# Making Multiple Inheritance Not Work in Python

*A destructive investigation of metaclasses and other hooks*

Bloomberg

Engineering

**PyCon Canada 2019**
**November 17, 2019**

**Andy Fundinger**
**Senior Engineer**
**afundinger1@bloomberg.net**
**@andriod**

**TechAtBloomberg.com**

# What is Inheritance

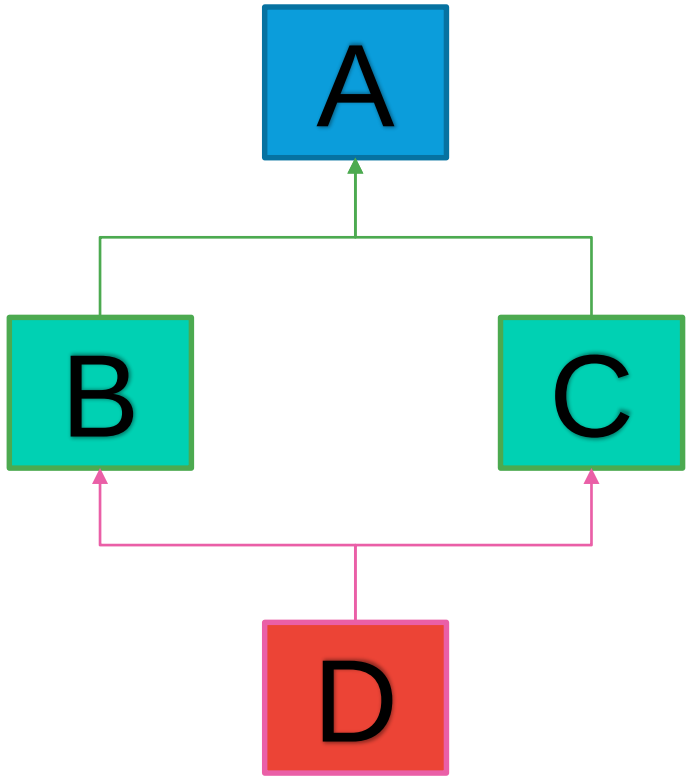
In object-oriented programming, **inheritance** is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation.

- Wikipedia

- First implemented in Simula, 1965
- Included in object-oriented languages ever since

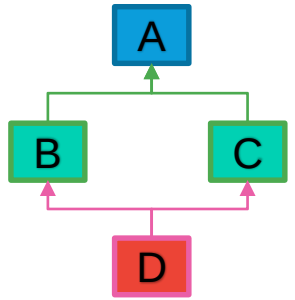
# The Diamond Problem

- Start with a base class
- Create two inherited classes
- Create a new class that inherits from both
- Now try to reason about this.



# Multiple Inheritance in Python 2

- Python's Old Style classes




```
class A:
    def __init__(self):
        print('A.__init__')
class B(A):
    def __init__(self):
        A.__init__(self)
        print('B.__init__')
class C(A):
    def __init__(self):
        A.__init__(self)
        print('C.__init__')
class D(B,C):
    def __init__(self):
        B.__init__(self)
        C.__init__(self)
        print('D.__init__')
```

```
D()
```

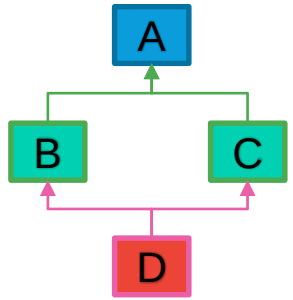
```
A.__init__

B.__init__

A.__init__

C.__init__

D.__init__

<__main__.D at 0x24ec8419320>
```

Bloomberg

Engineering

# Multiple Inheritance in Python 2.2 (and later)

- New Style Classes
- - the mro




```
class A:
    def __init__(self):

        print('A.__init__')
class B(A):
    def __init__(self):

        print('B.__init__')
class C(A):
    def __init__(self):

        print('C.__init__')
class D(B,C):
    def __init__(self):

        print('D.__init__')
```

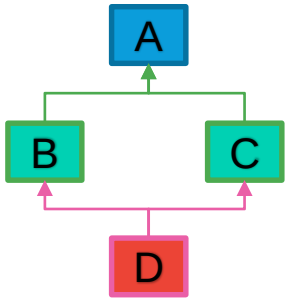
```
D.__mro__
```

(__main__.D, __main__.B, __main__.C, __main__.A, object)

Bloomberg

Engineering

# Multiple Inheritance in Python 2.2 (and later)




- New Style Classes
- - the mro
- - super()

```
class A:
    def __init__(self):
        super(A,self).__init__()
        print('A.__init__')
class B(A):
    def __init__(self):
        super(B,self).__init__()
        print('B.__init__')
class C(A):
    def __init__(self):
        super(C,self).__init__()
        print('C.__init__')
class D(B,C):
    def __init__(self):
        super(D,self).__init__()
        print('D.__init__')
```

```
D.__mro__
```

(__main__.D, __main__.B, __main__.C, __main__.A, object)

```
D()
```

A.__init__

C.__init__

B.__init__

D.__init__

Bloomberg

Engineering

# Where do we play our games?

TechAtBloomberg.com



Bloomberg

Engineering

# Class decorators - Python 2.6

```
def my_decorator(klass):
    print(klass)
    return klass

@my_decorator
class MyClass:pass
```

<class '__main__.MyClass'>

```
def graffiti_decorator(klass):
    klass.tag = 'Kilroy was here'
    return klass

@graffiti_decorator
class MyClass: pass

MyClass.tag
```

'Kilroy was here'

Bloomberg

Engineering

# The metaclass

- This was exposed at the Python level in Python 2.2
- This is the class of a class

```
class MyClass:pass
MyClass.__class__
```

```
type
```

```
class MyMeta(type):pass
class MyClass(metaclass=MyMeta):pass
MyClass.__class__
```

```
__main__.MyMeta
```

```
class Negativity: pass

bool(Negativity)
```

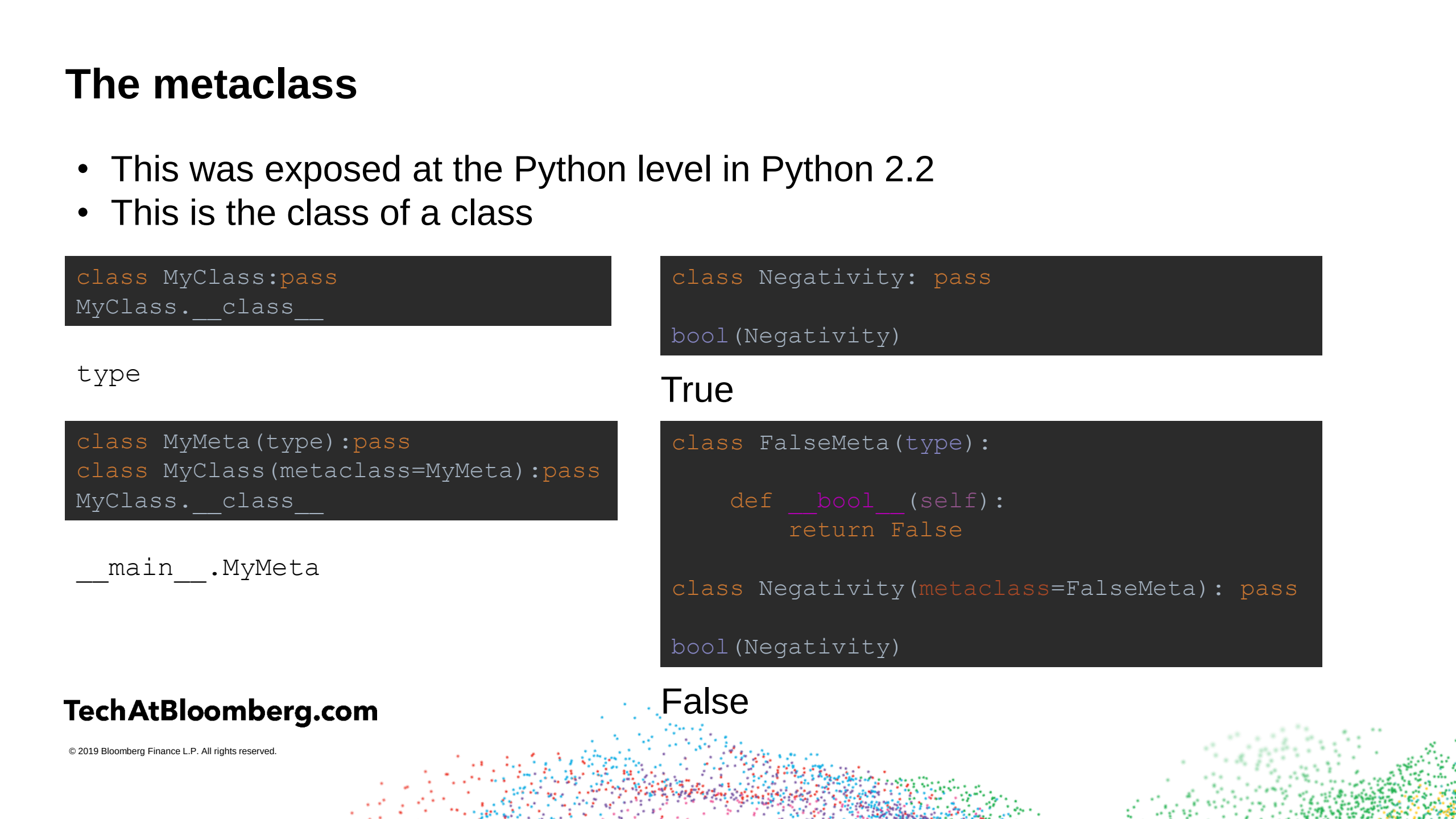
True

```
class FalseMeta(type):

    def __bool__(self):
        return False

class Negativity(metaclass=FalseMeta): pass

bool(Negativity)
```

False

TechAtBloomberg.com





# The metaclass

- This was exposed at the Python level in Python 2.2
- This is the class of a class

```
class MyMeta(type):
    def mro(cls):
        return super(MyMeta, cls).mro()

    def __new__(meta, name, bases, dct, **kwargs):
        return super(MyMeta, meta).__new__(meta, name, bases, dct, **kwargs)
```

TechAtBloomberg.com



Bloomberg

Engineering

# __subclasscheck__ and __instancecheck__

- Python 2.7 gave us operator overriding for the isinstance() and issubclass() via new magic methods in the metaclass
- Return True/False or call super()

```
def __instancecheck__(cls, instance):
    return super(mcls, cls).__instancecheck__(instance)
def __subclasscheck__(cls, subclass):
    return super(mcls, cls).__subclasscheck__(subclass)
```





# __subclasscheck__ and __instancecheck__

```
class CheckCheckedMeta(type):
    def __instancecheck__(cls, instance):
        print(f'instance checked {instance}')
        return super(CheckCheckedMeta, cls).__instancecheck__(instance)
    def __subclasscheck__(cls, subclass):
        print(f'subclass checked {subclass}')
        return super(CheckCheckedMeta, cls).__subclasscheck__(subclass)

class CheckChecker(metaclass=CheckCheckedMeta): pass
```

```
isinstance(None,CheckChecker)
```

instance checked None
False

```
issubclass(object,CheckChecker)
```

subclass checked <class 'object'>
False

Bloomberg

Engineering

# __init_subclass__

- Python 3.6 added the ability of a class to control the initialization subclasses
- Basically a built-in class decorator

```
def __init_subclass__(cls, **kwargs):
    super().__init_subclass__(**kwargs)
```

TechAtBloomberg.com



Bloomberg

Engineering

# __init_subclass__

- Python 3.6 added the ability of a class to control the initialization subclasses
- Basically a built-in class decorator

```
class SizedContainer:
    def __init_subclass__(cls, **kwargs):
        cls.cls_field_count = len(cls.__dict__)
        super().__init_subclass__(**kwargs)

class Jar(SizedContainer):
    pickles = 18
    juice_oz = 12

Jar.cls_field_count
```

4





Bloomberg

Engineering

Making Multiple Inheritance Not Work

# Let the Games Begin

TechAtBloomberg.com



Bloomberg

Engineering

# Pure Interfaces

- For some reason, we don't want **any** concrete methods in our interfaces

```
import abc


class SloppyInterface(abc.ABC):
    @abc.abstractmethod
    def fetch_bread(self): pass
    @abc.abstractmethod
    def fetch_meat(self): pass

    def make_sandwich(self):
        return [self.fetch_bread(),
                self.fetch_meat(),
                self.fetch_bread()]
```

```
SloppyInterface.make_sandwich()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent
call last)
<ipython-input-7-eb2a06b1418f> in <module>()
----> 1 SloppyInterface.make_sandwich()

AttributeError: type object 'SloppyInterface' has no
attribute 'make_sandwich'
```

```
SloppyInterface.__dict__.keys()
```

```
dict_keys(['__module__', 'fetch_bread', 'fetch_meat',
'__dict__', '__weakref__', '__doc__',
'__abstractmethods__', '_abc_impl'])
```

**TechAtBloomberg.com**

# Pure Interfaces

- For some reason, we don't want **any** concrete methods in our interfaces

```
import abc

@force_interface
class SloppyInterface(abc.ABC):
    @abc.abstractmethod
    def fetch_bread(self): pass
    @abc.abstractmethod
    def fetch_meat(self): pass

    def make_sandwich(self):
        return [self.fetch_bread(),
                self.fetch_meat(),
                self.fetch_bread()]
```

```
def force_interface(klass):
    for f, d in tuple(klass.__dict__.items()):
        if not hasattr(d,'__isabstractmethod__') \
        and f not in ('__dict__','__module__',
                      '__doc__','__weakref__',
                      '__abstractmethods__',
                      '_abc_impl'):
            delattr(klass, f)
    return klass
```

TechAtBloomberg.com



Bloomberg

Engineering

# Pure Interfaces

- Let’s not count on putting the decorator there

```
class ForceInterface(abc.ABC):
  def __init_subclass__(cls):
    super().__init_subclass__()
    for f, d in tuple(
            cls.__dict__.items()):
      if not hasattr(d,
            '__isabstractmethod__')\
        and f not in ('__dict__',
                      '__module__',
                      '__doc__',
                      '__weakref__',
                      '__abstractmethods__',
                      '_abc_impl'):
                delattr(cls, f)
```

```
class SloppyInterface(ForceInterface):
    @abc.abstractmethod
    def fetch_bread(self): pass
    @abc.abstractmethod
    def fetch_meat(self): pass

    def make_sandwich(self):
        return [self.fetch_bread(),
                self.fetch_meat(),
                self.fetch_bread()]
```

TechAtBloomberg.com



Bloomberg

Engineering

# Pure Interfaces

- Let’s not count on putting the decorator there

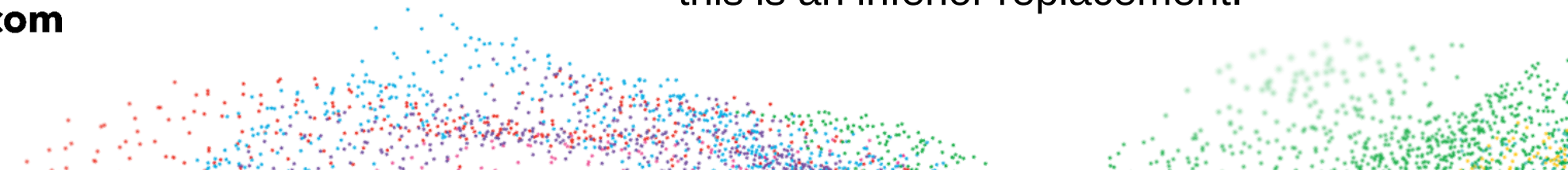
```
class ForceInterface(abc.ABC):
  def __init_subclass__(cls):
    super().__init_subclass__()
    for f, d in tuple(
            cls.__dict__.items()):
      if not hasattr(d,
             '__isabstractmethod__')\
        and f not in ('__dict__',
                  '__module__',
                  '__doc__',
                  '__weakref__',
                  '__abstractmethods__',
                  '_abc_impl'):
                delattr(cls, f)
```

```
class RealSandwich(SloppyInterface):
    def __init_subclass__(cls, **kwargs):
        pass

class BeefBrisket(RealSandwich):
    def fetch_bread(self):
        return "1 Slice Wonder Bread"
    def fetch_meat(self):
        return "Beef Brisket"
    def make_sandwich(self):
        return [self.fetch_bread(),
                self.fetch_meat(),
                self.fetch_bread()]
```

**Real Talk:** Abstract base classes are fine; this is an inferior replacement.







# Fake Interfaces

- Let’s assume we’re reading in data that used to be classes but is now just dicts.

```
class Stock(FakeInterface):
    type_flag = 'STOCK'

class PreferredStock(FakeInterface):
    type_flag = 'PREFFERRED'
```

```
ibmc = {'type_flag': 'STOCK', 'SYM':
        'IBM', 'Price': 135.47}
ibmp = {'type_flag': 'PREFERRED',
        'SYM': 'IBM-A', 'Price': None}
```

```
isinstance(ibmc, Stock)
```

**True**

```
isinstance(ibmc, PreferredStock)
```

**False**

```
isinstance(ibmp, PreferredStock)
```

**True**

```
issubclass(dict, Stock)
```

**False**

# Fake Interfaces

- There’s a metaclass to alter the behavior of isinstance()

```
class FakeInterfaceMeta(type):
    def __instancecheck__(cls, instance):
        if not hasattr(instance, 'get'):
            return super(FakeInterfaceMeta, cls).__instancecheck__(instance)
        return cls.type_flag == instance.get('type_flag', None)

class FakeInterface(metaclass=FakeInterfaceMeta): pass
```

```
class Stock(FakeInterface):
    type_flag = 'STOCK'

class PreferredStock(FakeInterface):
    type_flag = 'PREFFERRED'
```

**Real Talk:** This alone is not enough to justify using isinstance, but if you are already using it for other reasons, you can use this to enable special rules.

# Partially Disabled Inheritance

- I have some methods that I don't to be overridden

```
class Car(Vehicle):
    def start(self):
        print('Turn key')
```

```
Car().name
```

```
'Car'
```

```
Car().start()
```

```
Turn key
```

```
class Taxi(Car):
    name = 'Ride Share'
```

```
Taxi().start()
```

```
Turn key
```

```
Taxi().name
```

```
'Taxi'
```

```
Taxi.__mro__
```

```
(__main__.UtilityClass, __main__.Taxi,
__main__.Car, __main__.Vehicle, object)
```

```
class Vehicle(metaclass=MetaUtil):
    pass
```

Bloomberg

Engineering

# Partially Disabled Inheritance

- I have some methods that I don't to be overridden

```
class MetaUtil(type):
    def mro(cls):
        mro = type.mro(cls)
        if UtilityClass in mro:
            mro.remove(UtilityClass)
        mro = [UtilityClass]+\
            mro
        return mro

class UtilityClass:
    @property
    def name(self):
        return self.__class__.__name__

class Vehicle(metaclass=MetaUtil):
    pass
```

```
Taxi.__mro__
```

```
(__main__.UtilityClass, __main__.Taxi,
__main__.Car, __main__.Vehicle, object)
```

**Real Talk:** This might actually be the safest thing to use in here.

Bloomberg

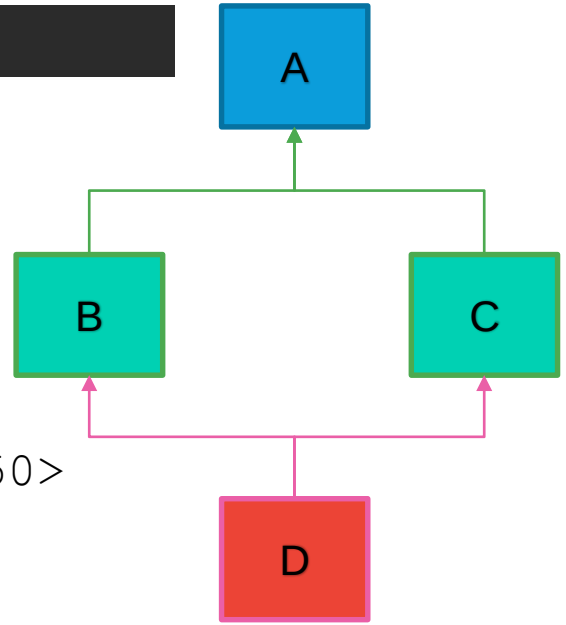
Engineering

# Single Inheritance Only

- Ok, I’m tired of that diamond from the beginning. Let’s put an end to it

```
class A(SimpleObject):
    def __init__(self):
        super(A,self).__init__()
        print('A.__init__')
class B(A):
    def __init__(self):
        super(B,self).__init__()
        print('B.__init__')
class C(A):
    def __init__(self):
        super(C,self).__init__()
        print('C.__init__')
class D(B,C):
    def __init__(self):
        super(D,self).__init__()
        print('D.__init__')
```

```
D()
A.__init__
B.__init__
D.__init__
<__main__.D at 0x1f82491ac50>
```




Bloomberg

Engineering

# Fully Disabled Inheritance

- There’s probably a metaclass back here…

```
class SimpleBase(type):
    def mro(cls):
        ret = [cls]
        while len(cls.__bases__):
            cls = cls.__bases__[0]
            ret.append(cls)
        return ret

class SimpleObject(metaclass=SimpleBase): pass
```

**Real Talk:** Again, the point is that we can pick and choose some features, not which features we're picking. I might particularly suggest removing bases.

TechAtBloomberg.com



Bloomberg

Engineering

# Fully Disabled Inheritance

- There’s probably a metaclass back here…

```
class SimpleBase(type):
    def __new__(meta, name, bases, dct, **kwargs):
        return super(SimpleBase, meta).__new__(meta, name, bases[0:1], dct, **kwargs)

class SimpleObject(metaclass=SimpleBase):
    pass
```

**Real Talk:** Again, the point is that we can pick and choose some features, not which features we're picking. I might particularly suggest removing bases.

TechAtBloomberg.com



Bloomberg

Engineering

# Conclusion

- We’ve seen five hooks that let us twist inheritance into a pickle:
  — @classdecorator
  — __initsubclass__
  — __subclasscheck__ and __instancecheck__
  — mro()
  — metaclass __new__()

- While it is rare that we want to tweak these, the results are dramatic when we do





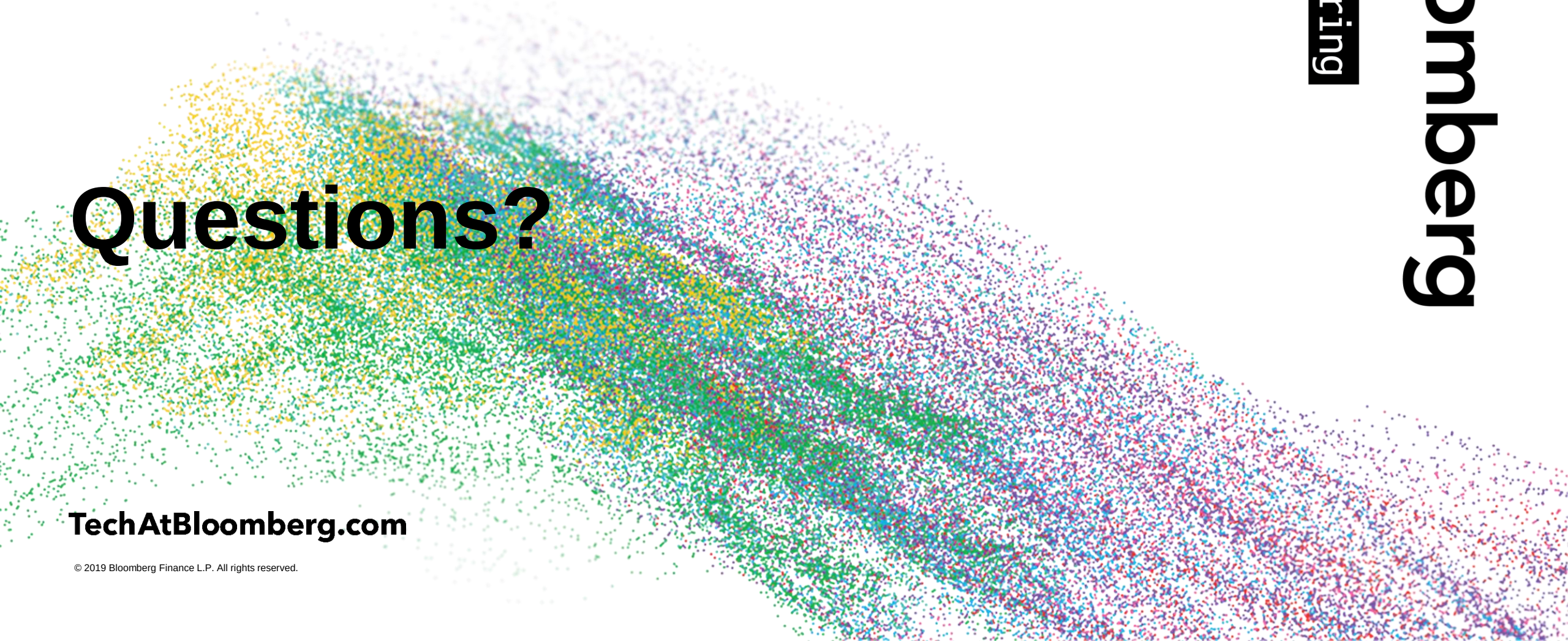
# Thank You!

https://www.bloomberg.com/careers

Bloomberg

Engineering

# Questions?



TechAtBloomberg.com