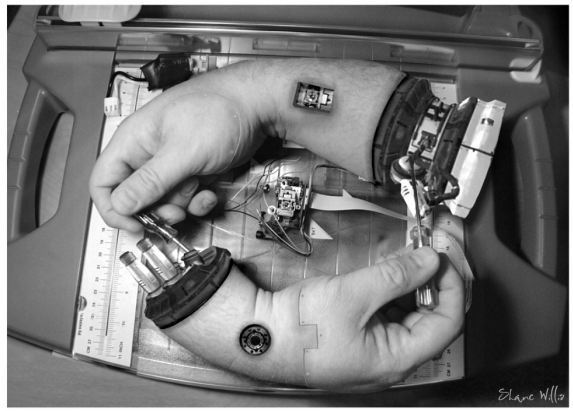
Metaprogramming in Python



Make software great again!

Giulio Angiani - UniPr

Metaprogramming

- **Metaprogramming** is a programming technique in which computer programs have the ability to treat programs as their data
- What does it mean?:
  - a program can be designed to read, generate, analyse or transform other programs, and even modify itself while running
- the language in which the metaprogram is written is called the *metalanguage*
- the ability of a programming language to be its own metalanguage is called *reflection*

Wikipedia

Examples of metaprogramming

- Generative programming (code generation)
  - *Quine* (self-generating program)

```
>>>s = 's = %r\nprint(s%%s)'
>>>print(s%s)

s = 's = %r\nprint(s%%s)'
print(s%s)
```

PYTHON

Examples of metaprogramming

- Generative programming
  - run time code generation or evalutation

PYTHON

```
>>> exec("""d = {'code': 'run time created!'}""")
>>> d['code']
'run time created!'

>>> s = "{'code': 'evaluated'}"
>>> d = eval(s)
>>> d['code']
'evaluated'
```

Examples of metaprogramming

- other language generation program

PYTHON

```
s = "<?php \n"
s +="   $x = 0; \n"
for i in range(3):
    s+= "   $x = $x + %s; \n" % str(i+1)
s += "  print($x);\n ?>"
open('dummy.php', 'w').write(s)
```

PHP

```
<?php
   $x = 0;
   $x = $x + 1;
   $x = $x + 2;
   $x = $x + 3;
  print($x);
 ?>
```

Generators

- programs to generate code (for example classes)

PYTHON

```
class_name = "User"
class_code = """

class %(class_name)s:

    def __init__(self, id=''):
        self.__id = id

    def __repr__(self):
        return "I'm a %(class_name)s instance; my id is %%s " %% self.__id
""" % vars()

print(class_code)
```

Generators

- generated code

PYTHON

```
class User:

    def __init__(self, id=''):
        self.__id = id

    def __repr__(self):
        return "I'm a User instance; my id is %s " % self.__id
```

But MetaProgramming is much more...

Reflection

Self-modifying code

Meta-classes

Decorators

Simply doing things with code...

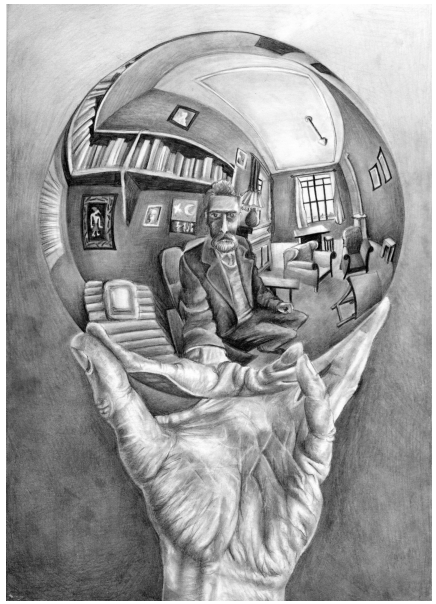
Reflection

Reflection

Def: the ability of a computer program

to ***examine***, ***inspect***, and ***modify***

its own structure and behaviours at ***runtime***



How are we made?

- know type of an element
- know methods of a class
- know info of an object

PYTHON

```
>>> x = 1
>>> x.__class__
<class 'int'>
>>> x.__class__.__name__
'int'
>>> x.__class__.__class__
<class 'type'>
>>> x.__class__.__class__.__class__
<class 'type'>
>>> x.__class__.__class__.__class__.__class__
<class 'type'>
>>> ....
```

Class inspection

PYTHON

```
class Dummy:
    def __init__(self,x):
        self.__x = x

    def getX(self):
        return self.__x

    def setX(self, x):
        self.__x = x
```

Class inspection

```
Instance class   =  <class '__main__.Dummy'>
Instance content =  {'_Dummy__x': 5}

__module__ __main__
__weakref__ <attribute '__weakref__' of 'Dummy' objects>
setX <function Dummy.setX at 0x7fcf4f373400>
__doc__ DOCSTRING::: Dummy class. It manages just a variable
__init__ <function Dummy.__init__ at 0x7fcf4f3730d0>
__dict__ <attribute '__dict__' of 'Dummy' objects>
getX <function Dummy.getX at 0x7fcf4f373378>

Class methods (all that contains __call__ method)
Method :  <function Dummy.setX at 0x7fcf4f373400>
Method :  <function Dummy.__init__ at 0x7fcf4f3730d0>
Method :  <function Dummy.getX at 0x7fcf4f373378>
```

Self-modifying code

Adding methods dinamically to an instance

PYTHON

```
>> p = Dummy(5)
>> g = Dummy(10)
>> print ("p.getX() = ", p.getX())
p.getX() =  5
>> print ("g.getX() = ", g.getX())
g.getX() =  10
>> def incX(self):
      self.x +=1

>> p.incX = MethodType(incX, p)
p.incX()
>> print ("p.getX() = ", p.getX())
p.getX() =  6
>> print ("g.getX() = ", g.getX())
g.getX() =  10
```

Adding methods dinamically to a class

PYTHON

```
# add method to a class
>> p = Dummy(5)
>> g = Dummy(10)
>> def incX(self):
       self.x +=1

>> Dummy.incX = incX
>> p.incX()
>> g.incX()
>> print ("p.getX() = ", p.getX())
p.getX() =  6
>> print ("g.getX() = ", g.getX())
g.getX() =  11
```

So, I will use private attributes only!

PYTHON

```
d = Dummy(10)
print("d.getY() Before...", d.getY())
print("d.__dict__ : ", d.__dict__)
d.__dict__["_Dummy__y"] = 20
print("d.getY() After...", d.getY())
```

OUTPUT

```
('d.getY() Before...', 0)
('d.__dict__ : ', {'x': 10, '_Dummy__y': 0})
('d.getY() After...', 20)
```

Decorators

Metaprogramming with decorators

A **decorator** is a function that creates a wrapper around another function

The wrapper is a **new function** that works exactly like the original function (same arguments, same return value) except that *some kind of extra processing is carried out*

Decorators

We illustrate basics with a simple problem of debugging

We have a simple function... and the same function with debugging...

```python
def add(x,y):
    return x+y

def add(x,y):
    print ('Add')
    return x+y
```

Uhm... it looks like the only way to debug....

Decorators

Many functions with debug...

```
def add(x,y):
    print ('Add')
    return x+y

def sub(x,y):
    print ('Sub')
    return x-y

def mult(x,y):
    print ('Mul')
    return x*y

def div(x,y):
    print ('Div')
    return x/y
```

A debugging decorator

We write a function that gets another function as argument, use it for debugging and then return it

PYTHON

```
from functools import wraps
def debug(func):
    msg = func.__qualname__
    # useful for easy-readable traceback
    # or using the same docstring....
    @wraps(func)    ## mandatory or weird things happen!
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper
```

- it's the same of using

PYTHON

```
func = debug(func)
```

Warning!

- When you use a decorator, you're replacing one function with another. In other words, if you have a decorator

```python
def logged(func):
    def with_logging(*args, **kwargs):
        print(func.__name__ + " was called")
        return func(*args, **kwargs)
    return with_logging
```

- then when you say

```python
@logged
def f(x):
   """does some math"""
   return x + x * x
```

Warning!

- it's exactly the same as saying

```python
def f(x):
    """does some math"""
    return x + x * x
f = logged(f)
```

- and your function f is replaced with the function with_logging. Unfortunately, this means that if you then say

```python
print(f.__name__)
```

- you will get **with_logging** instead of **f**

A debugging decorator

A decorator creates a "wrapper" function around the provided function **func**

@wraps copies metadata

- name and doc string and function attributes

PYTHON

```
from functools import wraps
def debug(func):
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
return wrapper
```

ex: 00_decorator.py

Decorator syntax

The definition of a function and wrapping almost always occur together

```python
@debug
def add(x,y):
    return x+y
```

- is the same of

```python
def add(x,y):
    return x+y
add = debug(add)
```

ex: 00_1_decorator_syntax.py

Variation: Decorator with arguments

For example: debug with prefixes

```python
@decorator(args)
def func():
    pass
```

- is evaluated as

```python
func = decorator(args)(func)
```

Variation: Decorator with arguments

For example: debug with prefixes

PYTHON

```
def debug(prefix=''):
    def decorate(func):
        msg = prefix + " ::: " + func.__qualname__
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(msg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

@debug(prefix='ADD')
def add(x,y):
    return x+y
```

ex: 00_2_decorator_with_args.py

Decore all methods at once - Class decorators

One decorator application

- Covers all definitions within the class
- It even mostly works.

PYTHON

```
@classdecorator
class Spam:
    def grok(self):
        pass
    def bar(self):
        pass
    def foo(self):
        pass
```

ex: 03_decore_all_methods_of_a_class.py

Class decorator

- Walk through class dictionary
- Identify callables (e.g., methods)
- Wrap with a decorator

PYTHON

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

ex: 03_decore_all_methods_of_a_class.py

Decoring all classes

What if I have more classes ?


```
@debugmethods
class Base:
    pass

@debugmethods
class First(Base):
    pass

@debugmethods
class Second:
    pass

@debugmethods
class Third:
    pass
```

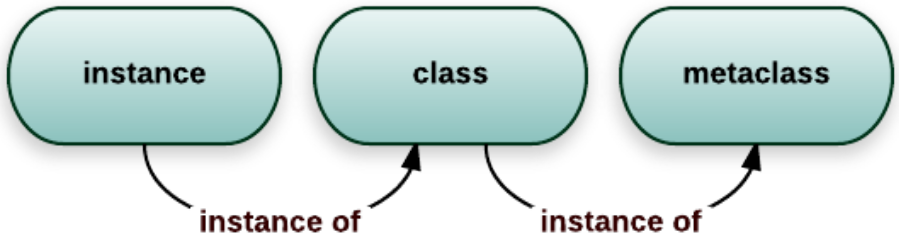
title: Which solution ?

Solution: A Metaclass

What is a metaclass?

A metaclass is the class of a class.
Like a class defines how an instance of the class behaves, a metaclass defines how a class behaves.

A class is an instance of a metaclass.



Understand type of something

```
>>> x = 1
>>> type(x)
<class 'int'>
>>> type(x.__class__)
<class 'type'>
```

PYTHON

- and for a generic class?

```
>>> d = Dummy(10)
>>> type(d)
<class '__main__.Dummy'>
>>> type(d.__class__)
<class 'type'>
```

PYTHON

Type is the default metaclass

"type" is itself a class and it is its own type

A metaclass is most commonly used as a class-factory.

Like you create an instance of the class by calling the class,
Python creates a **new class** by calling the **metaclass**.

We can do 'extra things' when creating a class!

Creating Types

PYTHON

```
class Base:
    pass

class Example(Base):
    def __init__(name):
        self.name = name
    def whoami():
        return "I'm " + self.name


>> print ("Base class = ", Base.__class__)
('Base class = ', <type 'type'>)
>> print ("Example class = ", Example.__class__)
('Example class = ', <type 'type'>)
```

Class deconstructing

Consider the class **Example**:

PYTHON

```
class Example(Base):
    def __init__(name):
        self.name = name
    def whoami():
        return "I'm " + self.name
```

What are its components?

```
- Name ("Example")
- Base classes (Base,)
- Functions (__init__,whoami)
```

ex: 02_create_a_class_runtime.py

Class definition process

What happens during class definition?

- Step1: Body of class is isolated

```
body= """
    def __init__(self, name):
        self.name = name
    def whoami(self):
        return "I'm a Example object"
"""
```
PYTHON

- Step 2: The class dictionary is created

```
clsdict = type.__prepare__('Example', (Base,))
```
PYTHON

- This dictionary serves as local namespace for statements in the class body. By default, it's a simple dictionary (more later)

Class definition process

What happens during class definition?

- Step3: Body is executed in returned dict

```python
exec(body, globals(), clsdict)
```

PYTHON

- Afterwards, clsdict is populated

```
>> clsdict
{'whoami': <function whoami at 0x7f4cd58a2598>, '__init__': <function __init__ at 0x7f4cd77dae18>}
>> ne = Example("myname")
>> ne.whoami()
I'm a Example object
```

PYTHON

ex: 02_create_a_class_runtime.py

Changing the metaclass

- **metaclass** keyword argument
- Sets the class used for creating the type
- By default, it's set to 'type', but you can change it to something else

PYTHON

```
class Example(Base, metaclass=type):
    def __init__(name):
        self.name = name
    def whoami():
        return "I'm " + self.name
```

- To **define** a new metaclass you typically inherit from type and redefine **__new__** or **__init__**

PYTHON

```
class mytype(type):
    def __new__(cls, name, bases, clsdict):
        clsobj = super().__new__(cls, name, bases, clsdict)
        return clsobj
```

Using a metaclass

Metaclasses get information about class definitions at the time of definition

- Can inspect this data
- Can modify this data

Question: Why would you use one?

- Metaclasses propagate down hierarchies
- Think of it as a genetic mutation

Using metaclass for decoring all classes

Class gets created normally
Immediately wrapped by class decorator

PYTHON

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

class debugmeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsobj = super().__new__(cls, clsname, bases, clsdict)
        clsobj = debugmethods(clsobj)
    return clsobj
```

ex: 07_decore_all_classes_in_a_module_using_metaclass.py

Standard boring operations...

PYTHON

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

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

class Host:
    def __init__(self, address, port):
        self.address = address
        self.port = port
```

Inheritance and MP solution

PYTHON

```
class Structure:
    _fields = []
    def __init__(self, *args):
        for name, val in zip(self._fields, args):
            setattr(self, name, val)

class Stock(Structure):
    _fields = ['name', 'shares', 'price']

class Point(Structure):
    _fields = ['x', 'y']

class Host(Structure):
    _fields = ['address', 'port']
```

setter and getter

```
class P:
    def __init__(self,x):
        self.__x = x
    def getX(self):
        return self.__x
    def setX(self, x):
        self.__x = x

p1 = P(5)
p2 = P(10)
p1.setX(p1.getX() + p2.getX())
```

p1.x = p1.x + p2.x

setter and getter by decorators

PYTHON

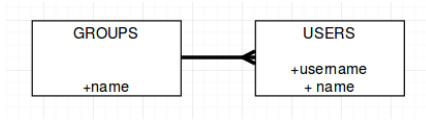
```
class P:
    def __init__(self,x):
        self.__x = x

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        self.__x = x
```

A real use of metaclasses: Django

Django is a MVC framework in python



Model layer
- maps DB tables into classes
- maps DB relations into classes-relations

example:

PYTHON

```python
class Group(models.Model):
    name = models.CharField(max_length=200)

class User(models.Model):
    username = models.CharField(max_length=50)
    name = models.CharField(max_length=200)
    group = models.ForeignKey(Group, related_name='group_users')
```

A real use of metaclasses: Django

Uses **ModelBase** as metaclass for all models

PYTHON

```
class ModelBase(type):
    """
    Metaclass for all models.
    """
    def __new__(cls, name, bases, attrs):
        super_new = super(ModelBase, cls).__new__

[+++ code +++]

    # Add all attributes to the class.
    for obj_name, obj in attrs.items():
        new_class.add_to_class(obj_name, obj)

[+++ code +++]
```

A real use of metaclasses: Django

Uses **ModelBase** as metaclass for all models

PYTHON

```
def add_to_class(cls, name, value):
    if hasattr(value, 'contribute_to_class'):
        value.contribute_to_class(cls, name)
    else:
        setattr(cls, name, value)
```

Another example: type checking in python

```python
def addition(number, other_number):
    return number + other_number
```

PYTHON

vs

```python
def addition(number: int, other_number: int) -> int:
    return number + other_number
```

PYTHON

but...

Another example: type checking in python

PYTHON

```
def addition(number, other_number):
    return number + other_number
```

result...

INTERACTIVE SHELL

```
>>> print(addition(1,2))
3
>>> print(addition('1','2'))
12
>>> print(addition(1,'2'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in addition
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Another example: type checking in python

PYTHON

```
def addition(number: int, other_number: int) -> int:
    return number + other_number
```

result is the same...

INTERACTIVE SHELL

```
>>> print(addition(1,2))
3
>>> print(addition('1','2'))
12
>>> print(addition(1,'2'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in addition
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Solution: decorator!

PYTHON

```
def validate_input(obj, **kwargs):
    hints = get_type_hints(obj) # iterate all type hints
    for attr_name, attr_type in hints.items():
        if attr_name == 'return':
            continue
        if not isinstance(kwargs[attr_name], attr_type):
            raise TypeError('TYPE CHECKING ERROR : Argument %r is not of type %s' % (attr_name, attr

def type_check(decorator):
    @wraps(decorator)
    def wrapped_decorator(*args, **kwargs):
        # translate *args into **kwargs
        func_args = getfullargspec(decorator)[0]
        kwargs.update(dict(zip(func_args, args)))
        validate_input(decorator, **kwargs)
        return decorator(**kwargs)
    return wrapped_decorator
```

Solution: decorator!

INTERACTIVE SHELL

```
>>> print(addition(1,2))
3
>>> print(addition('1','2'))
Traceback (most recent call last):
  File "00_type_checking.py", line 43, in <module>
    print(addition('1', '2'))
  File "00_type_checking.py", line 29, in wrapped_decorator
    validate_input(func, **kwargs)
  File "00_type_checking.py", line 17, in validate_input
    'TYPE CHECKING ERROR : Argument %r is not of type %s' % (attr_name, attr_type)
TypeError: TYPE CHECKING ERROR : Argument 'other_number' is not of type <class 'int'>
```

Another example: web page protection

PYTHON+FLASK

```
from functools import wraps
def protected_by_login(fnz):
    @wraps(fnz)
    def f(environ, *args, **kwargs):
        msg = "protected_by_login"
        if not environ.get('logged_user', None):
            print("User not logged!")
            return notavailable(environ)
        else:
            return fnz(environ, *args, **kwargs)
    return f

@protected_by_login
def home(environ):
### Use a decorator or you must insert this code in esch function...
#   if not environ.get('logged_user', None):
#       print("User not logged")
#       return notavailable(environ)
    return "User Dashboard - Protected!"""
```

Thanks for your attention!

Questions??

Giulio Angiani
Universita' degli Studi di Parma