

Software Design for Data Science

Modules, Packages, and Imports

*Melissa Winstanley
University of Washington
January 25, 2024*

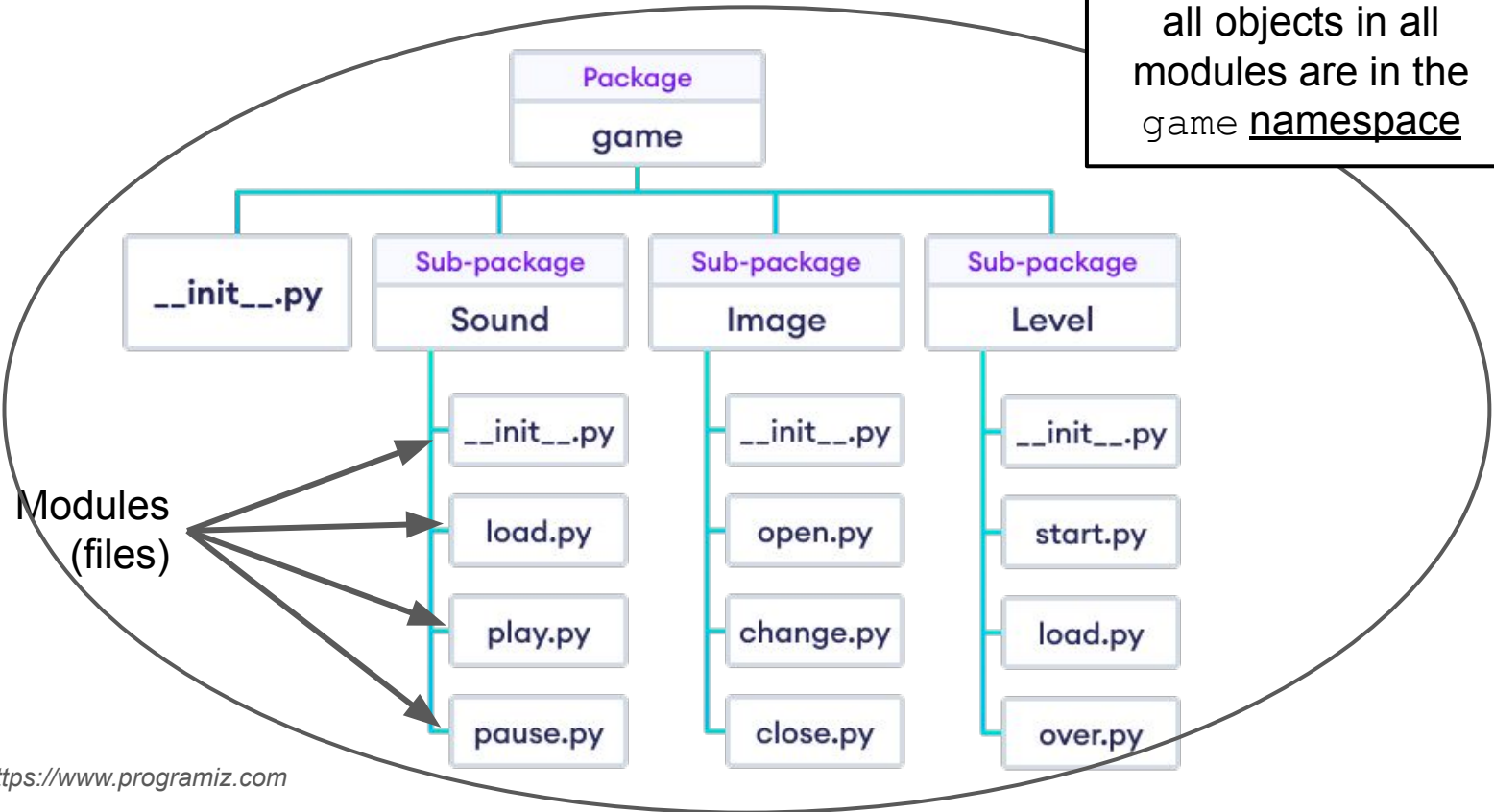


Definitions

- **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  
  test_core.py
```

← Unit tests go into their own subpackage

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

Exercise: Make it work

1. Go to the course website and download the “imports” folder from the lecture materials for week 4. This uses code from last week.
2. Update the imports so that you can run both `main.py` and `run/runner.py`

Hint: there are multiple ways to run python code. We've seen

```
python path/to/file.py
```

But you can also run a module with:

```
python -m path.to.module
```

What's the difference? The `-m` version keeps the path as the current directory! Try it out and print `sys.path`