Metaclass Programming with Python

Torsten Marek
<marek@ifi.uzh.ch>

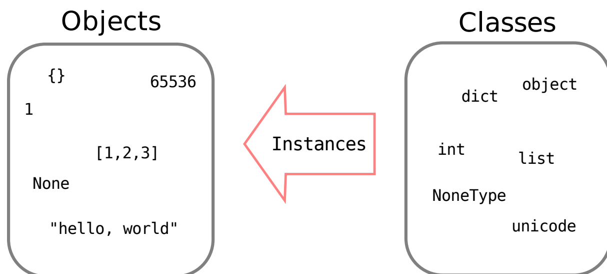
Universität Zürich
Institut für Computerlinguistik

June 30th, 2009

1. The Object Model for Classes
2. Customizing Class Creation
3. Applications of Metaclasses
4. Metaclasses and You

Tiny Little World

Tiny Little World



Classes and Objects

- abstract vs. concrete
- behavior of objects is determined by their classes
- “normal” code works with concrete objects

Classes as Objects

Everything is an Object

- classes live in the same space as their instances
- classes are "normal" objects

Classes as Objects

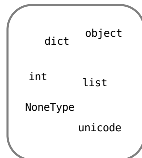
Everything is an Object

- classes live in the same space as their instances
- classes are "normal" objects

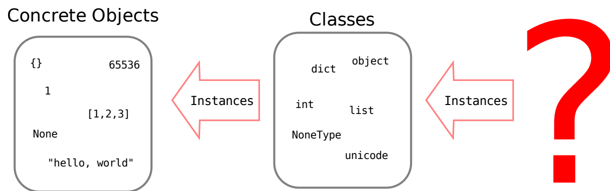
Examples

```
>>> intdict = defaultdict(int)
>>> intdict["abc"]
0
>>> ts = [int, float, str, unicode, list, dict]
>>> [t for t in ts if hasattr(t, "__len__")]
[<type 'str'>, <type 'unicode'>,
 <type 'list'>, <type 'dict'>]
>>> [t() for t in ts]
[0, 0.0, '', u'', [], {}]
```

Classes as Objects II



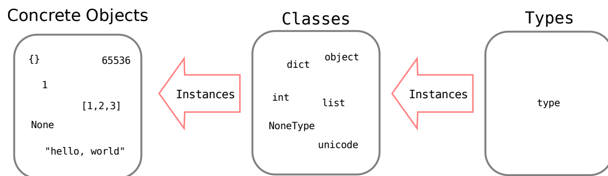
Classes as Objects II



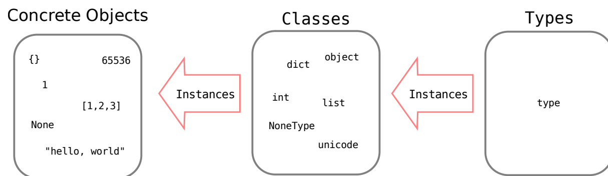
Classes as Objects

- every object has a type, which determines its behavior
- classes must be instances of *something*
- what is the class of a class?

Types/Metaclasses



Types/Metaclasses



Classes of Classes

- Classes are derived from metaclasses/types
- `type` is for classes what `object` is for instances

Instances, everywhere

`type`

- classes are instances of `type`
- `type` is again just an object, living in the same space
- Can we create arbitrary classes at runtime?

Instances, everywhere

`type`

- classes are instances of `type`
- `type` is again just an object, living in the same space
- Can we create arbitrary classes at runtime?

Two Faces of the `type` Builtin

- `type(obj)`: returns the type of an object
  - `isinstance(obj, type(obj)) == True`
- `type(name, bases, dict)`: **returns a new type**

type at Work

Creating Types at Runtime

```
def say(self, msg):
    print msg

Speaker = type("Speaker", (object, ), {"say": say})
s = Speaker()
s.say("Hello World!")
```

Arguments

type at Work

Creating Types at Runtime

```
def say(self, msg):
    print msg

Speaker = type("Speaker", (object, ), {"say": say})
s = Speaker()
s.say("Hello World!")
```

Arguments

1. type name
   - Speaker.__class__.__name__

type at Work

Creating Types at Runtime

```
def say(self, msg):
    print msg

Speaker = type("Speaker", (object, ), {"say": say})
s = Speaker()
s.say("Hello World!")
```

Arguments

1. type name
   - `Speaker.__class__.__name__`
2. base classes
   - `Speaker.__class__.__bases__`

type at Work

Creating Types at Runtime

```
def say(self, msg):
    print msg

Speaker = type("Speaker", (object, ), {"say": say})
s = Speaker()
s.say("Hello World!")
```

Arguments

1. type name
   - Speaker.__class__.__name__
2. base classes
   - Speaker.__class__.__bases__
3. class dictionary
   - basis for Speaker.__class__.__dict__

Spot the Difference

```
def say(self, msg):
  print msg
Speaker = type("Speaker",
               (object, ),
               {"say": say})
```

```
class Speaker(object):
  def say(self, msg):
    print msg
```

Spot the Difference

```python
def say(self, msg):
  print msg
Speaker = type("Speaker",
               (object, ),
               {"say": say})
```

```python
class Speaker(object):
  def say(self, msg):
    print msg
```

Solution

- There is (close to) none
- `class` statement is *syntactic sugar*
- makes sure namespaces stay clean
- **Important**: class object is created after statement block

Instantiation vs. Inheritance

Two Different Relationships

- Instantiation: between template and concrete object
  - *Bender is an actual bending unit.*
- Inheritance: between general and specific template
  - *A bending unit is a special kind of robot.*

Instantiation vs. Inheritance

Two Different Relationships

- Instantiation: between template and concrete object
  - *Bender is an actual bending unit.*
- Inheritance: between general and specific template
  - *A bending unit is a special kind of robot.*

- `isinstance(1, int)`
- `isinstance(1, object)`
- `issubclass(int, object)`
- `isinstance(int, type)`
- `isinstance(object, type)`

- `issubclass(1, int)`
- `issubclass(1, object)`
- `isinstance(int, object)`
- `issubclass(int, type)`
- `issubclass(object, type)`
- `isinstance(1, type)`

We can create new objects and classes.

Can we also create new **metaclasses**?

1 The Object Model for Classes

2 Customizing Class Creation

3 Applications of Metaclasses

4 Metaclasses and You

Custom Metaclasses

Changing the Meta Class

- by default, Python uses `type` as a metaclass of a newly-created class
- custom metaclass can be set with the magic attribute `__metaclass__`

Custom Metaclasses

Changing the Meta Class

- by default, Python uses `type` as a metaclass of a newly-created class
- custom metaclass can be set with the magic attribute `__metaclass__`

`__metaclass__` Contract

- can be any callable object
- must take three arguments: name, bases and dictionary
- should return a new type that has been created using `type`

Interface Demo

Implementation

```python
def boring_meta(name, bases, dict_):
    print "Name: ", name
    print "Bases: ", bases
    print "Class Dictionary: ", ", ".join(dict_)
    return type(name, bases, dict_)

class SomeClass(object):
    __metaclass__ = boring_meta
    def foo(self): pass
```

Interface Demo

Implementation

```
def boring_meta(name, bases, dict_):
    print "Name: ", name
    print "Bases: ", bases
    print "Class Dictionary: ", ", ".join(dict_)
    return type(name, bases, dict_)

class SomeClass(object):
    __metaclass__ = boring_meta
    def foo(self): pass
```

```
Name:  SomeClass
Bases:  (<type 'object'>,)
Members  __module__, foo, __metaclass__
```

A More “Useful” Example

Problem

- 3rd-party modules that use classes with camel-case-methods
- you happen to dislike camel case (as you should)

A More “Useful” Example

Problem

- 3rd-party modules that use classes with camel-case-methods
- you happen to dislike camel case (as you should)

Solution

- write a metaclass that turns PEP-8 style function names into camel case names
- write your classes according to your style, camel case names are never seen by you

The Camelizer

Implementation

```
import types

def pep8_to_cc(n): ...

def add_camelcase(cls_name, bases, dict_):
  for name, obj in dict_.items():
    if (not name.startswith("_")
        and isinstance(obj, types.FunctionType)):
        dict_[pep8_to_cc(name)] = obj
  return type(cls_name, bases, dict_)
```

The Camelizer

Implementation

```
import types

def pep8_to_cc(n): ...

def add_camelcase(cls_name, bases, dict_):
  for name, obj in dict_.items():
    if (not name.startswith("_")
        and isinstance(obj, types.FunctionType)):
       dict_[pep8_to_cc(name)] = obj
  return type(cls_name, bases, dict_)
```

```
>>> class SomeClass(object):
...   __metaclass__ = add_camelcase
...   def say_ni(self): print "Ni!"
>>> SomeClass().sayNi()
```

Something Actually Useful

A Real Problem

- super-class method is overridden in class
- name of super-class method changes, e.g. due to refactoring
- method name in derived class is not updated

Something Actually Useful

A Real Problem

- super-class method is overridden in class
- name of super-class method changes, e.g. due to refactoring
- method name in derived class is not updated

Inspiration from Java

- `@Override` annotation on methods as a hint to the compiler
- if annotated method does not override anything, compilation results in a failure

Compliance Checks in Metaclasses

Application for Metaclasses

- test should be carried out ...
  - as **early** as possible
  - exactly **once**

Compliance Checks in Metaclasses

Application for Metaclasses

- test should be carried out ...
  - as **early** as possible
  - exactly **once**
- ideal time: class creation

Compliance Checks in Metaclasses

Application for Metaclasses

- test should be carried out ...
  - as **early** as possible
  - exactly **once**
- ideal time: class creation

Solution Sketch

- method decorator `@override`
- metaclass that checks all decorated methods
- **Warning:** code is not production-ready

Implementation

The Decorator

```
def override(func):
  func._must_override = True
  return func
```

Implementation

The Decorator

```
def override(func):
  func._must_override = True
  return func
```

Comments

- just stores some metadata on the function
- used similar to Java's annotations
  ...though decorators are much more powerful
- **Important**: class object does not exist at this point, so constraint cannot be checked right away

The Metaclass

```
def get_all_bases(classes):
  return list(
    chain.from_iterable(inspect.getmro(c)
                        for c in classes))

class MethodCheckerType(type):
  def __new__(mcs, cls_name, bases, dict_):
    all_bases = get_all_bases(bases)
    for name, obj in dict_.iteritems():
      if getattr(obj, "_must_override", False):
        assert any(hasattr(base_cls, name)
                   for base_cls in all_bases)

    return type.__new__(mcs, cls_name, bases, dict_)
```

In Use

```
>>> class BaseClass(object):
...     __metaclass__ = MethodCheckerType
...     def foo(self): pass

>>> class WrongDerivedClass(BaseClass):
...     @override
...     def do_foo(self): pass
AssertionError
```

In Use

```
>>> class BaseClass(object):
...     __metaclass__ = MethodCheckerType
...     def foo(self): pass

>>> class WrongDerivedClass(BaseClass):
...     @override
...     def do_foo(self): pass
AssertionError
```

Remarks & Extensions

- does not work if annotation is forgotten
- better error message
- should check if superclass object is actually a method
- warn if overridden methods are not annotated

Metaclasses in Python 3.0

Class Decorators

- similar to method decorators, just for classes
- post-processing for classes
- more declarative

Metaclasses in Python 3.0

Class Decorators

- similar to method decorators, just for classes
- post-processing for classes
- more declarative

PEP-3115: Rationale

- post-processing can be done by class decorators
- metaclasses come fairly late in the class creation process
- allow the metaclass to provide the dictionary which is used for storing the namespace

Metaclass Changes

Use Cases for Early Involvement

- preserve order of declarations
  - automatic mapping to C structs
- injection of volatile field constructors
- forward references within a class body

Metaclass Changes

Use Cases for Early Involvement

- preserve order of declarations
  - automatic mapping to C structs
- injection of volatile field constructors
- forward references within a class body

Implementation

- metaclass types can provide a method `__prepare__`
- called with name and bases
- returns a mapping object which is used to store the namespace
- **optional!**

Extended Syntax for Class Definitions

PEP-3115: Rationale

- metaclass can provide the namespace
- must be known before class body is evaluated
- no way to pass arguments to metaclasses directly

Extended Syntax for Class Definitions

PEP-3115: Rationale

- metaclass can provide the namespace
- must be known before class body is evaluated
- no way to pass arguments to metaclasses directly

Extended Syntax

```
class Foo(base1, metaclass=mymeta, private=True):
  ...
```

- `metaclass` to specify the metaclass object
- other keyword arguments are passed to the metaclass

1 The Object Model for Classes

2 Customizing Class Creation

3 Applications of Metaclasses

4 Metaclasses and You

Before We Begin

Metaclasses and You

- metaclasses are just one way to customize classes
- customization prior to *instantiation* is just one way to have customized classes

The Code Ahead

- real (or almost . . .) examples
- consume responsibly!

Object-Relational Mappers

The Idea

- classes whose attributes can be mapped to tables in relational databases (and back)
- knowledge about attributes needed
  - data type
  - relation to other objects
  - possible constraints
- ideally done in a intuitive & declarative style

Object-Relational Mappers

The Idea

- classes whose attributes can be mapped to tables in relational databases (and back)
- knowledge about attributes needed
    - data type
    - relation to other objects
    - possible constraints
- ideally done in a intuitive & declarative style

Challenges

- needs extensive compliance checks
- database table must be created from field descriptions
- efficient way to enumerate all fields at runtime
- no leaky abstractions

Example: Django Models

Model Example

```
class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
```

Taken from http://docs.djangoproject.com/en/dev/topics/db/models/

Example: Django Models

Model Example

```
class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
```

Taken from http://docs.djangoproject.com/en/dev/topics/db/models/

Using a model

```
a = Musician(name="Everett",
             first_name="Mark", instrument="Guitar")
a.save()
```

Django Model Guts

`django.models.Model`

- sets the metaclass to `django.models.ModelBase`
- provides standard behavior like `save`

Django Model Guts

`django.models.Model`

- sets the metaclass to `django.models.ModelBase`
- provides standard behavior like `save`

`django.models.ModelBase`

- creates the class
- ensures that data fields are added to the class
- registers models at a central location

Lazy Attribute Lookup

Problem

- PyQt4 has code to compile or load UIs dynamically
- parsing & object setup is always the same
- needless code duplication in UI parser

Lazy Attribute Lookup

Problem

- PyQt4 has code to compile or load UIs dynamically
- parsing & object setup is always the same
- needless code duplication in UI parser

Solution

- use metaclasses to provide proxy/recording objects when generating code
- all function all and object accesses get recorded
- only instantiation of UI classes needs to be changed, all other code in UI loader is the same
- either runs on recording or actual UI objects

Proxy Namespace

```
>>> class QtCore(ProxyNamespace): pass
>>> str(QtCore.QDate(1, 2, 3))
'QtCore.QDate(1, 2, 3)'
>>> str(qtproxies.QtCore.QDate(1, 2, 3)\
                .toString().length())
'QtCore.QDate(1, 2, 3).toString().length()'
```

- some widget classes need special handling for some methods
- but can be written as a normal classes

Proxy Namespace

```
>>> class QtCore(ProxyNamespace): pass
>>> str(QtCore.QDate(1, 2, 3))
'QtCore.QDate(1, 2, 3)'
>>> str(qtproxies.QtCore.QDate(1, 2, 3)\
                .toString().length())
'QtCore.QDate(1, 2, 3).toString().length()'
```

- some widget classes need special handling for some methods
- but can be written as a normal classes

ProxyType Metaclass

- customized attribute lookup for class objects
- on-the-fly generation of literal objects
- sets namespace information
- not a generic framework, customized for the UI loading use case

Call Dispatch

Problem

- certain methods in a class need to be registered as event handlers
- examples: visitors, parsers
- dispatch dictionary should be known before event dispatch
- declarative way for registering dispatch methods preferred

Call Dispatch

Problem

- certain methods in a class need to be registered as event handlers
- examples: visitors, parsers
- dispatch dictionary should be known before event dispatch
- declarative way for registering dispatch methods preferred

Solution

- annotate event handlers via decorators
- create dispatch dictionary in metaclass

Event-based Parsing with ElementTree

```
class HtmlParser(IterParseHandler):
  @element("p")
  def _handle_paragraph(self, elem):
    ...

  @element("div")
  @element("span")
  def _handle_neutral(self, elem):
    ...

hp = HtmlParser()
with open("some_file", "r") as f:
  hp.parse(f)
```

Event-based Parsing with ElementTree

```
class HtmlParser(IterParseHandler):
  @element("p")
  def _handle_paragraph(self, elem):
    ...

  @element("div")
  @element("span")
  def _handle_neutral(self, elem):
    ...

hp = HtmlParser()
with open("some_file", "r") as f:
  hp.parse(f)
```

Remarks

- use schema to enforce that all elements are handled

Modified Classes

Far-reaching Changes

- metaclasses allow heavy modifications
- modified attribute lookup
- changed object instantiation behavior
- magic methods for class objects

Java, again: Enums

Yes, I did program a lot of Java

```
public enum Planet {
    public static final double G = 6.67300e-11;
    VENUS   (4.869e+24, 6.0518e+6),
    EARTH   (5.976e+24, 6.37814e+6),
    MARS    (6.421e+23, 3.3972e+6),
    ...
    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public double mass()   { return mass; }
    public double radius() { return radius; }
    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
}
```

Adapted from http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html

Enums, Adapted

Adaptation

- Java code is still quite verbose
- most boilerplate code can be provided by metaclasses

Enums, Adapted

Adaptation

- Java code is still quite verbose
- most boilerplate code can be provided by metaclasses

```
import enum

class Planet(enum.Enum):
    G = 6.67300e-11;

    __fields__ = ["mass", "radius"]

    VENUS = enum.member(4.869e+24, 6.0518e+6)
    EARTH = enum.member(5.976e+24, 6.37814e+6)
    MARS  = enum.member(6.421e+23, 3.3972e+6)

    @property
    def surface_gravity(self):
        return self.G * self.mass / self.radius**2
```

Enum's Inner Workings

`enum.Enum`: base class

- sets the metaclass
- provides equality tests, `__repr__`

Enum's Inner Workings

enum.Enum: base class

- sets the metaclass
- provides equality tests, `__repr__`

enum._EnumType: the metaclass

- creates the instances, correct number of fields etc.
- ensures immutability
- creates property objects for all fields
- provides methods for iterating over all enum members
- each enum class object (**not instance**) is a sequence

Enums.v3

Putting New Metaclasses to Use

```
import enum3

class Planet(enum3.Enum, fields=["mass", "radius"]):
    G = 6.67300e-11;

    VENUS = Planet(4.869e+24, 6.0518e+6)
    EARTH = Planet(5.976e+24, 6.37814e+6)
    MARS  = Planet(6.421e+23, 3.3972e+6)

    @property
    def surface_gravity(self):
        return self.G * self.mass / self.radius**2
```

- even possible to define fields in class body

1. The Object Model for Classes
2. Customizing Class Creation
3. Applications of Metaclasses
4. Metaclasses and You

Clever Code

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*
>
> — Brian W. Kernighan

How to Use Metaclasses

Hide Metaclasses

- do not expose directly
- only set via base classes
- allows for easy transitions to other techniques

How to Use Metaclasses

Hide Metaclasses

- do not expose directly
- only set via base classes
- allows for easy transitions to other techniques

Where to Use

- tool for library/framework writers
- overkill for one-off situations

Don't Use Metaclasses

Technical Problems

- easily depend on implementation details
- conflicts with other metaclasses
- hard-to-find/strange bugs
- almost always an easier solution

Don't Use Metaclasses

Technical Problems

- easily depend on implementation details
- conflicts with other metaclasses
- hard-to-find/strange bugs
- almost always an easier solution

Social/Philosophical Problems

- other programmers likely won't understand your code
- break default assumptions on code
    - ...it's what they are there for, after all

Alternatives to Metaclasses

Concrete Problems

- modified attribute access
    - `__getattr__`, descriptors
- post-processing (adding extra functions, registering)
    - explicit function calls
    - class decorators

Alternatives to Metaclasses

Concrete Problems

- modified attribute access
  - `__getattr__`, descriptors
- post-processing (adding extra functions, registering)
  - explicit function calls
  - class decorators

Other Techniques

- mixins
- descriptors
- code generation

A Summary, In Questions

Do I Need to Know About Metaclasses?

Deliberately **avoiding knowledge** won't make you a better programmer.

A Summary, In Questions

Do I Need to Know About Metaclasses?

Deliberately **avoiding knowledge** won't make you a better programmer.

Should I Use Metaclasses?

Make your own mistakes, **but learn from them**.

A Summary, In Questions

Do I Need to Know About Metaclasses?

Deliberately **avoiding knowledge** won't make you a better programmer.

Should I Use Metaclasses?

Make your own mistakes, **but learn from them**.

What Do I Gain From Metaclasses?

Elegant & intuitive code in **classes** through complex code in **metaclasses** and loss of explicitness.

The End

Thanks a lot for your attention!

Questions? Comments?