The Python Ecosystem

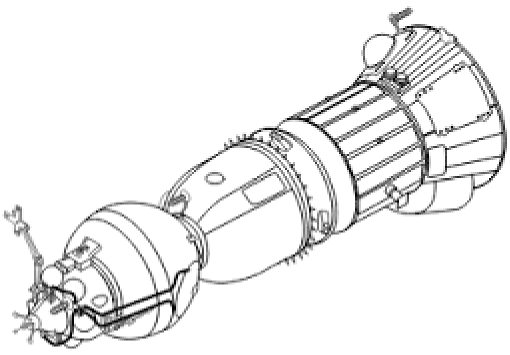
Overview

- Modules
- Namespaces
- Packages
- Virtual Environments

Learning Objectives

- Detail the key features of modules
- Define how packages work
- Identify how scope and namespaces work
- Discover the value of libraries and how to install them
- Define virtual environments
- Implement a virtual environment.

Modules



Modules

Currently you've been writing code inside one file.

As the amount of code in your python file gets longer, it may start to become unmaintainable.

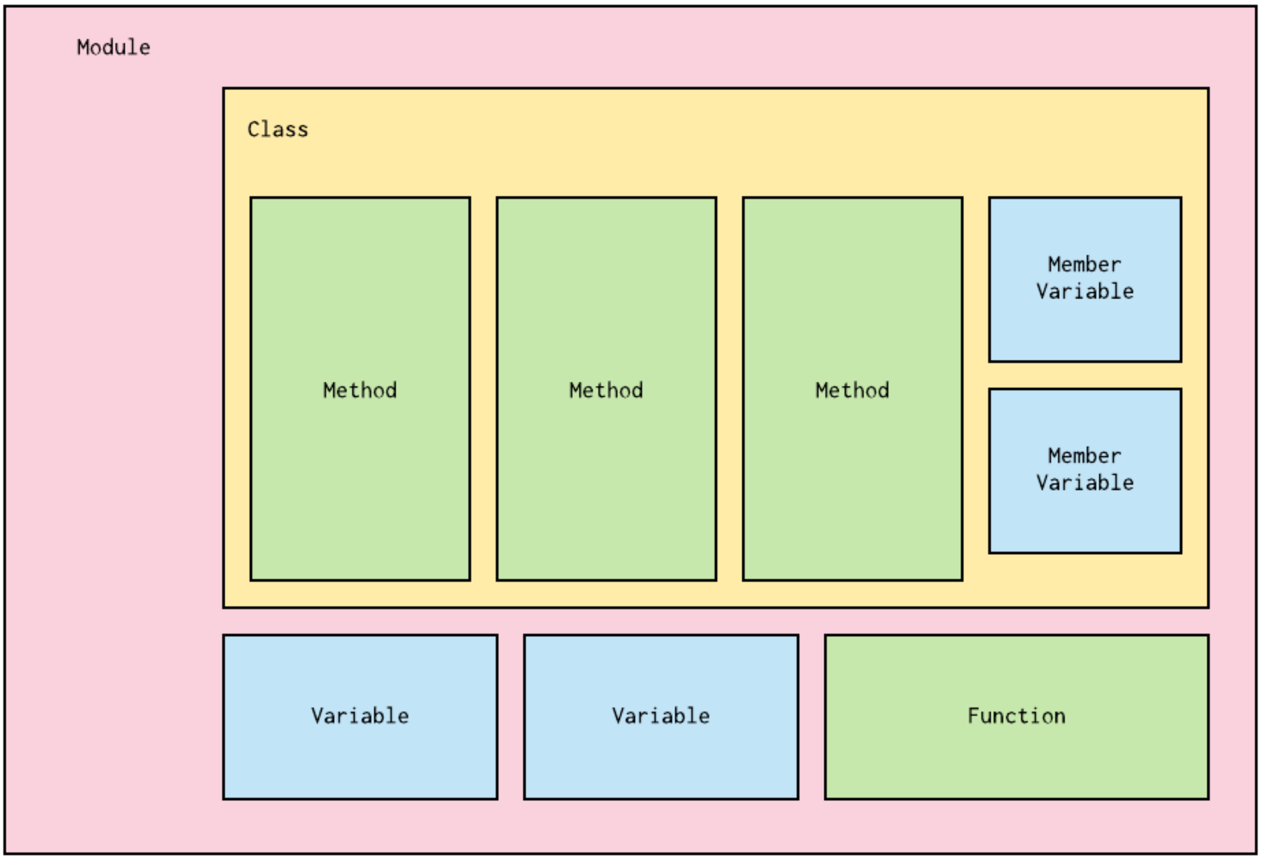
To alleviate this, you can split the code into many files.

These files are called **modules**.

Let the learners know that they've already been using modules without realising, as every python file is considered a module.

Modules

A module is a self-contained collection of functions, variables and classes which are available under a **namespace**.



Namespace is mentioned but not defined but will be further down.

Member variable: A variable that is associated with a specific object and accessible for all its methods (member functions).

Talk through the diagram.

Example 1

Use the **import** keyword to use *module1* inside *module2*:

```
# module1.py
def print_name(name):
  print(name)
```

```
#module2.py
import module1
module1.print_name("Jane")
```

This is known as bringing it into scope.

Example 2

We can also import only what we need from a module:

```
# module1.py
def print_name(name):
  print(name)

def print_age(age):
  print(age)
```

```
#module2.py
from module1 import print_name
module1.print_name("Jane")
```

This time, the import statement has changed to:

```
from module_name import something
```

Let the learners see that you can import variables and classes as well as functions

---

Referencing Modules

Global: Modules in the same directory, the standard library or 3rd party library

```
import module_in_the_same_folder  # Same library
import sys                        # Standard library
import numpy                      # 3rd party library
```

Referencing Modules

Absolute: Module path always starts from the top of the directory

```
├── package1
│   ├── module1.py
│   └── module2.py
└── package2
    ├── module3.py
    ├── module4.py
```

```
    └── subpackage1
        └── module5.py
```

```python
from package1 import module1
from package1.module2 import function1
from package2 import class1
```

The imports are all different examples of how you could import into another module

---

Referencing Modules

Relative: Module path is relative to the file doing the import. It isn't good practise to do this.

```python
from .some_module import some_class
from ..some_package import some_function
from . import some_class
```

- A single dot means the module/package referenced is in the same directory
- Two dots mean that is in the parent directory

You can keep going up the filesystem hierarchy ../../..

You shouldn't do it as relative imports are messy. You're like to change directory structure.

Relative imports also aren't that readable.

Quiz Time! 🤓

Given the below project structure, how would you import the function `my_func` from `module_3` into `module_1`?

```
├── package1
│   └── module1.py
└── package2
    ├── module2.py
    └── subpackage1
        └── module3.py
```

1. `from module1 import my_func`
2. `from .package2.subpackage1.module3 import my_func`
3. `from package2.subpackage1.module3 import my_func`
4. `import my_func`

Answer: `3`

The correctness of this example is contingent on some context: If we run `module1.py` directly it will not know about `package2` as it won't be on the path.

Show worked example with app.py at same level as `package1` directory

---

Namespace

A namespace is a system to make sure all of the names (variables, functions etc.) in a program are unique and can be used without conflict.

In Python, when you create a statement such as...

```
x = 1
```

...a **symbolic name** is created that you can use to reference that object.

A variable in Python is a symbolic name which is a pointer to the object. You can access the object through the variable, but the data itself still exists inside the original object.

You can have multiple different symbolic names that point to the same object.

Examples

Local Namespace: Includes local names inside of a function. Created when the function is called and is destroyed when it returns.

Global Namespace: Includes names from various imported modules. Created when the module starts and is destroyed when it ends.

Built-In Namespace: Includes built-in functions and exception names.

Variable Scope

A scope is a portion of a program where a namespace can be accessed

```
def outer_function():
  b = 20
  def inner_func():
    c = 30

a = 10
```

- `a` is in the global namespace
- `b` is in the local namespace of `outer_function`

- `c` is in the local namespace of `inner_function`

Variable Scope

```
def outer_function():
  b = 20
  def inner_func():
    c = 30

a = 10
```

Inside `inner_function`, we can read and assign a new value to c, but we can only read a and b.

If we try to assign a value to `b`, it will create a new locally scoped variable which is different from the variable `b` in `outer_function`

And likewise with outer_scope trying to assign to c.

Quiz Time! 🤓

What will the two print statements output?

```
x = 300

def myfunc():
  x = 200
  print(x) # 1

myfunc()
print(x) # 2
```

1. `200, 300`
2. `300, 200`
3. `200, 200`
4. `300, 300`

Answer: `1`

If we assign to a variable inside a function, it will create a NEW local-scoped variable and will no longer refer to the same underlying object. e.g:

```
a = ["A","B"]

def func():
    a.append("C")
```

```
print(a)
func()
print(a)
```

```
a = ["A","B"]

def func():
    a = ["A","B","C"]

print(a)
func()
print(a)
```

```
a = ["A","B"]

def func():
    print(a) # Error - Not yet defined at this point
    a = ["A","B","C"]

print(a)
func()
print(a)
```

The same applies when a variable is passed into a function. The variables refer to the same object, but reassigning inside the fucntion only reassigns the local-scoped variable. The original variable in this case still points to the original object.

Guidance on Best Practice:

- Avoid accessing outer scopes directly, and instead take input as parameters
- Consider`return`-ing a value rather than modifying a reference

Why return rather than modify?

- Software development standard practice.
- A function that returns something has intuitive and unambiguous behaviour which is clearly described by its inputs and outputs

For multiple return values from a function (if necessary) we can use a dict object or some other class (we'll cover classes later)

To modify passed parameters without affecting the original at all, can use copy:

```
import copy

new_list = copy.deepcopy(old_list)
```

Also built-in: `list.copy()` Also built-in: `new_list = list(old_list)`

Packages

Packages

A collection of modules under one directory that are made available under a parent namespace.

```
├── package1
│   ├── module1.py
│   └── module2.py
└── package2
    ├── module3.py
    ├── module4.py
```

Library

Library

- A reusable collection of modules
- Distributed as packages and modules in Python
- Often built around a core goal such as data science, I/O, user interface etc.
- Examples include NumPy, TkInter, matplotlib

Library

The Python standard libraries are pre-installed in every Python runtime, so they're always available to you:

```
import random
# Print a random number between 0 and 100
print(random.randrange(100))
```

Installing Libraries

We can install external libraries with Python's package manager, `pip`:

Unix/MacOS:

```
$ pip install requests
```

Windows:

```
$ py -m pip install requests
```

Show demo with a simple app installing requests module so it can be used.

Q: Where do libraries come from? A: https://pypi.org (most commonly)

Many programming languages have package indexes to make publishing and sharing libraries easier (e.g. nuget for .NET, NPM for Node)

The package index is not a core part of the language itself, but managed by the wider community.

Installing Pip

Check if you have `pip` installed:

Unix/MacOS:

```
$ pip --version
```

Windows:

```
$ py -m pip --version
```

If your machine doesn't have it installed, you can install it here

Stop to let learners try it out

Exercise

Instructor to distribute exercises.

Virtual Environments

Problem

We will often have multiple Python projects on our machine, which may have the same packages installed but with different versions.

If project1 need v1.0 and project2 needs v2.0, then the requirements are in conflict and installing either version will leave one project being unable to run.

How do we fix this? We use **virtual environments**

Virtual Environments

- A self-contained directory that contains a Python installation for a particular version, along with additional packages
- Different applications can create and manage virtual environments (venv)

Virtual Env Installation

To create a virtual environment

```
$ python3 -m venv <path to virtual environment>
```

e.g:

```
$ python3 -m venv .venv
```

This will create a directory called `.venv` if it doesn't exist and creates directories containing a copy of Python, the standard library and other supporting files

.venv is a usual name for the folder, purely by standard convention

If `python3` doesn't work then `python`

Remember that `-m` is used to run a module as an executable script; venv in this case is a built-in module from the python installation which we can use.

Virtual Env Activation

Once created, you can **activate** it:

Windows:

```
$ .venv\Scripts\activate.bat
```

Unix/MacOS:

```
$ source .venv/bin/activate
```

It's important to note that the venv is only active during the lifetime of the shell. When you close the shell it is then terminated.

You can see the venv is activated as you'll have the name of the virtual env on your terminal prompt.

Virtual Env Deactivation

When you need to deactivate the env, it's as simple as:

```
$ deactivate
```

Managing Packages in venv

When you install a package through pip inside a `venv` this will keep the installation local to the project.

You should also store the package dependencies in a file called `requirements.txt`. You can generate and store a list of the dependencies using:

```
(.venv) $ pip freeze > requirements.txt
```

Then anyone else can use this file to install all required dependencies.

requirements.txt serves two requirements:

- Allows depenendencies for our project to be described
- Allows dependencies to be pinned at specific versions

Ask the class: Why may we want to pin versions?

- Avoid breaking changes
- Avoid security vulnerabilities (e.g. supply chain attack) in packages whose previous versions were fine.

---

requirements.txt file

The `requirements.txt` file created may look similar to this:

```
requests==2.7.0
novas==3.1.1.3
numpy==1.9.2
```

To install `requirements.txt` execute the code below in an active `venv`

```
(.venv) $ pip install -r requirements.txt
```

Useful pip Commands

Install a specific package version:

```
(.venv) $ pip install requests=2.6.0
```

Upgrade a package:

```
(.venv) $ pip install --upgrade requests
```

Uninstall a package:

```
(.venv) $ pip uninstall requests
```

List all packages installed in the virtual env:

```
(.venv) $ pip list
```

Quiz Time! 🤓

In Python, a virtual environment is a self-contained directory that contains...

1. A Python installation for a particular version without installing additional packages
2. A Python installation for a particular version, along with additional packages
3. A Python installation for Python 3, along with additional packages
4. A Python packages

Answer: 2

The version of python inside the virtual environment will be the same as the version of python used to create it. To create a venv of a different version, you would need multiple python versions installed and can specify a specific binary.

e.g. `someotherpythonbinary -m venv .venv`

Exercise

1. Create a directory on your system
2. Follow the above slides to setup a virtual environment in that folder
3. Activate the `venv`
4. Try installing some packages and list them
5. Create a `requirements.txt` files from your `venv`
6. When you are done, deactivate

Learning Objectives

- Detail the key features of modules
- Define how packages work
- Identify how scope and namespaces work
- Discover the value of libraries and how to install them
- Define virtual environments
- Implement a virtual environment.

Terms and Definitions Recap

Module: A file which contains a collection of functions, variables and classes, available under the same namespace

Namespace: A system to make sure all names in a program are unique and can be used without conflict

Variable Scope: A portion of a program where a namespace can be accessed

---

Terms and Definitions Recap

Package: A collection of modules under one directory

Library: A reusable collection of modules, distributed as modules or packages

Virtual Environment: A self-contained directory that contains a Python installation for a particular version, along with additional packages

Runtime Library: A program library designed to implement functions built into a programming language

---

Further Reading

Namespaces Further Explained

Virtual Environments

The Python Package Index