Let’s start in the beginning...

class Fraction(object):
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __mul__(self, other):
        return Fraction(self.numerator * other.numerator,
                        self.denominator * other.denominator)

    def print_fraction(self):
        print '{}/{}'.format(self.numerator, self.denominator)

>>> half = Fraction(1, 2)
>>> quarter = half * half
>>> quarter.print_fraction()
1/4
Why did we inherit `object`?

class Fraction(object):
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __mul__(self, other):
        return Fraction(self.numerator * other.numerator,
                        self.denominator * other.denominator)

    def print_fraction(self):
        print '{}/{}'.format(self.numerator, self.denominator)

• Inheritance syntax is not just for inheritance!
• Some inheritance is for metaclass propagation.
Metaclasses

- The scariest thing about metaclasses is the name.
- A metaclass is just like any other callable except that you usually call the metaclass using a class statement.
- Metaclasses let you make things that aren’t classes using the class statement.

```python
>>> class BooleanEnum(enum.Enum):
    ...    true = 1
    ...    false = 0

>>> type(BooleanEnum)
<class 'enum.EnumMeta'>
```

```python
>>> class BooleanClass(object):
    ...    true = 1
    ...    false = 0

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

```python
>>> class BooleanClass(object):
...     true = 1
...     false = 0
...     
...     type(BooleanClass)
<class 'type'>
>>> BooleanClass.true
1
>>> type(BooleanClass.true)
<class 'int'>
>>> BooleanClass()
<__main__.BooleanClass object at 0x7fcba08b8198>
```

```python
>>> class BooleanEnum(enum.Enum):
...     true = 1
...     false = 0
...     
...     type(BooleanEnum)
<class 'enum.EnumMeta'>
>>> BooleanEnum.true
<BooleanEnum.true: 1>
>>> type(BooleanEnum.true)
<enum 'BooleanEnum'>
>>> BooleanEnum()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() missing 1 required positional argument: 'value'
```

Metaclasses

- The scariest thing about metaclasses is the name.
- A metaclass is just like any other callable except that you usually call the metaclass using a class statement.
- Metaclasses let you make things that aren’t classes using the class statement.
- Metaclasses can be specified explicitly

```python
class MyABC:
    __metaclass__ = abc.ABCMeta

def some_method(self):
    data = []
```
Metaclasses

- The scariest thing about metaclasses is the name.
- A metaclass is just like any other callable except that you usually call the metaclass using a class statement.
- Metaclasses let you make things that aren’t classes using the class statement.
- Metaclasses can be specified explicitly

```python
__metaclass__ = abc.ABCMeta
class MyABC:
    def some_method(self):
        data = []
```
Metaclasses

- The scariest thing about metaclasses is the name.
- A metaclass is just like any other callable except that you usually call the metaclass using a class statement.
- Metaclasses let you make things that aren’t classes using the class statement.
- Metaclasses can be specified explicitly

```python
class MyABC:  # 2.x
    __metaclass__ = abc.ABCMeta
    def some_method(self):
        data = []
        for item in self.items:
            ...

class MyABC(metaclass=abc.ABCMeta):  # 3.x
    def some_method(self):
        data = []
        for item in self.items:
            ...
```
Metaclasses

- The scariest thing about metaclasses is the name.
- A metaclass is just like any other callable except that you usually call the metaclass using a class statement.
- Metaclasses let you make things that aren’t classes using the class statement.
- Metaclasses can be specified explicitly, but are usually taken from the parent class. Examples: Django models, SQLAlchemy models, new-style classes, the stdlib enum module, (sometimes) zope.interface interfaces.
Example - world’s most useless metaclass

```python
>>> def my_metaclass(name, bases, d):
...   print "just called:", name, bases, d
...   return 7
...   return 7

>>> class JustSeven("hello", "world"):
...   __metaclass__ = my_metaclass
just called: JustSeven ('hello', 'world') {'__module__': '__main__', '__metaclass__': <function my_metaclass at 0x7079636f6e>}

>>> print JustSeven
7
```
Example - using a metaclass without class syntax

class Fraction(object):
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __mul__(self, other):
        return Fraction(self.numerator * other.numerator,
                        self.denominator * other.denominator)

    def print_fraction(self):
        print '{}/{}'.format(self.numerator, self.denominator)

>>> type(Fraction)
<type 'type'>
Example - using a metaclass without class syntax

```python
def __init__(self, numerator, denominator):
    self.numerator = numerator
    self.denominator = denominator
def __mul__(self, other):
    return Fraction(self.numerator * other.numerator,
                    self.denominator * other.denominator)
def print_fraction(self):
    print '{}/{}'.format(self.numerator, self.denominator)
attributes = {'__init__': __init__,
              '__mul__': __mul__,
              'print_fraction': print_fraction}
Fraction = type('Fraction', (object,), attributes)
```
Metaclasses - details mostly out of scope

• `__prepare__`
  ○ Custom namespaces

• `__getdescriptor__`
  ○ soon?
Metaclasses - best practices and takeaways

- Metaclasses are invisible
  - Ask yourself: is this actually a class?
- The most common metaclass code:
  - In Python 2, there are two object systems: classic classes (`class Foo:`) and new-style classes (`class Foo(object):`)
  - The difference is their metaclass (`classobj` vs. `type`)
- For your own code, prefer class decorators to metaclasses
What happens when I make an instance?

class Fraction(object):
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __mul__(self, other):
        return Fraction(self.numerator * other.numerator,
                        self.denominator * other.denominator)

    def print_fraction(self):
        print '{}/{}'.format(self.numerator, self.denominator)

>>> half = Fraction(1, 2)
>>> quarter = half * half
>>> quarter.print_fraction()
1/4
What happens when I make an instance?

- Simple answer: the `__init__` method gets called
  - “For every problem there is an answer that is clear, simple, and wrong” -HL Mencken (paraphrased)
What happens when I make an instance?

● Creating an instance is just calling a metaclass instance.
  ○ `half = Fraction(1, 2)`

● If it is a class (not an instance of another metaclass)
  ○ `__new__` is called
    ■ `__new__` returns an instance (or something else)
  ○ if `__new__` returns an instance of the class
    ■ (e.g. `Fraction.__new__` returns a Fraction object, not some other object), `__init__` gets called
    ■ `__init__` receives an instance
    ■ `__new__` does not call `__init__`, the type calls `__init__`
Initialization - best practices and takeaways

- You can’t forget metaclasses when debugging tricky code
- When `__new__` is involved, you have to remember when `__init__` will be called and not
- If you’re defining `__new__`, write a function instead of a class
Attribute lookup

class Fraction(object):
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __mul__(self, other):
        return Fraction(self.numerator * other.numerator,
                        self.denominator * other.denominator)

    def print_fraction(self):
        print '{}/{}'.format(self.numerator, self.denominator)

>>> half = Fraction(1, 2)
>>> quarter = half * half
>>> quarter.print_fraction()
1/4
Attribute lookup

- For normal lookup, like `numerator` and `print_fraction`
  - `half.__getattribute__('numerator')` is called
    - First, the instance dictionary (or equivalent) is checked
      - This finds `self.numerator`
    - Then, the class (and all the parent classes) are checked
      - This finds `Fraction.print_fraction`
      - The descriptor protocol is invoked
    - Then, `__getattr__` is called
- For syntax that uses double-underscore attributes (like `*→__mul__`), the method is looked up *directly on the class*
Descriptors

class Fraction(object):
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __mul__(self, other):
        return Fraction(self.numerator * other.numerator,
                        self.denominator * other.denominator)

    def print_fraction(self):
        print '{}/{}'.format(self.numerator, self.denominator)

>>> half = Fraction(1, 2)
>>> quarter = half * half
>>> quarter.print_fraction()
1/4
Descriptors

- Descriptors are objects that do something when they are looked up as a class attribute (or set or deleted).
- Most common example: functions
  - Descriptors are how functions become methods
- Many language features are actually just descriptors:
  - functions/methods
  - properties
  - classmethods
  -staticmethods
Descriptors

- Getter descriptors
  - `instance.attr`

\[\text{C.__dict__['attr'].__get__(instance, C)}\]

- Plain function object
- Bound method

Becomes `self`
Descriptors

- Getter descriptors
  - `instance.attr`
    - `BaseClassWhereAttrIsDefined.__dict__['attr'].__get__(instance, type(instance))`
- `__set__` and `__delete__` work similarly
- In order to work, descriptors must be defined on a class, not on the instance itself
Descriptor gotchas

- Classes with `__call__` defined don’t automatically work as instances
  
  ```python
  class C(object):
      @decorator_class
      def decorated_method(self):
          ...
  ```

- In Python 2, Classic Classes do not fully support the descriptor protocol when used as descriptors
Attributes - best practices and takeaways

- `__getattribute__` → instance `__dict__` lookup → class `__dict__` lookup → `__getattr__`
  - `__setattr__`, `__delattr__` are a bit simpler

- Descriptors cause things to happen at lookup
  - Methods, properties, etc.
  - Callable classes (such as decorators) are not automatically method-like descriptors--`self` is not passed
  - Define new descriptors sparingly

- No code is special enough for custom `__getattribute__`
class Fraction(object):
    __slots__ = ['numerator', 'denominator']
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator
    def __mul__(self, other):
        return Fraction(self.numerator * other.numerator,
                        self.denominator * other.denominator)
    def print_fraction(self):
        print '{}/{}'.format(self.numerator, self.denominator)

>>> half = Fraction(1, 2)
>>> quarter = half * half
>>> quarter.print_fraction()
1/4
```python
class Fraction(object):
    __slots__ = ['numerator', 'denominator']
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator
    def __mul__(self, other):
        return Fraction(self.numerator * other.numerator,
                        self.denominator * other.denominator)
    def print_fraction(self):
        print '{}/{}'.format(self.numerator, self.denominator)

>>> half.real = half.numerator / half.denominator
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Fraction' object has no attribute 'real'
```
__slots__

- If an object defined in Python does not have a `__dict__`, it is using `__slots__`
  - Also possible for objects defined in C
- `__slots__` is used to save memory, not speed
- If the time has come to use `__slots__`, the time has probably come to write a C extension
All good things must come to an end

- The method `__del__` is called when Python garbage collects an object . . . **MAYBE**
- Python **does not** promise to call `__del__`
- `__del__` especially might not be called if your instance
  - is part of a reference cycle
  - survives until the Python interpreter shuts down
- If `__del__` causes a new reference to be made to an object, it won’t be garbage collected (and might be called again)
- `__del__` writes to stderr if there is an exception
__del__ - best practices and takeaways

- Don’t use __del__
- Use __del__ only as a backup
- The main ways to make sure something gets done:
  - with open(path) as f:
    data = parse_file(f)
  - try:
    x.do_work()
  finally:
    x.cleanup()
Final thoughts

- By understanding the details of how Python works, we see that **things that look simple can be very complex**
- Python provides hooks for almost everything, which are useful for
  - Understanding and using others’ code which uses them
  - Machete debugging: getting temporary debugging code to run