1 Importing a Module

To use the functions and classes offered by a module, you have to import the module:

```
import math

print(math.sqrt(10))
```

The above imports the math standard module, making all of the functions in that module namespaced by the module name. It imports all functions and all classes, if any.

You can import the module under a different name:

```
import math as Mathematics

print(Mathematics.sqrt(10))
```

You can import a single function, making it available without the module name namespace:

```
from math import sqrt

print(sqrt(10))
```

You can import a single function and make it available under a different name:

```
from math import cos as cosine

print(cosine(10))
```

You can import multiple modules in a row:

```
import os, sys, re
```

You can make an import as late as in a function definition:

```
def sqrtTen():
import math
print(math.sqrt(10))
```

Such an import only takes place when the function is called.

You can import all functions from the module without the module namespace, using an asterisk notation:

```
from math import *
print(sqrt(10))
```

However, if you do this inside a function, you get a warning in Python 2 and error in Python 3:

```
def sqrtTen():
from math import *
print(sqrt(10))
```

You can guard for a module not found:

```
try:
import custommodule
except ImportError:
pass
```

Modules can be different kinds of things:

- Python files
- Shared Objects (under Unix and Linux) with the .so suffix
- DLL's (under Windows) with the .pyd suffix
- Directories

Modules are loaded in the order they're found, which is controlled by `sys.path`. The current directory is always on the path.

Directories should include a file in them called `__init__.py`, which should probably include the other files in the directory.

Creating a DLL that interfaces with Python is covered in another section.

1.1 Imported Check

You can check whether a module has been imported as follows:

```
if "re" in sys.modules:
print("Regular expression module is ready for use.")
```

2 Creating a Module

2.1 From a File

The easiest way to create a module is by having a file called `mymod.py` either in a directory recognized by the `PYTHONPATH` variable or (even easier) in the same directory where you are working. If you have the following file `mymod.py`:

```
class Object1:
def __init__(self):
self.name = 'object 1'
```

you can already import this "module" and create instances of the object `Object1`.

```
import mymod
myobject = mymod.Object1()
from mymod import *
myobject = Object1()
```

2.2 From a Directory

It is not feasible for larger projects to keep all classes in a single file. It is often easier to store all files in directories and load all files with one command. Each directory needs to have a `__init__.py` file which contains python commands that are executed upon loading the directory.

Suppose we have two more objects called `Object2` and `Object3` and we want to load all three objects with one command. We then create a directory called `mymod` and we store three files called `Object1.py`, `Object2.py` and `Object3.py` in it. These files would then contain one object per file but this not required (although it adds clarity). We would then write the following `__init__.py` file:

```
from Object1 import *
from Object2 import *
from Object3 import *

__all__ = ["Object1", "Object2", "Object3"]
```

The first three commands tell python what to do when somebody loads the module. The last statement defining `__all__` tells python what to do when somebody executes `from mymod import *`. Usually we want to use parts of a module in other parts of a module, e.g. we want to use `Object1` in `Object2`. We can do this easily with an `from .  import *` command as the following file `Object2.py` shows:

```
from . import *

class Object2:
def __init__(self):
self.name = 'object␣2'
self.otherObject = Object1()
```

2.3 Making a Program Usable as a Module

In order to make a program usable both as a standalone program to be called from a command line and as a module, it is advisable that you place all code in functions and methods, designate one function as the main one, and call then main function when `__name__` built-in equals '__main__'. The purpose of doing so is to make sure that the code you have placed in the main function is not called when your program is imported as a module; the code would be called upon import if it were placed outside of functions and methods.

Your program, stored in `mymodule.py`, can look as follows:

```
def reusable_function(x, y):
return x + y

def main():
pass
```

```
# Any code you like

if __name__ == '__main__':
main()
```

The uses of the above program can look as follows:

```
from mymodule import reusable_function
my_result = reusable_function(4, 5)
```

3 Extending Module Path

When import is requested, modules are searched in the directories (and zip files?) in the module path, accessible via `sys.path`, a Python list. The module path can be extended as follows:

```
import sys
sys.path.append("/My/Path/To/Module/Directory")
from ModuleFileName import my_function
```

Above, if `ModuleFileName.py` is located at `/My/Path/To/Module/Directory` and contains a definition of `my_function`, the second line ensures that the third line actually works.

4 Module Names

Module names seem to be limited to alphanumeric characters and underscore; dash cannot be used. While `my-module.py` can be created and run, importing `my-module` fails. The name of a module is the name of the module file minus the .py suffix.

Module names are case sensitive. If the module file is called `MyModule.py`, doing `import mymodule` fails while `import MyModule` is fine.

`PEP 0008` recommends module names to be in all lowercase, with possible use of underscores.

Examples of module names from the standard library include `math`, `sys`, `io`, `re`, `urllib`, `difflib`, and `unicodedata`.

5 Built-in Modules

For a module to be built-in is not the same as to be part of the standard library. For instance, `re` is not a built-in module but rather a module written in Python. By contrast, `_sre` is a built-in module.

Obtaining a list of built-in module names:

```
print(sys.builtin_module_names)
print("_sre" in sys.builtin_module_names) # True
print("math" in sys.builtin_module_names) # True
```