

COMP(2041|9044) 25T2 — Python Modules

<https://www.cse.unsw.edu.au/~cs2041/25T2/>

Importing Code

```
>>> import math
>>> math.log(math.e)
1.0
>>> dir(math) # a module is itself an object
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
↪ 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil',
↪ 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc',
↪ 'exp', 'exp2', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
↪ 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
↪ 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
↪ 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
↪ 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
↪ 'trunc', 'ulp']
>>> help(math) # generated from docstrings
NAME
    math
DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.
```

Importing Code

Python module - a file containing function definitions and other Python.

```
import math                                # access names from math as math.name, e.g
    ↪ math.sin
from math import sin                       # access math.sin as sin
from math import sin as sine              # access math.sin as sine
from math import *                         # access all names from math without prefix
    ↪ (avoid)
import math as m                          # access names from math as m.name, e.g m.sin
```

Finding Modules

Python module - a file containing function definitions and other Python.

import searches the current directory and a series of standard directories and zip files for modules. **sys.path** contains the list (you can append directories you also want searched).

```
>>> import sys
>>> sys.path
['', '/usr/lib/python39.zip', '/usr/lib/python3.9',
↪  '/usr/lib/python3.9/lib-dynload',
'/usr/local/lib/python3.9/dist-packages', '/usr/lib/python3/dist-packages',
'/usr/lib/python3.9/dist-packages']
```

environment variable **PYTHONPATH** added to **sys.path**

```
$ PYTHONPATH=/home/z1234657/modules python3
...
>>> sys.path
['', '/home/z1234657/modules', '/usr/lib/python311.zip',
↪  '/usr/lib/python3.11', ...]
```

Namespaces

Python modules prevent accidental name collision when using modules from many sources

Python modules can control what names are exported by default (*)

Beware - Python does not prevent deliberate access or changes to any part of a module. Even internal names (not exported) can be changed.

```
>>> import circle
>>> circle.area(radius=2)
12.566370614359172
>>> import math
>>> math.pi = 4
>>> circle.area(radius=2)
16
```

Standard Modules

Python has over 200 standard modules available via an import statement.

We have already used:

```
import os
import re
import sys
import subprocess
```

Non-Standard Modules

Before writing code, look for existing code.

There are over 500,000 packages available on PyPI.

PyPI is the [Py]thon [P]ackage [I]ndex

PyPI is a website that allows you to search for and register your own packages.

Any packages listed on the index can be installed via `pip`.

Packages

A package is a collection of files.

These files contains the source code of, and installation instructions for, one (or more) modules.

The most common format for python packages is called a wheel.

A wheel is basically just a .zip file that contains files in a specially crafted format.

Most of the time you don't need to worry about wheels (or other package types) as they are automatically downloaded and installed.

pip is the standard package manager for Python.

pip stands for [P]ip [I]nstalls [P]ackages.

pip can install any package on PyPI (and be configured to also search other repositories)

To install a package, you can use the following command:

```
$ pip3 install <package_name>
```

or

```
$ python3 -m pip install <package_name>
```

You can also install a package from a local directory

```
$ pip3 install <package>.whl
```

or from git:

```
$ pip3 install git+<package_url>
```

pip also updates packages

```
$ pip3 install --upgrade <package_name>
```

and uninstalls packages

```
$ pip3 uninstall <package_name>
```

By default Python installs packages system-wide, which even on a single-user system can create conflicts: Project A needs version X of a package and project B needs version Y of a package.

A virtual environment allows package to be installed just for the project using them.

In python a virtual environment is a directory that contains a copy of the Python interpreter.

It can be used to isolate your project from the rest of the system.

```
# create a virtual environment:  
$ python3 -m venv <new_directory_name>  
# then activate it  
$ . <new_directory_name>/bin/activate  
# or on Windows:  
$ <new_directory_name>/Scripts/activate
```

Once activated, the python, python3, pip, etc commands will be run from within the virtual environment.

versioning

Python packages should be versioned using PEP440.

The full syntax for a PEP440 version is:

```
[N! ]N( .N) * [ {a | b | rc} N ] [ .postN ] [ .devN ]
```

most commonly only the `N( .N) *` part is used.

This defines a version of format `X.Y.Z`

Eg:

```
1.0.0  
2.0  
3.9.2  
4.2  
7.4.67.3.32
```

This is called a `final release`

versioning

It is most common to use three numbers `major.minor.micro`

Where:

the `major` version is incremented when there is a forward incompatible change.

the `minor` version is incremented when there is a backward incompatible change.

the `micro` version is incremented when there is a non-breaking change (eg bug fix).

If any number isn't specified, it is assumed to be 0.

Eg, all the following are the same:

```
5.7
5.7.0
5.7.0.0
5.7.0.0.0
# etc
```

version specifiers

version specifiers determine which version of a package to use.
without a version specifier, any version (usually the latest) is used.

An exact version is specified by using the `==` operator.

A minimum version is specified by using the `>=` operator (or exclusively `>`).

A maximum version is specified by using the `<=` operator (or exclusively `<`).

A excluded version is specified by using the `!=` operator.

A strict version is specified by using the `===` operator.

A compatible version is specified by using the `~=` operator.

version specifiers == vs ===

== is used to specify an exact version.

=== is used to specify a strict version.

=== is essentially a string comparison.

whereas == takes into account semantic information.

```
1.0 == 1.0.0 # True
```

```
1.0 === 1.0.0 # False
```

version specifiers `~=`

`~=` is used to specify a compatible version.

a compatible version of `X.Y` is `>= X.Y, == X.*`

That is: the minor version is greater than or equal, and the major version is the same.

version specifiers

multiple version specifiers can be used to restrict the version of a package.

```
~= 3.1.0, < 3.1.7, != 3.1.3
```

```
3.1.0
```

```
3.1.1
```

```
3.1.2
```

```
3.1.4
```

```
3.1.5
```

```
3.1.6
```

version specifiers

version specifiers can be used with pip.

```
$ python3 -m pip install 'regex~=2022.7.0,>2022.7.23,!=2022.7.24'
```

by default, pip will install the latest version of a package.

It can be very annoying to keep track of all the packages you need to install.

So instead, we can put them in a file, conventionally called `requirements.txt`.

The `requirements.txt` file is a simple text file that contains a list of package to install.

These can either not have a version specifier.

le. just a list of package.

Or they can have a version specifiers.

le. when you want to replicate an environment.

```
requests  
beautifulsoup4  
regex
```

```
requests ~= 2.0.0, <= 2.25, != 2.26.0, != 2.27.1  
beautifulsoup4 >= 5.4, < 5.10  
regex ~= 2022.7.0, > 2022.7.23, != 2022.7.24
```

requirements.txt

```
# `pip` can install package from a `requirements.txt` file directly.  
$ pip3 install -r requirements.txt  
  
# `pip` can generate a `requirements.txt` file with version specifiers.  
$ pip3 freeze > requirements.txt
```

`pip freeze` gives you a list of *all* packages and their versions.

Even those that were not directly installed (indirect requirements).

This can clutter up your `requirements.txt` file.

So `pip` also supports a `constraints` file.

constraints.txt

constraints.txt works exactly like requirements.txt
except that a package in constraints.txt will only be installed if they are also in requirements.txt.

```
$ pip3 install -r requirements.txt -c constraints.txt
$ cat requirements.txt
requests
beautifulsoup4
regex
$ cat constraints.txt
beautifulsoup4==4.11.1
certifi==2022.6.15
charset-normalizer==2.1.0
idna==3.3
regex==2022.7.25
requests==2.28.1
soupsieve==2.3.2.post1
urllib3==1.26.11
```