Advanced Python Supplement

John Strickler

Version 1.0, August 2025

Table of Contents

Chapter 1: Packaging

Objectives

- Create a pyproject.toml file
- Understand the types of wheels
- Generate an installable wheel
- Configure dependencies
- Configure executable scripts
- Distribute and deploy packages

Packaging overview

- Bundling project for distribution
- Uses build tools
- Needs metadata
- Extremely flexible

Packaging a project for distribution does not have to be complex. However, the tools are very flexible, and the amount of configuration can be overwhelming at first.

It boils down to these steps:

Create a virtual environment

While not absolutely necessary, creating a virtual environment for your project makes life easier, especially when it comes to dependency management.

Create a project layout

Arrange files and subfolders in the project folder. This can be *src* (recommended) or *flat*.

Specify metadata

Using `pyproject.toml`, specify metadata for your project. This metadata tells the build tools how and where to build and package your project.

Build the project

Use the build tools to create a *wheel* file. This wheel file can be distributed to developers.

Install the wheel file

Anyone who wants to install your project can use `pip` to install the project from the wheel file you built.

Upload the project to PyPI (Optional)

To share a project with everyone, upload it to the **PyPI** online repository.

Terminology

Here are some terms used in Python packaging. They will be explained in more detail in the following pages.

build backend

A module that does the actual creation of installable files (wheels). E.g., `setuptools` (>=61), `poetry-core`, `hatchling`, `pdm-backend`, `flit-core`.

build frontend

A user interface for a build backend. E.g., `pip`, `build`, `poetry`, `hatch`, `pdm`, `flit`

cookiecutter

A tool to generate the files and folders needed for a project.

dependency

A package needed by the current package.

editable install

An installation that is really a link to the development folder, so changes to the code are reflected whenever and wherever the package or module is imported.

package

Can refer to either a **distribution package** or an **import package**.

distribution package

A collection of code (usually a folder) to be bundled into a reusable (installable) "artifact" AKA *wheel* file. A distribution package can be used to install **modules**, **import packages**, **scripts**, or any combination of those items.

```
pip install distribution-package
```

import package

An installable module, usually implemented as a folder that contains one or more module files.

```
import import_package
```

PEP

Python Enhancement Proposal — a document that describes some aspect of Python. Similar to RFCs in the Internet world.

pip

The standard tool to install a Python package.

script

An executable Python script that is installed in the `scripts` (Windows) or `bin` (Mac/Linux) folder of your Python installation

toml

A file format similar to INI that is used for describing projects.

wheel

A file that contains everything needed to install a package

Project layout

A typical project has several parts: source code, documentation, tests, and metadata. These can be laid out in different ways, but most people either do a *flat* layout or a *src* layout.

A *flat* layout has the code in the top level of the project, and a *src* layout has code in a separate folder named `src`.

In the long run, the *src* layout seems to be the most readable, and that makes it the best practice.

Metadata goes in the `pyproject.toml` file.

Unit tests go in a folder named `tests`. Documentation, using a tool such as **Sphinx**, goes in a folder named `docs`.

You can add any other files or folders necessary for your project.

Typical layout

```
temperature
├──── README.md
├──── docs
│     ├──── Makefile
│     ├──── make.bat
│     └──── source
│         ├──── _static
│         ├──── _templates
│         ├──── conf.py
│         └──── index.rst
├──── pyproject.toml
├──── src
│     └──── temperature.py
└──── tests
    └──── test_temperature.py
```

the name of the project folder can be anything, but is typically the name of of the module or package you are creating.

The layouts on the following pages are not the only possibilities. You can combine them in whatever way works for your project.

Sample Project layouts

Module

```
MODULE_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml
 ├──── src
 │     └──── mymodule.py
 └──── tests
```

Code in mymodule.py will run as follows:

```
# import module (in a script)
# __name__ set to "mymodule"
import mymodule
from mymodule import MyClass, myfunction

# execute module (from command line)
# __name__ set to "__main__"
python mymodule.py
python -m mymodule
```


Module with callable scripts

```
MODULE_SCRIPTS_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml  # script names mapped to mymodule:function
 ├──── src
 │    └──── mymodule.py  # functions called by scripts
|       |─── function1()
|       |─── _function1_wrapper()
 └──── tests
```

In `pyproject.toml`

```
[project.scripts]
myscript='mymodule:_function1_wrapper'
```

Module can be run or imported normally as above, plus scripts defined in `pyproject.toml` can run directly from the command line

```
myscript
```

Package

```
PACKAGE_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│     └──── mypackage
│           ├──── mymodule1.py
│           └──── mymodule2.py
└──── tests
```

```
from mypackage import mymodule1
from mypackage.mymodule1 import MyClass, myfunction
```


Package with subpackages

```
PACKAGE_PROJECT
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│     └──── mypackage
│         └──── mysubpackage1
│             ├──── mymodule1.py
│             └──── mymodule2.py
│         └──── mysubpackage2
│             ├──── mymodule3.py
│             └──── mymodule4.py
└──── tests
```

```
from mypackage.subpackage1 import mymodule1
from mypackage.subpackage2.mymodule3 import MyClass, myfunction
```

Package callable with `python -m packagename`

```
PACKAGE_APP_PROJECT
 ├──── README.md
 ├──── docs
 ├──── pyproject.toml
 ├──── src
 │    └──── mypackage
 │        ├──── __main__.py.py  # module executed by  python  -m packagename
 │        ├──── mymodule1.py    # code to support main module
 │        └──── mymodule2.py    # code to support main module
 └──── tests
```

```
python -m mypackage
```


python -m *NAME*

- python -m module
  - executes entire module
- python -m package
  - executes module `__main__.py` in package
- includes `if __name__ == "__main__"` section

The command `python -m NAME` is designed to execute a module or package without knowing its exact location. It uses the module search mechanism to find and load the module. This searches the folders in `sys.path`.

If NAME is a package, it executes the module named `__main__.py` in the top level of the package.

If NAME is a module, it executes the entire module.

In both cases, the code in the `if __name__ == "__main__"` block (if any) is executed. If a module is imported, that code is not executed.

A list of standard modules that have a command line interface via `python -m` is here: https://docs.python.org/3/library/cmdline.html

Invoking Python

Assume the following code layout for the examples in the table

```
spam.py
ham
 ├──── __main__.py
 └──── toast.py
```

Invocation	Description	Example
*FILE and FOLDER specified as arguments to *python**		
`python FILE`	Run all code in `FILE`	`python spam.py` `python ham/toast.py`
`python FOLDER`	If `__main__.py` exists in `FOLDER`, run all code in `__main__.py`; otherwise, raise error	`python ham`
MODULE and PACKAGE found using sys.path		
`python -m MODULE`	Run all code in `MODULE`	`python -m spam`
`python -m PACKAGE`	Run all code in `PACKAGE.__init__.py` Run all code in `__main__`	`python -m ham`
`python -m PACKAGE.MODULE`	Run all code in `PACKAGE.__init__.py` Run all code in `PACKAGE.MODULE`	`python -m ham.toast`

Cookiecutter

- Creates standard layout
- Developed for Django
- Very flexible

cookiecutter is a utility written by Audrey and Roy Greenfeld to make it easy to replicate a standard setup for Django. However, it can be used create a layout for any type of project.

The `cookiecutter` command prompts you for information, then creates the project folder.

It uses a cookiecutter *template*, which is a folder, to create the new project. There are many templates on **github** to choose from, and you can easily create your own.

Syntax is

```
cookiecutter template-folder
```

The script copies the template layout (all folders and files) to a new folder which is the "slug" (short name) of your project. It inserts your project name in the appropriate places. It will do this in both file names and file contents.

There are two **cookiecutter** templates provided in the SETUP folder of the student files to generate the layouts on the previous pages:

- `cookiecutter-python-module`
- `cookiecutter-python-package`

The project layouts will be generated based on answers to the cookiecutter questions.

cookiecutter home page: https://github.com/audreyr/cookiecutter
cookiecutter docs: https://cookiecutter.readthedocs.io

Feel free to copy the cookiecutter templates and modify them for your own projects.

Another useful tool for generating Python projects is **PyScaffold**. Details at https://pyscaffold.org/en/stable/index.html.

cookiecutter-python-package/cookiecutter.json

```
{
    "package_name": "Package Name (can have spaces)",
    "package_slug": "{{ cookiecutter.package_name.lower().replace(' ','').replace('-
','_') }}",
    "package_description": "Short Description of the Package",
    "module_slug": "{{ cookiecutter.package_slug }}",
    "has_scripts": "n",
    "has_main": "n",
    "author_name": "Author Name",
    "author_email": "noone@nowhere.com",
    "author_url": "Author URL",
    "copyright_year": "2024",
    "readme_format": ["md", "rst"]
}
```

tree cookiecutter-python-package

```
/Users/jstrick/curr/courses/python/common/setup/cookiecutter-python-package
├── cookiecutter.json
├── hooks
│   ├── post_gen_project.py
│   └── pre_gen_project.py
└── {{cookiecutter.package_slug}}
    ├── README.{{cookiecutter.readme_format}}
    ├── docs
    │   ├── Makefile
    │   ├── make.bat
    │   └── source
    │       ├── _static
    │       ├── _templates
    │       ├── conf.py
    │       └── index.rst
    ├── pyproject.toml
    ├── src
    │   └── {{cookiecutter.package_slug}}
    │       ├── __init__.py
    │       ├── __main__.py
    │       └── {{cookiecutter.module_slug}}.py
    └── tests
        ├── __init__.py
        └── test_{{cookiecutter.module_slug}}.py

9 directories, 14 files
```


cookiecutter cookiecutter-python-package

```
[1/11] package_name (Package Name (can have spaces)): Log Processor
[2/11] package_slug (logprocessor): logproc
[3/11] package_description (Short Description of the Package): Process Log Files
[4/11] module_slug (logproc):
[5/11] has_scripts (n): y
[6/11] has_main (n): y
[7/11] author_name (Author Name): Sabrina Q. Programmer
[8/11] author_email (noone@nowhere.com): sabrinaq@gmail.com
[9/11] author_url (Author URL): https://www.sabrinaq.com
[10/11] copyright_year (2024):
[11/11] Select readme_format
1 - md
2 - rst
Choose from [1/2] (1): 1
```

tree logproc

```
logproc
├── README.md
├── docs
│   ├── Makefile
│   ├── make.bat
│   └── source
│       ├── _static
│       ├── _templates
│       ├── conf.py
│       └── index.rst
├── pyproject.toml
├── src
│   ├── logproc
│       ├── __init__.py
│       ├── __main__.py
│       └── logproc.py
└── tests
    ├── __init__.py
    └── test_logproc.py
```

Defining project metadata

- Create `pyproject.toml`
- Use `build` to build the package

The **modern** way to package a Python project is using the `pyproject.toml` config file. The specifications that support this are specified in **PEP 518** and **PEP 621**.

The **TOML** format is similar to `.ini` files, but adds some features.

The first part of the file is always required. It tells the `build` program what tools to use.

```
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"
```

Put all the project metadata that the build system will need to package and install your project after the `[build-system]` section.

```
[project]
name = "logproc"
version = "1.0.0"
authors = [
    { name="Author Name", email="sabrinaq@gmail.com" },
]
description = "Short Description of the Package"
readme = "README.rst"
requires-python = ">=3.0"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

dependencies = [
    'requests[security] < 3',
]
```

TOML value types arrays are similar to Python `list` and TOML tables (including inline) are similar to `dict`.

The rest of the file after the `[build-system]` section is not needed if you are also using `setup.cfg` and `setup.py`. However, best practice is to *not* use those legacy files, as all the data needed for building the package, installing it, and uploading it to **PyPI** can be contained in `pyproject.toml`, and can then be used by nearly any build *backend*.

Packages with scripts

- Provide utility scripts
- Run from command line
- Installed in `…/scripts` or `…/bin`
- Add config to `pyproject.toml`

It is easy to add one or more command-line scripts to your project. These scripts are created in the `scripts` (Windows) or `bin` (other OS) folders of your Python installation. While they require Python to be installed, they are run like any other command.

The scripts are based on functions in the module. Since the scripts are run from the CLI, they are not called with normal parameters. Instead, the functions access `sys.argv` for arguments, like any standalone Python script.

```
def _c2f_cli():
    """
    CLI utility script
    Called from command line as 'c2f'
    """
    cel = float(sys.argv[1])
    return c2f(cel)


def _f2c_cli():
    """
    CLI utility script
    Called from command line as 'f2c'
    """
    fahr = float(sys.argv[1])
    return f2c(fahr)
```

To configure scripts, add a section like the following to `pyproject.toml`. The names on the left are the installed script names. The values on the right are the module name and the function name, separated by a colon.

```
[project.scripts]
c2f = 'temperature:_c2f_cli'
f2c = 'temperature:_f2c_cli'
```

See the project `temperature_scripts` in EXAMPLES for details.

Editable installs

- Use `pip install -e package`
- Puts a link in library folder
- Allows testing as though module is installed

When using a `src` (or other name) folder for your codebase and `tests` for your test scripts, the tests need to find your package. While you could put the path to the `src` folder in PYTHONPATH, the best practice is to do an *editable install*.

This is an install that uses the path to your development folder. It achieves this by using a virtual environment. Then you can run your tests after making changes to your code, without having to reinstall the package.

From the top level folder of the project, type the following (you do not have to build the distribution for this step).

```
pip install -e .
```

Now the project is installed and is available to import or run like any other installed module.

Running unit tests

- Use editable install
- Just use `pytest` or `pytest -v`

To run the tests that you have created in the `tests` folder, just run `pytest` or `pytest -v` (verbose) in the top level folder of the project. Because the project was installed with an editable install, tests can import the module or package normally.

Example

```
$ pytest -v
=========================================================== test session starts
============================================================
platform darwin -- Python 3.9.17, pytest-7.1.2, pluggy-1.0.0 -- /Users/jstrick/opt/miniconda3/bin/python
cachedir: .pytest_cache
PyQt5 5.15.7 -- Qt runtime 5.15.2 -- Qt compiled 5.15.2
hypothesis profile 'default' ->
database=DirectoryBasedExampleDatabase('/Users/jstrick/curr/courses/python/common/examples/temperature/.hypothesis/examples')
rootdir: /Users/jstrick/curr/courses/python/common/examples/temperature
plugins: anyio-3.6.1, qt-4.1.0, remotedata-0.3.3, assert-utils-0.3.1, lambda-2.1.0, astropy-header-0.2.1, fixture-order-
0.1.4, common-subject-1.0.6, mock-3.8.2, typeguard-2.13.3, astropy-0.10.0, filter-subpackage-0.1.1, hypothesis-6.54.3,
openfiles-0.5.0, django-4.5.2, doctestplus-0.12.0, cov-3.0.0, arraydiff-0.5.0
collected 8 items

tests/test_temperature.py::test_c2f[100-212] PASSED
[ 12%]
tests/test_temperature.py::test_c2f[0-32] PASSED
[ 25%]
tests/test_temperature.py::test_c2f[37-98.6] PASSED
[ 37%]
tests/test_temperature.py::test_c2f[-40--40] PASSED
[ 50%]
tests/test_temperature.py::test_f2c[212-100] PASSED
[ 62%]
tests/test_temperature.py::test_f2c[32-0] PASSED
[ 75%]
tests/test_temperature.py::test_f2c[98.6-37] PASSED
[ 87%]
tests/test_temperature.py::test_f2c[-40--40] PASSED
[100%]

============================================================ 8 passed in 0.20s
=============================================================
```


Wheels

- 3 kinds of wheels
  - Universal wheels (pure Python; python 2 *and* 3 compatible
  - Pure Python wheels (pure Python; Python 2 *or* 3 compatible
  - Platform wheels (Platform-specific; binary)

A wheel is prebuilt distribution. Wheels can be installed with pip.

A *Universal wheel* is a pure Python package (no extensions) that can be installed on either Python 2 or Python 3. It has to have been carefully written that way.

A *Pure Python wheel* is a pure Python package that is specific to one version of Python (either 2 or 3). It can only be installed by a matching version of pip.

A *Platform wheel* is a package that has extensions, and thus is platform-specific.

Build systems automatically create the correct wheel type.

Building distributions

- `python -m build`
- Creates `dist` folder
- Binary distribution
  - `package-version.whl`
- Source distribution
  - `package-version.tar.gz`

To build the project, use

```
python -m build
```

This will create the wheel file (binary distribution) and a gzipped tar file (source distribution) in a folder named `dist`.

python -m build

```
* Creating virtualenv isolated environment...
* Installing packages in isolated environment... (setuptools>=61.0)
* Getting build dependencies for sdist...
running egg_info
writing src/temperature.egg-info/PKG-INFO
writing dependency_links to src/temperature.egg-info/dependency_links.txt
writing top-level names to src/temperature.egg-info/top_level.txt
reading manifest file 'src/temperature.egg-info/SOURCES.txt'
writing manifest file 'src/temperature.egg-info/SOURCES.txt'
* Building sdist...
```

... about 70 lines of output ...

```
running install_scripts
creating build/bdist.macosx-10.9-universal2/wheel/temperature-1.0.0.dist-info/WHEEL
creating '/Users/jstrick/curr/courses/python/common/examples/temperature/dist/.tmp-
_hr_mnd6/temperature-1.0.0-py3-none-any.whl' and adding 'build/bdist.macosx-10.9-
universal2/wheel' to it
adding 'temperature.py'
adding 'temperature-1.0.0.dist-info/METADATA'
adding 'temperature-1.0.0.dist-info/WHEEL'
adding 'temperature-1.0.0.dist-info/top_level.txt'
adding 'temperature-1.0.0.dist-info/RECORD'
removing build/bdist.macosx-10.9-universal2/wheel
Successfully built temperature-1.0.0.tar.gz and temperature-1.0.0-py3-none-any.whl
```

Installing a package

- Use `pip`
  - many options
  - can install just for user

A wheel makes installing packages simple. You can just use

```
pip install package.whl
```

This will install the package in the standard location for the current version of Python.

If you do not have permission to install modules in the standard location, you can do a user install, which installs modules under your home folder.

```
pip install --user package.whl
```


For more information

Python Packaging User Guide

https://packaging.python.org/en/latest/

Distributing Python Modules

https://docs.python.org/3/distributing/index.html

setuptools Quickstart

https://setuptools.pypa.io/en/latest/userguide/quickstart.html

Thoughts on the Python packaging ecosystem

https://pradyunsg.me/blog/2023/01/21/thoughts-on-python-packaging/

THE BASICS OF PYTHON PACKAGING IN EARLY 2023

https://drivendata.co/blog/python-packaging-2023

Structuring Your Project (from The Hitchhiker's Guide to Python)

https://docs.python-guide.org/writing/structure/

Chapter 1 Exercises

Exercise 1-1 (carddeck/*)

Step 1

Create a distributable module named `carddeck` from the `carddeck.py` and `card.py` modules in the root folder of the student files.

HINT: To do it the easy way, use the `cookiecutter-python-module` template. Add the two source files to the `src` folder. (remove any existing sample Python scripts in `src`).

> To do it the "hard" way, create the project layout by hand, create the `pyproject.toml` file, etc.

Step 2

Build a distribution (wheel file).

Step 3

Install the wheel file with `pip`. (The cookiecutter template automatically does an editable install)

Step 4

Then import the new module and create an instance of the `CardDeck` class. Shuffle the cards, and deal out all 52 cards.

Chapter 2: Database Access

Objectives

- Understand the Python DB API architecture
- Connect to a database
- Execute simple and parameterized queries
- Fetch single and multiple row results
- Execute non-query statements
- Get metadata about a query
- Start transactions and commit or rollback as needed

The DB API

- Most popular Python DB interface
- Specification, not abstract class
- Many modules for different DB implementations
- Hides actual DBMS implementation

To make database programming simpler and more consistent, Python provides the DB API. This is an API to standardize working with databases. When a package is written to access a database, it is written to conform to the API, and thus programmers do not have to learn a new set of methods and functions for each different database architecture.

DB API objects and methods

```
conn = package.connect(connection-arguments)
cursor = conn.cursor()
num_lines = cursor.execute(query)
num_lines = cursor.execute(query-with-placeholders, iterable)
num_lines = cursor.executemany(query-with-placeholders, iterable)
all_rows = cursor.fetchall()
some_rows = cursor.fetchmany(n)
one_row = cursor.fetchone()
conn.commit()
conn.rollback()
```

Table 1. Available Interfaces (using Python DB API-2.0)

Database	Python package
Firebird (and Interbase)	`KInterbasDB`
Databricks	`databricks-sql-connector`
Elasticsearch	`elasticsearch-dbapi`
Excel	`excel-dbapi`
Google BigQuery	`google-cloud-bigquery`
IBM DB2	`ibm-db`
Informix	`informixdb`
Ingres	`ingmod`
Microsoft SQL Server	`pymssql`
MySQL	`pymysql`
ODBC	`pyodbc`
Oracle	`oracledb` `cx_oracle (obsolete as of 2022)`
PostgreSQL	`psycopg` (previously `psycopg2`)
SAP DB (also known as "MaxDB")	`sapdbapi`
SQLite	`sqlite3`
Sybase	`Sybase`

This list is not comprehensive, and there are additional interfaces available for some of the listed DBMSs.

Connecting to a Server

- Import appropriate library
- Use connect() to get a connection object
- Specify host, database, username, password, etc.

To connect to a database server, import the package for the specific database. Use the package's `connect()` method to get a connection object. The `connect()` function requires the information needed to access the database, which may include the host, initial database, username, or password.

Argument names for the `connect()` method are not consistent across packages. Most `connect()` methods use individual arguments, such as ***host***, ***database***, etc., but some use a single string argument.

When finished with the connection, call the `close()` method on the connection object. Many database modules support the context manager `with` statement, and will automatically close the database when the `with` block is exited. Check the documentation to see how this is implemented for a particular database.

Example

```
import pymysql

conn = pymysql.connect (host = "dbserver",
                           user = "adeveloper",
                           passwd = "s3cr3t",
                           db = "samples")
# Interact with database here ...
conn.close()
```

```
import sqlite3

with sqlite3.connect('sample.db') as conn:
    # Interact with database here ...
```


`connect()` examples

PostgreSQL using `pyscopg`

```
conn = psycopg.connect(
    host="localhost",
    dbname="postgres",
    user="adeveloper",
    password='$3cr3t',
)
```

Oracle using `oracledb`

```
conn = oracledb.connect(
    user='adeveloper',
    password='$3cr3t',
    dsn="dbhost.example.com/orclpdb"
)
```

```
conn = oracledb.connect(
    user='adeveloper',
    password='$3cr3t',
    host="localhost",
    port=1521,
    servicename="orclpdb",
)
```

SqlLite3 using `sqlite3`

```
conn = sqlite3.connect('testdb')   # on-disk database(single file)
```

```
conn = sqlite3.connect(':memory:')  # in-memory database
```

Microsoft SQLServer using `pymssql`

```
conn = pymssql.connect(
    server=r"HOST\INSTANCE",
    user="adeveloper",  # SQLServer user, not Windows user
    password="s3cr3t",  # SQLServer password
    database="testdb")
```

```
conn = pymssql.connect (
    dsn="DSN",
)
```

MySQL/MariaDB using `pymysql`

```
conn = pymysql.connect (
    host="localhost",
    user="adeveloper",
    passwd="$3cr3t",
    db="testdb",
)
```

Any ODBC-compliant DB using `pyodbc`

```
conn = pyodbc.connect('''
   DRIVER={SQL Server};
   SERVER=localhost;
   DATABASE=testdb;
   UID=adeveloper;
   PWD=$3cr3t
''')
```

```
conn = pyodbc.connect('DSN=testdsn;PWD=$3cr3t')
```

> `pyodbc.connect()` has one (string) parameter, not multiple parameters

IBM DB2 using `ibm-db`

```
import ibm_db_dbi as db2
conn = db2.connect(

"DATABASE=testdb;HOSTNAME=localhost;PORT=50000;PROTOCOL=TCPIP;UID=db2inst1;PWD=scripts;",
    "",
    ""
)
```


Creating a Cursor

- Cursor can execute SQL statements
- Create with `cursor()` method
- Multiple cursors available
  - Standard cursor
    - Returns rows as tuples
  - Other cursors
    - Return dictionary
    - Return hybrid dictionary/list
    - Leave data on server

Once you have a connection object, call `cursor()` to create a cursor object. A cursor is an object that can execute SQL code and fetch results. Each connection may have one or more active cursors.

The default cursor for most packages returns each row as a tuple of values. There are optional cursors that can return data in different formats, or that control whether data is stored on the client or the server.

> The examples in this chapter are implemented with SQLite, since the `sqlite3` module is part of the standard library. Most of the examples are also implemented for PostgreSQL and MySQL. See `db_mysql_*.py` and `db_postgres_*.py` in EXAMPLES.

Querying data

- ***cursor***.execute(***query***)
  - Gets all data from query
  - Returns # rows in result set
- Cursor is iterator over result set
- Return rows as tuples of values

Once you have a cursor, you can use it to execute queries via the `execute()` method. The first argument to `execute()` is a string containing one SQL statement.

For queries, `execute()` returns the number of rows in the result set. For non-query statements, `execute()` returns the number of rows affected by the operation.

The cursor object is an iterator over the rows in the result set.

For standard cursors, all data is transferred from the database server to your program's memory when `execute()` is called.

> For `sqlite3`, `execute()` returns the cursor object, so you can say `execute(QUERY-STATEMENT).fetchall()`.

Example

db_sqlite_basics.py

```
import sqlite3

# conn = sqlite3.Connection(...)
with sqlite3.connect("../DATA/presidents.db") as conn:  # connect to the database

    s3_cursor = conn.cursor()  # get a cursor object

    # select specified columns from all presidents
    s3_cursor.execute('''
        select termnum, firstname, lastname, party
        from presidents
    ''')  # execute a SQL statement

    for term, firstname, lastname, party in s3_cursor:
        print(f"{term:2d} {firstname:25} {lastname:20} {party}")
    print()
```

db_sqlite_basics.py

```
 1 George                   Washington          no party
 2 John                     Adams               Federalist
 3 Thomas                   Jefferson           Democratic - Republican
 4 James                    Madison             Democratic - Republican
 5 James                    Monroe              Democratic - Republican
 6 John Quincy              Adams               Democratic - Republican
 7 Andrew                   Jackson             Democratic
 8 Martin                   Van Buren           Democratic
 9 William Henry            Harrison            Whig
10 John                     Tyler               Whig
```

…

```
39 James Earl 'Jimmy'       Carter              Democratic
40 Ronald Wilson            Reagan              Republican
41 George Herbert Walker    Bush                Republican
42 William Jefferson 'Bill' Clinton             Democratic
43 George Walker            Bush                Republican
44 Barack Hussein           Obama               Democratic
45 Donald John              Trump               Republican
46 Joseph Robinette         Biden               Democratic
47 Donald John              Trump               Republican
```


Fetch methods

- Alternate way to get rows
  - `fetchone()` get next row
  - `fetchmany()` get N rows
  - `fetchall()` get all remaining rows

Cursors provide three methods for returning query results. These can be used instead of iterating over the cursor.

`fetchone()` returns the next available row from the query results.

`fetchmany(n)` returns up to n rows. This is useful when the query returns a large number of rows.

`fetchall()` returns a tuple of all rows.

For all three methods, each row is returned as a tuple of values.

Example

db_sqlite_fetch.py

```
import sqlite3

# conn = sqlite3.Connection(...)
with sqlite3.connect("../DATA/presidents.db") as conn:  # connect to the database

    s3_cursor = conn.cursor()  # get a cursor object

    # select specified columns from all presidents
    s3_cursor.execute('''
        select termnum, firstname, lastname, party
        from presidents
    ''')  # execute a SQL statement

    row = s3_cursor.fetchone()
    print(f"{row = }")
    print('-' * 60)

    for row in s3_cursor.fetchmany(5):
        print(row)
    print('-' * 60)
```

```
    for row in s3_cursor.fetchall():
        print(row)
    print()
```

db_sqlite_fetch.py

```
row = (1, 'George', 'Washington', 'no party')
------------------------------------------------------------
(2, 'John', 'Adams', 'Federalist')
(3, 'Thomas', 'Jefferson', 'Democratic - Republican')
(4, 'James', 'Madison', 'Democratic - Republican')
(5, 'James', 'Monroe', 'Democratic - Republican')
(6, 'John Quincy', 'Adams', 'Democratic - Republican')
------------------------------------------------------------
(7, 'Andrew', 'Jackson', 'Democratic')
(8, 'Martin', 'Van Buren', 'Democratic')
```

...

```
(39, "James Earl 'Jimmy'", 'Carter', 'Democratic')
(40, 'Ronald Wilson', 'Reagan', 'Republican')
(41, 'George Herbert Walker', 'Bush', 'Republican')
(42, "William Jefferson 'Bill'", 'Clinton', 'Democratic')
(43, 'George Walker', 'Bush', 'Republican')
(44, 'Barack Hussein', 'Obama', 'Democratic')
(45, 'Donald John', 'Trump', 'Republican')
(46, 'Joseph Robinette', 'Biden', 'Democratic')
(47, 'Donald John', 'Trump', 'Republican')
```


Non-query statements

- Update database
- Returns count of rows affected
- Changes must be committed

The `execute()`method is also used to execute non-query statements, such as **CREATE**, **ALTER**, **UPDATE**, and **DROP**.

As with queries, the first argument is a string containing one SQL statement.

For most DB packages, `execute()` returns the number of rows affected.

To make changes to the database permanent, changes must be commited with `CONNECTION.commit()`.

Example

db_sqlite_add_row.py

```
from datetime import date
import sqlite3

with sqlite3.connect("../DATA/presidents.db") as s3conn:  # connect to database

    sql_insert = """
    insert into presidents
    (lastname, firstname, termstart, termend, birthplace, birthstate, birthdate,
deathdate,  party)
    values ('Ramirez', 'Mary', '2025-01-20', null, 'Topeka',
    'Kansas', '1968-09-22', null, 'Independent')
    """

    cursor = s3conn.cursor()

    try:
        cursor.execute(sql_insert)
    except (sqlite3.OperationalError, sqlite3.DatabaseError, sqlite3.DataError) as err:
        print(err)
        s3conn.rollback()
    else:
        s3conn.commit()
    finally:
        cursor.close()
```


Example

db_sqlite_delete_row.py

```
from datetime import date
import sqlite3


with sqlite3.connect("../DATA/presidents.db") as conn:  # connect to DB

    sql_delete = """
    delete from presidents
    where TERMNUM = 48
    """

    cursor = conn.cursor()  # get a cursor

    try:
        cursor.execute(sql_delete)
    except (sqlite3.DatabaseError, sqlite3.OperationalError, sqlite3.DataError) as err:
        print(err)
        conn.rollback()
    else:
        conn.commit()
    finally:
        cursor.close()
```

SQL Injection

- Hijacks SQL code
- Result of string formatting
- Always use parameterized statements

One kind of vulnerability in SQL code is called *SQL injection*. This happens when using string formatting and raw user input to build SQL statements. An attacker can embed malicious SQL commands in input data.

Since the programmer is generating the SQL code as a string, there is no way to check for malicious SQL code. It is best practice to use parameterized statements, which prevents any user input from being *injected* into the SQL statement.

see http://www.xkcd.com/327 for a well-known web comic on this subject.

Example

db_sql_injection.py

```
#
good_input = 'Google'
malicious_input = "'; drop table customers; -- "  # input would come from a web form, for
instance

naive_format = "select * from customers where company_name = '{}' and company_id != 0"

good_query = naive_format.format(good_input)  # string formatting naively adds the user
input to a field, expecting only a customer name
malicious_query = naive_format.format(malicious_input)  # string formatting naively adds
the user input to a field, expecting only a customer name

print("Good query:")
print(good_query)  # non-malicious input works fine
print()

print("Bad query:")
print(malicious_query)  # query now drops a table ('--' is SQL comment)
```

db_sql_injection.py

```
Good query:
select * from customers where company_name = 'Google' and company_id != 0

Bad query:
select * from customers where company_name = ''; drop table customers; -- ' and
company_id != 0
```

Parameterized Statements

- Prevent SQL injection
- More efficient updates
- Use placeholders in query
    - Placeholders vary by DB
- Pass iterable of parameters
- Use cursor.execute() or cursor.executemany()

When using parameterized statements, you specify a SQL statement as usual, but use placeholders for the user-supplied data. Add an argument to `.execute()` containing the values for the placeholders. Placeholders can be either positional or named. Some DBMSs support both.

Parameterized statements not only protect agains SQL injection, but they also are usually more efficient. The DBMS only parses the SQL statement once, then fills in data on each call to `.execute()`.

All SQL statements may be parameterized, including queries.

Different database modules use different placeholders. To see what kind of placeholder a module uses, check `_package_.paramstyle`.

Positional parameters

Positional parameters are filled in from an iterable. Values are filled in from left to right.

psycopg

```
cursor.execute("insert into users (name, id) values (%s, %s)", ["Bob", 123])
```

sqlite3

```
cursor.execute("insert into users (name, id) values (?, ?)", ["Bob", 123])
```

oracledb

```
cursor.execute("insert into users (name, id) values (:1, :2)", ["Bob", 123])
```


Named parameters

Named parameters are filled in from a dictionary. They have the advantage of not needing to be in a particular order.

oracledb supports specifying parameter values as named arguments to `.execute()`.

psycopg

```
cursor.execute(
    "insert into users (name, id) values (%(name)s, %(id)s),
    {'name':"Bob", 'id':123}
)
```

sqlite3

```
cursor.execute(
    "insert into users (name, id) values (:name, :id),
    {'name':"Bob", 'id':123}
)
```

oracledb

```
cursor.execute(
    "insert into users (name, id) values (:name, :id),
    {'name':"Bob", 'id':123}
)

cursor.execute(
    "insert into users (name, id) values (:name, :id),
    name="Bob", id=123
)
```

executemany()

`executemany()` takes a non-query statement plus an iterable of parameter iterables (such as a list of tuples). It will execute the query once for each item in the outer iterable.

`executemany()` may only be used for DML statements — another name for non-query statements.

```
users = [
    ('Bob', 123),
    ('Raul', 88),
    ('Lakshmi', 17),
]

cursor.executemany(
    "insert into users (name, id) values (%s, %s),
    users
)
```

```
users = [
    {'name: 'Bob', 'id': 123},
    {'name: 'Raul', 'id': 88},
    {'name: 'Lakshmi', 'id': 17},
]

cursor.executemany(
    "insert into users (name, id) values (%(name)s, %(id)s),
    users
)
```


Table 2. Placeholders for SQL Parameters

Python package	Positional Placeholder	Named Placeholder
`pymysql`	`%s`	`%(param_name)s`
`oracledb`	`:1`, `:2`, etc or `:param_name`	`:param_name`
`pyodbc`	`?`	None
`pymssql`	`%s` or `%d`	`%(param_name)s` *not* `%(param_name)d`
`Psycopg`	`%s`	`%(param_name)s`
`SQLite`	`?`	`:param_name`

Table 3. Placeholder types

Type	Description	Example
qmark	Question mark	`WHERE name=?`
numeric	Numeric, positional style	`WHERE name=:1`
named	Named style	`WHERE name=:name`
format	ANSI C printf format codes	`WHERE name=%s`
pyformat	Python extended format codes	`WHERE name=%(name)s`

`_package_.paramstyle` contains the parameter type for the given package.

Example

db_sqlite_parameterized.py

```
import sqlite3

# list of one-element tuples -- each tuple provides parameters for
# a SQL statement.
TERMS_TO_UPDATE = [(1,), (5,), (19,), (22,), (36,)]

PARTY_UPDATE = '''
update presidents
set party = "SURPRISE!"
where termnum = ?
'''  # ? is SQLite3 placeholder for SQL statement parameter; different DBMSs use
different placeholders

PARTY_QUERY = """
select termnum, firstname, lastname, party
from presidents
where termnum = ?
"""

with sqlite3.connect("../DATA/presidents.db") as s3conn:
    s3cursor = s3conn.cursor()
    # second argument to executemany() is iterable of parameters
    # each parameter is an iterable of values to fill in the placeholdelrs
    s3cursor.executemany(PARTY_UPDATE, TERMS_TO_UPDATE)

    for param in TERMS_TO_UPDATE:
        term = param[0]
        s3cursor.execute(PARTY_QUERY, (term,))
        print(s3cursor.fetchone())
```

db_sqlite_parameterized.py

```
(1, 'George', 'Washington', 'SURPRISE!')
(5, 'James', 'Monroe', 'SURPRISE!')
(19, 'Rutherford Birchard', 'Hayes', 'SURPRISE!')
(22, 'Grover', 'Cleveland', 'SURPRISE!')
(36, 'Lyndon Baines', 'Johnson', 'SURPRISE!')
```


Example

db_sqlite_restore_parties.py

```
import sqlite3

# be sure to put values in correct order for placeholders
RESTORE_DATA = [
    ('no party', 1),
    ('Democratic - Republican', 5),
    ('Republican', 19),
    ('Democratic', 22),
    ('Democratic', 36),
]

PARTY_UPDATE = '''
update presidents
set party = ?
where termnum = ?
'''  # ? is SQLite3 placeholder; other DBMSs may use other placeholders

PARTY_QUERY = """
select termnum, firstname, lastname, party
from presidents
where termnum = ?
"""

with sqlite3.connect("../DATA/presidents.db") as s3conn:
    s3cursor = s3conn.cursor()
    s3cursor.executemany(PARTY_UPDATE, RESTORE_DATA)
    s3conn.commit()

    # _ is throwaway variable
    for _, termnum in RESTORE_DATA:
        s3cursor.execute(PARTY_QUERY, [termnum])
        print(s3cursor.fetchone())
```

db_sqlite_restore_parties.py

```
(1, 'George', 'Washington', 'no party')
(5, 'James', 'Monroe', 'Democratic - Republican')
(19, 'Rutherford Birchard', 'Hayes', 'Republican')
(22, 'Grover', 'Cleveland', 'Democratic')
(36, 'Lyndon Baines', 'Johnson', 'Democratic')
```

Example

db_sqlite_bulk_insert.py

source

```
import sqlite3
import os
import csv

DATA_FILE = '../DATA/fruit_data.csv'

DB_NAME = 'fruits.db'
DB_TABLE = 'fruits'

SQL_CREATE_TABLE = f"""
create table {DB_TABLE} (
    id integer primary key,
    name varchar(30),
    unit varchar(30),
    unitprice decimal(6, 2)
)
"""  # SQL statement to create table

SQL_INSERT_ROW = f'''
insert into {DB_TABLE} (name, unit, unitprice) values (?, ?, ?)
'''  # parameterized SQL statement to insert one record

SQL_SELECT_ALL = f"""
select name, unit, unitprice from {DB_TABLE}
"""

def main():
    """
    Program entry point.

    :return: None
    """
    conn, cursor = get_connection()
    create_database(cursor)
    populate_database(conn, cursor)
    read_database(cursor)

    cursor.close()
    conn.close()
```



```
def get_connection():
    """
    Get a connection to the PRODUCE database

    :return: SQLite3 connection object.
    """
    if os.path.exists(DB_NAME):
        os.remove(DB_NAME)  # remove database if it exists

    conn = sqlite3.connect(DB_NAME)  # connect to (new) database
    cursor = conn.cursor()
    return conn, cursor


def create_database(cursor):
    """
    Create the fruit table

    :param conn: The database connection
    :return: None
    """
    cursor.execute(SQL_CREATE_TABLE)  # run SQL to create table


def populate_database(conn, cursor):
    """
    Add rows to the fruit table

    :param conn: The database connection
    :return: None
    """
    with open(DATA_FILE) as file_in:
        fruit_data = csv.reader(file_in, quoting=csv.QUOTE_NONNUMERIC)

        for row in fruit_data:
            try:
                # add a row to the table
                cursor.execute(SQL_INSERT_ROW, row)
            except sqlite3.DatabaseError as err:
                print(err)
                conn.rollback()
            else:
                # commit the inserts; without this, no data would be saved
                conn.commit()

def read_database(cursor):
    cursor.execute(SQL_SELECT_ALL)
    for name, unit, unitprice in cursor.fetchall():
```

```
        print(f'{name:12s} {unitprice:5.2f}/{unit}')


if __name__ == '__main__':
    main()
```

db_sqlite_bulk_insert.py

```
pomegranate   0.99/each
cherry        2.25/pound
apricot       3.49/pound
date          1.20/pound
apple         0.55/pound
lemon         0.69/each
kiwi          0.88/each
orange        0.49/each
lime          0.49/each
watermelon    4.50/each
guava         2.88/pound
papaya        1.79/pound
fig           2.29/pound
pear          1.10/pound
banana        0.65/pound
```

Example

db_sqlite_execute_many.py

```
import sqlite3
import os
import csv

DATA_FILE = '../DATA/fruit_data.csv'

DB_NAME = 'fruits.db'
DB_TABLE = 'fruits'

SQL_CREATE_TABLE = f"""
create table {DB_TABLE} (
    id integer primary key,
    name varchar(30),
    unit varchar(30),
    unitprice decimal(6, 2)
)
"""  # SQL statement to create table

SQL_INSERT_ROW = f'''
insert into {DB_TABLE} (name, unit, unitprice) values (?, ?, ?)
'''  # parameterized SQL statement to insert one record

SQL_SELECT_ALL = f"""
select name, unit, unitprice from {DB_TABLE}
"""


def main():
    """
    Program entry point.

    :return: None
    """
    conn, cursor = get_connection()
    create_database(cursor)
    populate_database(conn, cursor)

    # read database to confirm inserts
    read_database(cursor)

    cursor.close()
    conn.close()
```

```python
def get_connection():
    """
    Get a connection to the PRODUCE database

    :return: SQLite3 connection object.
    """
    if os.path.exists(DB_NAME):
        os.remove(DB_NAME)  # remove existing database if it exists

    conn = sqlite3.connect(DB_NAME)  # connect to (new) database
    cursor = conn.cursor()
    return conn, cursor


def create_database(cursor):
    """
    Create the fruit table

    :param conn: The database connection
    :return: None
    """
    cursor.execute(SQL_CREATE_TABLE)  # run SQL to create table


def populate_database(conn, cursor):
    """
    Add rows to the fruit table

    :param conn: The database connection
    :return: None
    """
    with open(DATA_FILE) as file_in:
        fruit_data = csv.reader(file_in, quoting=csv.QUOTE_NONNUMERIC)

        try:
            # iterate over rows of input
            # and add each row to database
            # fruit data is iterable of iterables
            #  OR, iterable of dictionaries
            cursor.executemany(SQL_INSERT_ROW, fruit_data)
        except sqlite3.DatabaseError as err:
            print(err)
            conn.rollback()
        else:
            # Commit the inserts. Without this, no data
            # would be saved
            conn.commit()
```



```
def read_database(cursor):
    cursor.execute(SQL_SELECT_ALL)
    for name, unit, unitprice in cursor.fetchall():
        print(f'{name:12s} {unitprice:5.2f}/{unit}')

if __name__ == '__main__':
    main()
```

db_sqlite_execute_many.py

```

pomegranate   0.99/each
cherry        2.25/pound
apricot       3.49/pound
date          1.20/pound
apple         0.55/pound
lemon         0.69/each
kiwi          0.88/each
orange        0.49/each
lime          0.49/each
watermelon    4.50/each
guava         2.88/pound
papaya        1.79/pound
fig           2.29/pound
pear          1.10/pound
banana        0.65/pound
```

Metadata

- ***cursor*.description** returns tuple of tuples
- Fields
  - name
  - type_code
  - display_size
  - internal_size
  - precision
  - scale
  - null_ok

Once a query has been executed, the cursor's `description` attribute is a tuple with metadata about the columns in the query. It contains one tuple for each column in the query, containing 7 values describing the column.

For instance, to get the names of the columns, you could say

```
names = [d[0] for d in cursor.description]
```

For non-query statements, `CURSOR.description` returns `None`.

The names are based on the query (with possible aliases), and not necessarily on the names in the table.

Not all of the fields will necessarily be populated. For instane, `sqlite3` only provides column names.

Example

db_sqlite_metadata.py

```
"""
    Provide metadata (tables and column names) for a Sqlite3 database
"""
from pprint import pprint
import sqlite3

DB_NAME = "../DATA/presidents.db"
TABLE_QUERY = '''select * from presidents where 1 = 2'''

def main():
    cursor = connect_to_db(DB_NAME)
    show_metadata(cursor)

def connect_to_db(database_file):
    with sqlite3.connect(database_file) as s3conn:
        return s3conn.cursor()

def show_metadata(cursor):
    cursor.execute(TABLE_QUERY)
    pprint(cursor.description)
    print()
    column_names = [column_data[0] for column_data in cursor.description]
    print(f"{column_names = }")

if __name__ == '__main__':
    main()
```

db_sqlite_metadata.py

```
(('termnum', None, None, None, None, None, None),
 ('lastname', None, None, None, None, None, None),
 ('firstname', None, None, None, None, None, None),
 ('termstart', None, None, None, None, None, None),
 ('termend', None, None, None, None, None, None),
 ('birthplace', None, None, None, None, None, None),
 ('birthstate', None, None, None, None, None, None),
 ('birthdate', None, None, None, None, None, None),
 ('deathdate', None, None, None, None, None, None),
 ('party', None, None, None, None, None, None))

column_names = ['termnum', 'lastname', 'firstname', 'termstart', 'termend', 'birthplace',
'birthstate', 'birthdate', 'deathdate', 'party']
```


Dictionary Cursors

- Indexed by column name
- Not standardized in the DB API

Some DB packages provide dictionary cursors, which return a dictionary for each row, instead of a tuple. The keys are the names of the columns, so columns can be accessed by name rather than position.

Each package that provides a dictionary cursor has its own way of creating a dictionary cursor, although they all work the same way.

> The `sqlite3` package provides a `Row` cursor, which can be indexed by position or by column name.

Table 4. Builtin Dictionary Cursors

Package	How to get a dictionary cursor

pymssql	`conn = pymssql.connect (…, as_dict=True)`
	`dcur = conn.cursor()`
	all cursors will be dict cursors

psycopg[1,2]	`import psycopg.extras`
	`conn = psycopg.connect(…)`
	`dcur = conn.cursor(cursor_factory=psycopg.extras.DictCursor)`
	only this cursor will be a dict cursor

sqlite3[1]	`conn = sqlite3.connect (…)`
	`dcur = conn.cursor()`
	`dcur.row_factory = sqlite3.Row`
	only this cursor will be a dict cursor
	
	`conn = sqlite3.connect (…)`
	`conn.row_factory = sqlite3.Row`
	`dcur = conn.cursor()`
	all cursors will be dict cursors

pymysql[1]	`import pymysql.cursors`
	
	`conn = pymysql.connect(…)`
	`dcur = conn.cursor(pymysql.cursors.DictCursor)`
	only this cursor will be a dict cursor
	
	`conn = pymysql.connect(…,`
	`cursorclass = pymysql.cursors.DictCursor`
	`)`
	all cursors will be dict cursors

oracledb	*Not available*
pyodbc	*Not available*
pgdb	*Not available*

[1] Cursor supports indexing by either key value (dict style) or integer position (list style), as well as iteration. [2] Also supports `RealDictCursor` which is an actual dictionary, and `NamedTupleCursor`, which is an actual `namedtuple`

Example

db_sqlite_dict_cursor.py

```
import sqlite3

s3conn = sqlite3.connect("../DATA/presidents.db")
# uncomment to make _all_ cursors dictionary cursors
# conn.row_factory = sqlite3.Row

NAME_QUERY = '''
    select firstname, lastname
    from presidents
    where termnum < 5
'''

cur = s3conn.cursor()

# select first name, last name from all presidents
cur.execute(NAME_QUERY)

for row in cur.fetchall():
    print(row)
print('-' * 50)

dict_cursor = s3conn.cursor()  # get a normal SQLite3 cursor

# make _this_ cursor a dictionary cursor
dict_cursor.row_factory = sqlite3.Row  # set the row factory to be a Row object

# Row objects are dict/list hybrids -- row[name] or row[pos]

# select first name, last name from all presidents
dict_cursor.execute(NAME_QUERY)

for row in dict_cursor.fetchall():
    print(row['firstname'], row['lastname'])  # index row by column name

print('-' * 50)
```

db_sqlite_dict_cursor.py

```
('George', 'Washington')
('John', 'Adams')
('Thomas', 'Jefferson')
('James', 'Madison')
--------------------------------------------------
George Washington
John Adams
Thomas Jefferson
James Madison
--------------------------------------------------
```


Generic alternate cursors

- Create generator function
  - Get column names from *cursor*.description()
  - For each row
    - Make object from column names and values
      - Dictionary
      - Named tuple
      - Dataclass

For database modules that don't provice a dictionary cursor, the `iterrows_asdict()` function described below can be used with a cursor from any DB API-compliant package.

The example uses the metadata from the cursor to get the column names, and forms a dictionary by zipping the column names with the column values. `db_iterrows` also provides `iterrows_asnamedtuple()`, which returns each row as a named tuple, and `iterrows_asdataclass()`, which returns each row as an instance of a custom dataclass.

The functions in `db_iterrows` return generator objects. When you loop over the generator object, each element is a dictionary, named tuple, or instance of a dataclass, depending on which function you called.

Example

db_iterrows.py

```
"""
Generic functions that can be used with any DB API compliant
package.

To use, pass in a cursor after execute()-ing a
SQL query. Then iterate over the generator that is
returned
"""
from collections import namedtuple
from dataclasses import make_dataclass

def get_column_names(cursor):
    return [desc[0] for desc in cursor.description]

def iterrows_asdict(cursor):
    '''Generate rows as dictionaries'''
    column_names = get_column_names(cursor)
    for row in cursor.fetchall():
        row_dict = dict(zip(column_names, row))
        yield row_dict

def iterrows_asnamedtuple(cursor):
    '''Generate rows as named tuples'''
    column_names = get_column_names(cursor)
    Row = namedtuple('Row', column_names)
    for row in cursor.fetchall():
        yield Row(*row)

def iterrows_asdataclass(cursor):
    '''Generate rows as dataclass instances'''
    column_names = get_column_names(cursor)
    Row = make_dataclass('row_tuple', column_names)

    for row in cursor.fetchall():
        yield Row(*row)
```


Example

db_sqlite_iterrows.py

```
"""
Generic functions that can be used with any DB API compliant
package.

To use, pass in a cursor after execute()-ing a
SQL query. Then iterate over the generator that is
returned
"""
import sqlite3
from db_iterrows import *

sql_select = """
SELECT firstname, lastname, party
FROM presidents
WHERE termnum > 39
"""

conn = sqlite3.connect("../DATA/presidents.db")

cursor = conn.cursor()

cursor.execute(sql_select)

for row in iterrows_asdict(cursor):
    print(row['firstname'], row['lastname'], row['party'])

print('-' * 60)


cursor.execute(sql_select)

for row in iterrows_asnamedtuple(cursor):
    print(row.firstname, row.lastname, row.party)

print('-' * 60)


cursor.execute(sql_select)

for row in iterrows_asdataclass(cursor):
    print(row.firstname, row.lastname, row.party)
```

db_sqlite_iterrows.py

```
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald John Trump Republican
Joseph Robinette Biden Democratic
Donald John Trump Republican
------------------------------------------------------------
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald John Trump Republican
Joseph Robinette Biden Democratic
Donald John Trump Republican
------------------------------------------------------------
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald John Trump Republican
Joseph Robinette Biden Democratic
Donald John Trump Republican
```

Transactions

- Transactions allow safer control of updates
- `commit()` to make database changes permanent
- `rollback()` to discard changes

Sometimes a database task involves more than one change to your database (i.e., more than one SQL statement). You don't want the first SQL statement to succeed and the second to fail; this would leave your database in a corrupt state.

To be certain of data integrity, use **transactions**. This lets you make multiple changes to your database and only commit the changes if all the SQL statements were successful.

For all packages using the Python DB API, a transaction is started when you connect. At any point, you can call `CONNECTION.commit()` to save the changes, or `CONNECTION.rollback()` to discard the changes. If you don't call `commit()` after modifying a table, the data will not be saved.

Example

```
for values in list_of_tuples:
    try:
        cursor.execute(query,values)
    except SQLError:
        dbconn.rollback()
    else:
        dbconn.commit()
```

Autocommit

- Commits after every statement
- Not a best practice

Autocommit mode calls `commit()` after every non-query statement. This is not generally considered a best practice. See the table below for how autocommit is implemented in various DB packages.

Table 5. How to turn on autocommit

Package	Method/Attribute
oracledb	`conn.autocommit = True`
ibm_db_api	`conn.set_autocommit(True)`

pymysql	`pymysql.connect(…, autocommit=True)`
	or
	`conn.autocommit(True)`

psycopg	`conn.autocommit = True`
sqlite3	`sqlite3.connect(dbname, isolation_level=None)`

pymysql only supports transaction processing when using the **InnoDB** engine

Object-relational Mappers

- No SQL required
- Maps a class to a table
- All DB work is done by manipulating objects
- Most popular Python ORMs
  - SQLAlchemy
  - Django (which is a complete web framework)

An Object-relational mapper is a module or framework that creates a level of abstraction above the actual database tables and SQL queries. As the name implies, a Python class (object) is mapped to the actual table.

The two most popular Python ORMs are SQLAlchemy which is a standalone ORM, and Django ORM. Django is a comprehensive Web development framework, which provides an ORM as a subpackage. SQLAlchemy is the most fully developed package, and is the ORM used by Flask and some other Web development frameworks.

Instead of querying the database, you call a search method on an object representing a table. To add a row to the table, you create a new instance of the table class, populate it, and call a method like save(). You can create a large, complex database system, complete with foreign keys, composite indices, and all the other attributes near and dear to a DBA, without writing the first line of SQL.

You can use Python ORMs in two ways.

One way is to design the database with the ORM. To do this, you create a class for each table in the database, specifying the columns with predefined classes from the ORM. Then you run an ORM command which executes the queries needed to build the database. If you need to make changes, you update the class definitions, and run an ORM command to synchronize the actual DBMS to your classes.

The second way is to map tables to an existing database. You create the classes to match the schemas that have already been defined in the database. Both SQLAlchemy and the Django ORM have tools to automate this process.

NoSQL

- Non-relational database
- Document-oriented
- Can be hierarchical (nested)
- Examples
  - MongoDB
  - Cassandra
  - Redis

A current trend in data storage are called "NoSQL" or non-relational databases. These databases consist of *documents*, which are indexed, and may contain nested data.

NoSQL databases don’t contain tables, and do not have relations.

While relational databases are great for tabular data, they are not as good a fit for nested data. Geo-spatial, engineering diagrams, and molecular modeling can have very complex structures. It is possible to shoehorn such data into a relational database, but a NoSQL database might work much better. Another advantage of NoSQL is that it can adapt to changing data structures, without having to rebuild tables if columns are added, deleted, or modified.

Some of the most common NoSQL database systems are MongoDB, Cassandra and Redis.

Chapter 2 Exercises

Exercise 2-1

Part A (president_sqlite.py)

For this exercise, use the SQLite database named `presidents.db` in the DATA folder. It has the following layout

Table 6. Layout of President Table

Field Name	SQLite Data Type	Python Data type	Null	Default
termnum	int(11)	int	YES	NULL
lastname	varchar(32)	str	YES	NULL
firstname	varchar(64)	str	YES	NULL
termstart	date	date	YES	NULL
termend	date	date	YES	NULL
birthplace	varchar(128)	str	YES	NULL
birthstate	varchar(32)	str	YES	NULL
birthdate	date	date	YES	NULL
deathdate	date	date	YES	NULL
party	varchar(32)	str	YES	NULL

Copy the `president.py` module in the top level folder of the student files to `president_sqlite.py` and then modify `president_sqlite.py` to get its data from the Presidents table, as described above, rather than from `presidents.txt`.

Part B (president_main.py)

Copy `president_main.py` to `president_main_sqlite.py` and modify it to import the `President` class from the `president_sqlite` module. It should import the `President` class from this new module that uses the database instead of a text file, but otherwise work the same as before.

Exercise 2-2 (add_pres_sqlite.py)

Add another president to the presidents database. Just make up the data for your new president.

SQL syntax for adding a record is

```
INSERT INTO table ("COL1-NAME",..) VALUES ("VALUE1",...)
```

To do a parameterized insert (the right way!):

```
INSERT INTO table ("COL1-NAME",..) VALUES (?,?,...)
```

Index

S

T