PyLearn_PyTutorial_6_Module

April 30, 2020

#

Learning Python

1 The Python Tutorial -> Modules

Link: https://docs.python.org/3/tutorial/modules.html

[51]:
```
!cat yp_fibo.py
```

```
# Fibonacci numbers module

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

def fibo(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

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

[30]:
```
import yp_fibo

yp_fibo.printFibo(7)

f = yp_fibo.fibo(7)
print(f)
```

```
print(yp_fibo.__name__)   # the module's name

fib = yp_fibo.fibo        # assign the function to a local name
f = fib(9)
print(f)
```

```
0 1 1 2 3 5
[0, 1, 1, 2, 3, 5]
yp_fibo
[0, 1, 1, 2, 3, 5, 8]
```

- The author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables.
- `import yp_fibo` does not enter the names of the functions defined in `yp_fibo` directly in the current symbol table;
- it only enters the module name fibo there.

[41]:
```
!python3 yp_fibo.py 1000
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

It runs the code:

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

which only runs if the module is executed as the "main" file:

[44]:
```
!cat yp_fibo2.py
```

```
def printFibo2(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fibo2(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
```

[45]:
```
from yp_fibo2 import printFibo2, fibo2

printFibo2(7)
```

```
f = fibo2(7)
print(f)

fib = fibo2        # assign the function to a local name
f = fib(9)
print(f)
```

```
0 1 1 2 3 5
[0, 1, 1, 2, 3, 5]
[0, 1, 1, 2, 3, 5, 8]
```

- `from yp_fibo2 import printFibo2, fibo2`imports names from a module directly into the importing module's symbol table,
- but it does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `yp_fibo2` is not defined).

[47]:
```
!cat yp_fibo3.py
```

```
def printFibo3(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fibo3(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
```

[48]:
```
from yp_fibo3 import *

printFibo3(7)

f = fibo3(7)
print(f)

fib = fibo3        # assign the function to a local name
f = fib(9)
print(f)
```

```
0 1 1 2 3 5
[0, 1, 1, 2, 3, 5]
[0, 1, 1, 2, 3, 5, 8]
```

- `import *` 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.

[49]:
```
import yp_fibo as ypfib

ypfib.printFibo(7)

f = ypfib.fibo(7)
print(f)

print(ypfib.__name__)    # the module's name

fib = ypfib.fibo         # assign the function to a local name
f = fib(9)
print(f)
```

```
0 1 1 2 3 5
[0, 1, 1, 2, 3, 5]
yp_fibo
[0, 1, 1, 2, 3, 5, 8]
```

[50]:
```
from yp_fibo import printFibo as printFibonacci, fibo as fibonacci

printFibonacci(7)

f = fibonacci(7)
print(f)

fib = fibonacci         # assign the function to a local name
f = fib(9)
print(f)
```

```
0 1 1 2 3 5
[0, 1, 1, 2, 3, 5]
[0, 1, 1, 2, 3, 5, 8]
```

Note For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use importlib.reload(), e.g. import importlib; importlib.reload(modulename).

1.0.1 The Module Search Path

[9]:
```
sys.path
```

[9]:
```
['/home/yp/dev/python/py-learning/PyLearn_PyTutorial_6_Module',
 '/opt/jupyterhub/lib/python38.zip',
 '/opt/jupyterhub/lib/python3.8',
 '/opt/jupyterhub/lib/python3.8/lib-dynload',
 '',
 '/opt/jupyterhub/lib/python3.8/site-packages',
 '/opt/jupyterhub/lib/python3.8/site-packages/IPython/extensions',
 '/home/yp/.ipython',
 '/home/yp/dev/python',
 '/home/yp/dev/python',
 '/home/yp/dev/python',
 '/home/yp/python']
```

[10]:
```
!echo $PATH
```

```
/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/opt/jupyterhub/bin
```

[11]:
```
!echo $PYTHONPATH
```

[13]:
```
sys.path.append('/home/yp/python')
sys.path
```

[13]:
```
['/home/yp/dev/python/py-learning/PyLearn_PyTutorial_6_Module',
 '/opt/jupyterhub/lib/python38.zip',
 '/opt/jupyterhub/lib/python3.8',
 '/opt/jupyterhub/lib/python3.8/lib-dynload',
 '',
 '/opt/jupyterhub/lib/python3.8/site-packages',
 '/opt/jupyterhub/lib/python3.8/site-packages/IPython/extensions',
 '/home/yp/.ipython',
 '/home/yp/dev/python',
 '/home/yp/dev/python',
 '/home/yp/dev/python',
 '/home/yp/python',
 '/home/yp/python',
 '/home/yp/python']
```

1.0.2 "Compiled" Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.version.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

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.

[61]:
```
!ls -la __pycache__
```

```
total 28
drwxrwxr-x 2 yp yp 4096 Apr 30 16:00 .
drwxr-xr-x 4 yp yp 4096 Apr 30 17:26 ..
-rw-r--r-- 1 yp yp  544 Apr 30 03:26 fibo.cpython-36.pyc
-rw-rw-r-- 1 yp yp  624 Apr 30 15:54 yp_fibo2.cpython-38.pyc
-rw-rw-r-- 1 yp yp  624 Apr 30 16:00 yp_fibo3.cpython-38.pyc
-rw-r--r-- 1 yp yp  638 Apr 30 13:10 yp_fibo.cpython-36.pyc
-rw-r--r-- 1 yp yp  621 Apr 30 15:09 yp_fibo.cpython-38.pyc
```

1.0.3 Standard Modules

[4]:
```
import sys
print(sys.ps1)
print(sys.ps2)
```

```
In :
…:
```

> The variables sys.ps1 and sys.ps2 define the strings used as primary and secondary prompts:

1.0.4 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:

[37]:
```
import yp_fibo
print(dir(yp_fibo))
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'fibo', 'printFibo']
```

[36]:
```
import sys
print(dir(sys))
```

```
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
'__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
'__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
'_base_executable', '_clear_type_cache', '_current_frames', '_debugmallocstats',
'_framework', '_getframe', '_git', '_home', '_xoptions', 'abiflags',
```

```
'addaudithook', 'api_version', 'argv', 'audit', 'base_exec_prefix',
'base_prefix', 'breakpointhook', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode',
'exc_info', 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags',
'float_info', 'float_repr_style', 'get_asyncgen_hooks',
'get_coroutine_origin_tracking_depth', 'getallocatedblocks', 'getcheckinterval',
'getdefaultencoding', 'getdlopenflags', 'getfilesystemencodeerrors',
'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount',
'getsizeof', 'getswitchinterval', 'gettrace', 'hash_info', 'hexversion',
'implementation', 'int_info', 'intern', 'is_finalizing', 'last_traceback',
'last_type', 'last_value', 'maxsize', 'maxunicode', 'meta_path', 'modules',
'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
'ps3', 'pycache_prefix', 'set_asyncgen_hooks',
'set_coroutine_origin_tracking_depth', 'setcheckinterval', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
'warnoptions']
```

Without arguments, dir() lists the names you have defined currently: It does not list the names of built-in functions and variables

[40]:
```
print(dir())
```

```
['In', 'Out', '_', '_12', '_13', '_17', '_18', '_19', '_20', '_21', '_22',
'_23', '_24', '_25', '_26', '_27', '_28', '_29', '_30', '_31', '_32', '_34',
'_39', '_7', '_9', '__', '___', '__builtin__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__', '_dh', '_exit_code', '_i',
'_i1', '_i10', '_i11', '_i12', '_i13', '_i14', '_i15', '_i16', '_i17', '_i18',
'_i19', '_i2', '_i20', '_i21', '_i22', '_i23', '_i24', '_i25', '_i26', '_i27',
'_i28', '_i29', '_i3', '_i30', '_i31', '_i32', '_i33', '_i34', '_i35', '_i36',
'_i37', '_i38', '_i39', '_i4', '_i40', '_i5', '_i6', '_i7', '_i8', '_i9', '_ih',
'_ii', '_iii', '_oh', 'builtins', 'exit', 'get_ipython', 'quit', 'sys',
'yp_fibo']
```

If you want a list of those, they are defined in the standard module builtins:

[41]:
```
import builtins
print(dir(builtins) )
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError',
'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
```

```
'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning',
'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError',
'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit',
'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
'Warning', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__',
'__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__',
'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes',
'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec',
'filter', 'float', 'format', 'frozenset', 'get_ipython', 'getattr', 'globals',
'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max',
'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
'property', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

1.0.5 Packages

Suppose we want to design a collection of modules (a "package") for the uniform handling of sound files and sound data.
Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
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
```

...

The `__init__.py` files are required to make Python treat directories containing the file as packages. `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable,

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.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

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, so it can be used as follows: "‘ echo.echofilter(input, output, delay=0.7, atten=4)

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:

```
echofilter(input, output, delay=0.7, atten=4)
```

`from package import item` -> 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.
`import item.subitem.subsubitem` -> 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.

1.0.6 Importing * From a Package

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

For example, the file `sound/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the sound package.

If `__all__` is not defined, the statement `from sound.effects import *`: - does not import all submodules from the package sound.effects into the current namespace; - it only ensures that the package sound.effects has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes: - any names defined (and submodules explicitly loaded) by `__init__.py`. - It also includes any submodules of the package that were explicitly loaded by previous import statements.

Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use import *, it is still considered bad practice in production code.

Remember, there is nothing wrong with using `from package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

1.0.7 Intra-package References

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages.

For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

You can also write **relative imports**, with the `from module import name` form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

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.

1.0.8 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.

2 END OF The Python Tutorial -> Modules