Chapter 5 - Modules

CS 171 - Computer Programming 1
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Modules

- Functions allow us to group pieces of code
  - to support re-use throughout the program;
- Modules allow us to group sets of functions
  - to support re-use by any program;
  - Custom data types (studied in the next chapter), can also be included;
- Python also supports the creation of *packages*:
  - sets of modules that are grouped together
  - normally when they provide related functionality

Outline

1. Modules and Packages
   - Modules
   - Packages
   - Custom Modules
     - The TextUtil Module
2. Overview of Python's Standard Library
   - String Handling
   - Mathematics and Numbers
   - Times and Dates
   - Algorithms and Collection Data Types

Outline

1. Modules and Packages
   - Modules
   - Packages
   - Custom Modules
2. Overview of Python's Standard Library

Modules

Modules in Python:

- (Simplistically) consist of a `.py` file
- Can contain any Python code
- All the code we have written so far, which was included in a single file, can be considered a module, as well as a program
- But,
  - we have written *programs* designed to be executed
  - *modules* are designed to be imported and used by other *programs*

Modules

- Several syntaxes can be used to import a module:

```
import _importable_
import _importable1_, _importable2_, ..., _importableN_
import _importable_ as _preferred_name_
```

- `importable` can be:
  - a module, such as `collections`
  - a package
  - a module in a package
    - each case is separated with a dot (`.`), such as `os.path`
- We will essentially use the two first possibilities;
  - They are safe and avoid name collisions;

Modules

- It is common to put all the `imports` at the beginning of `.py` files
  - possibly after the module's documentation
- It is recommended:
  - importing standard library modules first
  - then third-party library modules
  - finally, our own modules
- To import modules, we can also use the syntaxes:

```
from importable import object as preferred_name
from importable import object1, object2, ..., objectN
from importable import (object1, object2, object3, object4, object5,
    object6, ..., objectN)
from importable import *
```

Modules

- `from _importable_ import *` imports the objects from a module
- In this case:

```
from os.path import *
print(basename(filename))
```

  - almost 40 names are imported
    - including `dirname`, `exists` and `split`
  - so, your program can **not** use these names, e.g. for a variable
  - otherwise there is a name collision

Modules

- `from os.path import dirname`
  - imports function `dirname` from module `os.path`
- So, we can conveniently call `dirname()`
- But if later our program does `dirname = "."`
  - then `dirname` will reference the string, not the (imported) function
  - so, doing `dirname()` will raise a `TypeError` exception

Modules

Examples

```
import os
print(os.path.basename(filename)) # safe fully qualified access

import os.path as path
print(path.basename(filename))    # risk of name collision with path,
                                  # if path is used in your program

from os import path
print(path.basename(filename))    # risk of name collision with path

from os.path import basename
print(basename(filename))         # risk of name collision with basename

from os.path import *
print(basename(filename))         # risk of many name collisions,
                                  # as we are importing all the objects
```

Modules

- Be careful with naming your modules;
  - Avoid creating a module with the same name as one of Python's
- You can check whether a module name is in use by trying to import it:
  - `python -c "import Music"`
  - in the command line, running `python -c` executes a piece of code
- On an `Error`, e.g., a `ModuleNotFoundError`, the name is not in use

Modules

Example, adapted from https://docs.python.org/3/tutorial/modules.html

- stored in `fibo.py`:

```
# Fibonacci numbers module
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
```

```
>>> import fibo

>>> fib(10)
[...] NameError: name 'fib' is not defined
>>> fibo.fib(10)
0 1 1 2 3 5 8
>>> fibo.__name__
'fibo'
```

Packages

- A package is simply a directory
  - that contains a set of modules
  - and a file called `__init__.py`

```
Graphics/
        __init__.py
        Bmp.py
        Jpeg.py
        Png.py
        Tiff.py
        Xpm.py
```

- This would make sense if all modules shared the same purpose
  - they could all provide functions such as `load()` or `save()`
  - but, in this case, for handling different types of images
- `Graphics` should be a subdirectory inside our program's directory
  - or in the Python path, which you may find by doing:

```
>>> import sys
>>> print(sys.path)
```

Packages

- Again, be careful with package naming to avoid conflicts

```
Graphics/
        __init__.py
        Bmp.py
        Jpeg.py
        Png.py
        Tiff.py
        Xpm.py
```

- The name `Graphics` must not be used in the standard library

Packages

- Examples of how we could use our (imaginary) package:

```
import Graphics.Bmp
image = Graphics.Bmp.load("bashful.bmp")

import Graphics.Jpeg as Jpeg
image = Jpeg.load("doc.jpeg")

from Graphics import Png
image = Png.load("dopey.png")

from Graphics import Tiff as picture
image = picture.load("grumpy.tiff")
```

Packages

- We can load all the modules in a package in a single statement
- We edit `__init__.py` to specify which modules we want to load
  - And assign a list of module names to the (special) variable `__all__`:

```
__all__ = ["Bmp", "Jpeg", "Png", "Tiff", "Xpm"]
```

- Now we can define a different import statement

```
from Graphics import *
```

- This imports all the modules named in the package's `__all__` list.

- This strategy can also be applied in a module
- If we define an `__all__` list in a `_module_` itself
  - when someone does `from _module_ import *`, only the objects in `__all__` are imported

Custom Modules

- We will now create a module (which is actually simply a `.py` file);
- We will call it `TextUtil` (in the file `TextUtil.py`)
- It will have 3 functions:
  - `is_balanced()`, to check if a string has a balanced number of parenthesis
  - `shorten()`, presented in the previous chapter
  - `simplify()`, to strip spurious characters, e.g. whitespace, from a string

The TextUtil Module

- The structure of a module differs little from that of a program
- Typically the module starts with a triple quoted string:
  - providing an overview of the module's content and usage examples
- Here's the start of the `TextUtil.py` file:

```
"""
This module provides a few string manipulation functions.
>>> is_balanced("(Python (is (not (lisp))))")
True
>>> shorten("The Crossing", 10)
'The Cro...'
>>> simplify(" some text with spurious whitespace ")
'some text with spurious whitespace'
"""

import string
```

The TextUtil Module

- After the docstring come the imports, and then the rest of the module
- A module's docstring is available as `TextUtil.__doc__`
- Function `shorten` was already presented in the previous chapter
  - we will not repeat it here, but it could/should be included in the module

The TextUtil Module

- Here's function `simplify()`

```
def simplify(text, whitespace=string.whitespace, delete=""):
    r""" Returns the text with multiple spaces reduced to single spaces

        The whitespace parameter is a string of characters,
        each of which is considered to be a space.

        If delete is not empty it should be a string, in which case any
        characters in the delete string are excluded from the resultant
        string.

        >>> simplify(" this    and\n that\t too")
        'this and that too'

        >>> simplify("  Washington   D.C.\n")
        'Washington D.C.'

        >>> simplify("  Washington   D.C.\n", delete=",;:.")
        'Washington DC'

        >>> simplify(" disemvoweled ", delete="aeiou")
        'dsmvwld'
        """
```

The TextUtil Module

- Here's function `simplify()` continued

```
    result = []
    word = ""
    for char in text:
        if char in delete:
            continue
        elif char in whitespace:
            if word:
                result.append(word)
                word = ""
        else:
            word += char
    if word:
        result.append(word)
    return " ".join(result)
```

The TextUtil Module

- Here's function `is_balanced()`, without docstring

```
def is_balanced(text, brackets="()[]{}<>"):
    counts = {}
    left_for_right = {}
    for left, right in zip(brackets[::2], brackets[1::2]):
        assert left != right, "the bracket characters must differ"
        counts[left] = 0
        left_for_right[right] = left
    for c in text:
        if c in counts:
            counts[c] += 1
        elif c in left_for_right:
            left = left_for_right[c]
            if counts[left] == 0:
                return False
            counts[left] -= 1
    return not any(counts.values())
```

The TextUtil Module

- Using any of the functionality from `TextUtil` is now easy:

```
import TextUtil

text = " a puzzling conundrum "
text = TextUtil.simplify(text)
    # text == 'a puzzling conundrum'
```

- For this, `TextUtil.py` must be in the same directory as the program
  - or in any directory in the Python path

Outline

1 Modules and Packages

2 Overview of Python's Standard Library
- String Handling
- Mathematics and Numbers
- Times and Dates
- Algorithms and Collection Data Types

Python's Standard Library

- Python comes with a wide range of functionality readily available
- This is included in Python's standard library
- (Powerful) Third party libraries are also available
- It is important that you keep this in mind!
  - Some of the functionality you need may already have been implemented
- We will cover just a glimpse of the standard library
  - Using concise examples to demonstrate particular modules

String Handling

For manipulating strings,

- `string` module provides some useful constants
  - `string.ascii_letters`
  - `string.hexdigits`
- `textwrap` module can wrap lines of text
  - to a specified width
  - to minimize indentation
- the `difflib` module provides methods for comparing sequences
  - such as strings
  - and produces results in different formats

String Handling

- Python provides two different ways to write text to a file
  - using the `write()` method

    ```
    sys.stdout.write("Another error message\n")
    ```

  - using the `print()` function, with the `file` keyword argument

    ```
    print("An error message", file=sys.stdout)
    ```

  - Both lines are printed to `sys.stdout`
    - a file object that represents *standard output stream*, the console
  - At program startup, Python automatically creates and opens
    - `sys.stdin`, `sys.stdout` and `sys.stderr`

Example: The io.StringIO class

- Sometimes we want to capture in a string the output that we want to write in a file
- This can be achieved using the `io.StringIO` class

```
import io

sys.stdout = io.StringIO()
    # any text sent to sys.stdout will now be sent to
    # the io.StringIO object, and not the console

print("An error message", file=sys.stdout)
    # running this code, no output is shown: the string is not printed

x = sys.stdout.getvalue()
    # extracting the value sent to sys.stdout

sys.stdout = sys.__stdout__
    # this will revert sys.stdout to the console

print(x)
    # prints 'An error message'
```

Mathematics and Numbers

- Besides `int`, `float` and `complex` numbers, the library provides
  - `decimal.Decimal`
  - `fractions.Fraction`
- Three numeric libraries are available
  - `math`, for standard mathematical functions
  - `cmath`, for complex number mathematical functions
  - `random`, providing functions for random number generation
- The third-party `NumPy` package is quite useful for scientific programming

Times and Dates

- Functions and classes for date and time handling are available
  - `calendar`
  - `datetime`
- Time and date handling is a complex topic
  - calendars have varied in different places
  - daylight saving time and time zones vary
- Third-party libraries are quite good
  - `dateutil`

    `www.labix.org/python-dateutil`
  - `mxDateTime`

    `https://www.egenix.com/products/python/mxBase/mxDateTime/`
- The `time` module handles timestamps
  - These hold the number of seconds since the epoch (1970-01-01T00:00:00 on Unix)

Example: The calendar, datetime and time Modules

- Objects of type `datetime.datetime` are created programmatically
- UTC date/times objects are usually received from external sources

```
>>> import calendar, datetime, time

>>> moon_datetime_a = datetime.datetime(1969, 7, 20, 20, 17, 40)
    # the date and time Apollo 11 landed on the moon

>>> moon_datetime_a.isoformat()
'1969-07-20T20:17:40'

>>> moon_time = calendar.timegm(moon_datetime_a.utctimetuple())
    # this is an int and holds the number of seconds since the
    # epoch to the moon landing;

>>> moon_time
-14182940
    # this is a negative number, as the date occurred before the epoch

>>> datetime.datetime.utcnow().isoformat()
'2020-03-22T11:00:59.512478'
```

Algorithms and Collection Data Types

- The `bisect` module provides functions for searching sorted sequences
  - and for inserting items while preserving the sort order
  - These functions use the binary search algorithm, so they are fast
- Module `heapq` provides functions for turning a sequence into a heap
  - and for inserting/removing items while keeping the sequence as a heap
  - Heaps are collections where the first item is always the smallest
- The `collections` package provides the collection data types:
  - the `collections.defaultdict` dictionary
  - `collections.namedtuple`
  - both discussed earlier
- The `array` module provides the `array.array` sequence type
  - than can store numbers or characters in a very space-efficient way

Example: The heapq Module

- Provides functions for converting a list into a heap
  - and for adding and removing items from the heap
- A heap is a binary tree that respects the *heap property*
  - the first item/the root of the tree is the smallest item
  - each of the subtrees is also a heap

```
>>> import heapq
>>> heap=[]
>>> heapq.heappush(heap, (5, "rest"))
>>> heapq.heappush(heap, (2, "work"))
>>> heapq.heappush(heap, (4, "study"))
>>> heap
[(2, 'work'), (5, 'rest'), (4, 'study')]
```

Example: The heapq Module

- If we already have a list, we can turn it into a heap

```
>>> import heapq
>>> l = [(5, 'rest'), (4, 'study'), (2, 'rest')]
>>> heapq.heapify(l)
>>> l
[(2, 'rest'), (4, 'study'), (5, 'rest')]
```

- The smallest item can be removed using `heapq.heappop(heap)`

```
>>> l
[(2, 'rest'), (4, 'study'), (5, 'rest')]
>>> heapq.heappop(l)
(2, 'rest')
>>> l
[(4, 'study'), (5, 'rest')]
```