CHAPTER 22

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.
>
> — Martelli, Ravenscroft, and Holden, "Why properties are important"[1]

Data attributes and methods are collectively known as *attributes* in Python. A method is an attribute that is *callable*. *Dynamic attributes* present the same interface as data attributes—i.e., `obj.attr`—but are computed on demand. This follows Bertrand Meyer's *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]

There are several ways to implement dynamic attributes in Python. This chapter covers the simplest ways: the `@property` decorator and the `__getattr__` special method.

A user-defined class implementing `__getattr__` can implement a variation of dynamic attributes that I call *virtual attributes*: attributes that are not explicitly declared anywhere in the source code of the class, and are not present in the instance `__dict__`, but may be retrieved elsewhere or computed on the fly whenever a user tries to read a nonexistent attribute like `obj.no_such_attr`.

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

1 Alex Martelli, Anna Ravenscroft, and Steve Holden, *Python in a Nutshell,* 3rd ed. (O'Reilly), p. 123.

2 Bertrand Meyer, *Object-Oriented Software Construction*, 2nd ed. (Pearson), p. 57.

What's New in This Chapter

Most of the updates to this chapter were motivated by a discussion of `@functools.cached_property` (introduced in Python 3.8), as well as the combined use of `@property` with `@functools.cache` (new in 3.9). This affected the code for the `Record` and `Event` classes that appear in "Computed Properties" on page 845. I also added a refactoring to leverage the PEP 412—Key-Sharing Dictionary optimization.

To highlight more relevant features while keeping the examples readable, I removed some nonessential code—merging the old `DbRecord` class into `Record`, replacing `shelve.Shelve` with a `dict`, and deleting the logic to download the OSCON dataset—which the examples now read from a local file included in the *Fluent Python* code repository.

Data Wrangling with Dynamic Attributes

In the next few examples, we'll leverage dynamic attributes to work with a JSON dataset published by O'Reilly for the OSCON 2014 conference. Example 22-1 shows four records from that dataset.[3]

Example 22-1. 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",
```

3 OSCON—O'Reilly Open Source Conference—was a casualty of the COVID-19 pandemic. The original 744 KB JSON file I used for these examples is no longer online as of January 10, 2021. You'll find a copy of *osconfeed.json* in the example code repository.

```
        "affiliation": "Sharewave",
        "twitter": "sharewaveteam",
        "bio": "Robert ´r0ml´ Lefkowitz is the CTO at Sharewave, a startup..." }
    ],
    "venues": [
      { "serial": 1462,
        "name": "F151",
        "category": "Conference Venues" }
    ]
  }
}
```

Example 22-1 shows 4 of the 895 records in the JSON file. 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 maps to a list of records. In the full dataset, the `"events"`, `"speakers"`, and `"venues"` lists have dozens or hundreds of records, while `"conferences"` has only that one record shown in Example 22-1. Every record has a `"serial"` field, which is a unique identifier for the record within the list.

I used Python's console to explore the dataset, as shown in Example 22-2.

Example 22-2. Interactive exploration of osconfeed.json

```
>>> import json
>>> with open('data/osconfeed.json') as fp:
...     feed = json.load(fp)  ❶
>>> sorted(feed['Schedule'].keys())  ❷
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed['Schedule'].items()):
...     print(f'{len(value):3} {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 the serial number of that same speaker.

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

Exploring JSON-Like Data with Dynamic Attributes

Example 22-2 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 wrote `FrozenJSON`, which is simpler than most recipes because it supports reading only: it's just for exploring the data. `FrozenJSON` is also recursive, dealing automatically with nested mappings and lists.

Example 22-3 is a demonstration of `FrozenJSON`, and the source code is shown in Example 22-4.

Example 22-3. `FrozenJSON` from Example 22-4 allows reading attributes like `name`, and calling methods like `.keys()` and `.items()`

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

4 Two examples are `AttrDict` and `addict`.

```
<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 407, 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 22-3 exposes a minor issue with my code: trying to read a missing attribute should raise `AttributeError`, and not `KeyError` as shown. When I implemented the error handling to do that, the `__getattr__` method became twice as long, distracting from the most important logic I wanted to show. Given that users would know that a `FrozenJSON` is built from mappings and lists, I think the `KeyError` is not too confusing.

Example 22-4. 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
        try:
            return getattr(self.__data, name)  ❸
        except AttributeError:
            return FrozenJSON.build(self.__data[name])  ❹

    def __dir__(self):  ❺
        return self.__data.keys()

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

❶ Build a `dict` from the `mapping` argument. This ensures we get a mapping or something that can be converted to one. The double-underscore prefix on `__data` makes it a *private attribute*.

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

❸ If `name` matches an attribute of the instance `__data` `dict`, return that. This is how calls like `feed.keys()` are handled: the `keys` method is an attribute of the `__data` `dict`.

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

❺ Implementing `__dir__` suports the `dir()` built-in, which in turns supports auto-completion in the standard Python console as well as IPython, Jupyter Notebook, etc. This simple code will enable recursive auto-completion based on the keys in `self.__data`, because `__getattr__` builds `FrozenJSON` instances on the fly—useful for interactive exploration of the data.

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

❼ If `obj` is a mapping, build a `FrozenJSON` with it. This is an example of *goose typing*—see "Goose Typing" on page 442 if you need a refresher.

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

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

A `FrozenJSON` instance has the `__data` private instance attribute stored under the name `_FrozenJSON__data`, as explained in "Private and 'Protected' Attributes in Python" on page 382. Attempts to retrieve attributes by other names 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 `dict` methods 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.__data`, 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.

Note that `FrozenJSON` does not transform or cache the original dataset. As we traverse the data, `__getattr__` creates `FrozenJSON` instances again and again. That's OK for a dataset of this size, and for a script that will only be used to explore or convert the data.

5 The expression `self.__data[name]` is where a `KeyError` exception may occur. Ideally, 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`.

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` code doesn't handle attribute names that are Python keywords. For example, if you build an object like this:

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

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

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

You can always do this, of course:

```
>>> getattr(student, '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:

```
>>> student.class_
1982
```

This can be achieved by replacing the one-liner `__init__` from Example 22-4 with the version in Example 22-5.

Example 22-5. explore1.py: append an _ 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 a JSON record 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 a valid attribute name is not trivial. One solution would be to implement `__getitem__` to allow attribute access using notation like `x['2be']`. 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. `Frozen.JSON.build` is used by `__getattr__` to return a different type of object depending on the value of the attribute being accessed: 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. In Python, `__init__` gets `self` as the first argument, therefore the object already exists when `__init__` is called by the interpreter. Also, `__init__` cannot return anything. So it's really an initializer, not a constructor.

When a class is called to create an instance, the special method that Python calls on that class to construct an instance is `__new__`. It's a class method, but gets special treatment, so the `@classmethod` decorator is not applied to it. Python takes the instance returned by `__new__` and then passes it as the first argument `self` of `__init__`. We rarely need to code `__new__`, because the implementation inherited from `object` suffices for the vast majority of use cases.

If necessary, the `__new__` method can also return an instance of a different class. When that happens, the interpreter does not call `__init__`. In other words, Python's logic for building an object is similar to this pseudocode:

```
# pseudocode for object construction
def make(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 = make(Foo, 'bar')
```

Example 22-6 shows a variation of `FrozenJSON` where the logic of the former `build` class method was moved to `__new__`.

Example 22-6. explore2.py: using `__new__` instead of `build` to construct new objects that may or may not be instances of `FrozenJSON`

```
from collections import abc
import keyword

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 keyword.iskeyword(key):
                key += '_'
            self.__data[key] = value

    def __getattr__(self, name):
        try:
            return getattr(self.__data, name)
        except AttributeError:
            return FrozenJSON(self.__data[name])  ❹

    def __dir__(self):
        return self.__data.keys()
```

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 superclass. 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` class, which Python handles by calling `FrozenJSON.__new__`.

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`. 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.

The OSCON JSON dataset is structured in a way that is not helpful for interactive exploration. For example, the event at index `40`, titled `'There *Will* Be Bugs'` has two speakers, `3471` and `5199`. Finding the names of the speakers is awkward, because those are serial numbers and the `Schedule.speakers` list is not indexed by them. To get each speaker, we must iterate over that list until we find a record with a matching serial number. Our next task is restructuring the data to prepare for automatic retrieval of linked records.

Computed Properties

We first saw the `@property` decorator in Chapter 11, in the section, "A Hashable Vector2d" on page 374. In Example 11-7, I used two properties in `Vector2d` just to make the `x` and `y` attributes read-only. Here we will see properties that compute values, leading to a discussion of how to cache such values.

The records in the `'events'` list of the OSCON JSON data contain integer serial numbers pointing to records in the `'speakers'` and `'venues'` lists. For example, this is the record for a conference talk (with an elided description):

```
{ "serial": 33950,
  "name": "There *Will* Be Bugs",
  "event_type": "40-minute conference session",
  "time_start": "2014-07-23 14:30:00",
  "time_stop": "2014-07-23 15:10:00",
  "venue_serial": 1449,
  "description": "If you're pushing the envelope of programming...",
  "website_url": "http://oscon.com/oscon2014/public/schedule/detail/33950",
  "speakers": [3471, 5199],
  "categories": ["Python"] }
```

We will implement an `Event` class with `venue` and `speakers` properties to return the linked data automatically—in other words, "dereferencing" the serial number. Given an `Event` instance, Example 22-7 shows the desired behavior.

Example 22-7. Reading `venue` and `speakers` returns `Record` objects

```
>>> event  ❶
<Event 'There *Will* Be Bugs'>
>>> event.venue  ❷
<Record serial=1449>
>>> event.venue.name  ❸
'Portland 251'
>>> for spkr in event.speakers:  ❹
...     print(f'{spkr.serial}: {spkr.name}')
...
3471: Anna Martelli Ravenscroft
5199: Alex Martelli
```

❶ Given an `Event` instance…

❷ …reading `event.venue` returns a `Record` object instead of a serial number.

❸ Now it's easy to get the name of the `venue`.

❹ The `event.speakers` property returns a list of `Record` instances.

As usual, we will build the code step-by-step, starting with the `Record` class and a function to read the JSON data and return a `dict` with `Record` instances.

Step 1: Data-Driven Attribute Creation

Example 22-8 shows the doctest to guide this first step.

Example 22-8. Test-driving schedule_v1.py (from Example 22-9)

```
>>> records = load(JSON_PATH)  ❶
>>> speaker = records['speaker.3471']  ❷
>>> speaker  ❸
<Record serial=3471>
>>> speaker.name, speaker.twitter  ❹
('Anna Martelli Ravenscroft', 'annaraven')
```

❶ `load` a `dict` with the JSON data.

❷ The keys in `records` are strings built from the record type and serial number.

❸ `speaker` is an instance of the `Record` class defined in Example 22-9.

❹ Fields from the original JSON can be retrieved as `Record` instance attributes.

The code for *schedule_v1.py* is in Example 22-9.

Example 22-9. schedule_v1.py: reorganizing the OSCON schedule data

```
import json

JSON_PATH = 'data/osconfeed.json'

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

    def __repr__(self):
        return f'<{self.__class__.__name__} serial={self.serial!r}>'  ❷

def load(path=JSON_PATH):
    records = {}  ❸
    with open(path) as fp:
        raw_data = json.load(fp)  ❹
    for collection, raw_records in raw_data['Schedule'].items():  ❺
        record_type = collection[:-1]  ❻
        for raw_record in raw_records:
            key = f'{record_type}.{raw_record["serial"]}'  ❼
            records[key] = Record(**raw_record)  ❽
    return records
```

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

❷ Use the `serial` field to build the custom `Record` representation shown in Example 22-8.

❸ `load` will ultimately return a `dict` of `Record` instances.

❹ Parse the JSON, returning native Python objects: lists, dicts, strings, numbers, etc.

❺ Iterate over the four top-level lists named `'conferences'`, `'events'`, `'speakers'`, and `'venues'`.

❻ `record_type` is the list name without the last character, so `speakers` becomes `speaker`. In Python ≥ 3.9 we can do this more explicitly with `collection.removesuffix('s')`—see PEP 616—String methods to remove prefixes and suffixes.

❼ Build the key in the format `'speaker.3471'`.

❽ Create a `Record` instance and save it in `records` with the key.

The `Record.__init__` method illustrates an old 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 Memory with __slots__" on page 384. So, updating an instance `__dict__` with a mapping is a quick way to create a bunch of attributes in that instance.[7]

> Depending on the application, the `Record` class may need to deal with keys that are not valid attribute names, as we saw in "The Invalid Attribute Name Problem" on page 842. Dealing with that issue would distract from the key idea of this example, and is not a problem in the dataset we are reading.

The definition of `Record` in Example 22-9 is so simple that you may be wondering why I did not use it before, instead of the more complicated `FrozenJSON`. There are two 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. Second reason: `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 classes similar to `Record`, where each instance has an arbitrary set of attributes built from keyword arguments given to `__init__`: `types.SimpleNamespace`, `argparse.Namespace`, and `multiprocessing.managers.Namespace`. I wrote the simpler `Record` class to highlight the essential idea: `__init__` updating the instance `__dict__`.

After reorganizing the schedule dataset, we can enhance the `Record` class to automatically retrieve `venue` and `speaker` records referenced in an `event` record. We'll use properties to do that in the next examples.

Step 2: Property to Retrieve a Linked Record

The goal of this next version is: given an `event` record, reading its `venue` property will return a `Record`. This is similar to what the Django ORM does when you access a `ForeignKey` field: instead of the key, you get the linked model object.

7 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".

We'll start with the venue property. See the partial interaction in Example 22-10 as an example.

Example 22-10. Extract from the doctests of schedule_v2.py

```
>>> event = Record.fetch('event.33950')  ❶
>>> event  ❷
<Event 'There *Will* Be Bugs'>
>>> event.venue  ❸
<Record serial=1449>
>>> event.venue.name  ❹
'Portland 251'
>>> event.venue_serial  ❺
1449
```

❶ The `Record.fetch` static method gets a `Record` or an `Event` from the dataset.

❷ Note that `event` is an instance of the `Event` class.

❸ Accessing `event.venue` returns a `Record` instance.

❹ Now it's easy to find out the name of an `event.venue`.

❺ The `Event` instance also has a `venue_serial` attribute, from the JSON data.

`Event` is a subclass of `Record` adding a `venue` to retrieve linked records, and a specialized `__repr__` method.

The code for this section is in the *schedule_v2.py* module in the *Fluent Python* code repository. The example has nearly 60 lines, so I'll present it in parts, starting with the enhanced `Record` class.

Example 22-11. schedule_v2.py: `Record` class with a new `fetch` method

```
import inspect  ❶
import json

JSON_PATH = 'data/osconfeed.json'

class Record:

    __index = None  ❷

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

    def __repr__(self):
        return f'<{self.__class__.__name__} serial={self.serial!r}>'
```

```
    @staticmethod  ❸
    def fetch(key):
        if Record.__index is None:  ❹
            Record.__index = load()
        return Record.__index[key]  ❺
```

❶ `inspect` will be used in `load`, listed in Example 22-13.

❷ The `__index` private class attribute will eventually hold a reference to the `dict` returned by `load`.

❸ `fetch` is a `staticmethod` to make it explicit that its effect is not influenced by the instance or class on which it is called.

❹ Populate the `Record.__index`, if needed.

❺ Use it to retrieve the record with the given `key`.

> This is one example where the use of `staticmethod` makes sense. The `fetch` method always acts on the `Record.__index` class attribute, even if invoked from a subclass, like `Event.fetch()`—which we'll soon explore. It would be misleading to code it as a class method because the `cls` first argument would not be used.

Now we get to the use of a property in the `Event` class, listed in Example 22-12.

Example 22-12. schedule_v2.py: the `Event` class

```
class Event(Record):  ❶

    def __repr__(self):
        try:
            return f'<{self.__class__.__name__} {self.name!r}>'  ❷
        except AttributeError:
            return super().__repr__()

    @property
    def venue(self):
        key = f'venue.{self.venue_serial}'
        return self.__class__.fetch(key)  ❸
```

❶ `Event` extends `Record`.

❷ If the instance has a `name` attribute, it is used to produce a custom representation. Otherwise, delegate to the `__repr__` from `Record`.

❸ The `venue` property builds a `key` from the `venue_serial` attribute, and passes it to the `fetch` class method, inherited from `Record` (the reason for using `self.__class__` is explained shortly).

The second line of the `venue` method of Example 22-12 returns `self.__class__.fetch(key)`. Why not simply call `self.fetch(key)`? The simpler form works with the specific OSCON dataset because there is no event record with a `'fetch'` key. But, if an 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 `Record`. This is a subtle bug, and it could easily sneak through testing because it depends on the dataset.

> 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. These problems may explain why 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 studying the tricks and traps of dynamic attribute programming.

The final piece of this example is the revised `load` function in Example 22-13.

Example 22-13. schedule_v2.py: the `load` function

```
def load(path=JSON_PATH):
    records = {}
    with open(path) as fp:
        raw_data = json.load(fp)
    for collection, raw_records in raw_data['Schedule'].items():
        record_type = collection[:-1]  ❶
        cls_name = record_type.capitalize()  ❷
        cls = globals().get(cls_name, Record)  ❸
        if inspect.isclass(cls) and issubclass(cls, Record):  ❹
            factory = cls  ❺
        else:
            factory = Record  ❻
```

```
        for raw_record in raw_records:  ❼
            key = f'{record_type}.{raw_record["serial"]}'
            records[key] = factory(**raw_record)  ❽
    return records
```

❶ So far, no changes from the `load` in *schedule_v1.py* (Example 22-9).

❷ Capitalize the `record_type` to get a possible class name; e.g., `'event'` becomes `'Event'`.

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

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

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

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

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

❽ …the object stored in `records` is constructed by `factory`, which may be `Record` or a subclass like `Event`, 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` will automatically use those classes when building and saving records, instead of the default `Record` class.

We'll now apply the same idea to a new `speakers` property in the `Events` class.

Step 3: Property Overriding an Existing Attribute

The name of the `venue` property in Example 22-12 does not match a field name in records of the `"events"` collection. Its data comes from a `venue_serial` field name. In contrast, each record in the `events` collection has a `speakers` field with a list of serial numbers. We want to expose that information as a `speakers` property in `Event` instances, which returns a list of `Record` instances. This name clash requires some special attention, as Example 22-14 reveals.

Example 22-14. schedule_v3.py: the `speakers` property

```
    @property
    def speakers(self):
```

```
        spkr_serials = self.__dict__['speakers']  ❶
        fetch = self.__class__.fetch
        return [fetch(f'speaker.{key}')
                for key in spkr_serials]  ❷
```

❶ The data we want is in a `speakers` attribute, but we must retrieve it directly from the instance `__dict__` to avoid a recursive call to the `speakers` property.

❷ Return a list of all records with keys corresponding to the numbers in `spkr_serials`.

Inside the `speakers` method, trying to read `self.speakers` will invoke the property itself, quickly raising a `RecursionError`. However, if we read the same data via `self.__dict__['speakers']`, Python's usual algorithm for retrieving attributes is bypassed, the property is not called, and the recursion is avoided. For this reason, reading or writing data directly to an object's `__dict__` is a common Python metaprogramming trick.

> The interpreter evaluates `obj.my_attr` by first looking at the class of `obj`. If the class has a property with the `my_attr` name, that property shadows an instance attribute by the same name. Examples in "Properties Override Instance Attributes" on page 861 will demonstrate this, and Chapter 23 will reveal that a property is implemented as a descriptor—a more powerful and general abstraction.

As I coded the list comprehension in Example 22-14, my programmer's lizard brain thought: "This may be expensive." Not really, because events in the OSCON dataset have few speakers, so coding anything more complicated would be premature optimization. However, caching a property is a common need—and there are caveats. So let's see how to do that in the next examples.

Step 4: Bespoke Property Cache

Caching properties is a common need because there is an expectation that an expression like `event.venue` should be inexpensive.[8] Some form of caching could become necessary if the `Record.fetch` method behind the `Event` properties needed to query a database or a web API.

8 This is actually a downside of Meyer's Uniform Access Principle, which I mentioned in the opening of this chapter. Read the optional "Soapbox" on page 875 if you're interested in this discussion.

In the first edition *Fluent Python*, I coded the custom caching logic for the `speakers` method, as shown in Example 22-15.

Example 22-15. Custom caching logic using `hasattr` disables key-sharing optimization

```
    @property
    def speakers(self):
        if not hasattr(self, '__speaker_objs'):  ❶
            spkr_serials = self.__dict__['speakers']
            fetch = self.__class__.fetch
            self.__speaker_objs = [fetch(f'speaker.{key}')
                    for key in spkr_serials]
        return self.__speaker_objs  ❷
```

❶ If the instance doesn't have an attribute named `__speaker_objs`, fetch the speaker objects and store them there.

❷ Return `self.__speaker_objs`.

The handmade caching in Example 22-15 is straightforward, but creating an attribute after the instance is initialized defeats the PEP 412—Key-Sharing Dictionary optimization, as explained in "Practical Consequences of How dict Works" on page 102. Depending on the size of the dataset, the difference in memory usage may be important.

A similar hand-rolled solution that works well with the key-sharing optimization requires coding an `__init__` for the `Event` class, to create the necessary `__speaker_objs` initialized to `None`, and then checking for that in the `speakers` method. See Example 22-16.

Example 22-16. Storage defined in `__init__` to leverage key-sharing optimization

```
class Event(Record):

    def __init__(self, **kwargs):
        self.__speaker_objs = None
        super().__init__(**kwargs)

# 15 lines omitted...
    @property
    def speakers(self):
        if self.__speaker_objs is None:
            spkr_serials = self.__dict__['speakers']
            fetch = self.__class__.fetch
            self.__speaker_objs = [fetch(f'speaker.{key}')
                    for key in spkr_serials]
        return self.__speaker_objs
```

Examples 22-15 and 22-16 illustrate simple caching techniques that are fairly common in legacy Python codebases. However, in multithreaded programs, handmade caches like those introduce race conditions that may lead to corrupted data. If two threads are reading a property that was not previously cached, the first thread will need to compute the data for the cache attribute (`__speaker_objs` in the examples) and the second thread may read a cached value that is not yet complete.

Fortunately, Python 3.8 introduced the `@functools.cached_property` decorator, which is thread safe. Unfortunately, it comes with a couple of caveats, explained next.

Step 5: Caching Properties with functools

The `functools` module provides three decorators for caching. We saw `@cache` and `@lru_cache` in "Memoization with functools.cache" on page 320 (Chapter 9). Python 3.8 introduced `@cached_property`.

The `functools.cached_property` decorator caches the result of the method in an instance attribute with the same name. For example, in Example 22-17, the value computed by the `venue` method is stored in a `venue` attribute in `self`. After that, when client code tries to read `venue`, the newly created `venue` instance attribute is used instead of the method.

Example 22-17. Simple use of a `@cached_property`

```
    @cached_property
    def venue(self):
        key = f'venue.{self.venue_serial}'
        return self.__class__.fetch(key)
```

In "Step 3: Property Overriding an Existing Attribute" on page 852, we saw that a property shadows an instance attribute by the same name. If that is true, how can `@cached_property` work? If the property overrides the instance attribute, the `venue` attribute will be ignored and the `venue` method will always be called, computing the `key` and running `fetch` every time!

The answer is a bit sad: `cached_property` is a misnomer. The `@cached_property` decorator does not create a full-fledged property, it creates a *nonoverriding descriptor*. A descriptor is an object that manages the access to an attribute in another class. We will dive into descriptors in Chapter 23. The `property` decorator is a high-level API to create an *overriding descriptor*. Chapter 23 will include a through explanation about *overriding* versus *nonoverriding* descriptors.

For now, let us set aside the underlying implementation and focus on the differences between `cached_property` and `property` from a user's point of view. Raymond Hettinger explains them very well in the Python docs:

> The mechanics of `cached_property()` are somewhat different from `property()`. A regular property blocks attribute writes unless a setter is defined. In contrast, a `cached_property` allows writes.
>
> The `cached_property` decorator only runs on lookups and only when an attribute of the same name doesn't exist. When it does run, the `cached_property` writes to the attribute with the same name. Subsequent attribute reads and writes take precedence over the `cached_property` method and it works like a normal attribute.
>
> The cached value can be cleared by deleting the attribute. This allows the `cached_property` method to run again.[9]

Back to our `Event` class: the specific behavior of `@cached_property` makes it unsuitable to decorate `speakers`, because that method relies on an existing attribute also named `speakers`, containing the serial numbers of the event speakers.

> `@cached_property` has some important limitations:
>
> - It cannot be used as a drop-in replacement to `@property` if the decorated method already depends on an instance attribute with the same name.
> - It cannot be used in a class that defines `__slots__`.
> - It defeats the key-sharing optimization of the instance `__dict__`, because it creates an instance attribute after `__init__`.

Despite these limitations, `@cached_property` addresses a common need in a simple way, and it is thread safe. Its Python code is an example of using a *reentrant lock*.

The `@cached_property` documentation recommends an alternative solution that we can use with `speakers`: stacking `@property` and `@cache` decorators, as shown in Example 22-18.

Example 22-18. Stacking `@property` on `@cache`

```
@property  ❶
@cache  ❷
def speakers(self):
    spkr_serials = self.__dict__['speakers']
    fetch = self.__class__.fetch
```

9 Source: @functools.cached_property documentation. I know Raymond Hettinger authored this explanation because he wrote it as a response to an issue I filed: bpo42781—functools.cached_property docs should explain that it is non-overriding. Hettinger is a major contributor to the official Python docs and standard library. He also wrote the excellent "Descriptor HowTo Guide", a key resource for Chapter 23.

```
        return [fetch(f'speaker.{key}')
                for key in spkr_serials]
```

❶ The order is important: `@property` goes on top…

❷ …of `@cache`.

Recall from "Stacked Decorators" on page 322 the meaning of that syntax. The top three lines of Example 22-18 are similar to:

```
    speakers = property(cache(speakers))
```

The `@cache` is applied to `speakers`, returning a new function. That function then is decorated by `@property`, which replaces it with a newly constructed property.

This wraps up our discussion of read-only properties and caching decorators, exploring the OSCON dataset. In the next section, we start a new series of examples creating read/write properties.

Using a Property for Attribute Validation

Besides computing attribute values, properties are also used to enforce business rules by changing a public attribute into an attribute protected by a getter and setter without affecting client code. Let's work through an extended example.

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 an instance of a class, as in Example 22-19.

Example 22-19. 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 22-20 shows a problem.

Example 22-20. 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 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.
>
> — Jeff Bezos, founder and CEO of Amazon.com[10]

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 to 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 22-21 lists the code for a read/write `weight` property.

Example 22-21. 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
```

10 Direct quote by Jeff Bezos in the *Wall Street Journal* story, "Birth of a Salesman" (October 15, 2011). Note that as of 2021, you need a subscription to read the article.

```
    @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
```

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 All the methods that implement a property share 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 17: "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 23 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 from 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 callable.

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 22-22.

Example 22-22. bulkfood_v2b.py: same as Example 22-21, 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 389 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 22-23 illustrates this point.

Example 22-23. Instance attribute shadows the 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` 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 22-24.

Example 22-24. Instance attribute does not shadow the class property (continued from Example 22-23)

```
>>> 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
{'data': 'bar', 'prop': 'foo'}
>>> 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: `data` 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 22-25 picks up where Example 22-24 left off.

Example 22-25. New class property shadows the existing instance attribute (continued from Example 22-24)

```
>>> 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.data` does not start the search for `data` in `obj`. The search actually starts at `obj.__class__`, and only if there is no property named `data` in the class, Python looks in the `obj` instance itself. This applies to *overriding descriptors* in general, of which properties are just one example. Further treatment of descriptors must wait for Chapter 23.

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')
```

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 22-1 shows the help screens generated from the code in Example 22-26.

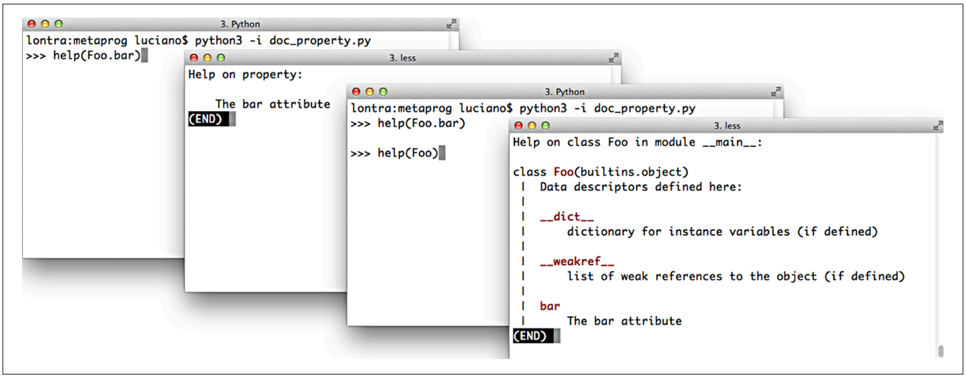

Figure 22-1. Screenshots of the Python console when issuing the commands `help(Foo.bar)` and `help(Foo)`. Source code is in Example 22-26.

Example 22-26. 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 factory to create quantity properties—so named because the managed attributes represent quantities that can't be negative or zero in the application. Example 22-27 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 22-27. 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 righthand side of an assignment is evaluated first, so when quantity() is invoked, the weight 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 solve that problem in Chapter 23.

Example 22-28 lists the implementation of the `quantity` property factory.[11]

Example 22-28. 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
        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.

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

The bits of Example 22-28 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 22-29, I create and inspect a LineItem instance, exposing the storage attributes.

Example 22-29. bulkfood_v2prop.py: exploring properties and storage attributes

```
>>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
>>> nutmeg.weight, nutmeg.price  ❶
(8, 13.95)
>>> nutmeg.__dict__  ❷
{'description': 'Moluccan nutmeg', 'weight': 8, 'price': 13.95}
```

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 861: 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 22-28 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 22-21. The LineItem definition in Example 22-27 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 23.

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

Handling Attribute Deletion

We can use the `del` statement to delete not only variables, but also attributes:

```
>>> class Demo:
...     pass
...
>>> d = Demo()
>>> d.color = 'green'
>>> d.color
'green'
>>> del d.color
>>> d.color
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Demo' object has no attribute 'color'
```

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_property.deleter` decorator wraps the method in charge of deleting the attribute managed by the property. As promised, silly Example 22-30 is inspired by the scene with the Black Knight from *Monty Python and the Holy Grail*.[12]

Example 22-30. blackknight.py

```
class BlackKnight:

    def __init__(self):
        self.phrases = [
            ('an arm', "'Tis but a scratch."),
            ('another arm', "It's just a flesh wound."),
            ('a leg', "I'm invincible!"),
            ('another leg', "All right, we'll call it a draw.")
        ]

    @property
    def member(self):
        print('next member is:')
```

12 The bloody scene is available on Youtube as I review this in October 2021.

```
        return self.phrases[0][0]

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

The doctests in *blackknight.py* are in Example 22-31.

Example 22-31. blackknight.py: doctests for Example 22-30 (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 configures 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 871. 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 of 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 save memory. `__slots__` is a `tuple` of strings naming the allowed attributes.[13] 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 attributes listed in `__slots__` will be allowed in those instances. Recall "Saving Memory with __slots__" on page 384 for more.

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 `__name__`, are not listed by `dir` either. You can customize the output of `dir` by implementing the `__dir__` special method, as we saw in Example 22-4. If the optional `object` argument is not given, `dir` lists the names in the current scope.

13 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.

`getattr(object, name[, default])`
: Gets the attribute identified by the `name` string from the `object`. The main use case is to retrieve attributes (or methods) whose names we don't know beforehand. 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. One great example of using `gettatr` is in the `Cmd.onecmd` method in the `cmd` package of the standard library, where it is used to get and execute a user-defined command.

`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` triggers 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 "3.3.11. 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 870. 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')`. If `attr` is a property, its deleter method is never called if the class implements `__delattr__`.

`__dir__(self)`
: Called when `dir` is invoked on the object, to provide a listing of attributes; e.g., `dir(obj)` triggers `Class.__dir__(obj)`. Also used by tab-completion in all modern Python consoles.

`__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 directly from Python code (the interpreter may bypass this in some cases, for example, to get the `__repr__` 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 dataset. 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 dataset to a `dict` storing 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 added the `Event` class, implementing automatic retrieval of linked records through properties. Computed property values sometimes require caching, and we covered a few ways of doing that.

After realizing that `@functools.cached_property` is not always applicable, we learned about an alternative: combining `@property` on top of `@functools.cache`, in that order.

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 instance attribute overriding by properties—to provide an elegant generic solution using the same number of lines as a single hand-coded 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, 3rd ed., 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 "Special Methods for Attribute Handling" on page 871 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 22-28.

Python in a Nutshell, 3rd ed., by Alex Martelli, Anna Ravenscroft, and Steve Holden (O'Reilly) is rigorous and objective. They devote only three pages to properties, but that's because the book follows an axiomatic presentation style: the preceding 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 Martelli et al., get to properties, they pack a lot of insights in those three pages—including what I selected to open this chapter.

Bertrand Meyer—quoted in the Uniform Access Principle definition in this chapter opening—pioneered the Design by Contract methodology, designed the Eiffel language, and wrote the excellent *Object-Oriented Software Construction*, 2nd ed. (Pearson). The first six chapters provide one of the best conceptual introductions to OO analysis and design I've seen. Chapter 11 presents Design by Contract, and Chapter 35 offers Meyer's assessments of some influential object-oriented languages: Simula, Smalltalk, CLOS (the Common Lisp Object System), Objective-C, C++, and Java, with brief comments on some others. Only in the last page of the book does he reveal that the highly readable "notation" he uses as pseudocode is Eiffel.

Soapbox

Meyer's Uniform Access Principle is aesthetically appealing. As a programmer using an API, I shouldn't have to care whether `product.price` simply fetches a data attribute or performs a computation. As a consumer and a citizen, I do care: in e-commerce today the value of `product.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 `product.price` is potentially too expensive or time-consuming. That's a problem with programming abstractions in general: they make it hard to reason about the runtime cost of evaluating an expression. On the other hand, abstractions let users accomplish more with less code. It's a trade-off. 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 often 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, `product.price` invokes the `price` getter; in Smalltalk, it's simply `product price`.

At the other end of the spectrum, the Java language allows the programmer to choose among four access-level modifiers—including the no-name default that the Java Tutorial calls "package-private."

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:[14]

```
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 22-27) 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 accessing `my_list.clear` would delete the contents of the list!

In the *Pingo* GPIO library, which I coauthored (mentioned in "The __missing__ Method" on page 91), 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 a Jupyter 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 members 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. For years I was under the impression that `property` was a function. In normal usage, it makes no difference.

14 Alex Martelli, *Python in a Nutshell*, 2nd ed. (O'Reilly), p. 101.

There are many good reasons for replacing constructors with factories.[15] 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 a 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 843) 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`, `defaultdict`, 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 from 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 library used the same convention. I am looking at you, `collections.OrderedDict` and `collections.defaultdict`.

15 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*, 3rd ed., by Joshua Bloch (Addison-Wesley).