Chapter 12. 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* back (loaded and re-created) from the byte stream. *Object persistence* relies on 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 files that use *keyed access* to enable selective reading and updating of 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; for this purpose we recommend O'Reilly's ***SQL in a Nutshell***, by Kevin Kline, Regina Obe, and Leo Hsu.) Unfortunately, despite the existence of SQL standards, no two RDBMSs 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**. SQLite, "a self-contained, server-less, zero-configuration, transactional SQL DB engine," 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 several **NoSQL** DBs, such as **Redis** and **MongoDB**, 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*, which means formatting for *data interchange*.

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

In this section, we cover Python's `csv`, `json`, `pickle`, and `shelve` modules. 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 csv Module

While the CSV (standing for *comma-separated values*[1]) format isn't usually considered a form of serialization, it is a widely used and convenient interchange format for tabular data. Since much data is tabular, CSV data is used a lot, despite some lack of agreement on exactly how it should be represented in files. To overcome this issue, the `csv` module provides a number of *dialects* (specifications of the way particular sources encode CSV data) and lets you define your own dialects. You can register additional dialects and list the available dialects by calling the `csv.list_dialects` function. For further information on dialects, consult **the module's documentation**.

csv functions and classes

The `csv` module exposes the functions and classes detailed in **Table 12-1**. It provides two kinds of readers and writers to let you handle CSV data rows in Python as either lists or dictionaries.

Table 12-1. Functions and classes of the `csv` module

`reader`	`reader(`*`csvfile`*`, dialect='excel', **`*`kw`*`)` Creates and returns a reader object *`r`*. *`csvfile`* can be any iterable object yielding text rows as `str`s (usually a list of lines or a file opened with `newline=''`[a]); `dialect` is the name of a registered dialect. To modify the dialect, add named arguments: their values override dialect fields of the same name. Iterating over *`r`* yields a sequence of lists, each containing the elements from one row of *`csvfile`*.
`writer`	`writer(`*`csvfile`*`, dialect='excel', **`*`kw`*`)` Creates and returns a writer object *`w`*. *`csvfile`* is an object with a `write` method (if a file, open it with `newline=''`); *`dialect`* is the name of a registered dialect. To modify the dialect, add named arguments: their values override dialect fields of the same name. *`w`*`.writerow` accepts a sequence of values and writes their CSV representation as a row to *`csvfile`*. *`w`*`.writerows` accepts an iterable of such sequences and

	calls `w.writerow` on each. You are responsible for closing *csvfile*.
`DictReader`	`DictReader(`*csvfile*`, fieldnames=`**`None`**`, restkey=`**`None`**`, restval=`**`None`**`, dialect='excel', *`*`args`*`, **`*`kw`*`)` Creates and returns an object *r* that iterates over *csvfile* to generate an iterable of dictionaries (-3.8 `OrderedDicts`), one for each row. When the `fieldnames` argument is given, it is used to name the fields in *csvfile*; otherwise, the field names are taken from the first row of *csvfile*. If a row contains more columns than field names, the extra values are saved as a list with the key `restkey`. If there are insufficient values in any row, then those column values will be set to `restval`. `dialect`, *kw*, and *args* are passed to the underlying reader object.
`DictWriter`	`DictWriter(`*csvfile*`, `*fieldnames*`, restval='', extrasaction='raise', dialect='excel', *`*`args`*`, **`*`kwds`*`)` Creates and returns an object *w* whose `writerow` and `writerows` methods take a dictionary or iterable of dictionaries and write them using the *csvfile*'s `write` method. *fieldnames* is a sequence of `str`s, the keys to the dictionaries. *restval* is the value used to fill up a dictionary that's missing some keys. `extrasaction` specifies what to do when a dictionary has extra keys not listed in *fieldnames*: when `'raise'`, the default, the function raises `ValueError` in such cases; when `'ignore'`, the function ignores such errors. `dialect`, *kw*, and *args* are passed to the underlying reader object. You are responsible for closing *csvfile* (usually a file opened with `newline=''`).

a Opening a file with `newline=''` allows the `csv` module to use its own newline processing and correctly handle dialects in which text fields may contain newlines.

A csv example

Here is a simple example using `csv` to read color data from a list of strings:

```
import csv

color_data = '''\
color,r,g,b
red,255,0,0
green,0,255,0
blue,0,0,255
cyan,0,255,255
```

```
magenta,255,0,255
yellow,255,255,0
'''.splitlines()

colors = {row['color']:
          row for row in csv.DictReader(color_data)}

print(colors['red'])
# prints: {'color': 'red', 'r': '255', 'g': '0', 'b': '0'}
```

Note that the integer values are read as strings. `csv` does not do any data conversion; that needs to be done by your program code with the `dicts` returned from `DictReader`.

The json Module

The standard library's `json` module supports serialization for Python's native data types (`tuple`, `list`, `dict`, `int`, `str`, etc.). To serialize instances of your own custom classes, you should implement corresponding classes inheriting from `JSONEncoder` and `JSONDecoder`.

json functions

The `json` module supplies four key functions, detailed in **Table 12-2**.

Table 12-2. Functions of the `json` module

`dump`	`dump(`*`value`*`,` *`fileobj`*`, skipkeys=`**`False`**`, ensure_ascii=`**`True`**`, check_circular=`**`True`**`, allow_nan=`**`True`**`, cls=JSONEncoder, indent=`**`None`**`, separators=(', ', ': '), default=`**`None`**`, sort_keys=`**`False`**`, **`*`kw`*`)` 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`. Each call to *`fileobj`*`.write` passes a text string as an argument. 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`**) raise 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.
`dump` *(cont.)*	When `ensure_ascii` is **`True`** (the default), all non-ASCII characters in the output are escaped; when it's **`False`**, they're output as is. When `check_circular` is **`True`** (the default), containers in *`value`* are checked for circular references and a `ValueError` exception is raised if any 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`** (the default), float scalars `nan`, `inf`, and `-inf` are output as their respective JavaScript equivalents, `NaN`, `Infinity`, and `-Infinity`; when it's **`False`**, the presence of such scalars raises a `ValueError`

exception.

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` that 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 it's an `int <= 0`, `dump` just inserts `\n` characters. When `indent` is **`None`** (the default), `dump` uses the most compact representation. `indent` can also be a `str`—for example, `'\t'`—and in that case `dump` uses that string for indenting.

`separators` must be a tuple with two items, respectively the strings used to separate items and to separate 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 it must return a serializable object or 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 **`False`**, they're output in whatever is their natural order of iteration (nowadays, for most mappings, insertion order).

`dumps`

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

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`.

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 to later call `load` more than once to deserialize the objects, as would be possible, for example, with `pickle` (discussed in the following section). So, technically, JSON serializes just one object per file. However, that one object can be a `list` or `dict` that can contain as many items as you wish.

`load`

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

	`object_pairs_hook=`**`None`**`, **`*`kw`*`)` 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`. The call to *fileobj*`.read` must return a text (Unicode) string. 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`, 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`, 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_hooks`'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: a `str` representing a `float`, an `int`, or one of the three special constants `'NaN'`, `'Infinity'`, or `'-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 the built-in function `float`, `parse_int` is `int`, and `parse_constant` is a function that returns one of the three special float scalars `nan`, `inf`, or `-inf`, as appropriate. For example, you could pass `parse_float=decimal.Decimal` to ensure that all numbers in the result that would normally be `floats` are instead decimals (covered in **"The decimal Module"**).
`loads`	`loads(`*s*`, cls=JSONDecoder, object_hook=`**`None`**`, parse_float=float, parse_int=int, parse_constant=`**`None`**`, object_pairs_hook=`**`None`**`, **`*`kw`*`)` Creates and returns the object *v* previously serialized

into the string `s`. All arguments 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 the `fileinput` module to iterate through all the files given as program arguments, and `json` to encode the 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)
with dbm.open('indexfilem', 'n') as dbm_out:
    for word, word_positions in word_pos.items():
        dbm_out[word] = json.dumps(word_positions)
```

We can then use `json` to deserialize the data stored in the DBM-like file *indexfilem*, as in the following example:

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

The pickle Module

The `pickle` module supplies factory functions, named `Pickler` and `Unpickler`, to generate objects (instances of nonsubclassable types, not classes) that wrap files and supply Python-specific serialization mechanisms. Serializing and deserializing via these modules is also known as *pickling* and *unpickling*.

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 UNDERWAY

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 several compact binary protocols. **Table 12-3** lists the available protocols.

Table 12-3. `pickle` protocols

Protocol	Format	Added in Python version	Description
`0`	ASCII	1.4[a]	Human-readable format, slow to serialize/deserialize
`1`	Binary	1.5	Early binary format, superseded by protocol 2
`2`	Binary	2.3	Improved support for later Python 2 features
`3`	Binary	3.0	(-3.8 default) Added specific support for `bytes` objects
`4`	Binary	3.4	(3.8+ default) Included support for very large objects
`5`	Binary	3.8	3.8+ Added features to support pickling as serialization for transport between processes, per PEP 574

a Or possibly earlier. This is the oldest version of documentation available at Python.org.

ALWAYS PICKLE WITH PROTOCOL 2 OR HIGHER

Always use *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.[2] `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 (i.e., attributes) of their respective modules. Consider the following example:

```
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 an `AttributeError`: 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 is nested inside the function `adder`. Similar issues apply to pickling nested functions and nested classes (i.e., classes not at the top level).

pickle functions and classes

The `pickle` module exposes the functions and classes listed in **Table 12-4**.

Table 12-4. Functions and classes of the `pickle` module

`dump`, `dumps`	`dump(`*`value`*`, `*`fileobj`*`, protocol=None, bin=None)`, `dumps(`*`value`*`, protocol=None, bin=None)` `dumps` returns a bytestring representing the 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`*`))`. The `protocol` parameter can be `0` (ASCII output, the slowest and bulkiest option), or a larger `int` for various kinds of binary output (see **Table 12-3**). Unless `protocol` is `0`, the *`fileobj`* parameter to dump must be open for binary writing. Do not pass the `bin` parameter, which exists only for compatibility with old versions of Python.
`load`, `loads`	`load(`*`fileobj`*`)`, `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()`.
`load`, `loads` *(cont.)*	`loads` creates and returns the object *`v`* represented by bytestring *`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 arguments `fix_imports`, `encoding`, and `errors` are provided for handling streams generated by Python 2 code; see the `pickle.loads` **documentation** for further information. **NEVER 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 topic 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 `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 the method `persistent_load`) and create your own persistence framework. However, this is an advanced topic 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)

with dbm.open('indexfilep', 'n') as dbm_out:
    for word, word_positions in word_pos.items():
        dbm_out[word] = pickle.dumps(word_positions, protocol=2)
```

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

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):

- 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 `__getnewargs__`, `__getstate__`, and `__setstate__` special methods 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__` attribute, `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 copied through such best-effort endeavors.

Pickling customization with the copyreg module

You can control how `pickle` serializes and deserializes objects of an arbitrary type by registering factory and reduction functions with the module `copyreg`. This is particularly, though not exclusively, useful when you define a type in a C-coded Python extension. The `copyreg` module supplies the functions listed in **Table 12-5**.

Table 12-5. Functions of the `copyreg` module

`constructor`	`constructor(`*`fcon`*`)` Adds *`fcon`* to the table of constructors, which lists all fac functions that `pickle` may call. *`fcon`* must be callable an normally a function.
`pickle`	`pickle(`*`type`*`,` *`fred`*`, fcon=`**`None`**`)` Registers function *`fred`* as the *reduction function* for typ where *`type`* must be a type object. To save an object *`o`* of *`type`*, the module `pickle` calls *`fred`*`(`*`o`*`)` and saves the res *`fred`*`(`*`o`*`)` must return a tuple `(fcon, t)` or `(fcon, t, d)`, *`fcon`* is a constructor and *`t`* is a tuple. To reload *`o`*, `pickl` *`o`*`=fcon(*`*`t`*`)`. Then, when *`fred`* also returns a *`d`*, `pickle` u restore *`o`*'s state (when *`o`* supplies `__setstate__`, *`o`*`.__setstate__(`*`d`*`)`; otherwise, *`o`*`.__dict__.update(`*`d`*`)`) described in the previous section. If *`fcon`* is not **`None`**, `pi` also calls `constructor(`*`fcon`*`)` to register *`fcon`* as a const `pickle` does not support pickling of code objects, but `ma` does. Here's how you could customize pickling to suppo objects by delegating the work to `marshal` thanks to `copy`

```
>>> import pickle, copyreg, marshal
>>> def marsh(x):
...     return marshal.loads, (marshal.dumps(x
...
>>> c=compile('2+2','','eval')
>>> copyreg.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

Be careful when using `marshal` in your code, as the preceding
ple does. `marshal`'s serialization isn't guaranteed to be stable a
versions, so using `marshal` means that programs written in oth
sions of Python may be unable to load the objects your progra
serialized.

The shelve Module

The `shelve` module orchestrates the modules `pickle`, `io`, and `dbm` (and its underlying modules for access to DBM-like archive files, as discussed in the following section) in order to provide a simple, lightweight persistence mechanism.

`shelve` supplies a function, `open`, that is polymorphic to `dbm.open`. The mapping *s* returned by `shelve.open` is less limited, however, than the mapping *a* returned by `dbm.open`. *a*'s keys and values must be strings.[3] *s*'s keys must also be strings, but *s*'s values may be of any pickleable types. `pickle` customizations (`copyreg`, `__getnewargs__`, `__getstate__`, and `__setstate__`) also apply to `shelve`, as `shelve` delegates serialization to `pickle`. Keys and values are stored as bytes. When strings are used, they are implicitly converted to the default encoding before being stored.

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 aren't stored back unless you assign the changed object back to the same index. For example:

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

You can finesse this issue by passing the named argument `writeback=True` when you call `shelve.open`, but this can 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)
with shelve.open('indexfiles','n') as sh_out:
    sh_out.update(word_pos)
```

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

```
import sys, shelve, linecache
with shelve.open('indexfiles') as sh_in:
    for word in sys.argv[1:]:
        if word not in sh_in:
            print(f'Word {word!r} not found in index file',
                                file=sys.stderr)
            continue
        places = sh_in[word]
        for fname, lineno in places:
            print(f'Word {word!r} occurs in line {lineno}'
                  f' of file {fname!r}:')
            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 `shelve` is higher level than the modules used in previous examples.

DBM Modules

DBM, a longtime Unix mainstay, is a family of libraries supporting data files containing pairs of bytestrings (*`key`*, *`data`*). DBM offers fast fetching and storing of the data given a key, a usage pattern known as *keyed access*. Keyed access, while nowhere near as powerful as the data access functionality of relational DBs, imposes less overhead, and it may suffice for some programs' needs. If DBM-like files are sufficient for your purposes, with this approach you can end up with a program that is smaller and faster than one using a relational DB.

DBM DATABASES ARE BYTES-ORIENTED

DBM databases require both keys and values to be bytes values. You will see in the example included later that the text input is explicitly encoded in UTF-8 before storage. Similarly, the inverse decoding must be performed when reading back the values.

DBM support in Python's standard library is organized in a clean and elegant way: the `dbm` package exposes two general functions, and within the same package live other modules supplying specific implementations.

BERKELEY DB INTERFACING

The bsddb module has been removed from the Python standard library. If you need to interface to a BSD DB archive, we recommend the excellent third-party package **bsddb3**.

The dbm Package

The dbm package provides the top-level functions described in **Table 12-6**.

Table 12-6. Functions of the dbm package

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, according to the rules shown in **Table 12-7**. 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 open"**.

Table 12-7. Flag values for dbm.open

Flag	Read-only?	If file exists:	If file does not exist:
'r'	Yes	Opens the file	Raises error
'w'	No	Opens the file	Raises error
'c'	No	Opens the file	Creates the file
'n'	No	Truncates the file	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 bytes as keys and values, and the only nonspecial 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 one way to ensure finalization, but the **`with`** statement, covered in **"The with Statement and Context Managers"**, is even better (you can use **`with`**, since *m* is a context manager).

`whichdb` `whichdb(`*filepath*`)`
Opens and reads the file specified by *filepath* to discover which `dbm` submodule created the file. `whichdb` returns **`None`** when the file does not exist or cannot be opened and read. It 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.dumb'`, or `'dbm.gdbm'`.

In addition to these two top-level functions, the `dbm` package contains specific modules, such as `ndbm`, `gnu`, and `dumb`, that provide various implementations of DBM functionality, which you normally access only via the these top-level functions. 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`. `dumb` has minimal DBM functionality and mediocre performance; its 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 no better submodule is available in 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 that must be readable in any Python installation. The `dumb` module supplies an `open` function polymorphic to `dbm`'s.

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 general approaches to the issue of encoding data as strings were covered in the opening section of this chapter, with the same example.) With this

simplification, a program to record word positions in files might be as follows:

```
import collections, fileinput, os, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = f'{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.encode('utf-8')] = sep2.join(
                        word_pos[word]
                ).encode('utf-8')
```

You 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:

```
import sys, os, dbm, linecache

sep = os.pathsep
sep2 = sep * 2
with dbm.open('indexfile') as dbm_in:
    for word in sys.argv[1:]:
        e_word = word.encode('utf-8')
        if e_word not in dbm_in:
            print(f'Word {word!r} not found in index file',
                  file=sys.stderr)
            continue
        places = dbm_in[e_word].decode('utf-8').split(sep2)
        for place in places:
            fname, lineno = place.split(sep)
            print(f'Word {word!r} occurs in line {lineno}'
                  f' of file {fname!r}:')
            print(linecache.getline(fname, int(lineno)), end='')
```

The Python Database API (DBAPI)

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, aka the DBAPI, as specified in **PEP 249**.

After importing any DBAPI-compliant module, you can 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 very ill-advisedly!) perform this task with Python's string formatting:

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

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, hampering SQL injection attacks.

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

```
c.execute('SELECT * FROM ATABLE WHERE 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 shown in **Table 12-8**.

Table 12-8. Possible values of the `paramstyle` attribute

`format`	The marker is `%s`, as in old-style string formatting (always w never use other type indicator letters, whatever the data's t query looks like:
	`c.execute('SELECT * FROM ATABLE WHERE AFIELD=%s',`
	`(some_value,))`

`named`	The marker is `:`*name*, and parameters are named. A query lo
	`c.execute('SELECT * FROM ATABLE WHERE AFIELD=:x',`
	`{'x':some_value})`

`numeric`	The marker is `:n`, giving the parameter's number, 1 and up. looks like:
	`c.execute('SELECT * FROM ATABLE WHERE AFIELD=:1',`
	`(some_value,))`

`pyformat`	The marker is `%(`*name*`)s`, and parameters are named. Alway never use other type indicator letters, whatever the data's t query looks like:
	`c.execute('SELECT * FROM ATABLE WHERE AFIELD=%(x)`
	`{'x':some_value})`

`qmark`	The marker is `?`. A query looks like: `c.execute('SELECT * FROM ATABLE WHERE AFIELD=?',`

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` or `%(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 `datetime` module (covered in **Chapter 13**), and strings or buffer types for BLOBs. The factory functions specified by the DBAPI are listed in **Table 12-9**. (The `*FromTicks` methods take an integer timestamp *`s`* representing the number of seconds since the epoch of module `time`, covered in **Chapter 13**.)

Table 12-9. DBAPI factory functions

`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 for integer timestamp *`s`*. 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 for integer timestamp *`s`*. 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 for integer timestamp *`s`*. For example, `TimestampFromTicks(time.time())` is the current date and time.

Type Description Attributes

A `Cursor` instance's `description` attribute describes the types and other characteristics of each column of the `SELECT` query you last executed 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 to
`dsn`	Name of the data source to use for the connection
`host`	Hostname on which the database is running
`password`	Password to use for the connection
`user`	Username to use 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 methods listed in **Table 12-10**.

Table 12-10. Methods of an instance x of class `Connection`

`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 the following section).
`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 provide DBAPI cursors even on RDBMSs that supply no real SQL cursors at all. An instance *c* of the class `Cursor` supplies many attributes and methods; the most frequently used ones are shown in **Table 12-11**.

Table 12-11. Commonly used attributes and methods of an instance c of class `Cursor`

`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 `SELECT` query or returned no usable description of the columns involved. A cursor's description is mostly usefu 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 differen schemas, including schemas that may not be fully know at the time you are writing your code.
`execute`	`c.execute(`*`statement`*`, parameters=`**`None`**`)` Executes a SQL *`statement`* string on the DB with the give `parameters`. `parameters` is a sequence when the module `paramstyle` is `'format'`, `'numeric'`, or `'qmark'`, and a mapping when `paramstyle` is `'named'` or `'pyformat'`. So 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 the given *`parameters`*. *`parameters`* is a sequence of sequences when the module's `paramstyle` is `'format'`, `'numeric'`, or `'qmark'`, and a sequence of mappings whe `paramstyle` is `'named'` or `'pyformat'`. For example, whe `paramstyle` is `'qmark'`, the statement:
	`c.executemany('UPDATE atable SET x=? ' 'WHERE y=?',(12,23),(23,34))`
	is equivalent to—but faster than—the two statements:
	`c.execute('UPDATE atable SET x=12 WHERE y=23')`
	`c.execute('UPDATE atable SET x=23 WHERE y=34')`

`fetchall`	`c.fetchall()` Returns all remaining rows from the last query as a sequence of tuples. Raises an exception if the last operat was not a `SELECT`.
`fetchmany`	`c.fetchmany(`*`n`*`)` Returns up to *`n`* remaining rows from the last query as a sequence of tuples. Raises an exception if the last operat was not a `SELECT`.

`fetchone`	`c.fetchone()` Returns the next row from the last query as a tuple. Rais an exception if the last operation was not a `SELECT`.
`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 changes so constantly, 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 and interface to very popular open source DBs:

ODBC modules
: 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, use **mxODBC**.

MySQL modules
: MySQL is a popular open source RDBMS, purchased by Oracle in 2010. Oracle's "official" DBAPI-compliant interface to it is **`mysql-connector-python`**. The MariaDB project also provides a DBAPI-compliant interface, **`mariadb`**, connecting to both MySQL and MariaDB (a GPL-licensed fork).

PostgreSQL modules
: PostgreSQL is another popular open source RDBMS. A widely used DBAPI-compliant interface to it is **`psycopg3`**, a rationalized rewrite and extension of the hallowed **`psycopg2`** package.

SQLite

SQLite is a C-coded library that implements a relational 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 smoother and more natural. We don't have the space in this book to cover every nook and cranny of these two powerful software systems; instead, we focus on the subset of functions that are most commonly used and most useful. For a greater level of detail, including examples and tips on best practices, see the documentation for **SQLite** and **`sqlite3`**, and Jay Kreibich's ***Using SQLite*** (O'Reilly).

Among others, the `sqlite3` package supplies the functions in **Table 12-12**.

Table 12-12. Some useful functions of the `sqlite3` module

`connect`	`connect(`*`filepath`*`, timeout=5.0, detect_types=0, isolation_level='', check_same_thread=`**`True`**`, factory=Connection, cached_statements=100, uri=`**`False`**`)` 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`*. If **`True`**, the `uri` argument activates SQLite's **URI functionality**, allowing a few extra options to be passed along with the filepath 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 following SQLite native types, converting them to the indicated Python types: • `BLOB`: Converted to `bytes` • `INTEGER`: Converted to `int` • `NULL`: Converted to **`None`** • `REAL`: Converted to `float` • `TEXT`: Depends on the `text_factory` attribute of the `Connection` instance, covered in **Table 12-13**; by default, `str` 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 table. To allow type name detection, pass as `detect_types` either of the constants `PARSE_COLNAMES` or `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 SQL `SELECT` 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'`, or `'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.
`connect` *(cont.)*	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, alas, common in multithreaded programming). If you're entirely confident about your threads' use of locks and other synchronization mechanisms, and you need to reuse a connection object among multiple threads, you can pass `check_same_thread=`**`False`**. `sqlite3` will then perform 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`*`)` 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`, and `bytes`. *`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`*`)` Registers *`callable`* as the *converter* from any value identified in SQL as being of type *`typename`* (see the description of the connect function's `detect_types` parameter 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 topic 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.

The sqlite3.Connection class

In addition to the methods common to all `Connection` classes of DBAPI-compliant modules, covered in **"Connection Objects"**, `sqlite3.Connection` supplies the methods and attributes in **Table 12-13**.

Table 12-13. Additional methods and attributes of the `sqlite3.Connection` class

`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 the `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 the schema and contents. Useful, for example, to persist an in-memory DB to 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 the following section, 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`, 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.

A `Connection` object can also be used as a context manager, to automatically commit database updates or roll back if an exception occurs; however, you will need to call `Connection.close()` explicitly to close the connection in this case.

The sqlite3.Row class

`sqlite3` also supplies the class `Row`. A `Row` object 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 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()
with connect:
    cursor.execute('CREATE TABLE IF NOT EXISTS Words '
                   '(Word TEXT, File TEXT, Line INT)')
    for line in fileinput.input():
        f, l = fileinput.filename(), fileinput.filelineno()
        cursor.executemany('INSERT INTO Words VALUES (:w, :f, :l)',
            [{'w':w, 'f':f, 'l':l} for w in line.split()])
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:]:
    cursor.execute('SELECT File, Line FROM Words '
                   'WHERE Word=?', [word])
    places = cursor.fetchall()
    if not places:
         print(f'Word {word!r} not found in index file',
               file=sys.stderr)
         continue
    for fname, lineno in places:
        print(f'Word {word!r} occurs in line {lineno}'
              f' of file {fname!r}:')
        print(linecache.getline(fname, lineno), end='')
connect.close()
```

1 In fact, “CSV” is something of a misnomer, since some dialects use tabs or other characters rather than commas as the field separator. It might be easier to think of them as “delimiter-separated values.”

2 Consider the third-party package `dill` if you need to extend `pickle` in this and other aspects.

3 `dbm` keys and values must be bytes; `shelve` will accept bytes or `str` and encode the strings transparently.