Descriptors and Metaclasses

Understanding and Using Python's More Advanced Features

A Tutorial at EuroPython 2018

24th May 2018

Edinburgh, UK

author: Dr.-Ing. Mike Müller
email: mmueller@python-academy.de
twitter: @pyacademy
version: 3.1

Python Academy - The Python Training Specialist

- Dedicated to Python training since 2006
- Wide variety of Python topics
- Experienced Python trainers
- Trainers are specialists and often core developers of taught software
- All levels of participants from novice to experienced software developers
- Courses for system administrators
- Courses for scientists and engineers
- Python for data analysis
- Open courses around Europe
- Customized in-house courses
- Python programming
- Development of technical and scientific software
- Code reviews
- Coaching of individuals and teams migrating to Python
- Workshops on developed software with detailed explanations

More Information: www.python-academy.com

Current Training Modules - Python Academy

As of 2018

Module Topic	Length (days)	In-house	Open
Python for Programmers	3	yes	yes
Python for Non-Programmers	4	yes	yes
Advanced Python	3	yes	yes
Python for Scientists and Engineers	3	yes	yes
Python for Data Analysis	3	yes	yes
Machine Learning with Python	3	yes	no
Professional Testing with Python	3	yes	yes
High Performance Computing with Python	3	yes	yes
Cython in Depth	2	yes	yes
Introduction to Django	4	yes	yes
Advanced Django	3	yes	yes
SQLAlchemy	1	yes	yes
High Performance XML with Python	1	yes	yes
Optimizing Python Programs	1	yes	yes
Python Extensions with Other Languages	1	yes	no
Data Storage with Python	1	yes	no
Introduction to Software Engineering with Python	1	yes	no
Introduction to wxPython	1	yes	no
Introduction to PySide/PyQt	1	yes	no
Overview of the Python Standard Library	1	yes	no
Threads and Processes in Python	1	yes	no
Windows Programming with Python	1	yes	no
Network Programming with Python	1	yes	no
Introduction to IronPython	1	yes	no

We offer on-site and open courses world-wide. We customize and extend training modules for your needs. We also provide consulting services such as code review, customized programming, and workshops.

More information: www.python-academy.com

Contents

1 Descriptors

1.1 Descriptors are Everywhere

Python uses descriptors at several places. Properties, functions and methods are based on the descriptor protocol. Descriptors determine how bound and unbound methods work and are involved in making static and class methods. Fortunately, you don't have to worry about them when using functions or static methods. On the other hand, understanding descriptors gives you a lot of insights how Python works.

1.2 Properties

Properties are examples for descriptors. A property can be constructed by the built-in function `property`:

```
>>> class Square(object):
...     def __init__(self, side):
...         self.side = side
...     def aget(self):
...         return self.side * self.side
...     def aset(self, value):
...         print("Can't set the area")
...     def adel(self):
...         print("Can't delete the area.")
...     area = property(aget, aset, adel, doc='Area of the square')
```

Now we can access `area` as it would be a normal attribute but the value is calculated on the fly:

```
>>> square = Square(5)
>>> square.area
25
>>> square.area = 20
Can't set the area
>>> square.side = 10
>>> square.area
100
```

Instead of using `area = property(aget, aset, adel, 'Area of the square')`, we can employ decorators:

```
"""Properties with decorators.
"""

from __future__ import print_function

class Square(object):
    """A square using properties with decorators.
    """

    def __init__(self, side):
```

```
        self.side = side

    @property
    def area(self):
        """Calculate the area of the square when the
        attribute is accessed."""
        return self.side * self.side

    @area.setter
    def area(self, value):
        """Don't allow setting."""
        print("Can't set the area")

    @area.deleter
    def area(self):
        """Don't allow deleting."""
        print("Can't delete the area.")


if __name__ == '__main__':

    square = Square(5)
    print('area:', square.area)
    print('try to set')
    square.area = 10
    print('area:', square.area)
    print('try to delete')
    del square.area
    print('area:', square.area)
```

The built-in decorator `property` marks the method `area` as a property. Now we can define our set and delete methods by using the same method name `area` and a decorator `area.setter` that is made up of the name of the method `area` and the attribute `setter`. The same rules apply to `area.deleter`.

We can also write a small convenience function that allows us to write the getter, setter and delete methods nested:

```
"""Use a decorator to allow nested properties.
"""

from __future__ import print_function

def nested_property(func):
    """Make defining properties simpler.
    """
    names = func()
    # We want the docstring from the decorated function.
    # If we do not set 'doc', we get the docstring from `fget`.
    names['doc'] = func.__doc__
```

```
    return property(**names)


class Square(object):
    """A square using properties with decorators.
    """

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

    @nested_property
    def area():
        """Property that defines is methods nested.
        """

        def fget(self):
            """
            Calculate the area of the square
            when the attribute is accessed.
            """
            return self.side * self.side

        def fset(self, value):
            """Don't allow setting."""
            print("Can't set the area")

        def fdel(self):
            """Don't allow deleting."""
            print("Can't delete the area.")

        return locals()


if __name__ == '__main__':

    square = Square(5)
    print('area:', square.area)
    print('try to set')
    square.area = 10
    print('area:', square.area)
    print('try to delete')
    del square.area
    print('area:', square.area)
    print(Square.area.__doc__)
```

1.3 Generalization

Descriptors allow to customize how attributes of a class or an instance are accessed or changed. We have to distinguish between data and non-data descriptors:

Data descriptors (a.k.a overriding descriptors) need to have a `__get__` and a `__set__` and optionally a `__delete__` method.

Non-data descriptors (a.k.a non-overriding descriptors) only have a `__get__` and no `__set__` method.

We define our data descriptor:

```
"""A typical data descriptor.
"""

from __future__ import print_function

class DataDescriptor(object):
    """A simple descriptor.
    """
    def __init__(self):
        self.value = 0
    def __get__(self, instance, cls):
        print('data descriptor __get__')
        return self.value
    def __set__(self, instance, value):
        print('data descriptor __set__')
        try:
            self.value = value.upper()
        except AttributeError:
            self.value = value
    def __delete__(self, instance):
        print("Don't like to be deleted." )
```

It prints a message if attributes are accessed or set. If the value that is to be set is a string it will be converted to all upper case.

Now we define another class that will use our data descriptor:

```
>>> class Strange(object):
...     attr = DataDescriptor()
```

and make an instance of it:

```
>>> strange = Strange()
```

We access our attribute:

```
>>> strange.attr
data descriptor __get__
0
```

This is equivalent to:

```
>>> type(strange).__dict__['attr'].__get__(strange, type(strange))
data descriptor __get__
0
```

We can also access the attribute through the class:

```
>>> Strange.attr
data descriptor __get__
0
```

This is equivalent to:

```
>>> Strange.__dict__['attr'].__get__(None, Strange)
data descriptor __get__
0
```

With `type` we get the class of an instance:

```
 >>> type(strange) is Strange
True
```

The object `__dict__` is a dictionary proxy of an class that holds its attributes:

```
>>> Strange.__dict__
<dictproxy object at 0x10119ab78>

>>> import pprint
>>> pprint.pprint(dict(Strange.__dict__))
{'__dict__': <attribute '__dict__' of 'Strange' objects>,
 '__doc__': None,
 '__module__': '__main__',
 '__weakref__': <attribute '__weakref__' of 'Strange' objects>,
 'attr': <datadescriptor.DataDescriptor object at 0x1011a5790>}
```

It is read-only. We cannot assign to it:

```
>>> Strange.__dict__['test'] = 5
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    Strange.__dict__['test'] = 5
TypeError: 'dictproxy' object does not support item assignment
```

Going through `__dict__`, we get our data descriptor:

```
>>> type(Strange.__dict__['attr'])
<class 'datadescriptor.DataDescriptor'>
```

Accessing `attr` with the normal attribute syntax we get our `__get__` method from the descriptor:

```
>>> type(Strange.attr)
data descriptor __get__
<type 'int'>
```

We can also set the attribute:

```
>>> strange.attr = 5
data descriptor __set__
>>> strange.attr
data descriptor __get__
5
```

When we set a string it will be upper cased:

```
>>> strange.attr = 'Hello'
data descriptor __set__
>>> strange.attr
data descriptor __get__
'HELLO'
```

Nothing is stored in the instance `__dict__`:

```
>>> strange.__dict__
{}
```

But when we set a new instance attribute it will be stored there:

```
>>> strange.new = 100
>>> strange.__dict__
{'new': 100}
```

`dir` shows the attribute `attr`:

```
>>> dir(strange)
['__class__', '__delattr__', '__dict__', '__doc__',
'__format__', '__getattribute__',     '__hash__',
'__init__', '__module__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__',     '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'attr', 'new']
```

It is stored in the class:

```
>>> Strange.attr
data descriptor __get__
```

```
'HELLO'

>>> strange.attr
data descriptor __get__
'HELLO'
```

When we try to delete our attribute, we just get what we implemented in `__delete__`:

```
>>> del strange.attr
Don't like to be deleted.
```

When we do not implement the `__set__` method we get a non-data descriptor:

```
>>> class NonDataDescriptor(object):
...     def __init__(self):
...         self.value = 0
...     def __get__(self, instance, cls):
...         print('non data descriptor __get__')
...         return self.value + 10
```

We add an attribute:

```
>>> class AddTen(object):
...     attr = NonDataDescriptor()
```

and make an instance:

```
>>> ten = AddTen()
>>> ten.attr
```

Now, we can access and set the attribute:

```
non data descriptor __get__
10
>>> ten.attr = 20
>>> ten.attr
20
```

But since our descriptor does not have a `__set__` method the set attribute ends up in the `__dict__` of the instance:

```
>>> ten.__dict__
{'attr': 20}

>>> ten.attr
20
```

The attribute of the class is still there:

```
>>> AddTen.attr
non data descriptor __get__
10
```

So the order of precedence is: data descriptors override instance dictionaries, which override non-data descriptors.

Descriptors are invoked by the `__getattribute__` method. Since this is only available for new-style classes, descriptors have to be new-style classes:

```
"""A typical data descriptor.
"""


class DataDescriptor:
    """A simple descriptor.
    """
    def __init__(self):
        self.value = 0
    def __get__(self, instance, cls):
        print 'data descriptor __get__'
        return self.value
    def __set__(self, instance, value):
        print 'data descriptor __set__'
        try:
            self.value = value.upper()
        except AttributeError:
            self.value = value
```

Overriding `__getattribute__` prevents automatic descriptor calls:

```
>>> class Overridden(object):
...     attr = DataDescriptor()
...     def __getattribute__(self, name):
...         print('no way')
...
>>> overridden = Overridden()
>>> overridden.attr
no way
```

There are two special methods that look very similar to `__getattr__` and `__getattribute__`. If Python cannot find an attribute it calls `__getattr__(object, name)`, where `name` is the attribute name as a string. The method `__getattribute__` is always called first and masks all attributes. If you use `self.attr` inside `__getattribute__` you trigger a recursive call of `__getattribute__` that eventually terminates with an exception due to the limited recursion depth.

1.4 Functions are Non-Data Descriptors

Functions have a `__get__` method and are non-data descriptors:

```
>>> def func():
...     pass
...
>>> func.__get__
<method-wrapper '__get__' of function object at 0x100468500>
```

Methods are just functions that are located in a class:

```
>>> class MyClass(object):
...     def meth(self):
...         pass
```

Accessing a method from an instance will call `__get__` with the instance as first argument. This is possible because the descriptor returns a bound method:

```
>>> inst = MyClass()
>>> inst.meth
<bound method MyClass.meth of <__main__.MyClass object at 0x10046ad50>>
```

If we access the method through the class we get an unbound method:

```
>>> MyClass.meth
<unbound method MyClass.meth>
```

In Python 3 things are simpler. There are no longer unbound methods only methods and functions:

```
# Python 3
>>> MyClass.meth
<function __main__.MyClass.meth>
>>> type(MyClass.meth)
function
>>> type(inst.meth)
method
```

In Python 2 we do have two different kinds do instance methods:

```
# Python 2
>>> type(MyClass.meth)
<type 'instancemethod'>
>>> type(inst.meth)
<type 'instancemethod'>
```

We can also see the function inside the `__dict__` of the class:

```
>>> MyClass.__dict__['meth']
<function meth at 0x1004682a8>
```

To get to the bound method `inst.meth` Python uses the `__get__` of the unbound method:

```
>>> bound_meth = type(inst).__dict__['meth'].__get__(inst, type(inst))
>>> bound_meth
<bound method MyClass.meth of <__main__.MyClass object at 0x1006e6a50>>
```

1.5 Descriptors Work in a Class

Descriptors are attributes of classes. Because they are often used to store data in attributes this may have unexpected implications if you are not aware of this fact. This simple example shows what happens when you use a descriptor:

```
# coding: utf-8

"""A descriptor works only in a class.

Storing attribute data directly in a descriptor
means sharing between instances.
"""

from __future__ import print_function


class DescriptorClassStorage(object):
    """Descriptor storing data in class."""

    def __init__(self, default=None):
        self.value = default

    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = value
```

We write a small test:

```
if __name__ == '__main__':
    class StoreClass(object):
        """All instances will share `attr`.
        """
        attr = DescriptorClassStorage(10)

    store1 = StoreClass()
```

```
    store2 = StoreClass()
    print('store1', store1.attr)
    print('store2', store2.attr)
    print('Setting store1 only.')
    store1.attr = 100
    print('store1', store1.attr)
    print('store2', store2.attr)
```

The output looks like this:

```
store1 10
store2 10
Setting store1 only.
store1 100
store2 100
```

Note that the value of `store2` changed and is the same as the value of `store1` even though we did not assign a new value to it. This might not be what you want.

1.6 Instance Storage with `WeakKeyDictionary`

If you wish to have different values for different instances, you can store the value in a dictionary in the descriptor, using the instance as a key. While this seems straight forward, it creates an additional reference to the instance, which will keep the instance alive even when it gets deleted. The `WeakKeyDictionary` from the `weakref` module can help to solve this problem, because it does not create a reference to the instance that would stick around after its deletion. Furthermore, we allow the user to provide a default value when instantiating the descriptor. The default of the default is `None`.

```
# coding: utf-8

"""A descriptor works only in a class.

We can store a different value for each instance in a dictionary
in the descriptor.
"""

from __future__ import print_function

from weakref import WeakKeyDictionary


class DescriptorWeakKeyDictStorage(object):
    """Descriptor that stores attribute data in instances.
    """

    def __init__(self, default=None):
        self._hidden = WeakKeyDictionary()
        self.default = default
```

```
    def __get__(self, instance, owner):
        return self._hidden.get(instance, self.default)

    def __set__(self, instance, value):
       self._hidden[instance] = value
```

Now we can change one attribute without affecting the other:

```
if __name__ == '__main__':
    class StoreInstance(object):
        """All instances have own `attr`.
        """
        attr1 = DescriptorWeakKeyDictStorage(10)
        attr2 = DescriptorWeakKeyDictStorage(-5)


    store1 = StoreInstance()
    store2 = StoreInstance()
    print('store1', store1.attr1)
    print('store2', store2.attr1)
    print('Setting store1 only.')
    store1.attr1 = 100
    store1.attr2 = -123
    store2.attr1 = 98765
    print('store1 attr1', store1.attr1)
    print('store1 attr2', store1.attr2)
    print('store2', store2.attr1)
    print('_hidden:', list(StoreInstance.__dict__['attr1']._hidden.items()))
    print('_hidden:', list(StoreInstance.__dict__['attr2']._hidden.items()))
    del store1
    print('Deleted store1')
    print('_hidden:', list(StoreInstance.__dict__['attr1']._hidden.items()))
```

The `WeakKeyDictionary` is empty after deletion of the instance used as key:

The output:

```
python weakkeydict_storage.py
store1 10
store2 10
Setting store1 only.
store1 100
store2 10
_hidden: [(<__main__.StoreInstance object at 0x10280a210>, 100)]
Deleted store1
_hidden: []
```

This implementation works only for hashbale instance. While instances are hashable by default, you can make them non-hashable by implementing `__eq__()` or raising a `NotImplementedError` in `__hash__()`:

```
from weakkeydict_storage import DescriptorWeakKeyDictStorage


class StoreInstance(object):
    """All instances have own `attr`.
    """
    attr1 = DescriptorWeakKeyDictStorage(10)
    attr2 = DescriptorWeakKeyDictStorage(-5)

    def __hash__(self):
        raise NotImplementedError
    #def __eq__(self, other):
    #    raise False

store1 = StoreInstance()
store2 = StoreInstance()
print('store1', store1.attr1)
print('store2', store2.attr1)
print('Setting store1 only.')
store1.attr1 = 100
store1.attr2 = -123
store2.attr1 = 98765
print('store1 attr1', store1.attr1)
print('store1 attr2', store1.attr2)
print('store2', store2.attr1)
print('_hidden:', list(StoreInstance.__dict__['attr1']._hidden.items()))
print('_hidden:', list(StoreInstance.__dict__['attr2']._hidden.items()))
del store1
print('Deleted store1')
print('_hidden:', list(StoreInstance.__dict__['attr1']._hidden.items()))
```

Let's have a look at an alternative.

1.7 Instance Storage a with Unique Identifier

Another way is to store the value in the instance itself. The function `uuid.uuid4()` provides a unique name that can be used to store the value:

```
# coding: utf-8

"""A descriptor works only in a class.

Storing attribute data in `instance.__dict__` can be a solution.
Creating a unique attribute name with the help of `uuid` and
using name mangling with the `__` prefix make things more robust.
```

```
"""

from __future__ import print_function

import uuid


class DescriptorInstanceStorage(object):
    """Descriptor that stores attribute data in instances.
    """

    def __init__(self, default=None):
        # unique name with uuid
        self._hidden_name = '__' + uuid.uuid4().hex
        self.default = default

    def __get__(self, instance, owner):
        return getattr(instance, self._hidden_name, self.default)

    def __set__(self, instance, value):
        setattr(instance, self._hidden_name, value)


if __name__ == '__main__':
    class StoreInstance(object):
        """All instances have own `attr`.
        """
        attr = DescriptorInstanceStorage(10)

    store1 = StoreInstance()
    store2 = StoreInstance()
    print('store1', store1.attr)
    print('store2', store2.attr)
    print('Setting store1 only.')
    store1.attr = 100
    print('store1', store1.attr)
    print('store2', store2.attr)
    print(store1.__dict__)
```

This introduces a strangely named instance attributed but using two leading underscores will hide the non-guessable name in an IDE or editor with tab completion.

1.8 Use Case - Attribute Checking

One interesting use case for descriptors is checking the attributes for certain conditions. Our descriptor takes an optional function `checker` that will be used to check the value we store in the instance:

```
# coding: utf-8
```

```
"""Example for descriptor that checks conditions on attributes.
"""
from __future__ import print_function

from weakref import WeakKeyDictionary


class Checked(object):
    """Descriptor that checks with a user-supplied check function
    if an attribute is valid.
    """

    def __init__(self, checker=None, default=None):
        self._values = WeakKeyDictionary()
        if checker:
            # checker must be a callable
            checker(default)
        self.checker = checker
        self.default = default

    def __get__(self, instance, owner):
        return self._values.get(instance, self.default)

    def __set__(self, instance, value):
        if self.checker:
            self.checker(value)
        self._values[instance] = value
```

We use a simple function `is_int` to make sure the value is an integer:

```
if __name__ == '__main__':

    def is_int(value):
        """Check if value is an integer.
        """
        if not isinstance(value, int):
            raise ValueError('Int required {} found'.format(type(value)))
```

Now we cannot use anything else as a value but an integer:

```
    class Restricted(object):
        """Use checked attributes.
        """
        attr1 = Checked(checker=is_int, default=10)
        attr2 = Checked(default=12.5)
        # Setting the default to float, `is_int` raises a `ValueError`.
        try:
```

```
            attr3 = Checked(checker=is_int, default=12.5)
        except ValueError:
            print('Cannot set default to float, must be int.')
            attr3 = Checked(checker=is_int, default=12)

    restricted = Restricted()
    print('attr1', restricted.attr1)
    restricted.attr1 = 100
    print('attr1', restricted.attr1)
    try:
        restricted.attr1 = 200.12
    except ValueError:
        print('Cannot set attr1 to float, must be int.')
        restricted.attr1 = 200
    print(restricted.attr1, restricted.attr2, restricted.attr3)
```

We see this output:

```
Cannot set default to float, must be int.
attr1 10
attr1 100
Cannot set attr1 to float, must be int.
```

1.9 When to Use Properties or Descriptors?

Properties seem to me more straight forward at the first look. But they have several disadvantages for some use cases:

1. You need to explicitly write `@property()` and `@meth.setter()` methods for each property. But you can re-use a descriptor.
2. You cannot use properties across different classes without inheritance. This is simple with a descriptor.

On the other hand, properties make it very easy to create read-only attributes. Only defining a `@property()` but no `@meth.setter()` will do. The default for a property without a setter will raise an attribute error when trying to set a property without a setter:

```
AttributeError
...
AttributeError: can't set attribute
```

1.10 Practical Tips for Using Descriptors

Descriptors can be complex. Therefore, things should kept as simple as possible. Some practical tips can help.

1.10.1 The "Same-Name-Trick"

You will get a nice validator, if you only implement a `__set__` method in the descriptor class and use the same name for the class attribute and the instance attribute:

```
class Percent(object):

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

    def __set__(self, instance, value):
        if not 0 <= value <= 100:
            msg ='Value must be between 0 and 100. Got: {}.'.format(value)
            raise ValueError(msg)
        instance.__dict__[self.attr_name] = value


class WithPercent(object):
    percent = Percent('percent')

    def __init__(self, percent):
        self.percent = percent
```

Important: You need to use the name `percent` twice when defining the class attribute, once as the attribute name and once as an argument in form of a string. Furthermore, you need to use the same name as the instance attribute name, i.e. `self.percent`.

Now, a percentage value between `0` and `100` works:

```
>>> valid = WithPercent(50)
... valid.percent
...
50
```

but a value outside this range will give you an appropriate error message:

```
>>> no_valid = WithPercent(150)
ValueError
...
ValueError: Value must be between 0 and 100. Got: 150.
```

1.10.2 Read-Only Attributes Require a `__set__` Method

You need to implement a `__set__` method if you want to make an attribute read_only:

```
class WrongReadOnly(object):

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

    def __get__(self, instance, owner):
        return 42
```

```
class Wrong(object):
    attr_wrong_ro = WrongReadOnly('attr_wrong_ro')
    def __init__(self, attr):
        attr_wrong_ro = attr


class RightReadOnly(object):

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

    def __get__(self, instance, owner):
        return 42

    def __set__(self, instance, value):
        raise AttributeError('attribute "{}" is read-only'.format(self.attr_name))

class Right(object):
    attr_ro = RightReadOnly('attr_ro')

    def __init__(self, attr):
        attr_ro = attr
```

Implementing only a `__get__` is not enough:

```
>>> w = Wrong(10)
>>> w.attr_wrong_ro
42
>>> w.attr_wrong_ro = 43
```

The `w.attr_wrong_ro = 43` is not intercepted by a `__set__` method of the descriptor. Therefore, the value `43` will be stored in `w.__dict__`:

```
>>> w.__dict__
{'attr_wrong_ro': 43}
```

and `43` will show up when you access the attribute the next time:

```
>>> w.attr_wrong_ro
43
```

Doing it right, you will get a an appropriate exception:

```
>>> r = Right(10)
>>> r.attr_ro
42
>>> r.attr_ro = 43
```

```
AttributeError
...
AttributeError: attribute "attr_ro" is read-only
```

1.10.3 Caching with a Non-Data Descriptor

You can take advantage of the features of a non-data descriptor for caching. There is no `__set__` method and the long calculation happens in the `__get__` method. In addition, the `__get__` method sets the calculated value as an instance attribute: `instance.__dict__[self.attr_name] = value`.

```
class Cache(object):
    def __init__(self, attr_name):
        self.attr_name = attr_name

    def __get__(self, instance, owner):
        import time
        print('doing expensive calculation at first access')
        time.sleep(1)
        value = 42
        instance.__dict__[self.attr_name] = value
        return value

class WithCache(object):
    attr = Cache('attr')
```

The long calculation will only happen the first time the attribute is accessed:

```
>>> c = WithCache()
>>> c.attr
doing expensive calculation at first access
42
```

Because the calculated value is stored in `c.__dict__`, and there is no `__get__` method in the descriptor, there will be no calculation for the second (and any subsequent) access:

```
>>> c.attr
42
```

1.10.4 Methods and Special Methods are Different

Special methods are treated differently from normal methods. For example, if we have a class that implements the special method `__contains__`:

```
class Special(object):
    def __contains__(self, item):
```

```
        print('got', item)
        return True
```

we can use the `in` operator on a instance of this class:

```
>>> s = Special()
>>> 'abc' in s
True
```

Assigning a new value to `__contains__` does not change this:

```
>>> s.__contains__ = 43
>>> 'abc' in s
True
```

The reason is that special methods are looked up in the class:

```
>>> s.__class__.__contains__(s, 'abc')
True
```

This is different for normal methods. This class has just one method:

```
class Unprotected(object):

    def meth(self):
        print('called')
        return 42
```

Calling this method works:

```
>>> u = Unprotected()
>>> u.meth()
42
```

But the attribute `method` can be overridden:

```
>>> u.meth = 43
>>> u.meth
43
```

Typically, this no problem and any trivial testing of a program will make these kind of errors obvious. But, just for fun, let's try to make Python to disallow this. This descriptor will do the work:

```
from functools import partial

class Protector(object):
```

```
    def __init__(self, cls, meth):
        self.cls = cls
        self.meth = meth

    def __get__(self, instance, owner):
        return partial(self.meth, self)

    def __set__(self, instance, value):
        raise AttributeError('cannot set method')
```

The `__init__` just stores the method and the class. The `__get__` method returns the bound method. Using `functools.partial` is really handy here. The result is a new function with the argument `self` already supplied as the first argument, i.e. instead of `self` and potential other arguments, it takes only the latter. This turns the function `self.meth` into a bound method. Now it is bound to the instance `self`. Finally, the `__set__` method prevents overriding the method.

For convenience, let's use a class decorator that applies the descriptor to all methods of a class:

```
def protect_methods(cls):
    for name, obj in cls.__dict__.items():
        if callable(obj):
            setattr(cls, name, Protector(cls, obj))
    return cls
```

All callable attributes get replaced by an instance of the descriptor `Protector`.

Now, let's apply this decorator to our class:

```
@protect_methods
class Protected(object):

    def meth1(self):
        print('called')
        return 42

    def meth2(self):
        print('called')
        return 142
```

Calling a method works as expected:

```
>>> p = Protected()
>>> p.meth1()
called
42
```

But assigning to a method name does not work anymore:

```
>>> p.meth1 = 43
AttributeError
...
AttributeError: cannot set method
```

1.11 Exercises

1. Write a descriptor that only allows positive numbers to be set.
2. Extend this descriptor to return rounded values with 2 decimals.
3. Prevent the deletion of this attribute.
4. Write a descriptor `Total` that takes a VAT rate at instantiation time. The getter calculates the total price from a net prices that is an attribute of the instance in the getter call. Prevent the setting of the total price. Write three classes `<Country>Price` for three different countries that use this descriptor with different VAT rates. Make instances of these classes and print total prices from given net prices.

2 Metaclasses

2.1 What is a Metaclass?

Metaclasses are a very advanced topic in Python. Nevertheless, they can be very useful for certain tasks. They can be used to change how classes are created. They are useful for frameworks and libraries that are used by application developers. They are also useful to introduce things like aspect-oriented programming, interface-oriented programming or prototype-based programming. They can automatically register classes, enforce constrains on classes, allow a declarative approach and other things of this type.

The concept is not very difficult to understand. Let's look at them using the differences between old-style and new-style classes in Python 2. There are no old-style classes in Python 3. We define a new-style class:

```
>>> class NewStyle(object):
...     pass
...
```

that is of type:

```
>>> type(NewStyle)
<type 'type'>
```

On the other hand, an old-style class:

```
>>> class OldStyle:
...     pass
...
```

is of type:

```
>>> type(OldStyle)   # Python 2
<type 'classobj'>
```

We can also provide the type by the class attribute `__metaclass__`:

```
>>> class NewStyle2:  # Python 2
...     __metaclass__ = type
...
```

Now we have also a new style class:

```
>>> type(NewStyle2)
<type 'type'>
```

A metaclass is also just the type of the class. Let's look at the way Python 2 finds the metaclass:

1. It looks into the dictionary of the class for the key `__metaclass__`. We put it there in the example above.

2. If it cannot find the key there, it looks in the tuple `__bases__` and takes the type of the first entry:

```
>>> NewStyle.__bases__
(<type 'object'>,)
>>> type(NewStyle.__bases__[0])
<type 'type'>
```

3. If it can not find the key there, it looks for a global variable `__metaclass__` making a default for all classes in the module possible:

```
>>> globals()['__metaclass__']
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
KeyError: '__metaclass__'
>>> __metaclass__ = type
>>> globals()['__metaclass__']
<type 'type'>
```

4. Finally, if it can not find the key there it uses `types.ClassType`, i.e. defaults to old style classes in Python 2:

```
>>> import types
>>> types.ClassType
<type 'classobj'>
>>>
```

There are no old-style classes in Python 3 and the default is always a new-style class, i.e. the metaclass is `type`. This is the Python 3 syntax for setting a metaclass:

```
>>> class Python3Class(metaclass=type):
...     pass
```

The normal method of defining a class:

```
>>> class C(object):
        x = 1
```

can be expressed as making an instance of `type` handing in the name of the class, the tuple with classes to inherit from, and a dictionary with the attributes of the class:

```
>>> C = type('C', (object,), dict(x=1))
```

Of course, in this way we can also add methods to the class:

```
>>> def __init__(self, a):
...     self.a = a
...
```

```
>>> def add(self, a, b):
...     return a + b
...

>>> C = type('C', (object,), dict(__init__=__init__, add=add))
>>> c = C(100)
>>> c.a
100
>>> c.add(30, 40)
70
>>>
```

We can define our own metaclass:

```
>>> class NewMeta(type):
...     def __str__(cls):
...         return 'This class is of type %s and has name %s.' \
                %(type(cls), cls.__name__)
...
```

And make a class of this new type:

```
>>> class C:  # Python 2
...     __metaclass__ = NewMeta
...

>>> class C(metaclass=NewMeta):  #Python 3
...    pass
```

If we print it we get the message we defined with `__str__`:

```
>>> print(C)
This class is of type <class '__main__.NewMeta'> and has name C.
```

The type of an instance of this class tells the same:

```
>>> c = C()
>>> print(type(c))
This class is of type <class '__main__.NewMeta'> and has name C.
```

2.2 Working with `__new__` and `__init__`

Most of the work in a metaclass is often done in either `__new__` or `__init__`, where `__new__` works before a class object exists and `__init__` needs to have a class object as a first argument:

We define how a class is defined:

```
>>> class MyMeta(type):
...     def __new__(mcl, name, bases, cdict):
...         print('mcl', mcl)
...         print('name', name)
...         print('bases', bases)
...         print('class dict', cdict)
...         return super(MyMeta, mcl).__new__(mcl, name, bases, cdict)
...
```

The first argument is `mcl`, which is the metaclass and we return the newly build class. When we define a class using this metaclass:

```
>>> class MyClass: # Python 2
...     __metaclass__ = MyMeta
...
mcl <class '__main__.MyMeta'>
name MyClass
bases ()
dict {'__module__': '__main__', '__metaclass__': <class '__main__.MyMeta'>}


>>> class MyClass(metaclass = MyMeta): # Python 3
...
mcl <class '__main__.MyMeta'>
name MyClass
bases ()
dict {'__module__': '__main__', '__metaclass__': <class '__main__.MyMeta'>}
```

we have access to all information that will be used to build the class.

On the other hand `__init__` needs the class as the first argument conventionally called `cls`:

```
>>> class MyMeta(type):
...     def __init__(cls, name, bases, cdict):
...         print('cls', cls)
...         print('name', name)
...         print('bases', bases)
...         print('class dict', cdict)
...         super(MyMeta, cls).__init__(name, bases, cdict)
```

There is no return here just like in the `__init__` of ordinary classes. The class object already existed before:

```
>>> class MyClass:
...     __metaclass__ = MyMeta
...
self <class '__main__.MyClass'>
name MyClass
```

```
bases ()
dict {'__module__': '__main__', '__metaclass__': <class '__main__.MyMeta'>}
```

Should you use `__new__` or `__init__`? There are a few rules here:

1. `__new__` will be called before `__init__`.
2. Changes to the class dictionary (`cdict`) need to be made in `__new__` to have any effect on the created class.

Recommendation: Use `__init__` if you only need to get information from the class dictionary and `__new__` if you need to actually change the class dictionary. This helps to prevent unwanted modifications in the class dictionary.

2.3 Supporting Python 2 and 3

While Python 2 and Python 3 have different syntax for metaclasses, it is possible to use a clever function from the Jinja2 project:

```
# file: meta_2_3.py

"""
The code is a bit hard to understand. The basic idea is exploiting the idea
that metaclasses can customize class creation and are picked by by the parent
class. This particular implementation uses a metaclass to remove its own parent
from the inheritance tree on subclassing. The end result is that the function
creates a dummy class with a dummy metaclass. Once subclassed the dummy
classes metaclass is used which has a constructor that basically instances a
new class from the original parent and the actually intended metaclass.
That way the dummy class and dummy metaclass never show up.

From:
 http://lucumr.pocoo.org/2013/5/21/porting-to-python-3-redux/#metaclass-syntax-changes

Used in:

* Jinja2
* SQLAlchemy
* future (python-future.org)

"""

from __future__ import print_function
import platform


# from jinja2/_compat.py
def with_metaclass(meta, *bases):
    # This requires a bit of explanation: the basic idea is to make a
    # dummy metaclass for one level of class instanciation that replaces
```

```
    # itself with the actual metaclass.  Because of internal type checks
    # we also need to make sure that we downgrade the custom metaclass
    # for one level to something closer to type (that's why __call__ and
    # __init__ comes back from type etc.).
    #
    # This has the advantage over six.with_metaclass in that it does not
    # introduce dummy classes into the final MRO.
    class metaclass(meta):
        __call__ = type.__call__
        __init__ = type.__init__

        def __new__(cls, name, this_bases, d):
            if this_bases is None:
                return type.__new__(cls, name, (), d)
            return meta(name, bases, d)

    return metaclass('temporary_class', None, {})


if __name__ == '__main__':
```

Now we can give our class a metaclass without triggering a syntax error:

```
    class BaseClass(object):
        pass


    class MetaClass(type):
        """Metaclass for Python 2 and 3.
        """
        def __init__(cls, name, bases, cdict):
            print('It works with {impl} version {ver}.'.format(
                impl=platform.python_implementation(),
                ver=platform.python_version()))
            super(MetaClass, cls).__init__(name, bases, cdict)


    class Class(with_metaclass(MetaClass, BaseClass)):
        # BaseClass is optional.
        pass
```

This program runs with Python 2 and 3 and supports PyPy as well:

```
$ python meta_2_3.py
It works with CPython version 2.7.9.
$ python3 meta_2_3.py
It works with CPython version 3.4.0.
$ pypy meta_2_3.py
```

```
It works with PyPy version 2.7.6.
$ pypy3 meta_2_3.py
It works with PyPy version 3.2.5.
```

2.4 Use Case Classes with Methods Only

We would like to have a class that cannot have class attributes other than methods. So we would like to forbid class attributes that cannot be called. Using a metaclass we can override its `__init__`:

```
#file: noclassattr.py

"""Preventing non-callable class attributes with a metaclass.
"""

from __future__ import print_function

class NoClassAttributes(type):
    """No non-callable class attributes allowed
    """
    def __init__(cls, name, bases, cdict):
        allowed = set(['__module__', '__metaclass__', '__doc__',
                       '__qualname__'])
        for key, value in cdict.items():
            if (key not in allowed) and (not callable(value)):
                msg = 'Found non-callable class attribute "%s". ' % key
                msg += 'Only methods are allowed.'
                raise Exception(msg)
        super(NoClassAttributes, cls).__init__(name, bases, cdict)
```

Except for a few attributes that are always present like `__module__`, `__doc__`, `__metaclass__` (Python 2) and `__qualname__` (Python 3), we don't allow attributes that are not callable and raise an exception if they are.

Now, we make a class that has only one method:

```
if __name__ == '__main__':

    from meta_2_3 import with_metaclass

    class AttributeChecker(with_metaclass(NoClassAttributes)):
        """Base class for meta.
        """
        pass

    class AttributeLess(AttributeChecker):
        """Only methods work.
        """
        def meth(self):
```

```
            """This is allowed'
            """
            print('Hello from AttributeLess.')

    attributeless = AttributeLess()
    attributeless.meth()


    class
```

This prints:

```
Hello from AttributeLess.
```

If we also add `a = 10`:

```
    class WithAttribute(AttributeChecker):
        """Has non-callable class attribute.
        Will raise an exception.
        """
        a = 10
        def meth(self):
            """This is allowed'
            """
            print('Hello from WithAttribute')
```

We get an exception:

```
Traceback (most recent call last):
...
Exception: Found non-callable class attribute "a". Only methods are allowed.
```

2.5 Metaclasses vs. Class Decorators

Class decorators can provide similar functionality as metaclasses. Let's re-write our example that allows only callable attributes with a class decorator:

```
# file: class_deco.py

def noclassattr_deco(cls):
    """Class decorator to allow only callable attributes.
    """
    allowed = set(['__module__', '__metaclass__', '__doc__', '__qualname__',
                   '__weakref__', '__dict__'])
    for key, value in cls.__dict__.items():
        if (key not in allowed) and (not callable(value)):
            msg = 'Found non-callable class attribute "%s". ' % key
```

```
            msg += 'Only methods are allowed.'
            raise Exception(msg)
    return cls
```

Now we can use our decorator to inject the desired behavior:

```
if __name__ == '__main__':

    @noclassattr_deco
    class AttributeLess(object):
        """Only methods work.
        """
        def meth(self):
            """This is allowed'
            """
            print('Hello from AttributeLess.')

    attributeless = AttributeLess()
    attributeless.meth()

    @noclassattr_deco
    class WithAttribute(object):
        """Has non-callable class attribute.
        Will raise an exception.
        """
        a = 10
        def meth(self):
            """This is allowed'
            """
            print('Hello from WithAttribute')
```

However, we need to add this decorator to every single class definition. With metaclass, we can use inheritance, i.e. specify the metaclass only for a parent class. All children will get this metaclass automatically.

2.6 Use Case Working with Slots

We combine slots with a descriptor and define a descriptor that allows only a given type:

```
# file: slotstyped.py

"""Use of descriptor and metaclass to get slots with
given types.
"""

from __future__ import print_function

class TypDescriptor(object):
    """Descriptor with type.
```

```
    """

    def __init__(self, data_type, default_value=None):
        self.name = None
        self._internal_name = None
        self.data_type = data_type
        if default_value:
            self.default_value = data_type(default_value)
        else:
            self.default_value = data_type()

    def __get__(self, instance, cls):
        return getattr(instance, self._internal_name, self.default_value)

    def __set__(self, instance, value):
        if not isinstance(value, self.data_type):
            raise TypeError('Required data type is %s. Got: %s' % (
            self.data_type, type(value)))
        setattr(instance, self._internal_name, value)

    def __delete__(self, instance):
        raise AttributeError('Cannot delete %r' % instance)
```

The `__init__` takes a data type and the `__set__` method makes sure that the value has this data type.

Now we have a new metaclass:

```
class TypeProtected(type):
    """Metaclass to save descriptor values in slots.
    """

    def __new__(mcl, name, bases, cdict):
        slots = []
        for attr, value in cdict.items():
            if isinstance(value, TypDescriptor):
                value.name = attr
                value._internal_name = '_' + attr
                slots.append(value._internal_name)
        cdict['__slots__'] = slots
        return super(TypeProtected, mcl).__new__(mcl, name, bases, cdict)
```

The method `__new__` adds all attributes that are instances of `TypeDescriptor` to the slots.

Now we can define our class with attributes that have a fixed data type:

```
if __name__ == '__main__':

    from meta_2_3 import with_metaclass
```

```
    class Typed(with_metaclass(TypeProtected)):
        pass

    class MyClass(Typed):
        """Test class."""
        attr1 = TypDescriptor(int)
        attr2 = TypDescriptor(float, 5.5)
```

Using the instance of this class does not allow to add new attributes because we work with slots:

```
    def main():
        """Test it.
        """
        my_inst = MyClass()
        print(my_inst.attr1)
        print(my_inst.attr2)
        print(dir(my_inst))
        print(my_inst.__slots__)
        my_inst.attr1 = 100
        print(my_inst.attr1)
        # this will fail
        try:
            my_inst.unknown = 100
        except AttributeError:
            print('cannot do this')

    main()
```

2.7 Use Case Counting The Number of Class Definitions

Metaclasses allow to add debug information. This example defines a new metaclass that counts how many classes are defined and stores their names:

```
# file: autometa_python2.py

"""Example usage of a metaclass.

We change the metaclass of classes that inherit from `object`.
"""

from __future__ import print_function

import __builtin__


class DebugMeta(type):
    """Metaclass to be used for debugging.
```

```
    """
    names = []
    counter = -1  # Do not count definition of new_object`.

    def __init__(cls, name, bases, cdict):
        """Store all class names and count how many classes are defined.
        """
        if DebugMeta.counter >= 0:
            DebugMeta.names.append('%s.%s' % (cls.__module__, name))
            super(DebugMeta, cls).__init__(name, bases, cdict)
        DebugMeta.counter += 1

    def report(cls):
        print('Defined %d classes.' % DebugMeta.counter)
        print(DebugMeta.names)
```

Now we use this metaclass for our new class and replace the built-in `object` with it:

```
class new_object(object):
    """Replacement for the built-in `object`.
    """
    __metaclass__ = DebugMeta


def set_new_meta():
    """We actually change a built-in. This is a very strong measure.
    """
    __builtin__.object = new_object
```

We can also use the new metaclass in other scripts. Calling `set_new_meta()` starts the logging of class definitions.

```
# file: use_autometa_python2.py

from autometa_python2 import set_new_meta

set_new_meta()
```

Now we define some classes:

```
class SomeClass1(object):
    """Test class.
    """
    pass


class SomeClass2(object):
```

```
    """Test class.
    """
    def __init__(self, arg1):
        self.arg1 = arg1

    def compute(self, arg2):
        return self.arg1 + arg2


class SomeClass3():
    """Test class. Does NOT inherit from object.
    """
    pass
```

and check if it works:

```
if __name__ == '__main__':

    def test():
        """Make an instance and write the report.
        """
        inst = SomeClass2(10)
        assert inst.compute(10) == 20
        object.report()

    test()
```

The output looks like this:

```
Defined 68 classes.
['threading._Verbose', 'threading._RLock',
...
```

Unfortunately, this approach has a problem if libraries use metaclasses in a certain way:

```
# file: problem_autometa_python2.py

from autometa_python2 import set_new_meta

set_new_meta()

import argparse
import doctest
import logging
import os
import shutil
import subprocess
import sys
```

```
import threading

# Comment out one of the lines below to see the problem.
# You may need to install the library.
# Try other large libraries you use.

# import matplotlib
# import numpy
# import sqlalchemy

object.report()
```

For example, if we uncomment `sqlalchemy` we get an error message:

```
...

TypeError: Error when calling the metaclass bases
    metaclass conflict: the metaclass of a derived class must be a
    (non-strict) subclass of the metaclasses of all its bases
```

The reason is multiple inheritance in SQLAlchemy. This is essentially what happens:

```
# file: base_conflict.py

from meta_2_3 import with_metaclass

class MetaClass1(type):
    pass

class MetaClass2(type):
    pass

class BaseClass1(with_metaclass(MetaClass1)):
    pass

class BaseClass2(with_metaclass(MetaClass2)):
    pass

class DoesNotWorkClass(BaseClass1, BaseClass2):
    pass
```

```
   class DoesNotWorkClass(BaseClass1, BaseClass2):
TypeError: Error when calling the metaclass bases
    metaclass conflict: the metaclass of a derived class must be a
    (non-strict) subclass of the metaclasses of all its bases
```

We need a situation like this:

```
# file: submeta.py

from meta_2_3 import with_metaclass

class BaseMetaClass(type):
    pass

class SubMetaClass(BaseMetaClass):
    pass

class BaseClass1(with_metaclass(BaseMetaClass)):
    pass

class BaseClass2(with_metaclass(SubMetaClass)):
    pass

class WorkingClass(BaseClass1, BaseClass2):
    pass
```

to make it work. I spent a few days trying different approaches but I could not solve this problem. However, Python 3 offers a solution.

There are a few changes when using Python 3:

1. The name of the module `__builtin__` changed to `builtins`.
2. There is a new syntax for setting the metaclass using a keyword argument instead of a class attribute.
3. Also, there are no longer old-style classes anymore and hence classes do not need to inherit from `object` anymore.
4. There is a new built-in function `__build_class__` with the signature `(func, name, *bases, metaclass=None, **kwds)` that Python 3 uses when creating a new class.

Let's override `__build_class__` to find out what classes are defined. We import some modules and define a custom exception:

```
# file: classwatcher.py

"""Find all defined classes.

Needs Python 3.
"""

import builtins
from collections import Counter


class MultipleInstancesError(Exception):
    """Allow only one instance.
```

```
    """
    pass
```

Our class ClassWatcher does the work:

```
class ClassWatcher(object):
    """After instantiation of this class, all newly defined classes will
       be counted.

    Only one instance of this class is allowed.
    """

    def __new__(cls, only_packages=frozenset(), ignore_packages=frozenset()):
        """
        only_packages: positive list of package names
                       Only these packages will be used.
        ignore_packages: negative list of package names
                         These packages will not be considered.

        The names in both sets are checked with `.startwith()'.
        This allows to filter for  `package` or `package.subpackage` and
        so on.
        For example, you can include `package` with `only_packages` and
        and then exclude `package.subpackage` with ignore_packages`.
        """
        if hasattr(cls, '_instance_exists'):
            msg = 'Only one instance of ClassWatcher allowed.'
            raise MultipleInstancesError(msg)
        cls._instance_exists = True
        cls.defined_classes = Counter()
        cls.activate(only_packages, ignore_packages)
        return super().__new__(cls)
```

The set `only_packages` specifies the names of the packages or modules we are interested in. We can also explicitly exclude packages or modules with `ignore_packages`. There can only be one instance of this class. An instance of `Counter` that will hold the number of defined classes. The method `activate()` starts watching class definitions.

Now, we define a new function that will replace the built-in `__build_class__`:

```
    @staticmethod
    def __build_class__(func, name, *bases, metaclass=type, **kwds):
        """Replacement for the the built-in `__build_class__`.

        Use on your own risk.
        """
        name = '{}.{}'.format(func.__module__, func.__qualname__)

        if not ClassWatcher.only_packages:
```

```
            add_name = True
        else:
            add_name = False
            for p_name in ClassWatcher.only_packages:
                if name.startswith(p_name):
                    add_name = True
        for p_name in ClassWatcher.ignore_packages:
            if name.startswith(p_name):
                add_name = False
        if add_name:
            ClassWatcher.defined_classes[name] += 1
        cls = ClassWatcher.orig__build_class__(func, name, *bases,
                                               metaclass=metaclass, **kwds)
        return cls
```

After checking if the class needs to be watched by going through the positive and negative name list, we increment our counter and call the original `__build_class__` function.

The methods `activate()` and `deactivate()` can turning on and off watching of class definitions:

```
    @classmethod
    def activate(cls, only_packages=frozenset(), ignore_packages=frozenset()):
        """Replace the built-in `__build_class__` with a customer version.
        """
        cls.orig__build_class__ = builtins.__build_class__
        builtins.__build_class__ = cls.__build_class__
        cls.only_packages = frozenset(only_packages)
        cls.ignore_packages = frozenset(ignore_packages)

    @classmethod
    def deactivate(cls):
        """Set built-in `__build_class__` back to real built-in.
        """
        builtins.__build_class__ = cls.orig__build_class__
```

The method `report()` produces nice output from the watching results:

```
    def report(self, limit=20):
        """Show results.
        """
        print('total defined classes:', sum(self.defined_classes.values()))
        print('total unique classes: ', len(self.defined_classes))
        all_names = self.defined_classes.most_common()
        width = max(len(name[0]) for name in all_names[:limit])
        count_width = 10
        print('{:{width}}{:>{count_width}}'.format('Name', 'Count',
            width=width, count_width=count_width))
        print('#' * (width + count_width))
        for counter, (cls_name, count) in enumerate(all_names, 1):
```

```
            print('{:{width}s}{:{count_width}d}'.format(cls_name, count,
                width=width, count_width=count_width))
            if counter >= limit:
                print('...')
                print('Skipped', len(all_names) - counter, 'additional lines.')
                break
```

Now we can test our program with some libraries.

```
# file: use_classwatcher.py

from classwatcher import ClassWatcher

watcher = ClassWatcher()

import argparse
import doctest
import logging
import os
import shutil
import subprocess
import sys
import threading

import numpy
import matplotlib
import pandas
import scipy
import sqlalchemy

watcher.report(20)
```

The output looks like this:

```
total defined classes: 1566
total unique classes:  1546
Name                                        Count
################################################
ctypes.CFunctionType                            6
sqlalchemy.sql.sqltypes.Comparator              4
sqlalchemy.sql.type_api.Comparator              3
pytz.lazy.LazyList                              3
pytz.lazy.LazySet                               3
sqlalchemy.util.compat.metaclass                3
dateutil.parser._result                         2
namedtuple_ArgSpec.ArgSpec                      2
namedtuple_DecimalTuple.DecimalTuple            2
ctypes._FuncPtr                                 2
```

```
pytz.exceptions.NonExistentTimeError         1
namedtuple__ASN1Object._ASN1Object           1
argparse._Section                            1
pandas.tseries.offsets.BQuarterBegin         1
sqlalchemy.sql.compiler.DDLCompiler          1
pandas.core.internals.ObjectBlock            1
numexpr.necompiler.Register                  1
ctypes.c_ushort                              1
dateutil.rrule.weekday                       1
pandas.io.pytables.WORMTable                 1
...
Skipped 1526 additional lines.
```

2.8 Metaclasses in the Wild

Metaclasses are used in several widely-used projects. These include SQLAlchemy, Django and Jinja2.

2.9 Be Prepared for Python 3

Python 3 offers the new special method `__prepare__`. This will be active even earlier than `__new__`. Actually, `__new__` gets an already created class dictionary. One implication is that the order in witch attributes are defined will be lost due to unordered nature of Python dictionaries.

One good example for using `__prepare__` is to record the definition order of attributes. First we need to write a dictionary-like class that will do the recording:

```
# file: pepare.py

"""Using `__prepare__` to preserve definition order of attributes.

Needs Python 3.
"""


class AttributeOrderDict(dict):
    """Dict-like object used for recording attribute definition order.
    """

    def __init__(self, no_special_methods=True, no_callables=True):
        self.member_order = []
        self.no_special_methods = no_special_methods
        self.no_callables = no_callables
        super().__init__()
```

We inherit from the built-in `dict` the optional parameters `no_special_methods` and `no_callables` determine if we want to also record special methods (with leading and trailing double underscores) and methods, i.e. callables. The recorded names will end up in the list `member_order` in the order they were defined.

We need to implement `__setitem__` to get it do what we want:

```
    def __setitem__(self, key, value):
        skip = False
        # Don't allow setting more than once.
        if key in self:
            raise AttributeError(
                'Attribute {} defined more than once.'.format(key))
        # Skip callables if not wanted.
        if self.no_callables:
            if callable(value):
                skip = True
        # Skip special methods if not wanted.
        if self.no_special_methods:
            if key.startswith('__') and key.endswith('__'):
                skip = True
        if not skip:
            self.member_order.append(key)
        super().__setitem__(key, value)
```

We do not allow more than one definition of an attribute with the same name. Furthermore, we filter for special methods and callables according to the given options. Finally, if the attribute name (`key`) is one we want, we append it to our ordered member list. We also call `__setitem__` from our parent class, i.e. `dict` to put the name value pair into this regular dictionary.

Now we implement our metaclass:

```
class OrderedMeta(type):
    """Meta class that helps to record attribute definition order.
    """

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        return AttributeOrderDict(**kwargs)
```

The `__prepare__` method takes `kwargs`. These are `no_special_methods` and `no_callables` and will be handed over to the constructor of `AttributeOrderDict`.

Now, we implement `__new__`:

```
    def __new__(mcs, name, bases, cdict, **kwargs):
        cls = type.__new__(mcs, name, bases, cdict)
        cls.member_order = cdict.member_order
        cls._closed = True
        return cls

    # Needed to use up kwargs.
    def __init__(cls, name, bases, cdict, **kwargs):
        super().__init__(name, bases, cdict)
```

It also takes `kwargs`, just like `__prepare__` and `__init__`. Actually, the only purpose for defining an `__init__` method here is to take these extra arguments. We create a new class and set `member_order`

that comes as an attribute with the class dictionary as a class attribute. In addition, we set `cls._closed` to `True`. This indicates that we don't want any more attributes to be set. This flag will be used in `__setattr__`:

```
    def __setattr__(cls, name, value):
        # Later attribute additions go through here.
        if getattr(cls, '_closed', False):
            raise AttributeError(
                'Cannot set attribute after class definition.')
        super().__setattr__(name, value)
```

The method `__setattr__` gets called for all attribute assignments after `__prepare__` is done. The first two calls to this method will be triggered by `cls.member_order = cdict.member_order` and `cls._closed = True` in `__new__`. It is important to use `super().__setattr__(name, value)` here to avoid the infinite recursion that `setattr(cls, name, value)` would cause.

Now, we can define our class:

```
if __name__ == '__main__':

    class MyClass(metaclass=OrderedMeta, no_callables=False):
        """Test class with extra attribute `member_order`.
        """
        attr1 = 1
        attr2 = 2

        def method1(self):
            pass

        def method2(self):
            pass

        attr3 = 3
        # attr3 = 3  # uncomment to trigger exception

    print(MyClass.member_order)
    # MyClass.attr4 = 4 # uncomment to trigger exception
```

It prints the attribute names in order:

```
['attr1', 'attr2', 'method1', 'method2', 'attr3']
```

Use `no_special_methods` and `no_callables` to customize what names should be in the list. Also, uncomment the second assignment to `attr3` and adding `attr4` after the class definition to see how our metaclass prevents these from happening.

2.10 Exercises

1. Create a metaclass that prevents using names with more than 30 characters for methods of a class.

2. Create a metaclass that makes a class require docstrings with an minimum length of 5 characters for all methods other than special methods, i.e. if the name matches the regular expression `^__[a-zA-Z\d]+__$`.
3. Use `autometa_python2.py` with Python 2 and `classwatcher.py` with Python 3 on some of your own modules or packages and some third-party libraries.