6. Modules
****

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

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__". For instance, use your favorite text editor to create a file called "fibo.py" in the current directory with the following contents:

```
# 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
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in "fibo" directly to the current *namespace* (see Python Scopes and Namespaces for more details); it only adds the module name "fibo" there. Using the module name you can access the functions:

```
>>> 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
```

6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. [1] (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, "modname.itemname".

Modules can import other modules. It is customary but not required to place all "import" statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the "import" statement that imports names from a module directly into the importing module's namespace. For example:

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

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, "fibo" is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 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.

Note that in general the practice of importing "*" from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

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
```

This is effectively importing the module in the same way that "import fibo" will do, with the only difference of it being available as "fib".

It can also be used when utilising "from" with similar effects:

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

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)".

6.1.1. Executing modules as scripts

When you run a Python module with

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

the code in the module will be executed, just as if you imported it, but with the "__name__" set to "__main__". That means that by adding this code at the end of your module:

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

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

```
$ 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
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

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. These module names are listed in "sys.builtin_module_names". 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 (by convention including a "site-packages" directory, handled by the "site" module).

More details are at The initialization of the sys.path module search path.

Note:

On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is **not** added to the module search path.

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. See section Standard Modules for more information.

6.1.3. "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 checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Also, 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.

Some tips for experts:

* You can use the "-O" or "-OO" switches on the Python command to reduce the size of a compiled module. The "-O" switch removes assert statements, the "-OO" switch removes both assert statements and __doc__ strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. "Optimized" modules have an "opt-" tag and are usually smaller. Future releases may change the effects of optimization.

* 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.

* The module "compileall" can create .pyc files for all modules in a directory.

* 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, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. 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. 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
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

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", or from a built-in default if "PYTHONPATH" is not set. You can modify it using standard list operations:

```
>>> 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']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
'__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
'__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
'_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', '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', 'pycache_prefix',
'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', '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:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
```

```
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

"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)
['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', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', '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',
'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
'zip']
```

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". Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

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"), so 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), so in addition you will be writing a never-ending stream of modules to perform these operations. 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
...
```

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 (unless using a *namespace package*, a relatively advanced feature). This prevents directories with a common name, such as "string", from 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.

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)
```

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.

Contrarily, 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.

6.4.1. Importing * From a Package

Now 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. Package authors may also decide not to support it, if they don't see a use for importing * from their package. 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.effects" package.

Be aware that submodules might become shadowed by locally defined names. For example, if you added a "reverse" function to the "sound/effects/__init__.py" file, the "from sound.effects import *" would only import the two submodules "echo" and "surround", but *not* the "reverse" submodule, because it is shadowed by the locally defined "reverse" function:

```
__all__ = [
"echo", # refers to the 'echo.py' file
"surround", # refers to the 'surround.py' file
"reverse", # !!! refers to the 'reverse' function now !!!
]

def reverse(msg: str): # <-- this name shadows the 'reverse.py' submodule
return msg[::-1] # in the case of a 'from sound.effects import *'
```

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.

6.4.2. 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's package. Since the main module does not have a package, modules intended for use as the main module of a Python application must always use absolute imports.

6.4.3. Packages in Multiple Directories

Packages support one more special attribute, "__path__". This is initialized to be a *sequence* of strings 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.

-[Footnotes]-

[1] In fact function definitions are also 'statements' that are 'executed'; the execution of a module-level function definition adds the function name to the module's global namespace.