OSC Extra

None

Version 1.0, July 2025

Table of Contents

About OSC Extra

Course Outline

Day 1

The actual schedule varies with circumstances. The last day may include *ad hoc* topics requested by students

Student files

You will need to load some student files onto your computer. The files are in a compressed archive. When you extract them onto your computer, they will all be extracted into a directory named **pyoscny-extra**. See the setup guides for details.

What's in the files?

pyoscny-extra contains all files necessary for the class
pyoscny-extra/EXAMPLES/ contains the examples from the course manuals.
pyoscny-extra/ANSWERS/ contains sample answers to the labs.
pyoscny-extra/DATA/ contains data used in examples and answers
pyoscny-extra/SETUP/ contains any needed setup scripts (may be empty)
pyoscny-extra/TEMP/ initially empty; used by some examples for output files

The following folders *may* be present:

pyoscny-extra/BIG_DATA/ contains large data files used in examples and answers
pyoscny-extra/NOTEBOOKS/ Jupyter notebooks for use in class
pyoscny-extra/LOGS/ initially empty; used by some examples to write log files

The student files do not contain Python itself. It will need to be installed separately. This may already have been done.

Examples

Most of the examples from the course manual are provided in EXAMPLES subdirectory.

It will look like this:

Example

cmd_line_args.py

```
import sys   # Import the sys module

print(sys.argv) # Print all parameters, including script itself

name = sys.argv[1]  # Get the first actual parameter
print("name is", name)
```

cmd_line_args.py apple mango 123

```
['/Users/jstrick/curr/courses/python/common/examples/cmd_line_args.py', 'apple', 'mango',
'123']
name is apple
```

Appendices

Classroom etiquette

Remote learning

- Mic off when you're not speaking. If multiple mics are on, it makes it difficult to hear
- The instructor doesn't know you need help unless you let them know via voice or chat.
- It's ok to ask for help a lot.
  - Ask questions. Ask questions. Ask questions.
  - ***INTERACT*** with the instructor and other students.
- Log off the remote S/W at the end of the day

In-person learning

- Noisemakers off
- No phone conversations
- Come and go quietly during class.

Please turn off cell phone ringers and other noisemakers.

If you need to have a phone conversation, please leave the classroom.

We're all adults here; feel free to leave the clasroom if you need to use the restroom, make a phone call, etc. You don't have to wait for a lab or break, but please try not to disturb others.

Please do not bring any exploding penguins to class. They might maim, dismember, or otherwise disturb your fellow students.

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-
wlwnfsqc/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: Consuming RESTful Data

Objectives

- Writing REST clients
  - Opening URLs and downloading data
  - Fetching data from RESTful servers
- Use the requests module to simplify authentication and proxies
- Uploading data to a server

The REST API

- Based on HTTP verbs
  - GET
    - get all records
    - get details for one record
  - POST add a new record
  - PUT replace a record (all fields)
  - PATCH update a record (some fields)
  - DELETE delete a record

REST stands for ***Re***presentational ***S***tate ***T***ransfer, first described (and named) by Roy Fielding in 2000. It is not a protocol or structure, but rather an architectural style resulting from a set of guidelines. It provides for loosely-coupled resource management over HTTP. REST does not enforce a particular implementation.

A RESTful site provides *resources*, which contain *records*. The same API typically contains more than one resource; each has a different *endpoint* (URL).

A RESTful API uses HTTP verbs to manipulate records. The same endpoint can be used for all access; what happens depends on a combination of which HTTP verb is used, plus whether there is more information on the URL.

HTTP Verbs

With just the endpoint (e.g., `https://www.wombats.com/api/v1/wombat`)

- GET retrieves links to all records. Query strings can be used to sort or filter the list.
- POST adds a new record

With endpoint plus record ID (e.g., `https://www.wombats.com/api/v1/wombat/123`)

- GET retrieves the details for that record
- PUT replaces the record
- PATCH updates the record
- DELETE removes the record

Example API

The Art Institute of Chicago API provides the following endpoints:

```
https://api.artic.edu/api/v1/artworks
https://api.artic.edu/api/v1/agents
https://api.artic.edu/api/v1/places
https://api.artic.edu/api/v1/galleries
https://api.artic.edu/api/v1/exhibitions
https://api.artic.edu/api/v1/agent-types
https://api.artic.edu/api/v1/agent-roles
https://api.artic.edu/api/v1/artwork-place-qualifiers
https://api.artic.edu/api/v1/artwork-date-qualifiers
https://api.artic.edu/api/v1/artwork-types
https://api.artic.edu/api/v1/category-terms
https://api.artic.edu/api/v1/images
https://api.artic.edu/api/v1/videos
https://api.artic.edu/api/v1/sounds
https://api.artic.edu/api/v1/texts
```

A list of public RESTful APIs is located here: https://github.com/public-apis/public-apis

When is REST not REST?

- REST is guidelines, not protocol
- Implementers are not consistent
- YMMV (your mileage may vary)

REST is a set of guidelines, not a specific protocol. Because of this, REST implementations vary widely. For instance, many APIs use more than one endpoint for the same resource. Many APIs do not return a list of links on a GET request to the endpoint, but the details for every resource. Many APIs vary widely on how you send credentials.

It is thus important to read the docs for each individual API to see exactly what they expect, and what they provide.

The good news is that the variations are mostly in the URLs you need to construct — making the requests and parsing out the data are generally about the same for most APIs.

Consuming REST APIs

- Use requests.*method*()
- Specify parameters, headers, etc. as dictionary
- Use response.content to get raw data
- Use response.json() to convert JSON to Python

The `requests` module is used to get data from RESTful APIs.

To add GET parameters, pass a dictionary of key-value parameters with the `params` keyword argument:

```
get_parameters = { 'key1': 'value1', ⋯ }
response = requests.get(URL, params=get_parameters)
```

For POST data, pass a dictionary with the `data` (dictionary) or `json` (**JSON** data) arguments.

```
post_data = { 'key1': 'value1', ⋯ }
response = requests.post(URL, data=post_data)
```

To use a proxy, pass a dictionary of protocol/url parameters with the `proxies` keyword

```
proxies={'http':'https://proxy.something.com:1234'}
response = requests.get(URL, proxies=proxies)
```

To convert a JSON response into a Python data structure, use the `.json()` method on the `response` object.

See the quickstart guide for requests at: https://requests.readthedocs.io/en/latest/user/quickstart/

Example

rest_consumer_omdb.py

```
import requests
from pprint import pprint

with open('omdbapikey.txt') as api_in:
    OMDB_API_KEY = api_in.read().rstrip()

OMDB_URL = "http://www.omdbapi.com"

def main():
    requests_params = {'t': 'Black Panther', "apikey": OMDB_API_KEY}
    response = requests.get(OMDB_URL, params=requests_params)
    if response.status_code == requests.codes.OK:
        raw_data = response.json()

        print(f"raw_data['Title']: {raw_data['Title']}")
        print(f"raw_data['Director']: {raw_data['Director']}")
        print(f"raw_data['Year']: {raw_data['Year']}")
        print(f"raw_data['Runtime']: {raw_data['Runtime']}")
        print()

        print('-' * 60)

        print("raw DATA:")
        pprint(response.json())
    else:
        print(f"response.status_code: {response.status_code}")

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

rest_consumer_omdb.py

```
raw_data['Title']: Black Panther
raw_data['Director']: Ryan Coogler
raw_data['Year']: 2018
raw_data['Runtime']: 134 min

------------------------------------------------------------
raw DATA:
{'Actors': "Chadwick Boseman, Michael B. Jordan, Lupita Nyong'o",
 'Awards': 'Won 3 Oscars. 124 wins & 289 nominations total',
 'BoxOffice': '$700,426,566',
 'Country': 'United States',
 'DVD': 'N/A',
 'Director': 'Ryan Coogler',
 'Genre': 'Action, Adventure, Sci-Fi',
 'Language': 'English, Swahili, Nama, Xhosa, Korean',
 'Metascore': '88',
 'Plot': "T'Challa, heir to the hidden but advanced kingdom of Wakanda, must "
         'step forward to lead his people into a new future and must confront '
         "a challenger from his country's past.",
 'Poster': 'https://m.media-
amazon.com/images/M/MV5BMTg1MTY2MjYzNV5BMl5BanBnXkFtZTgwMTc4NTMwNDI@._V1_SX300.jpg',
 'Production': 'N/A',
 'Rated': 'PG-13',
 'Ratings': [{'Source': 'Internet Movie Database', 'Value': '7.3/10'},
             {'Source': 'Rotten Tomatoes', 'Value': '96%'},
             {'Source': 'Metacritic', 'Value': '88/100'}],
 'Released': '16 Feb 2018',
 'Response': 'True',
 'Runtime': '134 min',
 'Title': 'Black Panther',
 'Type': 'movie',
 'Website': 'N/A',
 'Writer': 'Ryan Coogler, Joe Robert Cole, Stan Lee',
 'Year': '2018',
 'imdbID': 'tt1825683',
 'imdbRating': '7.3',
 'imdbVotes': '879,347'}
```

Table 1. Keyword Parameters for `requests` methods

Option	Data Type	Description
`allow_redirects`	`bool`	set to True if PUT/POST/DELETE redirect following is allowed
`auth`	`tuple`	authentication pair (user/token,password/key)
`cert`	`str or tuple`	path to cert file or ('cert', 'key') tuple
`cookies`	`dict or CookieJar`	cookies to send with request
`data`	`dict`	parameters for a POST or PUT request
`files`	`dict`	files for multipart upload
`headers`	`dict`	HTTP headers
`json`	`str`	JSON data to send in request body
`params`	`dict`	parameters for a GET request
`proxies`	`dict`	map protocol to proxy URL
`stream`	`bool`	if False, immediately download content
`timeout`	`float or tuple`	timeout in seconds or (connect timeout, read timeout) tuple
`verify`	`bool`	if True, then verify SSL cert

These can be used with any of the HTTP request types, as appropriate.

Table 2. `requests.Response` methods and attributes

Method/attribute	Definition
`apparent_encoding`	Returns the apparent encoding
`close()`	Closes the connection to the server
`content`	Content of the response, in bytes
`cookies`	A CookieJar object with the cookies sent back from the server
`elapsed`	A timedelta object with the time elapsed from sending the request to the arrival of the response
`encoding`	The encoding used to decode r.text
`headers`	A dictionary of response headers
`history`	A list of response objects holding the history of request (url)
`is_permanent_redirect`	True if the response is the permanent redirected url, otherwise False
`is_redirect`	True if the response was redirected, otherwise False
`iter_content()`	Iterates over the response
`iter_lines()`	Iterates over the lines of the response
`json()`	Converts JSON content to Python data structure (if the result was written in JSON format, if not it raises an error)
`links`	The header links
`next`	A PreparedRequest object for the next request in a redirection
`ok`	True if status_code is less than 400, otherwise False
`raise_for_status()`	If an error occur, this method a HTTPError object
`reason`	A text corresponding to the status code
`request`	The request object that requested this response
`status_code`	A number that indicates the status (200 is OK, 404 is Not Found)
`text`	The content of the response, in unicode
`url`	The URL of the response

Printing JSON

- Default output of JSON is ugly
- `pprint` makes structures human friendly
- Use `pprint.pprint()`

When debugging JSON data, the `print` command is not so helpful. The Python data structure parsed from JSON is just printed out all jammed together, one element after another, and is hard to read.

The `pprint` (pretty print) module will analyze a structure and print it out with indenting, to make it much easier to read.

By default, `pprint()` displays dictionary keys in sorted order. It usually makes sense to view the keys in the same order they were in the JSON file. To do this, add the `sort_dicts=False` parameter to `pprint()`.

You can customize the output even more with other parameters: `indent` (default 1) specifies how many spaces to indent nested structures; `width` (default 80) constrains the width of the output; `depth` (default unlimited) says how many levels to print – levels beyond depth are shown with '...'.

Requests sessions

- Share configuration across requests
- Can be faster
- Instance of requests.Session
- Supports context manager (*with* statement)

To make it more convenient to share HTTP information across multiple requests, **requests** provides *sessions*. You can use a session by creating an instance of **requests.Session**. Once the session is created, it contains attributes containing the named arguments to request methods, such as *session*.params or *session*.headers.

A Session object implements the context manager protocol, so it can be used with the **with** statement. This will automatically close the session.

These are normal Python dictionaries, so you can use the *update* method to add information:

```
with requests.Session() as session:
    session.params.update({'token': 'MY_TOKEN'})
    session.headers.update({'accept': 'application/xml'})

    response1 = session.get(...)
    response2 = session.get(...)
```

Then you can call the HTTP methods (**get**, **post**, etc.) from the session object. Any parameters passed to an HTTP method will overwrite those that already exist in the session, but do not update the session.

For more examples of using REST in a real-world situation, see `consume_omdb_main.py` in the `EXAMPLES` folder. This script imports various modules that connect to the OMDB API using various multitasking approaches.

Example

rest_consumer_omdb_sessions.py

```
import requests
from pprint import pprint

with open('omdbapikey.txt') as api_in:
    OMDB_API_KEY = api_in.read().rstrip()


OMDB_URL = "http://www.omdbapi.com"

MOVIE_TITLES = [
    'Black Panther',
    'Frozen',
    'Top Gun: Maverick',
    'Bullet Train',
    'Death on the Nile',
    'Casablanca',
]

def main():
    with requests.Session() as session:
        session.params.update({"apikey": OMDB_API_KEY})
        for movie_title in MOVIE_TITLES:
            params = {'t': movie_title}
            response = session.get(OMDB_URL, params=params)
            if response.status_code == requests.codes.OK:
                raw_data = response.json()
                print(f"raw_data['Title']: {raw_data['Title']}")
                print(f"raw_data['Director']: {raw_data['Director']}")
                print(f"raw_data['Year']: {raw_data['Year']}")
                print(f"raw_data['Runtime']: {raw_data['Runtime']}")
                print()



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

rest_consumer_omdb_sessions.py

```
raw_data['Title']: Black Panther
raw_data['Director']: Ryan Coogler
raw_data['Year']: 2018
raw_data['Runtime']: 134 min

raw_data['Title']: Frozen
raw_data['Director']: Chris Buck, Jennifer Lee
raw_data['Year']: 2013
raw_data['Runtime']: 102 min

raw_data['Title']: Top Gun: Maverick
raw_data['Director']: Joseph Kosinski
raw_data['Year']: 2022
raw_data['Runtime']: 130 min

raw_data['Title']: Bullet Train
raw_data['Director']: David Leitch
raw_data['Year']: 2022
raw_data['Runtime']: 127 min

raw_data['Title']: Death on the Nile
raw_data['Director']: Kenneth Branagh
raw_data['Year']: 2022
raw_data['Runtime']: 127 min

raw_data['Title']: Casablanca
raw_data['Director']: Michael Curtiz
raw_data['Year']: 1942
raw_data['Runtime']: 102 min
```

Authentication with requests

- Options
  - Basic-Auth
  - Digest
  - Custom
- Use **auth** argument

requests makes it easy to provide basic authentication to a web site.

In the simplest case, create a `requests.auth.HTTPBasicAuth` object with the username and password, then pass that to requests with the `auth` argument. Since this is a common use case, you can also just pass a `(user, password)` tuple to the `auth` parameter.

For digest authentication, use `requests.auth.HTTPDigestAuth` with the username and password.

For custom authentication, you can create your own auth class by inheriting from `requests.auth.AuthBase`.

For OAuth 1, OAuth 2, and OpenID, install `requests-oauthlib`. This additional module provides auth objects that can be passed in with the `auth` parameter, as above.

See https://docs.python-requests.org/en/latest/user/authentication/ for more details.

Example

http_basic_auth.py

```
import requests
from requests.auth import HTTPBasicAuth, HTTPDigestAuth

# base URL for httpbin
BASE_URL = 'https://httpbin.org'

# formats for httpbin
BASIC_AUTH_FMT = "/basic-auth/{}/{}"
DIGEST_AUTH_FMT = "/digest-auth/{}/{}/{}"

USERNAME = "spam"
PASSWORD = "ham"
BAD_PASSWORD = "toast"

REPORT_FMT = "{:35s} {}"

def main():
    basic_auth()
    digest()

def basic_auth():
    auth = HTTPBasicAuth(USERNAME, PASSWORD)
    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth good password", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=(USERNAME, PASSWORD),
    )
    print(REPORT_FMT.format("Basic auth good password (shortcut)", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, BAD_PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth bad password", response))

def digest():
    auth = HTTPDigestAuth(USERNAME, PASSWORD)
    response = requests.get(
```

```
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth good password", response))

    auth = HTTPDigestAuth(USERNAME, BAD_PASSWORD)
    response = requests.get(
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth bad password", response))


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

http_basic_auth.py

```
Basic auth good password            <Response [200]>
Basic auth good password (shortcut) <Response [200]>
Basic auth bad password             <Response [401]>
Digest auth good password           <Response [200]>
Digest auth bad password            <Response [401]>
```

Posting data to a RESTful server

- use requests.post(url, data=dict)

The **POST** operation adds a new record to a resource using the endpoint.

To post to a RESTful service, use the **data** argument to **request**'s **post** function. It takes a dictionary of parameters, which will be URL-encoded automatically.

If the POST is successful, the server should return response data with a link to the newly created record, with an HTTP response code of 201 ("created") rather than 200 ("OK").

Example

post_to_rest.py

```
from datetime import datetime
import time
import requests

URL = 'http://httpbin.org/post'

for i in range(3):
    response = requests.post(  # POST data to server
        URL,
        data={'date': datetime.now(),
            'label': 'test_' + str(i)
        },
        cookies={'python': 'testing'},
        headers={'X-Python': 'Guido van Rossum'},
    )
    if response.status_code in (requests.codes.OK, requests.codes.created):
        print(response.status_code)
        print(response.text)
        print()
        time.sleep(2)
```

post_to_rest.py

```
200
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "date": "2025-07-18 08:45:52.989850",
    "label": "test_0"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "48",
    "Content-Type": "application/x-www-form-urlencoded",
    "Cookie": "python=testing",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.31.0",
    "X-Amzn-Trace-Id": "Root=1-687a4201-44b3b3061e2dd6122d549bbf",
    "X-Python": "Guido van Rossum"
  },
  "json": null,
  "origin": "72.15.17.50",
  "url": "http://httpbin.org/post"
}


200
{
  "args": {},
  "data": "",
```

…

Other operations

- Use **requests** functions
  - put
  - delete
  - patch
  - head
  - options

For other HTTP operations, **requests** provides appropriately named functions.

Example

other_web_service_ops.py

```
import requests

print('PUT:')
r = requests.put("http://httpbin.org/put", data={'spam': 'ham'}) # send data via HTTP PUT
request
print(r.status_code, r.text)
print('-' * 60)

print('DELETE:')
r = requests.delete("http://httpbin.org/delete") # send HTTP DELETE request
print(r.status_code, r.text)
print('-' * 60)

print('HEAD:')
r = requests.head("http://httpbin.org/get") # get HTTP headers via HEAD request
print(r.status_code, r.text)
print(r.headers)
print('-' * 60)

print('OPTIONS:')
r = requests.options("http://httpbin.org/get") # get negotiated HTTP options
print(r.status_code, r.text)
print('-' * 60)
```

other_web_service_ops.py

```
PUT:
200 {
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "spam": "ham"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "8",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.31.0",
    "X-Amzn-Trace-Id": "Root=1-687a4222-1742cddd2b620d2c5c021bd8"
  },
  "json": null,
  "origin": "72.15.17.50",
  "url": "http://httpbin.org/put"
}

------------------------------------------------------------
DELETE:
200 {
  "args": {},
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "0",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.31.0",
    "X-Amzn-Trace-Id": "Root=1-687a4232-2f5a62cc433f61c51e1db540"
  },
  "json": null,
  "origin": "72.15.17.50",
  "url": "http://httpbin.org/delete"
}

------------------------------------------------------------
HEAD:
200
```

```
{'Date': 'Fri, 18 Jul 2025 12:46:49 GMT', 'Content-Type': 'application/json', 'Content-
Length': '304', 'Connection': 'keep-alive', 'Server': 'gunicorn/19.9.0', 'Access-Control-
Allow-Origin': '*', 'Access-Control-Allow-Credentials': 'true'}
------------------------------------------------------------
OPTIONS:
200
------------------------------------------------------------
```

Using Thunder Client

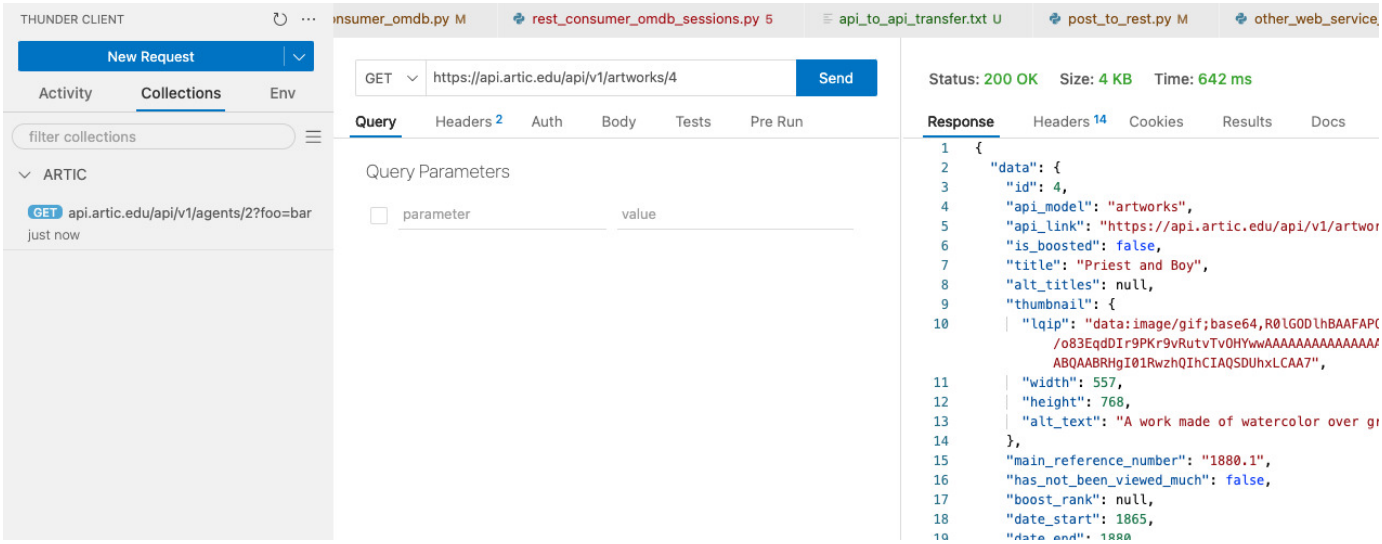
- Graphical HTTP Client
- Specify headers and parameters
- Save searches

Thunder Client is an extension to VS Code that provides a GUI-based HTTP client. It lets you specify URLs, plus headers, parameters, data, and any other information that needs to be sent to a web server. You can save searches for replay, and easily make changes to the requests.

This makes it really easy to experiment with REST endpoints without having to constantly re-run your Python scripts.

It also displays the JSON (or other) response in various modes, pretty-printed with syntax highlighting.

Install Thunder Client via the Extensions badge on the activity panel on the far left side of VS Code.



Chapter 2 Exercises

Exercise 2-1 (noaa_precip.py)

NOAA provides an API for climate data. The base page is https://www.ncdc.noaa.gov/cdo-web/webservices/v2#gettingStarted.

You will need to request a token via email. Specify the token with the `token` header. Use the `headers` parameter to requests.get().

You can get a list of the web services as described on the page.

For hourly precipitation, the dataset ID is "PRECIP_HLY".

The station ID for Boaz, Alabama is "COOP:010957"

Using your token, the dataset ID, and the station ID, fetch hourly precipitation data for that location from January 1, 1970 through January 31, 1970.

The endpoint is

https://www.ncdc.noaa.gov/cdo-web/api/v2/data

and you will need to specify the parameters **datasetid**, **stationid**, **startdate**, and **enddate**.

Remember that only a certain number of values are returned with each call. For purposes of this lab, just get the first page of values.

Try varying the parameters, based on the API docs.

See ANSWERS/noaa_metadata.py for a script to retrieve values for some of the endpoints.

Chapter 3: Advanced XML

Objectives

- lxml recap
- Understand what schemas are for
- Learn about XML namespaces
- Search with namespaces
- Search large XML files with iterparse()
- Focus in on XSD
- Validate an XML document
- Generate Data from XML
- Generate XML from data

lxml recap

- Fast, accurate XML parser
- `etree.parse()`
  - Creates `ElementTree` object
  - Contains `Element` objects (nested)
- `etree.iterparse()`
  - loops over tags
  - returns `Element` objects

The most popular XML parser for Python is `lxml`. It is not part of the standard library, but is included with many Python bundles, such as Anaconda. It can also be installed with **pip**.

While `lxml` has several subpackages, the most commonly used is `lxml.etree`, which is a superset of `xml.etree.ElementTree` from the standard library.

The **etree** interface has two primary approaches:

For XML files of less than gigabyte size, and if you need to navigate or modify the DOM, use `etree.parse()`, which returns an ElementTree object containing a tree of Element objects.

For large XML files, or when speed is important, use `etree.iterparse()`, which takes a **SAX** (stream) approach. This will iterate over the tags of a large XML document, and you can clear the memory of nodes as you process them.

`lxml.etree` is frequently aliased to `ET` or `et`.

ElementTree recap

- One ElementTree to contain the *document*
- Document is a tree of Elements
- Each Element has
  - Tag name (string)
  - Attributes (dictionary-like)
  - Text (string)
  - Tail (string)
  - Child elements (list-like)

In `ElementTree`, an XML document consists of a nested tree of `Element` objects. Each `Element` corresponds to an XML tag.

An `ElementTree` object serves as a wrapper for reading or writing the XML text.

If you are parsing existing XML, use `ElementTree.parse()`; this creates the `ElementTree` wrapper and the tree of elements. You can then navigate to, or search for, elements within the tree. You can also insert and delete new elements.

```
doc = et.parse('mydoc.xml')
for element in doc.findall('.//some_tag'):
    print(element.text)
```

If you are creating a new document from scratch, create a top-level (AKA "root") `Element`, then create child elements as needed.

When creating a new `Element`, you can initialize it with the tag name and any attributes. Once created, you can add the text that will be contained within the element's tags, or add other attributes.

When you are ready to save the XML into a file, initialize an `ElementTree` with the root element.

The `Element` object is a hybrid of list and dictionary. Access child elements by treating the element as a list. Access attributes by treating it as a dictionary, using the `.get()` method (`[]` indexing is not allowed).

```
element = root.find('sometag')
for child_element in element:
    print child_element.tag
print element.get('someattribute')
```

The `Element` object also has several useful properties: `tag` is the element's tag; `text` is the text contained inside the element; `tail` is the text following the element, before the next element.

Only the `tag` property of an `Element` is required.

You can use the `SubElement` class to easily add children (subelements) to an element:

```
e = et.Element('spam')
s = et.SubElement(e, 'ham')  # add <ham> under <spam>
s.text = 'toast'
```

Example

xml_planets_xpath3.py

```
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml')  # parse XML file

for planet in doc.findall('.//planet'):  # find all elements (relative to root element)
with tag "planet"
    print(planet.get('planetname'))      # get XML attribute value
    for moon in planet.findall('moon'):  # find all child elements with tag "moon"
        print(f'\t{moon.text}')  # print text contained in "moon" tag
```

xml_planets_xpath3.py

```
Mercury
Venus
Earth
    Moon
Mars
    Deimos
    Phobos
Jupiter
    Metis
    Adrastea
    Amalthea
    Thebe
    Io
    Europa
    Gannymede
```

...

Table 3. Element methods and properties

Method/Property	Description
`append(element)`	Add a subelement element to end of subelements
`attrib`	Dictionary of element's attributes
`clear()`	Remove all subelements
`find(path)`	Find first subelement matching path
`findall(path)`	Find all subelements matching path
`findtext(path)`	Shortcut for find(path).text
`get(attr)`	Get an attribute; Shortcut for attrib.get()
`getiterator()`	Returns an iterator over all descendants
`getiterator(path)`	Returns an iterator over all descendants matching path
`insert(pos,element)`	Insert subelement element at position pos
`items()`	Get all attribute values; Shortcut for attrib.items()
`keys()`	Get all attribute names; Shortcut for attrib.keys()
`remove(element)`	Remove subelement element
`set(attrib,value)`	Set an attribute value; shortcut for attr[attrib] = value
`tag`	The element's tag
`tail`	Text following the element
`text`	Text contained within the element

Table 4. ElementTree methods and properties

Property	Description
`find(path)`	Finds the first toplevel element with given tag; shortcut for getroot().find(path).
`findall(path)`	Finds all toplevel elements with the given tag; shortcut for getroot().findall(path).
`findtext(path)`	Finds element text for first toplevel element with given tag; shortcut for getroot().findtext(path).
`getiterator(path)`	Returns an iterator over all descendants of root node matching path. (All nodes if path not specified)
`getroot()`	Return the root node of the document
`parse(filename)` `parse(fileobj)`	Parse an XML source (filename or file-like object)
`write(filename,encoding)`	Writes XML document to filename, using encoding (Default us-ascii).

Using etree.iterparse()

- Parse large XML files
- Does not build DOM
- Can clear nodes after use

The only drawback to `etree.parse()` is that it puts the entire XML document in memory. To process very large XML files (multi-gigabyte or larger), use `etree.iterparse()`. This is a SAX-oriented parser that parses the XML file serially.

To use `iterparse()`, pass in the name of the file to parse, plus the name of the tag you're looking for. It will return an iterator that will iterate over each instance of the specified tag as an etree `Element`, which will, as usual, include the tags contained within it. Tags outside the specified the tag will be skipped.

When finished with each tag, be sure to delete the element to free up all possible memory.

See `xml_iterparse_check_mem.py` and `xml_no_iterparse_check_mem.py` to see how much memory is saved by using `iterparse()`.

See https://lxml.de/3.2/parsing.html#iterparse-and-iterwalk for more about **iterparse()**. There are many option to fine-tune its behavior.

Example

xml_iterparse.py

```
from lxml.etree import iterparse


def main():
    # Create a 'context' -- start the parser, skipping all but specified tag
    doc = iterparse("../BIG_DATA/pubmed19n0001.xml", tag='PubmedArticle')

    # Loop over each occurrence of tag ("end event").
    # Use enumerate() to count how many elements found
    for i, (event, element) in enumerate(doc, 1):
        # Within the found element (tag="PubmedArticle") find subelements and get their
text
        article_title = element.findtext('.//ArticleTitle')
        year_completed = element.findtext('.//DateCompleted/Year')
        month_completed = element.findtext('.//DateCompleted/Month')
        clear_element(element)  # Clear element after all wanted data is extracted
        print(f"{month_completed}/{year_completed} {article_title[:70]}")

    print(f"Total count: {i}")


def clear_element(element):
    element.clear()  # Remove element from memory
    while element.getprevious() is not None:  # Loop over siblings
        # Delete parents (saves memory left over in empty nodes)
        del element.getparent()[0]


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

xml_iterparse.py

```
01/1976 Formate assay in body fluids: application in methanol poisoning.
01/1976 Delineation of the intimate details of the backbone conformation of py
01/1976 Metal substitutions incarbonic anhydrase: a halide ion probe study.
01/1976 Effect of chloroquine on cultured fibroblasts: release of lysosomal hy
01/1976 Atomic models for the polypeptide backbones of myohemerythrin and heme
01/1976 Studies of oxygen binding energy to hemoglobin molecule.
01/1976 Maturation of the adrenal medulla--IV. Effects of morphine.
01/1976 Comparison between procaine and isocarboxazid metabolism in vitro by a
01/1976 Radiochemical assay of glutathione S-epoxide transferase and its enhan
01/1976 Digitoxin metabolism by rat liver microsomes.
```

...

```
01/1979 [Is it possible to predict the antihypertensive action of a beta-block
01/1979 [Data from the study of effort blood pressure profile in hypertensive
01/1979 [Utilization of pindolol in arterial hypertension. Study during rest a
01/1979 [Value of a single oral dose of pindolol].
01/1979 [Place of beta-blockers in the treatment of essential arterial hyperte
01/1979 [Long-term utilization of beta-blockers in arterial hypertension].
01/1979 [Treatment by beta-blockers of arterial hypertension in patients with
01/1979 [beta-blockers in hypertension with renal failure].
01/1979 [Beta-blockers and arterial hypertension in the aged patient].
Total count: 30000
```

Example

memorychecker.py

```
import os
import psutil

class MemoryChecker():
    """
    Callable class to get current memory use of program.

    Instances of this class may be called, at which time
    they will return current memory use.
    """

    def __init__(self):
        self.process = psutil.Process(os.getpid())  # Get PID of current process

    def __call__(self):
        return self.process.memory_info().rss  # Return memory use for PID

if __name__ == '__main__':
    mc = MemoryChecker()
    print(mc())  # can call at any time to get current memory use
```

Example

xml_iterparse_check_mem.py

```
from lxml.etree import iterparse
from memorychecker import MemoryChecker

def main():
    doc = iterparse("../BIG_DATA/pubmed19n0001.xml", tag='PubmedArticle')  # Start parser

    mem_checker = MemoryChecker()  # Create memory checker object

    for i, (event, element) in enumerate(doc, 1):  # Loop over (and count) found elements
        year_completed = element.findtext('.//DateCompleted/Year')  # Get text from
subelement
        month_completed = element.findtext('.//DateCompleted/Month')
        clear_element(element)
        current_mem_use = mem_checker()  # Get current memory use

        print(f"{i:5d}. {month_completed}/{year_completed} {current_mem_use:,d}")

    print(f"Total count: {i}")


def clear_element(element):
    element.clear()  # Remove memory used by current node
    while element.getprevious() is not None:  # Loop over siblings already seen
        del element.getparent()[0]  # Remove sibling


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

xml_iterparse_check_mem.py

```
    1. 01/1976 17,629,184
    2. 01/1976 17,657,856
    3. 01/1976 17,657,856
    4. 01/1976 17,657,856
    5. 01/1976 17,657,856
    6. 01/1976 17,727,488
    7. 01/1976 17,727,488
    8. 01/1976 17,727,488
    9. 01/1976 17,727,488
   10. 01/1976 17,727,488
```

...

```
29992. 01/1979 19,075,072
29993. 01/1979 19,075,072
29994. 01/1979 19,075,072
29995. 01/1979 19,075,072
29996. 01/1979 19,075,072
29997. 01/1979 19,075,072
29998. 01/1979 19,075,072
29999. 01/1979 19,075,072
30000. 01/1979 19,075,072
Total count: 30000
```

Working with namespaces

- Prefix for specific set of XML tags
- May be by company or data domain
- Pain in the neck
- Supported by LXML.etree and xml.etree.ElementTree

An XML *namespace* is a prefix for a specific set of XML tags. Namespaces cause extra work, yet they may be needed when combining information from multiple sources, due to duplicate tag names.

For instance, a construction company might have a "pool" tag to represent whether a house has a pool; a software company might use "pool" to represent a group of threads or processes.

Namespaces are defined with the reserved XML attribute name `xmlns`. It can be in two forms – `xmlns=URL` designates a default namespace, while `xmlns:prefix=URL` declares tags that start with `prefix:` to be in that namespace.

You can search for tags or attributes with namespace prefixes using `etree.parse()` or `etree.iterparse()`; you can use `ElementMaker` to create XML with the appropriate prefixes.

Example

Sample ODT (Open Office) document. ODT files use namespaces for different parts of the document.

```
      <text:p text:style-name="P21">XML is an extensible markup language, ultimately
derived from SGML. It is designed as a metalanguage, with which sublanguages can be
defined for various areas of interest. For instance, NOAA can define a subset of XML that
describes hurricane data, and Amazon can define one that describes an online order.
</text:p>
      <text:p text:style-name="P21">An XML file consists of a header entry, followed by
one or more XML elements. Each element has a tag (name) and <text:s/>is enclosed in angle
brackets. If the element contains other elements, or text, there are separate start and
end tags. If the element does not have any contents, there is just one tag. If there is
an end tag, it starts with a slash in front of the tag name. If there is no end tag, the
element ends with a slash. </text:p>
      <text:p text:style-name="P21">Elements can also have attributes, which are
name=value pairs. Attributes go inside the start tag.</text:p>
      <text:p text:style-name="EX-Heading">Examples</text:p>
      <text:p text:style-name="EX-Code">&lt;year&gt;1927&lt;/year&gt;</text:p>
      <text:p text:style-name="EX-Code">&lt;<text:span text:style-name="T23">id
value="123"/&gt;</text:span></text:p>
      <text:p text:style-name="P33"><text:line-break/>&lt;book id="123"&gt;&lt;title&gt;A
Study in Scarlet&lt;/title&gt;&lt;author&gt;A. Conan
Doyle&lt;/author&gt;&lt;/book&gt;</text:p>
      <text:p text:style-name="P33"/>
      <text:p text:style-name="P33"/>
      <text:p text:style-name="P6">lxml recap</text:p>
      <text:list xml:id="list134210767341594" text:continue-numbering="true" text:style-
name="MT-Major_20_point_20_bullets">
        <text:list-item>
          <text:p text:style-name="P73">Fast, accurate XML parser</text:p>
        </text:list-item>
        <text:list-item>
          <text:p text:style-name="P73">Creates ElementTree object</text:p>
        </text:list-item>

      <text:list-item>
          <text:p text:style-name="P73">Contains Element<text:span text:style-
name="T23">s (nested)</text:span></text:p>
        </text:list-item>
      </text:list>
```

Using etree.parse() with namespaces

- Namespaces available via *doc*.nsmap
- Pass namespaces to find...() methods
- Search tags or attributes

When using `etree.parse()` to create an `ElementTree` document, a dictionary of the namespaces is available as *document*.nsmap. You can search for tags or attributes which have a namespace prefix by passing this dictionary to the `find()` or `findall()` methods.

You could also build the dictionary manually, or remove namespaces you will not need.

Example

xml_ns.py

```
from pprint import pprint
import lxml.etree as ET

XML_FILE = '../DATA/libreoffice_content.xml'

TAG_LIST = [  # List of tags (with ns prefix) to find
    'style:font-face',
    'text:list-style',
]

ATTRIBUTE_LIST = [  # List of attributes (with ns prefix) to find
    'text:name',
    'style:name',
]


def main():

    doc = ET.parse(XML_FILE)  # Parse XML file into ElementTree

    root = doc.getroot()  # Get the root Element

    print("Namespace map:\n")
    pprint(root.nsmap)  # Display the namespace map as found in the XML document
    print("\n\n")

    print("Finding nodes by tags:\n")
```

```
    for tag in TAG_LIST:
        xpath = './/' + tag  # Create xpath search string for NS:TAG
        node = root.find(xpath, root.nsmap)  # Find tag; NS prefix is looked up in NS map
        print(f"Searching for tag [{xpath}]\n")
        print_node(node)

    print("Finding nodes by attributes:\n")

    for attribute in ATTRIBUTE_LIST:
        xpath = f'.//*[@{attribute}]'  # Create xpath search string for NS:ATTRIBUTE
        node = root.find(xpath, root.nsmap)
        print(f"Searching for attribute [{xpath}]\n")
        print_node(node)


def print_node(node):
    node_as_string = ET.tostring(node, pretty_print=True).decode()  # Convert Element to
bytes object with pretty_printing, then decode to str
    print(node_as_string, '\n')

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

xml_ns.py

```
Namespace map:

{'calcext': 'urn:org:documentfoundation:names:experimental:calc:xmlns:calcext:1.0',
 'chart': 'urn:oasis:names:tc:opendocument:xmlns:chart:1.0',
 'css3t': 'http://www.w3.org/TR/css3-text/',
 'dc': 'http://purl.org/dc/elements/1.1/',
 'dom': 'http://www.w3.org/2001/xml-events',
 'dr3d': 'urn:oasis:names:tc:opendocument:xmlns:dr3d:1.0',
 'draw': 'urn:oasis:names:tc:opendocument:xmlns:drawing:1.0',
 'drawooo': 'http://openoffice.org/2010/draw',
 'field': 'urn:openoffice:names:experimental:ooo-ms-interop:xmlns:field:1.0',
 'fo': 'urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0',
 'form': 'urn:oasis:names:tc:opendocument:xmlns:form:1.0',
 'formx': 'urn:openoffice:names:experimental:ooxml-odf-interop:xmlns:form:1.0',
 'grddl': 'http://www.w3.org/2003/g/data-view#',
 'loext': 'urn:org:documentfoundation:names:experimental:office:xmlns:loext:1.0',
 'math': 'http://www.w3.org/1998/Math/MathML',
 'meta': 'urn:oasis:names:tc:opendocument:xmlns:meta:1.0',
 'number': 'urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0',
 'of': 'urn:oasis:names:tc:opendocument:xmlns:of:1.2',
 'office': 'urn:oasis:names:tc:opendocument:xmlns:office:1.0',
 'officeooo': 'http://openoffice.org/2009/office',
 'ooo': 'http://openoffice.org/2004/office',
 'oooc': 'http://openoffice.org/2004/calc',
 'ooow': 'http://openoffice.org/2004/writer',
 'rpt': 'http://openoffice.org/2005/report',
 'script': 'urn:oasis:names:tc:opendocument:xmlns:script:1.0',
 'style': 'urn:oasis:names:tc:opendocument:xmlns:style:1.0',
 'svg': 'urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0',
 'table': 'urn:oasis:names:tc:opendocument:xmlns:table:1.0',
```

Using etree.iterparse() with namespaces

- Make one pass to get namespaces
  - May save as file
- Add namespaces to desired tags
- Call `iterparse` as usual

One way to use `etree.iterparse()` with namespaces is to make two passes.

In the first pass, find all the namespace mappings by using the 'start-ns' event and put them into a dictionary where the key is the namespace prefix and the value is the full namespace, which is surrounded by `{}`.

Next, for each tag you want to find, use the mapping you created to replace the namespace prefix with the actual namespace.

Finally, pass a list of those tags as the second argument to `.iterparse()`.

Example

xml_iterparse_ns.py

```
from lxml.etree import iterparse

XML_FILE = '../DATA/libreoffice_content.xml'


TAG_LIST = [  # List of tags to find, with namespace prefixes
    'style:font-face',
    'text:list-level-style-number',
]


def main():
    nsmap = get_nsmap()  # Create the namespace mapping dictionary

    tag_list = [replace_prefix(tag, nsmap) for tag in TAG_LIST]  # Replace prefixes with
actual namespaces, as used by etree

    context = iterparse(XML_FILE, tag=tag_list)  # Open the file for parsing

    for i, (event, element) in enumerate(context, 1):  # Iterate over file, finding tags
as needed
        print(element.tag, element.text)
        # Do something with the element here. 'element' is a normal etree Element object,
which you can search, etc.

        element.clear()  # When finished with the element, remove it, since we don't need
it. If we don't do this, we will run out of memory when processing huge XML files

        while element.getprevious() is not None:  # Also clear siblings. if we don't do
this, on a really large XML file we'll leave empty nodes, and the memory use will add up.
On a sub-GB file, probably not so important
            del element.getparent()[0]

    print(f"found {i} elements")


def get_nsmap():
    """
    Parse the entire file to get the embedded namespaces,
    and turn them into a dictionary for later use.

    Note -- you could write this to run separately, saving the nsmap in a file
    (pickle would work great here) and then reuse it for
```

```
    subsequent searches, rather than always having to parse twice.

    :return: Dict of namespace mappings
    """
    # Parse document looking for beginning namespace tags
    # Returns tuples of (event, element) (in this case event is always 'start-ns')
    # Each element is a ("prefix", "namespace") tuple
    parser = iterparse(XML_FILE, events=('start-ns',))

    # Use generator expression to pull elements out of the parser,
    # which become the key:value pairs of a new dictionary mapping prefixes to namespaces
    # Note: This reads the entire XML file.
    nsmap = dict(element for event, element in parser)

    return nsmap  # Return NS map as dict


def replace_prefix(nstag, nsmap):
    """
    Replace the tag prefix with the actual namespace, surrounded by "{}", because
    this is what etree.iterparse() needs.

    :param nstag: A tag in the form:  "prefix:tag"
    :param nsmap: The dictionary of namespace mappings
    :return: A string with the full namespace as used by etree: "{full-namespace}tag"
    """
    prefix, tag = nstag.split(':')  # split "prefix:tag" into separate strings
    map = nsmap.get(prefix, "ERROR")   # get the namespace for that prefix
    return f"{{{map}}}{tag}"  # return the namespace and tag in etree format

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

xml_iterparse_ns.py

```
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
{urn:oasis:names:tc:opendocument:xmlns:style:1.0}font-face None
```

...

```
{urn:oasis:names:tc:opendocument:xmlns:text:1.0}list-level-style-number None
{urn:oasis:names:tc:opendocument:xmlns:text:1.0}list-level-style-number None
{urn:oasis:names:tc:opendocument:xmlns:text:1.0}list-level-style-number None
{urn:oasis:names:tc:opendocument:xmlns:text:1.0}list-level-style-number None
{urn:oasis:names:tc:opendocument:xmlns:text:1.0}list-level-style-number None
{urn:oasis:names:tc:opendocument:xmlns:text:1.0}list-level-style-number None
{urn:oasis:names:tc:opendocument:xmlns:text:1.0}list-level-style-number None
{urn:oasis:names:tc:opendocument:xmlns:text:1.0}list-level-style-number None
{urn:oasis:names:tc:opendocument:xmlns:text:1.0}list-level-style-number None
found 90 elements
```

E-Factory

- XML creation shortcuts
- Simpler than Element/Subelement
- E objects may be nested

The `E` object (AKA **E-Factory**), available from the `lxml.builder` subpackage, provides an easy way to generate XML (or HTML).

Attributes of `E` objects automatically become virtual functions that return `Element` objects which contain the tag name of the function, and can be passed any text to enclose.

If the element needs any XML attributes, pass them as named arguments to the `.tag()` method.

```
    T = E.title("Welcome to Python", id="123")
```

is the same as

```
    T = Element("title", id="123")
    T.text = "Welcome to Python)
```

That may not seem like a big improvement, but E functions may be nested, so you can say

```
doc = E.doc(
      E.title("Welcome to Python"),
      E.p("Paragraph 1"),
     E.p("Paragraph 2"),
)
```

To add attributes, use named parameters, as with `Element`. All virtual functions return an `Element` object.

Example

xml_efactory_simple.py

```
#
import lxml.etree as ET
from lxml.builder import E  # import E object

xml = (
    E.animals(  # use E to create nested tags
        E.animal('wombat', species="Vombatus ursinus"),
        E.animal('bushbaby', species="Galago senegalensis")
    )
)

print(ET.tostring(xml, pretty_print=True).decode())
```

xml_efactory_simple.py

```
<animals>
  <animal species="Vombatus ursinus">wombat</animal>
  <animal species="Galago senegalensis">bushbaby</animal>
</animals>
```

Example

xml_create_knights_efactory.py

```
'''Create, print, and save a new XML document'''
from lxml.builder import E  # import E object
import lxml.etree as ET

FILE_NAME = 'knights.xml'


def main():
    '''Program entry point'''
    knight_info = get_knight_info()
    knight_root = build_tree(knight_info)
    knight_doc = ET.ElementTree(knight_root)
    write_doc(knight_doc)


def get_knight_info():
    '''Read knight data from the file'''
    info = []
    with open('../DATA/knights.txt') as kn:
        for line in kn:
            flds = line[:-1].split(':')
            info.append(flds)
    return info


def build_tree(knight_recs):
    '''Build the new XML document'''
    tree = ET.Element('knights')
    for knight_rec in knight_recs:
        knight = E.knight(  # create <knight> tag
            E.title(knight_rec[1]),  # nest other tags inside <knight>
            E.color(knight_rec[2]),
            E.quest(knight_rec[3]),
            E.comment(knight_rec[4]),
            name=knight_rec[0],
        )

        tree.append(knight)  # add <knight> tag to <knights> tag

    return tree


def write_doc(doc):
```

```
    '''Write the XML document out to a file, pretty-printing if available'''
    doc.write(FILE_NAME, pretty_print=True)


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

ElementMaker

- Similar to E-Factory
- From etree.builder
- Pre-define tags
- Includes namespaces
- Avoids typos
- Can be used for XML or HTML

The `ElementMaker` class provides an Element factory that is namespace-aware. Create an instance of `ElementMaker` with a namespace, and a namespace mapping, then use properties of the `ElementMaker` instance to create functions. The properties do not have to be pre-defined.

The functions will then create the appropriate `Element` objects. Pass a string to a function for the tag contents; named parameters become attributes.

Elements may be nested, as usual, to create a document.

Example

xml_elementmaker.py

```
#
import csv
from lxml import etree
from lxml.builder import ElementMaker  # import ElementMaker

NS_RIVER_URL = "http://www.cja-tech.com/ns/river"  # (fake) URL for namespace

E = ElementMaker(  # create ElementMaker with namespace
    namespace=NS_RIVER_URL,
    nsmap={'lr': NS_RIVER_URL}
)

RIVER_LIST = E.riverlist  # use ElementMaker to create individual element makers
RIVER = E.river
RIVER_NAME = E.name
RIVER_LENGTH = E.length

doc = RIVER_LIST()  # create <riverlist> tag

with open('../DATA/longest_rivers.csv') as rivers_in:
    rdr = csv.reader(rivers_in)
    for row in rdr:  # iterate over records in file
        doc.append(
            RIVER(  # use element makers to create <river> elements, with other elements
nested inside
                RIVER_NAME(row[0]),
                RIVER_LENGTH(row[1])
            )
        )

print(etree.tostring(doc, pretty_print=True).decode())  # pretty-print XML
etree.ElementTree(doc).write('longest_rivers_ns.xml',
                             pretty_print=True)  # write XML to file
```

xml_elementmaker.py

```
<lr:riverlist xmlns:lr="http://www.cja-tech.com/ns/river">
  <lr:river>
    <lr:name>Nile</lr:name>
    <lr:length>4132</lr:length>
  </lr:river>
  <lr:river>
    <lr:name>Amazon River</lr:name>
    <lr:length>3976</lr:length>
  </lr:river>
  <lr:river>
    <lr:name>Yangtze</lr:name>
    <lr:length>3917</lr:length>
```

Example

xml_create_knights_elementmaker_ns.py

```
'''Create, print, and save a new XML document'''
from lxml.builder import ElementMaker  # import ElementMaker
import lxml.etree as ET

FILE_NAME = 'knights.xml'
NAMESPACE_URL = 'http://www.cja-tech.com/knights'  # URL (fake) for namespace

E = ElementMaker(  # create ElementMaker with namespace kt
    namespace=NAMESPACE_URL,
    nsmap={'kt': NAMESPACE_URL},
)


def main():
    '''Program entry point'''
    knight_info = get_knight_info()
    knight_root = build_tree(knight_info)
    knight_doc = ET.ElementTree(knight_root)
    write_doc(knight_doc)


def get_knight_info():
    '''Read knight data from the file'''
    info = []
    with open('../DATA/knights.txt') as kn:
        for line in kn:
            flds = line[:-1].split(':')
            info.append(flds)
    return info


def build_tree(knight_recs):
    '''Build the new XML document'''
    KNIGHTS = E.knights  # use element maker to make Knight elements
    KNIGHT = E.knight
    TITLE = E.title
    COLOR = E.color
    QUEST = E.quest
    COMMENT = E.comment

    knights_list = [
        KNIGHT(
            TITLE(kr[1]),
```

```
            COLOR(kr[2]),
            QUEST(kr[3]),
            COMMENT(kr[4]),
        ) for kr in knight_recs
    ]  # use a list comprehension to create list of <knight> elements
    knights = KNIGHTS(  # pass all knight elements to <knights> element
        *knights_list
    )
    return knights


def write_doc(doc):
    '''Write the pretty-printed XML document out to a file'''
    doc.write(FILE_NAME, pretty_print=True)


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

XML Schemas

- XML metadata
- Define domain-specific subset of XML
- Use to validate or generate XML
- Three primary schema languages – Document Type Definition (DTD) – XML Schema Definition (XSD) – Relax NG

While XML is a free-form, extensible language (hence the name), if two companies wanted to share invoice information, they couldn't just each separately make up the tags and attributes they would use. They would need to agree to use the same set of tags and attributes.

This is the purpose of an **XML schema** – to formalize a set of XML tags so that two or more parties can easily share information using the same XML layout.

There are three primary schema languages.

The first is **DTD**. It has been around the longest, but is not as much used currently. It uses its own syntax to describe allowable element tags, element attributes, and value types.

The second schema language is **XSD**. This was the most popular format for a long time, and is itself written in XML, so no separate parser is needed. XSD adds namespaces, which solve the problem of multiple schemas using the same element name.

The third language is **Relax NG**, which stands for "REgular LAnguage for XML Next Generation". It is easier to write, and great if you just need validation, but not as good for data binding or converting objects to or from XML.

This section will focus on XSD.

Schema validation

- Use a validating parser
- Load schema into XML Schema object
- Create XMLParser object with schema
- Use etree.parse() with parser
- Raises error on invalid XML

To validate an XML document with a schema, first create an `XMLSchema` object. You can pass the `XMLSchema` constructor either a file (ending in .xsd) or an already-parsed schema document.

Then create an `XMLParser` object, passing in the schema object with the `schema=` parameter.

Finally, parse the XML document as usual with etree.parse(), but pass in the parser with the `parser=` parameter.

If the XML is valid, `parse()` will return a normal `ElementTree` object. If the XML is invalid, it will raise `etree.ParseError` with a message describing where the XML document is invalid.

Example

xml_validate_schema.py

```
#
import os
from lxml import etree

XML_BASE = '../DATA' # set global variables for file locations
PRES_SCHEMA_PATH = os.path.join(XML_BASE, 'presidents.xsd')
PRES_XML_PATH = os.path.join(XML_BASE, 'presidents.xml')
BAD_PRES_XML_PATH = os.path.join(XML_BASE, 'presidents_bad.xml')

pres_schema = etree.XMLSchema(file=PRES_SCHEMA_PATH) # create schema from schema (.xsd)
file
pres_parser = etree.XMLParser(schema=pres_schema) # create XML parser that uses schema

def validate(xml_path):
    try:
        pres_doc = etree.parse(xml_path, parser=pres_parser) # try to parse XML
    except etree.ParseError as err: # catch error if XML is invalid
        print(f"Error Parsing {xml_path}:")
        print(err)
    else:
        print(f"Parsed {xml_path} OK:")
        print(pres_doc) # XML is valid
    print(('-' * 60))

for xml_doc in PRES_XML_PATH, BAD_PRES_XML_PATH: # iterate over list of (2) XML files and
try to validate
    validate(xml_doc)
```

xml_validate_schema.py

```
Parsed ../DATA/presidents.xml OK:
<lxml.etree._ElementTree object at 0x10999dcc0>
-------------------------------------------------------------
Error Parsing ../DATA/presidents_bad.xml:
Element 'termbegin': This element is not expected. Expected is ( termstart ). (<string>,
line 0)
-------------------------------------------------------------
```

Generating data structures from schemas

- Use generateDS
- Uses schema to generate class factories

To generate data structures (i.e., classes) from schemas, you can use the `generateDS` package. This is a module that can be run as a command line utility.

The simplest syntax is:

```
generateDS.py -o modulename.py schema.xsd
```

Once you have run the command, you can import modulename.py.

The module will provide factories, with which you can populate the fields of the nested data structures programmatically. When you have populated the data structures, you can turn the structure into a string. Then you can write the XML string to a file.

There are many variations on the above sequence.

Example

xml_from_presidents.py

```
from io import StringIO  # import StringIO module which treats a string like a file
object
import presidentsDS  # import module which was created separately from presidents.xsd

# to create presidentsDS module:
# python generateDS.py -o presidentsDS.py presidents.xsd

def main():
    pres_data = get_presidents_data('../DATA/presidents.txt')
    pres_list = build_presidents_xml(pres_data)
    xml_string = build_xml_string(pres_list)
    print(xml_string)
    write_xml_to_file(xml_string, 'presidents_generated.xml')


def get_presidents_data(file_name):
    with open(file_name) as presidents_in:
        records = [line.rstrip('\n\r').split(':') for line in presidents_in]
    return records


def build_presidents_xml(presidents_data):
    pres_list = presidentsDS.presidents.factory()  # use generated factory to create
<presidents> element

    for president_info in presidents_data:
        pres = presidentsDS.presidentType.factory()  # use other factories to create
nested data
        name = presidentsDS.nameType.factory()
        birth = presidentsDS.birthType.factory()
        death = presidentsDS.deathType.factory()
        termstart = presidentsDS.termstartType.factory()
        termend = presidentsDS.termendType.factory()

        name.set_last(president_info[1])  # set values for elements using factories
        name.set_first(president_info[2])
        pres.set_name(name)
        make_date(birth, president_info[3])
        pres.set_birth(birth)
        make_date(death, president_info[4])
        pres.set_death(death)
        pres.set_birthplace(president_info[5])
        pres.set_birthstate(president_info[6])
```

```
        make_date(termstart, president_info[7])
        pres.set_termstart(termstart)
        make_date(termend, president_info[8])
        pres.set_termend(termend)
        pres.set_party(president_info[9])

        pres_list.add_president(pres)

    return pres_list


def make_date(date_obj, date_str):
    """
    Convert date string from file into XML date element

    :param date_obj: date object from date factory
    :param date_str: date string to convert to date object
    :return: date object with data from string
    """
    if date_str == 'NONE':
        year, month, day = ("",) * 3
    else:
        year, month, day = date_str.split('-')
    date_obj.set_year(year)
    date_obj.set_month(month)
    date_obj.set_day(day)


def build_xml_string(pres_list):
    xml_str = StringIO()  # create StringIO() object
    pres_list.export(xml_str, 0)  # write XML into string object (like writing to file)
    return xml_str.getvalue()


def write_xml_to_file(xml, file_name):
    with open(file_name, 'w') as pres_out:
        pres_out.write(xml)  # save XML as file


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

xml_from_presidents.py

```
<presidents>
    <president>
        <name>
            <last>Washington</last>
            <first>George</first>
        </name>
        <termstart>
            <year>1789</year>
            <month>04</month>
            <day>30</day>
        </termstart>
        <termend>
            <year>1797</year>
            <month>03</month>
            <day>04</day>
        </termend>
        <birthplace>Westmoreland County</birthplace>
        <birthstate>Virginia</birthstate>
        <birth>
            <year>1732</year>
```

Generating XML from data structures

- Use package dicttoxml
- Turns dicts and other objects into XML
- Many ways to customize

The `dicttoxml` package will convert a Python data structure into XML. There are many ways to customize it. By default each value with be enclosed in an item tag, and will have type attributes. You can change this to give more "normal" looking XML.

Example

xml_rivers_to_xml_default.py

```
#
import csv
from dicttoxml import dicttoxml # import dicttoxml module
import lxml.etree as ET

rivers = {'rivers': []} # create dictionary where key is tag and value is empty list
with open('../DATA/longest_rivers.csv') as rivers_in:
    rdr = csv.reader(rivers_in)
    for row in rdr: # for each river in source file, add dictionary with river into to
main dictionary
        rivers['rivers'].append({'name': row[0], 'length': row[1]})

snippet = dicttoxml(rivers)  # convert dictionary to XML string

river_root = ET.fromstring(snippet) # create lxml element tree from snippet string
river_doc = ET.ElementTree(river_root) # create lxml doc from element tree

print(ET.tostring(river_doc, pretty_print=True, xml_declaration=True).decode())  # print
XML

river_doc.write('longest_rivers_default.xml', pretty_print=True, xml_declaration=True) #
save XML to file
```

xml_rivers_to_xml_default.py

```
<?xml version='1.0' encoding='ASCII'?>
<root>
  <rivers type="list">
    <item type="dict">
      <name type="str">Nile</name>
      <length type="str">4132</length>
    </item>
    <item type="dict">
      <name type="str">Amazon River</name>
      <length type="str">3976</length>
    </item>
    <item type="dict">
      <name type="str">Yangtze</name>
      <length type="str">3917</length>
    </item>
```

...

```
    <item type="dict">
      <name type="str">R&#237;o de la Plata</name>
      <length type="str">3030</length>
    </item>
    <item type="dict">
      <name type="str">Congo River</name>
      <length type="str">2922</length>
    </item>
    <item type="dict">
      <name type="str">Amur River</name>
      <length type="str">2763</length>
    </item>
  </rivers>
</root>
```

Example

xml_rivers_to_xml.py

```
import csv
from dicttoxml import dicttoxml  # import dicttoxml module
import lxml.etree as ET
from pprint import pprint

rivers = {'rivers': []}  # create dictionary where key is tag and value is empty list

with open('../DATA/longest_rivers.csv') as rivers_in:
    rdr = csv.reader(rivers_in)
    for name, length in rdr:
        rivers['rivers'].append({'name': name, 'length': length})  # append sub-
dictionary to dictionary tag element

pprint(rivers)
print('-' * 60)

snippet = dicttoxml(rivers, attr_type=False, root=False,
                    item_func=lambda x: x.rstrip('s'))  # create XML text snippet from
dictionary

river_root = ET.fromstring(snippet)  # Parse XML text into tree of Elements
river_doc = ET.ElementTree(river_root)  # create ElementTree object from Element tree

print(ET.tostring(river_doc, pretty_print=True, xml_declaration=True).decode())  #
convert XML doc to string (should be same as earlier snippet from dicttoxml()

river_doc.write('longest_rivers.xml', pretty_print=True,
                xml_declaration=True)  # write XML doc out to file
```

xml_rivers_to_xml.py

```
{'rivers': [{'length': '4132', 'name': 'Nile'},
            {'length': '3976', 'name': 'Amazon River'},
            {'length': '3917', 'name': 'Yangtze'},
            {'length': '3902', 'name': 'Mississippi River'},
            {'length': '3445', 'name': 'Yenisei River'},
            {'length': '3395', 'name': 'Yellow River'},
            {'length': '3364', 'name': 'Ob River'},
            {'length': '3030', 'name': 'Río de la Plata'},
            {'length': '2922', 'name': 'Congo River'},
            {'length': '2763', 'name': 'Amur River'}]}
-------------------------------------------------------------
<?xml version='1.0' encoding='ASCII'?>
<rivers>
  <river>
    <name>Nile</name>
    <length>4132</length>
  </river>
  <river>
    <name>Amazon River</name>
    <length>3976</length>
```

...

```
{'rivers': [{'length': '4132', 'name': 'Nile'},
            {'length': '3976', 'name': 'Amazon River'},
            {'length': '3917', 'name': 'Yangtze'},
            {'length': '3902', 'name': 'Mississippi River'},
            {'length': '3445', 'name': 'Yenisei River'},
            {'length': '3395', 'name': 'Yellow River'},
            {'length': '3364', 'name': 'Ob River'},
            {'length': '3030', 'name': 'Río de la Plata'},
            {'length': '2922', 'name': 'Congo River'},
            {'length': '2763', 'name': 'Amur River'}]}
-------------------------------------------------------------
<?xml version='1.0' encoding='ASCII'?>
```

Chapter 3 Exercises

Exercise 3-1 (presidents_factory.py)

Using either `presidents.txt` or `presidents.csv` for input, use the `lxml` E-Factory to create an XML file from the data. Name the output file `potus.xml`.

Index

