ECE326

PROGRAMMING LANGUAGES

Lecture 17 : Python Metaclass

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Metaclass

- A class whose instances are classes
- Defines behaviour of classes and their instances
- Similar to class decorator in purpose
- Differences
  - Class decorator
    - Rebinds class name to a callable
    - Can modify the existing class or add a wrapper around existing class
  - Metaclass
    - Inserts or routes specialized logic during class creation

type

- The base metaclass
  - Creates all classes, including metaclasses
  - All classes are instance of ‘type’
  - type(name, bases, attrs)

```
>> Foo = type(‘Foo’, (object, ), {‘a’ : 1})
>> Foo.a
1
>> type(Foo)
<class ‘type’>

# equivalent to this:
class Foo:
    a = 1
```

__class__

- The class type of the instance

```
>> Foo = type(‘Foo’, (object, ), {‘a’ : 1})
>> f = Foo()
>> type(f)
<class '__main__.Foo'>
>> f.__class__
<class '__main__.Foo'>
>> Foo
<class '__main__.Foo'>
>> f.__class__.__bases__
(<class 'object'>,)
>> f.__dict__
{'__module__': '__main__', 'count': 1, '__dict__': <attribute
'__dict__' of 'Foo' objects>, '__doc__': None, …}
```

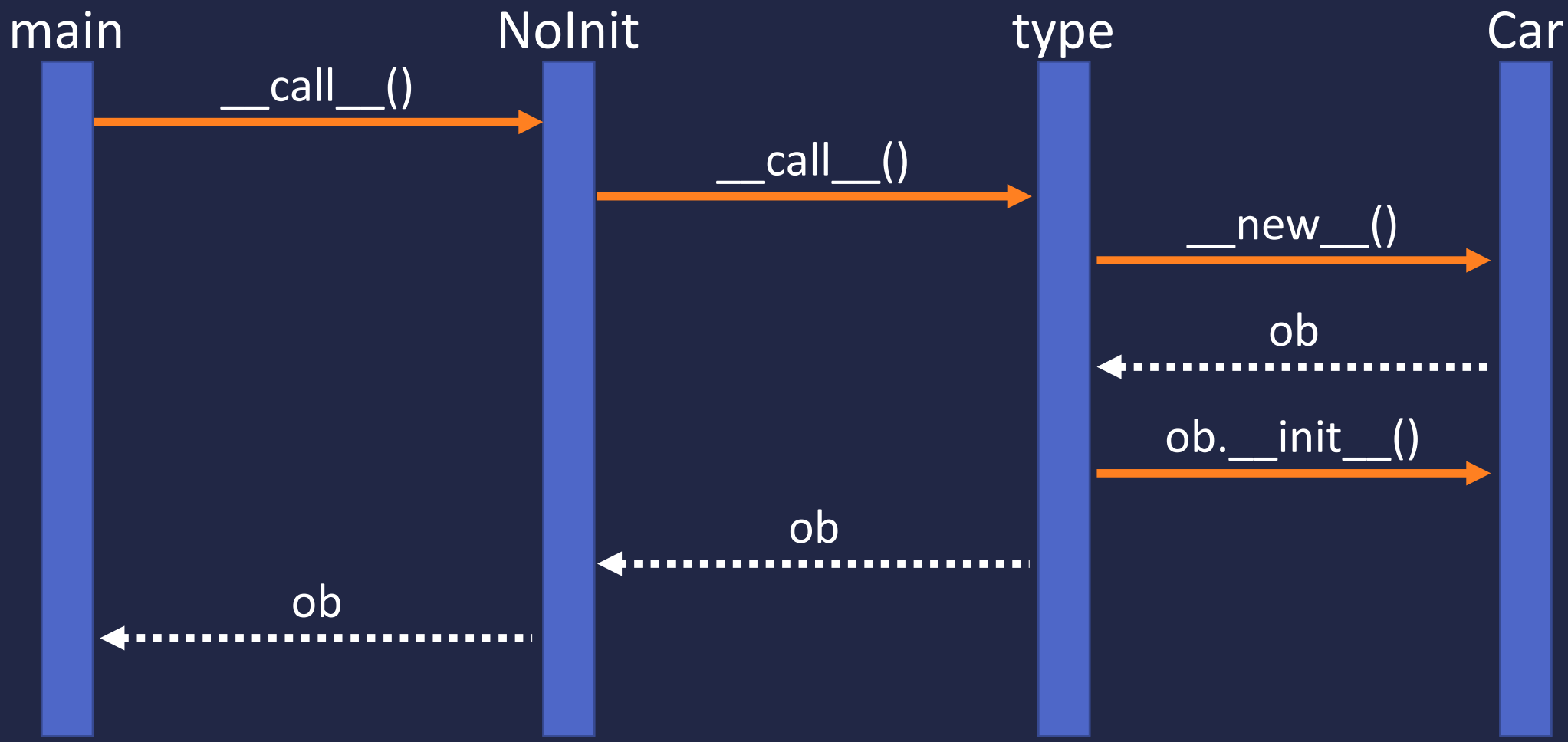
Metaclass

- Basic example
  - Class that does not need an __init__ method

```
class NoInit(type):             # metaclass inherits from type
    def __call__(cls, *pargs, **kwargs):
        inst = type.__call__(cls, *pargs)
        [ setattr(inst, k, v) for k, v in kwargs.items() ]
        return inst

class Car(metaclass=NoInit):
    def __str__(self):
        return " ".join([k+"="+v for k, v in vars(self).items()])

>> car = Car(make="Mazda", model="CX-5", year="2019", color="White")
>> str(car)
‘make=Mazda model=CX-5 year=2019 color=White’
>> type(car), type(Car)
(<class '__main__.Car'>, <class '__main__.NoInit'>)
```

type of *Car* is *NoInit*!

What Happened?



__new__

- The actual “constructor”
- __init__ is actually just an initializer
- Customizes instantiation of the object
- __new__(cls)
- Default implementation

```
class Foo:
    # __new__ is a static method (does not take Foo as 1st argument)
    def __new__(cls):
        # super() will eventually call object.__new__
        # object is the base class of all classes
        return super().__new__(cls)
```

Singleton

```
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 Factory(metaclass=Singleton):
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "<Factory: " + self.name + ">"

>> print(Factory("chocolate"))
<Factory: chocolate>
>> print(Factory("shoe"))
<Factory: chocolate>
```

Metaclass

- Can be used as parent to supply class methods

```
class Counter(type):
    def __init__(cls, name,
                 bases, attrs):
        cls.counter = 0

    def up(cls):
        cls.counter += 1

    def down(cls):
        cls.counter -= 1

    def __repr__(cls):
        return "<Counter: %d>"%(
            cls.counter)
```

```
class Animal(metaclass=Counter):
    def __init__(self, species):
        self.species = species
        Animal.up()

    def __del__(self):
        Animal.down()

def animal_test():
    a = Animal("monkey")
    print(repr(Animal))

>> animal_test()
<Counter: 1>
>> print(repr(Animal))
<Counter: 0>
```

Name Lookup

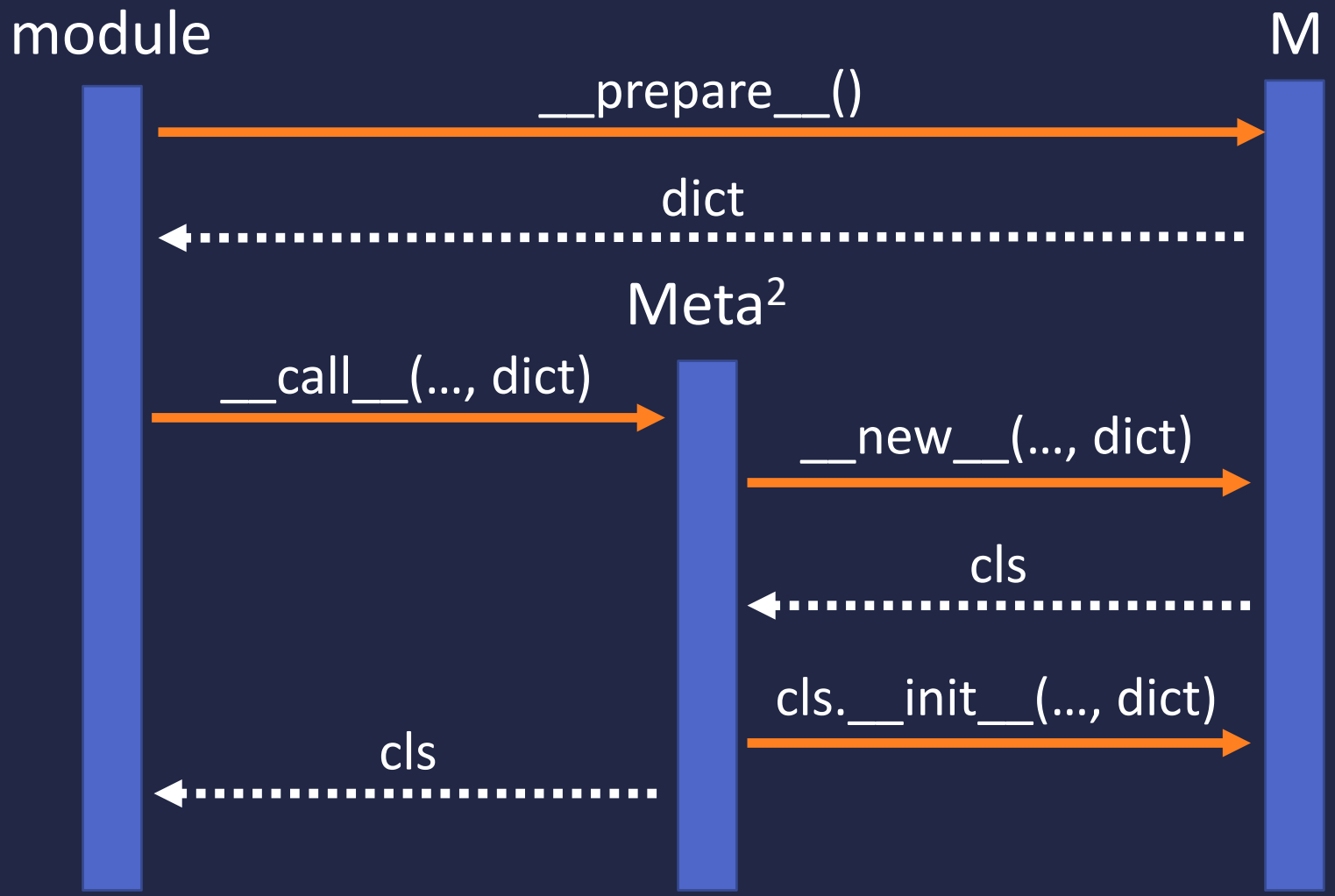
- An instance inherits its class's attributes
  - Also including attributes of super classes of its class
- A class inherits its metaclass's attributes
  - Also including attributes of its super classes
- But – an instance does not inherit metaclass attributes

```
>> a = Animal("horse")
>> a.up()                    # defined in Counter
AttributeError: 'Animal' object has no attribute 'up'
```

__del__

- Called when instance is about to be deleted
- Typically used to do additional clean-up
  - E.g. close log files
  - E.g. update global variables (such the counter)
  - E.g. release ownership of resources (such as cache entry)
- Careful
  - It is not guaranteed to be executed
  - It all depends on the garbage collector
- Do not confuse with __delete__ (used by descriptor)

Class Creation




```
class A(metaclass=M):
    a = 1
    def foo(self):
        pass
```

Meta² metametaclass

Metametaclass

- The metaclass of a metaclass
- Begins the process of creating a new class
  - Via __call__
  - In contrast, a metaclass's __call__ function initiates the process of creating a new instance
- Usually, `type` is the metaclass of other metaclasses
  - Unless metaclass is specified when defining a metaclass
  - Similar to instance creation, type.__call__ will execute __new__ and __init__ of the metaclass to create a new class

__prepare__

- Provides a dictionary-like object to store attributes
- By default, returns Python dictionary – dict()
- Exists for performance reasons
- Special features
  - E.g. ordered dictionary for fast lookup

```
class Meta(type):
    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        return {}
```

__new__ and __init__

- Constructor and initializer for the class

```
class Meta(type):
    …
    def __new__(mcs, name, bases, attrs, **kwargs):
        return super().__new__(mcs, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **kwargs):
        return super().__init__(name, bases, attrs)
```

- __init__ is rarely used since __new__ does more
  - However, it is useful when inheritance is involved
  - In contrast, class decorator cannot be subclassed

Metaclass Inheritance

- A derived class can have many base classes
- Each base class may have its own metaclass
- Subclasses inherits base class's metaclass
  - The inheritance tree of metaclass must be linear!

```
>> class Meta1(type): pass
>> class Meta2(type): pass
>> class Base1(metaclass=Meta1): pass
>> class Base2(metaclass=Meta2): pass
>> class Foobar(Base1, Base2): pass
TypeError: metaclass conflict: the metaclass of a derived
class must be a (non-strict) subclass of the metaclasses
of all its bases
```

Metaclass Inheritance

- Rationale
  - Each class can only have one metaclass
  - Resolving metaclass must be unambiguous
  - Most specialized metaclass is chosen

```
>> class Meta(type): pass
>> class SubMeta(Meta): pass
>> class Base1(metaclass=Meta): pass
>> class Base2(metaclass=SubMeta): pass
>> class Foobar(Base1, Base2): pass
>> type(Foobar)
<class '__main__.SubMeta'>
```

Name Resolution

- Metaclasses come *last* in name resolution order
  - After all super classes have been checked
    - i.e. done after method resolution order fails
  - Then metaclasses are checked, in reverse order of inheritance

```
>> class N(type): foo = 3              # metaclass
>> class M(N): attr = 1                # metaclass
>> class A: attr = 2                   # super class
>> class B(A): pass                    # super class
>> class C(B, metaclass=M): pass
>> inst = C()
>> inst.attr, C.attr, C.foo
(2, 2, 3)
```

Metafunction

- Metaclass only needs to be callable
- If __prepare__ is not defined, Python dict is used
- If inheritance not needed, can use a function!
  - Can also use a functor
- Caveat
  - The function is used as a metametaclass
  - The type is the type of the return value of the function

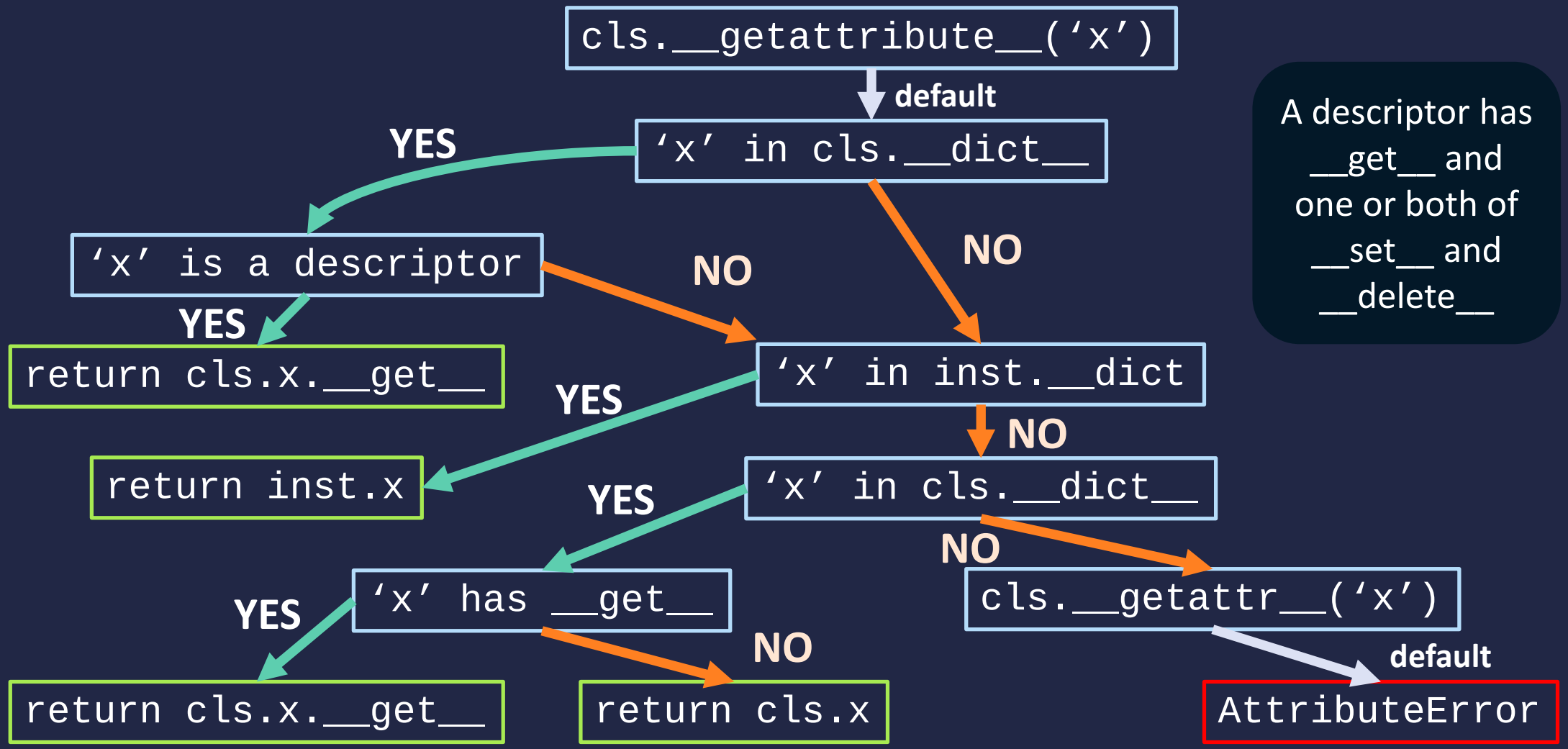
```
def MetaFunc(cls, bases, attrs):
    attrs["hello"] = 5
    return type(cls, bases, attrs)

class Foo(metaclass=MetaFunc):
    pass

>> Foo.hello
5
>> type(Foo)
<class ‘type’>
```

Operator Overloading

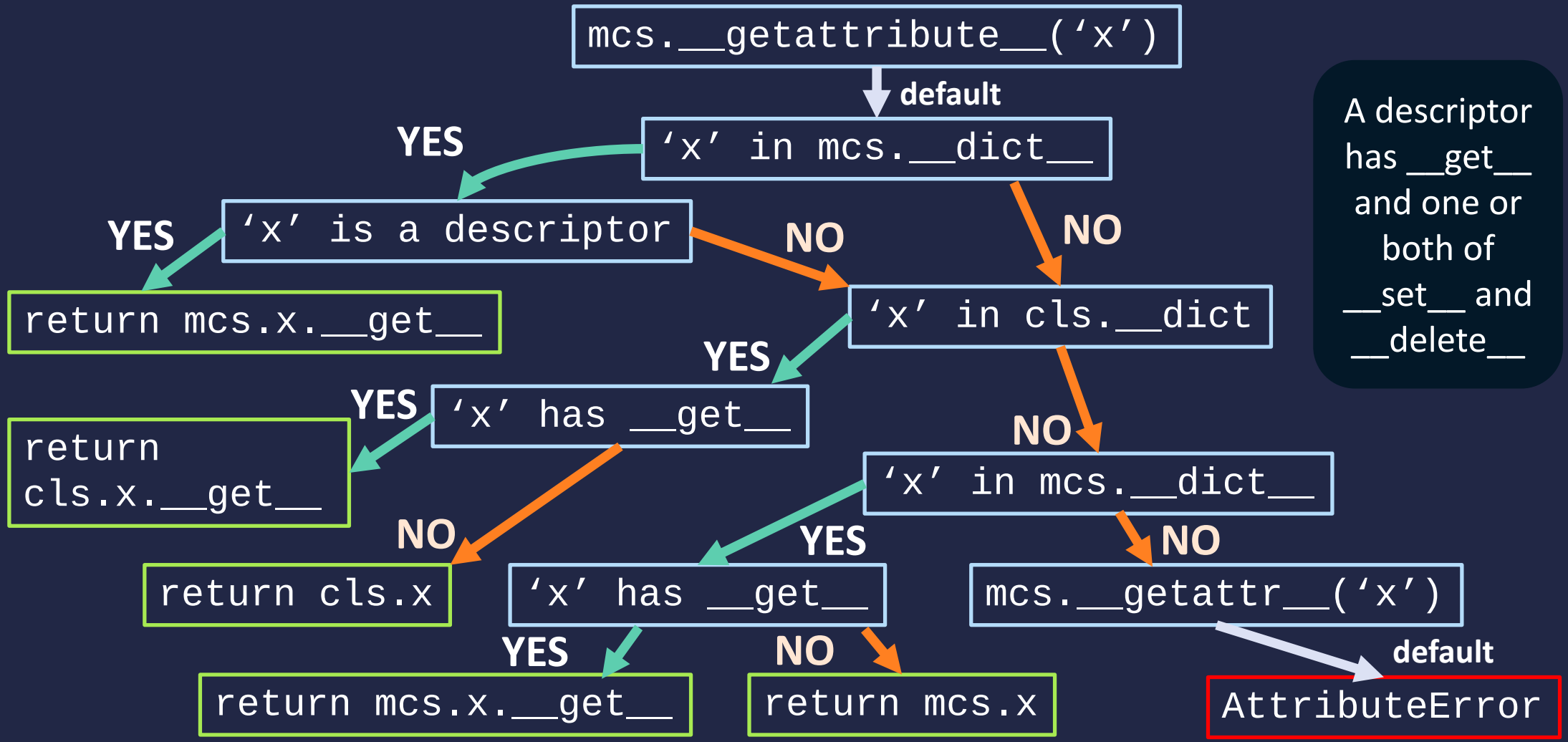
- With metaclasses, classes can also have their operators overloaded, similar to an instance

```
class A(type):
    def __getitem__(cls, i):
        return cls.data[i]
    def __getattr__(cls, name):
        return getattr(cls.data, name)

class B(metaclass=A):
    data = 'spam’

>> B[0]
‘s’
>> B.upper()
‘SPAM’
```

Object Attribute Lookup



Class Attribute Lookup



Type Slot

- A table of built-in methods
  - Operator overloading methods
    - E.g. __add__, __str__
  - Attribute interception methods
    - E.g. __getattr__, __setattr__
  - Attribute descriptors
- Look up for these methods go through *type slots*
  - Much simpler and faster
  - Not all built-in methods go through type slots
    - E.g. __prepare__