Modules and packages

Programmazione Avanzata 2025-26

Goals

- Split a large program in multiple files
- Make re-usable library of classes
- Import and use additional libraries

Modules

- Modules are collections of
  - Classes, functions, variables, and declarations
- The **`import`** statement runs the module source and makes the definitions available
- The defined names are created in a separated namespace to avoid confusion

Modules

- Standard library modules
  - **`random`**, **`math`**, **`csv`**, **`dataclasses`**, … and 200 more
  - https://docs.python.org/3/library/index.html
- User-defined modules
  - Programmer's Python sources
  - Files and directories in programmer's project
  - Files and directories in the Python search path
- Downloaded modules
  - From https://pypi.org/ (over 500k projects)
  - Install using pip command/tool

Creating a Python module

- Programmer just have to create a `.py` file
  - In the same directory of the main file
  - Should contain declarations, only
- The name of the file is the name of the module
  - The argument of `import`
- All names defined at the top-level become visible properties of the module
  - Constants
  - Functions
  - Classes
  - Variables (bad idea)

Creating a Python module

```
STD_COLOR = "White"
class Car:
   def __init__(self):
      self.licensePlate = 0
      self.bodyColor = STD_COLOR
      self.turnedOn = False
   def __str__(self):
      ...
   def __repr__(self):
      ...
```

car.py

```
import car
```

main.py

The import statement

- The instruction `import module_name`
  - Imports the definitions from `module_name`
  - They will be accessible as `module_name.definition`
  - Example:

    After having written `import math`, use `math.sin(math.pi)`
- The instruction `import module_name as alt_name`
  - Imports the definitions from `module_name`
  - They will be accessible as `alt_name.definition`
  - Example
    - After having `written import cmath as c`, use `c.sqrt(-1)`

The from … import statement

- Instruction from **`module_name import name(s)`**
  - Imports one or more name(s) from **`module_name`**, and make them available in the current namespace
  - Example: **`from math import pi, sin, cos`**, then **`sin(pi)`**
- Instruction **`module_name import name as alt_name`**
  - Imports one name from **`module_name`**, and make it available in the current namespace as **`alt_name`**
  - Example: **`from cmath import sqrt as csqrt`**, then **`csqrt(-1)`**
- Instruction **`from module_name import *`**
  - Imports all available names from **`module_name`**, and make them available in the current namespace, except names starting with **_** (underscore), which are ignored
  - Dangerous, may have conflicts with local names or other module's names

Querying available names

- The **`dir()`** function shows the list of names defined in a module
  - **`dir()`**: names defined (at the top level) of the current file
  - **`dir(module_name)`**: names defined in the imported module
- Several dunder methods, plus user-defined names

```
>>> import car
>>> print dir(car)
['Car', 'STD_COLOR', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__’, …]
```

Modules and executable statements

- A module should not contain executable statements
  - The import statement runs the module file to create the definitions of the various names
  - If there are any instructions outside the defined classes/functions, they will be executed, too
- If **`car.py`** contains instructions, like, e.g., **`c = Car()`**, followed by **`print(c)`**
- Then, the instruction **`import car`** in **`main.py`** would cause
  - Defining a new top-level name (**`c`**)
  - Calling the function **`print()`**
  - Printing in the console
  - All this should not happen!

Solving the problem

- It is useful to have some code inside the module
    - Usually, test code to verify that the module works correctly
    - Sometimes, a whole program (with its top-level code) may be used as a module for a larger problem
- It is helpful to allow in-module code, but a mechanism is needed
- The solution is to check if the file is the top-level one, or an imported one, looking at variable **`__name__`**
- If **`__name__`** is equal to **`__main__`**, it means that module is run by the interpreter, otherwise imported

Solving the problem

- Thus, one could write in the module car.py

```
if __name__ == "__main__":
   c = Car()
   print(c)
```

- Or, better

```
def main():
   c = Car()
   print(c)

if __name__ == "__main__":
   _main()
```

Packages

- When an application grows, it is no longer viable to have all the Python file in a single directory
- It is possible to split groups of files in separate directories, called packages
  - Each directory is a package
  - The files of the directory are modules
  - They can be imported with **`package_name.module_name`**

Importing from packages

- The traditional syntax still applies
  - `import pkg.mod`
  - `from pgk.mod import name`
  - `from pgk.mod import name as alt_name`
- Additionally, it is possible to import modules from a package
  - `from pkg import mod`
  - `from pkg import mod as alt_mod_name`

Package initialization

- Traditionally, the directory containing a package will also contain a special file named **`__init__.py`**
- It was mandatory until Python 3.3, now it is optional
- Can contain initialization statements, that are run when importing any module from the package

Working with external modules

- To access a module from pypi.org, the programmer must first install the module in the local Python interpreter
- Packages can be installed using pip
  - Search a project on https://pypi.org/
  - Install with **`pip install project_name`**
    - **`pip install flet`**
    - **`pip install mariadb`**
- Only installed packages can be imported

Virtual Environments

- Different projects may require different packages
  - The programmer’s local Python library will contain all sorts of packages, that are used by some project
  - When shipping a project, it may not be not clear which packages are needed to run it
- Python has a mechanism for separating the packages needed by each project: virtual environments
  - A a local “copy” of the Python interpreter, alongside with the packages needed for that project
  - Stored in the `.venv` directory

Where packages are searched for

- The **`import`** statement searches packages
  - In the current project directories
  - In the current virtual environment’s library
  - In a set of directories defined by the Python installation
- To check actual configuration

```
>>> import sys
>>> print(sys.path)
['/Applications/PyCharm.app/Contents/plugins/python-
ce/helpers/pydev',
'/Applications/PyCharm.app/Contents/plugins/python-
ce/helpers/third_party/thriftpy', …]
```

File requirements.txt

- A project may require several external packages
  - Installed with pip
  - Stored in the virtual environment
- How can a programmer declare the information about the required packages?
  - So that other people may install them in their system
  - So that one can control which version numbers are installed
- Adding a file `requirements.txt` to the project
  - Contains one line per package
  - May optionally specify the version number
  - PyCharm helps to synchronize the file with the `import` statements