It's **Pythons All The Way Down**

Python Types &
Metaclasses
Made
Simple

Mark Smith
Nexmo

It's **Pythons All The Way Down**

Python Types &
Metaclasses
Made
Simple

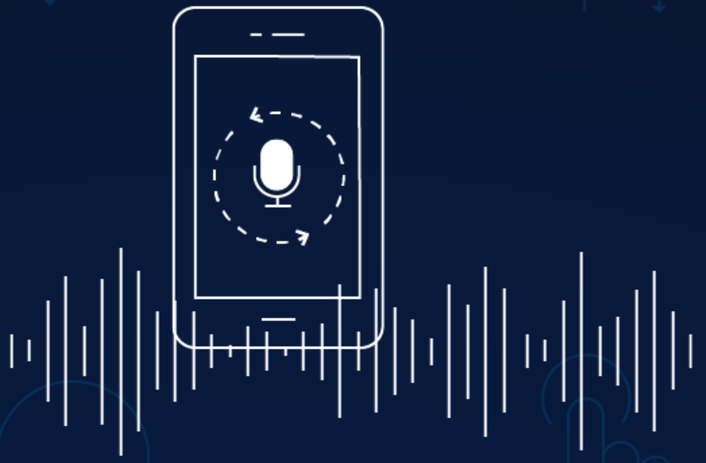
Mark Smith
Nexmo

@Judy2k

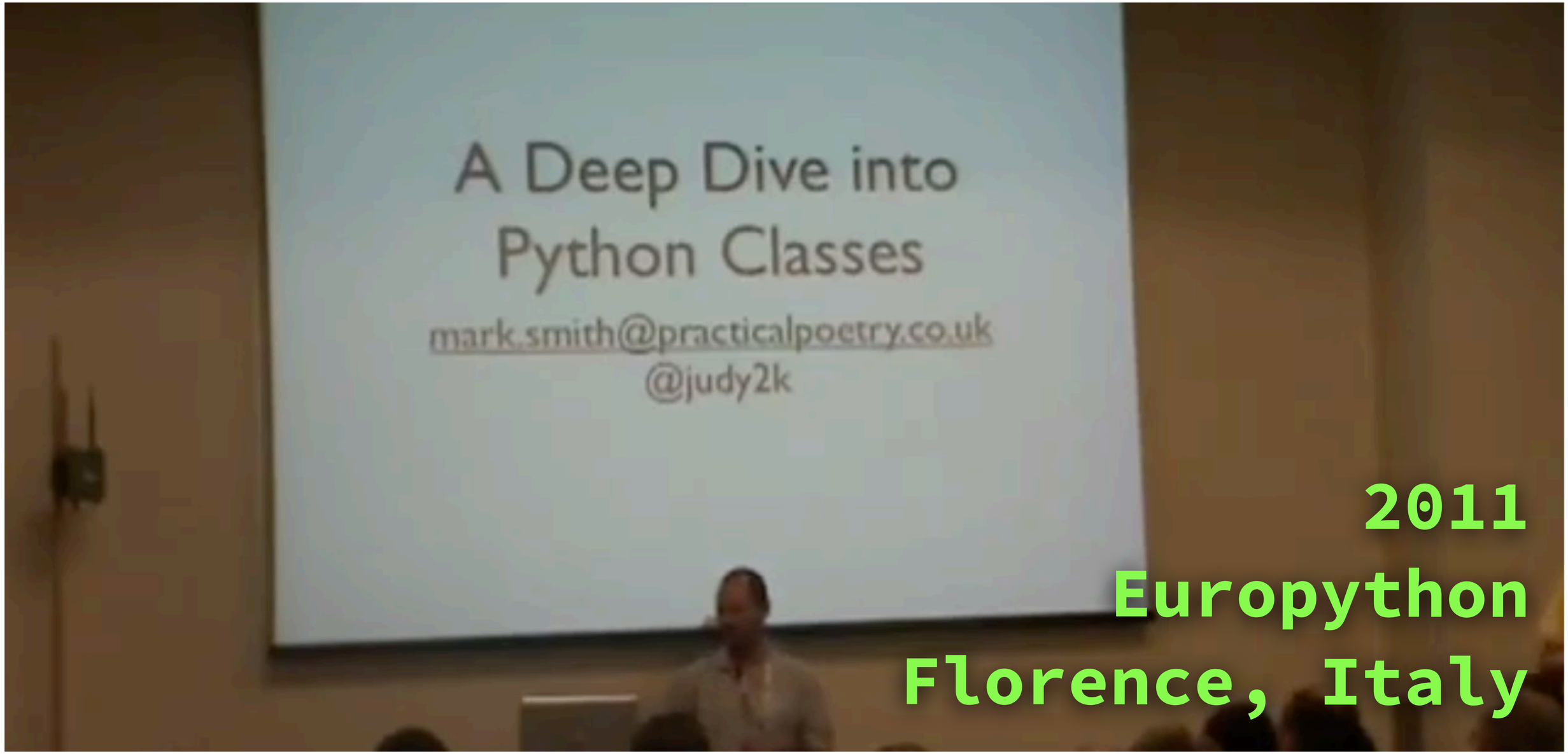
Mark Smith
Developer Advocate
Nexmo

nexmo®

The Vonage® API Platform




A Deep Dive into
Python Classes
mark.smith@practicalpoetry.co.uk
@judy2k



2011
Europython
Florence, Italy

Agenda

- Python Types
- Python Classes
- Python Metaclasses

Python Types

"In computer science and computer programming, a data type or simply *type* is an attribute of data which tells the compiler or interpreter how the programmer intends to use the data.

A data type **constrains** the values that an expression, such as a variable or a function, might take. This data type **defines** **the operations** that can be done on the data, the **meaning** **of the data,** and the way values of that type can be stored."

Wikipedia

Capabilities

Capabilities

Constraints

Capabilities

Constraints

Meaning

Capabilities

Capabilities

```
>>> 12 + 12
```

Capabilities

```
>>> 12 + 12
24
```

Capabilities

```
>>> 12 + 12
24

>>> dir(int)
```

Capabilities

```
>>> 12 + 12
24

>>> dir(int)
[..., '__add__', ...]
```

Capabilities

```
>>> 12 + 12
24

>>> dir(int)
[..., '__add__', ...]

>>> a = 12
>>> a.__add__(12)
```

Capabilities

```
>>> 12 + 12
24

>>> dir(int)
[..., '__add__', ...]

>>> a = 12
>>> a.__add__(12)
24
```

Constraints

Constraints

```
>>> 12 + "12"
```

Constraints

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

Meaning

Meaning

```
>>> type(d)
```

Meaning

```
>>> type(d)
<class 'datetime.datetime'>
```

Meaning

```
>>> type(d)
<class 'datetime.datetime'>

>>> type(p)
```

Meaning

```
>>> type(d)
<class 'datetime.datetime'>

>>> type(p)
<class 'builtin_function_or_method'>
```

Constraints & Meaning

Constraints & Meaning

```
"12" + 12
```

Constraints & Meaning

```
"12" + 12
```

Constraints & Meaning

```
"12" + 12

a, b = read_two_items_from_database()
```

Constraints & Meaning

```
"12" + 12

a, b = read_two_items_from_database()
a + b
```

Constraints & Meaning

```
"12" + 12

a, b = read_two_items_from_database()
a + b
"1212"
```

Python's Core Types

bool

int

float

complex

bytes

str

function

class

Python's Core Types

```
>>> type(True)
<class 'bool'>

>>> type(1.1)
<class 'float'>

>>> type(2 + 3j)
<class 'complex'>

>>> type(b"52s")
<class 'bytes'>
```

```
>>> type("How long am
I?")
<class 'str'>
```

@judy2k

Functions & Classes

Functions & Classes

```
>>> def fun():
...     pass
```

Functions & Classes

```
>>> def fun():
...     pass

>>> type(fun)
<class 'function'>
```

Functions & Classes

```
>>> def fun():
...     pass

>>> type(fun)
<class 'function'>
```

```
>>> class War:
...     pass
```

Functions & Classes

```
>>> def fun():
...     pass

>>> type(fun)
<class 'function'>
```

```
>>> class War:
...     pass

>>> type(War)
<class 'type'>
```

Functions & Classes

```
>>> def fun():
...     pass

>>> type(fun)
<class 'function'>
```

```
>>> class War:
...     pass

>>> type(War)
<class 'type'>
```

Wait, wat?

```
>>> type(War)
<class 'type'>

>>> type
<class 'type'>

>>> type(type)
<class 'type'>
```

War & type are classes

```
>>> type(War) == (type(type))
True
```

Capabilities

&

Constraints

Classes are mutable

Classes are mutable

```
>>> War.useful_value = "abcde"
```

Classes are mutable

```
>>> War.useful_value = "abcde"

>>> dir(War)
['__class__', ..., '__weakref__',
'useful_value']
```

Classes are mutable

Classes are mutable

```
>>> type.useful_value = "abcde"
```

Classes are mutable

```
>>> type.useful_value = "abcde"

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/
extension type 'type'
```

NOT ALL Classes are mutable

```
>>> type.useful_value = "abcde"

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/
extension type 'type'
```

Questions?

Python Classes

What is a class?

A class is instructions for creating an *instance*

And a class provides behaviour

A class is a *thing* for constructing instances.

A Simple Example

```
class Car:
    def __init__(self, color):
        self._color = color

    def drive(self):
        print("You are driving the car")

my_car = Car('red')
```

The Instance

The Instance

```
>>> my_car
<__main__.Car object at 0x10f71fc50>
```

The Instance

```
>>> my_car
<__main__.Car object at 0x10f71fc50>

>>> dir(my_car)
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_color', 'drive']
```

Inside the Instance

Inside the Instance

```
>>> my_car.__dict__
{'_color': 'red'}
```

Inside the *Class*

Inside the *Class*

```
>>> my_car.__class__
<class '__main__.Car'>
```

Inside the *Class*

```
>>> my_car.__class__
<class '__main__.Car'>

>>> my_car.__class__.__dict__
mappingproxy({
    '__module__': '__main__',
    '__init__': <function Car.__init__>,
    'drive': <function Car.drive>,
    '__dict__': <attribute '__dict__' of 'Car' objects>,
    '__weakref__': <attribute '__weakref__' of 'Car'
objects>,
    '__doc__': None})
```

@judy2k

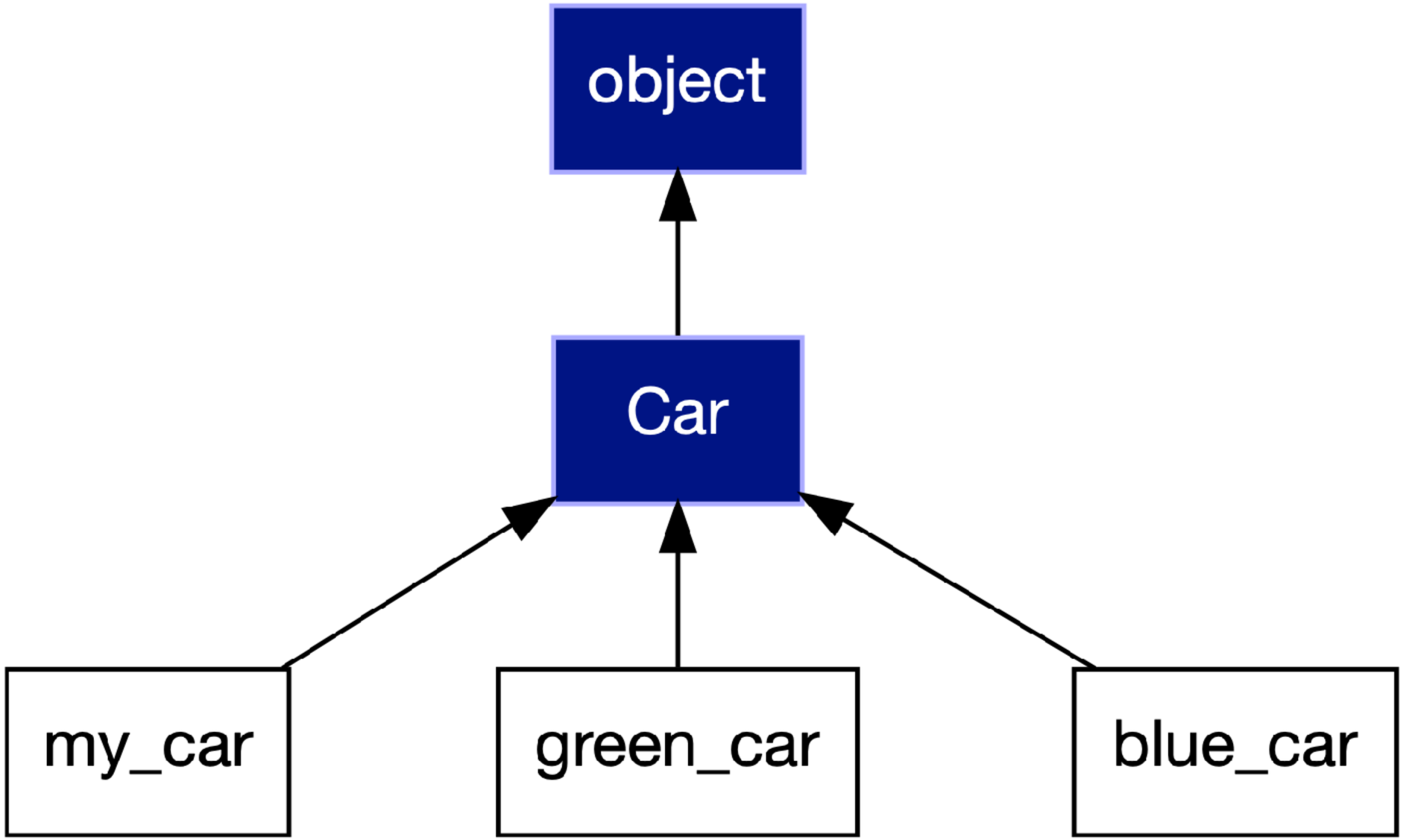
Inside *object*

Inside *object*

```
>>> my_car.__class__.__bases__
(<class 'object'>,)
```

Inside *object*

```
>>> my_car.__class__.__bases__
(<class 'object'>,)

>>> object.__dict__.keys()
dict_keys(['__repr__', '__hash__', '__str__',
'__getattribute__', '__setattr__', '__delattr__',
'__lt__', '__le__', '__eq__', '__ne__', '__gt__',
'__ge__', '__init__', '__new__', '__reduce_ex__',
'__reduce__', '__subclasshook__',
'__init_subclass__', '__format__', '__sizeof__',
'__dir__', '__class__', '__doc__'])
```


object
Car
my_car
green_car
blue_car

Defining Classes

```
class ClassName(Parent1, ...):
```

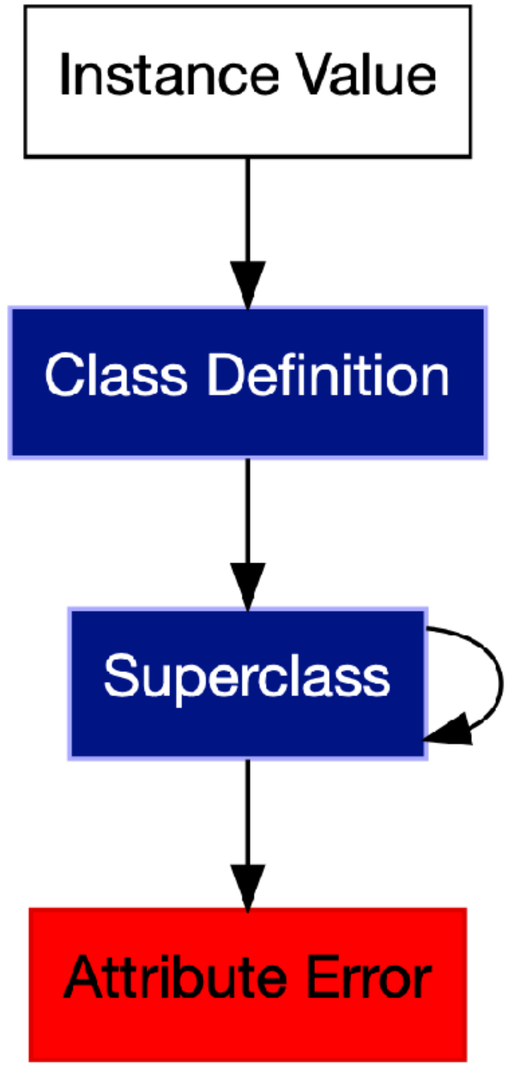
```
def __new__(
    cls,
    ...):
```

```
def __init__(
    self,
    ...):
```

Inheritance

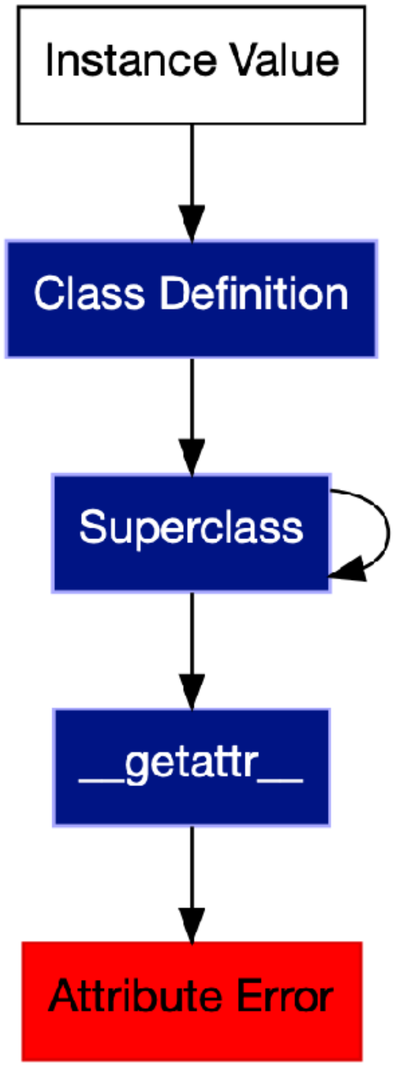
What happens when you get an attribute?

Attribute Lookup



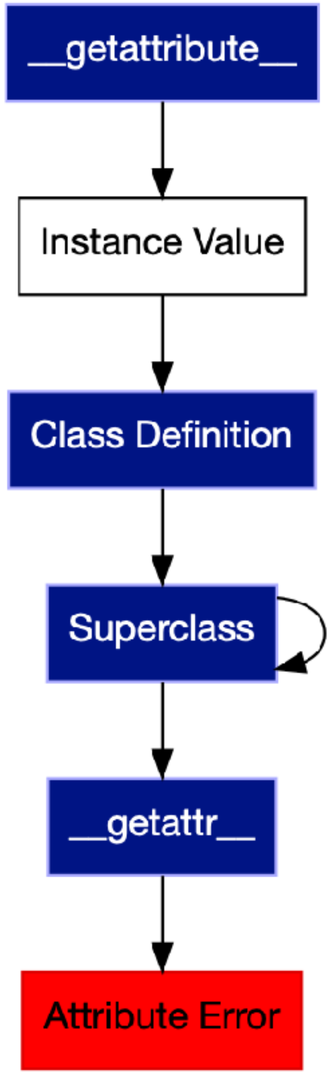
__getattr__(
self,
name)

Attribute Lookup

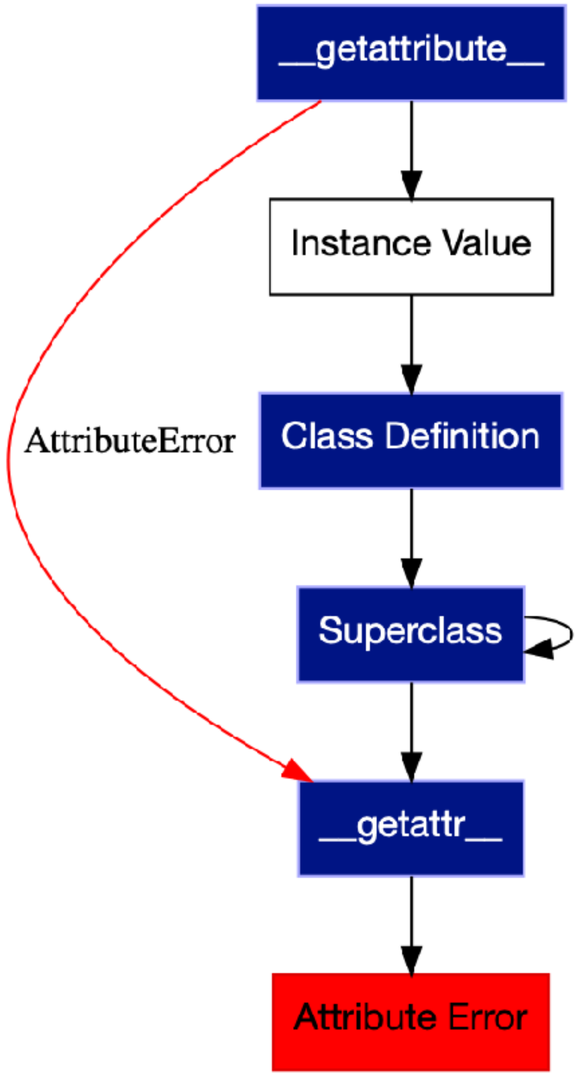


```
__getattribute__(
    self,
    name)
```

Attribute Lookup



Attribute Lookup

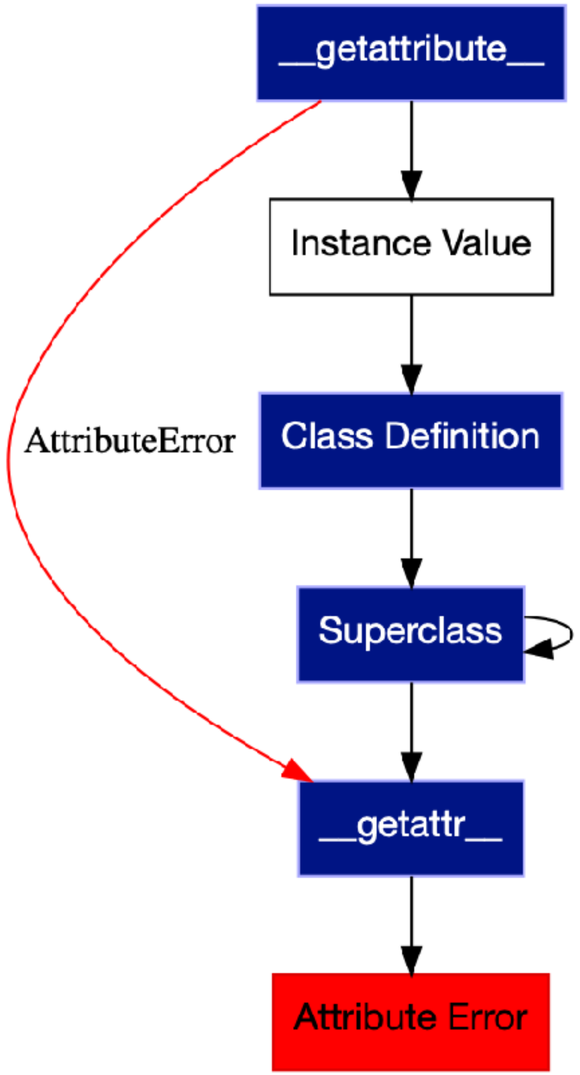


inspect.**getmro**()

super()

Descriptors

Attribute Lookup



What Descriptors Are *For*

- @property
- @classmethod
- @staticmethod

A descriptor is an object attribute with “binding behavior”, **one whose attribute access has been overridden** by methods in the descriptor protocol.

Those methods are `__get__()`, `__set__()`, and `__delete__()`.

If any of those methods are defined for an object, it is said to be a descriptor.

– Raymond Hettinger
Descriptor HowTo Guide

Example Descriptor

Example Descriptor

```
class SimpleDescriptor:
    def __get__(self, instance, owner):
        return f"You called __get__ with: {self!r}, {instance!r}, {owner!r}"
```

Example Descriptor

```
class SimpleDescriptor:
    def __get__(self, instance, owner):
        return f"You called __get__ with: {self!r}, {instance!r}, {owner!r}"


class JustAnOrdinaryClass:
    get_me = SimpleDescriptor()
```

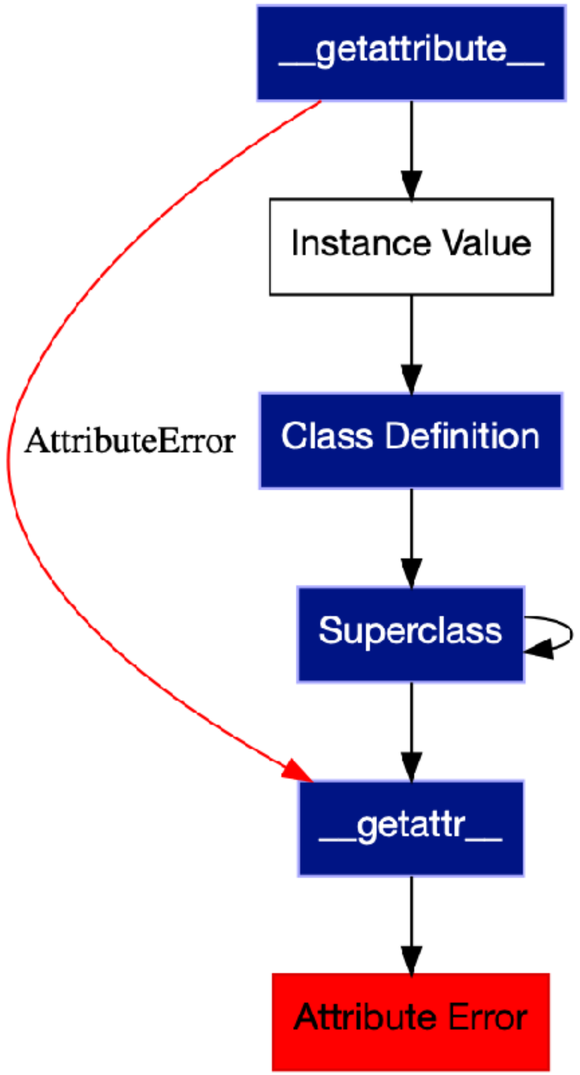
Example Descriptor

```
class SimpleDescriptor:
    def __get__(self, instance, owner):
        return f"You called __get__ with: {self!r}, {instance!r}, {owner!r}"


class JustAnOrdinaryClass:
    get_me = SimpleDescriptor()


>>> an_instance = JustAnOrdinaryClass()
```

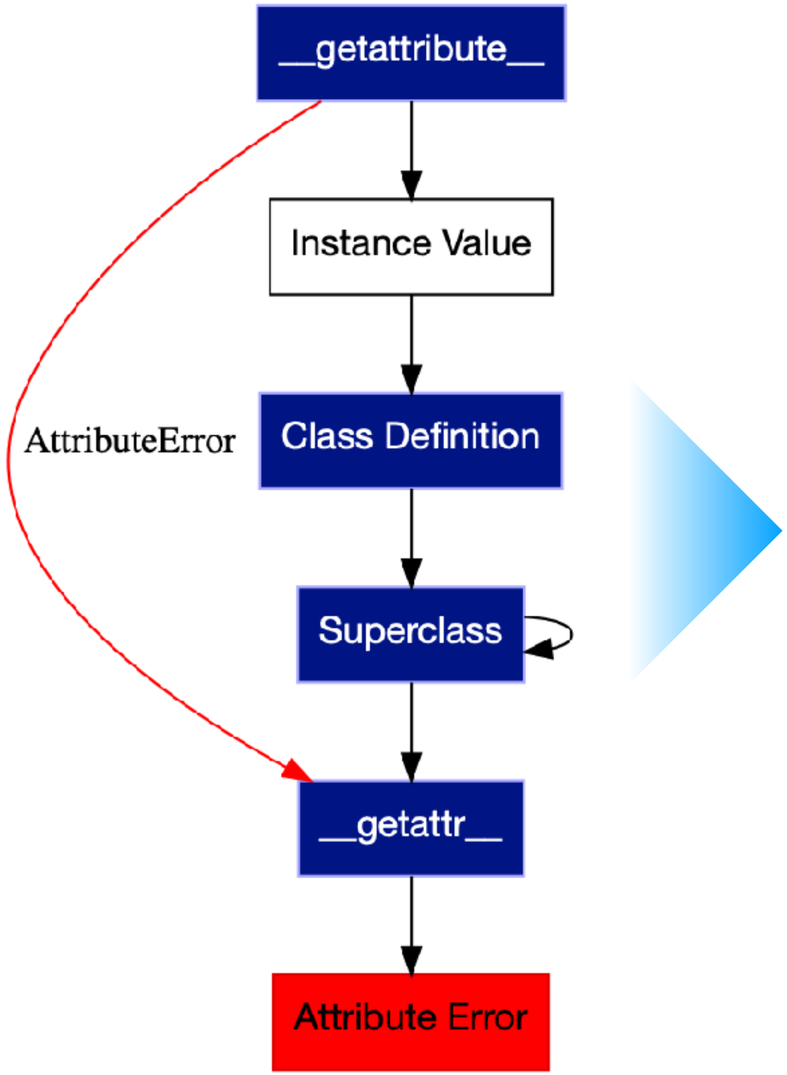
Example Descriptor

```
class SimpleDescriptor:
    def __get__(self, instance, owner):
        return f"You called __get__ with: {self!r}, {instance!r}, {owner!r}"


class JustAnOrdinaryClass:
    get_me = SimpleDescriptor()


>>> an_instance = JustAnOrdinaryClass()
>>> an_instance.get_me
"You called __get__ with:
    <__main__.SimpleDescriptor object at 0x10a86b3c8>,
    <__main__.JustAnOrdinaryClass object at 0x10a86bb38>,
    <class '__main__.JustAnOrdinaryClass'>"
```

Attribute Lookup



Attribute Lookup



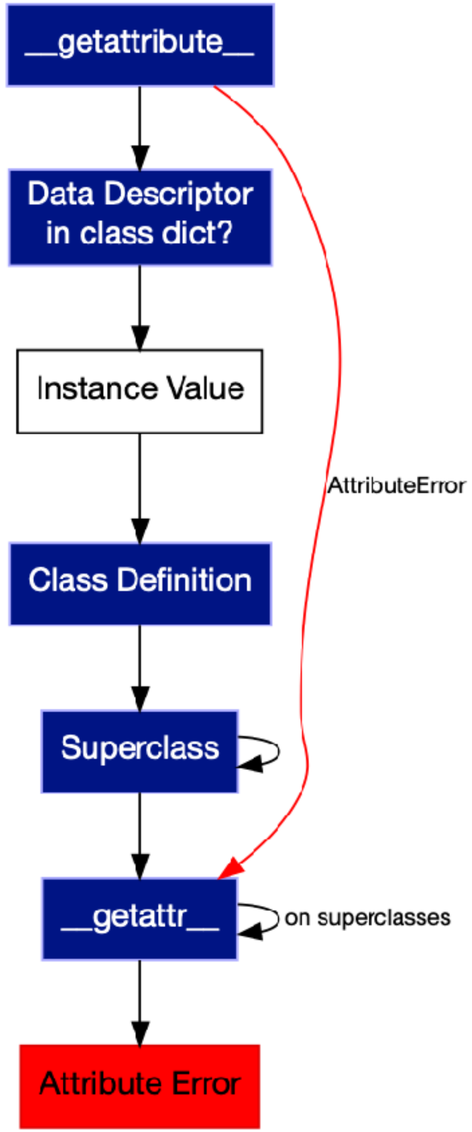
If it's a descriptor, call it

Data
vs.
Non-Data Descriptors

A *data* descriptor

```
class SimpleDescriptor:
    def __get__(self, instance, owner):
        return f"You called __get__ with: {self!r}, {instance!r}, {owner!r}"

    def __set__(self, instance, value):
        self._value = value
```

Attribute Lookup



If it's a descriptor, call it

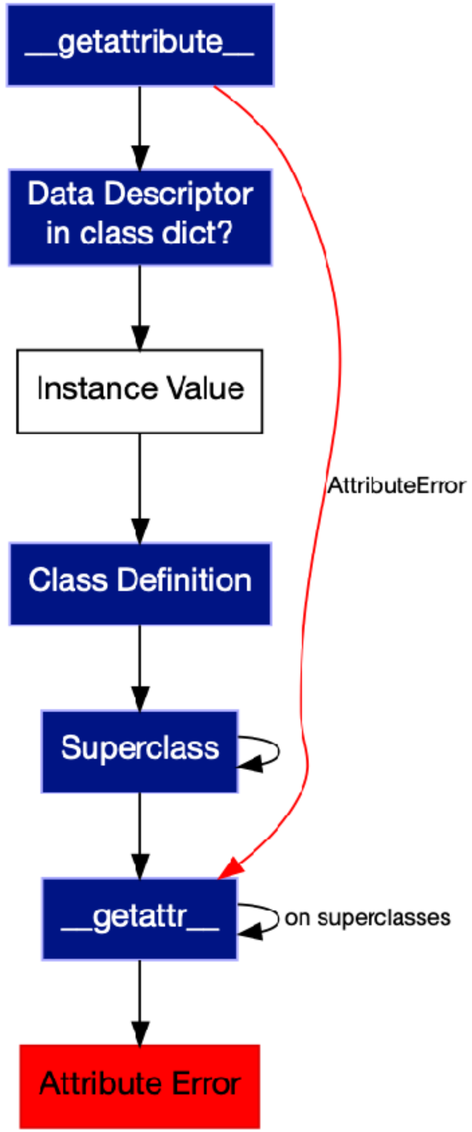
WHY???

Methods

What if *functions* were descriptors?

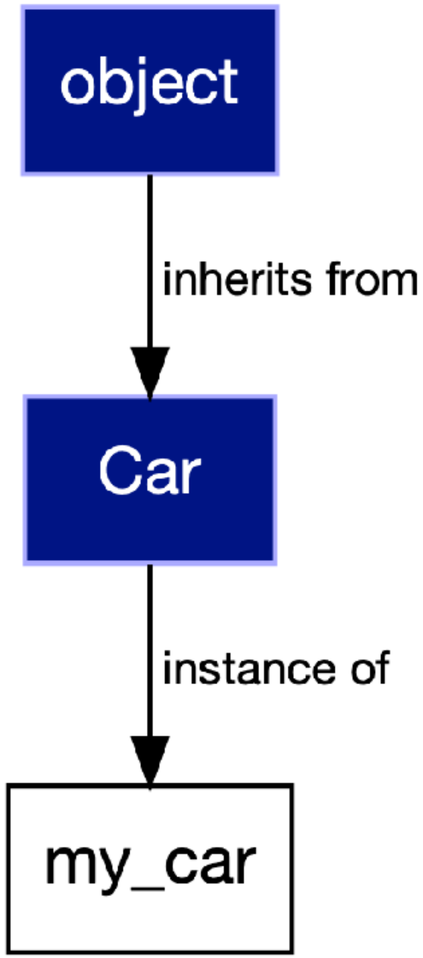
```
>>> Car.drive
<function Car.drive at 0x103ddd048>

>>> my_car.drive
<bound method Car.drive of <__main__.Car object at 0x103dd3470>>

>>> Car.drive.__get__(my_car, Car)
<bound method Car.drive of <__main__.Car object at 0x103dd3470>>
```

Attribute Lookup

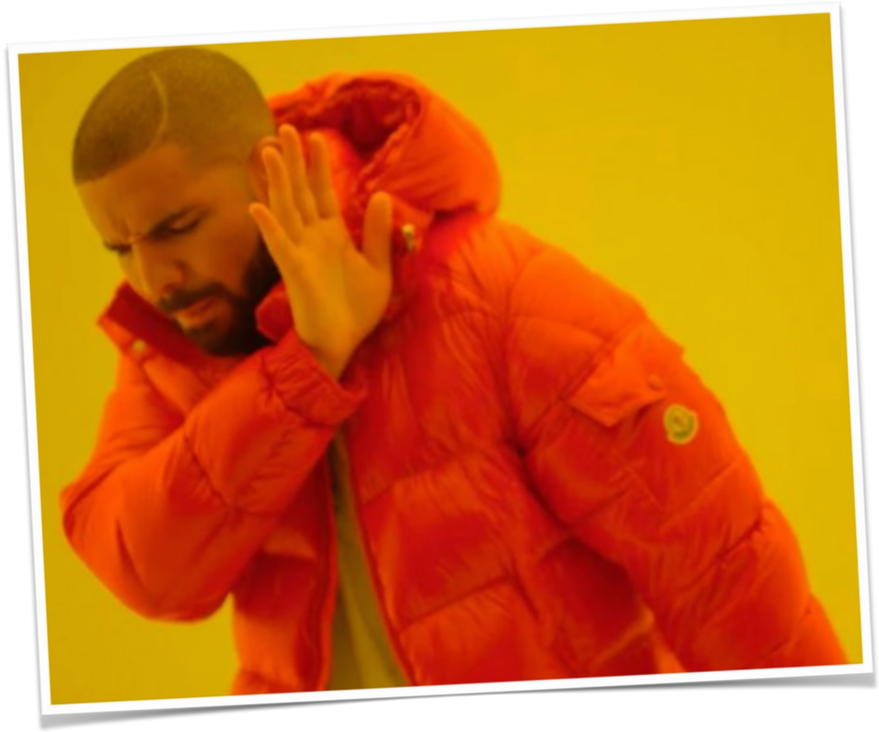


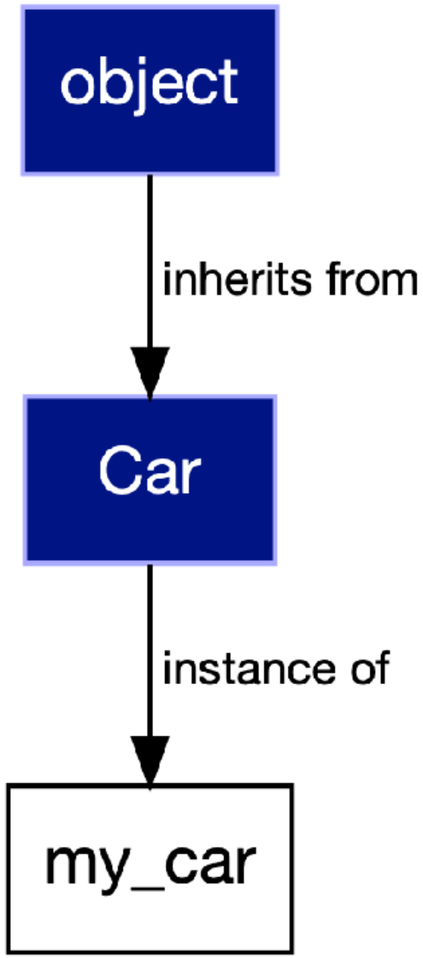
If it's a descriptor, call it

Inheritance

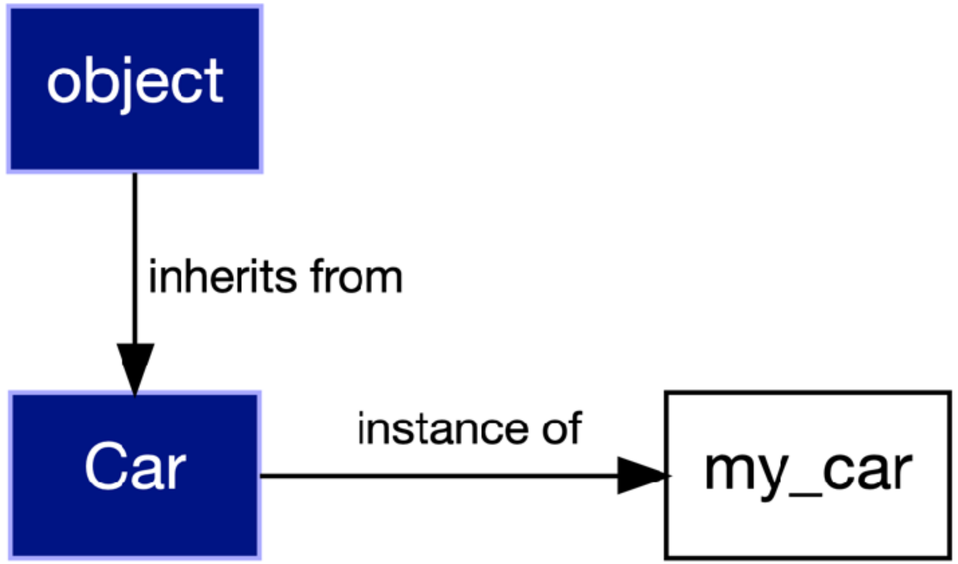


Inheritance

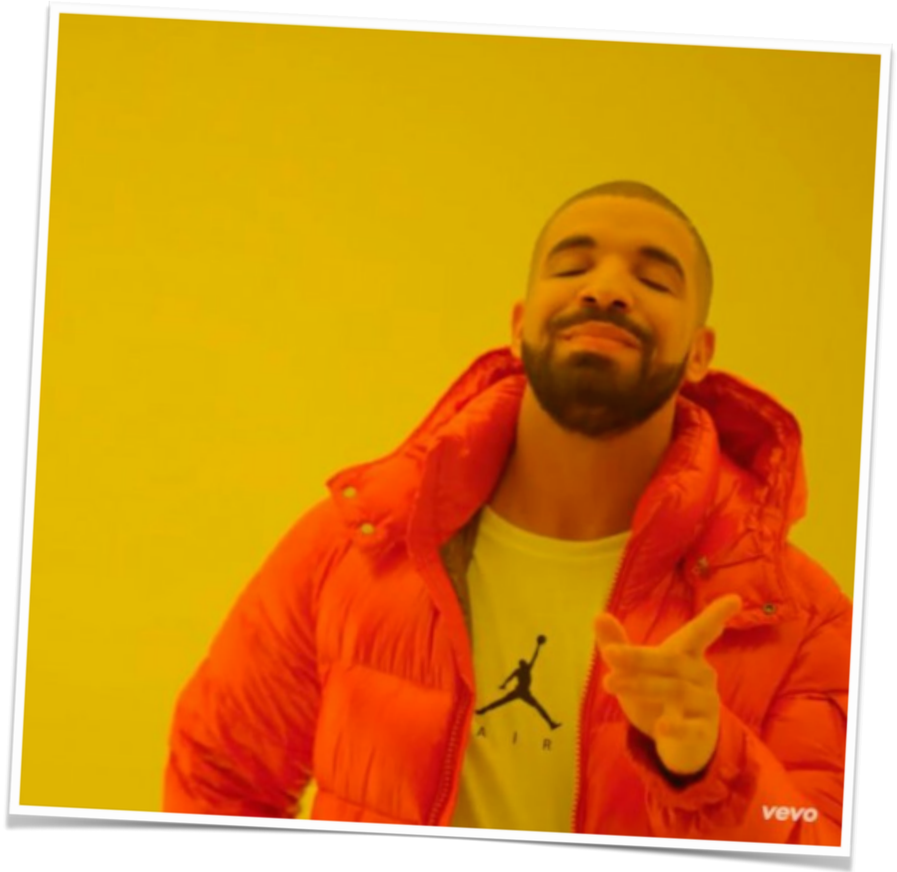


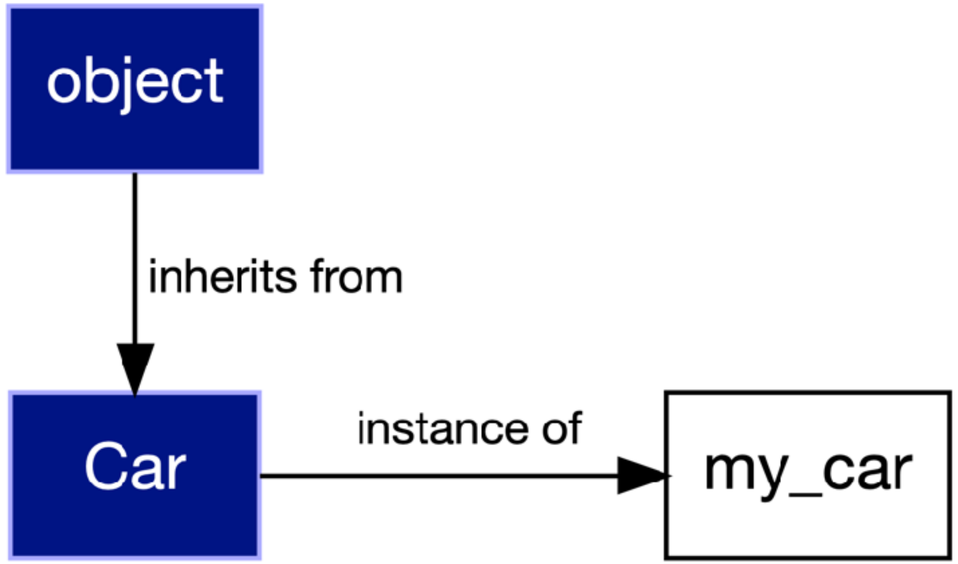


Inheritance



Inheritance





@judy2k

Questions?

Python Metaclasses

Uses for metaclasses

Uses for metaclasses

- Register a class on definition

Uses for metaclasses

- Register a class on definition
- Initialise attributes (usually to set a name)

Uses for metaclasses

- Register a class on definition
- Initialise attributes (usually to set a name)
- Modify a class, based on its definition

Uses for metaclasses

- Register a class on definition
- Initialise attributes (usually to set a name)
- Modify a class, based on its definition
- Ensure subclass implementation

Uses for metaclasses

- Register a class on definition
- Initialise attributes (usually to set a name)
- Modify a class, based on its definition
- Ensure subclass implementation
- Totally mess with the way a class behaves

Using *type* to construct instances.

Type Is Overloaded

```
# Get the type of an instance:
type(my_object) # -> <class MyClass>

# Create a NEW type!
type('ClassName', bases, classdict)
# -> <class ClassName>
```

type Example

```
def init_func(self, color):
        self._color = color

def drive(self):
    print("You are driving the car")

Car = type(
    'Car',
    (object,),
    {
        '__init__': init_func,
        'drive': drive,
    })

my_car = Car('red')
```

@judy2k

This is actually really powerful...

3 Mixins

```
class A:
    def a(self):
        print("You called a")

class B:
    def b(self):
        print("You called b")

class C:
    def c(self):
        print("You called c")
```

Create a class for each combo...

```
# Loop through AB, AC, BC ...
for parents in itertools.combinations([A, B, C], 2):
    classname = ''.join([c.__name__ for c in parents])
    globals()[classname] = type(classname, parents, {})
```

Test it out...

Test it out...

```
>>> AB.__bases__
(<class '__main__.A'>, <class '__main__.B'>)
```

Test it out...

```
>>> AB.__bases__
(<class '__main__.A'>, <class '__main__.B'>)
>>> my_ab = AB()
```

Test it out...

```
>>> AB.__bases__
(<class '__main__.A'>, <class '__main__.B'>)
>>> my_ab = AB()
>>> my_ab.a()
You called a
```

Test it out...

```
>>> AB.__bases__
(<class '__main__.A'>, <class '__main__.B'>)
>>> my_ab = AB()
>>> my_ab.a()
You called a
>>> my_ab.b()
You called b
```

Test it out...

```
>>> AB.__bases__
(<class '__main__.A'>, <class '__main__.B'>)
>>> my_ab = AB()
>>> my_ab.a()
You called a
>>> my_ab.b()
You called b
>>> my_ab.c()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AB' object has no attribute 'c'
```

A metaclass is a callable which returns a class.

A function as a metaclass

```
def stupid_metaclass(classname, bases, attrdict):
    return type(classname, bases, attrdict)


class MyClass(metaclass=stupid_metaclass):
    pass


>>> type(MyClass)
<class 'type'>
```

A metaclass is the class of a class.

Applying A Metaclass

```
class MyMeta(type):
    pass

class MyClass(metaclass=MyMeta):
    pass

instance = MyClass()
```

Defining Metaclasses

```
class MyMeta(type):
```

```
__prepare__(
        cls,
        name,
        bases,
        **kwargs)
```

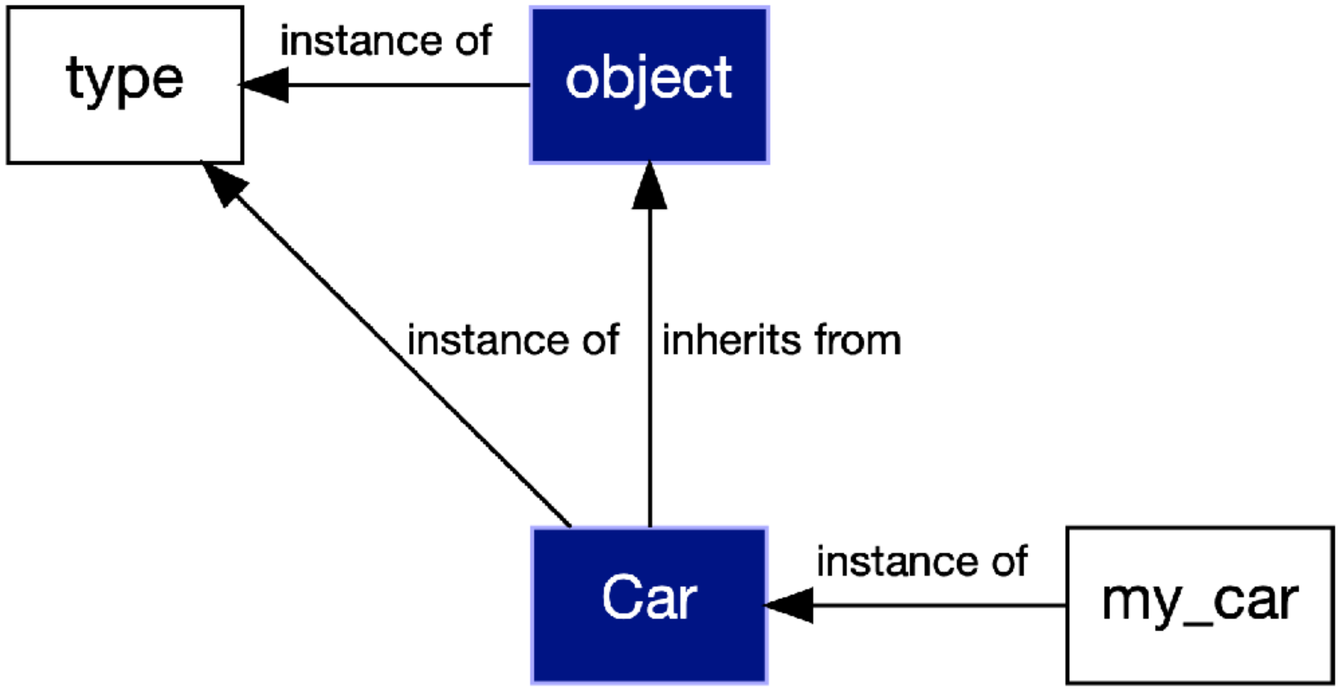
```
__new__(
    mcs,
    name,
    bases,
    classdict,
    **kwargs)
```

```
__init__(
    cls,
    name,
    bases,
    classdict,
    **kwargs)
```

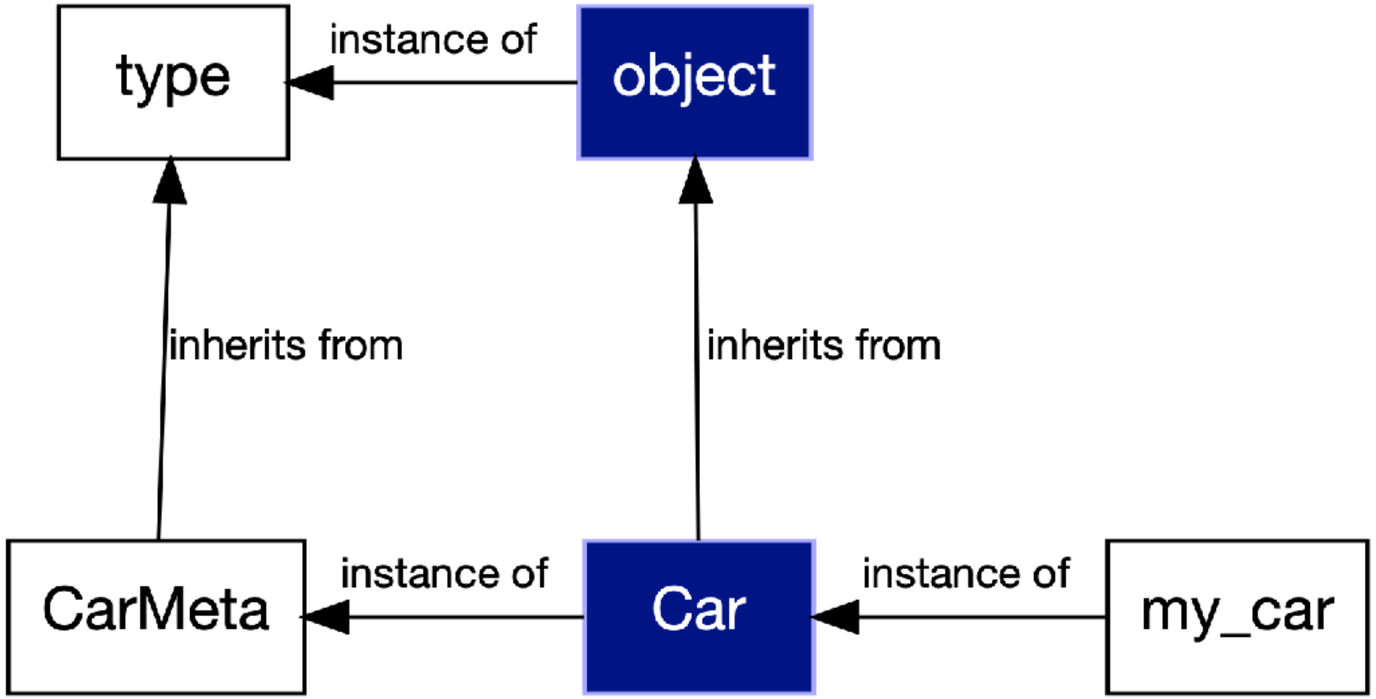
Keyword Arguments

```
class MyMeta(type):
    def __new__(self, classname, bases, attrdict, private):
        if private:
            # Do something clever here
            pass
        return super().__new__(
                self, classname, bases, attrdict)


class MyClass(metaclass=MyMeta, private=True):
    pass
```

Metaclasses & Inheritance



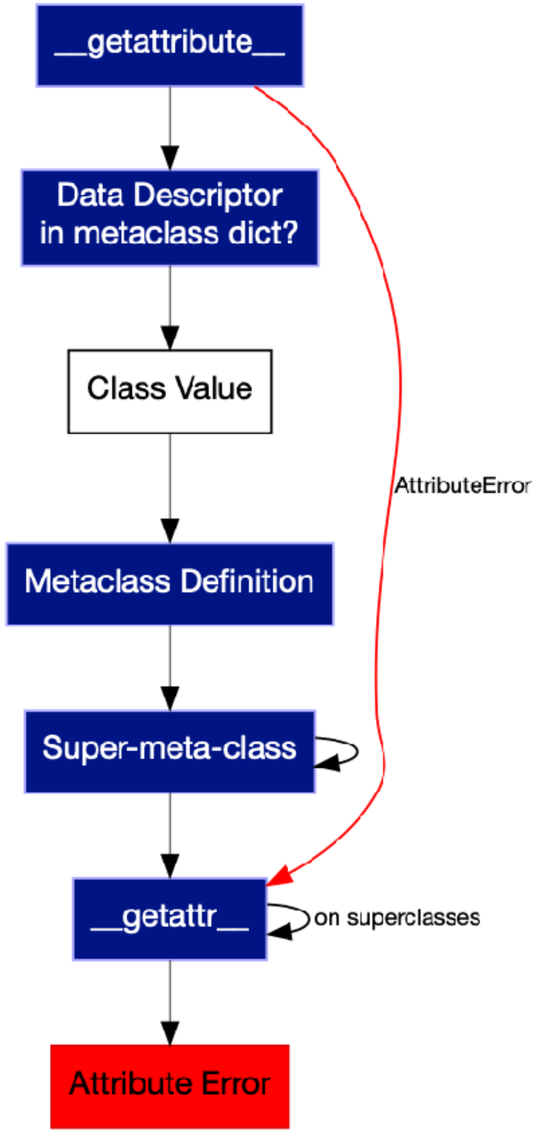
Metaclasses & Inheritance



Class Attribute Lookup

... is slightly different

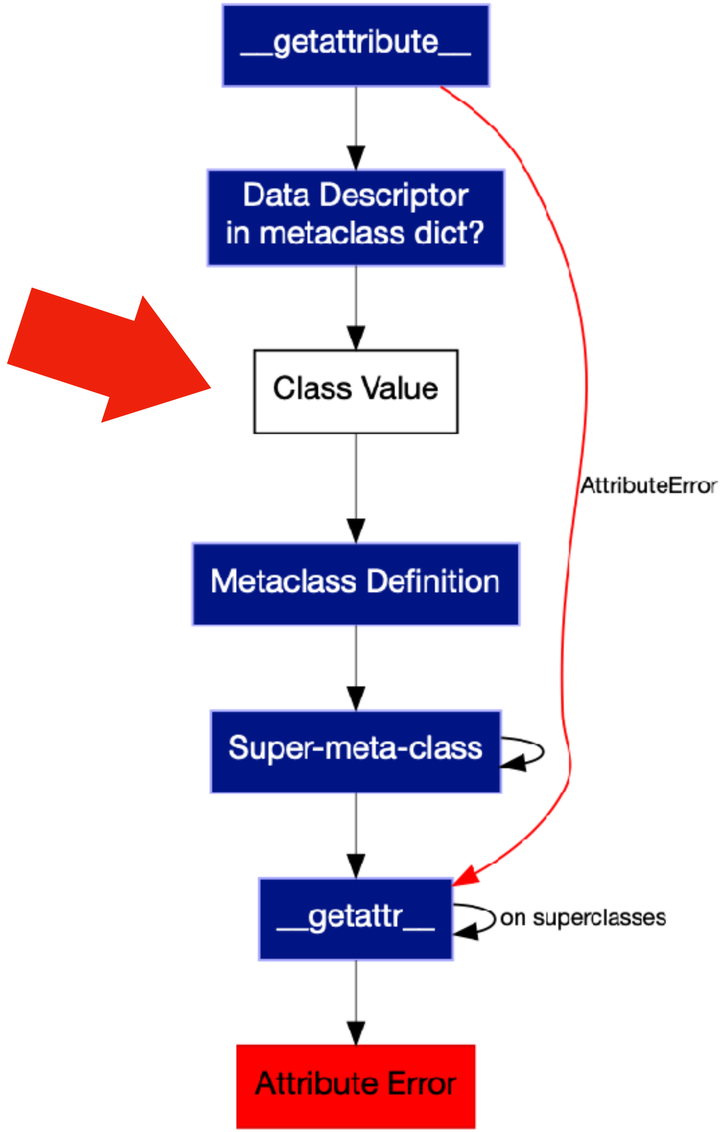
Class Attribute Lookup



If it's a descriptor, call it

Class Attribute Lookup

***Also* calls descriptors on the class**



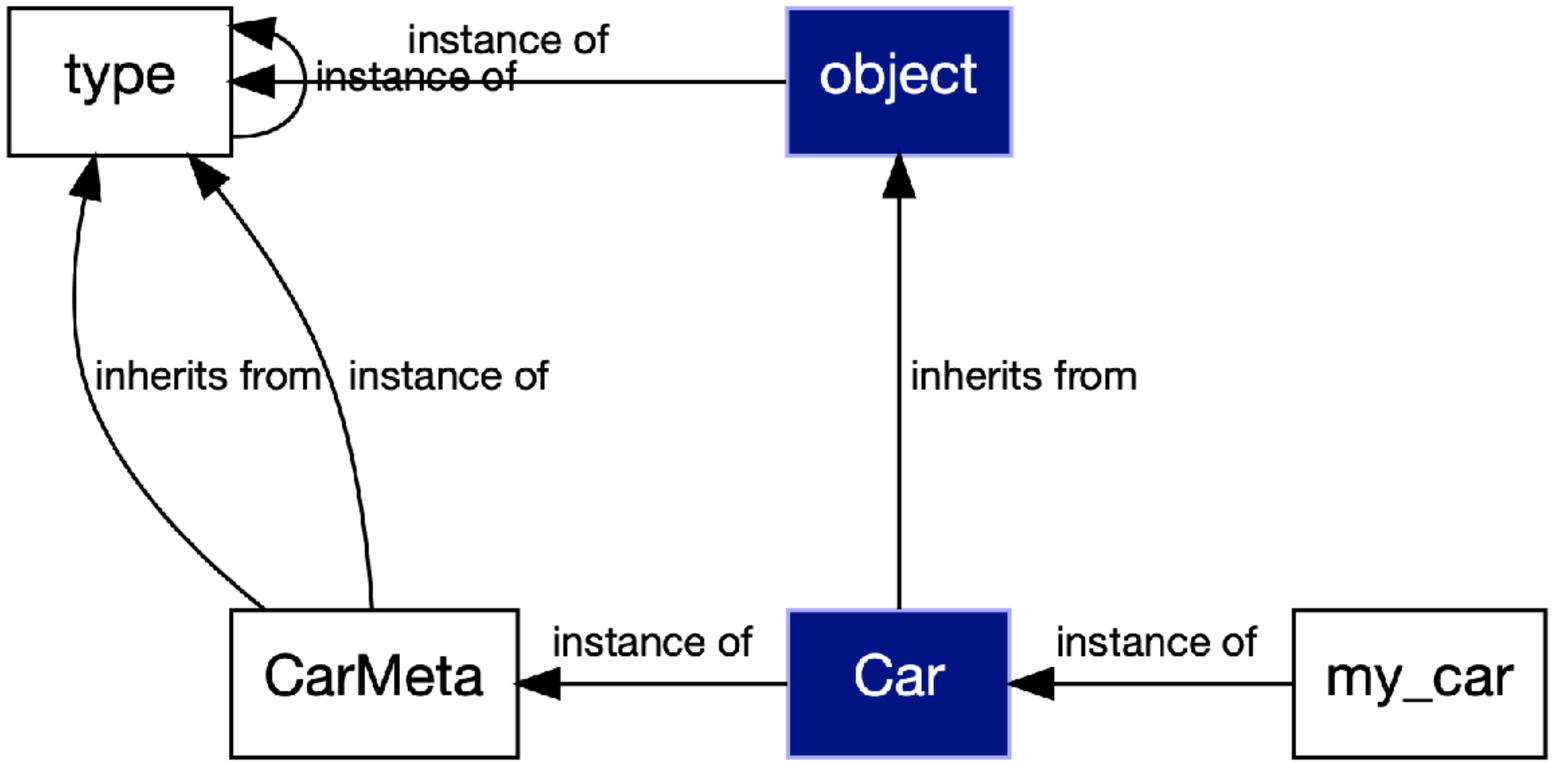
If it's a descriptor, call it

It's ~~python~~**type** all the way down

```
>>> type(CarMeta)
<class 'type'>

>>> type(type)
<class 'type'>

>>> type(type(type))
<class 'type'>
```

@judy2k

Instanception



@judy2k

mro() & __mro__

```
class AlphabeticMeta(type):
    def mro(self):
        default_mro = super().mro()
        return sorted(
            default_mro,
            key=attrgetter('__name__'))
```

__mro__

- __**mro**__ is calculated on *class creation*
- __**mro**__ is meant to be read-only
- __**mro**__ is recalculated whenever you update __**bases**__ on your class

@Judy2k

Metaclasses + Base classes

Examples

```
# Extend Model, get ModelBase metaclass:
class Employee(Model):
    pass


# Extend ABC, get ABCMeta metaclass:
class Driveable(ABC):
    pass
```

Metaclasses in the real world.

Uses for metaclasses

- Register a class on definition
- Initialise attributes (usually to set a name)
- Modify a class, based on its definition
- Ensure subclass implementation
- Totally mess with the way a class behaves (change inheritance, for example)

ABCMeta

Can't instantiate an abstract class

```
# Parent class:
class Driveable(abc.ABC):
    @abc.abstractmethod
    def drive(self):
        pass

# This is *also* abstract - no `drive` method:
class Car(Driveable):
    pass

my_car = Car()
# Traceback (most recent call last):
#   File "abc_example.py", line 13, in <module>
#     my_car = Car()
# TypeError: Can't instantiate abstract class Car
  with abstract methods drive
```

@judy2k

Implement abstractmethods

```
# Parent class:
class Driveable(abc.ABC):
    @abc.abstractmethod
    def drive(self):
        pass

class Car(Driveable):
    def drive(self):
        print("Driving!")

my_car = Car()
my_car.drive()
Driving!
```

Django

A Django Model

```
class FoodRating(models.Model):
    food_name = models.CharField(max_length=255)
    rating = models.IntegerField()
```

Register that the class exists

```
new_class._meta.apps.register_model(
    new_class._meta.app_label,
    new_class)
```

Set the name & model of each field attribute

```
class ModelBase(type):
    def __new__(cls, name, bases, attrs, **kwargs):
        ...
        for obj_name, obj in contributable_attrs.items():
            new_class.add_to_class(obj_name, obj)

class Model(metaclass=ModelBase):
   def add_to_class(cls, name, value):
        value.contribute_to_class(cls, name)

class Field:
    def contribute_to_class(cls, name):
        self.set_attributes_from_name(name)
        self.model = cls
```

@judy2k

```
objects
MyModel.DoesNotExist
MyModel.MultipleObjectsReturned
```

```
def subclass_exception(name, bases, module,
attached_to):
    return type(name, bases, {
          '__module__': module,
          '__qualname__': '%s.%s' %
(attached_to.__qualname__, name),
     })
```

Model -> Meta

```
class FoodRating(models.Model):
    food_name = models.CharField(max_length=255)
    rating = models.IntegerField()

    class Meta:
        ordering = ['-rating', 'food_name']
```

This is **not** a metaclass

Better than metaclasses

descriptor.__set_name__

Class decorators

Django Model without Metaclasses?

```
@model
class RankedFood:
    name = Field('VARCHAR(128)')
    score = Field('INT')
```

```
class Field:
    def __init__(self, sql_type):
        self._sql_type = sql_type

    def __set_name__(self, owner, name):
        self._name = name

    def sql_definition(self):
        return f"{self._name} {self._sql_type}"
```

Decorator Implementation

```
def create(cls):
    print(cls._fields())
    fields_part = ', '.join(field.sql_definition() for field in
cls._fields())
    return f"CREATE TABLE {cls.__name__} ({fields_part})"


def fields(cls):
    return [v for v in cls.__dict__.values() if isinstance(v, Field)]

# model decorator:
def model(cls):
    cls.create = classmethod(create)
    cls._fields = classmethod(fields)
    return cls
```

Using It

```
@model
class RankedFood:
    name = Field('VARCHAR(128)')
    score = Field('INT')


print(RankedFood.create())
CREATE TABLE RankedFoods (name VARCHAR(128),
score INT)
```

Code generation

In Conclusion

- **Inheritance** is complicated
- **Metaclasses** are not that simple or that complex
- **Descriptors** are more complicated than they may appear
- **Use** all of these features *sparingly*
- Don't be afraid of the **magic**!

Questions?

Slides & Code: bit.ly/ametaclass

Follow Me On Twitter: **@Judy2k**!