11. Persistence and Databases - Python in a Nutshell, 3rd Edition

Chapter 11. Persistence and Databases

Python supports several ways of persisting data. One way, *serialization*, views data as a collection of Python objects. These objects can be *serialized* (saved) to a byte stream, and later *deserialized* (loaded and re-created) back from the byte stream. *Object persistence* lives on top of serialization, adding features such as object naming. This chapter covers the Python modules that support serialization and object persistence.

Another way to make data persistent is to store it in a database (DB). One simple category of DBs are just file formats that use *keyed access* to enable selective reading and updating of relevant parts of the data. This chapter covers Python standard library modules that support several variations of such a file format, known as *DBM*.

A *relational DB management system* (RDBMS), such as PostgreSQL or Oracle, offers a more powerful approach to storing, searching, and retrieving persistent data. Relational DBs rely on dialects of *Structured Query Language* (SQL) to create and alter a DB's schema, insert and update data in the DB, and query the DB with search criteria. (This book does not provide reference material on SQL. We recommend *SQL in a Nutshell*, by Kevin Kline [O'Reilly].) Unfortunately, despite the existence of SQL standards, no two RDBMSes implement exactly the same SQL dialect.

The Python standard library does not come with an RDBMS interface. However, many third-party modules let your Python programs access a specific RDBMS. Such modules mostly follow the Python Database API 2.0 standard, also known as the *DBAPI*. This chapter covers the DBAPI standard and mentions a few of the most popular third-party modules that implement it.

A DBAPI module that is particularly handy—because it comes with every standard installation of Python—is sqlite3, which wraps SQLite, "a self-contained, server-less, zero-configuration, transactional SQL DB engine," which is the most widely deployed relational DB engine in the world. We cover `sqlite3` in "SQLite".

Besides relational DBs, and the simpler approaches covered in this chapter, there exist Python-specific object DBs such as ZODB, as well as many NoSQL DBs, each with Python interfaces. We do not cover advanced nonrelational DBs in this book.

Serialization

Python supplies several modules to *serialize* (save) Python objects to various kinds of byte streams and *deserialize* (load and re-create) Python objects back from streams. Serialization is also known as *marshaling*, or as "formatting for *data interchange*."

Serialization approaches span the range from (by now) language-independent JSON to (low-level, Python-version-specific) `marshal`, both limited to elementary data types, through richer but Python-specific `pickle` in addition to rich cross-language formats such as XML, YAML, protocol buffers, and MessagePack.

In this section, we cover JSON and `pickle`. We cover XML in Chapter 23. `marshal` is too low-level to use in applications; should you need to maintain old code using it, refer to the online docs. As for protocol buffers, MessagePack, YAML, and other data-interchange/serialization approaches (each with specific advantages and weaknesses), we cannot cover everything in this book; we recommend studying them via the resources available on the web.

The json Module

The standard library's `json` module supplies four key functions:

dump `dump(value,fileobj,skipkeys=False,ensure_ascii=True, check_circular=True,allow_nan=True,cls=JSONEncoder,indent=None, separators=(', ', ': '),encoding='utf-8',default=None, sort_keys=False,**kw)`

`dump` writes the JSON serialization of object `value` to file-like object `fileobj`, which must be opened for writing in text mode, via calls to `fileobj.write`. In v3, each call to `fileobj.write` passes as argument a text (Unicode) string; in v2, it passes a plain (byte) string obtained with encoding `encoding`. Argument `encoding` is only allowed in v2: in v3, `dump` does not perform any text encoding.

When `skipkeys` is `True` (by default, it's `False`), `dict` keys that are not scalars (i.e., are not of types `bool`, `float`, `int`, `str`, or `None`; in v2, also `long` and `unicode`) are silently skipped, instead of raising an exception. In any case, keys that *are* scalars are turned into strings (e.g., `None` becomes `"null"`): JSON only allows strings as keys in its mappings.

When `ensure_ascii` is `True` (as it is by default), all non-ASCII characters in the output are escaped; when it's `False`, they're output as is.

When `check_circular` is `True` (as it is by default), containers in `value` are checked for circular references, raising a `ValueError` exception if circular references are found; when it's `False`, the check is skipped, and many different exceptions can get raised as a result (even a crash is possible).

When `allow_nan` is `True` (as it is by default), float scalars `nan`, `inf`, and `-inf` are output as their respective JavaScript equivalents, `NaN`, `Infinity`, `-Infinity`; when it's `False`, the presence of such scalars raises a `ValueError`.

You can optionally pass `cls` in order to use a customized subclass of `JSONEncoder` (such advanced customization is rarely needed and we don't cover it in this book); in this case, `**kw` gets passed in the call to `cls` which instantiates it. By default, encoding uses the `JSONEncoder` class directly.

When `indent` is an `int>0`, `dump` "pretty-prints" the output by prepending that many spaces to each array element and object member; when an `int<=0`, `dump` just inserts `\n` characters; when `None`, which is the default, `dump` uses the most compact representation. In v3 only, `indent` can be a `str`—for example, `'\t'`—and in that case `dump` uses that string for indenting.

`separators` must be a pair (a tuple with two items), respectively the strings used to separate items, and keys from values. You can explicitly pass `separators=(',',':')` to ensure `dump` inserts no whitespace.

You can optionally pass `default` in order to transform some otherwise nonserializable objects into serializable ones; `default` is a function, called with a single argument that's a nonserializable object, and must return a serializable object or else raise `ValueError` (by default, the presence of nonserializable objects raises `ValueError`).

When `sort_keys` is `True` (by default, it's `False`), mappings are output in sorted order of their keys; when it's `False`, they're output in whatever is their natural order of iteration (essentially random for `dict`s, but you could use a `collections.OrderedDict` to control iteration order).

dumps `dumps(value,skipkeys=False,ensure_ascii=True,check_circular=True, allow_nan=True,cls=JSONEncoder,indent=None,separators=(' ', ': '), encoding='utf-8',default=None,sort_keys=False,**kw)`

`dumps` returns the string that's the JSON serialization of object `value`—that is, the string that `dump` would write to its file-object argument. All arguments to `dumps` have exactly the same meaning as the arguments to `dump`.

load

JSON serializes just one object per file

JSON is not what is known as a *framed format*: this means it is *not* possible to call `dump` more than once in order to serialize multiple objects into the same file, nor later call `load` more than once to deserialize the objects, as would be possible with `pickle`. So, technically, JSON serializes just one object per file. However, you can make that one object be a `list`, or `dict`, which in turn can contain as many items as you wish.

`load(fileobj,encoding='utf-8',cls=JSONDecoder,object_hook=None, parse_float=float,parse_int=int,parse_constant=None, object_pairs_hook=None,**kw)`

`load` creates and returns the object `v` previously serialized into file-like object `fileobj`, which must be opened for reading in text mode, getting `fileobj`'s contents via a call to `fileobj.read`. In v3, the call to `fileobj.read` must return a text (Unicode) string; in v2, it can alternatively return a plain (byte) string that gets decoded with encoding `encoding`. The argument `encoding` is only allowed in v2: in v3, `load` does not perform any text decoding.

The functions `load` and `dump` are complementary. In other words, a single call to `load(f)` deserializes the same value previously serialized when `f`'s contents were created by a single call to `dump(v,f)` (possibly with some alterations: e.g., all dictionary keys are turned into strings).

You can optionally pass `cls` in order to use a customized subclass of `JSONDecoder` (such advanced customization is rarely needed and we don't cover it in this book); in this case, `**kw` gets passed in the call to `cls`, which instantiates it. By default, decoding uses the `JSONDecoder` class directly.

You can optionally pass `object_hook` or `object_pairs_hook` (if you pass both, `object_hook` is ignored and only `object_pairs_hook` is used), a function that lets you implement custom decoders. When you pass `object_hook` but not `object_pairs_hook`, then, each time an object is decoded into a `dict`, `load` calls `object_hook` with the `dict` as the only argument, and uses `object_hook`'s return value instead of that `dict`. When you pass `object_pairs_hook`, then, each time an object is decoded, `load` calls `object_pairs_hook` with, as the only argument, a list of the pairs of `(key, value)` items of the object, in the order in which they are present in the input, and uses `object_pairs_hook`'s return value; this lets you perform specialized decoding that potentially depends on the order of `(key, value)` pairs in the input.

`parse_float`, `parse_int`, and `parse_constant` are functions called with a single argument that's a `str` representing a `float`, an `int`, or one of the three special constants `'NaN'`, `'Infinity'`, `'-Infinity'`; `load` calls the appropriate function each time it identifies in the input a `str` representing a number, and uses the function's return value. By default, `parse_float` is `float`, `parse_int` is `int`, and `parse_constant` returns one of the three special float scalars `nan`, `inf`, `-inf`, as appropriate. For example, you could pass `parse_float=decimal.Decimal` to ensure that all numbers in the result that would normally be `float`s are instead decimals (covered in "The decimal Module").

loads `loads(s,encoding='utf-8',cls=JSONDecoder,object_hook=None, parse_float=float,parse_int=int,parse_constant=None, object_pairs_hook=None,**kw)`

`loads` creates and returns the object `v` previously serialized into the string `s`. All argument to `loads` have exactly the same meaning as the arguments to `load`.

A JSON example

Say you need to read several text files, whose names are given as your program's arguments, recording where each distinct word appears in the files. What you need to record for each word is a list of `(filename, linenumber)` pairs. The following example uses `json` to encode lists of `(filename, linenumber)` pairs as strings and store them in a DBM-like file (as covered in "DBM Modules"). Since these lists contain tuples, each containing a string and a number, they are within `json`'s abilities to serialize.

```
import collections, fileinput, json, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno
()
    for word in line.split():
        word_pos[word].append(pos)
dbm_out = dbm.open('indexfilem', 'n')
for word in word_pos:
    dbm_out[word] = json.dumps(word_pos[word])
dbm_out.close()
```

(In v3 only, `dbm.open` is a context manager, so we could indent the second `for` loop as the body of a statement `with dbm.open('indexfilem', 'n') as dbm_out:` and omit the `dbm_out.close()`; however, the example, as coded, works in both v2 and v3, except that in v2, to ensure the example works across platforms, you'd import and use the module `anydbm` instead of the package `dbm`, and that also applies to the following example.) We also need `json` to deserialize the data stored in the DBM-like file *indexfilem*, as shown in the following example:

```
import sys, json, dbm, linecache
dbm_in = dbm.open('indexfilem')
for word in sys.argv[1:]:
    if word not in dbm_in:
                                 not found in index
         print('Word {!r}file'                     .format(word
),
               file=sys.stderr)
         continue
    places = json.loads(dbm_in[word])
    for fname, lineno in places:
        print('Word {!r} occurs in line {} of file {}:'.format(
               word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```

The pickle and cPickle Modules

The pickle and, in v2, cPickle modules supply factory functions, named Pickler and Unpickler, to generate objects that wrap file-like objects and supply Python-specific serialization mechanisms. Serializing and deserializing via these modules is also known as *pickling* and *unpickling*.

In v2 only, the difference between the modules is that, in pickle, Pickler and Unpickler are classes, so you can inherit from these classes to create customized serializer objects, overriding methods as needed. In cPickle, on the other hand, Pickler and Unpickler are factory functions that generate instances of non-subclassable types, not classes. Performance is much better with cPickle, but inheritance is not feasible. In the rest of this section, we'll be talking about the module pickle, but everything applies to cPickle too. v3 only supplies pickle, which is quite fast and supplies Pickler and Unpickler as classes.

Serialization shares some of the issues of deep copying, covered in "The copy Module". The pickle module deals with these issues in much the same way as the copy module does. Serialization, like deep copying, implies a recursive walk over a directed graph of references. pickle preserves the graph's shape: when the same object is encountered more than once, the object is serialized only the first time, and other occurrences of the same object serialize references to that single value. pickle also correctly serializes graphs with reference cycles. However, this means that if a mutable object o is serialized more than once to the same Pickler instance p, any changes to o after the first serialization of o to p are not saved.

Don't alter objects while their serialization is in progress

For clarity, correctness, and simplicity, don't alter objects that are being serialized while serialization to a Pickler instance is in progress.

pickle can serialize with a legacy ASCII protocol or with one of a few compact binary ones. In v2, the ASCII protocol 0 is the default, for backward compatibility, but you should normally explicitly request binary protocol 2, the v2-supported protocol that's most parsimonious of time and storage space. In v3, protocols range from 0 to 4, included; the default is 3, which is usually a reasonable choice, but you may explicitly specify protocol 2 (to ensure that your saved pickles can be loaded by v2 programs), or protocol 4, incompatible with earlier versions but with performance advantages for very large objects.

Always pickle with protocol 2 or higher

Always specify at least protocol 2. The size and speed savings can be substantial, and binary format has basically no downside except loss of compatibility of resulting pickles with truly ancient versions of Python.

When you reload objects, pickle transparently recognizes and uses any protocol that the Python version you're currently using supports.

pickle serializes classes and functions by name, not by value. pickle can therefore deserialize a class or function only by importing it from the same module where the class or function was found when pickle serialized it. In particular, pickle can normally serialize and deserialize classes and functions only if they are top-level names for their module (i.e., attributes of their module). For example, consider the following:

```
def adder(augend):
    def inner(addend, augend=augend): return addend+
augend
    return inner
plus5 = adder(5)
```

This code binds a closure to name plus5 (as covered in "Nested functions and nested scopes")—a nested function `inner` plus an appropriate outer scope. Therefore, trying to pickle `plus5` raises a `pickle.PicklingError` exception (in v2; just `AttributeError` in v3): a function can be pickled only when it is top-level, and the function `inner`, whose closure is bound to the name `plus5` in this code, is not top-level but rather nested inside the function `adder`. Similar issues apply to pickling nested functions and nested classes (i.e., classes that are not top-level).

Functions and classes of pickle and cPickle

The `pickle` module (and, in v2 only, the module `cPickle`) exposes the following functions and classes:

dump, dumps

`dump(value,fileobj,protocol=None,bin=None)`
`dumps(value,protocol=None,bin=None)`

`dumps` returns a bytestring representing object `value`. `dump` writes the same string to the file-like object `fileobj`, which must be opened for writing. `dump(v,f)` is like `f.write(dumps(v))`. Do not pass the `bin` parameter, which exists only for compatibility with old versions of Python. The `protocol` parameter can be `0` (the default in v2, for compatibility reasons; ASCII output, slowest and bulkiest), `1` (binary output that's compatible with very old versions of Python), or `2` (fastest and leanest in v2); in v3, it can also be `3` or `4`. We recommend that, in v2, you always pass the value `2`; in v3, pass `2` if you need the result to be compatible with v2, otherwise `3` (the v3 default) or `4`. Unless `protocol` is `0` or (in v2) absent, implying ASCII output, the `fileobj` parameter to `dump` must be open for binary writing.

load, loads

`load(fileobj)`
`loads(s)` in v2; in v3, `loads(s,*,fix_imports=True, encoding="ASCII", errors="strict")`

The functions `load` and `dump` are complementary. In other words, a sequence of calls to `load(f)` deserializes the same values previously serialized when `f`'s contents were created by a sequence of calls to `dump(v,f)`. `load` reads the right number of bytes from file-like object `fileobj` and creates and returns the object `v` represented by those bytes. `load` and `loads` transparently support pickles performed in any binary or ASCII protocol. If data is pickled in any binary format, the file must be open as binary for both `dump` and `load`. `load(f)` is like `Unpickler(f).load()`.

`loads` creates and returns the object `v` represented by byte string `s`, so that for any object `v` of a supported type, `v==loads(dumps(v))`. If `s` is longer than `dumps(v)`, `loads` ignores the extra bytes. Optional v3-only arguments `fix_imports`, `encoding`, and `errors`, are provided for handling v2-generated streams. See pickle.loads v3 docs.

Don't unpickle untrusted data

Unpickling from an untrusted data source is a security risk; an attacker could exploit this vulnerability to execute arbitrary code.

Pickler

`Pickler(fileobj,protocol=None,bin=None)`

Creates and returns an object `p` such that calling `p.dump` is equivalent to calling the function `dump` with the `fileobj`, `protocol`, and `bin` arguments passed to `Pickler`. To serialize many objects to a file, `Pickler` is more convenient and faster than repeated calls to `dump`. You can subclass `pickle.Pickler` to override `Pickler` methods (particularly the method `persistent_id`) and create your own persistence framework. However, this is an advanced issue and is not covered further in this book.

Unpickler `Unpickler(fileobj)`

Creates and returns an object `u` such that calling the `u.load` is equivalent to calling function `load` with the `fileobj` argument passed to `Unpickler`. To deserialize many objects from a file, `Unpickler` is more convenient and faster than repeated calls to the function `load`. You can subclass `pickle.Unpickler` to override `Unpickler` methods (particularly method `persistent_load`) and create your own persistence framework. However, this is an advanced issue and is not covered further in this book.

A pickling example

The following example handles the same task as the `json` example shown earlier, but uses `pickle` instead of `json` to serialize lists of `(filename, linenumber)` pairs as strings:

```
import collections, fileinput, pickle, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno
()
    for word in line.split():
        word_pos[word].append(pos)
dbm_out = dbm.open('indexfilep','n')
for word in word_pos:
    dbm_out[word] = pickle.dumps(word_pos[word],2)
dbm_out.close()
```

We can then use `pickle` to read back the data stored to the DBM-like file *indexfilep*, as shown in the following example:

```
import sys, pickle, dbm, linecache
dbm_in = dbm.open('indexfilep')
for word in sys.argv[1:]:
    if word not in dbm_in:
                          not found in index
         print('Word {!r}file'                    .format(word
),
               file=sys.stderr)
         continue
    places = pickle.loads(dbm_in[word])
    for fname, lineno in places:
        print('Word {!r} occurs in line {} of file {}:'.format(
               word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```

In v2, in both examples, to ensure the code works across platforms, you'd import and use the module `anydbm` instead of the package `dbm`.

Pickling instances

In order for `pickle` to reload an instance `x`, `pickle` must be able to import `x`'s class from the same module in which the class was defined when `pickle` saved the instance. Here is how `pickle` saves the state of instance object `x` of class `T` and later reloads the saved state into a new instance `y` of `T` (the first step of the reloading is always to make a new empty instance `y` of `T`, except where we explicitly say otherwise in the following):

- When `T` supplies the method `__getstate__`, `pickle` saves the result `d` of calling `T.__getstate__(x)`.
- When `T` supplies the method `__setstate__`, `d` can be of any type, and `pickle` reloads the saved state by calling `T.__setstate__(y, d)`.
- Otherwise, `d` must be a dictionary, and `pickle` just sets `y.__dict__ = d`.
- Otherwise, when `T` supplies the method `__getnewargs__`, and `pickle` is pickling with protocol 2 or higher, `pickle` saves the result `t` of calling `T.__getnewargs__(x)`; `t` must be a tuple.
- `pickle`, in this one case, does not start with an empty `y`, but rather creates `y` by executing `y = T.__new__(T, *t)`, which concludes the reloading.
- Otherwise, by default, `pickle` saves as `d` the dictionary `x.__dict__`.
- When `T` supplies the method `__setstate__`, `pickle` reloads the saved state by calling `T.__setstate__(y, d)`.
- Otherwise, `pickle` just sets `y.__dict__ = d`.

All the items in the `d` or `t` object that `pickle` saves and reloads (normally a dictionary or tuple) must in turn be instances of types suitable for pickling and unpickling (AKA *pickleable* objects), and the procedure just outlined may be repeated recursively, if necessary, until `pickle` reaches primitive pickleable built-in types (dictionaries, tuples, lists, sets, numbers, strings, etc.).

As mentioned in "The copy Module", the special methods `__getnewargs__`, `__getstate__`, and `__setstate__` also control the way instance objects are copied and deep-copied. If a class defines `__slots__`, and therefore its instances do not have a `__dict__`, `pickle` does its best to save and restore a dictionary equivalent to the names and values of the slots. However, such a class should define `__getstate__` and `__setstate__`; otherwise, its instances may not be correctly pickleable and copy-able through such best-effort endeavors.

Pickling customization with the copy_reg module

You can control how `pickle` serializes and deserializes objects of an arbitrary type by registering factory and reduction functions with the module `copy_reg`. This is particularly, though not exclusively, useful when you define a type in a C-coded Python extension. The `copy_reg` module supplies the following functions:

constructor `constructor(fcon)`

Adds `fcon` to the table of constructors, which lists all factory functions that `pickle` may call. `fcon` must be callable and is normally a function.

pickle `pickle(type,fred,fcon=None)`

Registers function `fred` as the *reduction function* for type `type`, where `type` must be a type object. To save an object `o` of type `type`, the module `pickle` calls `fred(o)` and saves the result. `fred(o)` must return a tuple `(fcon,t)` or `(fcon,t,d)`, where `fcon` is a constructor and `t` is a tuple. To reload `o`, `pickle` uses `o=fcon(*t)`. Then, when `fred` also returned a `d`, `pickle` uses `d` to restore `o`'s state (when `o` supplies `__setstate__`, `o.__setstate__(d)`; otherwise, `o.__dict__.update(d)`), as in "Pickling instances". If `fcon` is not `None`, `pickle` also calls `constructor(fcon)` to register `fcon` as a constructor.

`pickle` does not support pickling of code objects, but `marshal` does. Here's how you could customize pickling to support code objects by delegating the work to `marshal` thanks to `copy_reg`
:

```
>>> import pickle, copy_reg, marshal
>>> def marsh(x): return marshal.loads, (marshal.dumps(x),)
...
>>> c=compile('2+2','','eval')
>>> copy_reg.pickle(type(c), marsh)
>>> s=pickle.dumps(c, 2)
>>> cc=pickle.loads(s)
>>> print eval(cc)
4
```

Using marshal makes your code Python-version-dependent

Beware using `marshal` in your code, as the preceding example does: `marshal`'s serialization can vary with every version of Python, so using `marshal` means you may be unable to load objects serialized with one Python version, with other versions.

The shelve Module

The `shelve` module orchestrates the modules `pickle` (or `cPickle`, in v2, when available), `io` (in v3; in v2, `cStringIO` when available, and otherwise `StringIO`), and `dbm` (and its underlying modules for access to DBM-like archive files, as discussed in "DBM Modules"; that's in v3—`anydbm` in v2) in order to provide a simple, lightweight persistence mechanism.

`shelve` supplies a function `open` that is polymorphic to `anydbm.open`. The mapping `s` returned by `shelve.open` is less limited than the mapping `a` returned by `anydbm.open`. `a`'s keys and values must be strings. `s`'s keys must also be strings, but `s`'s values may be of any pickleable types. `pickle` customizations (`copy_reg`, `__getnewargs__`, `__getstate__`, and `__setstate__`) also apply to `shelve`, as `shelve` delegates serialization to `pickle`.

Beware of a subtle trap when you use `shelve` with mutable objects: when you operate on a mutable object held in a shelf, the changes don't "take" unless you assign the changed object back to the same index. For example:

```
import shelves = shelve.open('data')s['akey'] = list(range(4))print(s['akey'])
                      [0, 1, 2,                              # trying direct
           # prints: 3]          s['akey'].append(9)         mutation
                                                          [0, 1, 2,
print(s['akey'])           # doesn't "take"; prints: 3]          x = s['akey']
              # fetch the                                    # perform
              object           x.append(9)                 mutation          s['akey']
                  # key step: store the object
 = x              back!                              print(s['akey'])
                           [0, 1, 2, 3,
# now it "takes", prints: 9]
```

You can finesse this issue by passing the named argument `writeback=True` when you call `shelve.open`, but beware: if you do pass that argument, you may seriously impair the performance of your program.

A shelving example

The following example handles the same task as the earlier `json` and `pickle` examples, but uses `shelve` to persist lists of `(`*`filename`*`,`*`linenumber`*`)` pairs:

```
import collections, fileinput, shelve
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno
()
    for word in line.split():
        word_pos[word].append(pos)
sh_out = shelve.open('indexfiles','n')
sh_out.update(word_pos)
sh_out.close()
```

We must use `shelve` to read back the data stored to the DBM-like file *indexfiles*, as shown in the following example:

```
import sys, shelve, linecache
sh_in = shelve.open('indexfiles')

for word in sys.argv[1:]:
    if word not in sh_in:
                           not found in index
         print('Word {!r}file'                    .format(word
),
               file=sys.stderr)
         continue
    places = sh_in[word]
    for fname, lineno in places:
        print('Word {!r} occurs in line {} of file {}:'.format(
               word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```

These two examples are the simplest and most direct of the various equivalent pairs of examples shown throughout this section. This reflects the fact that the module `shelve` is higher-level than the modules used in previous examples.

DBM Modules

DBM, a long-time Unix tradition, is a family of libraries supporting data files with pairs of strings `(key,data)`, with fast fetching and storing of the data given a key, a usage pattern known as *keyed access*. Keyed access, while nowhere as powerful as the data-access functionality of relational DBs, imposes less overhead, yet may suffice for a given program's needs. If DBM-like files are sufficient, you may end up with a program that is smaller and faster than one using a relational DB.

v3 organizes DBM support in Python's standard library in a clean and elegant way: the package `dbm` exposes two general functions, and within the same package live other modules supplying specific implementations. v2's support, while functionally equivalent, evolved over time in a less organized fashion, thus ending up in a less elegant arrangement, as a collection of top-level modules not organized into a package. This section briefly presents v3's organization first, and then v2's, with references from the latter back to the former to show equivalent functionality.

The v3 dbm Package

v3's `dbm` package supplies the following two top-level functions:

open `open(filepath,flag='r',mode=0o666)`

Opens or creates the DBM file named by `filepath` (any path to a file) and returns a mapping object corresponding to the DBM file. When the DBM file already exists, `open` uses the function `whichdb` to determine which DBM submodule can handle the file. When `open` creates a new DBM file, it chooses the first available `dbm` submodule in the following order of preference: `gnu`, `ndbm`, `dumb`.

`flag` is a one-character string that tells `open` how to open the file and whether to create it, as shown in Table 11-1. `mode` is an integer that `open` uses as the file's permission bits if `open` creates the file, as covered in "Creating a "file" Object with io.open".

Table 11-1. Flag values for anydbm.open

Flag	Read-only?	If file exists	If file does not exist
`'r'`	Yes	`open` opens the file.	`open` raises error.
`'w'`	No	`open` opens the file.	`open` raises error.
`'c'`	No	`open` opens the file.	`open` creates the file.
`'n'`	No	`open` truncates the file.	`open` creates the file.

`dbm.open` returns a mapping object `m` with a subset of the functionality of dictionaries (covered in "Dictionary Operations"). `m` only accepts strings as keys and values, and the only non-special mapping methods *m* supplies are *m*`.get`, *m*`.keys`, and *m*`.setdefault`. You can bind, rebind, access, and unbind items in `m` with the same indexing syntax *m*`[`*key*`]` that you would use if `m` were a dictionary. If `flag` is `'r'`, `m` is read-only, so that you can only access `m`'s items, not bind, rebind, or unbind them. You can check if a string `s` is a key in `m` with the usual expression *s* `in` *m*; you cannot iterate directly on `m`, but you can, equivalently, iterate on *m*`.keys()`.

One extra method that `m` supplies is *m*`.close`, with the same semantics as the `close` method of a "file" object. Just like for "file" objects, you should ensure *m*`.close()` is called when you're done using `m`. The `try`/`finally` statement (covered in "try/finally") is a good way to ensure finalization, but the `with` statement, covered in "The with Statement", is even better than `try`/`finally` (you can use `with`, since `m` is a context manager).

whichdb `whichdb(`*filename*`)`

Opens and reads the file specified by `filename` to discover which `dbm` submodule created the file. `whichdb` returns `None` when the file does not exist or cannot be opened and read. `whichdb` returns `''` when the file exists and can be opened and read, but it is not possible to determine which `dbm` submodule created the file (typically, this means that the file is not a DBM file). If it can find out which module can read the DBM-like file, `whichdb` returns a string that names a `dbm` submodule, such as `'dbm.ndbm'`, `'dbm.dumbdbm'`, or `'dbm.gdbm'`.

In addition to these two top-level functions, the v3 package `dbm` contains specific modules, such as `ndbm`, `gnu`, and `dumb`, which provide various implementations of DBM functionality, which you normally access only via the two top-level functions of the package `dbm`. Third-party packages can install further implementation modules in `dbm`.

The only implementation module of the `dbm` package that's guaranteed to exist on all platforms is `dumb`. The `dumb` submodule of `dbm` has minimal DBM functionality and mediocre performance. `dumb`'s only advantage is that you can use it anywhere, since `dumb` does not rely on any library. You don't normally `import dbm.dumb`: rather,

import
dbm , and let dbm.open supply the best DBM module available, defaulting to dumb if nothing else is available on the current Python installation. The only case in which you import dumb directly is the rare one in which you need to create a DBM-like file guaranteed to be readable from any Python installation. The dumb module supplies an open function polymorphic to dbm's.

v2's dbm modules

v2's anydbm module is a generic interface to any other DBM module. anydbm supplies a single factory function open, equivalent to v3's dbm.open.

v2's whichdb module supplies a single function, whichdb, equivalent to v3's dbm.whichdb.

DBM implementation modules, in v2, are top-level ones, named dbm, gdbm, dumbdbm, and so forth, and otherwise paralleling v3 dbm submodules.

Examples of DBM-Like File Use

DBM's keyed access is suitable when your program needs to record persistently the equivalent of a Python dictionary, with strings as both keys and values. For example, suppose you need to analyze several text files, whose names are given as your program's arguments, and record where each word appears in those files. In this case, the keys are words and, therefore, intrinsically strings. The data you need to record for each word is a list of *(filename, linenumber)* pairs. However, you can encode the data as a string in several ways—for example, by exploiting the fact that the path separator string os.pathsep (covered in "Path-String Attributes of the os Module") does not normally appear in filenames. (More solid, general, and reliable approaches to the issue of encoding data as strings are covered with the same example in "Serialization".) With this simplification, the program that records word positions in files might be as follows, in v3:

```
import collections, fileinput, os, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = '{}{}{}'.format(
        fileinput.filename(), os.pathsep, fileinput.filelineno
())
    for word in line.split():
        word_pos[word].append(pos)
sep2 = os.pathsep * 2
with dbm.open('indexfile','n') as dbm_out:
    for word in word_pos:
        dbm_out[word] = sep2.join(word_pos[word])
```

We can read back the data stored to the DBM-like file *indexfile* in several ways. The following example accepts words as command-line arguments and prints the lines where the requested words appear, in v3:

```
import sys, os, dbm, linecache
dbm_in = dbm.open('indexfile')
sep = os.pathsep
sep2 = sep * 2
for word in sys.argv[1:]:
    if word not in dbm_in:
                          not found in index
         print('Word {!r}file'                     .format(word
),
               file=sys.stderr)
         continue
    places = dbm_in[word].split(sep2)
    for place in places:
        fname, lineno = place.split(sep)
        print('Word {!r} occurs in line {} of file {}:'.format(
               word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```

In v2, in both examples, import and use the module `anydbm` instead of the package `dbm`.

Berkeley DB Interfacing

v2 comes with the `bsddb` package, which wraps the Berkeley Database (also known as BSD DB) library if that library is installed on your system and your Python installation is built to support it. However, `bsddb` is deprecated in v2, and not present in v3, so we cannot recommend it. If you do need to interface to a BSD DB archive, we recommend instead the excellent third-party package bsddb3.

The Python Database API (DBAPI) 2.0

As we mentioned earlier, the Python standard library does not come with an RDBMS interface (except for `sqlite3`, covered in "SQLite", which is a rich implementation, not just an interface). Many third-party modules let your Python programs access specific DBs. Such modules mostly follow the Python Database API 2.0 standard, also known as the DBAPI, as specified in PEP 249.

After importing any DBAPI-compliant module, call the module's `connect` function with DB-specific parameters. `connect` returns `x`, an instance of `Connection`, which represents a connection to the DB. `x` supplies `commit` and `rollback` methods to deal with transactions, a `close` method to call as soon as you're done with the DB, and a `cursor` method to return `c`, an instance of `Cursor`. `c` supplies the methods and attributes used for DB operations. A DBAPI-compliant module also supplies exception classes, descriptive attributes, factory functions, and type-description attributes.

Exception Classes

A DBAPI-compliant module supplies the exception classes `Warning`, `Error`, and several subclasses of `Error`. `Warning` indicates anomalies such as data truncation on insertion. `Error`'s subclasses indicate various kinds of errors that your program can encounter when dealing with the DB and the DBAPI-compliant module that interfaces to it. Generally, your code uses a statement of the form:

```
try:
    ...
except module.Error as err:
    ...
```

to trap all DB-related errors that you need to handle without terminating.

Thread Safety

When a DBAPI-compliant module has a `threadsafety` attribute greater than `0`, the module is asserting some level of thread safety for DB interfacing. Rather than relying on this, it's usually safer, and always more portable, to ensure that a single thread has exclusive access to any given external resource, such as a DB, as outlined in "Threaded Program Architecture".

Parameter Style

A DBAPI-compliant module has an attribute called `paramstyle` to identify the style of markers used as placeholders for parameters. Insert such markers in SQL statement strings that you pass to methods of `Cursor` instances, such as the method `execute`, to use runtime-determined parameter values. Say, for example, that you need to fetch the rows of DB table `ATABLE` where field `AFIELD` equals the current value of Python variable `x`. Assuming the cursor instance is named `c`, you *could* theoretically (but ill-advisedly) perform this task with Python's string formatting:

```
          'SELECT * FROM ATABLE WHERE
c.execute(AFIELD=                            {!r}'.format(x))
```

Avoid SQL query string formatting: use parameter substitution

String formatting is *not* the recommended approach. It generates a different string for each value of `x`, requiring statements to be parsed and prepared anew each time; it also opens up the possibility of security weaknesses such as SQL injection vulnerabilities. With parameter substitution, you pass to `execute` a single statement string, with a placeholder instead of the parameter value. This lets `execute` parse and prepare the statement just once, for better performance; more importantly, parameter substitution improves solidity and security.

For example, when a module's `paramstyle` attribute (described next) is `'qmark'`, express the preceding query as:

```
          'SELECT * FROM ATABLE WHERE
c.execute(AFIELD=?'                            , (some_value,))
```

The read-only string attribute `paramstyle` tells your program how it should use parameter substitution with that module. The possible values of `paramstyle` are:

`format`

> The marker is `%s`, as in old-style string formatting (always with `s`: never use other type indicator letters, whatever the data's type). A query looks like:

```
           'SELECT * FROM ATABLE WHERE
c.execute(AFIELD=                             %s', (some_value,))
```

`named`

The marker is `:name`, and parameters are named. A query looks like:

```
           'SELECT * FROM ATABLE WHERE
c.execute(AFIELD=:x'                             , {'x':some_value})
```

`numeric`

The marker is `:n`, giving the parameter’s number, `1` and up. A query looks like:

```
           'SELECT * FROM ATABLE WHERE
c.execute(AFIELD=:1'                             , (some_value,))
```

`pyformat`

The marker is `%(name)s`, and parameters are named. Always use `s`: never use other type indicator letters, whatever the data’s type. A query looks like:

```
           'SELECT * FROM ATABLE WHERE
c.execute(AFIELD=                             %(x)s', {'x':some_value})
```

`qmark`

The marker is `?`. A query looks like:

```
           'SELECT * FROM ATABLE WHERE
c.execute(AFIELD=?'                             , (x,))
```

When parameters are named (i.e., when `paramstyle` is `'pyformat'` or `'named'`), the second argument of the `execute` method is a mapping. Otherwise, the second argument is a sequence.

format and pyformat only accept type indicator s

The *only* valid type indicator letter for `format` or `pyformat` is `s`; neither accepts any other type indicator—for example, never use `%d` nor `%(name)d`. Use `%s` or `%(name)s` for all parameter substitutions, regardless of the type of the data.

Factory Functions

Parameters passed to the DB via placeholders must typically be of the right type: this means Python numbers (integers or floating-point values), strings (bytes or Unicode), and `None` to represent SQL `NULL`. There is no type universally used to represent dates, times, and binary large objects (BLOBs). A DBAPI-compliant module supplies factory functions to build such objects. The types used for this purpose by most DBAPI-compliant modules are those supplied by the modules `datetime` or `mxDateTime` (covered in Chapter 12), and strings or buffer types for BLOBs. The factory functions specified by the DBAPI are as follows:

Binary	`Binary(`*`string`*`)` Returns an object representing the given `string` of bytes as a BLOB.
Date	`Date(`*`year`*`,`*`month`*`,`*`day`*`)` Returns an object representing the specified date.
DateFromTicks	`DateFromTicks(`*`s`*`)` Returns an object representing the date `s` seconds after the epoch of module `time`, covered in Chapter 12. For example, `DateFromTicks(time.time())` means "today."
Time	`Time(`*`hour`*`,`*`minute`*`,`*`second`*`)` Returns an object representing the specified time.
TimeFromTicks	`TimeFromTicks(`*`s`*`)` Returns an object representing the time `s` seconds after the epoch of module `time`, covered in Chapter 12. For example, `TimeFromTicks(time.time())` means "the current time of day."
Timestamp	`Timestamp(`*`year`*`,`*`month`*`,`*`day`*`,`*`hour`*`,`*`minute`*`,`*`second`*`)` Returns an object representing the specified date and time.
TimestampFromTicks	`TimestampFromTicks(`*`s`*`)` Returns an object representing the date and time `s` seconds after the epoch of module `time`, covered in Chapter 12. For example, `TimestampFromTicks( time.time())` is the current date and time.

Type Description Attributes

A `Cursor` instance's attribute `description` describes the types and other characteristics of each column of the `SELECT` query you last `execute`d on that cursor. Each column's *type* (the second item of the tuple describing the column) equals one of the following attributes of the DBAPI-compliant module:

`BINARY`	Describes columns containing BLOBs
`DATETIME`	Describes columns containing dates, times, or both
`NUMBER`	Describes columns containing numbers of any kind
`ROWID`	Describes columns containing a row-identification number
`STRING`	Describes columns containing text of any kind

A cursor's description, and in particular each column's type, is mostly useful for introspection about the DB your program is working with. Such introspection can help you write general modules and work with tables using different schemas, including schemas that may not be known at the time you are writing your code.

The connect Function

A DBAPI-compliant module's `connect` function accepts arguments that depend on the kind of DB and the specific module involved. The DBAPI standard recommends that `connect` accept named arguments. In particular, `connect` should at least accept optional arguments with the following names:

`database`	Name of the specific database to connect
`dsn`	Data-source name to use for the connection
`host`	Hostname on which the database is running
`password`	Password to use for the connection
`user`	Username for the connection

Connection Objects

A DBAPI-compliant module's `connect` function returns an object `x` that is an instance of the class `Connection`. `x` supplies the following methods:

close	`x.close()` Terminates the DB connection and releases all related resources. Call `close` as soon as you're done with the DB. Keeping DB connections open needlessly can be a serious resource drain on the system.
commit	`x.commit()` Commits the current transaction in the DB. If the DB does not support transactions, `x.commit()` is an innocuous no-op.
cursor	`x.cursor()` Returns a new instance of the class `Cursor`, covered in "Cursor Objects".
rollback	`x.rollback()` Rolls back the current transaction in the DB. If the DB does not support transactions, `x.rollback()` raises an exception. The DBAPI recommends that, for DBs that do not support transactions, the class `Connection` supplies no `rollback` method, so that `x.rollback()` raises `AttributeError`: you can test whether transactions are supported with `hasattr(x,'rollback')`.

Cursor Objects

A `Connection` instance provides a `cursor` method that returns an object `c` that is an instance of the class `Cursor`. A SQL cursor represents the set of results of a query and lets you work with the records in that set, in sequence, one at a time. A cursor as modeled by the DBAPI is a richer concept, since it's the only way your program executes SQL queries in the first place. On the other hand, a DBAPI cursor allows you only to advance in the sequence of results (some relational DBs, but not all, also provide higher-functionality cursors that are able to go backward as well as forward), and does not support the SQL clause `WHERE CURRENT OF CURSOR`. These limitations of DBAPI cursors enable DBAPI-compliant modules to easily provide DBAPI cursors even on RDBMSes that provide no real SQL cursors at all. An instance of the class `Cursor` `c` supplies many attributes and methods; the most frequently used ones are the following:

close `c.close()`

Closes the cursor and releases all related resources.

description A read-only attribute that is a sequence of seven-item tuples, one per column in the last query executed:

```
name, typecode, displaysize, internalsize, precision,
scale, nullable
```

`c.description` is `None` if the last operation on `c` was not a `SELECT` query or returned no usable description of the columns involved. A cursor’s description is mostly useful for introspection about the DB your program is working with. Such introspection can help you write general modules that are able to work with tables using different schemas, including schemas that may not be fully known at the time you are writing your code.

execute `c.execute(statement,parameters=None)`

Executes a SQL `statement` string on the DB with the given `parameters`. `parameters` is a sequence when the module’s `paramstyle` is `'format'`, `'numeric'`, or `'qmark'`, and a mapping when `paramstyle` is `'named'` or `'pyformat'`. Some DBAPI modules require the sequences to be specifically tuples.

executemany `c.executemany(statement,*parameters)`

Executes a SQL `statement` on the DB, once for each item of the given `parameters`. `parameters` is a sequence of sequences when the module’s `paramstyle` is `'format'`, `'numeric'`, or `'qmark'`, and a sequence of mappings when `paramstyle` is `'named'` or `'pyformat'`. For example, the statement:

```
              'UPDATE atable SET x=?
c.executemany('
              'WHERE
              y=?'       ,(12,23),(23,34
))
```

when `paramstyle` is `'qmark'`, is equivalent to—but faster than—the two statements:

```
          'UPDATE atable SET x=12 WHERE
c.execute(y=23'                              )

          'UPDATE atable SET x=23 WHERE
c.execute(y=34'                              )
```

fetchall `c.fetchall()`

Returns all remaining result rows from the last query as a sequence of tuples. Raises an exception if the last operation was not a `SELECT` query.

fetchmany	`c.fetchmany(n)` Returns up to `n` remaining result rows from the last query as a sequence of tuples. Raises an exception if the last operation was not a `SELECT` query.
fetchone	`c.fetchone()` Returns the next result row from the last query as a tuple. Raises an exception if the last operation was not a `SELECT` query.
rowcount	A read-only attribute that specifies the number of rows fetched or affected by the last operation, or `-1` if the module is unable to determine this value.

DBAPI-Compliant Modules

Whatever relational DB you want to use, there's at least one (often more than one) Python DBAPI-compliant module downloadable from the Internet. There are so many DBs and modules, and the set of possibilities is so constantly shifting, that we couldn't possibly list them all, nor (importantly) could we maintain the list over time. Rather, we recommend you start from the community-maintained wiki page, which has at least a fighting chance to be complete and up-to-date at any time.

What follows is therefore only a very short, time-specific list of a very few DBAPI-compliant modules that, at the time of writing, are very popular themselves, and interface to very popular open source DBs.

ODBC

> Open DataBase Connectivity (ODBC) is a standard way to connect to many different DBs, including a few not supported by other DBAPI-compliant modules. For an ODBC-compliant DBAPI-compliant module with a liberal open source license, use pyodbc; for a commercially supported one, mxODBC.

MySQL

> MySQL is a popular open source RDBMS, currently owned by Oracle. Oracle's own "official" DBAPI-compliant interface to it is MySQL Connector/Python.

PostgreSQL

> PostgreSQL is an excellent open source RDBMS. The most popular DBAPI-compliant interface to it is psycopg2.

SQLite

SQLite is "a self-contained, server-less, zero-configuration, transactional SQL database engine," which is the most widely deployed DB engine in the world—it's a C-coded library that implements a DB within a single file, or even in memory for sufficiently small and transient cases. Python's standard library supplies the package `sqlite3`, which is a DBAPI-compliant interface to `SQLite`.

`SQLite` has rich advanced functionality, with many options you can choose; `sqlite3` offers access to much of that functionality, plus further possibilities to make interoperation between your Python code and the underlying DB even smoother and more natural. In this book, we don't cover every nook and cranny of these two powerful software systems; we focus on the subset that is most commonly used and most useful. For a great level of detail, including examples and tips about best practices, see SQLite's documentation and sqlite3's online documentation. If you want a book on the subject, we recommend O'Reilly's *Using SQLite*.

Package `sqlite3` supplies the following functions:

connect

`connect(`*`filepath`*`, `*`timeout`*`=5.0, `*`detect_types`*`=0, `*`isolation_level`*`='', `*`check_same_thread`*`=True, `*`factory`*`=Connection, `*`cached_statements`*`=100, `*`uri`*`=False)`

`connect` connects to the `SQLite` DB in the file named by `filepath` (creating it if necessary) and returns an instance of the `Connection` class (or subclass thereof passed as `factory`). To create an in-memory DB, pass `':memory:'` as the first argument, `filepath`.

The `uri` argument can be passed only in v3; if `True`, it activates SQLite's URI functionality, allowing a few extra options to be passed, along with the file path, via the `filepath` argument.

`timeout` is the number of seconds to wait, before raising an exception, if another connection is keeping the DB locked in a transaction.

`sqlite3` directly supports only the SQLite native types: `BLOB`, `INTEGER`, `NULL`, `REAL`, and `TEXT` (any other type name is treated as `TEXT` unless properly detected and passed through a converter registered with the function `register_converter`, covered later in this section), converting as follows:

SQLite type	**Python type**
`BLOB`	`buffer` in v2, `bytes` in v3
`INTEGER`	`int` (or, in v2, `long` for very large values)
`NULL`	`None`
`REAL`	`float`
`TEXT`	depends on the `text_factory` attribute of the `Connection` instance, covered later in this section; by default, `unicode` in v2, `str` in v3

To allow type name detection, pass as `detect_types` either of the constants `PARSE_COLNAMES` and `PARSE_DECLTYPES`, supplied by the `sqlite3` package (or both, joining them with the `|` bitwise-or operator).

When you pass `detect_types=sqlite3.PARSE_COLNAMES`, the type name is taken from the name of the column in the SELECT SQL statement that retrieves the column; for example, a column retrieved as *`foo`*` AS [`*`foo`*` CHAR(10)]` has a type name of `CHAR`.

When you pass `detect_types=sqlite3.PARSE_DECLTYPES`, the type name is taken from the declaration of the column in the original `CREATE TABLE` or `ALTER TABLE` SQL statement that added the column; for example, a column declared as *`foo`*` CHAR(10)` has a type name of `CHAR`.

When you pass `detect_types=sqlite3.PARSE_COLNAMES|sqlite3.PARSE_DECLTYPES`, both mechanisms are used, with precedence given to the column name when it has at least two words (the second word gives the type name in this case), falling back to the type that was

	given for that column at declaration (the first word of the declaration type gives the type name in this case). `isolation_level` lets you exercise some control over how SQLite processes transactions; it can be `''` (the default), `None` (to use *autocommit* mode), or one of the three strings `'DEFERRED'`, `'EXCLUSIVE'`, and `'IMMEDIATE'`. The SQLite online docs cover the details of types of transactions and their relation to the various levels of file locking that SQLite intrinsically performs. By default, a connection object can be used only in the Python thread that created it, to avoid accidents that could easily corrupt the DB due to minor bugs in your program; minor bugs are common in multithreaded programming. If you're entirely confident about your threads' use of locks and other synchronization mechanisms, and do need to reuse a connection object among multiple threads, you can pass `check_same_thread=False`: then, `sqlite3` performs no checks, trusting your assertion that you know what you're doing and that your multithreading architecture is 100% bug-free—good luck! `cached_statements` is the number of SQL statements that `sqlite3` caches in a parsed and prepared state, to avoid the overhead of parsing them repeatedly. You can pass in a value lower than the default `100` to save a little memory, or a larger one if your application uses a dazzling variety of SQL statements.
register_adapter	`register_adapter(`*`type`*`,`*`callable`*`)` `register_adapter` registers `callable` as the adapter, from any object of Python type `type`, to a corresponding value of one of the few Python types that `sqlite3` handles directly—`int`, `float`, `str` (in v3, also `bytes`; in v2, also `buffer`, `long`, `unicode`—also note that, in v2, a `str` must be encoded in `'utf-8'`). `callable` must accept a single argument, the value to adapt, and return a value of a type that `sqlite3` handles directly.
register_converter	`register_converter(`*`typename`*`,`*`callable`*`)` `register_converter` registers `callable` as the converter, from any value identified in SQL as being of a type named `typename` (see parameter `detect_types` to function `connect` for an explanation of how the type name is identified), to a corresponding Python object. `callable` must accept a single argument, the string form of the value obtained from SQL, and return the corresponding Python object. The `typename` matching is case-sensitive.

In addition, `sqlite3` supplies the classes `Connection`, `Cursor`, and `Row`. Each can be subclassed for further customization; however, this is an advanced issue that we do not cover further in this book. The `Cursor` class is a standard DBAPI cursor class, except for an extra convenience method `executescript` accepting a single argument, a string of multiple statements separated by `;` (no parameters). The other two classes are covered in the following sections.

class sqlite3.Connection

In addition to the methods common to all `Connection` classes of DBAPI-compliant modules, covered in "Connection Objects", `sqlite3.Connection` supplies the following methods and other attributes:

create_aggregate	`create_aggregate(`*`name`*`,`*`num_params`*`,`*`aggregate_class`*`)` `aggregate_class` must be a class supplying two instance methods: `step`, accepting exactly `num_param` arguments, and `finalize`, accepting no arguments and returning the final result of the aggregate, a value of a type natively supported by `sqlite3`. The aggregate function can be used in SQL statements by the given `name`.
create_collation	`create_collation(`*`name`*`,`*`callable`*`)` `callable` must accept two bytestring arguments (encoded in `'utf-8'`), and return `-1` if the first must be considered “less than” the second, `1` if it must be considered “greater than,” and `0` if it must be considered “equal,” for the purposes of this comparison. Such a collation can be named by the given `name` in a SQL `ORDER BY` clause in a `SELECT` statement.
create_function	`create_function(`*`name`*`,`*`num_params`*`,`*`func`*`)` `func` must accept exactly `num_params` arguments and return a value of a type natively supported by `sqlite3`; such a user-defined function can be used in SQL statements by the given `name`.
interrupt	`interrupt()` Call from any other thread to abort all queries executing on this connection (raising an exception in the thread using the connection).
isolation_level	A read-only attribute that’s the value given as `isolation_level` parameter to the `connect` function.
iterdump	`iterdump()` Returns an iterator that yields strings: the SQL statements that build the current DB from scratch, including both schema and contents. Useful, for example, to persist an in-memory DB to one on disk for future reuse.
row_factory	A callable that accepts the cursor and the original row as a tuple, and returns an object to use as the real result row. A common idiom is *`x`*`.row_factory=sqlite3.Row`, to use the highly optimized `Row` class covered in “class sqlite3.Row”, supplying both index-based and case-insensitive name-based access to columns with negligible overhead.
text_factory	A callable that accepts a single bytestring parameter and returns the object to use for that TEXT column value—by default, `str` in v3, `unicode` in v2, but you can set it to any similar callable.
total_changes	The total number of rows that have been modified, inserted, or deleted since the connection was created.

class sqlite3.Row

`sqlite3` supplies the class `Row`, which is mostly like a `tuple` but also supplies the method `keys()`, returning a list of column names, and supports indexing by a column name as an extra alternative to indexing by column number.

A sqlite3 example

The following example handles the same task as the examples shown earlier in the chapter, but uses `sqlite3` for persistence, without creating the index in memory:

```
import fileinput, sqlite3
connect = sqlite3.connect('database.db')
cursor = connect.cursor()
               'CREATE TABLE IF NOT EXISTS Words
cursor.execute('
               '(Word TEXT, File TEXT, Line
               INT)'                             )
for line in fileinput.input():
    f, l = fileinput.filename(), fileinput.filelineno()
                       'INSERT INTO Words VALUES (:w, :f,
    cursor.executemany(:l)'
,
        [{'w':w, 'f':f, 'l':l} for w in line.split()])
connect.commit()
connect.close()
```

We can then use `sqlite3` to read back the data stored in the DB file *database.db*, as shown in the following example:

```
import sys, sqlite3, linecache
connect = sqlite3.connect('database.db')
cursor = connect.cursor()
for word in sys.argv[1:]:
                   'SELECT File, Line FROM Words
    cursor.execute('
                   'WHERE
                   Word=?'       , [word])
    places = cursor.fetchall()
    if not places:
                          not found in index
         print('Word {!r}file'                    .format(word
),
               file=sys.stderr)
         continue
    for fname, lineno in places:
        print('Word {!r} occurs in line {} of file {}:'.format(
               word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```