Requirements

- Python 3.3 or more recent
- Don't even attempt on any earlier version
- Support files:
  
  http://www.dabeaz.com/py3meta
Welcome!

- An advanced tutorial on two topics
  - Python 3
  - Metaprogramming
- Honestly, can you have too much of either?
- No!

Metaprogramming

- In a nutshell: code that manipulates code
- Common examples:
  - Decorators
  - Metaclasses
  - Descriptors
- Essentially, it's doing things with code
Why Would You Care?

- Extensively used in frameworks and libraries
- Better understanding of how Python works
- It's fun
- It solves a practical problem

DRY
DRY
Don't Repeat Yourself

DRY
Don't Repeat Yourself
Don't Repeat Yourself

- Highly repetitive code sucks
- Tedious to write
- Hard to read
- Difficult to modify

This Tutorial

- A modern journey of metaprogramming
- Highlight unique aspects of Python 3
- Explode your brain
Target Audience

• Framework/library builders
• Anyone who wants to know how things work
• Programmers wishing to increase "job security"

Reading

• Tutorial loosely based on content in "Python Cookbook, 3rd Ed."
• Published May, 2013
• You'll find even more information in the book
Preliminaries

Basic Building Blocks

```python
def func(args):
    statement1
    statement2
    statement3
...

class A:
    def method1(self, args):
        statement1
        statement2
    def method2(self, args):
        statement1
        statement2
...
```
Statements

statement1
statement2
statement3
...

- Perform the actual work of your program
- Always execute in two scopes
  - globals - Module dictionary
  - locals - Enclosing function (if any)
- exec(statements [, globals [, locals]])

Functions

def func(x, y, z):
    statement1
    statement2
    statement3
    ...

- The fundamental unit of code in most programs
- Module-level functions
- Methods of classes
Calling Conventions

def func(x, y, z):
    statement1
    statement2
    statement3
    ...

• Positional arguments
  func(1, 2, 3)

• Keyword arguments
  func(x=1, z=3, y=2)

Default Arguments

def func(x, debug=False, names=None):
    if names is None:
        names = []
    ...

    func(1)
    func(1, names=['x', 'y'])

• Default values set at definition time
• Only use immutable values (e.g., None)
*args and **kwargs

def func(*args, **kwargs):
    # args is tuple of position args
    # kwargs is dict of keyword args
    ...

    func(1, 2, x=3, y=4, z=5)

    args = (1, 2)
    kwargs = {
        'x': 3,
        'y': 4,
        'z': 5
    }

    func(*args, **kwargs)
    # same as
    func(1, 2, x=3, y=4, z=5)
Keyword-Only Args

```python
def recv(maxsize, *, block=True):
    ...

def sum(*args, initial=0):
    ...

• Named arguments appearing after '*' can only be passed by keyword

recv(8192, block=False)      # Ok
recv(8192, False)            # Error
```

Closures

• You can make and return functions

```python
def make_adder(x, y):
    def add():
        return x + y
    return add
```

• Local variables are captured

```python
>>> a = make_adder(2, 3)
>>> b = make_adder(10, 20)
>>> a()
5
>>> b()
30
>>>```
Classes

class Spam:
    a = 1
    def __init__(self, b):
        self.b = b
    def imethod(self):
        pass

>>> Spam.a                # Class variable
1
>>> s = Spam(2)
>>> s.b                  # Instance variable
2
>>> s.imethod()          # Instance method

Different Method Types

Usage

class Spam:
    def imethod(self):
        pass
        @classmethod
    def cmethod(cls):
        pass
        @staticmethod
    def smethod():
        pass

s = Spam()

s.imethod()

Spam.cmethod()

Spam.smethod()
Special Methods

class Array:
    def __getitem__(self, index):
        ...
    def __setitem__(self, index, value):
        ...
    def __delitem__(self, index):
        ...
    def __contains__(self, item):
        ...

• Almost everything can be customized

Inheritance

class Base:
    def spam(self):
        ...

class Foo(Base):
    def spam(self):
        ...
        # Call method in base class
        r = super().spam()
Dictionaries

- Objects are layered on dictionaries

```python
class Spam:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def foo(self):
        pass
```

- Example:

```python
>>> s = Spam(2,3)
>>> s.__dict__
{'y': 3, 'x': 2}
>>> Spam.__dict__['foo']
<function Spam.foo at 0x10069fc20>
```
Problem: Debugging

- Will illustrate basics with a simple problem
- Debugging
- Not the only application, but simple enough to fit on slides

Debugging with Print

- A function
  
  ```python
def add(x, y):
    return x + y
  ```

- A function with debugging
  
  ```python
def add(x, y):
    print('add')
    return x + y
  ```

- The one and only true way to debug...
Many Functions w/ Debug

def add(x, y):
    print('add')
    return x + y

def sub(x, y):
    print('sub')
    return x - y

def mul(x, y):
    print('mul')
    return x * y

def div(x, y):
    print('div')
    return x / y
Decorators

- A decorator is a function that creates a wrapper around another function
- The wrapper is a new function that works exactly like the original function (same arguments, same return value) except that some kind of extra processing is carried out

A Debugging Decorator

```python
from functools import wraps

def debug(func):
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper

# Application (wrapping)
func = debug(func)
```
A Debugging Decorator

from functools import wraps

def debug(func):
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper

A decorator creates a "wrapper" function

Around a function that you provide
Function Metadata

from functools import wraps

def debug(func):
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper

• @wraps copies metadata
  • Name and doc string
  • Function attributes

The Metadata Problem

• If you don't use @wraps, weird things happen

    def add(x,y):
        "Adds x and y"
        return x+y
    add = debug(add)

>>> add.__qualname__
'wrapper'
>>> add.__doc__
>>> help(add)
Help on function wrapper in module __main__:
__main__.wrapper(*args, **kwargs)
Decorator Syntax

• The definition of a function and wrapping almost always occur together

```python
def add(x, y):
    return x+y
add = debug(add)
```

• `@decorator syntax performs the same steps`

```python
@debug
def add(x, y):
    return x+y
```

Example Use

```python
@debug
def add(x, y):
    return x + y

@debug
def sub(x, y):
    return x - y

@debug
def mul(x, y):
    return x * y

@debug
def div(x, y):
    return x / y
```
Example Use

```python
@debug
def add(x, y):
    return x + y

@debug
def sub(x, y):
    return x - y

@debug
def mul(x, y):
    return x * y

@debug
def div(x, y):
    return x / y
```

Each function is decorated, but there are no other implementation details.

Big Picture

- Debugging code is isolated to single location
- This makes it easy to change (or to disable)
- User of a decorator doesn't worry about it
- That's really the whole idea
Variation: Logging

from functools import wraps
import logging

def debug(func):
    log = logging.getLogger(func.__module__)
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        log.debug(msg)
        return func(*args, **kwargs)
    return wrapper

Variation: Optional Disable

from functools import wraps
import os

def debug(func):
    if 'DEBUG' not in os.environ:
        return func
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper

• Key idea: Can change decorator independently of code that uses it
Debugging with Print

- A function with debugging
  
  ```python
  def add(x, y):
      print('add')
      return x + y
  ```

- Everyone knows you really need a prefix
  
  ```python
  def add(x, y):
      print('***add')
      return x + y
  ```

- You know, for grepping...

Decorators with Args

- Calling convention
  
  ```python
  @decorator(args)
  def func():
      pass
  ```

- Evaluates as
  
  ```python
  func = decorator(args)(func)
  ```

- It's a little weird--two levels of calls
Decorators with Args

from functools import wraps

def debug(prefix=' '):
    def decorate(func):
        msg = prefix + func.__qualname__
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(msg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

• Usage

@debug(prefix='***')
def add(x,y):
    return x+y
from functools import wraps, partial

def debug(func=None, *, prefix=''):  # A test of your function calling skills...
    if func is None:
        return partial(debug, prefix=prefix)

    msg = prefix + func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper

• Use as a simple decorator

    @debug
    def add(x, y):
        return x + y

• Or as a decorator with optional configuration

    @debug(prefix='***')
    def add(x, y):
        return x + y
Debug All Of This

- Debug all of the methods of a class

```python
class Spam:
    @debug
def grok(self):
        pass
    @debug
def bar(self):
        pass
    @debug
def foo(self):
        pass
```

- Can you decorate all methods at once?

Class Decorator

```
def debugmethods(cls):
    for name, val in vars(cls).items():
        if callable(val):
            setattr(cls, name, debug(val))
    return cls
```

- Idea:
  - Walk through class dictionary
  - Identify callables (e.g., methods)
  - Wrap with a decorator
Example Use

@debugmethods
class Spam:
    def grok(self):
        pass
    def bar(self):
        pass
    def foo(self):
        pass

• One decorator application
• Covers all definitions within the class
• It even mostly works...

Limitations

@debugmethods
class BrokenSpam:
    @classmethod
def grok(cls):
        pass
    @staticmethod
def bar():
        pass

• Only instance methods get wrapped
• Why? An exercise for the reader...
Variation: Debug Access

def debugattr(cls):
    orig_getattribute = cls.__getattribute__
    
    def __getattribute__(self, name):
        print('Get:', name)
        return orig_getattribute(self, name)
    cls.__getattribute__ = __getattribute__

    return cls

• Rewriting part of the class itself

Example

@debugattr
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

>>> p = Point(2, 3)
>>> p.x
Get: x
2
>>> p.y
Get: y
3
>>>
**Debug All The Classes**

```python
@debugmethods
class Base:
    ...

@debugmethods
class Spam(Base):
    ...

@debugmethods
class Grok(Spam):
    ...

@debugmethods
class Mondo(Grok):
    ...
```

- Many classes with debugging
- Didn't we just solve this?
- Bleah!!

**Solution: A Metaclass**

```python
class debugmeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsobj = super().__new__(cls, clsname, bases, clsdict)
        clsobj = debugmethods(clsobj)
        return clsobj
```

- **Usage**

```python
class Base(metaclass=debugmeta):
    ...

class Spam(Base):
    ...
```
Solution: A Metaclass

class debugmeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsobj = super().__new__(cls, clsname, bases, clsdict)
        clsobj = debugmethods(clsobj)
        return clsobj

• Idea
  • Class gets created normally

• Class gets created normally
  • Immediately wrapped by class decorator
Types

- All values in Python have a type
- Example:

```python
>>> x = 42
>>> type(x)
<type 'int'>
>>> s = "Hello"
>>> type(s)
<type 'str'>
>>> items = [1,2,3]
>>> type(items)
<type 'list'>
```
Types and Classes

• Classes define new types

```python
class Spam:
    pass
```

```python
>>> s = Spam()
>>> type(s)
<class '__main__.Spam'>
``` 

• The class is the type of instances created
• The class is a callable that creates instances

Types of Classes

• Classes are instances of types

```python
>>> type(int)
<class 'int'>
>>> type(list)
<class 'list'>
>>> type(Spam)
<class '__main__.Spam'>
>>> isinstance(Spam, type)
True
``` 

• This requires some thought, but it should make some sense (classes are types)
Creating Types

• Types are their own class (builtin)
  
  ```python
  class type:
  ... 
  ```

  ```python
  >>> type
  <class 'type'>
  >>>
  ```

• This class creates new "type" objects
• Used when defining classes

Classes Deconstructed

• Consider a class:
  
  ```python
  class Spam(Base):
    def __init__(self, name):
      self.name = name
    def bar(self):
      print "I'm Spam.bar"
  ```

• What are its components?
  • Name ("Spam")
  • Base classes (Base,)
  • Functions (__init__,bar)
Class Definition Process

• What happens during class definition?
  ```python
  class Spam(Base):
      def __init__(self, name):
          self.name = name
      def bar(self):
          print "I'm Spam.bar"
  ```

• Step 1: Body of class is isolated
  ```python
type.__prepare__('Spam', (Base,))
```

Class Definition

• Step 2: The class dictionary is created
  ```python
  clsdict = type.__prepare__('Spam', (Base,))
  ```

• This dictionary serves as local namespace for statements in the class body

• By default, it's a simple dictionary (more later)
Class Definition

- Step 3: Body is executed in returned dict
  
  ```python
  exec(body, globals(), clsdict)
  ```

- Afterwards, clsdict is populated

  ```python
  >>> clsdict
  {'__init__': <function __init__ at 0x4da10>,
   'bar': <function bar at 0x4dd70>}
  ```

- Step 4: Class is constructed from its name, base classes, and the dictionary

  ```python
  >>> Spam = type('Spam', (Base,), clsdict)
  >>> Spam
  <class '__main__.Spam'>
  >>> s = Spam('Guido')
  >>> s.bar()
  I'm Spam.bar
  ```
Changing the Metaclass

- metaclass keyword argument
- Sets the class used for creating the type

```python
class Spam(metaclass=type):
    def __init__(self, name):
        self.name = name
    def bar(self):
        print "I'm Spam.bar"
```

- By default, it's set to 'type', but you can change it to something else

Defining a New Metaclass

- You typically inherit from type and redefine `__new__` or `__init__`

```python
class mytype(type):
    def __new__(cls, name, bases, clsdict):
        clsobj = super().__new__(cls, name, bases, clsdict)
        return clsobj
```

- To use

```python
class Spam(metaclass=mytype):
    ...
```
Using a Metaclass

- Metaclasses get information about class definitions at the time of definition
  - Can inspect this data
  - Can modify this data
- Essentially, similar to a class decorator
- Question: Why would you use one?

Inheritance

- Metaclasses propagate down hierarchies

```python
class Base(metaclass=mytype):
    ...

class Spam(Base):    # metaclass=mytype
    ...

class Grok(Spam):    # metaclass=mytype
    ...
```
- Think of it as a genetic mutation
Solution: Reprise

class debugmeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsobj = super().__new__(cls, clsname, bases, clsdict)
        clsobj = debugmethods(clsobj)
        return clsobj

• Idea
  • Class gets created normally
  • Immediately wrapped by class decorator

Debug The Universe

class Base(metaclass=debugmeta):
    ...

class Spam(Base):
    ...

class Grok(Spam):
    ...

class Mondo(Grok):
    ...

• Debugging gets applied across entire hierarchy
• Implicitly applied in subclasses
Big Picture

• It's mostly about wrapping/rewriting
  • Decorators : Functions
  • Class Decorators: Classes
  • Metaclasses : Class hierarchies
• You have the power to change things

Interlude
Journey So Far

- Have seen "classic" metaprogramming
- Already widely used in Python 2
- Only a few Python 3 specific changes

Journey to Come

- Let's build something more advanced
- Using techniques discussed
- And more...
Problem : Structures

class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Host:
    def __init__(self, address, port):
        self.address = address
        self.port = port

Why must I keep writing these boilerplate init methods?
A Solution: Inheritance

```python
class Structure:
    _fields = []
def __init__(self, *args):
    if len(args) != self._fields:
        raise TypeError('Wrong # args')
    for name, val in zip(self._fields, args):
        setattr(self, name, val)

class Stock(Structure):
    _fields = ['name', 'shares', 'price']

class Point(Structure):
    _fields = ['x', 'y']

class Host(Structure):
    _fields = ['address', 'port']
```

Usage

```python
>>> s = Stock('ACME', 50, 123.45)
>>> s.name
'ACME'
>>> s.shares
50
>>> s.price
123.45

>>> p = Point(4, 5)
>>> p.x
4
>>> p.y
5
```
Some Issues

- No support for keyword args
  ```python
  >>> s = Stock('ACME', price=123.45, shares=50)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: __init__() got an unexpected keyword argument 'shares'
  ```

- Missing calling signatures
  ```python
  >>> import inspect
  >>> print(inspect.signature(Stock))
  (*args)
  ```
New Approach: Signatures

• Build a function signature object

```python
from inspect import Parameter, Signature

fields = ['name', 'shares', 'price']
parms = [ Parameter(name,
    Parameter.POSITIONAL_OR_KEYWORD)
    for name in fields]
sig = Signature(parms)
```

• Signatures are more than just metadata

Signature Binding

• Argument binding

```python
def func(*args, **kwargs):
    bound_args = sig.bind(*args, **kwargs)
    for name, val in bound_args.arguments.items():
        print(name, '=', val)
```

• `sig.bind()` binds positional/keyword args to signature
• `.arguments` is an `OrderedDict` of passed values
Signature Binding

• Example use:

```python
>>> func('ACME', 50, 91.1)
name = ACME
shares = 50
price = 91.1

>>> func('ACME', price=91.1, shares=50)
name = ACME
shares = 50
price = 91.1
```

• Notice: both positional/keyword args work

Signature Binding

• Error handling

```python
>>> func('ACME', 50)
Traceback (most recent call last):
...
TypeError: 'price' parameter lacking default value

>>> func('ACME', 50, 91.1, 92.3)
Traceback (most recent call last):
...
TypeError: too many positional arguments
```

• Binding: it just "works"
from inspect import Parameter, Signature

def make_signature(names):
    return Signature(
        Parameter(name,
            Parameter.POSITIONAL_OR_KEYWORD)
        for name in names)

class Structure:
    __signature__ = make_signature([])
    def __init__(self, *args, **kwargs):
        bound = self.__signature__.bind(*args, **kwargs)
        for name, val in bound.arguments.items():
            setattr(self, name, val)

class Stock(Structure):
    __signature__ = make_signature(['name','shares','price'])

class Point(Structure):
    __signature__ = make_signature(['x', 'y'])

class Host(Structure):
    __signature__ = make_signature(['address', 'port'])
Solution w/Signatures

```python
>>> s = Stock('ACME', shares=50, price=91.1)
>>> s.name
'ACME'
>>> s.shares
50
>>> s.price
91.1
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
```

New Problem

• This is rather annoying

```python
class Stock(Structure):
    __signature__ = make_signature(['name', 'shares', 'price'])

class Point(Structure):
    __signature__ = make_signature(['x', 'y'])

class Host(Structure):
    __signature__ = make_signature(['address', 'port'])
```

• Can't it be simplified in some way?
Solutions

- Ah, a problem involving class definitions
  - Class decorators
  - Metaclasses
- Which seems more appropriate?
- Let's explore both options

Class Decorators

```python
def add_signature(*names):
    def decorate(cls):
        cls.__signature__ = make_signature(names)
        return cls
    return decorate

Usage:
@add_signature('name','shares','price')
class Stock(Structure):
    pass

@add_signature('x','y')
class Point(Structure):
    pass
```
Metaclass Solution

class StructMeta(type):
    def __new__(cls, name, bases, clsdict):
        clsobj = super().__new__(cls, name, bases, clsdict)
        sig = make_signature(clsobj._fields)
        setattr(clsobj, '__signature__', sig)
        return clsobj

class Structure(metaclass=StructMeta):
    _fields = []
    def __init__(self, *args, **kwargs):
        bound = self.__signature__.bind(*args, **kwargs)
        for name, val in bound.arguments.items():
            setattr(self, name, val)
Usage

class Stock(Structure):
    _fields = ['name', 'shares', 'price']

class Point(Structure):
    _fields = ['x', 'y']

class Host(Structure):
    _fields = ['address', 'port']

• It's back to original 'simple' solution
• Signatures are created behind scenes

Considerations

• How much will the Structure class be expanded?

• Example: supporting methods

  class Structure(metaclass=StructMeta):
    _fields = []
    ...
    def __repr__(self):
        args = ', '.join(repr(getattr(self, name))
        for name in self._fields)
        return type(self).__name__ + '(' + args + ')

• Is type checking important?

  isinstance(s, Structure)
Advice

- Use a class decorator if the goal is to tweak classes that might be unrelated
- Use a metaclass if you're trying to perform actions in combination with inheritance
- Don't be so quick to dismiss techniques (e.g., 'metaclasses suck so .... blah blah')
- All of the tools are meant to work together

Owning the Dot

Q: "Who's in charge here?"
A: "In charge? I don't know, man."
Problem: Correctness

- Types like a duck, rhymes with ...

```python
>>> s = Stock('ACME', 50, 91.1)
>>> s.name = 42
>>> s.shares = 'a heck of a lot'
>>> s.price = (23.45 + 2j)

>>> 
```

- Bah, real programmers use Haskell!

Properties

- You can upgrade attributes to have checks

```python
class Stock(Structure):
    _fields = ['name', 'shares', 'price']

@property
    def shares(self):
        return self._shares

(getter)

@shares.setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('expected int')
        if value < 0:
            raise ValueError('Must be >= 0')
        self._shares = value
```

(setter)
Properties

• Example use:

```python
>>> s = Stock('ACME', 50, 91.1)
>>> s.shares = 'a lot'
Traceback (most recent call last):
  ...  
TypeError: expected int
>>> s.shares = -10
Traceback (most recent call last):
  ...  
ValueError: Must be >= 0
>>> s.shares = 37
>>> s.shares
37
```  

• Example use:

```python
>>> s = Stock('ACME', 50, 91.1)
>>> s.shares = 'a lot'
Traceback (most recent call last):
  ...  
TypeError: expected int
>>> s.shares = -10
Traceback (most recent call last):
  ...  
ValueError: Must be >= 0
>>> s.shares = 37
>>> s.shares
37
```  

An Issue

• It works, but it quickly gets annoying

```python
@property
def shares(self):
    return self._shares

@shares.setter
def shares(self, value):
    if not isinstance(value, int):
        raise TypeError('expected int')
    if value < 0:
        raise ValueError('Must be >= 0')
    self._shares = value
```  

• Imagine writing same code for many attributes
A Complexity

- Want to simplify, but how?
- Two kinds of checking are intertwined
- Type checking: int, float, str, etc.
- Validation: >, >=, <, <=, !=, etc.
- Question: How to structure it?

Descriptor Protocol

- Properties are implemented via descriptors
  
  ```python
  class Descriptor:
      def __get__(self, instance, cls):
          ...
      def __set__(self, instance, value):
          ...
      def __delete__(self, instance)
          ...
  
  - Customized processing of attribute access
  ```
Descriptor Protocol

• Example:

```python
class Spam:
    x = Descriptor()

s = Spam()

s.x            # x.__get__(s, Spam)
s.x = value    # x.__set__(s, value)
del s.x        # x.__delete__(s)
```

• Customized handling of a specific attribute

A Basic Descriptor

```python
class Descriptor:
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]
```
A Basic Descriptor

class Descriptor:
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]
A Simpler Descriptor

class Descriptor:
    def __init__(self, name=None):
        self.name = name

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        raise AttributeError("Can't delete")

• You don't need __get__() if it merely returns the normal dictionary value

Descriptor Usage

class Stock(Structure):
    _fields = ['name', 'shares', 'price']
    name = Descriptor('name')
    shares = Descriptor('shares')
    price = Descriptor('price')

• If it works, will capture set/delete operations

    >>> s = Stock('ACME', 50, 91.1)
    >>> s.shares
    50
    >>> s.shares = 50    # shares.__set__(s, 50)
    >>> del s.shares
    Traceback (most recent call last):
    ...
    AttributeError: Can't delete
    >>>
Type Checking

class Typed(Descriptor):
    ty = object
    def __set__(self, instance, value):
        if not isinstance(value, self.ty):
            raise TypeError('Expected %s' % self.ty)
        super().__set__(instance, value)

• Specialization

class Integer(Typed):
    ty = int
class Float(Typed):
    ty = float
class String(Typed):
    ty = str

Usage

class Stock(Structure):
    _fields = ['name', 'shares', 'price']
    name = String('name')
    shares = Integer('shares')
    price = Float('price')

• Example:

    >>> s = Stock('ACME', 50, 91.1)
    >>> s.shares = 'a lot'
    Traceback (most recent call last):
      ...
    TypeError: Expected <class 'int'>
    >>> s.name = 42
    Traceback (most recent call last):
      ...
    TypeError: Expected <class 'str'>
    >>>
Value Checking

class Positive(Descriptor):
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super().__set__(instance, value)

• Use as a mixin class

class PosInteger(Integer, Positive):
    pass

class PosFloat(Float, Positive):
    pass

Usage

class Stock(Structure):
    _fields = ['name', 'shares', 'price']
    name = String('name')
    shares = PosInteger('shares')
    price = PosFloat('price')

• Example:

    >>> s = Stock('ACME', 50, 91.1)
    >>> s.shares = -10
    Traceback (most recent call last):
      ...
    ValueError: Expected >= 0
    >>> s.shares = 'a lot'
    Traceback (most recent call last):
      ...
    TypeError: Expected <class 'int'>
Building Blocks!

class PosInteger(Integer, Positive):
    pass

Understanding the MRO

class PosInteger(Integer, Positive):
    pass

>>> PosInteger.__mro__
<class 'PosInteger'>,
<class 'Integer'>,
<class 'Typed'>,
<class 'Positive'>,
<class 'Descriptor'>,
<class 'object'>)

• Base order matters (e.g., int before < 0)

This chain defines the order in which the value is checked by different __set__() methods
Length Checking

class Sized(Descriptor):
    def __init__(self, *args, maxlen, **kwargs):
        self.maxlen = maxlen
        super().__init__(*args, **kwargs)

    def __set__(self, instance, value):
        if len(value) > self.maxlen:
            raise ValueError('Too big')
        super().__set__(instance, value)

• Use:

    class SizedString(String, Sized):
        pass

Usage

class Stock(Structure):
    _fields = ['name', 'shares', 'price']
    name = SizedString('name', maxlen=8)
    shares = PosInteger('shares')
    price = PosFloat('price')

• Example:

    >>> s = Stock('ACME', 50, 91.1)
    >>> s.name = 'ABRACADABRA'
    Traceback (most recent call last):
    ...
    ValueError: Too big
    >>>
Pattern Checking

```python
import re
class Regex(Descriptor):
    def __init__(self, *args, pat, **kwargs):
        self.pat = re.compile(pat)
        super().__init__(*args, **kwargs)

    def __set__(self, instance, value):
        if not self.pat.match(value):
            raise ValueError('Invalid string')
        super().__set__(instance, value)

• Use:

class SizedRegexString(String, Sized, Regex):
    pass
```

Usage

```python
class Stock(Structure):
    _fields = ['name', 'shares', 'price']
    name = SizedRegexString('name', maxlen=8,
                              pat='[A-Z]+$')
    shares = PosInteger('shares')
    price = PosFloat('price')

• Example:

>>> s = Stock('ACME', 50, 91.1)
>>> s.name = 'Head Explodes!'
Traceback (most recent call last):
 ... ValueError: Invalid string
```

Copyright (C) 2013, http://www.dabeaz.com
Whoa, Whoa, Whoa

• Mixin classes with `__init__()` functions?
  ```python
class SizedRegexString(String, Sized, Regex):
    pass
  ```

• Each with own unique signature
  ```python
  a = String('name')
  b = Sized(maxlen=8)
  c = Regex(pat='[A-Z]+$')
  ```

• This works, how?

Keyword-only Args

```python
SizedRegexString('name', maxlen=8, pat='[A-Z]+$')
```

```python
class Descriptor:
  def __init__(self, name=None):
    ...

class Sized(Descriptor):
  def __init__(self, *args, maxlen, **kwargs):
    ...
    super().__init__(*args, **kwargs)

class Regex(Descriptor):
  def __init__(self, *args, pat, **kwargs):
    ...
    super().__init__(*args, **kwargs)
```
Keyword-only Args

SizedRegexString('name', maxlen=8, pat='[A-Z]+$')

class Descriptor:
    def __init__(self, name=None):
        ...

class Sized(Descriptor):
    def __init__(self, *args, maxlen, **kwargs):
        ...
        super().__init__(*args, **kwargs)

class Regex(Descriptor):
    def __init__(self, *args, pat, **kwargs):
        ...
        super().__init__(*args, **kwargs)

Keyword-only argument is isolated and removed from all other passed args

"Awesome, man!"
Annoyance

class Stock(Structure):
    _fields = ['name', 'shares', 'price']
    name = SizedRegexString('name', maxlen=8,
                            pat='^[A-Z]+$')
    shares = PosInteger('shares')
    price = PosFloat('price')

• Still quite a bit of repetition
• Signatures and type checking not unified
• Maybe we can push it further
A New Metaclass

from collections import OrderedDict
class StructMeta(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return OrderedDict()

    def __new__(cls, name, bases, clsdict):
        fields = [key for key, val in clsdict.items()
                  if isinstance(val, Descriptor)]
        for name in fields:
            clsdict[name].name = name

        clsobj = super().__new__(cls, name, bases,
                                 dict(clsdict))
        sig = make_signature(fields)
        setattr(clsobj, '__signature__', sig)
        return clsobj

New Usage

class Stock(Structure):
    name = SizedRegexString(maxlen=8, pat='^[A-Z]+$')
    shares = PosInteger()
    price = PosFloat()

• Oh, that's rather nice...
New Metaclass

```python
from collections import OrderedDict
class StructMeta(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return OrderedDict()

    def __new__(cls, name, bases, clsdict):
        fields = [key for key, val in clsdict.items()
                  if isinstance(val, Descriptor)]
        for name in fields:
            clsdict[name].name = name
        clsobj = super().__new__(cls, name, bases,
            dict(clsdict))
        sig = make_signature(fields)
        setattr(clsobj, '__signature__', sig)
        return clsobj
```

__prepare__() creates and returns dictionary to use for execution of the class body.

An OrderedDict will preserve the definition order.

Ordering of Definitions

```python
class Stock(Structure):
    name = SizedRegexString(maxlen=8, pat='[A-Z]+$')
    shares = PosInteger()
    price = PosFloat()

    clsdict = OrderedDict(
        ('name', <class 'SizedRegexString'>),
        ('shares', <class 'PosInteger'>),
        ('price', <class 'PosFloat'>)
    )
```
Duplicate Definitions

- If inclined, you could do even better
- Make a new kind of dict
  
  ```python
  class NoDupOrderedDict(OrderedDict):
    def __setitem__(self, key, value):
      if key in self:
        raise NameError('%s already defined' % key)
      super().__setitem__(key, value)
  ```

- Use in place of OrderedDict

Duplicate Definitions

```python
class Stock(Structure):
  name = String()
  shares = PosInteger()
  price = PosFloat()
  shares = PosInteger()
```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in Stock
  File "./typestruct.py", line 107, in __setitem__
    raise NameError('%s already defined' % key)
NameError: shares already defined

- Won't pursue further, but you get the idea
New Metaclass

from collections import OrderedDict
class StructMeta(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return OrderedDict()

    def __new__(cls, name, bases, clsdict):
        fields = [key for key, val in clsdict.items() if isinstance(val, Descriptor)]
        for name in fields:
            clsdict[name].name = name
        clsobj = super().__new__(cls, name, bases, dict(clsdict))
        sig = make_signature(fields)
        setattr(clsobj, '__signature__', sig)
        return clsobj

Collect Descriptors and set their names

Name Setting

- Old code
  class Stock(Structure):
      _fields = ['name', 'shares', 'price']
      name = SizedRegexString('name', ...)
      shares = PosInteger('shares')
      price = PosFloat('price')

- New Code
  class Stock(Structure):
      name = SizedRegexString(...)
      shares = PosInteger()
      price = PosFloat()

Names are set from dict keys
New Metaclass

```python
from collections import OrderedDict
class StructMeta(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return OrderedDict()

    def __new__(cls, name, bases, clsdict):
        fields = [key for key, val in clsdict.items() if isinstance(val, Descriptor)]
        for name in fields:
            clsdict[name].name = name
        clsobj = super().__new__(cls, name, bases, dict(clsdict))
        sig = make_signature(fields)
        setattr(clsobj, '__signature__', sig)
        return clsobj
```

**Make the class and signature exactly as before.**

**A technicality: Must create a proper dict for class contents**

```python
from collections import OrderedDict
class StructMeta(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return OrderedDict()

    def __new__(cls, name, bases, clsdict):
        fields = [key for key, val in clsdict.items() if isinstance(val, Descriptor)]
        for name in fields:
            clsdict[name].name = name
        clsobj = super().__new__(cls, name, bases, dict(clsdict))
        sig = make_signature(fields)
        setattr(clsobj, '__signature__', sig)
        return clsobj
```
The Costs

• Option 1 : Simple

```python
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

• Option 2 : Meta

```python
class Stock(Structure):
    name = SizedRegexString(...)
    shares = PosInteger()
    price = PosFloat()
```
A Few Tests

- **Instance creation**
  
  ```
  s = Stock('ACME', 50, 91.1)
  ```

- **Attribute lookup**
  
  ```
  s.price
  ```

- **Attribute assignment**
  
  ```
  s.price = 10.0
  ```

- **Attribute assignment**
  
  ```
  s.name = 'ACME'
  ```

<table>
<thead>
<tr>
<th>Simple</th>
<th>Meta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.07s</td>
<td>91.8s</td>
</tr>
<tr>
<td></td>
<td>(86x)</td>
</tr>
<tr>
<td>0.08s</td>
<td>0.08s</td>
</tr>
<tr>
<td>0.11s</td>
<td>3.40s</td>
</tr>
<tr>
<td>(31x)</td>
<td></td>
</tr>
<tr>
<td>0.14s</td>
<td>8.14s</td>
</tr>
<tr>
<td>(58x)</td>
<td></td>
</tr>
</tbody>
</table>
Thoughts

• Several large bottlenecks
  • Signature enforcement
  • Multiple inheritance/super in descriptors
• Can anything be done without a total rewrite?

Code Generation

```python
def _make_init(fields):
    code = 'def __init__(self, %s):
        ' % ', '.join(fields)
    for name in fields:
        code += '   self.%s = %s
    return code

Example:
>>> code = _make_init(['name','shares','price'])
>>> print(code)
def __init__(self, name, shares, price):
    self.name = name
    self.shares = shares
    self.price = price

>>> 
```

Copyright (C) 2013, http://www.dabeaz.com
class StructMeta(type):
    ...
    def __new__(cls, name, bases, clsdict):
        fields = [ key for key, val in clsdict.items()
            if isinstance(val, Descriptor) ]

        for name in fields:
            clsdict[name].name = name

        if fields:
            exec(_make_init(fields),globals(),clsdict)

        clsobj = super().__new__(cls, name, bases,
            dict(clsdict))
        setattr(clsobj, '_fields', fields)
        return clsobj
New Code

class Structure(metaclass=StructMeta):
    pass

class Stock(Structure):
    name = SizedRegexString(...)
    shares = PosInteger()
    price = PosFloat()

Instance creation:
    Simple                           1.1s
    Old Meta (w/signatures)         91.8s
    New Meta (w/exec)              17.6s

New Thought

class Descriptor:
    ...
    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

class Typed(Descriptor):
    def __set__(self, instance, value):
        if not isinstance(value, self.ty):
            raise TypeError('Expected %s' % self.ty)
        super().__set__(instance, value)

class Positive(Descriptor):
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super().__set__(instance, value)
Reformulation

class Descriptor(metaclass=DescriptorMeta):
    def __init__(self, name=None):
        self.name = name

    @staticmethod
    def set_code():
        return [
            'instance.__dict__[self.name] = value'
        ]

    def __delete__(self, instance):
        raise AttributeError("Can't delete")

• Change __set__ to a method that returns source
• Introduce a new metaclass (later)

Reformulation

class Typed(Descriptor):
    ty = object

    @staticmethod
    def set_code():
        return [
            'if not isinstance(value, self.ty):',
            '    raise TypeError("Expected %s"\self.ty)'
        ]

class Positive(Descriptor):
    @staticmethod
    def set_code(self):
        return [
            'if value < 0:',
            '    raise ValueError("Expected >= 0")'
        ]
class Sized(Descriptor):
    def __init__(self, *args, maxlen, **kwargs):
        self.maxlen = maxlen
        super().__init__(*args, **kwargs)

    @staticmethod
    def set_code():
        return [
            'if len(value) > self.maxlen:','
            '    raise ValueError("Too big")'
        ]

import re

class RegexPattern(Descriptor):
    def __init__(self, *args, pat, **kwargs):
        self.pat = re.compile(pat)
        super().__init__(*args, **kwargs)

    @staticmethod
    def set_code():
        return [
            'if not self.pat.match(value):','
            '    raise ValueError("Invalid string")'
        ]
Generating a Setter

def _make_setter(dcls):
    code = 'def __set__(self, instance, value):
    for d in dcls.__mro__:
        if 'set_code' in d.__dict__:
            for line in d.set_code():
                code += '    ' + line + '
    return code

• Takes a descriptor class as input
• Walks its MRO and collects output of set_code()
• Concatenate to make a __set__() method

Example Setters

>>> print(_make_setter(Descriptor))
def __set__(self, instance, value):
    instance.__dict__[self.name] = value

>>> print(_make_setter(PosInteger))
def __set__(self, instance, value):
    if not isinstance(value, self.ty):
        raise TypeError("Expected %s" % self.ty)
    if value < 0:
        raise ValueError("Expected >= 0")
    instance.__dict__[self.name] = value

>>>
## Descriptor Metaclass

```python
class DescriptorMeta(type):
    def __init__(self, clsname, bases, clsdict):
        if '__set__' not in clsdict:
            code = _make_setter(self)
            exec(code, globals(), clsdict)
            setattr(self, '__set__',
                    clsdict['__set__'])
        else:
            raise TypeError('Define set_code()')

class Descriptor(metaclass=DescriptorMeta):
    ...
```

- For each Descriptor class, create setter code
- `exec()` and drop result onto created class

### Just to be Clear

```python
class Stock(Structure):
    name = SizedRegexString(...)  
    shares = PosInteger()         
    price = PosFloat()
```

- User has no idea about this code generation
- They're just using the same code as before
- It's an implementation detail of descriptors
## New Performance

- **Instance creation**
  
  ```python
  s = Stock('ACME',50,91.1)  # 1.07s
  ```

- **Attribute lookup**
  
  ```python
  s.price  # 0.08s
  ```

- **Attribute assignment**
  
  ```python
  s.price = 10.0  # 0.11s
  ```

- **Attribute assignment**
  
  ```python
  s.name = 'ACME'  # 0.14s
  ```

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>Meta</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = Stock('ACME',50,91.1)</td>
<td>1.07s</td>
<td>91.8s (86x)</td>
<td>7.19s (6.7x)</td>
</tr>
<tr>
<td>s.price</td>
<td>0.08s</td>
<td>0.08s</td>
<td>0.08s</td>
</tr>
<tr>
<td>s.price = 10.0</td>
<td>0.11s</td>
<td>3.40s (31x)</td>
<td>1.11s (10x)</td>
</tr>
<tr>
<td>s.name = 'ACME'</td>
<td>0.14s</td>
<td>8.14s (58x)</td>
<td>2.95s (21x)</td>
</tr>
</tbody>
</table>

---

## The Horror! The Horror!

[@alex_gaynor](http://twitter.com/alex_gaynor)
Remaining Problem

- Convincing a manager about all of this

```python
class Stock(Structure):
    name = SizedRegexString(maxlen=8, pat='[A-Z]+$')
    shares = PosInteger()
    price = PosFloat()

class Point(Structure):
    x = Integer()
    y = Integer()

class Address(Structure):
    hostname = String()
    port = Integer()
```

Solution: XML

```xml
<structures>
    <structure name="Stock">
        <field type="SizedRegexString" maxlen="8" pat="'[A-Z]+$'">name</field>
        <field type="PosInteger">shares</field>
        <field type="PosFloat">price</field>
    </structure>
    <structure name="Point">
        <field type="Integer">x</field>
        <field type="Integer">y</field>
    </structure>
    <structure name="Address">
        <field type="String">hostname</field>
        <field type="Integer">port</field>
    </structure>
</structures>
```
Solution: XML

```xml
<structures>
  <structure name="Stock">
    <field type="SizedRegexString" maxlen="8" pat="^[A-Z]+$">name</field>
    <field type="PosInteger">shares</field>
    <field type="PosFloat">price</field>
  </structure>
  <structure name="Point">
    <field type="Integer">x</field>
    <field type="Integer">y</field>
  </structure>
  <structure name="Address">
    <field type="String">hostname</field>
    <field type="Integer">port</field>
  </structure>
</structures>
```

+5 extra credit
Regex + XML

XML to Classes

- XML Parsing

```python
from xml.etree.ElementTree import parse

def _xml_to_code(filename):
    doc = parse(filename)
    code = 'import typestruct as _ts
for st in doc.findall('structure'):
    code += _xml_struct_code(st)
return code
```

- Continued...
XML to Classes

```python
def _xml_struct_code(st):
    stname = st.get('name')
    code = 'class %s(_ts.Structure):
        ' % stname
    for field in st.findall('field'):
        name = field.text.strip()
        dtype = '_ts.' + field.get('type')
        kwargs = ', '.join('%s=%s' % (key, val)
                          for key, val in field.items()
                          if key != 'type')
        code += '   %s = %s(%s)
        ' % (name, dtype, kwargs)
    return code
```

Example

```python
>>> code = _xml_to_code('data.xml')
>>> print(code)
import typestruct as _ts
class Stock(_ts.Structure):
    name = _ts.SizedRegexString(maxlen=8, pat='[A-Z]+')
    shares = _ts.PosInteger()
    price = _ts.PosFloat()
class Point(_ts.Structure):
    x = _ts.Integer()
    y = _ts.Integer()
class Address(_ts.Structure):
    hostname = _ts.String()
    port = _ts.Integer()

>>> 
```
Now WHAT!?!?
- Allow structure .xml files to be imported
- Using the import statement
- Yes!

Import Hooks

- `sys.meta_path`
  ```python
  >>> import sys
  >>> sys.meta_path
  [<class '_frozen_importlib.BuiltinImporter'>,
   <class '_frozen_importlib.FrozenImporter'>,
   <class '_frozen_importlib.PathFinder'>]
  >>>
  ```
  - A collection of importer/finder instances
An Experiment

class MyImporter:
    def find_module(self, fullname, path=None):
        print('*** Looking for', fullname)
        return None

>>> sys.meta_path.append(MyImporter())
>>> import foo

*** Looking for foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'foo'

• Yes, you've plugged into the import statement

Structure Importer

class StructImporter:
    def __init__(self, path):
        self._path = path

    def find_module(self, fullname, path=None):
        name = fullname.rpartition('.')[1]
        if path is None:
            path = self._path
        for dn in path:
            filename = os.path.join(dn, name+'.xml')
            if os.path.exists(filename):
                return StructXmlLoader(filename)
        return None
class StructImporter:
    def __init__(self, path):
        self._path = path

    def find_module(self, fullname, path=None):
        name = fullname.rpartition('.')[-1]
        if path is None:
            path = self._path
        for dn in path:
            filename = os.path.join(dn, name+'.xml')
            if os.path.exists(filename):
                return StructXmlLoader(filename)
        return None

Walk path, check for existence of .xml file and return a loader
import imp

class StructXMLLoader:
    def __init__(self, filename):
        self._filename = filename

    def load_module(self, fullname):
        mod = sys.modules.setdefault(fullname,
                                      imp.new_module(fullname))
        mod.__file__ = self._filename
        mod.__loader__ = self
        code = _xml_to_code(self._filename)
        exec(code, mod.__dict__, mod.__dict__)
        return mod
XML Module Loader

```python
import imp
class StructXMLLoader:
    def __init__(self, filename):
        self._filename = filename

    def load_module(self, fullname):
        mod = sys.modules.setdefault(fullname,
            imp.new_module(fullname))
        mod.__file__ = self._filename
        mod.__loader__ = self
        code = _xml_to_code(self._filename)
        exec(code, mod.__dict__, mod.__dict__)
        return mod
```

Convert XML to code and `exec()` resulting source

Installation and Use

- Add to `sys.meta_path`

  ```python
def install_importer(path=sys.path):
    sys.meta_path.append(StructImporter(path))

install_importer()
```

- From this point, structure .xml files will import

  ```
  >>> import datadefs
  >>> s = datadefs.Stock('ACME', 50, 91.1)
  >>> s.name
  'ACME'
  >>> datadefs
  <module 'datadefs' from './datadefs.xml'>
  ```

```
Look at the Source

```python
>>> datadefs
<module 'datadefs' from './datadefs.xml'>

>>> import inspect
>>> print(inspect.getsource(datadefs))
<structures>
    <structure name="Stock">
        <field type="SizedRegexString" maxlen="8" pat="'[A-Z]+$'">name</field>
        <field type="PosInteger">shares</field>
        <field type="PosFloat">price</field>
    </structure>
...
```

Final Thoughts

(probably best to start packing up)
Extreme Power

• Think about all of the neat things we did

   class Stock(Structure):
       name = SizedRegexString(maxlen=8, pat='[A-Z]+$')
       shares = PosInteger()
       price = PosFloat()

• Descriptors as building blocks

• Hiding of annoying details (signatures, etc.)

• Dynamic code generation

• Even customizing import

Hack or by Design?

• Python 3 is designed to do this sort of stuff
  • More advanced metaclasses (e.g., __prepare__)
  • Signatures
  • Import hooks
  • Keyword-only args

• Observe: I didn't do any mind-twisting "hacks" to work around a language limitation.
Python 3 FTW!

• Python 3 makes a lot of little things easier

• Example: Python 2 keyword-only args
  ```python
def __init__(self, *args, **kwargs):
    self.maxlen = kwargs.pop('maxlen')
...
```

• In Python 3
  ```python
def __init__(self, *args, maxlen, **kwargs):
    self.maxlen = maxlen
...
```

• There are a lot of little things like this

Just the Start

• We've only scratched the surface

• Function annotations
  ```python
def add(x:int, y:int) -> int:
    return x + y
```

• Non-local variables
  ```python
def outer():
    x = 0
    def inner():
        nonlocal x
        x = newvalue
    ...
```
Just the Start

- Context managers
  ```python
  with m:
    ...
  ```
- Frame-hacks
  ```python
  import sys
  f = sys._getframe(1)
  ```
- Parsing/AST-manipulation
  ```python
  import ast
  ```

You Can, But Should You?

- Metaprogramming is not for "normal" coding
- Frameworks/libraries are a different story
- If using a framework, you may be using this features without knowing it
- You can do a lot of cool stuff
- OTOH: Keeping it simple is not a bad strategy
That is All!

- Thanks for listening
- Hope you learned a few new things
- Buy the "Python Cookbook, 3rd Ed." (O'Reilly)
- Twitter: @dabeaz