

Software Design for Data Science

Modules, Packages, and Imports

*Melissa Winstanley
University of Washington
January 31, 2023*

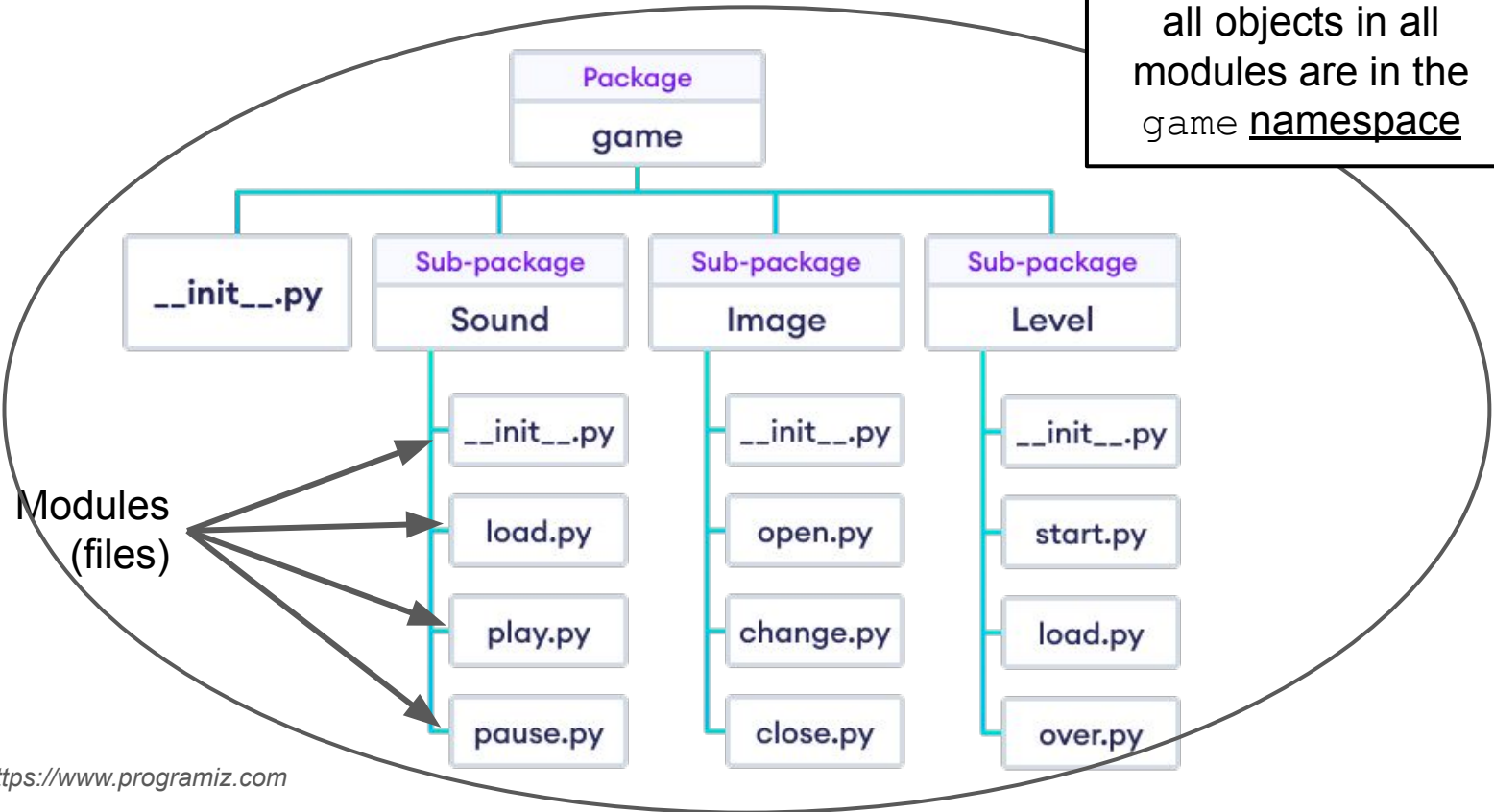


Definitions

- **Object:** Most things in Python, e.g. function, variable, class.
- **Module:** A *.py script; carries the name as the file.
- **Namespace:** A mapping of unique names to objects.
- **Package:** A directory-like concept that can hold multiple Python objects, modules, or subpackages under same namespace.

Definitions: Example

when running
`import game`
all objects in all
modules are in the
game namespace



Imports

Imports

There are 4 different syntaxes for writing import statements.

1. `import <package>`
 - a. **Brings** <package> into the current namespace
2. `import <module>`
 - a. **Brings** <module> into the current namespace
3. `from <package> import <module or subpackage or object>`
 - a. **Brings** <module or subpackage or object> into the current namespace
4. `from <module> import <object>`
 - a. **Brings** <object> into the current namespace

Using imports

Let `x` be whatever name comes after `import`.

- If `x` is the name of a module or package, then to use objects defined in `x`, you have to write `x.object`

```
# for these cases
```

```
import <package>
```

```
import <module>
```

```
from <package> import <module or subpackage>
```

- If `x` is a variable name, then it can be used directly.
- If `x` is a function name, then it can be invoked with `x()`

```
from <package> import <object>
```

```
from <module> import <object>
```

Multiple imports

```
from <module> import <object1>, <object2>, <object3>
```

OR

```
from <module> import (  
    <object1>,  
    <object2>,  
    <object3>,  
)
```

Import as

Optionally, `as Y` can be added after any `import X` statement:

```
import X as Y
```

```
from X import (  
    foo as Y,  
    bar as Z,  
)
```


Import order of search

How does Python find modules/packages to import?

1. Python Standard Library packages/modules (eg `math`, `sys`)
2. Packages/modules in `sys.path`:
 - a. If run with the Python interpreter (ie just `python`)
 - i. `sys.path[0]` is the empty string `''`, which means the current working directory
 - b. If run with `python <module.py>`:
 - i. `sys.path[0]` is the path to `<module>.py`
 - c. Directories in `PYTHONPATH` environment variable
 - d. Default `sys.path` locations - dependent on Python installation

Absolute vs relative imports

Absolute Imports (works in Python 2 and 3):

Import from the top-level Python package. e.g., `import <package>`

Relative Imports:

Import based on your `sys.path` location.

Explicit Relative Imports (works in Python 2 and 3):

Import using `.` (current directory) and `..` (parent directory) notation

Implicit Relative Imports (only Python 2):

Discontinued

Wildcard imports

Bringing EVERYTHING from a package or module into the current namespace

```
from <package> import *
```

BAD BAD BAD! Why?

- Conflicts between names defined locally and the ones imported
- Reduces code readability - where did a name come from?
- Clutters the local namespace, which makes debugging more difficult

Example in the materials for today

Directory Structure

Basic codebase in a git repository

```
myproject/  
  README.md  
  LICENSE  
  myproject/  
    __init__.py  
    core.py  
    subpackage/  
      __init__.py  
      subpackage.py  
    submodule/  
      submodule.py  
  tests/  
    __init__.py  
    test_core.py
```

Basic codebase in a git repository

myproject/



This is your git repository, usually matching the name on Github

Basic codebase in a git repository

myproject/

README.md



Markdown-formatted or plain text file describing the package

Basic codebase in a git repository

myproject/
 README.md
 LICENSE



Software license specifying how others may use
your code

Basic codebase in a git repository

```
myproject/  
  README.md  
  LICENSE  
myproject/
```

The Python package, helpful for
`import myproject` into separate namespaces.

Basic codebase in a git repository


```
myproject/  
  README.md  
  LICENSE  
  myproject/  
    __init__.py
```

← This module marks the directory as a Python package and is run upon import

Basic codebase in a git repository

```
myproject/  
  README.md  
  LICENSE  
  myproject/  
    __init__.py  
    core.py
```

Other modules in the package containing
importable code.



Basic codebase in a git repository

```
myproject/  
  README.md  
  LICENSE  
  myproject/  
    __init__.py  
    core.py  
    subpackage/  
      __init__.py  
      subpackage.py
```

Packages can have subpackages (and sub-subpackages, etc.) to any depth. They contain their own `__init__.py` files.



Basic codebase in a git repository

```
myproject/  
  README.md  
  LICENSE  
  myproject/  
    __init__.py  
    core.py  
    subpackage/  
      __init__.py  
      subpackage.py  
  submodule/  
    submodule.py
```

Packages can also have modules to any depth - ie without the `__init__.py`, this directory is just a collection of modules.

Basic codebase in a git repository

```
myproject/  
  README.md  
  LICENSE  
  myproject/  
    __init__.py  
    core.py  
    subpackage/  
      __init__.py  
      subpackage.py  
    submodule/  
      submodule.py  
  tests/  
    __init__.py ← Unit tests go into their own subpackage  
    test_core.py
```

Basic codebase in a git repository

```
myproject/  
  README.md  
  LICENSE  
  myproject/  
    __init__.py  
    core.py  
    subpackage/  
      __init__.py  
      subpackage.py  
    submodule/  
      submodule.py  
  tests/  
    __init__.py  
    test_core.py
```

But wait - what actually goes in `__init__.py`?

1. `__init__.py` is empty

Example: [urllib](#)

- Pros
 - Simple
- Cons
 - You don't get a *namespace*
 - You have to specify a specific submodule to import
 - Importing just the package is useless
 - Important to have good module names

Eg:

```
import urllib.parse
urllib.parse.urlparse("https://google.com")
```

Not

```
import urllib
urllib.urlparse("https://google.com")
```

2. `__init__.py` imports from subpackages/modules

Import from subpackages and submodules

Determine order of import (where that's important)

Example: [tomllib](#) (library that parses TOML files)

- Pros
 - Can create a helpful namespace
 - Interface of your package is more clear
- Cons
 - Interface of package is not super clear

3. `__init__.py` defines the interface for the package

Define the interface for the package

Importing/piecing together submodules

Example: [json](#)


- Pros
 - Great for someone reading your package to know where to start
 - Functionality is clear
- Cons
 - What goes in `__init__.py` vs submodule?

4. `__init__.py` defines everything




Example: [collections](#)


- Pros
 - One single file
 - Can work well for small packages
- Cons
 - Can be unwieldy if package is big





Distributing a Python package




[Pull requests](#) [Issues](#) [Codespaces](#) [Marketplace](#) [Explore](#)


  

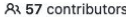

 **pandas-dev / pandas** Public






   


[Code](#) [Issues 3.6k](#) [Pull requests 160](#) [Actions](#) [Projects 1](#) [Security](#) [Insights](#)

 **pandas** / README.md [Go to file](#) [...](#)

 **MarcoGorelli** CI remove scorecards (#50269) ... [X](#) Latest commit f759c33 on Dec 15, 2022 [History](#)

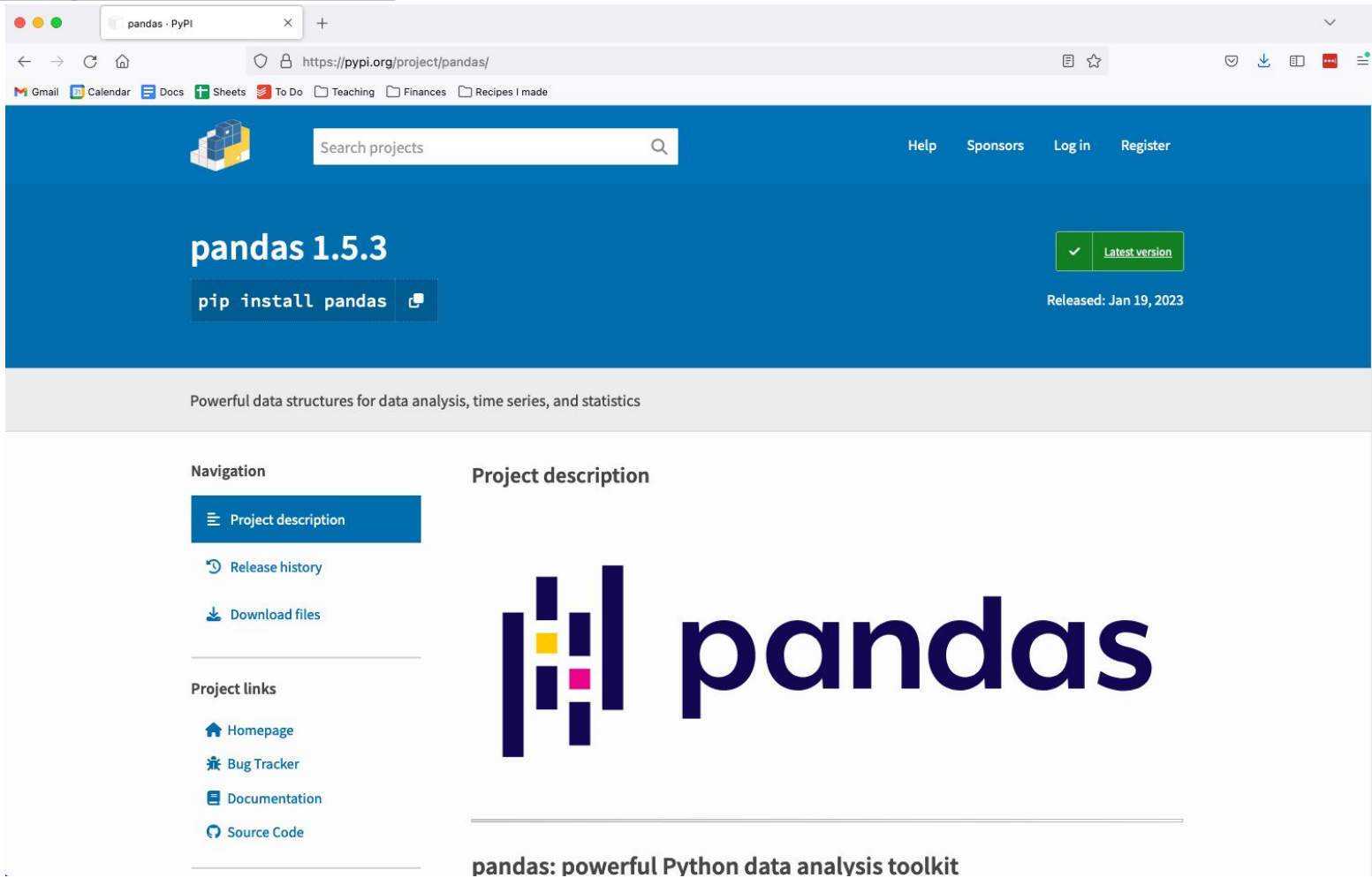
 57 contributors  +39

 170 lines (132 sloc) | 10.1 KB   [Raw](#) [Blame](#)  



pandas: powerful Python data analysis toolkit

PyPI



The screenshot shows the pandas project page on the PyPI website. The browser's address bar displays `https://pypi.org/project/pandas/`. The page features a blue header with the pandas logo, a search bar, and navigation links for Help, Sponsors, Log in, and Register. The main content area highlights the current version, pandas 1.5.3, as the latest version, with a green checkmark and a 'Latest version' button. Below this, a button for 'pip install pandas' is visible. The release date is noted as 'Released: Jan 19, 2023'. A grey banner below the header states 'Powerful data structures for data analysis, time series, and statistics'. The left sidebar contains a 'Navigation' section with links to 'Project description' (selected), 'Release history', and 'Download files'. Below this is a 'Project links' section with links to 'Homepage', 'Bug Tracker', 'Documentation', and 'Source Code'. The main content area is titled 'Project description' and features the pandas logo, which consists of a stylized 'p' made of vertical bars of varying heights and colors (blue, yellow, pink), followed by the word 'pandas' in a large, bold, dark blue font. At the bottom of the page, a tagline reads 'pandas: powerful Python data analysis toolkit'.

pandas - PyPI

https://pypi.org/project/pandas/

Gmail Calendar Docs Sheets To Do Teaching Finances Recipes I made

Search projects

Help Sponsors Log in Register

pandas 1.5.3

✓ Latest version

Released: Jan 19, 2023

`pip install pandas`

Powerful data structures for data analysis, time series, and statistics


Navigation

- Project description
- Release history
- Download files

Project links

- Homepage
- Bug Tracker
- Documentation
- Source Code

Project description



pandas

pandas: powerful Python data analysis toolkit

Extending project structure for PyPI

```
myproject/  
  README.md  
  LICENSE  
  myproject/  
    __init__.py  
    setup.py  
    core.py  
    subpackage/  
    submodule/  
    tests/
```



Contains metadata for the package for PyPI.
Often uses `distutils` or `setuptools`
standards.
Can contain abstract vital dependencies

Extending project structure for PyPI

```
myproject/  
  README.md  
  LICENSE  
  myproject/  
    __init__.py  
    setup.py  
    requirements.txt  
    core.py  
    subpackage/  
    submodule/  
    tests/
```

Contains absolute dependencies, especially useful if you're publishing an application.

Can be generated using:

```
conda list --export > requirements.txt
```

OR

```
pip freeze > requirements.txt
```

Extending project structure for PyPI

```
myproject/
```

```
    README.md
```

```
    LICENSE
```

```
    myproject/
```

```
        __init__.py
```

```
        setup.py
```

```
        requirements.txt
```

```
        MANIFEST.in
```


```
        core.py
```

```
        subpackage/
```

```
        submodule/
```

```
        tests/
```

Specify data and files that should also be packaged in addition to the Python modules



Example setup.py

```
import setuptools

setuptools.setup(
    name="example-pkg-your-username",
    version="0.0.1",
    author="Example Author",
    author_email="author@example.com",
    description="A small example package",
    install_requires=['docutils>=0.3'],
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/pypa/sampleproject",
    packages=setuptools.find_packages(),
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
)
```

More examples

Check out existing Python packages. These may be more complicated:

<https://github.com/numpy/numpy>

<https://github.com/pandas-dev/pandas>

Submitting your package to PyPI

Update your code and version number. Run your test suites and ensure your code works as intended. Create a PyPI account if you don't have one already

Create your source, and if desired, binary distribution:

```
$ python setup.py bdist egg upload [options]  
$ python setup.py bdist wininst [options]  
$ python setup.py sdist [options]
```

Install `twine` package to submit builds to PyPI.

(Can install using `conda install twine`, `pip install twine`, etc.)

```
$ twine upload dist/*
```