



## Metaclasses

Walter Cazzola

### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

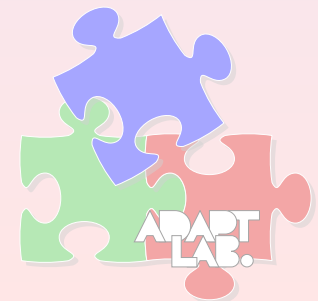
### References

# Metaclasses

## How to Silently Extend Classes (Part 3)

Walter Cazzola

Dipartimento di Informatica  
Università degli Studi di Milano  
e-mail: [cazzola@di.unimi.it](mailto:cazzola@di.unimi.it)  
twitter: [@w\\_cazzola](https://twitter.com/w_cazzola)





# Metaclasses

## What's a Metaclass?

### Metaclasses

Walter Cazzola

#### Metaclasses

##### Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

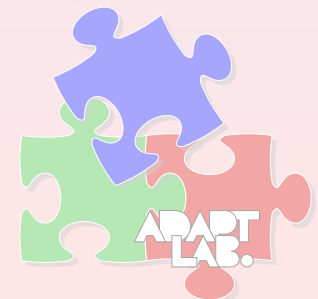
#### References

Metaclasses are a mechanism to gain a **high-level of control** over how a set of classes works.

- They permit to intercept and augment class creation;
- they provide an API to insert extra-logic at the end of **class** statement;
- they provide a general protocol to manage class objects in a program.

Note,

- the added logic does not rebind the class name to a decorator callable, but rather routes creation of the class itself to specialized logic.
- metaclasses add code to be run at class creation time and not at instance creation time





# Metaclasses

## The Metaclass Model

### Metaclasses

Walter Cazzola

#### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

#### References

## Classes Are Instances of **type**

```
[11:44]cazzola@hymir:~/esercizi-pa>python3
>>> from circle import *
>>> type(circle)
<class 'type'>
>>> circle.__class__
<class 'type'>
>>> c = circle(3)
>>> type(c)
<class 'circle.circle'>
>>> c.__class__
<class 'circle.circle'>
>>> type([])
<class 'list'>
>>> type(type([]))
<class 'type'>
```

## Metaclasses Are Subclasses of **type**

- **type** is a class that generates user-defined classes.
- Metaclasses are subclasses of the **type** class.
- Class objects are instances of the **type** class, or a subclass thereof.
- Instance objects are generated from a class.

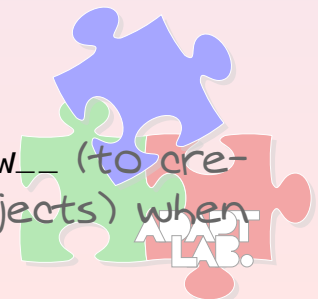
## Class Statement Protocol

- at the end of class statement, after filling `__dict__`, python calls

```
class = type(classname, superclasses, attributedict)
```

to create the **class** object.

- **type** object defines a `__call__` operator that calls `__new__` (to create class objects) and `__init__` (to create instance objects) when **type** object is called





# Metaclasses

## The Metaclass Declaring & Coding

### Metaclasses

Walter Cazzola

#### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

#### References

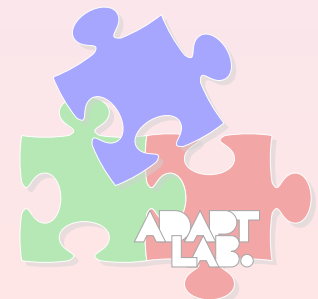
## Declaring Metaclasses

To create a class with a custom metaclass you have just to list the desired metaclass as a keyword argument in the **class** header.

```
class Spam(metaclass=Meta): pass
```

## Coding Metaclasses

- subtype **type**
- override `__new__`, `__init__` and `__call__` operators





# Metaclasses

## The Metaclass Declaring & Coding (Cont'd)

### Metaclasses

Walter Cazzola

#### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

#### References

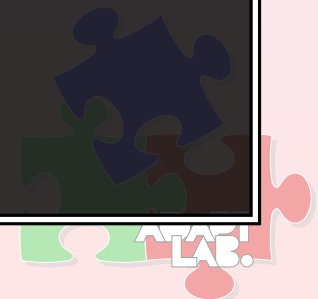
```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('In MetaOne.new: ', classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)
    def __init__(Class, classname, supers, classdict):
        print('In MetaOne init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Eggs: pass
print('making class')

class Spam(Eggs, metaclass=MetaOne):    # Inherits from Eggs, instance of Meta
    data = 1                            # Class data attribute
    def meth(self, arg): pass           # Class method attribute

print('making instance')
X = Spam()
print('data:', X.data)
```

```
[17:13]cazzola@hymir:~/esercizi-pa/metaclass>python3 metaone.py
making class
In MetaOne.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0xb79d99ac>}
In MetaOne init:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0xb79d99ac>}
...init class object: ['__module__', 'data', 'meth', '__doc__']
making instance
data: 1
```





## Metaclasses

Walter Cazzola

### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

### References

# Metaclasses

## Metaclasses vs Superclasses

In spite of the syntax meta- and superclasses are quite different

- Metaclasses inherit from the **type** class
- Metaclass declarations are inherited by subclasses
- Metaclass attributes are not inherited by class instances

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('In MetaOne.new:', classname)
        return type.__new__(meta, classname, supers, classdict)
    def toast(self):
        print('toast')

class Super(metaclass=MetaOne):
    def spam(self):
        print('spam')

class C(Super):
    def eggs(self):
        print('eggs')

X = C()
X.eggs()      # Defined in C
X.spam()      # Inherited from Super
X.toast()     # Not inherited from metaclass
```

```
[17:29]cazzola@hymir:~/esercizi-pa/metaclass>python3 MetaAndSuper.py
In MetaOne.new: Super
In MetaOne.new: C
eggs
spam
Traceback (most recent call last):
  File "MetaAndSuper.py", line 16, in <module>
    X.toast()
AttributeError: 'C' object has no attribute 'toast'
```



# Metaclasses

## Metaclass-Based Augmentation

### Metaclasses

Walter Cazzola

#### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

#### References

```
def eggsfunc(obj): return obj.value * 4
def hamfunc(obj, value): return value + 'ham'

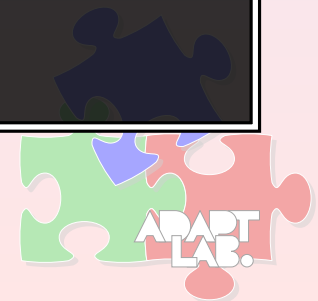
class Extender(type):
    def __new__(meta, classname, supers, classdict):
        classdict['eggs'] = eggsfunc
        classdict['ham'] = hamfunc
        return type.__new__(meta, classname, supers, classdict)

class Client1(metaclass=Extender):
    def __init__(self, value): self.value = value
    def spam(self): return self.value * 2

class Client2(metaclass=Extender): value = 'ni?'

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bacon'))
Y = Client2()
print(Y.eggs())
print(Y.ham('bacon'))
```

```
[18:01]cazzola@hymir:~/esercizi-pa/metaclass>python3 extender.py
Ni!Ni!
Ni!Ni!Ni!Ni!
baconham
ni?ni?ni?ni?
baconham
```





# Metaclasses

## Applying Decorators to Methods: The Decorators!

### Metaclasses

Walter Cazzola

#### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

#### References

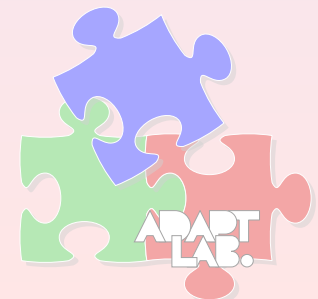
```
[21:18]cazzola@hymir:~/aux_work/projects/python/esercizi-pa/metaclass/decorators>ls
__init__.py timer.py tracer.py
```

```
# timer.py
import time

def timer(label='', trace=True):
    def onDecorator(func):
        def onCall(*args, **kwargs):
            start = time.clock()
            result = func(*args, **kwargs)
            elapsed = time.clock() - start
            onCall.alltime += elapsed
            print('{0}{1}: {2:.5f}, {3:.5f}'.format(
                label, func.__name__, elapsed, onCall.alltime))
        return result
    onCall.alltime = 0
    return onCall
return onDecorator
```

```
# tracer.py

def tracer(func):
    calls = 0
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('call {0} to {1}'.\
            format(calls, func.__name__))
        return func(*args, **kwargs)
    return onCall
```







# Metaclasses

## Applying Decorators to Methods: The Decoration!

### Metaclasses

Walter Cazzola

#### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

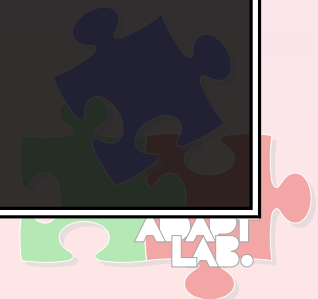
#### References

```
from decorators.tracer import tracer

class Person:
    @tracer
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    @tracer
    def giveRaise(self, percent):      # giveRaise = tracer(giveRaise)
        self.pay *= (1.0 + percent)  # onCall remembers giveRaise
    @tracer
    def lastName(self):               # lastName = tracer(lastName)
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)                  # Runs onCall(sue, .10)
print(sue.pay)
print(bob.lastName(), sue.lastName()) # Runs onCall(bob), remembers lastName
```

```
[21:30]cazzola@hymir:~/esercizi-pa/metaclass>python3 Person1.py
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```





# Metaclasses

## Applying Decorators to Methods: Through a Metaclass!

### Metaclasses

Walter Cazzola

#### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

#### References

```
from types import FunctionType
from decorators.tracer import tracer

class MetaTrace(type):
    def __new__(meta, classname, supers, classdict):
        for attr, attrval in classdict.items():
            if type(attrval) is FunctionType:
                classdict[attr] = tracer(attrval)
        return type.__new__(meta, classname, supers, classdict)

class Person(metaclass=MetaTrace):
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self): return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())
```

```
[21:45]cazzola@hymir:~/esercizi-pa/metaclass>python3 Person2.py
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```



# Metaclasses

## Applying Decorators to Methods

### Metaclasses

Walter Cazzola

#### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

#### References

```
from types import FunctionType
from decorators.tracer import tracer
from decorators.timer import timer
```

```
def decorateAll(decorator):
    class MetaDecorate(type):
        def __new__(meta, classname, supers, classdict):
            for attr, attrval in classdict.items():
                if type(attrval) is FunctionType:
                    classdict[attr] = decorator(attrval)
            return
    type.__new__(meta, classname, supers, classdict)
    return MetaDecorate
```

```
class Person(metaclass=decorateAll(tracer)):
    ...
```

```
print('--- tracer')
bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
```

```
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())
```

```
class Person(
    metaclass=decorateAll(timer(label='**'))):
    ...
```

```
print('--- timer')
bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())
print('{0:.5f}'.format(Person.__init__.alltime))
print('{0:.5f}'.format(Person.giveRaise.alltime))
print('{0:.5f}'.format(Person.lastName.alltime))
```

```
[21:47]cazzola@hymir:~/esercizi-pa/metaclass>python3 Person3.py
```

```
--- tracer
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```

```
--- timer
**__init__: 0.00000, 0.00000
**__init__: 0.00000, 0.00000
Bob Smith Sue Jones
**giveRaise: 0.00000, 0.00000
110000.0
**lastName: 0.00000, 0.00000
**lastName: 0.00000, 0.00000
Smith Jones
0.00000
0.00000
0.00000
```



# References

## Metaclasses

Walter Cazzola

### Metaclasses

Definition

metaclass model

metaclass coding

meta vs super

metaclass-based  
augmentation

metaclasses at  
work

### References

- ▶ Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.  
**Practical Programming: An Introduction to Computer Science Using Python.**  
The Pragmatic Bookshelf, second edition, 2009.
- ▶ Mark Lutz.  
**Learning Python.**  
O'Reilly, fourth edition, November 2009.
- ▶ Mark Pilgrim.  
**Dive into Python 3.**  
Apress\*, 2009.
- ▶ Mark Summerfield.  
**Programming in Python 3: A Complete Introduction to the Python Language.**  
Addison-Wesley, October 2009.

