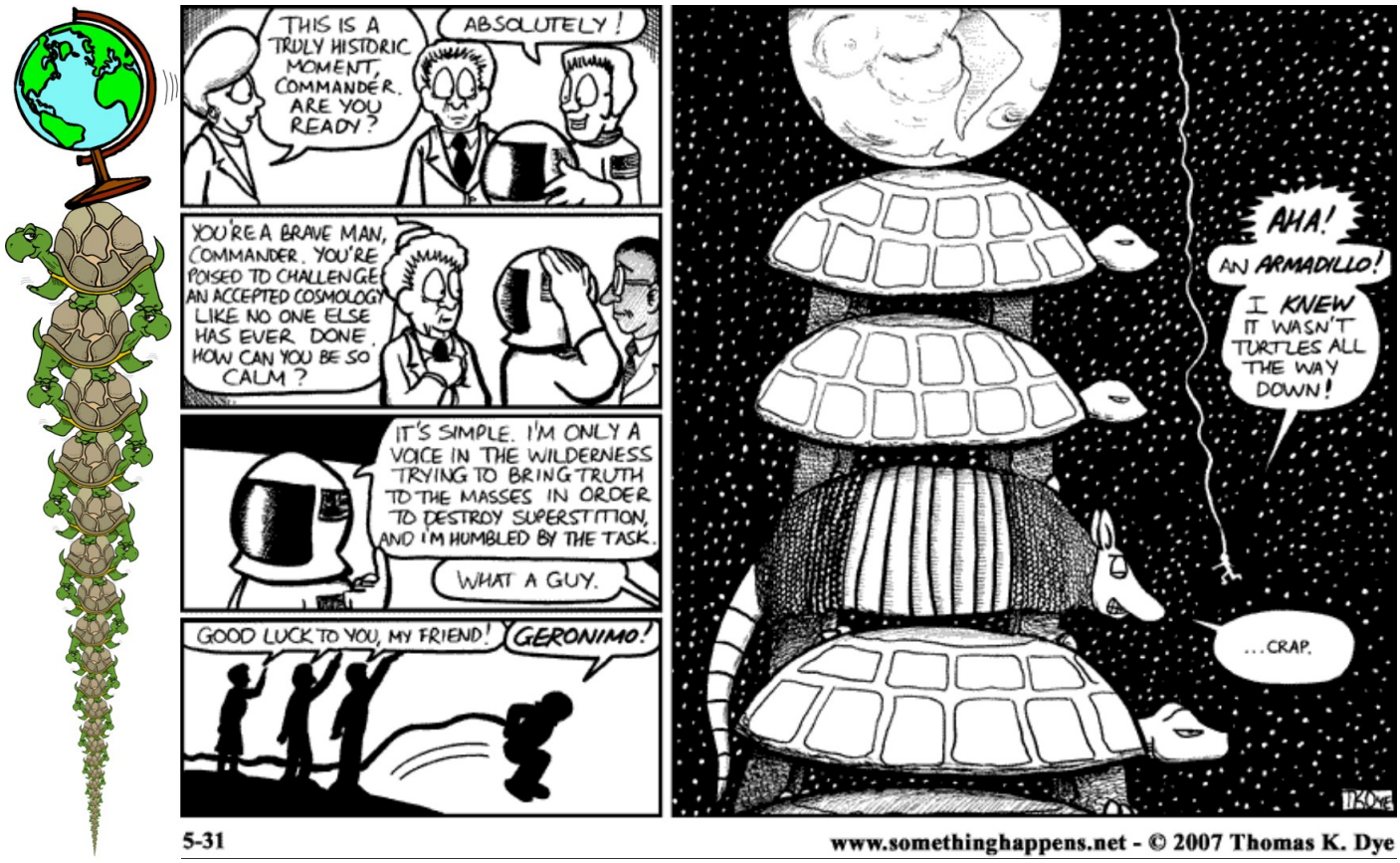
It's metaclasses all the way down:

Understanding and using Python's metaprogramming facilities



What is metaprogramming?

> Metaprogramming is a technique of writing computer programs that can treat themselves as data, so you can introspect, generate, and/or modify them while running

- Python has the ability to introspect its basic elements such as functions, classes, or types and to create or modify them on the fly.
- Higher order functions allow us to add/modify functionality of existing functions, methods, or classes.
- Special methods of classes that allow us to interfere with the creation of class objects. These are called **Metaclasses** and allow us to even completely redesign the Python's object-oriented paradigm.
- A selection of different tools that allow programmers to work directly with code either in its raw plain text format or in the more programmatically accessible **Abstract Syntax Tree (AST)** form. This approach is allows for really extraordinary things, such as extending Python's language syntax or even creating your own Domain Specific Language (DSL), see e.g. Hy (https://github.com/hylang/hy).

Why would you use metaclasses?

Well, usually you don't.

> 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**

However:

> The potential uses for metaclasses are boundless. Some ideas that have been explored include logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization. **Python Documentation**

Quick example 1: Django ORM

A typical use for a metaclass is creating an API (e.g. the Django ORM). It allows us to define something like this:

```
class Person(models.Model):
    name = models.CharField(max_length=30)
    age = models.IntegerField()
```

But if we do this:

```
guy = Person(name='bob', age=35)
print(guy.age)
```

- It won't return an `IntegerField` object. It will return an `int`, and can even take it directly from the database.
- This is possible because `models.Model` uses the `ModelBase` metaclass and it uses some magic that will turn the `Person` you just defined with simple statements into a complex hook to a database field.
- Without specifying an `__init__` method, I am able to instantiate a `Person` object. This is done through the metaclass.
- Django makes something complex look simple by exposing a simple API and using metaclasses, recreating code from this API to do the real job behind the scenes.

Quick example 2: Abstract Base Classes

- Python does not have special syntax for abstract classes.
- We are able to implement this behavior using metaclasses.
- Classes that use the meta-class can use `@abstractmethod` and `@abstractproperty` to define abstract methods and properties.
- The metaclass will ensure that derived classes override the abstract methods and properties.
- Also, classes that implement the ABC without actually inheriting from it can register as implementing the interface, so that `issubclass` and `isinstance` will work.
- When we want to enforce contracts between classes in python just as interfaces do in Java, abstract base classes is the way to go.

In [1]:
```
from abc import ABCMeta, abstractmethod

class Vehicle(metaclass=ABCMeta):
    @abstractmethod
    def change_gear(self):
        pass

    @abstractmethod
    def start_engine(self):
        pass

class Car(Vehicle): # subclass the ABC, abstract methods MUST be overridden
    def __init__(self, make, model, color):
        self.make = make
        self.model = model
        self.color = color

try:
    car = Car('Toyota', 'Avensis', 'silver')
except TypeError as e:
    print(e)
```

```
Can't instantiate abstract class Car with abstract methods change_gear, start
_engine
```

You are using them already

- You are using metaclasses even if you are not aware of using them.
- They make things possible in Python that wouldn't be otherwise or would require substantial changes to the language.
- You are a programmer willing to go out of his comfort zone, learn new things and challenge common practices.

but but but... I want to make my own metaclasses!

OK, I got a simple problem for you

- You are friends with some guys, they are quite good programmers.
- The learned Python but all have a Javascript background (you know, from that **other** meetup).
- They wrote a Python library that you really want to use.
- They used `camelCase` for all their methods instead of `underscore_method_names`.
- You really want to use their library but love `get_first_name` and hate `getFirstName`

Possible solutions

1. Override `__getattr__`:
    - Renaming must happen everytime you access the method.
    - When trying to get attribute/method of an object, the object's superclasses are looked up **before** `__getattr__`. Problems with inherited methods.
2. Introspection. Use `inspect.getmembers(MyClass, inspect.isfunction)` and rename functions. This works but:
    - You need to run this everytime you load each class.
    - Is too specific. What if you want to add extra functionality in how this classes operate (e.g. make a class Singleton or make each class register itself somewhere when it's being used)?
3. Class decorator:
    - Need to redefine a new nested class for every class you want to alter.
    - You will run into problems with inheritance.
4. Use metaclasses.

Warning! Metaclasses ahead!

Classes as objects

- In most languages, classes are just pieces of code that describe how to produce an object.
- But classes are more than that in Python. Classes are objects too.
- This object (the class) is itself capable of creating objects (the instances), and this is why it's a class.
  But still, it's an object, and therefore:
  - You can assign it to a variable.
  - You can copy it.
  - You can add attributes to it.
  - You can pass it as a function parameter.

In [2]:
```
class ObjectCreator:
    pass
print(ObjectCreator)  # you can pass a class as a parameter because it's an ob
ject
```

```
<class '__main__.ObjectCreator'>
```

In [3]:
```
print(hasattr(ObjectCreator, 'new_attribute'))
ObjectCreator.new_attribute = 'foo' # you can add attributes to a class
print(hasattr(ObjectCreator, 'new_attribute'))
print(ObjectCreator.new_attribute)
```

```
False
True
foo
```

In [4]:
```
ObjectCreatorMirror = ObjectCreator # you can assign a class to a variable
print(ObjectCreatorMirror.new_attribute)
print(ObjectCreatorMirror())
```

```
foo
<__main__.ObjectCreator object at 0x00000132FE593780>
```

Creating classes dynamically

1. Remember the function `type`? The good old function that lets you know what type an object is.
2. `type` can also create classes on the fly. `type` can take the description of a class as parameters, and return a class as `type(name, bases, attrs)`.

    - `name` is a string giving the name of the class to be constructed.
    - `bases` is a tuple giving the parent classes of the class to be constructed.
    - `attrs` is a dictionary of the attributes and methods of the class to be constructed.

For example:

```
class Foo:
    bar = True
```

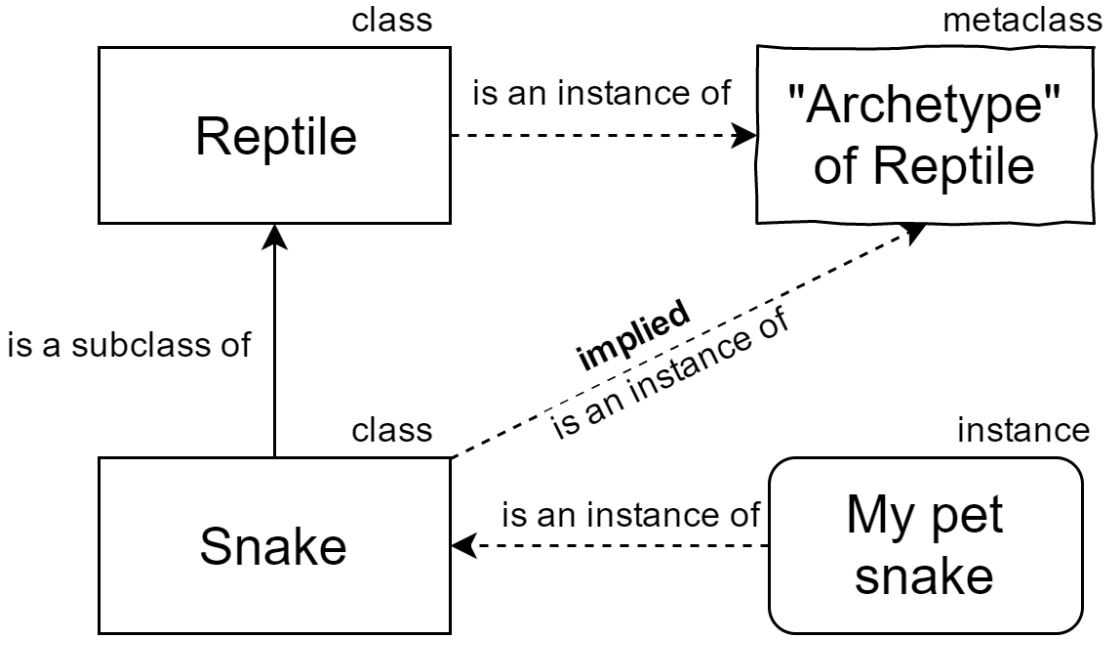
is equivalent with:

```
Foo = type('Foo', (), {'bar':True})
```

What are metaclasses

1. Metaclasses are the "stuff" that creates classes. You define classes in order to create objects, right?
2. We learned that Python classes are objects.
3. Well, metaclasses are what creates these objects. They are the classes' classes, you can picture them this way:

`MyClass = MetaClass()`, `MyObject = MyClass()`



What can act as a metaclass?

- Something that can create a class, i.e. anything that subclasses or uses `type`.
- Any python **callable**. A callable is an object that can be invoked with the function operator `()`.
  - Any Python class. `MyClass()` calls the `__init__()` function of the class
  - Any Python function or instance whose class implements `__call__()`. `foo()` is equivalent to `foo.__call__()`. This is true in both cases (i.e. all functions objects implement `__call__`).
- i.e. a metaclass is something that we call with some parameters and returns a class object
- When you write:

  ```
  class A:
    ...
  ```

  you are actually *not creating* the class, you *describe the class* and `type` creates this class object for you.
- Each class can have **one** metaclass, however this metaclass can subclass **multiple** metaclasses.

Our example with a function as metaclass

In [5]:
```
import re

def convert(name):
    s1 = re.sub('(.)([A-Z][a-z]+)', r'\1_\2', name)
    return re.sub('([a-z0-9])([A-Z])', r'\1_\2', s1).lower()

# the metaclass will automatically get passed the same arguments that we pass
 to `type`
def camel_to_snake_case(name, bases, attrs):
    """Return a class object, with its attributes from camelCase to snake_cas
e."""
    print("Calling the metaclass camel_to_snake_case to construct class: {}".f
ormat(name))

    # pick up any attribute that doesn't start with '__' and snakecase it
    snake_attrs = {}
    for attr_name, attr_val in attrs.items():
        if not name.startswith('__'):
            snake_attrs[convert(attr_name)] = attr_val
        else:
            snake_attrs[attr_name] = attr_val
    return type(name, bases, snake_attrs) # let `type` do the class creation
```

In [6]:
```
class MyVector(metaclass=camel_to_snake_case):
    def addToVector(self, other): pass
    def subtractFromVector(self, other): pass
    def calculateDotProduct(self, other): pass
    def calculateCrossProduct(self, other): pass
    def calculateTripleProduct(self, other): pass

print([a for a in dir(MyVector) if not a.startswith('__')])
```

```
Calling the metaclass camel_to_snake_case to construct class: MyVector
['add_to_vector', 'calculate_cross_product', 'calculate_dot_product', 'calcul
ate_triple_product', 'subtract_from_vector']
```

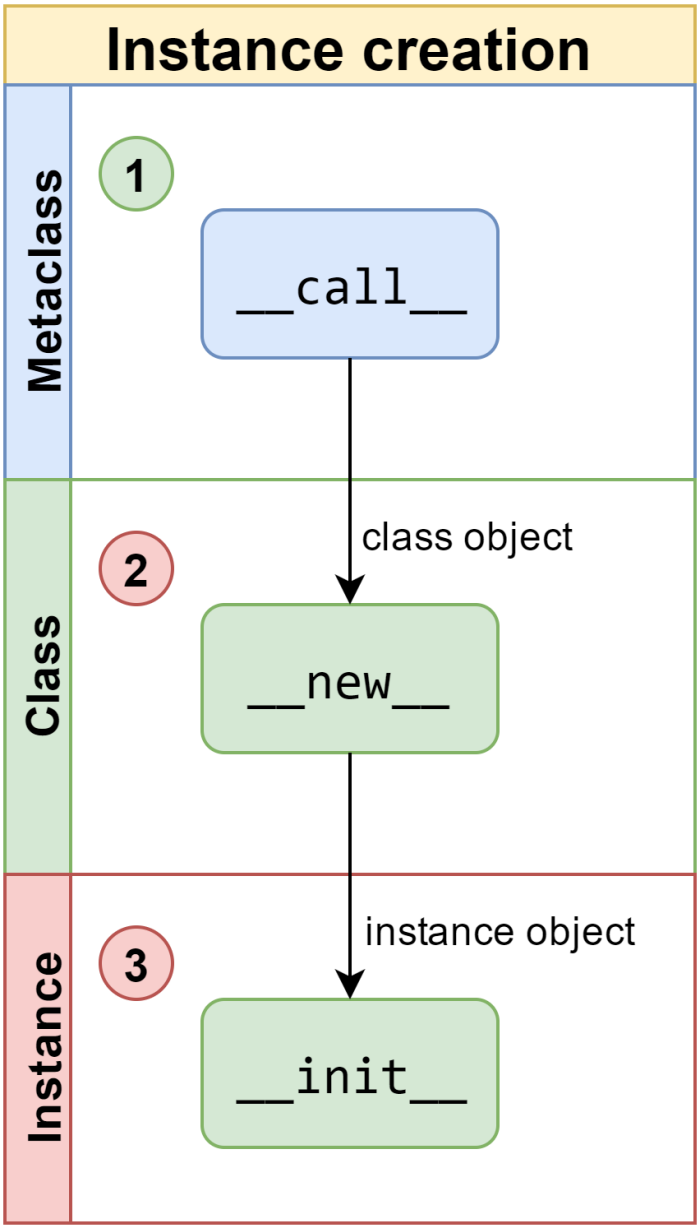
Instance creation: __new__ and __init__

In [7]:
```
def meta_function(name, bases, attrs):
    print('Calling meta_function')
    return type(name, bases, attrs)


class MyClass1(metaclass=meta_function):
    def __new__(cls, *args, **kwargs):
        """
        Called to create a new instance of class `cls`. __new__ takes the clas
s
        of which an instance was requested as its first argument. The remainin
g
        arguments are those passed to the object constructor expression
        (the call to the class). The return value of __new__ should be the
        new object instance (usually an instance of cls).
        """
        print('MyClass1.__new__({}, *{}, **{})'.format(cls, args, kwargs))
        return super().__new__(cls)

    def __init__(self, *args, **kwargs):
        """
        Called after the instance has been created (by __new__), but before it

        is returned to the caller. The arguments are those passed to the objec
t
        constructor. Note: both __new__ and __init__ receive the same argument
s.
        """
        print('MyClass1.__init__({}, *{}, **{})'.format(self, args, kwargs))
```

```
Calling meta_function
```

In [8]:
```
a = MyClass1(1, 2, 3, x='ex', y='why')
```

```
MyClass1.__new__(<class '__main__.MyClass1'>, *(1, 2, 3), **{'y': 'why', 'x':
 'ex'})
MyClass1.__init__(<__main__.MyClass1 object at 0x00000132FE5A4E80>, *(1, 2,
 3), **{'y': 'why', 'x': 'ex'})
```



Class creation: `__prepare__`, `__new__`, `__init__`, `__call__`

In [9]:
```
class MyMeta(type):
    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        """
        Called before the class body is executed and it must return a dictiona
ry-like object
        that's used as the local namespace for all the code from the class bod
y.
        """
        print("Meta.__prepare__(mcs={}, name={}, bases={}, **{}".format(
            mcs, name, bases, kwargs))
        return {}

    def __new__(mcs, name, bases, attrs, **kwargs):
        """
        Like __new__ in regular classes, which returns an instance object of t
he class
        __new__ in metaclasses returns a class object, i.e. an instance of the
 metaclass
        """
        print("MyMeta.__new__(mcs={}, name={}, bases={}, attrs={}, **{}".forma
t(
            mcs, name, bases, list(attrs.keys()), kwargs))
        return super().__new__(mcs, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **kwargs):
        """
        Like __init__ in regular classes, which initializes the instance objec
t of the class
        __init__ in metaclasses initializes the class object, i.e. the instanc
e of the metaclass
        """
        print("MyMeta.__init__(cls={}, name={}, bases={}, attrs={}, **{}".form
at(
            cls, name, bases, list(attrs.keys()), kwargs))
        super().__init__(name, bases, attrs)

        # Note: all three above methods receive as arguments:
        # 1. The name, bases and attrs of the future class that will be create
d
        # 2. Keyword arguments passed in the class inheritance list

    def __call__(cls, *args, **kwargs):
        """
        This is called when we make an instance of the class constructed with
 the metaclass
        """
        print("MyMeta.__call__(cls={}, args={}, kwargs={}".format(cls, args, k
wargs))
        self = super().__call__(*args, **kwargs)
        print("MyMeta.__call__ return: ", self)
        return (self)

print("Metaclass MyMeta created")
```

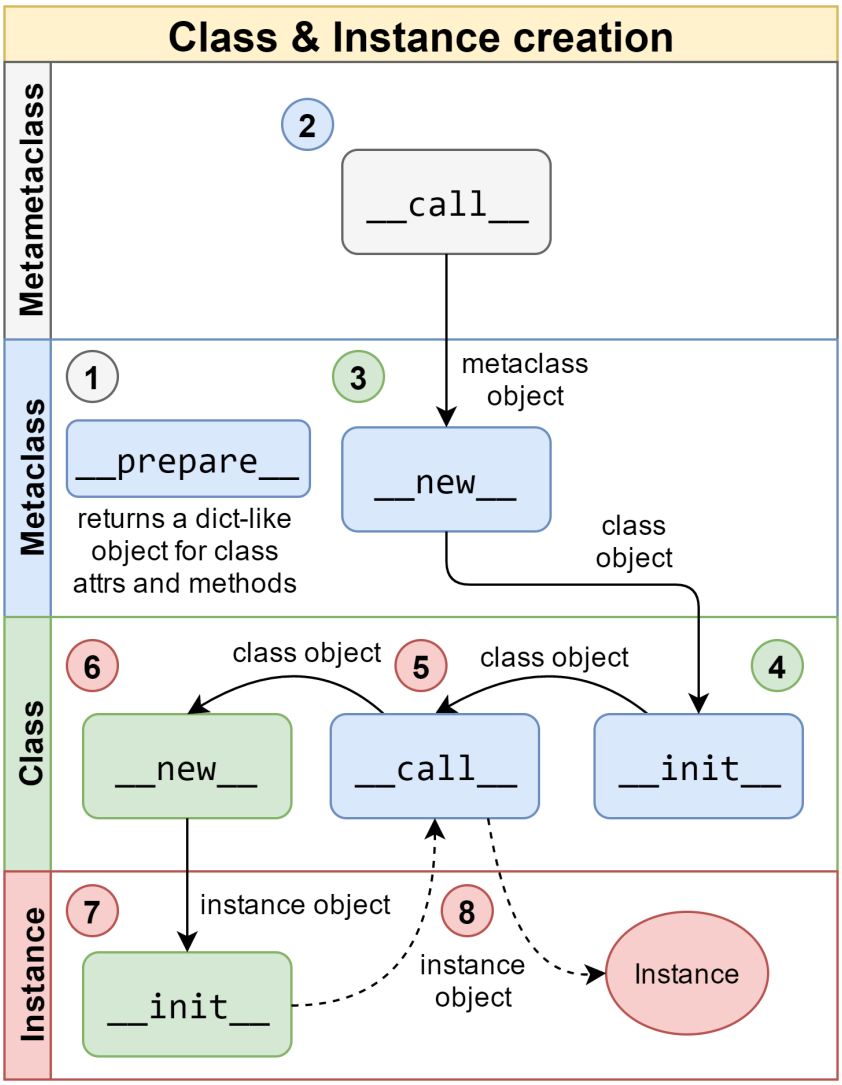
```
Metaclass MyMeta created
```

In [10]:
```
class MyClass2(metaclass=MyMeta, extra=1):
    def __new__(cls, s, a=0, b=0):
        print("MyClass2.__new__(cls={}, s={}, a={}, b={})".format(cls, s, a,
b))
        return super().__new__(cls)

    def __init__(self, s, a=0, b=0):
        print("MyClass2.__init__(self={}, s={}, a={}, b={})".format(self, s,
a, b))
        self.a, self.b = a, b

print("Class MyClass created")
```

```
Meta.__prepare__(mcs=<class '__main__.MyMeta'>, name=MyClass2, bases=(), **
{'extra': 1}
MyMeta.__new__(mcs=<class '__main__.MyMeta'>, name=MyClass2, bases=(), attrs=
['__new__', '__qualname__', '__module__', '__init__'], **{'extra': 1}
MyMeta.__init__(cls=<class '__main__.MyClass2'>, name=MyClass2, bases=(), att
rs=['__new__', '__qualname__', '__module__', '__init__'], **{'extra': 1}
Class MyClass created
```

In [11]:
```
a = MyClass2('hello', a=1, b=2)
print("MyClass instance created: ", a)
```

```
MyMeta.__call__(cls=<class '__main__.MyClass2'>, args=('hello',), kwargs=
{'a': 1, 'b': 2}
MyClass2.__new__(cls=<class '__main__.MyClass2'>, s=hello, a=1, b=2)
MyClass2.__init__(self=<__main__.MyClass2 object at 0x00000132FE5A9E48>, s=he
llo, a=1, b=2)
MyMeta.__call__ return:  <__main__.MyClass2 object at 0x00000132FE5A9E48>
MyClass instance created:  <__main__.MyClass2 object at 0x00000132FE5A9E48>
```



Our example with a class as metaclass

In [12]:
```
class CamelToSnake(type):
    def __new__(mcs, name, bases, attrs):
        # pick up any attribute that doesn't start with '__' and snakecase it
        snake_attrs = {}
        for attr_name, attr_val in attrs.items():
            if not name.startswith('__'):
                snake_attrs[convert(attr_name)] = attr_val
            else:
                snake_attrs[attr_name] = attr_val
        return super().__new__(mcs, name, bases, snake_attrs)
```

In [13]:
```
class MyVector(metaclass=CamelToSnake):
    def addToVector(self, other): pass
    def subtractFromVector(self, other): pass
    def calculateDotProduct(self, other): pass
    def calculateCrossProduct(self, other): pass
    def calculateTripleProduct(self, other): pass

print([a for a in dir(MyVector) if not a.startswith('__')])
```

```
['add_to_vector', 'calculate_cross_product', 'calculate_dot_product', 'calcul
ate_triple_product', 'subtract_from_vector']
```

Make our class a Singleton

In [14]:
```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class SnakeSingleton(CamelToSnake, Singleton):
    pass

class MyVector(metaclass=SnakeSingleton):
    def addToVector(self, other): pass
    def subtractFromVector(self, other): pass
    def calculateDotProduct(self, other): pass
    def calculateCrossProduct(self, other): pass
    def calculateTripleProduct(self, other): pass

print([a for a in dir(MyVector) if not a.startswith('__')])
v1 = MyVector(); v2 = MyVector()
print(v1 is v2)
```

```
['add_to_vector', 'calculate_cross_product', 'calculate_dot_product', 'calcul
ate_triple_product', 'subtract_from_vector']
True
```

References

1. StackOverflow - What is a metaclass in Python? (http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python)
2. StackOverflow - Good real-world uses of metaclasses (e.g. in Python) (http://stackoverflow.com/questions/2907498/good-real-world-uses-of-metaclasses-e-g-in-python)
3. StackOverflow - Creating a singleton in Python (http://stackoverflow.com/questions/6760685/creating-a-singleton-in-python)
4. ionel's codelog - Understanding Python Metaclasses (https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/)
5. Eli Bendersky - Python metaclasses by example (http://eli.thegreenplace.net/2011/08/14/python-metaclasses-by-example/)
6. M. Jaworski, T. Ziade - Expert Python Programming, 2nd edition (2006), Packt Publishing
7. David Beazley - Python Essential Reference, 4th edition (2009), Addison-Wesley

Thank you!