all people living in their country:

```
class Country(object):
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)


class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople


    def addPerson(self, person):
        self.people.append(person)

    def join_country(self,country):
        self.country = country
        country.addCity(self)

        for i in range(self.numPeople):
                person(i).join_city(self)


class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
```

```
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x
```

Listing All Class Members

The `dir()` function can be used to get a list of the members of a class:

```
dir(Class)
```

For example:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__
```

It is common to look only for "non-magic" members. This can be done using a simple comprehension that lists members with names not starting with `__`:

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 's
```

Caveats:

Classes can define a `__dir__()` method. If that method exists calling `dir()` will call `__dir__()`, otherwise Python will try to create a list of members of the class. This means that the dir function can have unexpected results. Two quotes of importance from the official python documentation :

> If the object does not provide **dir** (), the function tries its best to gather information from the object's **dict** attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom **getattr** ().

> **Note:** Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

Private Class Members

Private class members do not exist in Python, however there is a convention according to which a name prefixed with an underscore `_` should be considered a non-public.

To further avoid name clashes with names defined in subclasses, a mechanism called *name mangling* has been introduced. Any identifier prefixed with double underscores, e.g. `__membername` will be textually replaced with `_classname__membername`.

For example, for class:

```
class Class:

    def public(self):
        pass

    def _non_public(self):
        pass

    def __non_public_mangled(self):
        pass
```

`dir(Class)` will list the following members:

```
['_Class__non_public_mangled', '_non_public', 'public', ...]
```

Properties

Python classes support **properties** , which look like regular object variables, but with the possibility of attaching custom behavior and documentation.

```
class MyClass(object):

    def __init__(self):
       self._my_string = ""

    @property
    def string(self):
        """A profoundly important string."""
        return self._my_string

    @string.setter
    def string(self, new_value):
        assert isinstance(new_value, str), \
               "Give me a string, not a %r!" % type(new_value)
        self._my_string = new_value

    @string.deleter
    def x(self):
        self._my_string = None
```

The object's of class `MyClass` will *appear* to have have a property `.string`, however it's behavior is now tightly controlled:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

As well as the useful syntax as above, the property syntax allows for validation, or other augmentations to be added to those attributes. This could be especially useful with public APIs - where a level of help should be given to the user.

Another common use of properties is to enable the class to present 'virtual attributes' - attributes which aren't actually stored but are computed only when requested.

```
class Character(object):
    def __init__(name, max_hp):
        self._name = name
        self._hp = max_hp
        self._max_hp = max_hp

    # Make hp read only by not providing a set method
    @property
    def hp(self):
        return self._hp

    # Make name read only by not providing a set method
    @property
    def name(self):
        return self.name

    def take_damage(self, damage):
        self.hp -= damage
        self.hp = 0 if self.hp <0 else self.hp

    @property
    def is_alive(self):
        return self.hp != 0

    @property
    def is_wounded(self):
        return self.hp < self.max_hp if self.hp > 0 else False

    @property
    def is_dead(self):
        return not self.is_alive

bilbo = Character('Bilbo Baggins', 100)
bilbo.hp
# out : 100
bilbo.hp = 200
# out : AttributeError: can't set attribute
```

Borg Class

Alex Martelli provides an alternative to the popular Singleton Design Pattern where each instance of the class references the same attributes as the others, yet remains a distinct object.

This is accomplished by the following:

```
class Borg:
    __shared_state = {}

    def __init__(self):
```

```
    def __init__(self):
        self.__dict__ = self.__shared_state
```

Because a dictionary is a mutable type in Python, the value is shared by all the variables assigned to it. If one mutates, they all mutate. This allows us to change the attributes of all Borg objects in unison by setting them within a shared class-wide dictionary attribute.

The following code shows this in action:

```
borg_a = Borg()
borg_b = Borg()
borg_a.bark = True
```

We can now test if the bark attribute has been shared.

```
borg_b.bark
# Out: True
```

Default values for instance variables

If the variable contains a value of an immutable type (e.g. a string) then it is okay to assign a default value like this

```
class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width  * self.height

# Create some instances of the class
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # blue

red_rectangle = Rectangle(2, 3, 'red')
print(red_rectangle.color) # red
```

One needs to be careful when initializing mutable objects such as lists in the constructor. Consider the following example:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=[0,0], color='blue'):
        self.width = width
        self.height = height
        self.pos = pos
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [4, 0] r2's pos has changed as well
```

This behavior is caused by the fact that in Python default parameters are bound at function execution and not at function declaration. To get a default instance variable that's not shared among instances, one should use a construct like this:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # default value is [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [0, 0] r2's pos hasn't changed
```

See also Mutable Default Arguments and “Least Astonishment” and the Mutable Default Argument .

Singleton class

A singleton is a pattern that restricts the instantiation of a class to one instance/object. For more info on python singleton design patterns, see here .

```
class Singleton:
    def __new__(cls):
        try:
            it = cls.__it__
        except AttributeError:
            it = cls.__it__ = object.__new__(cls)
        return it

    def __repr__(self):
        return '<{}>'.format(self.__class__.__name__.upper())

    def __eq__(self, other):
         return other is self
```

Another method is to decorate your class. Following the example from this answer create a Singleton class:

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.

    The decorated class can define one `__init__` function that
    takes only the `self` argument. Other than that, there are
    no restrictions that apply to the decorated class.

    To get the singleton instance, use the `Instance` method. Trying
    to use `__call__` will result in a `TypeError` being raised.

    Limitations: The decorated class cannot be inherited from.

    """

    def __init__(self, decorated):
        self._decorated = decorated

    def Instance(self):
        """
        Returns the singleton instance. Upon its first call, it creates a
        new instance of the decorated class and calls its `__init__` method.
        On all subsequent calls, the already created instance is returned.

        """
        try:
            return self._instance
        except AttributeError:
            self._instance = self._decorated()
            return self._instance

    def __call__(self):
        raise TypeError('Singletons must be accessed through `Instance()`.')

    def __instancecheck__(self, inst):
```

To use you can use the `Instance` method

```
@Singleton
class Single:
    def __init__(self):
        self.name=None
        self.val=0
    def getName(self):
        print(self.name)

x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # outputs I'm single
y.getName() # outputs I'm single
```

Descriptors and Dotted Lookups

Descriptors are objects that are (usually) attributes of classes and that have any of `__get__` , `__set__` , or `__delete__` special methods.

Data Descriptors have any of `__set__` , or `__delete__`

These can control the dotted lookup on an instance, and are used to implement functions, `staticmethod` , `classmethod` , and `property` . A dotted lookup (e.g. instance `foo` of class `Foo` looking up attribute `bar` - i.e. `foo.bar`) uses the following algorithm:

1. `bar` is looked up in the class, `Foo` . If it is there and it is a **Data Descriptor** , then the data descriptor is used. That's how `property` is able to control access to data in an instance, and instances cannot override this. If a **Data Descriptor** is not there, then

2. `bar` is looked up in the instance `__dict__`. This is why we can override or block methods being called from an instance with a dotted lookup. If `bar` exists in the instance, it is used. If not, we then
3. look in the class `Foo` for `bar`. If it is a **Descriptor**, then the descriptor protocol is used. This is how functions (in this context, unbound methods), `classmethod`, and `staticmethod` are implemented. Else it simply returns the object there, or there is an `AttributeError`

Syntax

Parameters

Remarks