California State University Fullerton

CPSC-235P

Python Programming

Stephen T. May

Python Tutorial

Section 6

Modules

https://docs.python.org/release/3.9.6/tutorial/index.html

Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print(“Hello world!”)
```

- Mixed example

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

6. Modules

- If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost
- Python has a way to put definitions in a file and use them in a *script*
- A *module* is a file containing Python definitions and statements
- The file name is the module name with the suffix `.py` appended
- Within a module, the module's name (as a string) is available as the value of the global variable `__name__`

```
# filename: fibo.py

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'
```

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. More on Modules

- Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module
- Modules can import other modules
- The imported module names are placed in the importing module's global symbol table
- There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table
- Using `*` imports all names except those beginning with an underscore (_)

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

- If the module name is followed by `as`, then the name following as is bound directly to the imported module

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1.1. Executing Modules as Scripts

- When you run a Python module from the command line the code in the module will be executed, with the `__name__` set to "`__main__`"

```
python fibo.py <arguments>
```

- The code that parses the command line only runs if the module is executed as the “main” file
- Add code at the end of the module to make the file usable as a script

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

- If the module is imported, the code is not run

```
>>> import fibo
>>>
```

6.1.2. The Module Search Path

- When a module named *spam* is imported, the interpreter first searches for a built-in module with that name
- If not found, it then searches for a file named *spam.py* in a list of directories given by the variable `sys.path`.
- `sys.path` is initialized from these locations:
  - The directory containing the input script (or the current directory when no file is specified)
  - `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable PATH)
  - The installation-dependent default
- After initialization, Python programs can modify `sys.path`
- The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path
- This means that scripts in that directory will be loaded instead of modules of the same name in the library directory
- This is an error unless the replacement is intended

6.1.3. “Compiled” Python files

- Python caches the compiled version of each module in the `__pycache__` directory under the name `module.version.pyc`
- For example, in CPython release 3.3 the compiled version of *spam.py* would be cached as `__pycache__/spam.cpython-33.pyc`
- Python checks the modification date of the source against the compiled version to see if it’s out of date and needs to be recompiled
- The compiled modules are platform-independent, so the same library can be shared among systems with different architectures
- Python does not check the cache in two circumstances
  - First, it always recompiles and does not store the result for the module that’s loaded directly from the command line
  - Second, it does not check the cache if there is no source module
- To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.
- A program doesn’t run any faster when it is read from a `.pyc` file than when it is read from a `.py` file; the only thing that’s faster about `.pyc` files is the speed with which they are loaded.
- There is more detail on this process, including a flow chart of the decisions, in *PEP 3147*.

6.2. Standard Modules

- Python comes with a library of standard modules
- Some modules are built into the interpreter
- The set of such modules is a configuration option which also depends on the underlying platform
- For example, the `winreg` module is only provided on Windows systems
- `sys` is built into every Python interpreter
- The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts
- The variable `sys.path` is a list of strings that determines the interpreter's search path for modules
- It is initialized to a default path taken from the environment variable `PYTHONPATH`
- You can modify it using standard list operations

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3. The `dir()` Function

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

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```

- Without arguments, `dir()` lists the names you have defined currently

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

- `dir()` does not list the names of built-in functions and variables
- If you want a list of those, they are defined in the standard module `builtins`

```
>>> import builtins
>>> dir(builtins)
TRY IT…
```

6.4. 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`
- Modules save the authors of different modules from having to worry about each other's global variable names
- Dotted module names save the authors of multi-module packages from having to worry about each other's module names
- 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, described later

6.4. Packages (cont.)

- Suppose you want to design a collection of modules (a "*package*") for the uniform handling of sound files and sound data
- There are many different sound file formats (usually recognized by their extension, for example: `.wav, .aiff, .au`)
- You may need to create and maintain a growing collection of modules for the conversion between the various file formats
- There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect)
- You will be writing a never-ending stream of modules to perform these operations

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format
              __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
              ...
```

6.4. Packages (cont.)

- Users of the package can import submodules from the package
- Users of the package can import desired functions directly
- 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
- When using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package
- The last item can be a module or a package but can't be a class or function or variable defined in the previous item

```
# Import submodule - referenced with its full name
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

# Import submodule - referenced without its package prefix
from sound.effects import echo
echo.echofilter(input, output, delay=0.7, atten=4)
```

```
# Import only function
from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7, atten=4)
```

6.4.1. Importing * from a Package

- What happens when the user writes `from sound.effects import *`?
- Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all
- This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported
- The only solution is for the package author to provide an explicit index of the package
- The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when from package import * is encountered
- It is up to the package author to keep this list up-to-date when a new version of the package is released

```
# File: sound/effects/__init__.py
__all__ = ["echo", "surround", "reverse"]
```

6.4.2. Intra-package References

- Absolute imports use the complete reference to refer to submodules of siblings packages
- Relative imports use use leading dots to indicate the current and parent packages involved in the relative import
- Note that relative imports are based on the name of the current module
- Since the name of the main module is always "`__main__`", modules intended for use as the main module of a Python application must always use absolute imports

```
# Filename: sound/filters/vcoder.py
from sound.effects import echo
```

```
# Filename: sound/effects/surround.py
from . import echo # looks in current package: sound/effects
from .. import formats # looks in parent package: sound
from ..filters import equalizer # looks in filters package under
parent package: sound/filters
```

6.4.3. Packages in Multiple Directories

- Packages support one more special attribute, `__path__`
- This is initialized to be a list containing the name of the directory holding the package's `__init__.py` before the code in that file is executed
- This variable can be modified; doing so affects future searches for modules and subpackages contained in the package
- While this feature is not often needed, it can be used to extend the set of modules found in a package