why-metaclasses

September 11, 2017

Photo by Mike Lewinski / CC BY 2.0

```
In [5]: # TODO: find way to HIDE (but run) this block.
        # Stubs so the examples works
        class DeathRay(object):
            def __init__(self, *args, **kwargs): pass
            def vaporize(self, *args, **kwargs): pass
        class TimeMachine(object):
            def go(self, *args, **kwargs):
                pass
```

1 The Cyborg class

```
In [ ]: import time

        class Cyborg(object):

            def __init__(self, name):
                self.name = name
                self.weapon = DeathRay(ammunition=25)
                self.teleporter = TimeMachine()

            def travel(self, destination, year):
                self.teleporter.go(destination, year)
                time.sleep(0.25)  # not instant, but almost

            def attack(self, target):
                self.weapon.vaporize(target)
```

This is our class, and we want to log the calls to *every* method.

2 Attempt 1: print() calls everywhere

No point in denying it: this is what many of us *would* do if not supervised by an adult.

```
In [ ]: import time
```

```
        class Cyborg(object):

            def __init__(self, name):
                print("Creating new Cyborg with name '{}'".format(name))
                self.name = name
                self.weapon = DeathRay(ammunition=25)
                self.teleporter = TimeMachine()

            def travel(self, destination, year):
                print("Travelling to {} and year {}".format(destination, year))
                self.teleporter.go(destination, year)
                time.sleep(0.25)  # not instant, but almost

            def attack(self, target):
                print("Attacking {}".format(target))
                self.weapon.vaporize(target)
```

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements" Brian W. Kernighan.

```
In [ ]: robot = Cyborg('T-1000')
        robot.travel('Los Angeles', 1995)
        robot.attack('Sarah Connor')
        robot.attack('John Connor')
```

3 Attempt 2: the `logging` module

```
In [1]: import time
        import logging
        logging.basicConfig(format='%(asctime)s %(message)s', level=logging.DEBUG)

        class Cyborg(object):

            def __init__(self, name):
                logging.info("Creating new Cyborg with name '%s'", name)
                self.name = name
                self.weapon = DeathRay(ammunition=25)
                self.teleporter = TimeMachine()

            def travel(self, destination, year):
                logging.info("Travelling to %s and year %s", destination, year)
                self.teleporter.go(destination, year)
                time.sleep(0.25)  # not instant, but almost

            def attack(self, target):
                logging.info("Attacking %s", target)
                self.weapon.vaporize(target)
```

```
In [ ]: robot = Cyborg('T-1000')
        robot.travel('Los Angeles', 1995)
        robot.attack('Sarah Connor')
        robot.attack('John Connor')
```

4 Problem: there're logging calls everywhere

- We must add manually the call to e.g. `logging.info()` to every method.
- Whoever adds a new method in the future might forget to do it.
- What if we had three *hundred* methods, instead of three?
- Also, copy-pasting function calls is boring.

```
In [ ]: import time
        import logging
        logging.basicConfig(format='%(asctime)s %(message)s', level=logging.DEBUG)

        class Cyborg(object):

            def __init__(self, name):
                logging.info("Creating new Cyborg with name '%s'", name)
                self.name = name
                self.weapon = DeathRay(ammunition=25)
                self.teleporter = TimeMachine()
                self.alive = True

            def travel(self, destination, year):
                logging.info("Travelling to %s and year %s", destination, year)
                self.teleporter.go(destination, year)
                time.sleep(0.25)  # not instant, but almost

            def attack(self, target):
                logging.info("Attacking %s", target)
                self.weapon.vaporize(target)

            def selfdestroy(self):
                self.alive = False
```

We added `Cyborg.selfdestroy()`, but forgot to add the logging call.

5 Detour: Decorators Crash Course

What are decorators and what are they useful for?

Photo by Kamil Porembiski / CC BY-SA 2.0

6 First-class objects

> "A first class object is an entity that can be dynamically created, destroyed, passed to a function, returned as a value, and have all the rights as other variables in the programming language have" --- StackOverflow

- Functions are first-class objects.
- Classes are first-class objects too.
- In fact, in Python *everything* is a first-class object.

This mean that if we take e.g. a function, we can...

6.1 Assign it to a variable

Take a function, for example. We can assign it to a variable like any other object.

```
In [ ]: numbers = [3, 5, 7, 1]
        print(max(numbers))

In [ ]: func = max
        print(func(numbers))  # same as calling max()
```

No parentheses because we are not calling the function.

6.2 Pass it as argument

... *to* another function:

```
In [ ]: def call(func, values):
            return func(values)

        print(call(max, numbers))
        print(call(min, numbers))
```

This is used in real life for callbacks, among others.

6.3 Return it

... *from* another function:

```
In [ ]: import random

        func = random.choice([max, min])
        print("Function:", func)
        print("Result:", func(numbers))
```

6.4 Nested functions

Define it within another function, and return it!

```
In [ ]: def get_power_function(exponent):
            """Returns a function to compute the exponent-th power."""

            def power(n):
                return n ** exponent
            return power

        square = get_power_function(2)
        cube   = get_power_function(3)
        n = 4

        print("Number:", 4)
        print("Square:", square(n))
        print("Cube  :", cube(n))
```

7 So, decorators...

... are "wrappers" that let us execute code *before* and *after* the function that they decorate without modifying the function itself. We:

- Take the function as an argument.
- Add some behaviour, wrapping it in a new function.
- Return (and later use) the new function.

```
In [ ]: import time

        def measure_time(func):
            """A decorator for measuring the execution time of a function."""

            def wrapped(*args):
                tstart = time.time()
                result = func(*args)
                tend = time.time()
                tdelta = tend - tstart
                print("Function call took {} seconds".format(tdelta))
                return result

            return wrapped
```

Note: no `**kwargs` for simplicity's sake.
A more Pythonic approach, by the way, would be to use finally:

```
In [ ]: import time

        def measure_time(func):
```

```
        """A decorator for measuring the execution time of a function."""

        def wrapped(*args):
            try:
                tstart = time.time()
                return func(*args)
            finally:
                tend = time.time()
                tdelta = tend - tstart
                print("Function call took {} seconds".format(tdelta))

        return wrapped
```

No need to store the result in a variable to return it later.
Let's now decorate something:

```
In [ ]: def square_everything(numbers):
            """Return the square of all the numbers."""

            result = []
            for n in numbers:
                result.append(n ** 2)
            return result

        func = measure_time(square_everything)
        print(func(numbers))
```

8 Missing attributes

A problem with our decorated function is that we lose attributes such as the name or docstring.

```
In [ ]: def square_everything(numbers):
            """Return the square of all the numbers."""

            result = []
            for n in numbers:
                result.append(n ** 2)
            return result

        print("Name:", square_everything.__name__)
        print("Docstring:", square_everything.__doc__)
```

However...

```
In [ ]: func = measure_time(square_everything)

        print("Name:", func.__name__)
        print("Docstring:", func.__doc__)
```

This is unfortunate, as these are great for e.g. debugging.

9 functools.wraps()

`wraps()` function allows us to overwrite the function attributes (`__name__`, `__doc__`, `__module__`, etc) attributes of the wrapper function with those of the *original* function.

```
In [ ]: import functools
        import time

        def measure_time(func):
            """A decorator for measuring the execution time of a function."""

            @functools.wraps(func)  # note this
            def wrapped(*args):
                try:
                    tstart = time.time()
                    return func(*args)
                finally:
                    tend = time.time()
                    tdelta = tend - tstart
                    print("Function call took {} seconds".format(tdelta))

            return wrapped
```

In this manner, the changes to the function are transparent.

```
In [ ]: def square_everything(numbers):
            """Return the square of all the numbers."""

            result = []
            for n in numbers:
                result.append(n ** 2)
            return result

        func = measure_time(square_everything)

        print("Name:", func.__name__)
        print("Docstring:", func.__doc__)
```

10 Python's Decorator Syntax

But we don't want to call our function `func`. Let's keep the original name:

```
In [ ]: def square_everything(numbers):
            """Return the square of all the numbers."""

            result = []
            for n in numbers:
                result.append(n ** 2)
```

```
        return result

    square_everything = measure_time(square_everything)
    print(square_everything(numbers))
```

Instead of...

```
In [ ]: square_everything = measure_time(square_everything)
```

... we can apply a decorator using this shortcut:

```
In [ ]: @measure_time
        def square_everything(numbers):
            result = []
            for n in numbers:
                result.append(n ** 2)
            return result

        print(square_everything(numbers))
```

That is: apply `measure_time` to `square_everything` and store it in `square_everything`. We're effective replacing it with the updated, wrapped version.

11 There's much more

- We can chain decorators, applying 2+ to the same function.
- We can have decorators that take arguments.
- A closely-related concept are *closures*.

But this is enough for now.

11.0.1 Recommended readings:

- Decorator Basics on Stack Overflow.
- The closures that moved Spielberg at PyConES 2016.

12 Going back to our problem...

Photo by Sumarie Slabber / CC BY-ND 2.0

We left it here, with our login calls *everywhere*.

```
In [ ]: import time
        import logging
        logging.basicConfig(format='%(asctime)s %(message)s', level=logging.DEBUG)

        class Cyborg(object):

            def __init__(self, name):
```

```
            logging.info("Creating new Cyborg with name '%s'", name)
            self.name = name
            self.weapon = DeathRay(ammunition=25)
            self.teleporter = TimeMachine()
            self.alive = True

        def travel(self, destination, year):
            logging.info("Travelling to %s and year %s", destination, year)
            self.teleporter.go(destination, year)
            time.sleep(0.25)  # not instant, but almost

        def attack(self, target):
            logging.info("Attacking %s", target)
            self.weapon.vaporize(target)
```

13 Attempt 3: method decorators

```
In [2]: import functools

        def log(func):
            @functools.wraps(func)
            def wrapped(*args):  # omit **kwargs for simplicity
                logging.info('Called %s, args=%s', func.__name__, args)
                return func(*args)
            return wrapped

        @log
        def cube(n):
            """Return the third power of 'n'."""
            return n ** 3

        print(cube(2))
```

```
2017-09-10 14:02:11,235 Called cube, args=(2,)


8
```

So going back to our `Cyborg` class...

```
In [3]: import time
        import logging

        def log(func):
            @functools.wraps(func)
            def wrapped(*args):  # omit **kwargs for simplicity
                # args[1:] so that we don't print 'self'
```

```
            logging.info('Called %s%s', func.__name__, args[1:])
            return func(*args)
        return wrapped


    class Cyborg(object):

        @log
        def __init__(self, name):
            self.name = name
            self.weapon = DeathRay(ammunition=25)
            self.teleporter = TimeMachine()

        @log
        def travel(self, destination, year):
            self.teleporter.go(destination, year)
            time.sleep(0.25)  # not instant, but almost

        @log
        def attack(self, target):
            self.weapon.vaporize(target)
```

```
In [6]: robot = Cyborg('T-1000')
        robot.travel('Los Angeles', 1995)
        robot.attack('Sarah Connor')
        robot.attack('John Connor')
```

```
2017-09-10 14:02:28,398 Called __init__('T-1000',)
2017-09-10 14:02:28,400 Called travel('Los Angeles', 1995)
2017-09-10 14:02:28,652 Called attack('Sarah Connor',)
2017-09-10 14:02:28,653 Called attack('John Connor',)
```

14 This didn't solve the problem, though

- We have replaced calls to `logging.info()` with calls to `@log`.
- Whoever adds a new method in the future might forget to *decorate* it.
- We still could have three *hundred* methods instead of three.
- Also, copy-pasting method decorations is still boring.

15 Attempt 4: class decorators

Let a second decorator do the work for us → decorate all the methods.

```
In [13]: def decorate_all_methods(decorator):
             def wrapped(cls):
                 for attr_name in cls.__dict__:
```

```
                    attr = getattr(cls, attr_name)
                    if callable(attr):
                        # It's a function, decorate it
                        setattr(cls, attr_name, decorator(attr))
                return cls
            return wrapped
```

```
In [14]: @decorate_all_methods(log)
         class Cyborg(object):

             def __init__(self, name):
                 self.name = name
                 self.weapon = DeathRay(ammunition=25)
                 self.teleporter = TimeMachine()

             def travel(self, destination, year):
                 self.teleporter.go(destination, year)
                 time.sleep(0.25)  # not instant, but almost

             def attack(self, target):
                 self.weapon.vaporize(target)
```

```
In [15]: robot = Cyborg('T-1000')
         robot.travel('Los Angeles', 1995)
         robot.attack('Sarah Connor')
         robot.attack('John Connor')
```

```
2017-09-10 14:03:59,759 Called __init__('T-1000',)
2017-09-10 14:03:59,761 Called travel('Los Angeles', 1995)
2017-09-10 14:04:00,013 Called attack('Sarah Connor',)
2017-09-10 14:04:00,015 Called attack('John Connor',)
```

16 We did it! Hoo-ray!

```
In [16]: @decorate_all_methods(log)
         class Cyborg(object):

             def __init__(self, name):
                 self.name = name
                 self.weapon = DeathRay(ammunition=25)
                 self.teleporter = TimeMachine()

             def travel(self, destination, year):
                 self.teleporter.go(destination, year)
                 time.sleep(0.25)  # not instant, but almost

             def attack(self, target):
                 self.weapon.vaporize(target)
```

17 We *didn't* solve the problem, though

- We just moved one level up the ladder of abstraction.
- We have replaced decorating methods with decorating *classes*.
- Whoever adds a new *class* in the future might forget to *decorate* it.
- What if we had one *hundred* classes instead of just one?

```
In [17]: @decorate_all_methods(log)
         class Ninja(object): pass
             # ...

         class Human(object): pass
             # ...

         class Terminator(object): pass
             # ...
```

18 Nested classes

Another limitation is that out decorator will also decorate *nested* classes, even if that's not what we want.

```
In [18]: @decorate_all_methods(log)
         class Cyborg(object):

             class Chainsaw(object):

                 # This method was also decorated.
                 def vaporize(self, victim): pass
                     # ...

             def __init__(self, name):
                 self.name = name
                 self.weapon = Cyborg.Chainsaw()

             def attack(self, target):
                 self.weapon.vaporize(target)

         robot = Cyborg('T-1000')
         robot.attack('Sarah Connor')
```

```
2017-09-10 14:04:03,445 Called __init__('T-1000',)
2017-09-10 14:04:03,446 Called Chainsaw()
2017-09-10 14:04:03,448 Called attack('Sarah Connor',)
```

19 There are more limitations

For example, if we want to be able to add *classes*:

```
In [ ]: Android = Human + Robot  # get new class and use it
        replicant = Android('Roy Batty')
```

19.0.1 Recommended readings:

- Python metaclasses vs class decorators on Stack Overflow.
- What are Python metaclasses useful for? also by Alex Martelli.

There're things that simply *cannot* be done with class decorators.
But *metaclasses* can.

20 Help me, Obi-Wan Kenobi. You're my only hope

Photo by texaus1 / CC BY 2.0

21 Classes and instances

- Every object is the instance of a class.
- We can check this with the built-in `type()`.

```
In [ ]: number = 21
        word = "blue"

        print(type(number))
        print(type(word))
```

Let's check our own class too:

```
In [ ]: class Cyborg(object):

            def __init__(self, name):
                self.name = name
                self.weapon = DeathRay(ammunition=25)
                self.teleporter = TimeMachine()

            def travel(self, destination, year):
                self.teleporter.go(destination, year)
                time.sleep(0.25)  # not instant, but almost

            def attack(self, target):
                self.weapon.vaporize(target)

        arnold = Cyborg("T-800")
        print(type(arnold))
```

22 Wait, but...

- (1) *Every* object is an instance of a class.
- (2) We said that classes are *objects* too.
- From (1) and (2) it follows that classes are instances of... what?

```
In [ ]: print(type(int))
        print(type(str))
        print(type(Cyborg))
```

So all classes are instances of class `type`.

23 Down the rabbit hole

And `type` is an instance of...

```
In [ ]: print(type(type))
```

Everything is an instance of `type`, including *itself*.

24 The Most Important Slide

If you're going to remember only one thing, it should be this:

> Instances of a class are to **classes** what classes are to **metaclasses**.

TODO: necesitamos nuestro propio diagrama, because of copyright

- In the same way a class defines the creation and behaviour of an instance of that class...
- ... metaclasses allow us to define the creation and behaviour of our **classes**.

25 `type()` to define new classes

As we just saw, the built-in `type()` returns the type on an object.

```
In [ ]: numbers = [1, 2, 3]
        print(type(numbers))
```

However, it can also be called with *three* arguments to return a **new type** object. From the docs:

> With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The *name* string is the class name and becomes the `__name__` attribute; the *bases* tuple itemizes the base classes and becomes the `__bases__` attribute; and the *dict* dictionary is the namespace containing definitions for class body and is copied to a standard dictionary to become the `__dict__` attribute. For example, the following two statements create identical `type` objects:

```
In [ ]: class X(object):
            a = 1

        # The above class is identical to:
        X = type('X', (object,), dict(a=1))
```

26 Another example

```
In [ ]: name = "Cylon"
        base = (object,)

        # We need 'self', of course.
        def attack(self, victims):
            print("Destroy all {}!".format(victims))

        body = {'attack': attack}

        Cylon = type(name, base, body)
        centurion = Cylon()
        centurion.attack("humans")
```

27 How is this useful?

What's the purpose of creating classes dynamically with `type()`?

- We definitely don't need them *often*.
- However, *sometimes* is the appropriate solution.
- For example, for GUI programming it's very convenient to define classes on the fly and instantiate widgets with them.

27.1 A real-life example: SQLAlchemy

> There is only one way to register a database table with SQLAlchemy: create a `Model` class describing the table (not unlike Django's models). To get SQLAlchemy to recognize a table, a class for that table must be created in some way. Since `sandman` doesn't have any advanced knowledge of the database structure, it can't rely on pre-made model classes to register tables. Rather, it needs to introspect the database and create these classes on the fly. Sound familiar? **Any time you're creating new classes dynamically, `type` is the correct/only choice**.

Improve Your Python: Metaclasses and Dynamic Classes With Type, by Jeff Knupp.

28 Metaclasses — finally!

It took us a while, but we're finally here.

Photo by Chris / CC BY 2.0

29 The most stupid metaclass ever

```
In [19]: class Meta42(type):
             def __new__(meta,name,bases,kwargs):
                 return 42

         class Cyborg42(metaclass=Meta42):
             pass

         print(Cyborg42)
```

```
42
```

Woahhhh! What happened here? As you can see, the '**new**' method of the metaclass is in charge of creating the class itself. In this example we simply return a number, which is totally allowed even if the number is not itself a class. Python trust us (it should't).

30 The yolooo metaclass

```
In [21]: def make_yolooo(func):
             def wrapper(*args,**kwargs):
                 try:
                     return func(*args,**kwargs)
                 except Exception as ex:
                     print(f"{ex.__class__.__name__} supressed")
             return wrapper

         class MetaYolooo(type):
             def __new__(meta, base, names, kwargs):
                 for name,attr in kwargs.items():
                     if callable(attr):
                         kwargs[name] = make_yolooo(attr)
                 return super().__new__(meta,base,names,kwargs)
```

In this example, we are creating a tecorator that will supress all exceptions raised in the decorated function. To make it work we only need to create a metaclass that automatically decorates all mehods inside a class.

At this point we only need to use our new metaclass like this:

```
In [22]: class DefectiveCyborg(object,metaclass=MetaYolooo):

             def __init__(self, name):
                 raise ValueError

             def travel(self, destination, year):
                 raise ZeroDivisionError
```

```
            def attack(self, targetb:
                raise RuntimeError

        robot = DefectiveCyborg('T-1000')
        robot.travel('Los Angeles', 1995)
        robot.attack('Sarah Connor')
        robot.attack('John Connor')
```

```
ValueError supressed
ZeroDivisionError supressed
RuntimeError supressed
RuntimeError supressed
```

31 The registering metaclass

Now for some real_life examples. If we are designing some library that allows the user to declare its own classes based on the ones we provide (imagine for example Django, SQLAlchemy, Luigy, Ansible, ...etc) and we want to perform some kind of operations based on the classes that the user has defined along with some of our own design. It would be very handy if we can have automatically a registry of all the clases that have been declared using the ones we provide. This is very easy with metaclasses:

```
In [37]: from types import MappingProxyType
         class MetaRegister(type):

             _registry = dict()

             def __new__(meta, name, bases, kwargs):
                 kwargs["_registry"] = MappingProxyType(meta._registry)
                 cls = super().__new__(meta,name,bases,kwargs)
                 meta._registry[name] = cls
                 return cls
```

We can now define the base class our library will provide:

```
In [38]: class Cyborg(object,metaclass=MetaRegister):
             pass
```

The user of our library will declare some classes based on this last one:

```
In [39]: class Terminator(Cyborg):
             pass

         class Ninja(object):
             pass

         class CyborgNinja(Cyborg,Ninja):
             pass
```

Now we have all the clases available as a read-only attribute inside every class and as a read-write attribute inside the metaclass:

```
In [42]: print(MetaRegister._registry)
         print(CyborgNinja._registry)
```

```
{'Cyborg': <class '__main__.Cyborg'>, 'Terminator': <class '__main__.Terminator'>, 'CyborgNinja
{'Cyborg': <class '__main__.Cyborg'>, 'Terminator': <class '__main__.Terminator'>, 'CyborgNinja
```

Note that we can only modify the metaclass registry as the ones inside the regular classes are read-only:

```
In [43]: # We can edit MetaRegister._registry:
         MetaRegister._registry["tuple"] = tuple
         # But now CyborgNinja._registry:
         CyborgNinja._registry["list"] = list
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-43-0cca2523b0f9> in <module>()
  2 MetaRegister._registry["tuple"] = tuple
  3 # But now CyborgNinja._registry:
----> 4 CyborgNinja._registry["list"] = list


TypeError: 'mappingproxy' object does not support item assignment
```

32 The type enforcing metaclass

Let's play a bit with some dark magic. We can start declaring the following descriptor that checks if the value assigned to it is positive:

```
In [53]: class Positive:
             def __set_name__(self,owner,name):
                 self.name = name
             def __get__(self,instance,owner):
                 return instance.__dict__[self.name]
             def __set__(self,instance, value):
                 if value > 0:
                     instance.__dict__[self.name] = value
                 else:
                     raise ValueError("The value is not positive")
```

We can use this descriptor in the usual way:

```
In [48]: class Wallet(object):
             money  = Positive()

         my_wallet = Wallet()
         my_wallet.money = 5
         print(my_wallet.money)
         my_wallet.money = -5
```

```
5
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-48-454ba590f612> in <module>()
          5 my_wallet.money = 5
          6 print(my_wallet.money)
    ----> 7 my_wallet.money = -5


        <ipython-input-45-32becd2013e5> in __set__(self, instance, value)
          8             instance.__dict__[self.name] = value
          9         else:
    ---> 10             raise ValueError("The value is not positive")


        ValueError: The value is not positive
```

Python 3 has this awesome feature called "type annotations" that allows us to indicate some typing information about the variables, methods, classes and functions. This is usually not enforced, but we can play a bit with that. Let's define the following metaclass:

```
In [64]: import collections
         class MetaAnnotations(type):
             @classmethod
             def __prepare__(metacls, name, base, **kwargs):
                 return collections.ChainMap({},{"enforce_positive":Positive})

             def __new__(meta,name,bases,kwargs):
                 return super().__new__(meta,name,bases,kwargs.maps[0])

             def __init__(cls,name,bases,kwargs):
                 for name,val in getattr(cls,"__annotations__",{}).items():
                     desc = val()
                     desc.__set_name__(cls,name)
                     setattr(cls,name,desc)
```

We can now define our wallet again with type annotations:

```
In [65]: class Wallet(object,metaclass=MetaAnnotations):
             money  : enforce_positive

         my_wallet = Wallet()
         my_wallet.money = 5

In [66]: print(my_wallet.money)
         my_wallet.money = -5
```

```
5
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-66-04b9f4f5dcb6> in <module>()
  1 print(my_wallet.money)
----> 2 my_wallet.money = -5


<ipython-input-53-32becd2013e5> in __set__(self, instance, value)
  8             instance.__dict__[self.name] = value
  9         else:
---> 10             raise ValueError("The value is not positive")


ValueError: The value is not positive
```

Notice that we also achieve something really weird: "enforce_positive" is not defined anywere as a variable! Our metaclass is allowing us to use the name 'enforce_positive' as an alias for the 'Positive' without needing to import the name into our scope. Awesome!

We can check that using a regular variable in our class definition:

```
In [68]: class Wallet(object,metaclass=MetaAnnotations):
             money = enforce_positive

         my_wallet = Wallet()
         print(my_wallet.money)
```

```
<class '__main__.Positive'>
```

33 Abstract base clases

Some times is very handy to pay a visit to the old good C++ world and bring some abstract base classes as a souvenir. These are very handy if we want to define interfaces or if we want to enforce some common properties when the users of our library will extend and define their own classes based on the ones we provide. Python make this very easy for us with the 'abc' library:

```
In [75]: from abc import ABCMeta, abstractmethod

         class MetaCyborg(metaclass=ABCMeta):
             @abstractmethod
             def travel(self, destination, year):
                 ...

             @abstractmethod
             def attack(self, target):
                 ...
```

Once we have defined our abstract base class we can use it like this:

```
In [76]: class PrototypeCyborg(MetaCyborg):
             def __init__(self, name):
                 self.name = name

             def travel(self, destination, year):
                 print("Traveling really far")

         class Cyborg(MetaCyborg):

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

             def travel(self, destination, year):
                 print("Traveling really far")

             def attack(self,target):
                 print("Atacking really hard")

In [77]: robot = Cyborg('T-1000')

In [78]: #This doesn't work
         robot = PrototypeCyborg('T-1000')


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-78-280f37ca8d7c> in <module>()
```

```
      1 #This doesn't work
----> 2 robot = PrototypeCyborg('T-1000')


TypeError: Can't instantiate abstract class PrototypeCyborg with abstract methods atta
```

34 No more init

Tired of writing the same piece of code thousands of times to declare '**init**'and assign the variables inside? Don't worry because we have exactly the stupid idea you need! The 'NoMoreInit' metaclass:

```
In [89]: import collections
         class MetaNoMoreInit(type):
             @classmethod
             def __prepare__(metacls, name, base, **kwargs):
                 return collections.OrderedDict()
             def __new__(meta,name,bases,kwargs):
                 if "__init__" not in kwargs:
                     annotations = kwargs["__annotations__"]
                     def __init__(self,*args,**kwargs):
                         if args or kwargs.keys() != annotations.keys():
                             raise ValueError("Incorrect arguments, my lord!")
                         self.__dict__.update(kwargs)
                     kwargs["__init__"] = __init__
                 return super().__new__(meta,name,bases,kwargs)
```

We can now use type annotations (of course) to declare the variables that our class will define and we will have a '**init**' prepared for us:

```
In [90]: class Easy(metaclass=MetaNoMoreInit):
             x:int
             y:float
             z:str

         myobj = Easy(x=1,y=3.4,z="Hello")
         print(myobj.x)
         print(myobj.y)
         print(myobj.z)
```

```
1
3.4
Hello
```

35 Back to our problem

Photo by Anthony Quintano / CC BY 2.0

In our pursue for purity and maximum generalization we look back at what we have achieved and we are pleased. But suddenly....something is wrong. Something is still verbose. Something is still.....explicit:

```
In [94]: # See this line? It's... explicit
         class HappyCyborg(object,metaclass = MetaRegister):
             ...
```

We still need to write `metaclass = MyMetaclass` every time we declare a new base class. We don't need to do that in every child class that inherits from it but we still need to do that with the parents. It wouldn't be great if we had a way to avoid that?

36 Enter the `__build_class__` hook

Every time python defines a new class, before doing anything else (even before resolving the metaclasses) it executes a function. This function is part of the Cpython interpreter (is written in C) and is in charge of building the class attribute dictionary among other things. We can actually substitute it behaviour thanks to a hook that the interpreter exposes for us.

```
In [1]: import builtins
        print(builtins.__build_class__) # <--- This is the hook
```

```
<built-in function __build_class__>
```

We an use this to automatically inject out metaclass in the `metaclass` attribute of every class that is ever defined:

```
In [2]: __builtins_build_class = builtins.__build_class__

        def _create_custom_build(metaclass):
            def _custom_build(func, name, *bases, **kwargs):
                print([base is not object and isinstance(base, type) for base in bases])
                print(bases)
                if not any((base is not object and isinstance(base, type) for base in bases)):
                    if 'metaclass' in kwargs:
                        pass
                    else:
                        kwargs['metaclass'] = metaclass

                return __builtins_build_class(func, name, *bases, **kwargs)
            return _custom_build
```

To use it we can substitute the original `__build_class__`:

```
In [3]: builtins.__build_class__ = _create_custom_build(lambda *a,**k:42)

        class A(object):
            pass

        class AHappyAndNormalClass(A):
            pass

        print(AHappyAndNormalClass)
```

```
[False]
(<class 'object'>,)
[False]
(42,)
42
```

We can return everything to normal reasigning the old `__build_class__` back.

```
In [16]: builtins.__build_class__ = __builtins_build_class
```

37 A last crazy idea

We are happy, we are fulfilled. Everything is now stupidly implicit and that makes us great programmers. But after a while we notice something....something that is wrong. **We still need to write out hook in every program we write!**. That makes us sad. It wouldn't be great if we had a way to avoid that?

38 Enter the `*.pth` files

Python has this thing called `*.pth` files. The deal with them is the following: if python finds a .pth file when in `lib/pythonX.Y/site-packages` that starts with `import` it will **EXECUTE** the file.

> A path configuration file is a file whose name has the form name.pth and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to sys.path. Non-existing items are never added to sys.path, and no check is made that the item refers to a directory rather than a file. No item is added to sys.path more than once. Blank lines and lines beginning with # are skipped. Lines starting with import (followed by space or tab) are executed.

So we can create a `*.pth` file that performs the same operations as our `__build_class__` hook and it will be automatically executed every time we initialize the interpreter.

pip -V pip 9.0.1 from /usr/lib/python3.6/site-packages (python 3.6)
cat /usr/lib/python3.6/site-packages/metathing.pth

```
import metathing
```

cat /usr/lib/python3.6/site-packages/metathing.py

```
class Meta42(type):
    def __new__(meta,name,bases,kwargs):
        return 42

__builtins_build_class = builtins.__build_class__

def _create_custom_build(metaclass):
    def _custom_build(func, name, *bases, **kwargs):
        if not any((base is not object and isinstance(base, type) for base in bases)):
            if 'metaclass' in kwargs:
                pass
            else:
                kwargs['metaclass'] = metaclass

        return __builtins_build_class(func, name, *bases, **kwargs)
    return _custom_build
builtins.__build_class__ = _create_custom_build(Meta42)
```

39 Parting Words

> Metaclasses are deeper magic than 99% of users should ever 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) — Tim Peters.

We may never *need* them — but now we *know* what they are.
Photo by G0DeX / CC BY 2.0