CHAPTER 19

Dynamic Attributes and Properties

> The crucial importance of properties is that their existence makes it perfectly safe and indeed advisable for you to expose public data attributes as part of your class's public interface.[1]
>
> — Alex Martelli
> *Python contributor and book author*

Data attributes and methods are collectively known as *attributes* in Python: a method is just an attribute that is *callable*. Besides data attributes and methods, we can also create properties, which can be used to replace a public data attribute with *accessor methods* (i.e., getter/setter), without changing the class interface. This agrees with the *Uniform access principle*:

> All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.[2]

Besides properties, Python provides a rich API for controlling attribute access and implementing dynamic attributes. The interpreter calls special methods such as `__getattr__` and `__setattr__` to evaluate attribute access using dot notation (e.g., `obj.attr`). A user-defined class implementing `__getattr__` can implement "virtual attributes" by computing values on the fly whenever somebody tries to read a nonexistent attribute like `obj.no_such_attribute`.

Coding dynamic attributes is the kind of metaprogramming that framework authors do. However, in Python, the basic techniques are so straightforward that anyone can put them to work, even for everyday data wrangling tasks. That's how we'll start this chapter.

1. Alex Martelli, *Python in a Nutshell, 2E* (O'Reilly), p. 101.
2. Bertrand Meyer, *Object-Oriented Software Construction*, 2E, p. 57.

Data Wrangling with Dynamic Attributes

In the next few examples, we'll leverage dynamic attributes to work with a JSON data feed published by O'Reilly for the OSCON 2014 conference. Example 19-1 shows four records from that data feed.[3]

Example 19-1. Sample records from osconfeed.json; some field contents abbreviated

```
{ "Schedule":
  { "conferences": [{"serial": 115 }],
    "events": [
      { "serial": 34505,
        "name": "Why Schools Don´t Use Open Source to Teach Programming",
        "event_type": "40-minute conference session",
        "time_start": "2014-07-23 11:30:00",
        "time_stop": "2014-07-23 12:10:00",
        "venue_serial": 1462,
        "description": "Aside from the fact that high school programming...",
        "website_url": "http://oscon.com/oscon2014/public/schedule/detail/34505",
        "speakers": [157509],
        "categories": ["Education"] }
    ],
    "speakers": [
      { "serial": 157509,
        "name": "Robert Lefkowitz",
        "photo": null,
        "url": "http://sharewave.com/",
        "position": "CTO",
        "affiliation": "Sharewave",
        "twitter": "sharewaveteam",
        "bio": "Robert ´r0ml´ Lefkowitz is the CTO at Sharewave, a startup..." }
    ],
    "venues": [
      { "serial": 1462,
        "name": "F151",
        "category": "Conference Venues" }
    ]
  }
}
```

Example 19-1 shows 4 out of the 895 records in the JSON feed. As you can see, the entire dataset is a single JSON object with the key `"Schedule"`, and its value is another mapping with four keys: `"conferences"`, `"events"`, `"speakers"`, and `"venues"`. Each of those four keys is paired with a list of records. In Example 19-1, each list has one record, but in the full dataset, those lists have dozens or hundreds of records—with the exception

3. You can read about this feed and rules for using it at "DIY: OSCON schedule". The original 744KB JSON file is still online as I write this. A copy named *osconfeed.json* can be found in the *oscon-schedule/data/* directory in the *Fluent Python* code repository.

of "conferences", which holds just the single record shown. Every item in those four lists has a "serial" field, which is a unique identifier within the list.

The first script I wrote to deal with the OSCON feed simply downloads the feed, avoiding unnecessary traffic by checking if there is a local copy. This makes sense because OSCON 2014 is history now, so that feed will not be updated.

There is no metaprogramming in Example 19-2; pretty much everything boils down to this expression: `json.load(fp)`, but that's enough to let us explore the dataset. The `osconfeed.load` function will be used in the next several examples.

Example 19-2. osconfeed.py: downloading osconfeed.json (doctests are in Example 19-3)

```
from urllib.request import urlopen
import warnings
import os
import json

URL = 'http://www.oreilly.com/pub/sc/osconfeed'
JSON = 'data/osconfeed.json'


def load():
    if not os.path.exists(JSON):
        msg = 'downloading {} to {}'.format(URL, JSON)
        warnings.warn(msg)   ❶
        with urlopen(URL) as remote, open(JSON, 'wb') as local:   ❷
            local.write(remote.read())

    with open(JSON) as fp:
        return json.load(fp)   ❸
```

❶ Issue a `warning` if a new download will be made.

❷ `with` using two context managers (allowed since Python 2.7 and 3.1) to read the remote file and save it.

❸ The `json.load` function parses a JSON file and returns native Python objects. In this feed, we have the types: `dict`, `list`, `str`, and `int`.

With the code in Example 19-2, we can inspect any field in the data. See Example 19-3.

Example 19-3. osconfeed.py: doctests for Example 19-2

```
    >>> feed = load()   ❶
    >>> sorted(feed['Schedule'].keys())   ❷
    ['conferences', 'events', 'speakers', 'venues']
    >>> for key, value in sorted(feed['Schedule'].items()):
    ...     print('{:3} {}'.format(len(value), key))   ❸
    ...
```

```
  1 conferences
484 events
357 speakers
 53 venues
>>> feed['Schedule']['speakers'][-1]['name']   ❹
'Carina C. Zona'
>>> feed['Schedule']['speakers'][-1]['serial']   ❺
141590
>>> feed['Schedule']['events'][40]['name']
'There *Will* Be Bugs'
>>> feed['Schedule']['events'][40]['speakers']   ❻
[3471, 5199]
```

❶ `feed` is a `dict` holding nested dicts and lists, with string and integer values.

❷ List the four record collections inside `"Schedule"`.

❸ Display record counts for each collection.

❹ Navigate through the nested dicts and lists to get the name of the last speaker.

❺ Get serial number of that same speaker.

❻ Each event has a `'speakers'` list with 0 or more speaker serial numbers.

Exploring JSON-Like Data with Dynamic Attributes

Example 19-2 is simple enough, but the syntax `feed['Schedule']['events'][40]['name']` is cumbersome. In JavaScript, you can get the same value by writing `feed.Schedule.events[40].name`. It's easy to implement a `dict`-like class that does the same in Python—there are plenty of implementations on the Web.[4] I implemented my own `FrozenJSON`, which is simpler than most recipes because it supports reading only: it's just for exploring the data. However, it's also recursive, dealing automatically with nested mappings and lists.

Example 19-4 is a demonstration of `FrozenJSON` and the source code is in Example 19-5.

Example 19-4. FrozenJSON from Example 19-5 allows reading attributes like name and calling methods like .keys() and .items()

```
>>> from osconfeed import load
>>> raw_feed = load()
>>> feed = FrozenJSON(raw_feed)   ❶
>>> len(feed.Schedule.speakers)   ❷
357
>>> sorted(feed.Schedule.keys())   ❸
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed.Schedule.items()):   ❹
...     print('{:3} {}'.format(len(value), key))
```

4. An often mentioned one is AttrDict; another, allowing quick creation of nested mappings is addict.

```
...
  1 conferences
484 events
357 speakers
 53 venues
>>> feed.Schedule.speakers[-1].name   ❺
'Carina C. Zona'
>>> talk = feed.Schedule.events[40]
>>> type(talk)   ❻
<class 'explore0.FrozenJSON'>
>>> talk.name
'There *Will* Be Bugs'
>>> talk.speakers   ❼
[3471, 5199]
>>> talk.flavor   ❽
Traceback (most recent call last):
  ...
KeyError: 'flavor'
```

❶ Build a `FrozenJSON` instance from the `raw_feed` made of nested dicts and lists.

❷ `FrozenJSON` allows traversing nested dicts by using attribute notation; here we show the length of the list of speakers.

❸ Methods of the underlying dicts can also be accessed, like `.keys()`, to retrieve the record collection names.

❹ Using `items()`, we can retrieve the record collection names and their contents, to display the `len()` of each of them.

❺ A `list`, such as `feed.Schedule.speakers`, remains a list, but the items inside are converted to `FrozenJSON` if they are mappings.

❻ Item 40 in the `events` list was a JSON object; now it's a `FrozenJSON` instance.

❼ Event records have a `speakers` list with speaker serial numbers.

❽ Trying to read a missing attribute raises `KeyError`, instead of the usual `AttributeError`.

The keystone of the `FrozenJSON` class is the `__getattr__` method, which we already used in the `Vector` example in "Vector Take #3: Dynamic Attribute Access" on page 284, to retrieve `Vector` components by letter—`v.x`, `v.y`, `v.z`, etc. It's essential to recall that the `__getattr__` special method is only invoked by the interpreter when the usual process fails to retrieve an attribute (i.e., when the named attribute cannot be found in the instance, nor in the class or in its superclasses).

The last line of Example 19-4 exposes a minor issue with the implementation: ideally, trying to read a missing attribute should raise `AttributeError`. I actually did implement the error handling, but it doubled the size of the `__getattr__` method and distracted from the most important logic I wanted to show, so I left it out for didactic reasons.

As shown in Example 19-5, the `FrozenJSON` class has only two methods (`__init__`, `__getattr__`) and a `__data` instance attribute, so attempts to retrieve an attribute by any other name will trigger `__getattr__`. This method will first look if the `self.__data` `dict` has an attribute (not a key!) by that name; this allows `FrozenJSON` instances to handle any `dict` method such as `items`, by delegating to `self.__data.items()`. If `self.___data` doesn't have an attribute with the given `name`, `__getattr__` uses `name` as a key to retrieve an item from `self.__dict`, and passes that item to `FrozenJSON.build`. This allows navigating through nested structures in the JSON data, as each nested mapping is converted to another `FrozenJSON` instance by the `build` class method.

Example 19-5. explore0.py: turn a JSON dataset into a FrozenJSON holding nested FrozenJSON objects, lists, and simple types

```
from collections import abc


class FrozenJSON:
    """A read-only façade for navigating a JSON-like object
       using attribute notation
    """

    def __init__(self, mapping):
        self.__data = dict(mapping)   ❶

    def __getattr__(self, name):   ❷
        if hasattr(self.__data, name):
            return getattr(self.__data, name)   ❸
        else:
            return FrozenJSON.build(self.__data[name])   ❹

    @classmethod
    def build(cls, obj):   ❺
        if isinstance(obj, abc.Mapping):   ❻
            return cls(obj)
        elif isinstance(obj, abc.MutableSequence):   ❼
            return [cls.build(item) for item in obj]
        else:   ❽
            return obj
```

❶ Build a `dict` from the `mapping` argument. This serves two purposes: ensures we got a `dict` (or something that can be converted to one) and makes a copy for safety.

❷ `__getattr__` is called only when there's no attribute with that `name`.

❸ If `name` matches an attribute of the instance `__data`, return that. This is how calls to methods like `keys` are handled.

❹ Otherwise, fetch the item with the key `name` from `self.__data`, and return the result of calling `FrozenJSON.build()` on that.[5]

❺ This is an alternate constructor, a common use for the `@classmethod` decorator.

❻ If `obj` is a mapping, build a `FrozenJSON` with it.

❼ If it is a `MutableSequence`, it must be a list,[6] so we build a `list` by passing every item in `obj` recursively to `.build()`.

❽ If it's not a `dict` or a `list`, return the item as it is.

Note that no caching or transformation of the original feed is done. As the feed is traversed, the nested data structures are converted again and again into `FrozenJSON`. But that's OK for a dataset of this size, and for a script that will only be used to explore or convert the data.

Any script that generates or emulates dynamic attribute names from arbitrary sources must deal with one issue: the keys in the original data may not be suitable attribute names. The next section addresses this.

The Invalid Attribute Name Problem

The `FrozenJSON` class has a limitation: there is no special handling for attribute names that are Python keywords. For example, if you build an object like this:

```
>>> grad = FrozenJSON({'name': 'Jim Bo', 'class': 1982})
```

You won't be able to read `grad.class` because `class` is a reserved word in Python:

```
>>> grad.class
  File "<stdin>", line 1
    grad.class
             ^
SyntaxError: invalid syntax
```

You can always do this, of course:

```
>>> getattr(grad, 'class')
1982
```

But the idea of `FrozenJSON` is to provide convenient access to the data, so a better solution is checking whether a key in the mapping given to `FrozenJSON.__init__` is a keyword, and if so, append an `_` to it, so the attribute can be read like this:

5. This line is where a `KeyError` exception may occur, in the expression `self.__data[name]`. It should be handled and an `AttributeError` raised instead, because that's what is expected from `__getattr__`. The diligent reader is invited to code the error handling as an exercise.
6. The source of the data is JSON, and the only collection types in JSON data are `dict` and `list`.

```
>>> grad.class_
1982
```

This can be achieved by replacing the one-liner `__init__` from Example 19-5 with the version in Example 19-6.

Example 19-6. explore1.py: append a _ to attribute names that are Python keywords

```
    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if keyword.iskeyword(key):    ❶
                key += '_'
            self.__data[key] = value
```

❶ The `keyword.iskeyword(…)` function is exactly what we need; to use it, the `keyword` module must be imported, which is not shown in this snippet.

A similar problem may arise if a key in the JSON is not a valid Python identifier:

```
>>> x = FrozenJSON({'2be':'or not'})
>>> x.2be
  File "<stdin>", line 1
    x.2be
      ^
SyntaxError: invalid syntax
```

Such problematic keys are easy to detect in Python 3 because the `str` class provides the `s.isidentifier()` method, which tells you whether `s` is a valid Python identifier according to the language grammar. But turning a key that is not a valid identifier into valid attribute name is not trivial. Two simple solutions would be raising an exception or replacing the invalid keys with generic names like `attr_0`, `attr_1`, and so on. For the sake of simplicity, I will not worry about this issue.

After giving some thought to the dynamic attribute names, let's turn to another essential feature of `FrozenJSON`: the logic of the `build` class method, which is used by `__getattr__` to return a different type of object depending on the value of the attribute being accessed, so that nested structures are converted to `FrozenJSON` instances or lists of `FrozenJSON` instances.

Instead of a class method, the same logic could be implemented as the `__new__` special method, as we'll see next.

Flexible Object Creation with `__new__`

We often refer to `__init__` as the constructor method, but that's because we adopted jargon from other languages. The special method that actually constructs an instance is `__new__`: it's a class method (but gets special treatment, so the `@classmethod` decorator

is not used), and it must return an instance. That instance will in turn be passed as the first argument `self` of `__init__`. Because `__init__` gets an instance when called, and it's actually forbidden from returning anything, `__init__` is really an "initializer." The real constructor is `__new__`—which we rarely need to code because the implementation inherited from `object` suffices.

The path just described, from `__new__` to `__init__`, is the most common, but not the only one. The `__new__` method can also return an instance of a different class, and when that happens, the interpreter does not call `__init__`.

In other words, the process of building an object in Python can be summarized with this pseudocode:

```
# pseudo-code for object construction
def object_maker(the_class, some_arg):
    new_object = the_class.__new__(some_arg)
    if isinstance(new_object, the_class):
        the_class.__init__(new_object, some_arg)
    return new_object

# the following statements are roughly equivalent
x = Foo('bar')
x = object_maker(Foo, 'bar')
```

Example 19-7 shows a variation of `FrozenJSON` where the logic of the former `build` class method was moved to `__new__`.

Example 19-7. explore2.py: using new instead of build to construct new objects that may or may not be instances of FrozenJSON

```
from collections import abc


class FrozenJSON:
    """A read-only façade for navigating a JSON-like object
       using attribute notation
    """

    def __new__(cls, arg):  ❶
        if isinstance(arg, abc.Mapping):
            return super().__new__(cls)  ❷
        elif isinstance(arg, abc.MutableSequence):  ❸
            return [cls(item) for item in arg]
        else:
            return arg

    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if iskeyword(key):
                key += '_'
```

```
            self.__data[key] = value

    def __getattr__(self, name):
        if hasattr(self.__data, name):
            return getattr(self.__data, name)
        else:
            return FrozenJSON(self.__data[name])    ❹
```

❶ As a class method, the first argument `__new__` gets is the class itself, and the remaining arguments are the same that `__init__` gets, except for `self`.

❷ The default behavior is to delegate to the `__new__` of a super class. In this case, we are calling `__new__` from the `object` base class, passing `FrozenJSON` as the only argument.

❸ The remaining lines of `__new__` are exactly as in the old `build` method.

❹ This was where `FrozenJSON.build` was called before; now we just call the `FrozenJSON` constructor.

The `__new__` method gets the class as the first argument because, usually, the created object will be an instance of that class. So, in `FrozenJSON.__new__`, when the expression `super().__new__(cls)` effectively calls `object.__new__(FrozenJSON)`, the instance built by the `object` class is actually an instance of `FrozenJSON`—i.e., the `__class__` attribute of the new instance will hold a reference to `FrozenJSON`—even though the actual construction is performed by `object.__new__`, implemented in C, in the guts of the interpreter.

There is an obvious shortcoming in the way the OSCON JSON feed is structured: the event at index `40`, titled `'There *Will* Be Bugs'` has two speakers, `3471` and `5199`, but finding them is not easy, because those are serial numbers, and the `Schedule.speakers` list is not indexed by them. The `venue` field, present in every `event` record, also holds the a serial number, but finding the corresponding `venue` record requires a linear scan of the `Schedule.venues` list. Our next task is restructuring the data, and then automating the retrieval of linked records.

Restructuring the OSCON Feed with shelve

The funny name of the standard `shelve` module makes sense when you realize that `pickle` is the name of the Python object serialization format—and of the module that converts objects to/from that format. Because pickle jars are kept in shelves, it makes sense that `shelve` provides `pickle` storage.

The `shelve.open` high-level function returns a `shelve.Shelf` instance—a simple key-value object database backed by the `dbm` module, with these characteristics:

- `shelve.Shelf` subclasses `abc.MutableMapping`, so it provides the essential methods we expect of a mapping type
- In addition, `shelve.Shelf` provides a few other I/O management methods, like `sync` and `close`; it's also a context manager.
- Keys and values are saved whenever a new value is assigned to a key.
- The keys must be strings.
- The values must be objects that the `pickle` module can handle.

Consult the documentation for the `shelve`, `dbm`, and `pickle` modules for the details and caveats. What matters to us now is that `shelve` provides a simple, efficient way to reorganize the OSCON schedule data: we will read all records from the JSON file and save them to a `shelve.Shelf`. Each key will be made from the record type and the serial number (e.g., `'event.33950'` or `'speaker.3471'`) and the value will be an instance of a new `Record` class we are about to introduce.

Example 19-8 shows the doctests for the *schedule1.py* script using `shelve`. To try it out interactively, run the script as `python -i schedule1.py` to get a console prompt with the module loaded. The `load_db` function does the heavy work: it calls `oscon feed.load` (from Example 19-2) to read the JSON data and saves each record as a `Record` instance in the `Shelf` object passed as `db`. After that, retrieving a speaker record is as easy as `speaker = db['speaker.3471']`.

Example 19-8. Trying out the functionality provided by schedule1.py (Example 19-9)

```
>>> import shelve
>>> db = shelve.open(DB_NAME)   ❶
>>> if CONFERENCE not in db:   ❷
...     load_db(db)   ❸
...
>>> speaker = db['speaker.3471']   ❹
>>> type(speaker)   ❺
<class 'schedule1.Record'>
>>> speaker.name, speaker.twitter   ❻
('Anna Martelli Ravenscroft', 'annaraven')
>>> db.close()   ❼
```

❶ `shelve.open` opens an existing or just-created database file.

❷ A quick way to determine if the database is populated is to look for a known key, in this case *`conference.115`*—the key to the single `conference` record.[7]

❸ If the database is empty, call `load_db(db)` to load it.

7. I could also do `len(db)`, but that would be costly in a large `dbm` database.

❹ Fetch a `speaker` record.

❺ It's an instance of the `Record` class defined in Example 19-9.

❻ Each `Record` instance implements a custom set of attributes reflecting the fields of the underlying JSON record.

❼ Always remember to close a `shelve.Shelf`. If possible, use a `with` block to make sure the `Shelf` is closed.[8]

The code for *schedule1.py* is in Example 19-9.

Example 19-9. schedule1.py: exploring OSCON schedule data saved to a shelve.Shelf

```
import warnings

import osconfeed    ❶

DB_NAME = 'data/schedule1_db'
CONFERENCE = 'conference.115'


class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)    ❷


def load_db(db):
    raw_data = osconfeed.load()    ❸
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():    ❹
        record_type = collection[:-1]    ❺
        for record in rec_list:
            key = '{}.{}'.format(record_type, record['serial'])    ❻
            record['serial'] = key    ❼
            db[key] = Record(**record)    ❽
```

❶ Load the *osconfeed.py* module from Example 19-2.

❷ This is a common shortcut to build an instance with attributes created from keyword arguments (detailed explanation follows).

❸ This may fetch the JSON feed from the Web, if there's no local copy.

❹ Iterate over the collections (e.g., `'conferences'`, `'events'`, etc.).

❺ `record_type` is set to the collection name without the trailing `'s'` (i.e., `'events'` becomes `'event'`).

❻ Build `key` from the `record_type` and the `'serial'` field.

8. A fundamental weakness of doctest is the lack of proper resource setup and guaranteed tear-down. I wrote most tests for *schedule1.py* using `py.test`, and you can see them at Example A-12.

❼ Update the `'serial'` field with the full `key`.

❽ Build `Record` instance and save it to the database under the `key`.

The `Record.__init__` method illustrates a popular Python hack. Recall that the `__dict__` of an object is where its attributes are kept—unless `__slots__` is declared in the class, as we saw in "Saving Space with the __slots__ Class Attribute" on page 264. So, updating an instance `__dict__` with a mapping is a quick way to create a bunch of attributes in that instance.[9]

I am not going to repeat the details we discussed earlier in "The Invalid Attribute Name Problem" on page 591, but depending on the application context, the `Record` class may need to deal with keys that are not valid attribute names.

The definition of `Record` in Example 19-9 is so simple that you may be wondering why we did not use it before, instead of the more complicated `FrozenJSON`. There are a couple reasons. First, `FrozenJSON` works by recursively converting the nested mappings and lists; `Record` doesn't need that because our converted dataset doesn't have mappings nested in mappings or lists. The records contain only strings, integers, lists of strings, and lists of integers. A second reason is that `FrozenJSON` provides access to the embedded `__data dict` attributes—which we used to invoke methods like `keys`—and now we don't need that functionality either.

The Python standard library provides at least two classes similar to our `Record`, where each instance has an arbitrary set of attributes built from keyword arguments to the constructor: `multiprocessing.Namespace` (documentation, source code), and `argparse.Namespace` (documentation, source code). I implemented `Record` to highlight the essence of the idea: `__init__` updating the instance `__dict__`.

After reorganizing the schedule dataset as we just did, we can now extend the `Record` class to provide a useful service: automatically retrieving `venue` and `speaker` records referenced in an `event` record. This is similar to what the Django ORM does when you access a `models.ForeignKey` field: instead of the key, you get the linked model object. We'll use properties to do that in the next example.

9. By the way, `Bunch` is the name of the class used by Alex Martelli to share this tip in a recipe from 2001 titled "The simple but handy *collector of a bunch of named stuff* class".

Linked Record Retrieval with Properties

The goal of this next version is: given an `event` record retrieved from the shelf, reading its `venue` or `speakers` attributes will not return serial numbers but full-fledged record objects. See the partial interaction in Example 19-10 as an example.

Example 19-10. Extract from the doctests of schedule2.py

```
>>> DbRecord.set_db(db)  ❶
>>> event = DbRecord.fetch('event.33950')  ❷
>>> event  ❸
<Event 'There *Will* Be Bugs'>
>>> event.venue  ❹
<DbRecord serial='venue.1449'>
>>> event.venue.name  ❺
'Portland 251'
>>> for spkr in event.speakers:  ❻
...     print('{0.serial}: {0.name}'.format(spkr))
...
speaker.3471: Anna Martelli Ravenscroft
speaker.5199: Alex Martelli
```

❶ `DbRecord` extends `Record`, adding database support: to operate, `DbRecord` must be given a reference to a database.

❷ The `DbRecord.fetch` class method retrieves records of any type.

❸ Note that `event` is an instance of the `Event` class, which extends `DbRecord`.

❹ Accessing `event.venue` returns a `DbRecord` instance.

❺ Now it's easy to find out the name of an `event.venue`. This automatic dereferencing is the goal of this example.

❻ We can also iterate over the `event.speakers` list, retrieving `DbRecords` representing each speaker.

Figure 19-1 Provides an overview of the classes we'll be studying in this section:

`Record`
: The `__init__` method is the same as in *schedule1.py* (Example 19-9); the `__eq__` method was added to facilitate testing.

`DbRecord`
: Subclass of `Record` adding a `__db` class attribute, `set_db` and `get_db` static methods to set/get that attribute, a `fetch` class method to retrieve records from the database, and a `__repr__` instance method to support debugging and testing.

`Event`
: Subclass of `DbRecord` adding `venue` and `speakers` properties to retrieve linked records, and a specialized `__repr__` method.

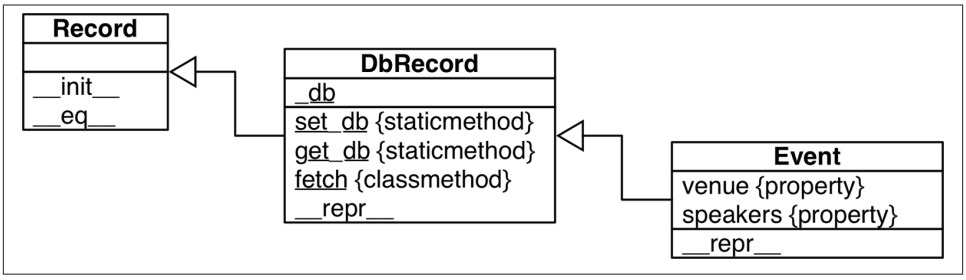

Figure 19-1. UML class diagram for an enhanced Record class and two subclasses: DbRecord and Event.

The `DbRecord.__db` class attribute exists to hold a reference to the opened `shelve.Shelf` database, so it can be used by the `DbRecord.fetch` method and the `Event.venue` and `Event.speakers` properties that depend on it. I coded `__db` as a private class attribute with conventional getter and setter methods because I wanted to protect it from accidental overwriting. I did not use a property to manage `__db` because of a crucial fact: properties are class attributes designed to manage instance attributes.[10]

The code for this section is in the *schedule2.py* module in the *Fluent Python* code repository. Because the module tops 100 lines, I'll present it in parts.[11]

The first statements of *schedule2.py* are shown in Example 19-11.

Example 19-11. schedule2.py: imports, constants, and the enhanced Record class

```
import warnings
import inspect  ❶

import osconfeed

DB_NAME = 'data/schedule2_db'  ❷
CONFERENCE = 'conference.115'


class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __eq__(self, other):  ❸
```

10. The StackOverflow topic "Class-level read only properties in Python" has solutions to read-only attributes in classes, including one by Alex Martelli. The solutions require metaclasses, so you may want to read Chapter 21 before studying them.

11. The full listing for *schedule2.py* is in Example A-13, together with *py.test* scripts in "Chapter 19: OSCON Schedule Scripts and Tests" on page 708.

```
        if isinstance(other, Record):
            return self.__dict__ == other.__dict__
        else:
            return NotImplemented
```

❶ `inspect` will be used in the `load_db` function (Example 19-14).

❷ Because we are storing instances of different classes, we create and use a different database file, `'schedule2_db'`, instead of the `'schedule_db'` of Example 19-9.

❸ An `__eq__` method is always handy for testing.

In Python 2, only "new style" classes support properties. To write a new style class in Python 2 you must subclass directly or indirectly from `object`. `Record` in Example 19-11 is the base class of a hierarchy that will use properties, so in Python 2 its declaration would start with:[12]

```
class Record(object):
    # etc...
```

The next classes defined in *schedule2.py* are a custom exception type and `DbRecord`. See Example 19-12.

Example 19-12. schedule2.py: MissingDatabaseError and DbRecord class

```
class MissingDatabaseError(RuntimeError):
    """Raised when a database is required but was not set."""    ❶


class DbRecord(Record):    ❷

    __db = None    ❸

    @staticmethod    ❹
    def set_db(db):
        DbRecord.__db = db    ❺

    @staticmethod    ❻
    def get_db():
        return DbRecord.__db

    @classmethod    ❼
    def fetch(cls, ident):
        db = cls.get_db()
        try:
```

12. Explicitly subclassing from `object` in Python 3 is not wrong, just redundant because all classes are new-style now. This is one example where breaking with the past made the language cleaner. If the same code must run in Python 2 and Python 3, inheriting from `object` should be explicit.

```
            return db[ident]   ❽
        except TypeError:
            if db is None:   ❾
                msg = "database not set; call '{}.set_db(my_db)'"
                raise MissingDatabaseError(msg.format(cls.__name__))
            else:   ❿
                raise

    def __repr__(self):
        if hasattr(self, 'serial'):   ⓫
            cls_name = self.__class__.__name__
            return '<{} serial={!r}>'.format(cls_name, self.serial)
        else:
            return super().__repr__()   ⓬
```

❶ Custom exceptions are usually marker classes, with no body. A docstring explaining the usage of the exception is better than a mere `pass` statement.

❷ `DbRecord` extends `Record`.

❸ The `__db` class attribute will hold a reference to the opened `shelve.Shelf` database.

❹ `set_db` is a `staticmethod` to make it explicit that its effect is always exactly the same, no matter how it's called.

❺ Even if this method is invoked as `Event.set_db(my_db)`, the `__db` attribute will be set in the `DbRecord` class.

❻ `get_db` is also a `staticmethod` because it will always return the object referenced by `DbRecord.__db`, no matter how it's invoked.

❼ `fetch` is a class method so that its behavior is easier to customize in subclasses.

❽ This retrieves the record with the `ident` key from the database.

❾ If we get a `TypeError` and `db` is `None`, raise a custom exception explaining that the database must be set.

❿ Otherwise, re-raise the exception because we don't know how to handle it.

⓫ If the record has a `serial` attribute, use it in the string representation.

⓬ Otherwise, default to the inherited `__repr__`.

Now we get to the meat of the example: the `Event` class, listed in Example 19-13.

Example 19-13. schedule2.py: the Event class

```
class Event(DbRecord):   ❶

    @property
    def venue(self):
        key = 'venue.{}'.format(self.venue_serial)
```

```
        return self.__class__.fetch(key)    ❷

    @property
    def speakers(self):
        if not hasattr(self, '_speaker_objs'):    ❸
            spkr_serials = self.__dict__['speakers']    ❹
            fetch = self.__class__.fetch    ❺
            self._speaker_objs = [fetch('speaker.{}'.format(key))
                                  for key in spkr_serials]    ❻
        return self._speaker_objs    ❼

    def __repr__(self):
        if hasattr(self, 'name'):    ❽
            cls_name = self.__class__.__name__
            return '<{} {!r}>'.format(cls_name, self.name)
        else:
            return super().__repr__()    ❾
```

❶ `Event` extends `DbRecord`.

❷ The `venue` property builds a `key` from the `venue_serial` attribute, and passes it to the `fetch` class method, inherited from `DbRecord` (see explanation after this example).

❸ The `speakers` property checks if the record has a `_speaker_objs` attribute.

❹ If it doesn't, the `'speakers'` attribute is retrieved directly from the instance `__dict__` to avoid an infinite recursion, because the public name of this property is also `speakers`.

❺ Get a reference to the `fetch` class method (the reason for this will be explained shortly).

❻ `self._speaker_objs` is loaded with a list of `speaker` records, using `fetch`.

❼ That list is returned.

❽ If the record has a `name` attribute, use it in the string representation.

❾ Otherwise, default to the inherited `__repr__`.

In the `venue` property of Example 19-13, the last line returns `self.__class__.fetch(key)`. Why not write that simply as `self.fetch(key)`? The simpler formula works with the specific dataset of the OSCON feed because there is no event record with a `'fetch'` key. If even a single event record had a key named `'fetch'`, then within that specific `Event` instance, the reference `self.fetch` would retrieve the value of that field, instead of the `fetch` class method that `Event` inherits from `DbRecord`. This is a subtle bug, and it could easily sneak through testing and blow up only in production when the venue or speaker records linked to that specific `Event` record are retrieved.

When creating instance attribute names from data, there is always the risk of bugs due to shadowing of class attributes (such as methods) or data loss through accidental overwriting of existing instance attributes. This caveat is probably the main reason why, by default, Python dicts are not like JavaScript objects in the first place.

If the `Record` class behaved more like a mapping, implementing a dynamic `__getitem__` instead of a dynamic `__getattr__`, there would be no risk of bugs from overwriting or shadowing. A custom mapping is probably the Pythonic way to implement `Record`. But if I took that road, we'd not be reflecting on the tricks and traps of dynamic attribute programming.

The final piece of this example is the revised `load_db` function in Example 19-14.

Example 19-14. schedule2.py: the load_db function

```
def load_db(db):
    raw_data = osconfeed.load()
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():
        record_type = collection[:-1]   ❶
        cls_name = record_type.capitalize()   ❷
        cls = globals().get(cls_name, DbRecord)   ❸
        if inspect.isclass(cls) and issubclass(cls, DbRecord):   ❹
            factory = cls   ❺
        else:
            factory = DbRecord   ❻
        for record in rec_list:   ❼
            key = '{}.{}'.format(record_type, record['serial'])
            record['serial'] = key
            db[key] = factory(**record)   ❽
```

❶ So far, no changes from the `load_db` in *schedule1.py* (Example 19-9).

❷ Capitalize the `record_type` to get a potential class name (e.g., `'event'` becomes `'Event'`).

❸ Get an object by that name from the module global scope; get `DbRecord` if there's no such object.

❹ If the object just retrieved is a class, and is a subclass of `DbRecord`…

❺ …bind the `factory` name to it. This means `factory` may be any subclass of `DbRecord`, depending on the `record_type`.

❻ Otherwise, bind the `factory` name to `DbRecord`.

❼ The `for` loop that creates the `key` and saves the records is the same as before, except that…

❽ …the object stored in the database is constructed by `factory`, which may be `DbRecord` or a subclass selected according to the `record_type`.

Note that the only `record_type` that has a custom class is `Event`, but if classes named `Speaker` or `Venue` are coded, `load_db` will automatically use those classes when building and saving records, instead of the default `DbRecord` class.

So far, the examples in this chapter were designed to show a variety of techniques for implementing dynamic attributes using basic tools such as `__getattr__`, `hasattr`, `getattr`, `@property`, and `__dict__`.

Properties are frequently used to enforce business rules by changing a public attribute into an attribute managed by a getter and setter without affecting client code, as the next section shows.

Using a Property for Attribute Validation

So far, we have only seen the `@property` decorator used to implement read-only properties. In this section, we will create a read/write property.

LineItem Take #1: Class for an Item in an Order

Imagine an app for a store that sells organic food in bulk, where customers can order nuts, dried fruit, or cereals by weight. In that system, each order would hold a sequence of line items, and each line item could be represented by a class as in Example 19-15.

Example 19-15. bulkfood_v1.py: the simplest LineItem class

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

That's nice and simple. Perhaps too simple. Example 19-16 shows a problem.

Example 19-16. A negative weight results in a negative subtotal

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> raisins.subtotal()
69.5
>>> raisins.weight = -20  # garbage in...
>>> raisins.subtotal()    # garbage out...
-139.0
```

This is a toy example, but not as fanciful as you may think. Here is a true story from the early days of Amazon.com:

> We found that customers could order a negative quantity of books! And we would credit their credit card with the price and, I assume, wait around for them to ship the books.[13]
>
> — Jeff Bezos
> *Founder and CEO of Amazon.com*

How do we fix this? We could change the interface of `LineItem` to use a getter and a setter for the `weight` attribute. That would be the Java way, and it's not wrong.

On the other hand, it's natural to be able set the `weight` of an item by just assigning to it; and perhaps the system is in production with other parts already accessing `item.weight` directly. In this case, the Python way would be to replace the data attribute with a property.

LineItem Take #2: A Validating Property

Implementing a property will allow us to use a getter and a setter, but the interface of `LineItem` will not change (i.e., setting the `weight` of a `LineItem` will still be written as `raisins.weight = 12`).

Example 19-17 lists the code for a read/write `weight` property.

Example 19-17. bulkfood_v2.py: a LineItem with a weight property

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight   ❶
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property   ❷
    def weight(self):   ❸
        return self.__weight   ❹

    @weight.setter   ❺
    def weight(self, value):
        if value > 0:
            self.__weight = value   ❻
        else:
            raise ValueError('value must be > 0')   ❼
```

13. Direct quote by Jeff Bezos in the *Wall Street Journal* story "Birth of a Salesman" (October 15, 2011).

❶ Here the property setter is already in use, making sure that no instances with negative `weight` can be created.

❷ `@property` decorates the getter method.

❸ The methods that implement a property all have the name of the public attribute: `weight`.

❹ The actual value is stored in a private attribute `__weight`.

❺ The decorated getter has a `.setter` attribute, which is also a decorator; this ties the getter and setter together.

❻ If the value is greater than zero, we set the private `__weight`.

❼ Otherwise, `ValueError` is raised.

Note how a `LineItem` with an invalid weight cannot be created now:

```
>>> walnuts = LineItem('walnuts', 0, 10.00)
Traceback (most recent call last):
    ...
ValueError: value must be > 0
```

Now we have protected `weight` from users providing negative values. Although buyers usually can't set the price of an item, a clerical error or a bug may create a `LineItem` with a negative `price`. To prevent that, we could also turn `price` into a property, but this would entail some repetition in our code.

Remember the Paul Graham quote from Chapter 14: "When I see patterns in my programs, I consider it a sign of trouble." The cure for repetition is abstraction. There are two ways to abstract away property definitions: using a property factory or a descriptor class. The descriptor class approach is more flexible, and we'll devote Chapter 20 to a full discussion of it. Properties are in fact implemented as descriptor classes themselves. But here we will continue our exploration of properties by implementing a property factory as a function.

But before we can implement a property factory, we need to have a deeper understanding of properties.

A Proper Look at Properties

Although often used as a decorator, the `property` built-in is actually a class. In Python, functions and classes are often interchangeable, because both are callable and there is no `new` operator for object instantiation, so invoking a constructor is no different than invoking a factory function. And both can be used as decorators, as long as they return a new callable that is a suitable replacement of the decorated function.

This is the full signature of the `property` constructor:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

All arguments are optional, and if a function is not provided for one of them, the corresponding operation is not allowed by the resulting property object.

The `property` type was added in Python 2.2, but the `@` decorator syntax appeared only in Python 2.4, so for a few years, properties were defined by passing the accessor functions as the first two arguments.

The "classic" syntax for defining properties without decorators is illustrated in Example 19-18.

Example 19-18. bulkfood_v2b.py: same as Example 19-17 but without using decorators

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    def get_weight(self):   ❶
        return self.__weight

    def set_weight(self, value):   ❷
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')

    weight = property(get_weight, set_weight)   ❸
```

❶ A plain getter.

❷ A plain setter.

❸ Build the `property` and assign it to a public class attribute.

The classic form is better than the decorator syntax in some situations; the code of the property factory we'll discuss shortly is one example. On the other hand, in a class body with many methods, the decorators make it explicit which are the getters and setters, without depending on the convention of using `get` and `set` prefixes in their names.

The presence of a property in a class affects how attributes in instances of that class can be found in a way that may be surprising at first. The next section explains.

Properties Override Instance Attributes

Properties are always class attributes, but they actually manage attribute access in the instances of the class.

In "Overriding Class Attributes" on page 267 we saw that when an instance and its class both have a data attribute by the same name, the instance attribute overrides, or shadows, the class attribute—at least when read through that instance. Example 19-19 illustrates this point.

Example 19-19. Instance attribute shadows class data attribute

```
>>> class Class:  # ❶
...     data = 'the class data attr'
...     @property
...     def prop(self):
...         return 'the prop value'
...
>>> obj = Class()
>>> vars(obj)  # ❷
{}
>>> obj.data  # ❸
'the class data attr'
>>> obj.data = 'bar' # ❹
>>> vars(obj)  # ❺
{'data': 'bar'}
>>> obj.data  # ❻
'bar'
>>> Class.data  # ❼
'the class data attr'
```

❶ Define `Class` with two class attributes: the `data` data attribute and the `prop` property.

❷ `vars` returns the `__dict__` of `obj`, showing it has no instance attributes.

❸ Reading from `obj.data` retrieves the value of `Class.data`.

❹ Writing to `obj.data` creates an instance attribute.

❺ Inspect the instance to see the instance attribute.

❻ Now reading from `obj.data` retrieves the value of the instance attribute. When read from the `obj` instance, the instance `data` shadows the class `data`.

❼ The `Class.data` attribute is intact.

Now, let's try to override the `prop` attribute on the `obj` instance. Resuming the previous console session, we have Example 19-20.

Example 19-20. Instance attribute does not shadow class property (continued from Example 19-19)

```
>>> Class.prop  # ❶
<property object at 0x1072b7408>
>>> obj.prop  # ❷
'the prop value'
>>> obj.prop = 'foo'  # ❸
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
>>> obj.__dict__['prop'] = 'foo'  # ❹
>>> vars(obj)  # ❺
{'prop': 'foo', 'attr': 'bar'}
>>> obj.prop  # ❻
'the prop value'
>>> Class.prop = 'baz'  # ❼
>>> obj.prop  # ❽
'foo'
```

❶ Reading `prop` directly from `Class` retrieves the property object itself, without running its getter method.

❷ Reading `obj.prop` executes the property getter.

❸ Trying to set an instance `prop` attribute fails.

❹ Putting `'prop'` directly in the `obj.__dict__` works.

❺ We can see that `obj` now has two instance attributes: `attr` and `prop`.

❻ However, reading `obj.prop` still runs the property getter. The property is not shadowed by an instance attribute.

❼ Overwriting `Class.prop` destroys the property object.

❽ Now `obj.prop` retrieves the instance attribute. `Class.prop` is not a property anymore, so it no longer overrides `obj.prop`.

As a final demonstration, we'll add a new property to `Class`, and see it overriding an instance attribute. Example 19-21 picks up where Example 19-20 left off.

Example 19-21. New class property shadows existing instance attribute (continued from Example 19-20)

```
>>> obj.data  # ❶
'bar'
>>> Class.data  # ❷
'the class data attr'
>>> Class.data = property(lambda self: 'the "data" prop value')  # ❸
>>> obj.data  # ❹
'the "data" prop value'
>>> del Class.data  # ❺
```

```
>>> obj.data  # ❻
'bar'
```

❶ `obj.data` retrieves the instance `data` attribute.

❷ `Class.data` retrieves the class `data` attribute.

❸ Overwrite `Class.data` with a new property.

❹ `obj.data` is now shadowed by the `Class.data` property.

❺ Delete the property.

❻ `obj.data` now reads the instance `data` attribute again.

The main point of this section is that an expression like `obj.attr` does not search for `attr` starting with `obj`. The search actually starts at `obj.__class__`, and only if there is no property named `attr` in the class, Python looks in the `obj` instance itself. This rule applies not only to properties but to a whole category of descriptors, the *overriding descriptors*. Further treatment of descriptors must wait for Chapter 20, where we'll see that properties are in fact overriding descriptors.

Now back to properties. Every Python code unit—modules, functions, classes, methods—can have a docstring. The next topic is how to attach documentation to properties.

Property Documentation

When tools such as the console `help()` function or IDEs need to display the documentation of a property, they extract the information from the `__doc__` attribute of the property.

If used with the classic call syntax, `property` can get the documentation string as the `doc` argument:

```
weight = property(get_weight, set_weight, doc='weight in kilograms')
```

When `property` is deployed as a decorator, the docstring of the getter method—the one with the `@property` decorator itself—is used as the documentation of the property as a whole. Figure 19-2 shows the help screens generated from the code in Example 19-22.

Figure 19-2. Screenshots of the Python console when issuing the commands help(Foo.bar) and help(Foo). Source code in Example 19-22.

Example 19-22. Documentation for a property

```
class Foo:

    @property
    def bar(self):
        '''The bar attribute'''
        return self.__dict__['bar']

    @bar.setter
    def bar(self, value):
        self.__dict__['bar'] = value
```

Now that we have these property essentials covered, let's go back to the issue of protecting both the `weight` and `price` attributes of `LineItem` so they only accept values greater than zero—but without implementing two nearly identical pairs of getters/setters by hand.

Coding a Property Factory

We'll create a `quantity` property factory—so named because the managed attributes represent quantities that can't be negative or zero in the application. Example 19-23 shows the clean look of the `LineItem` class using two instances of `quantity` properties: one for managing the `weight` attribute, the other for `price`.

Example 19-23. bulkfood_v2prop.py: the quantity property factory in use

```
class LineItem:
    weight = quantity('weight')   ❶
    price = quantity('price')     ❷
```

```
    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight    ❸
        self.price = price

    def subtotal(self):
        return self.weight * self.price    ❹
```

❶ Use the factory to define the first custom property, `weight`, as a class attribute.

❷ This second call builds another custom property, `price`.

❸ Here the property is already active, making sure a negative or `0` `weight` is rejected.

❹ The properties are also in use here, retrieving the values stored in the instance.

Recall that properties are class attributes. When building each `quantity` property, we need to pass the name of the `LineItem` attribute that will be managed by that specific property. Having to type the word `weight` twice in this line is unfortunate:

```
    weight = quantity('weight')
```

But avoiding that repetition is complicated because the property has no way of knowing which class attribute name will be bound to it. Remember: the right side of an assignment is evaluated first, so when `quantity()` is invoked, the `price` class attribute doesn't even exist.

Improving the `quantity` property so that the user doesn't need to retype the attribute name is a nontrivial metaprogramming problem. We'll see a workaround in Chapter 20, but real solutions will have to wait until Chapter 21, because they require either a class decorator or a metaclass.

Example 19-24 lists the implementation of the `quantity` property factory.[14]

Example 19-24. bulkfood_v2prop.py: the quantity property factory

```
def quantity(storage_name):    ❶

    def qty_getter(instance):    ❷
        return instance.__dict__[storage_name]    ❸

    def qty_setter(instance, value):    ❹
        if value > 0:
            instance.__dict__[storage_name] = value    ❺
```

14. This code is adapted from "Recipe 9.21. Avoiding Repetitive Property Methods" from *Python Cookbook, 3E* by David Beazley and Brian K. Jones (O'Reilly).

```
            else:
                raise ValueError('value must be > 0')

    return property(qty_getter, qty_setter)   ❻
```

❶ The `storage_name` argument determines where the data for each property is stored; for the `weight`, the storage name will be `'weight'`.

❷ The first argument of the `qty_getter` could be named `self`, but that would be strange because this is not a class body; `instance` refers to the `LineItem` instance where the attribute will be stored.

❸ `qty_getter` references `storage_name`, so it will be preserved in the closure of this function; the value is retrieved directly from the `instance.__dict__` to bypass the property and avoid an infinite recursion.

❹ `qty_setter` is defined, also taking `instance` as first argument.

❺ The `value` is stored directly in the `instance.__dict__`, again bypassing the property.

❻ Build a custom property object and return it.

The bits of Example 19-24 that deserve careful study revolve around the `storage_name` variable. When you code each property in the traditional way, the name of the attribute where you will store a value is hardcoded in the getter and setter methods. But here, the `qty_getter` and `qty_setter` functions are generic, and they depend on the `storage_name` variable to know where to get/set the managed attribute in the instance `__dict__`. Each time the `quantity` factory is called to build a property, the `storage_name` must be set to a unique value.

The functions `qty_getter` and `qty_setter` will be wrapped by the `property` object created in the last line of the factory function. Later when called to perform their duties, these functions will read the `storage_name` from their closures, to determine where to retrieve/store the managed attribute values.

In Example 19-25, I create and inspect a `LineItem` instance, exposing the storage attributes.

Example 19-25. bulkfood_v2prop.py: the quantity property factory

```
>>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
>>> nutmeg.weight, nutmeg.price   ❶
(8, 13.95)
>>> sorted(vars(nutmeg).items())   ❷
[('description', 'Moluccan nutmeg'), ('price', 13.95), ('weight', 8)]
```

❶ Reading the `weight` and `price` through the properties shadowing the namesake instance attributes.

❷ Using `vars` to inspect the `nutmeg` instance: here we see the actual instance attributes used to store the values.

Note how the properties built by our factory leverage the behavior described in "Properties Override Instance Attributes" on page 608: the `weight` property overrides the `weight` instance attribute so that every reference to `self.weight` or `nutmeg.weight` is handled by the property functions, and the only way to bypass the property logic is to access the instance `__dict__` directly.

The code in Example 19-25 may be a bit tricky, but it's concise: it's identical in length to the decorated getter/setter pair defining just the `weight` property in Example 19-17. The `LineItem` definition in Example 19-23 looks much better without the noise of the getter/setters.

In a real system, that same kind of validation may appear in many fields, across several classes, and the `quantity` factory would be placed in a utility module to be used over and over again. Eventually that simple factory could be refactored into a more extensible descriptor class, with specialized subclasses performing different validations. We'll do that in Chapter 20.

Now let us wrap up the discussion of properties with the issue of attribute deletion.

Handling Attribute Deletion

Recall from the Python tutorial that object attributes can be deleted using the `del` statement:

```
del my_object.an_attribute
```

In practice, deleting attributes is not something we do every day in Python, and the requirement to handle it with a property is even more unusual. But it is supported, and I can think of a silly example to demonstrate it.

In a property definition, the `@my_propety.deleter` decorator is used to wrap the method in charge of deleting the attribute managed by the property. As promised, Example 19-26 is a silly example showing how to code a property deleter.

Example 19-26. blackknight.py: inspired by the Black Knight character of "Monty Python and the Holy Grail"

```
class BlackKnight:

    def __init__(self):
        self.members = ['an arm', 'another arm',
                        'a leg', 'another leg']
        self.phrases = ["'Tis but a scratch.",
                        "It's just a flesh wound.",
                        "I'm invincible!",
```

```
                    "All right, we'll call it a draw."]

    @property
    def member(self):
        print('next member is:')
        return self.members[0]

    @member.deleter
    def member(self):
        text = 'BLACK KNIGHT (loses {})\n-- {}'
        print(text.format(self.members.pop(0), self.phrases.pop(0)))
```

The doctests in *blackknight.py* are in Example 19-27.

Example 19-27. blackknight.py: doctests for Example 19-26 (the Black Knight never concedes defeat)

```
    >>> knight = BlackKnight()
    >>> knight.member
    next member is:
    'an arm'
    >>> del knight.member
    BLACK KNIGHT (loses an arm)
    -- 'Tis but a scratch.
    >>> del knight.member
    BLACK KNIGHT (loses another arm)
    -- It's just a flesh wound.
    >>> del knight.member
    BLACK KNIGHT (loses a leg)
    -- I'm invincible!
    >>> del knight.member
    BLACK KNIGHT (loses another leg)
    -- All right, we'll call it a draw.
```

Using the classic call syntax instead of decorators, the `fdel` argument is used to set the deleter function. For example, the `member` property would be coded like this in the body of the `BlackKnight` class:

```
    member = property(member_getter, fdel=member_deleter)
```

If you are not using a property, attribute deletion can also be handled by implementing the lower-level `__delattr__` special method, presented in "Special Methods for Attribute Handling" on page 617. Coding a silly class with `__delattr__` is left as an exercise to the procrastinating reader.

Properties are a powerful feature, but sometimes simpler or lower-level alternatives are preferable. In the final section of this chapter, we'll review some the core APIs that Python offers for dynamic attribute programming.

Essential Attributes and Functions for Attribute Handling

Throughout this chapter, and even before in the book, we've used some of the built-in functions and special methods Python provides for dealing with dynamic attributes. This section gives an overview of them in one place, because their documentation is scattered in the official docs.

Special Attributes that Affect Attribute Handling

The behavior of many of the functions and special methods listed in the following sections depend on three special attributes:

`__class__`
: A reference to the object's class (i.e., `obj.__class__` is the same as `type(obj)`). Python looks for special methods such as `__getattr__` only in an object's class, and not in the instances themselves.

`__dict__`
: A mapping that stores the writable attributes of an object or class. An object that has a `__dict__` can have arbitrary new attributes set at any time. If a class has a `__slots__` attribute, then its instances may not have a `__dict__`. See `__slots__` (next).

`__slots__`
: An attribute that may be defined in a class to limit the attributes its instances can have. `__slots__` is a `tuple` of strings naming the allowed attributes.[15] If the `'__dict__'` name is not in `__slots__`, then the instances of that class will not have a `__dict__` of their own, and only the named attributes will be allowed in them.

Built-In Functions for Attribute Handling

These five built-in functions perform object attribute reading, writing, and introspection:

`dir([object])`
: Lists most attributes of the object. The official docs say `dir` is intended for interactive use so it does not provide a comprehensive list of attributes, but an "interesting" set of names. `dir` can inspect objects implemented with or without a `__dict__`. The `__dict__` attribute itself is not listed by `dir`, but the `__dict__` keys are listed. Several special attributes of classes, such as `__mro__`, `__bases__`, and

15. Alex Martelli points out that, although `__slots__` can be coded as a `list`, it's better to be explicit and always use a `tuple`, because changing the list in the `__slots__` after the class body is processed has no effect, so it would be misleading to use a mutable sequence there.

`__name__` are not listed by `dir` either. If the optional `object` argument is not given, `dir` lists the names in the current scope.

`getattr(object, name[, default])`
: Gets the attribute identified by the `name` string from the `object`. This may fetch an attribute from the object's class or from a superclass. If no such attribute exists, `getattr` raises `AttributeError` or returns the `default` value, if given.

`hasattr(object, name)`
: Returns `True` if the named attribute exists in the `object`, or can be somehow fetched through it (by inheritance, for example). The documentation explains: "This is implemented by calling getattr(object, name) and seeing whether it raises an AttributeError or not."

`setattr(object, name, value)`
: Assigns the `value` to the named attribute of `object`, if the `object` allows it. This may create a new attribute or overwrite an existing one.

`vars([object])`
: Returns the `__dict__` of `object`; `vars` can't deal with instances of classes that define `__slots__` and don't have a `__dict__` (contrast with `dir`, which handles such instances). Without an argument, `vars()` does the same as `locals()`: returns a `dict` representing the local scope.

Special Methods for Attribute Handling

When implemented in a user-defined class, the special methods listed here handle attribute retrieval, setting, deletion, and listing.

Attribute access using either dot notation or the built-in functions `getattr`, `hasattr`, and `setattr` trigger the appropriate special methods listed here. Reading and writing attributes directly in the instance `__dict__` does not trigger these special methods—and that's the usual way to bypass them if needed.

"Section 3.3.9. Special method lookup" of the "Data model" chapter warns:

> For custom classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary.

In other words, assume that the special methods will be retrieved on the class itself, even when the target of the action is an instance. For this reason, special methods are not shadowed by instance attributes with the same name.

In the following examples, assume there is a class named `Class`, `obj` is an instance of `Class`, and `attr` is an attribute of `obj`.

For every one of these special methods, it doesn't matter if the attribute access is done using dot notation or one of the built-in functions listed in "Built-In Functions for Attribute Handling" on page 616. For example, both `obj.attr` and `getattr(obj, 'attr', 42)` trigger `Class.__getattribute__(obj, 'attr')`.

`__delattr__(self, name)`
: Always called when there is an attempt to delete an attribute using the `del` statement; e.g., `del obj.attr` triggers `Class.__delattr__(obj, 'attr')`.

`__dir__(self)`
: Called when `dir` is invoked on the object, to provide a listing of attributes; e.g., `dir(obj)` triggers `Class.__dir__(obj)`.

`__getattr__(self, name)`
: Called only when an attempt to retrieve the named attribute fails, after the `obj`, `Class`, and its superclasses are searched. The expressions `obj.no_such_attr`, `getattr(obj, 'no_such_attr')`, and `hasattr(obj, 'no_such_attr')` may trigger `Class.__getattr__(obj, 'no_such_attr')`, but only if an attribute by that name cannot be found in `obj` or in `Class` and its superclasses.

`__getattribute__(self, name)`
: Always called when there is an attempt to retrieve the named attribute, except when the attribute sought is a special attribute or method. Dot notation and the `getattr` and `hasattr` built-ins trigger this method. `__getattr__` is only invoked after `__getattribute__`, and only when `__getattribute__` raises `AttributeError`. To retrieve attributes of the instance `obj` without triggering an infinite recursion, implementations of `__getattribute__` should use `super().__getattribute__(obj, name)`.

`__setattr__(self, name, value)`
: Always called when there is an attempt to set the named attribute. Dot notation and the `setattr` built-in trigger this method; e.g., both `obj.attr = 42` and `setattr(obj, 'attr', 42)` trigger `Class.__setattr__(obj, 'attr', 42)`.

In practice, because they are unconditionally called and affect practically every attribute access, the `__getattribute__` and `__setattr__` special methods are harder to use correctly than `__getattr__`—which only handles nonexisting attribute names. Using properties or descriptors is less error prone than defining these special methods.

This concludes our dive into properties, special methods, and other techniques for coding dynamic attributes.

Chapter Summary

We started our coverage of dynamic attributes by showing practical examples of simple classes to make it easier to deal with a JSON data feed. The first example was the `FrozenJSON` class that converted nested dicts and lists into nested `FrozenJSON` instances and lists of them. The `FrozenJSON` code demonstrated the use of the `__getattr__` special method to convert data structures on the fly, whenever their attributes were read. The last version of `FrozenJSON` showcased the use of the `__new__` constructor method to transform a class into a flexible factory of objects, not limited to instances of itself.

We then converted the JSON feed to a `shelve.Shelf` database storing serialized instances of a `Record` class. The first rendition of `Record` was a few lines long and introduced the "bunch" idiom: using `self.__dict__.update(**kwargs)` to build arbitrary attributes from keyword arguments passed to `__init__`. The second iteration of this example saw the extension of `Record` with a `DbRecord` class for database integration and an `Event` class implementing automatic retrieval of linked records through properties.

Coverage of properties continued with the `LineItem` class, where a property was deployed to protect a `weight` attribute from negative or zero values that make no business sense. After a deeper look at property syntax and semantics, we created a property factory to enforce the same validation on `weight` and `price`, without coding multiple getters and setters. The property factory leveraged subtle concepts—such as closures and the instance attribute overriding by properties—to provide an elegant generic solution using the same number of lines as a single handcoded property definition.

Finally, we had a brief look at handling attribute deletion with properties, followed by an overview of the key special attributes, built-in functions, and special methods that support attribute metaprogramming in the core Python language.

Further Reading

The official documentation for the attribute handling and introspection built-in functions is Chapter 2, "Built-in Functions" of *The Python Standard Library*. The related special methods and the `__slots__` special attribute are documented in The Python Language Reference in "3.3.2. Customizing attribute access". The semantics of how special methods are invoked bypassing instances is explained in "3.3.9. Special method lookup". In Chapter 4, "Built-in Types," of the Python Standard Library, "4.13. Special Attributes" covers `__class__` and `__dict__` attributes.

Python Cookbook, 3E by David Beazley and Brian K. Jones (O'Reilly) has several recipes covering the topics of this chapter, but I will highlight three that are outstanding: "Recipe 8.8. Extending a Property in a Subclass" addresses the thorny issue of overriding the methods inside a property inherited from a superclass; "Recipe 8.15. Delegating Attribute Access" implements a proxy class showcasing most special methods from "Spe-

cial Methods for Attribute Handling" on page 617 in this book; and the awesome "Recipe 9.21. Avoiding Repetitive Property Methods," which was the basis for the property factory function presented in Example 19-24.

Python in a Nutshell, 2E (O'Reilly), by Alex Martelli, covers only Python 2.5 but the fundamentals still apply to Python 3 and his treatment is rigorous and objective. Martelli devotes only three pages to properties, but that's because the book follows an axiomatic presentation style: the previous 15 pages or so provide a thorough description of the semantics of Python classes from the ground up, including descriptors, which are how properties are actually implemented under the hood. So by the time he gets to properties, he can pack a lot of insights in those three pages—including that which I selected to open this chapter.

Bertrand Meyer, quoted in the *Uniform Access Principle* definition in this chapter opening, wrote the excellent *Object-Oriented Software Construction, 2E* (Prentice-Hall). The book is more than 1,250 pages long, and I confess I did not read it all, but the first six chapters provide one of the best conceptual introductions to OO analysis and design I've seen, Chapter 11 introduces Design by Contract (Meyer invented the method and coined the term), and Chapter 35 offers his assessments of some key OO languages: Simula, Smalltalk, CLOS (the Lisp OO extension), Objective-C, C++, and Java, with brief comments on some others. Meyer is also the inventor of the pseudo-pseudocode: only in the last page of the book he reveals that the "notation" he uses throughout as pseudocode is in fact Eiffel.

Soapbox

Meyer's *Uniform Access Principle* (sometimes called UAP by acronym-lovers) is aesthetically appealing. As a programmer using an API, I shouldn't have to care whether `coconut.price` simply fetches a data attribute or performs a computation. As a consumer and a citizen, I do care: in ecommerce today the value of `coconut.price` often depends on who is asking, so it's certainly not a mere data attribute. In fact, it's common practice that the price is lower if the query comes from outside the store—say, from a price-comparison engine. This effectively punishes loyal customers who like to browse within a particular store. But I digress.

The previous digression does raise a relevant point for programming: although the Uniform Access Principle makes perfect sense in an ideal world, in reality users of an API may need to know whether reading `coconut.price` is potentially too expensive or time consuming. As usual in matters of software engineering, Ward Cunningham's original Wiki hosts insightful arguments about the merits of the Uniform Access Principle.

In object-oriented programming languages, application or violations of the Uniform Access Principle usually revolve around the syntax of reading public data attributes versus invoking getter/setter methods.

Smalltalk and Ruby address this issue in a simple and elegant way: they don't support public data attributes at all. Every instance attribute in these languages is private, so every access to them must be through methods. But their syntax makes this painless: in Ruby, `coconut.price` invokes the `price` getter; in Smalltalk, it's simply `coconut price`.

At the other end of the spectrum, the Java language allows the programmer to choose among four access level modifiers.[16] The general practice does not agree with the syntax established by the Java designers, though. Everybody in Java-land agrees that attributes should be `private`, and you must spell it out every time, because it's not the default. When all attributes are private, all access to them from outside the class must go through accessors. Java IDEs include shortcuts for generating accessor methods automatically. Unfortunately, the IDE is not so helpful when you must read the code six months later. It's up to you to wade through a sea of do-nothing accessors to find those that add value by implementing some business logic.

Alex Martelli speaks for the majority of the Python community when he calls accessors "goofy idioms" and then provides these examples that look very different but do the same thing:[17]

```
someInstance.widgetCounter += 1
# rather than...
someInstance.setWidgetCounter(someInstance.getWidgetCounter() + 1)
```

Sometimes when designing APIs, I've wondered whether every method that does not take an argument (besides `self`), returns a value (other than `None`), and is a pure function (i.e., has no side effects) should be replaced by a read-only property. In this chapter, the `LineItem.subtotal` method (as in Example 19-23) would be a good candidate to become a read-only property. Of course, this excludes methods that are designed to change the object, such as `my_list.clear()`. It would be a terrible idea to turn that into a property, so that merely acessing `my_list.clear` would delete the contents of the list!

In the Pingo.io GPIO library (mentioned in "The __missing__ Method" on page 72), much of the user-level API is based on properties. For example, to read the current value of an analog pin, the user writes `pin.value`, and setting a digital pin mode is written as `pin.mode = OUT`. Behind the scenes, reading an analog pin value or setting a digital pin mode may involve a lot of code, depending on the specific board driver. We decided to use properties in Pingo because we want the API to be comfortable to use even in interactive environments like iPython Notebook, and we feel `pin.mode = OUT` is easier on the eyes and on the fingers than `pin.set_mode(OUT)`.

Although I find the Smalltalk and Ruby solution cleaner, I think the Python approach makes more sense than the Java one. We are allowed to start simple, coding data mem-

16. Including the no-name default that the Java Tutorial calls "package-private."

17. Alex Martelli, *Python in a Nutshell, 2E* (O'Reilly), p. 101.

bers as public attributes, because we know they can always be wrapped by properties (or descriptors, which we'll talk about in the next chapter).

`__new__` Is Better Than `new`

Another example of the Uniform Access Principle (or a variation of it) is the fact that function calls and object instantiation use the same syntax in Python: `my_obj = foo()`, where `foo` may be a class or any other callable.

Other languages influenced by C++ syntax have a `new` operator that makes instantiation look different than a call. Most of the time, the user of an API doesn't care whether `foo` is a function or a class. Until recently, I was under the impression that `property` was a function. In normal usage, it makes no difference.

There are many good reasons for replacing constructors with factories.[18] A popular motive is limiting the number of instances, by returning previously built ones (as in the Singleton pattern). A related use is caching expensive object construction. Also, sometimes it's convenient to return objects of different types depending on the arguments given.

Coding a constructor is simpler; providing a factory adds flexibility at the expense of more code. In languages that have a `new` operator, the designer of an API must decide in advance whether to stick with a simple constructor or invest in factory. If the initial choice is wrong, the correction may be costly—all because `new` is an operator.

Sometimes it may also be convenient to go the other way, and replace a simple function with a class.

In Python, classes and functions are interchangeable in many situations. Not only because there's no `new` operator, but also because there is the `__new__` special method, which can turn a class into a factory producing objects of different kinds (as we saw in "Flexible Object Creation with __new__" on page 592) or returning prebuilt instances instead of creating a new one every time.

This function-class duality would be easier to leverage if PEP 8 — Style Guide for Python Code did not recommend `CamelCase` for class names. On the other hand, dozens of classes in the standard library have lowercase names (e.g., `property`, `str`, `defauldict`, etc.). So maybe the use of lowercase class names is a feature, and not a bug. But however we look at it, the inconsistent capitalization of classes in the Python standard library poses a usability problem.

Although calling a function is not different than calling a class, it's good to know which is which because of another thing we can do with a class: subclassing. So I personally use `CamelCase` in every class that I code, and I wish all classes in the Python standard

18. The reasons I am about to mention are given in the Dr. Dobbs Journal article titled "Java's new Considered Harmful", by Jonathan Amsterdam and in *"Consider static factory methods instead of constructors"*, which is Item 1 of the award-winning book *Effective Java* (Addison-Wesley) by Joshua Bloch.

library used the same convention. I am looking at you, `collections.OrderedDict` and `collections.defaultdict`.