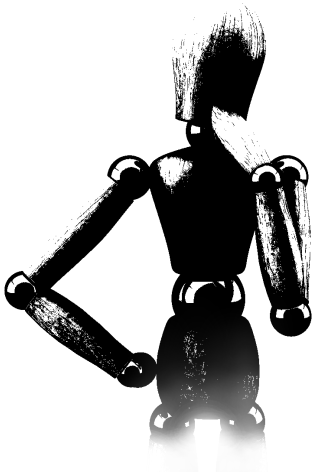
mannequin Documentation

Release 0.1

Dustin Lacewell

Aug 04, 2018

Contents



Contents:

CHAPTER 1

Writing Models

1.1 Code Listing

The main class in **mannequin** is the `Model`. It represents your object and contains all of the data fields provided in its declaration. Other than that, the `Model` base-class provides only a few other methods for the book-keeping of said `Fields`.

Here is the `Model` implementation:

```
class Model(object):
    version = "unknown"

    def gather_fields(self, cls=None, sub=None):
        """Utility to locate class-attributed Fields"""
        assert cls or sub

        self.fields = dict()

        def _gather_fields():
            for name, value in vars(type(self)).items():
                if cls:
                    if isinstance(value, cls):
                        self.bind_field(name, value)
                elif sub:
                    if issubclass(type(value), sub):
                        self.bind_field(name, value)

        _gather_fields()
        self.fields_gathered(self.fields)

    def bind_field(self, name, field):
        field.parent = self
        self.fields[name] = field
```

(continues on next page)

(continued from previous page)

```
    def fields_gathered(self, fields):
        pass
```

1.2 Initialization

When you instantiate your `Model` subclasses, they will already feature the various **Field** descriptors you defined on those subclasses. In fact, nothing about the declarative technique supported by **mannequin** depends on any of the code in this class.

The `gather_fields` method here is provided purely as a convenience. It introspects the class and determines each of the instance attributes are subclasses of the **mannequin** `Field` type. It stores each of `Fields` into a `.fields` dictionary on the `Model` so that you can use for whatever you'd like.

While gathering the fields, the `bind_field` method will be called for each. This is one place you can hook in the case you happen to need to "post-process" each field. The other place is the `fields_gathered` method which will be called with the final dictionary of Fields.

As you can see the `Model` class is very minimal. However this makes more room for your application specific methods. These models are your data objects after all and your subclasses will likely feature a number of methods relevant to that type, in addition to any Fields you put there.

CHAPTER 2

Writing Fields

2.1 Descriptors!

The second class in **mannequin** is the `Field` and it is slightly more interesting than the `Model`. It represents the various *data attributes* of your objects but does so in an interesting way. The `Field` class is what is called a **Descriptor** in Python. This is a *special object* that, when assigned to the attribute of a class, takes on some special properties.

For a full explanation see the Descriptor How-to Guide

Essentially, a Descriptor is an object that implements a `__get__` and a `__set__` method. When a Descriptor instance is assigned to an **attribute of a class**, instances of that class will aquire the Descriptor. Because the attribute is a Descriptor, **all access and assignment** to that attribute is *controlled by these methods* on the Descriptor. Here is a trivial example of a Descriptor that returns squares of it's internal value:

```
class SquaredDescriptor(object):
    def __init__(self):
        self.__value = None

    def __get__(self, obj, obj_type):
        try:
            return self.__value * self.__value
        except TypeError:
            return self.__value

    def __set__(self, obj, value):
        self.__value = value


class Dummy(object):
    squared = SquaredDescriptor()
```

We can see this descriptor in action in the interactive session below:

```
# first create an instance of the Dummy class
>>> obj = Dummy()

# our instance has the `squared` attribue
>>> print obj.squared
None

# if we assign a numerical value...
>>> obj.squared = 5

# it is squared when accessed
>>> print obj.squared
25

# according to the implementation
>>> obj.squared = "five"

# non-numeric values should be returned as-is
>>> print obj.squared
five
```

The nature of Descriptors is what makes the `Field` class interesting and useful. Since assignment can be mediated through the `Field` it can provide *data sanitation* or *parsing* benefits. The base `Field` class has a couple methods already for implementing such behaviors. Here is the base `Field` implementation below:

```
class Field(object):

    defaults = tuple()

    def __init__(self, **kwargs):
        for (defname, defvalue) in self.defaults:
            if defname in kwargs:
                setattr(self, defname, kwargs.pop(defname))
            if not hasattr(self, defname):
                setattr(self, defname, dict(self.defaults)[defname])

        if kwargs:
            raise TypeError("Unexpected field initialization parameters: " +
                ', '.join(kwargs.keys()))

    def clean(self, value):
        return value

    def validate(self, value):
        pass

    def __get__(self, obj, objtype):
        return getattr(obj, "__field_{0}".format(id(self)))

    def __set__(self, obj, value):
        cleaned_value = self.clean(value)
        self.validate(cleaned_value)
        setattr(obj, '__field_{0}'.format(id(self)), cleaned_value)
```

When you instantiate a `Field` class the first thing that the base implementation does, is loop through the `Field.defaults`. This tuple designates what the **optional** initialization parameters are for the `Field`. Any keyword arguments passed to the `Field` that it doesn't expect will raise a `TypeError`. Any *missing* keyword arguments will be given the corresponding default from the `defaults` attribute.

The second important thing about the `Field` class is that it is a *Descriptor*. Once instantiated and assigned to a `Model` Class declaration, all access and assignment will be regulated by the `Field` instance. We can see that the base `Field` implementation provides some basic handling here:

```
def __set__(self, obj, value):
    cleaned_value = self.clean(value)
    self.validate(cleaned_value)
    setattr(obj, '__field_{0}'.format(id(self)), cleaned_value)
```

When we assign a value to a Field descriptor a few things happen. The first is that the value is passed to `Field.clean()`. The default implementation simply returns the value "as is"; however this is a great method in which you can provide your own santiation or other parsing operations. Next, the *cleaned value* is passed to `Field.validate()`. The default implementation here does nothing, but you can stick in your own validation code by overriding the method.

Lastly, once your Field considers the data value as cleaned and validated, the value is stored in a slightly obsfucated manner. Foremost, the value is stored on the ***Model instance*** that the `Field` is bound to. The attribute name given to the value interpolates the *Python object ID* of the current ***Field instance***. The current `Field` instance is used so that multiple `Fields` of the same type can be bound to the same `Model`.

```
def __get__(self, obj, objtype):
    return getattr(obj, "__field_{0}".format(id(self)))
```

Here we can see that when accessing the `Field` value, the same attribute name is generated and the value is returned.

2.2 Continuing from here

Now that you have a good idea about how both mannequin `Models` and `Fields` work, head over to the XML Parser Tutorial to see how a real library can be written using mannequin and the declarative technique.

CHAPTER 3

Glossary

application protocol In computer network programming, the application layer is an abstraction layer reserved for communications protocols and methods designed for process-to- process communications across an Internet Protocol (IP) computer network. Application layer protocols use the underlying transport layer protocols to establish host-to-host connections.

binary stream A binary file is a computer file that is not a text file; it may contain any type of data, encoded in binary form for computer storage and processing purposes. Many binary file formats contain parts that can be interpreted as text; for example, some computer document files containing formatted text, such as older Microsoft Word document files, contain the text of the document but also contain formatting information in binary form.

declarative In computer science, declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow. Many languages applying this style attempt to minimize or eliminate side effects by describing what the program should accomplish, rather than describing how to go about accomplishing it (the how is left up to the language's implementation). This is in contrast with imperative programming, in which algorithms are implemented in terms of explicit steps.

django Django is an open source web application framework, written in Python, which follows the model–view–controller architectural pattern. It was originally developed to manage several news- oriented sites for The World Company of Lawrence, Kansas, and was released publicly under a BSD license in July 2005; the framework was named after guitarist Django Reinhardt. In June 2008 it was announced that a newly formed Django Software Foundation will maintain Django in the future. https://djangoproject.com/

namedtuple Returns a new tuple subclass named typename. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful __repr__() method which lists the tuple contents in a name=value format. http://docs.python.org/2/library/collections.html#collections.namedtuple

package A Python package is a module defined by a directory, containing a `__init__.py` file, and can contain other modules or other packages within it.

```
package/
    __init__.py
    subpackage/
```

(continues on next page)

(continued from previous page)

```
        __init__.py
    submodule.py
```

see also, namespace package

packet In computer networking, a packet is a formatted unit of data carried by a packet mode computer network. Computer communications links that do not support packets, such as traditional point-to-point telecommunications links, simply transmit data as a series of bytes, characters, or bits alone. When data is formatted into packets, the bitrate of the communication medium can be better shared among users than if the network were circuit switched.

struct This module performs conversions between Python values and C structs represented as Python strings. This can be used in handling binary data stored in files or from network connections, among other sources. It uses Format Strings as compact descriptions of the layout of the C structs and the intended conversion to/from Python values. http://docs.python.org/2/library/struct.html

Full Documentation: http://readthedocs.org/docs/mannequin/

CHAPTER 4

Overview

mannequin is very simple.

It is a small library that helps you create *declarative* models for your own libraries and applications using Python class definitions. *Declarative* models are a nice way to define the *structure of your data or objects*. Using Python classes for this keeps it natural and familiar.

If you've ever encountered the Python web-framework *Django* you might be familiar with it's `Models` or `Forms`. Django uses this *declarative* technique to allow you to naturally define tables in your database:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

Or the types and ordering of fields of your web-forms:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

There are other database libraries that use this technique to represent database schemas like SQLAlchemy and Axiom.

Let's look at a theoretical example of unpacking binary data from structures. With the standard *struct* module this can be a painful exercise. Given some imagined *packet* structure, unpacking a *binary stream* into the various fields is cumbersome:

```
header = struct.unpack('B',data[0])
length = struct.unpack('B',data[1])
typeID = struct.unpack('!I',data[2:6])
param1 = struct.unpack('!H',data[6:8])
```

(continues on next page)

(continued from previous page)

```
param2 = struct.unpack('!H',data[8:10])
param3 = struct.unpack('!H',data[10:12])
param4 = struct.unpack('!H',data[12:14])
name = struct.unpack('20s',data[14:38])
checksum = struct.unpack('!I',data[38:42])
footer = struct.unpack('B',data[42])
```

Yikes! Even if we ask `struct` to unpack all of the fields at once, we are then relegated to numerical indexing. We can use *namedtuple* but we still have the feeling that there has to be a better way:

```
fields = struct.unpack('!BBI4H20sIB', data)

fields[0] # get the header

# this might be a more comfortable alternative, perhaps:

(header, length, typeID, param1, param2,
param3, param4, name_string, checksum, footer,
) = struct.unpack("!2B I 4H 24s I B", data)
```

One could imagine a library that uses the same sort of *class-based schema delcaratives* that Django does to solve this problem. Here is a hypothetical definition of the same packet structure as above:

```
class TCPPacket(PacketModel):

    endian = BIG

    header = fields.Byte()
    length = fields.Byte()
    type = fields.Integer()
    params = fields.List(4, fields.Short())
    name = fields.String(20)
    checksum = fields.Integer()
    footer = fields.Byte()
```

The obvious advantage here is *readability*. But there are some other not so obvious advantages. The fact that this packet declaration is a class means that it can be *subclassed into more specific implementations*, perhaps adding *additional fields*. If we were implementing an *application protocol* we could implement the header of our protocol in a base class and use that in the actual implementation of our various packet types.

Another advantage is that it keeps the *handling* of each specific packet *close to the structure definition*. Each class declarative can contain methods specific to usage inside your application.

Since we are using `Field` objects to define the types of our various packet fields we also gain the ability to do *implicit validation on data*. For example, if we had an application protocol that featured an authentication mechanism the Field classes can work harder for us than in the `TCPPacket` example:

```
class UsernameField(fields.String):
    def __init__(self):
        # UsernameField provides an implicit length to String
        super(UsernameField, self).__init__(32)

    def clean(self, value):
        try:
            # lookup user in database
            # and return it
```

(continues on next page)

(continued from previous page)

```
            return User.objects.get(username=cleaned)
        except User.DoesNotExist, e:
            # ValidationError indicates this field failed
            # to clean
            raise ValidationError(e.message)

    def validate(self, cleaned):
        # recieves actual user instance from self.clean()
        if not user.active:
            msg = "%s is not a currently activated user." % cleaned.username
            raise ValidationError(msg) # indicate failure to validate

class PasswordField(fields.String):
    def __init__(self):
        # PasswordField provides an implicit length to String
        super(UsernameField, self).__init__(32)

    def validate(self, cleaned):
        user = self.parent.user
        try:
            # check the password for the user
            user.check_password(cleaned)
            # authenticate user if no exception
            user.authenticate()
        except AuthenticationError, e:
            raise ValidationError(e.message)

class LoginPacket(MyAppPacket):
    # MyAppPacket provides MyApp's protocol header fields
    user = UsernameField() # verifies user exists in database
    password = PasswordField() # authenticates user on validation
```

CHAPTER 5

Getting Started

The easiest way to get started is to checkout the examples in the source repository. It may be beneficial to read about `Models` and `Fields`. You may also enjoy the tutorial which describes how to use **mannequin** to create a declarative XML parser.

Read the *Writing Models* documentation to get started.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index