ECE326

PROGRAMMING LANGUAGES

Lecture 10 : Python Metaclass

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Metaclass

- A class whose instances are classes
  - Classes are blueprints for instances
  - Metaclasses are blueprints for classes
- *Not* the same as super class
- Defines behaviour of classes and their instances
- 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
```

Metaclass

- Inherits from `type`
- To use a metaclass, specify it as a keyword argument
- Class passed to MetaClass.__init__ after it is defined

```
class MetaClass(type):
    def __init__(cls, name, bases, attrs):
        print(“I created ” + name)

class Class(metaclass=MetaClass):
    pass

# same as this
# Class = type(“Class”, (), {…})
# MetaClass.__init__(Class, “Class”, (), {…})
```

Special Attributes

- *instance*.__class__
  - The class type of the instance

```
f.__class__ # shows <class '__main__.Foo'>
```

- *klass*.__name__
  - The name of the class

```
Foo.__name__ # shows ‘Foo’
```

- *klass*.__bases__
  - A list of all the base classes for a class

```
Foo.__bases__ # shows (<class 'object'>,)
```

```
class Foo:
    pass

f = Foo()
```

object is the base class of all classes!

instance.__new__

- The actual “constructor”
- __init__ is 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)
    # e.g. to call this function, you need to write Foo.__new__(Foo)
    def __new__(cls, *pa, **kwa):
       return object.__new__(cls)
```

instance.__del__

- Called when object is about to be deleted
- Typically used to do additional clean-up
  - E.g. close log files
  - E.g. update global variables
  - E.g. release ownership of resources (such as cache entry)
- Careful
  - It is not guaranteed to be executed
    - Unless you explicitly use the **del** operator
- Do not confuse with __delete__ (used by descriptor)

Metaclass

- Can be used as parent to supply class methods
  - Like how instance methods are defined in class body
- Name lookup rule
  - An instance can access its class’s attributes
    - Also including attributes of super classes of its class
  - A class can access its metaclass’s attributes
    - Also including attributes of its super classes
  - But – an instance cannot access metaclass attributes

Operator Overloading

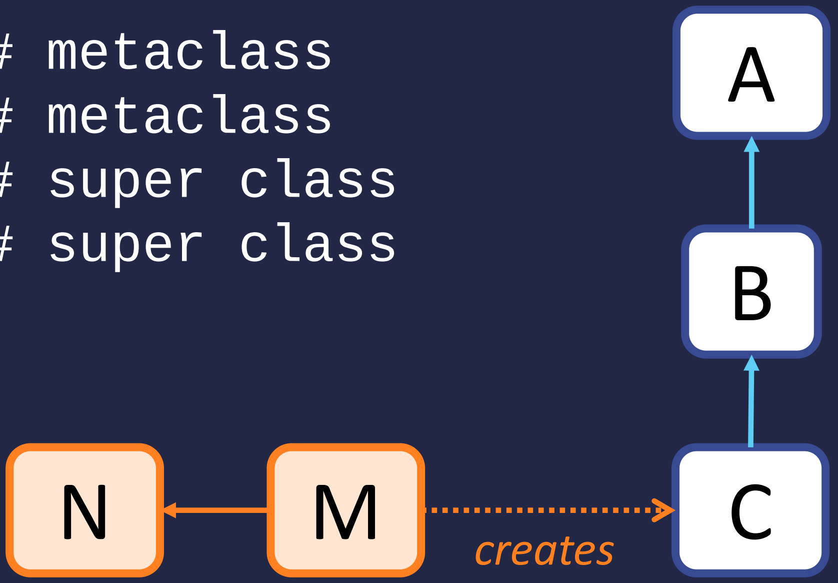
- Metaclass can overload the operators of their classes
  - Works same as overloading operators for instances

```
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’
```

Name Resolution

- Metaclasses come *last* in name resolution order
  - After all super classes have been checked
  - Then metaclasses are checked, in reverse order of inheritance

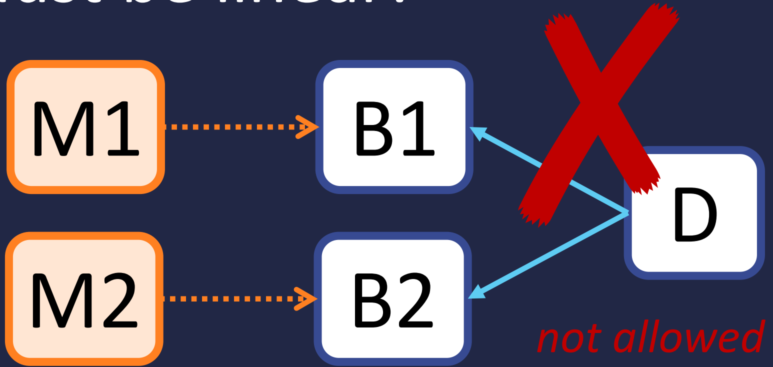
```
>> class N(type): y = 3             # metaclass
>> class M(N): x = 1                # metaclass
>> class A: x = 2                   # super class
>> class B(A): pass                 # super class
>> class C(B, metaclass=M): pass
>> inst = C()
>> inst.x, C.x, C.y
(2, 2, 3)
```



Metaclass Inheritance

- Derived class can have many base classes
  - Each base class may have its own metaclass
- Derived class inherits base class’s metaclass
  - The inheritance tree of metaclass must be linear!

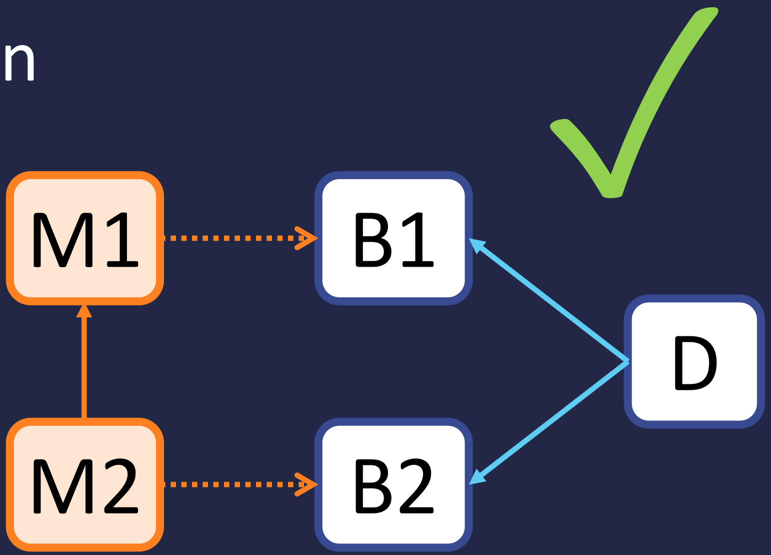
```
>> class M1(type): pass
>> class M2(type): pass
>> class B1(metaclass=M1): pass
>> class B2(metaclass=M2): pass
>> class D(B1, B2): pass
TypeError: metaclass conflict: the metaclass of a derived
class must be a (non-strict) subclass of the metaclasses
of all its bases
```



Metaclass Inheritance

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

```
>> class M1(type): pass
>> class M2(M1): pass
>> class B1(metaclass=M1): pass
>> class B2(metaclass=M2): pass
>> class D(B1, B2): pass
>> type(D)
<class '__main__.M2'>
```



Metaclass.__call__

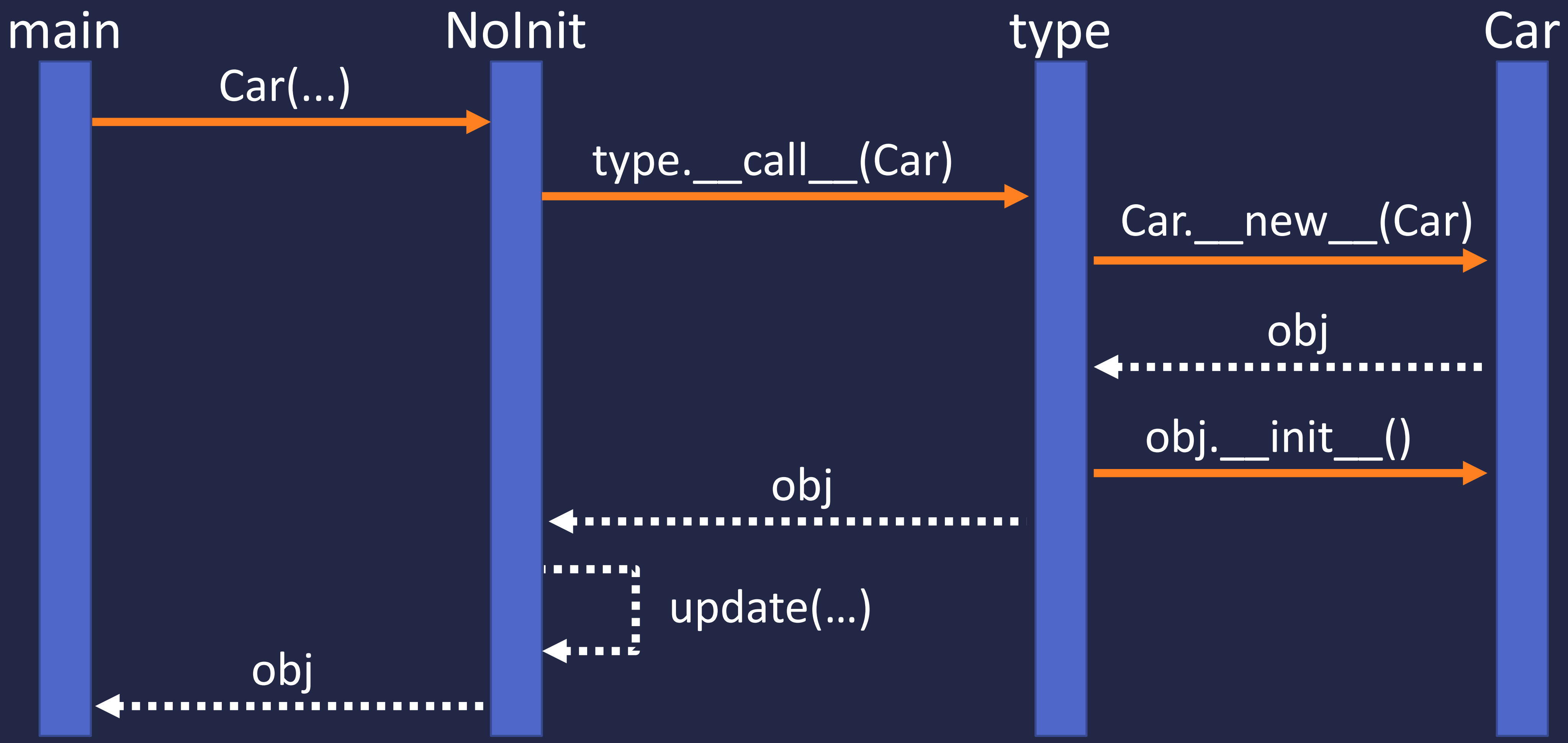
- Intercepts instance creation
- Default implementation

```
class MetaClass(type):
    def __call__(cls, *pa, **kwa):
        # this calls object.__new__
        return type.__call__(cls, *pa, **kwa)

class Klass(metaclass=MetaClass):
    pass

# Same as MetaClass.__call__(Klass, [5], {‘a’:2, ‘b’:3})
inst = Klass(5, a=2, b=3)
```

Metaclass.__call__

```
class NoInit(type):
    def __call__(cls, **kwargs):
        inst = type.__call__(cls)
        inst.__dict__.update(kwargs)
        return inst

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

>> car = Car(make="Mazda", model="CX-5", year="2019", color="White")
>> print(car)  # make=Mazda model=CX-5 year=2019 color=White
```

Instance Creation



__prepare__

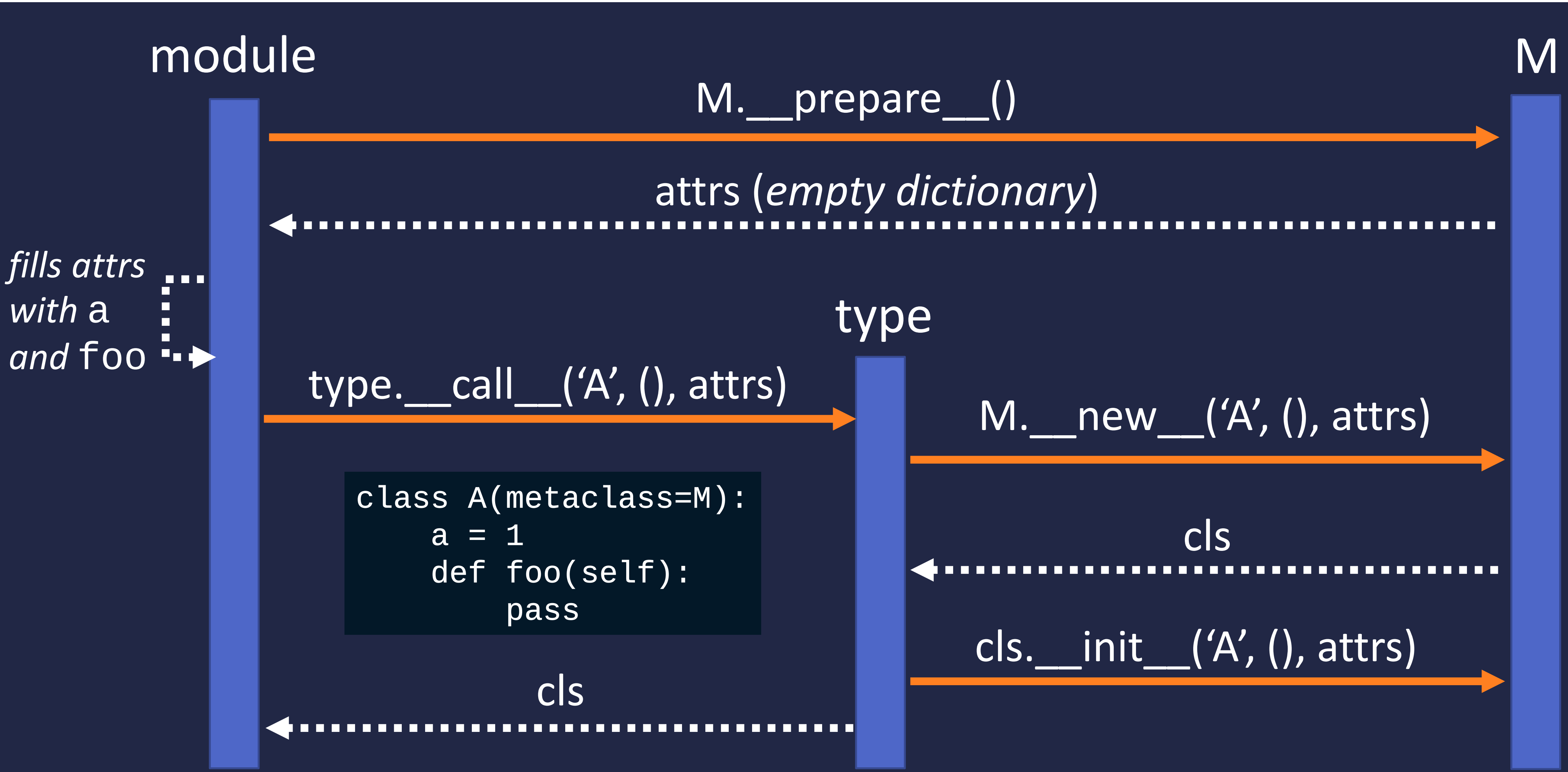
- Provides a dictionary-like object to store attributes
- By default, returns Python dictionary – dict()
- Exists for performance and correctness reasons
  - E.g. ordered dictionary
    - Order of iteration guaranteed same as order of insertion

```
import collections
class Meta(type):
    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        return collections.OrderedDict()
```

Metametaclass

- The metaclass of a metaclass
- Begins the process of creating a new class
  - Via __call__
  - In contrast, a metaclass's __call__ method 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

Class Creation



module

M

M.__prepare__()

attrs (*empty dictionary*)

fills attrs with a *and* foo

type

type.__call__('A', (), attrs)

M.__new__('A', (), attrs)

cls

cls.__init__('A', (), attrs)

cls

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

Metaclass.__new__

- Constructor for the class

```
class Meta(type):
    # called before the class is created
    def __new__(mcs, name, bases, attrs, **kwargs):
        return super().__new__(mcs, name, bases, attrs)

    # called after the class is created
    def __init__(cls, name, bases, attrs, **kwargs):
        super().__init__(name, bases, attrs)
```

- Usually you pick one to override, but not both
  - Use __new__ for error checking
  - Use __init__ for processing

Metafunction

- Metaclass only needs to be callable
- If inheritance not needed, can use a function!
- The type is the type of the return value of the function

```
# uses same interface as type(name, bases, attrs)
def MetaFunc(name, bases, attrs):
    attrs["hello"] = 5              # add hello attribute
    return type(name, bases, attrs)

class Foo(metaclass=MetaFunc):
    bar = 3
# same as Foo = MetaFunc(‘Foo’, (), {‘bar’ : 3})
```