Python *classes*

```
>>> class MyClass:
...     class_attribute_one = "One attribute"
...
...     def __init__(self, *args):
...         self.state = list(args)
...         self.__my_atrr = False
...
...     def get_state(self):
...         return self.state
...
...     def add_state(self, state):
...         self.state.append(state)
...
...     def __bool__(self):
...         print(self.__str__())
...         return bool(self.state)
...
...     def __str__(self):
...         return f"MyClass with state {self.state}"
```

Creating instance

```
>>> inst = MyClass(1, 2, 3)
>>> inst.__class__
<class '__main__.MyClass'>

>>> inst.get_state()
[1, 2, 3]
>>> inst.add_state(4)
>>> inst.get_state()
[1, 2, 3, 4]

>>> str(inst)
'MyClass with state [1, 2, 3, 4]'

>>> bool(inst)
MyClass with state [1, 2, 3, 4]
True
```

__dict__

```
>>> pprint(MyClass.__dict__)
mappingproxy({'__bool__': <function MyClass.__bool__ at 0x7f97f727bf28>,
              '__dict__': <attribute '__dict__' of 'MyClass' objects>,
              '__doc__': None,
              '__init__': <function MyClass.__init__ at 0x7f97f727b9d8>,
              '__module__': '__main__',
              '__str__': <function MyClass.__str__ at 0x7f97f727bbf8>,
              '__weakref__': <attribute '__weakref__' of 'MyClass' objects>,
              'add_state': <function MyClass.add_state at 0x7f97f727b8c8>,
              'class_attribute_one': 'One attribute',
              'get_state': <function MyClass.get_state at 0x7f97f727bc80>})

>>> pprint(inst.__dict__)
{'_MyClass__my_atrr': False, 'state': [1, 2, 3, 4]}
```

Descriptors

To be a ***descriptor***, a class must have at least one of ***__get__***, ***__set__***, and ***__delete__*** implemented.

Descriptors

```
>>> class Meter:
...     '''Descriptor for a meter.'''
...     def __init__(self, value=0.0):
...         self.value = float(value)
...     def __get__(self, instance, owner):
...         return self.value
...     def __set__(self, instance, value):
...         self.value = float(value)
...
... class Foot:
...     '''Descriptor for a foot.'''
...     def __get__(self, instance, owner):
...         return instance.meter * 3.2808
...     def __set__(self, instance, value):
...         instance.meter = float(value) / 3.2808
...
... class Distance:
...     '''Class to represent distance holding two descriptors for feet and
...     meters.'''
...     meter = Meter()
...     foot = Foot()
```

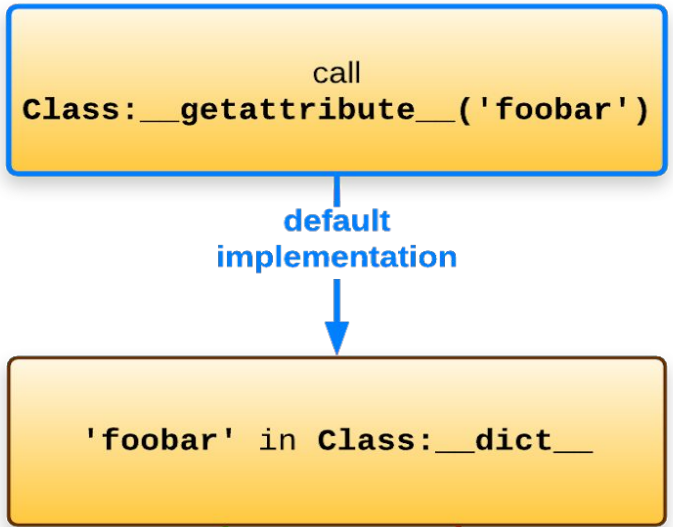
Descriptors

```
>>> Distance().meter = 10.

>>> dist = Distance()

>>> dist.meter = 10.

>>> dist.foot
32.808


>>> dist.foot = 15

>>> dist.meter
4.57205559619605
```

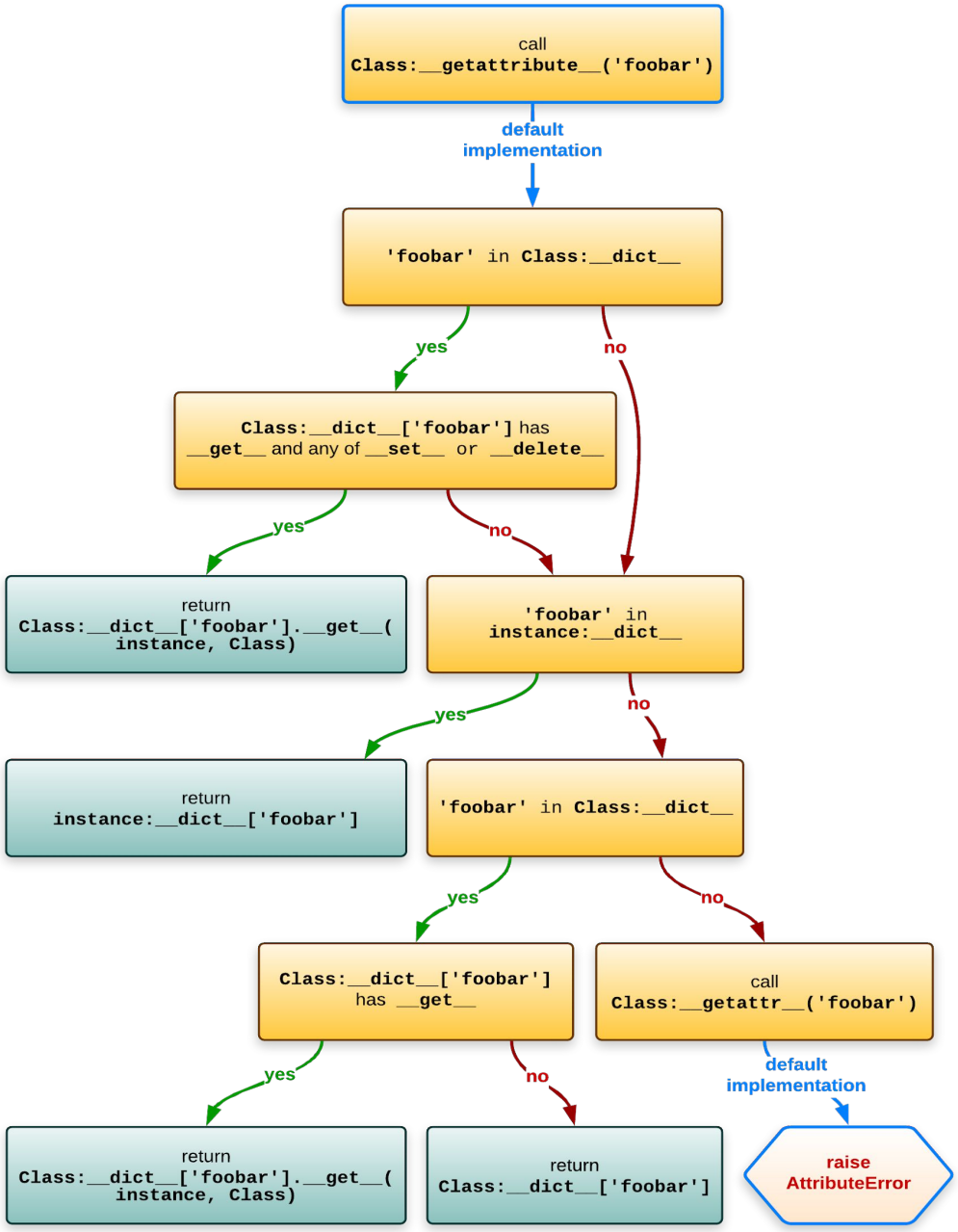
What happens?

```
>>> inst = MyClass(1, 2, 3)
>>> inst.foobar
```

__getattribute__



https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/


call
Class:__getattribute__('foobar')
default implementation
'foobar' in Class:__dict__
yes
no
Class:__dict__['foobar'] has __get__ and any of __set__ or __delete__
yes
no
return Class:__dict__['foobar'].__get__(instance, Class)
'foobar' in instance:__dict__
yes
no
return instance:__dict__['foobar']
'foobar' in Class:__dict__
yes
no
Class:__dict__['foobar'] has __get__
call Class:__getattr__('foobar')
yes
no
default implementation
return Class:__dict__['foobar'].__get__(instance, Class)
return Class:__dict__['foobar']
raise AttributeError

What happens?

```
>>> inst = MyClass(1, 2, 3)
>>> MyClass.__dict__['add_state'](inst, 42)
~
>>> inst.add_state(42)
```

What happens?

```
>>> inst = MyClass(1, 2, 3)
>>> MyClass.__dict__['add_state'](inst, 42)
~
>>> inst.add_state(42)
```

Class is object

```
>>> isinstance(MyClass, object)
True


>>> def object_maker(class_, *args, **kwargs):
...         return class_(*args, **kwargs)


>>> my_cls_inst = object_maker(MyClass, 1, 2, 3, 4, 5)
... print(my_cls_inst)
MyClass with state [1, 2, 3, 4, 5]
```

Class is statement

```
>>> class StatementClass:
...     a, b, c = 1, 2, 3
...     for _ in range(10):
...         a += 1
...         b += 1
...         c += 1
...
...
... print(StatementClass.a, StatementClass.b, StatementClass.c)
11 12 13
```

Class is statement

```
>>> class StatementClass:
...     a, b, c = 1, 2, 3
...     for _ in range(10):
...         a += 1
...         b += 1
...         c += 1
...
...
... print(StatementClass.a, StatementClass.b, StatementClass.c)
11 12 13
```

Properties

```
>>> class State:
...     def __init__(self, state=0):
...         self._state = state
...
...     @property
...     def is_zero(self):
...         return self._state == 0

>>> inst = State()
>>> inst.is_zero
True
```

Properties

```
>>> class State:
...     def __init__(self, state=0):
...         self._state = state
...
...     @property
...     def is_zero(self):
...         return self._state == 0

>>> inst.is_zero = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

can't set attribute
```

Properties

```
>>> class Data:
...     def __init__(self, field_1):
...         self.__field_1 = field_1
...
...     @property
...     def field_1(self):
...         return self.__field_1
...
...     @field_1.setter
...     def field_1(self, value):
...         self.__field_1 = value * 10

>>> inst = Data(1)

>>> inst.field_1
1

>>> inst.field_1 = 2
>>> inst.field_1
20
```

Properties

```
>>> class Data:
...     def __init__(self, field_1):
...         self.__field_1 = field_1
...
...     @property
...     def field_1(self):
...         return self.__field_1
...
...     @field_1.setter
...     def field_1(self, value):
...         self.__field_1 = value * 10

>>> inst = Data(1)

>>> inst.field_1
1

>>> inst.field_1 = 2
>>> inst.field_1
20
```

Dunder methods

```
>>> class Int:
...     def __init__(self, value):
...         self.value = int(value)
...
...     def __lt__(self, other):
...         if isinstance(other, Int):
...             return self.value < other.value
...         return self.value < other
...
...     def __eq__(self, other):
...         if isinstance(other, Int):
...             return self.value == other.value
...         return self.value == other
```

Dunder methods

```
>>> class Int:
...     ...

>>> Int(1) < 1
False


>>> Int(1) < Int(2)
True


>>> Int(1) == 1
True


>>> Int(1) == Int(2)
False
```

Dunder methods

```
>>> class Int:
...     def __init__(self, value):
...         self.value = int(value)
...
...     def __hash__(self):
...         return hash(self.value)
...
...     def __eq__(self, other):
...         if isinstance(other, Int):
...             return self.value == other.value
...         return self.value == other

>>> d = {Int(1), 1}

>>> d
{<__main__.Int object at 0x7f97f77d99e8>}

>>> d = {Int(1): "Hello", 1: "world"}
>>> d
{<__main__.Int object at 0x7f97fbbcedd8>: 'world'}
```

Dunder methods

__eq__(self, other)
Defines behavior for the equality operator, ==.

__ne__(self, other)
Defines behavior for the inequality operator, !=.

__lt__(self, other)
Defines behavior for the less-than operator, <.

__gt__(self, other)
Defines behavior for the greater-than operator, >.

__le__(self, other)
Defines behavior for the less-than-or-equal-to operator, <=.

__ge__(self, other)
Defines behavior for the

greater-than-or-equal-to operator, >=

Sequence

__len__(self)

Returns the length of the container. Part of the protocol for both immutable and mutable containers.

__getitem__(self, key)
Defines behavior for when an item is accessed, using the notation self[key]. This is also part of both the mutable and immutable container protocols. It should also raise appropriate exceptions: TypeError if the type of the key is wrong and KeyError if there is no corresponding value for the key.

__setitem__(self, key, value)
Defines behavior for when an item is assigned to, using the notation self[nkey] = value. This is part of the mutable container protocol. Again, you should raise KeyError and TypeError where appropriate.

__delitem__(self, key)
Defines behavior for when an item is deleted (e.g. del self[key]). This is only part of the mutable container protocol. You must raise the appropriate exceptions when an invalid key is used.

Sequence

__iter__(self)
Should return an iterator for the container. Iterators are returned in a number of contexts, most notably by the iter() built in function and when a container is looped over using the form for x in container:. Iterators are their own objects, and they also must define an __iter__ method that returns self.

__reversed__(self)
Called to implement behavior for the reversed() built in function. Should return a reversed version of the sequence. Implement this only if the sequence class is ordered, like list or tuple.

__contains__(self, item)
__contains__ defines behavior for membership tests using in and not in. Why isn't this part of a sequence protocol, you ask? Because when __contains__ isn't defined, Python just iterates over the sequence and returns True if it comes across the item it's looking for.

__missing__(self, key)
__missing__ is used in subclasses of dict. It defines behavior for whenever a key is accessed that does not exist in a dictionary (so, for instance, if I had a dictionary d and said d["george"] when "george" is not a key in the dict, d.__missing__("george") would be called).

Inheritance

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Inheritance

```
>>> class Base:
...     def __init__(self, atr):
...         self.atr = atr
...
...     def do_smth(self):
...         return self.atr ** 2

>>> class A(Base):
...     def do_smth(self):
...         return self.atr ** 3

>>> base = Base(10)
>>> base.do_smth()
100

>>> a = A(10)
>>> a.do_smth()
1000
```

Inheritance

```
>>> class Base:
...     def __init__(self, atr):
...         self.atr = atr
...
...     def do_smth(self):
...         return self.atr ** 2

>>> class A(Base):
...     ...

>>> issubclass(A, Base)
True
>>> issubclass(A, object)
True
```

Inheritance

```
>>> class Base:
...     def __init__(self, atr):
...         self.atr = atr
...
...     def do_smth(self):
...         return self.atr ** 2

>>> class A(Base):
...     def do_smth(self):
...         return self.atr ** 3

>>> base = Base(10)
>>> base.do_smth()
100

>>> a = A(10)
>>> a.do_smth()
1000
```

Inheritance

```
>>> class Base:
...     def __init__(self, atr):
...         self.atr = atr
...
...     def do_smth(self):
...         return self.atr ** 2

>>> class A(Base):
...     def __init__(self, atr):
...         super().__init__(atr * 2)
...
...     def do_smth(self):
...         return self.atr ** 3

>>> a = A(10)
>>> a.do_smth()
8000
```

Inheritance

```
>>> class Base:
...     def __init__(self, atr):
...         self.atr = atr
...
...     def do_smth(self):
...         return self.atr ** 2

>>> class A(Base):
...     def __init__(self, atr):
...         # super(A, self)
...         super().__init__(atr * 2)
...
...     def do_smth(self):
...         return self.atr ** 3

>>> a = A(10)
>>> a.do_smth()
8000
```

Inheritance: mro

```
>>> class Base:
        ...

>>> class A(Base):
        ...

>>> A.mro()
[<class '__main__.A'>, <class '__main__.Base'>, <class
'object'>]
```

Multiple inheritance

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Multiple inheritance

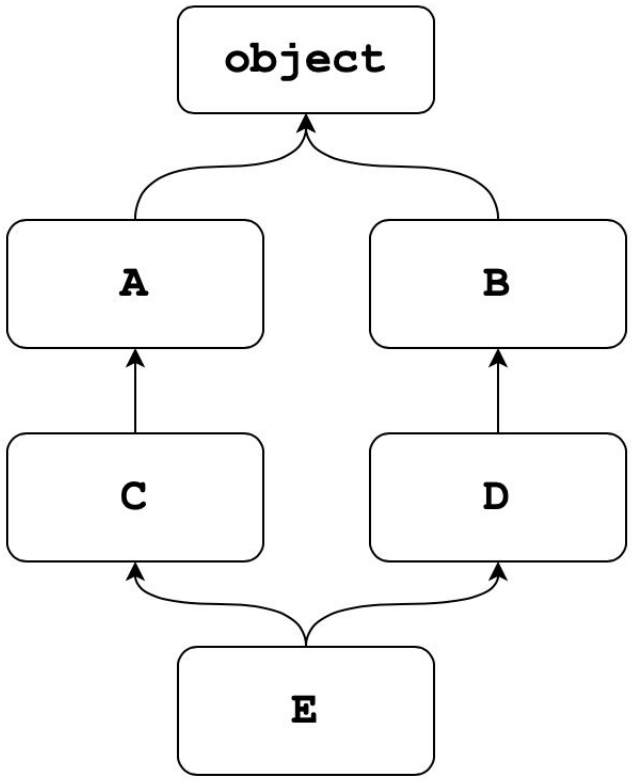
```
>>> class Base1:
...     def do_smth(self):
...         print("Base1")

>>> class Base2:
...     def do_smth(self):
...         print("Base2")

>>> class A(Base1, Base2):
...     def do_smth(self):
...         super().do_smth()

>>> A().do_smth()
Base1

>>> A.mro()
[<class '__main__.A'>, <class '__main__.Base1'>, <class
'__main__.Base2'>, <class 'object'>]
```

Multiple inheritance

```
>>> class A():
...     ...

>>> class B():
...     ...

>>> class C(A):
...     ...

>>> class D(B):
...     ...

>>> class E(C, D):
...     ...

>>> E.mro()
[<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.D'>, <class '__main__.B'>, <class 'object'>]
```

Multiple inheritance

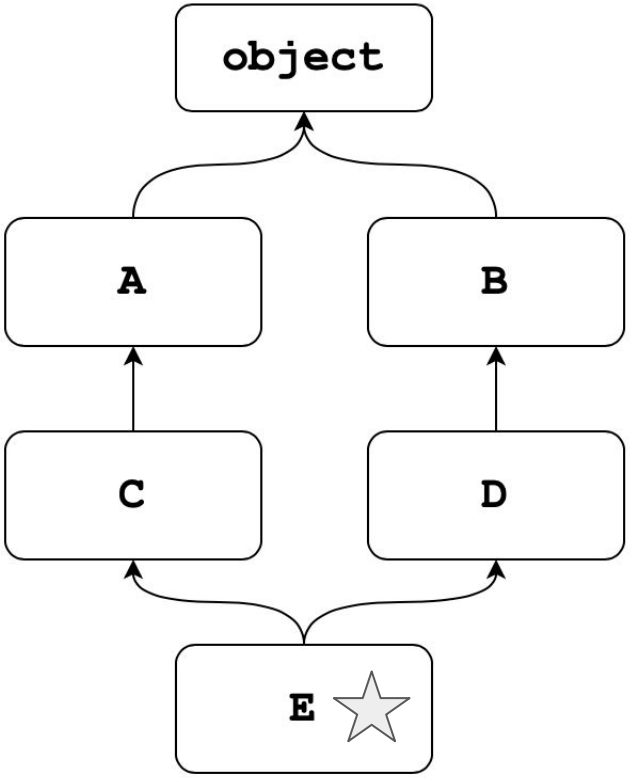
```
>>> class A():
...     ...

>>> class B():
...     ...

>>> class C(A):
...     ...

>>> class D(B):
...     ...

>>> class E(C, D):
...     ...
```




```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.D'>, <class '__main__.B'>, <class 'object'>]
```

Multiple inheritance

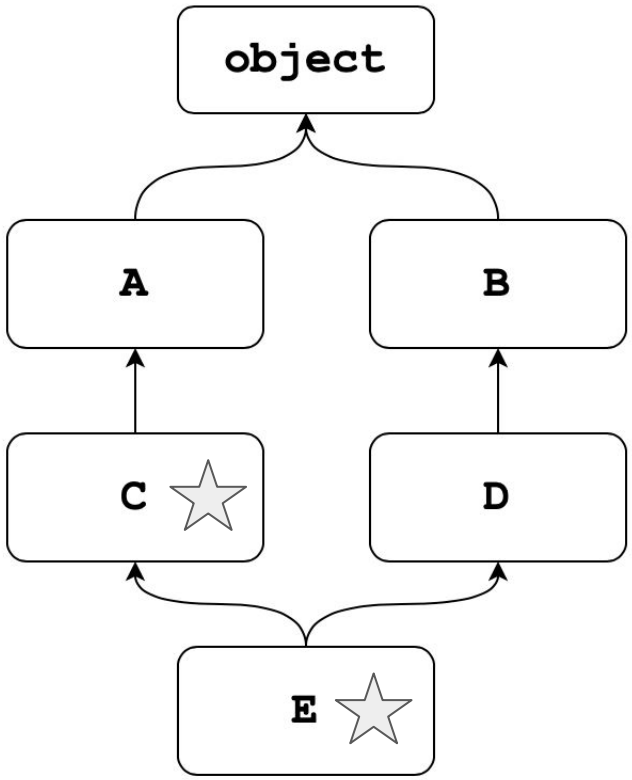
```
>>> class A():
...     ...

>>> class B():
...     ...

>>> class C(A):
...     ...

>>> class D(B):
...     ...

>>> class E(C, D):
...     ...
```




```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.D'>, <class '__main__.B'>, <class 'object'>]
```

Multiple inheritance

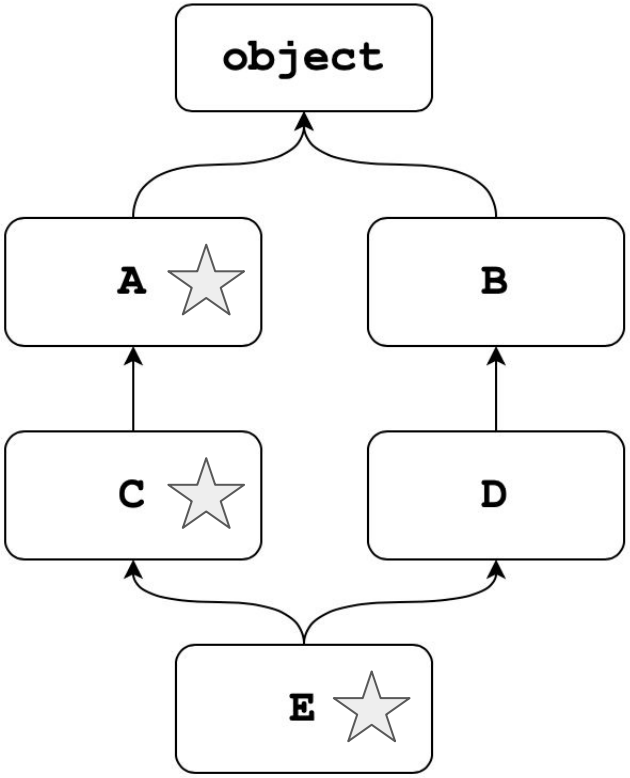
```
>>> class A():
...     ...

>>> class B():
...     ...

>>> class C(A):
...     ...

>>> class D(B):
...     ...

>>> class E(C, D):
...     ...
```




```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.D'>, <class '__main__.B'>, <class 'object'>]
```

Multiple inheritance

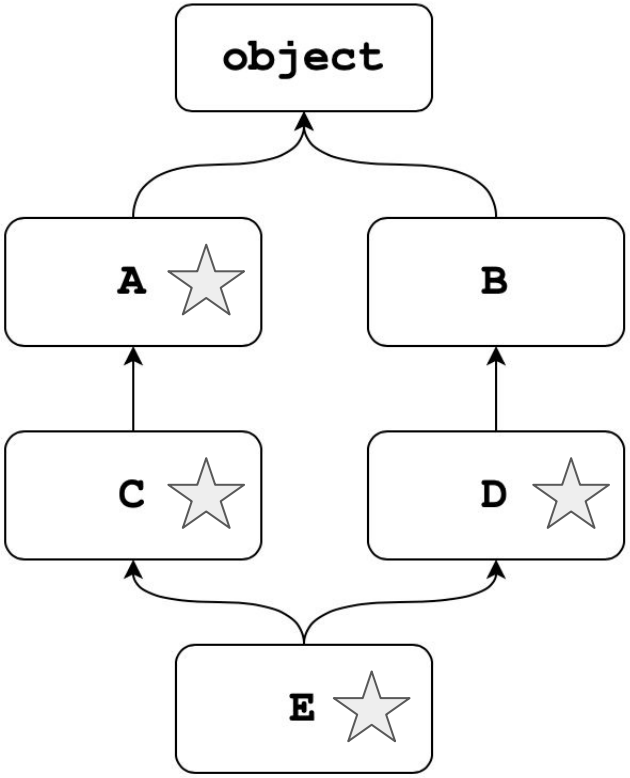
```
>>> class A():
...     ...

>>> class B():
...     ...

>>> class C(A):
...     ...

>>> class D(B):
...     ...

>>> class E(C, D):
...     ...
```




```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.D'>, <class '__main__.B'>, <class 'object'>]
```

Multiple inheritance

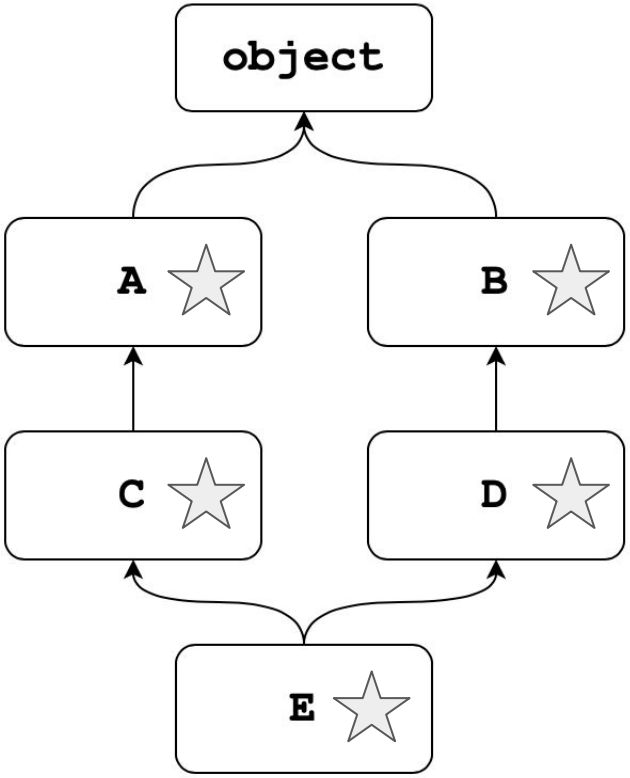
```
>>> class A():
...     ...

>>> class B():
...     ...

>>> class C(A):
...     ...

>>> class D(B):
...     ...

>>> class E(C, D):
...     ...
```




```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.D'>, <class '__main__.B'>, <class 'object'>]
```

Multiple inheritance

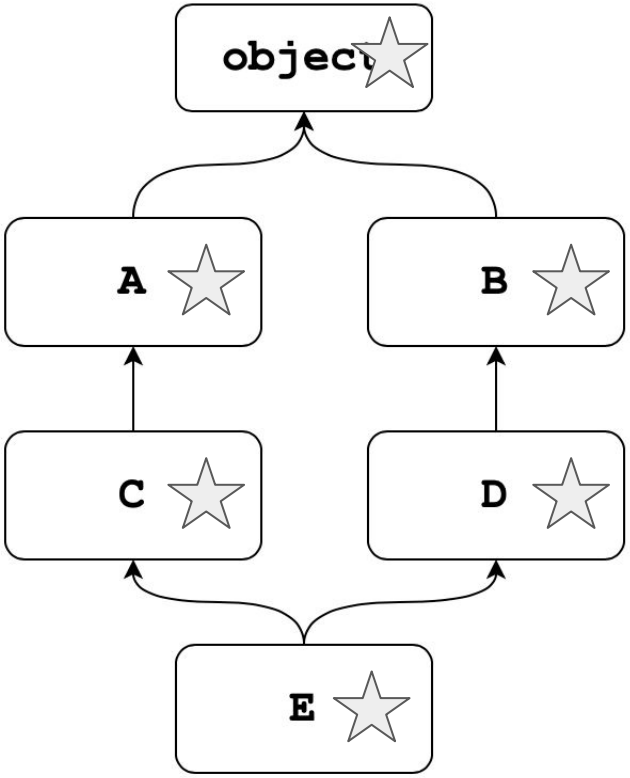
```
>>> class A():
...     ...

>>> class B():
...     ...

>>> class C(A):
...     ...

>>> class D(B):
...     ...

>>> class E(C, D):
...     ...
```




```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.D'>, <class '__main__.B'>, <class 'object'>]
```

Multiple inheritance

```
>>> class A():
...     ...

>>> class B():
...     ...

>>> class C(A):
...     ...

>>> class D(B):
...     ...

>>> class E(C, D):
...     ...
```




```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.D'>, <class '__main__.B'>, <class 'object'>]
```

Inconsistent mro

```
>>> class M:
...     ...

>>> class N:
...     ...

>>> class B(N, M):
...     ...

>>> class A(M, N):
...     ...

>>> class D(A, B):
...     ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method resolution
order (MRO) for bases M, N

Cannot create a consistent method resolution
order (MRO) for bases M, N
```

To read

rszalski.github.io/magicmethods/

blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/

code.activestate.com/recipes/577720-how-to-use-super-effectively/