Modular Programming

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Functions, modules and packages

They represent ways to help break up a program into smaller, easier to understand and more maintainable pieces. What we study during this course:

- **Procedural programming** - break the program up into functions
- **Modular programming - break the program up into modules organized according to packages**
- **Object-oriented programming** - See the program as a collection of objects that "*talk*" to each other

Modules

Modular programming - a software design technique that increases the extent to which software is composed of independent, interchangeable components called **modules**, each of which does one aspect within the program and contains everything necessary to accomplish this.

Modules

- Independent
- Interchangeable

Modular programming 101

+ Break up large(r) programs into smaller, easier to understand units
+ Help group related functions, classes and functionalities
+ Allow reusing implemented functionalities at a larger scale than single-functions
+ Management of naming conflicts between functions and modules
+ Allow studying a program's structure right from the IDE, source control, Windows Explorer, Finder etc.
+ Allows working on programs by many people at once without merge conflicts[1]
– Knowledge required to use effectively
– Might introduce problems related to imports, namespaces

[1] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line

Modules in Python

A Python module[2] is a *.py* file containing Python executable statements and definitions.

- **Name**: The file name is the module name with the suffix ”**.py**” appended
- **Docstring**: triple-quoted module doc string that defines the contents of the module file. Provide summary of the module and a description about the module's contents, purpose and usage.
- **Executable statements**: function definitions, module variables, initialization code

[2] `https://docs.python.org/3/tutorial/modules.html`

Modules in Python

How to define a Python module

- Write a **.py** file ☺
- Write it in C and dynamically load it at runtime[3] (remember CPython[4]?)
- Some modules are called built-in and are loaded by default; while you could extend these, you really shouldn't

[3] https://docs.python.org/3/extending/extending.html

[4] https://realpython.com/cpython-source-code-guide/#whats-in-the-source-code

Importing modules

- Importing a module means giving access to its local symbol table in the context of the importing code
- Use the *import* keyword to import modules
  - *import spam* places the name *spam* in the symbol table. Definitions in the *spam* module can be accessed using it
  - *from spam import is_prime* will add *is_prime* into the local symbol table
  - *from spam import is_prime as p* will add *is_prime* into the local symbol table under the alias of *p*
  - *from spam import* $*$ will add all names defined in *spam* that do not start with an underscore to the local symbol table[5]

Examine the symbol table

Use the built-in *dir()* function

[5] https://realpython.com/python-modules-packages/

Importing modules - the do nots

Things you **can** do but maybe should not, and why... ☺

- Use the import keyword inside functions – import statements should be at the start of the module's code to allow easily checking module dependencies
- Import everything from another module – module might include things we don't need or care about, or we could overwrite existing names (now or in the future, as both modules could be under development)
- Catch *ImportError* so that the program does not crash when searched for modules cannot be found – this might be okay, but make sure you know what you're doing

Module search path

Where does the **'import spam'** statement search for module *spam.py*?

1. Directory from where the current script was run
2. Directories specified by **PYTHONPATH** environment variable
3. Directories specified by the **PYTHONHOME** environment variable (an installation-dependent default path)

Module search path

Available through the *sys.path* variable

If the module name is not found, an **ImportError** exception is raised.

Demo

Environment Variables

This website has more info on accessing and changing environment variables in Windows/macOS/Linus - `https://www3.ntu.edu.sg/home/ehchua/programming/howto/Environment_Variables.html`

Learning more about modules

- **dir(module_name)** can be used to examine the module's symbol tables.
- **help(module_name)** can be used to get help on the module, its data types and functions.
- **pydoc** - A module that allows you to save extracted documentation to HTML format. Best used in command line at the operating system prompt.

Packages

- Modules help avoid naming collisions between module-level names (e.g., variables, functions, classes)
- Packages help avoid naming collisions between modules
- A Python package is a directory on the filesystem, and may contain an *__init__.py* file that includes package initialization code
- **A.B** denotes submodule **B** found in package **A**
- The same rules apply for importing packages as with modules

Packages

- Packages[6] are a way of structuring Python's module namespace by using "dotted module names"
- **A.B** denotes submodule **B** found in package **A**.
- The same rules apply for importing packages as with modules
- On the drive, directory hierarchies represent packages, so **B.py** will be found in a directory called **A**
- Each package directory contains an *__init__.py* file, telling Python to interpret it as a collection of modules
- *__init__.py* can be empty, or include package initialization code.

[6] https://docs.python.org/3/tutorial/modules.html#packages

Modules and packages examples

Modules

Take a look at the code from the **ex28_modules** example

Modules and packages

A modular version of the rational numbers calculator is available at **ex29_modular_calc**

Required modules for A6

Create modules for:

- **User Interface** - Functions related to user interaction. Contains input and data validation, print operations. This is the only module where input/print operations are present.
- **Functions** - Contains functions required to implement program features. These functions communicate via input parameters, return parameters and raising exceptions.
- **Start** - Code that starts the program by calling the required UI function(s)