TATA ELXSI

Python Scripting

Learning & Development Team

Monday, November 2, 2020

Module – 6 : Modules

Disclaimer

- This material is developed for in house training at **TATA ELXSI**.
- *TATA ELXSI is not responsible for any errors or omissions, or for the results obtained from the use of this information. All information in this material is provided "as is", with no guarantee of completeness, timeliness or of the results obtained from the use of this information...*
- *This is a training material and the information contained herein is not intended to be a source of advice or research outcome with respect to the material presented, and the information and/or documents/examples contained in this material do not constitute authors or TATA ELXSI's advice.*
- The training material therefore was solely developed for educational purposes. Currently, commercialization of the prototype does not exist, nor is the prototype available for general usage.
- **TATA ELXSI**, its staff, its students or any other participants can use it as a part of training. This material should not be found on any website.
- For the preparation of this material we have referred from the below mentioned links or books.
- Excerpts of these material has been taken where there is no copy right infringements.

Agenda

- Why use modules?
- Using the os, sys modules
- How import works
- Different ways of importing
- Byte-compiled .pyc files
- Making your own Modules
- Creating your own Modules
- Module Namespaces
- Changing the module search path

What is module?

- A module is a Python object with arbitrarily named attributes that you can bind and reference.
- Simply, a module is a file consisting of Python code.
- A module allows you to logically organize your Python code.
- Grouping related code into a module makes the code easier to understand and use.
- A module can define functions, classes and variables.
- A module can also include run able code.

Module

hello.py

```
def print_func( par ):
    print ("Hello : ", par)
    return
```

hello_main.py

```
import  hello

hello.print_func('world')
```

The import Statement:

- You can use any Python source file as a module by executing an import statement in some other Python source file.

- The import has the following syntax:
  - *import importable*
  - *import importable1, importable2, ..., importableN*
  - *import importable as preferred_name*

Import statement

- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

- A search path is a list of directories that the interpreter searches before importing a module.

- Here are some other import syntaxes:
  - *from importable import object as preferred_name*
  - *from importable import object1, object2, ..., objectN*
  - *from importable import (object1, object2, object3, object4, object5,object6, ..., objectN)*

Module name : 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()
```

```
def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

- import fibo
- >>>
- >>> fibo.fib(1000)
- 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
- >>> fibo.fib2(100)
- [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
- >>> fibo.__name__
- 'fibo‘

- If you intend to use a function often you can assign it to a local name:
- >>>
- >>> fib = fibo.fib
- >>> fib(500)
- 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
-

Import Example

- directly to module's symbol table.
- For example:

```
>>> from fibo import fib, fib2
```

OR

```
>>> from fibo import *
>>> fib(500)
```

1 1 2 3 5 8 13 21 34 55 89 144 233 377

- This imports all names except those beginning with an underscore (_).
- In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

create a .pyc

- Python automatically compiles your script to compiled code, so called byte code, before running it.
- When a module is imported for the first time, or when the source is more recent than the current compiled file, a .pyc file containing the compiled code will usually be created in the same directory as the .py file.
- When you run the program next time, Python uses this file to skip the compilation step.

create a .pyc

- Running a script is not considered an import and no .pyc will be created.

- if you have a script file abc.py that imports another module xyz.py, when you run abc, xyz.pyc will be created

- If you need to create a .pyc file for a module that is not imported, you can use the py_compile and compileall modules.

- The py_compile module can manually compile any module.
- One way is to use the py_compile.compile function in that module interactively:

```
>>> import py_compile
>>> py_compile.compile('abc.py')
```

optimization

- You can also automatically compile all files in a directory or directories using the compileall module.

- $ python -m compileall .

Standard Modules

- Python comes with a library of standard modules which are ,built into the interpreter.
- One particular module deserves some attention: sys , which is built into every Python interpreter.
- The variables sys.ps1 and sys.ps2 define the strings used as primary and secondary prompts:

- >>> import sys
- >>> sys.ps1
- '>>> '
- >>> sys.ps2
- '... '

Standard Modules

- The built-in function dir() is used to find out which names a module defines.
- It returns a sorted list of strings:

- >>> import sys
- >>> dir(sys)

['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__', '__package__', '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe', '_home', '_mercurial', '
_xoptions', 'api_vers...ion', 'argv',…….

Standard Modules

- >>> import builtins
- >>> dir(builtins)

- ['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',……
- ……
- ……
- …….

variables

- >>> names=['vikas','suhas','ullas']
- >>> print (names)
- ['vikas', 'suhas', 'ullas']

- >>> dir()
- ['__builtins__', '__doc__', '__loader__', '__name__', '__package__', 'builtins', 'names', 'sys']

- >>>

Packages

- Packages are a way of structuring Python's module namespace by using "dotted module names".
- For example, the module name A.B designates a submodule named B in a package named A.
- The use of dotted module names saves from name collision.

Packages

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

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

__init__.py file

- When importing the package, Python searches through the directories on sys.path looking for the package subdirectory.

- The __init__.py files are required to make Python treat directories containing the file as packages.

- This prevents directories with a common name, such as string, unintentionally hiding valid modules that occur later on the module search path.

- In the simplest case, __init__.py can just be an empty file, but it can also execute initialization code for the package or set the __all__ variable.

usage

- Users of the package can import individual modules from the package, for example:

- import sound.effects.echo
- This loads the submodule sound.effects.echo. It must be referenced with its full name.

- An alternative way of importing the submodule is:

- from sound.effects import echo
- This also loads the submodule echo, and makes it available without its package prefix

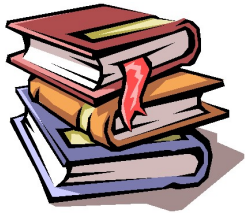
- Yet another variation is to import the desired function or variable directly:

- from sound.effects.echo import echofilter
- Again, this loads the submodule echo, but this makes its function echofilter() directly available:

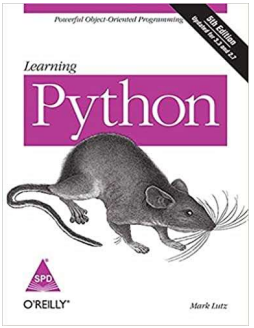
- Note that when using from package import item, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable.

- The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an ImportError exception is raised.

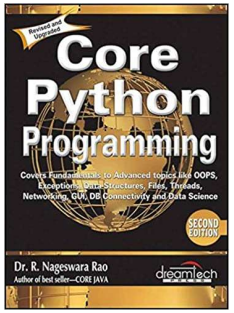
References



- **Python 3.x.x documentation**: https://docs.python.org/3/



- **Learning Python: Powerful Object-Oriented Programming: 5th Edition**



- **Core Python Programming [**by R. Nageswara Rao (Author)]

Thank you

Learning & Development

Tata Elxsi

Bangalore

www.tataelxsi.com