Operator Functions Set 1 & Set 2
Data Types
Introduction
Arrays Set 1, Set 2
String Methods Set 1, Set 2, Set 3
String Template Class & String Formatting using %
List Methods Set 1, Set 2, Set 3
Tuples & Sets
Dictionary Methods Set 1, Set 2
ChainMap
Explore More...
Control Flow
Loops and Control Statements
Counters & Accessing Counters
Iterators & Iterator Functions Set 1, Set 2
Generators
Explore More...
Functions
Function Decorators
Returning Multiple Values
Yield instead of Return
Python Closures & Coroutine
Explore More...
Modules
Introduction

Numeric Functions & Logarithmic and Power functions
Calender Functions Set 1, Set 2
Complex Numbers Introduction & Important functions
Explore More...
Object Oriented Concepts
Class, Object and Members
Data Hiding and Object Printing
Inheritance, Subclass and super
Class method vs static method & Class or Static Variables
Explore More...
Exception Handling
Exception Handling
User-Defined Exceptions
Built-in Exceptions
Libraries and Functions
Timeit
Numpy Set 1, Set 2
Get and Post
import module & reload module
Collection Modules Deque, Namedtuple & Heap
Explore More...
Machine Learning with Python
Classifying data using Support Vector Machines(SVMs) in Python

K means Clustering
How to get synonyms/antonyms from NLTK WordNet in Python?
Explore More...
Misc
Sql using Python & MongoDB and Python
Json formatting & Python Virtual environment
Metaprogramming with Metaclasses in Python
Python Input Methods for Competitive Programming
Explore More...
Applications and Projects
Creating a proxy webserver Set 1, Set 2
Send Messsage to FB friend
Twitter Sentiment Analysis & Whatsapp using Python
Desktop Notifier & Junk File Organizer
Explore More...

Metaprogramming with Metaclasses in Python

At first word **Metaprogramming** seems very funky and alien thing but if you have ever worked with decorators or metaclasses, your were doing metaprogramming there. **In nutshell we can say metaprogramming is the code which manipulates code.**

In this article we are going to discuss about **Metaclasses**, why and when we should use them and what are the alternatives. This is fairly advance Python topic and following prequisite is expected –

- OOP concept in Python
- Decorators in Python

Note: This article considers Python 3.3 and above

Metaclasses

In Python everything have some type associated with it. For example if we have a variable having integer value then it's type is int. You can get type of anything using **type()** function.

```
num = 23
print("Type of num is:", type(num))

lst = [1, 2, 4]
print("Type of lst is:", type(lst))

name = "Atul"
print("Type of name is:", type(name))
```

Run on IDE

Output:

```
Type of num is: <class 'int'>
Type of lst is: <class 'list'>
Type of name is: <class 'str'>
```

Every type in Python is defined by Class. So in above example, unlike C or Java where int, char, float are primary data types, in Python they are object of int class or str class. So we can make a new type by creating a class of that type. For example we can create a new type *Student* by creating *Student* class.

```
class Student:
    pass
stu_obj = Student()

# Print type of object of Student class
print("Type of stu_obj is:", type(stu_obj))
```

Run on IDE

Output:

```
Type of stu_obj is: <class '__main__.Student'>
```

A Class is also an object, and just like any other object it's a instance of something called **Metaclass**. A special class **type** creates these *Class* object. The **type** class is default **metaclass** which is responsible for making classes. For example in above example if we try to find out the type of *Student* class, it comes out to be a **type**.

```
class Student:
    pass

# Print type of Student class
print("Type of Student class is:", type(Student))
```

Run on

Output:

```
Type of Student class is: <class 'type'>
```

Because Classes are also an object, they can be modified in same way. We can add or subtract fields or methods in class in same way we did with other objects. For example –

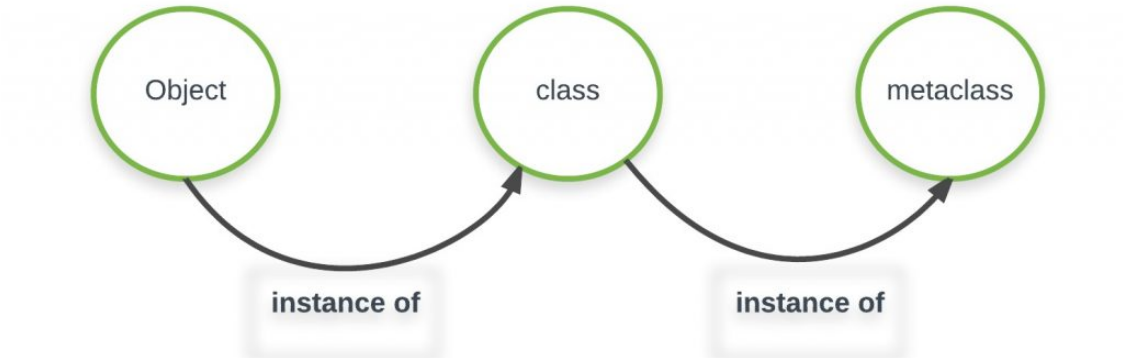
```
# Defined class without any
# class methods and variables
class test:pass

# Defining method variables
test.x = 45

# Defining class methods
test.foo = lambda self: print('Hello')

# creating object
myobj = test()

print(myobj.x)
myobj.foo()
```

Run on IDE

Output:

```
45
Hello
```

This whole meta thing can be summarized as – **Metaclass create Classes and Classes creates objects**



Metaclass is responsible for generation of classes, so we can write our own custom metaclasses to modify the way classes are generated by performing extra actions or injecting code. Usually we do not need custom metaclasses but sometime it's necessary.
There are problems for which metaclass and non-metaclass based solutions are available (often simpler) but in some cases only metaclass can solve the problem. We will discuss such problem in this article.

Creating custom Metaclass

To create our custom metaclass, our custom metaclass have to inherit **type** metaclass and usually override –

- **__new__():** It's a method which is called before __init__(). It creates the object and return it. We can overide this method to control how the objects are created.
- **__init__():** This method just initialize the created object passed as parameter

We can create classes using **type()** function directly. It can be called in following ways –

1. When called with only one argument, it returns the type. We have seen it before in above examples.
2. When called with three parameters, it creates a class. Following arguments are passed to it –
    1. Class name
    2. Tuple having base classes inherited by class
    3. **Class Dictionary:** It serves as local namespace for the class, populated with class methods and variables

Consider this example –

```
def test_method(self):
    print("This is Test class method!")

# creating a base class
class Base:
    def myfun(self):
        print("This is inherited method!")

# Creating Test class dynamically using
# type() method directly
Test = type('Test', (Base, ), dict(x="atul", my_method=test_method))

# Print type of Test
print("Type of Test class: ", type(Test))

# Creating instance of Test class
test_obj = Test()
print("Type of test_obj: ", type(test_obj))

# calling inherited method
test_obj.myfun()

# calling Test class method
test_obj.my_method()

# printing variable
print(test_obj.x)
```

Run on IDE

Output:

```
Type of Test class:  <class 'type'>
Type of test_obj:  <class '__main__.Test'>
This is inherited method!
This is Test class method!
atul
```

Now let's create a metaclass without using **type()** directly. In the following example we will be creating a

metaclass ***MultiBases*** which will check if class being created have inherited from more than one base classes. If so, it will raise an error.

```
# our metaclass
class MultiBases(type):
    # overriding __new__ method
    def __new__(cls, clsname, bases, clsdict):
        # if no of base classes is greator than 1
        # raise error
        if len(bases)>1:
            raise TypeError("Inherited multiple base classes!!!")

        # else execute __new__ method of super class, ie.
        # call __init__ of type class
        return super().__new__(cls, clsname, bases, clsdict)

# metaclass can be specified by 'metaclass' keyword argument
# now MultiBase class is used for creating classes
# this will be propagated to all subclasses of Base
class Base(metaclass=MultiBases):
    pass

# no error is raised
class A(Base):
    pass

# no error is raised
class B(Base):
    pass

# This will raise an error!
class C(A, B):
    pass
```

Run on IDE

Output:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 8, in __new__
TypeError: Inherited multiple base classes!!!
```

Solving problem with metaclass

There are some problems which can be solved by decorators (easily) as well as by metaclasses. But there are few problems whose result can only be achived by metaclasses. For example consider a very simple problem of code repetition.

We want to debug class methods, what we want is that whenever class method executes, it should print it's fully qualified name before executing it's body.

Very first solution that comes in our mind is using **method decorators**, following is the sample code –

```
from functools import wraps

def debug(func):
    '''decorator for debugging passed function'''

    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Full name of this method:", func.__qualname__)
```

```
        return func(*args, **kwargs)
    return wrapper

def debugmethods(cls):
    '''class decorator make use of debug decorator
       to debug class methods '''

    # check in class dictionary for any callable(method)
    # if exist, replace it with debugged version
    for key, val in vars(cls).items():
        if callable(val):
            setattr(cls, key, debug(val))
    return cls

# sample class
@debugmethods
class Calc:
    def add(self, x, y):
        return x+y
    def mul(self, x, y):
        return x*y
    def div(self, x, y):
        return x/y

mycal = Calc()
print(mycal.add(2, 3))
print(mycal.mul(5, 2))
```

Run on IDE

Output:

```
Full name of this method: Calc.add
5
Full name of this method: Calc.mul
10
```

This solution works fine but there is one problem, what if we want to apply this method decorator to all sub-classes which inherit this ***Calc*** class. In that case we have to separately apply method decorator to every sub-class just like we did with ***Calc*** class.

The problem is if we have many such subclasses, then in that case we won't like adding decorator to each one separately. If we know beforehand that every subclass must have this debug property, then we should look up to metaclass based solution.

Have a look at this **metaclass based solution**, the idea is that classes will be created normally and then immediately wrapped up by debug method decorator –

```
from functools import wraps

def debug(func):
    '''decorator for debugging passed function'''

    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Full name of this method:", func.__qualname__)
        return func(*args, **kwargs)
    return wrapper

def debugmethods(cls):
    '''class decorator make use of debug decorator
       to debug class methods '''
```

```
    for key, val in vars(cls).items():
        if callable(val):
            setattr(cls, key, debug(val))
    return cls

class debugMeta(type):
    '''meta class which feed created class object
       to debugmethod to get debug functionality
       enabled objects'''

    def __new__(cls, clsname, bases, clsdict):
        obj = super().__new__(cls, clsname, bases, clsdict)
        obj = debugmethods(obj)
        return obj

# base class with metaclass 'debugMeta'
# now all the subclass of this
# will have debugging applied
class Base(metaclass=debugMeta):pass

# inheriting Base
class Calc(Base):
    def add(self, x, y):
        return x+y

# inheriting Calc
class Calc_adv(Calc):
    def mul(self, x, y):
        return x*y

# Now Calc_adv object showing
# debugging behaviour
mycal = Calc_adv()
print(mycal.mul(2, 3))
```

Run on IDE

Output:

```
Full name of this method: Calc_adv.mul
6
```

When to use Metaclasses

Most of the time we are not using metaclasses, they are like black magic and usually for something complicated, but few cases where we use metaclasses are –

- As we have seen in above example, metaclasses propogate down the inheritence hierarchies. It will affect all the subclasses as well. If we have such situation, then we should use metaclasses.
- If we want to change class automatically, when it is created
- If you are API developer, you might use metaclasses

As quoted by Tim Peters

> *Metaclasses are deeper magic that 99% of users should never worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).*

References

- http://www.dabeaz.com/py3meta/Py3Meta.pdf
- https://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python

This article is contributed by **Atul Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Python

Recommended Posts:

Function Decorators in Python | Set 1 (Introduction)
Python Virtual Environment | Introduction
Graph Plotting in Python | Set 1
Bloom Filters – Introduction and Python Implementation
Convert Text to Speech in Python

(Login to Rate and Mark)

0 Average Difficulty : **0/5.0**
No votes yet.

Add to TODO List
Mark as DONE

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

Share this post!

 Contact Us! About Us! Careers! Privacy Policy