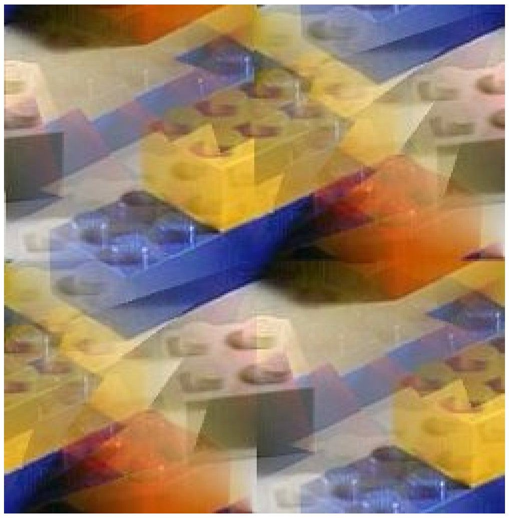
MODULAR PROGRAMMING AND MODULES

MODULAR PROGRAMMING



Modular programming is a software design technique, which is based on the general principal of modular design. Modular design is an approach which has been proven as indispensable in engineering even long before the first computers. Modular design means that a complex system is broken down into smaller parts or components, i.e. modules. These components can be independently created and tested. In many cases, they can be even used in other systems as well.

There is hardly any product nowadays, which doesn't heavily rely on modularisation, like cars, mobile phones. Computers belong to those products which are modularised to the utmost. So, what's a must for the hardware is an unavoidable necessity for the software running on the computers.

If you want to develop programs which are readable, reliable and maintainable without too much effort, you have to use some kind of modular software design. Especially if your application has a certain size. There exists a variety of concepts to design software in modular form. Modular programming is a software design technique to split your code into separate parts. These parts are called modules. The focus for this separation should be to have modules with no or just few dependencies upon other modules. In other words: Minimization of dependencies is the goal. When creating a modular system, several modules are built separately and more or less independently. The executable application will be created by putting them together.

IMPORTING MODULES

So far we haven't explained what a Python module is. To put it in a nutshell: every file, which has the file extension .py and consists of proper Python code, can be seen or is a module! There is no special syntax required to make such a file a module. A module can contain arbitrary objects, for example files, classes or attributes. All those objects can be accessed after an import. There are different ways to import a modules. We demonstrate this with the math module:

```
import math
```

The module math provides mathematical constants and functions, e.g. π (math.pi), the sine function (math.sin()) and the cosine function (math.cos()). Every attribute or function can only be accessed by putting "math." in front of the name:

```
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
>>> math.cos(math.pi/2)
6.123031769111886e-17
>>> math.cos(math.pi)
-1.0
```

It's possible to import more than one module in one import statement. In this case the module names are separated by commas:

```
import math, random
```

import statements can be positioned anywhere in the program, but it's good style to place them directly at the beginning of a program.

If only certain objects of a module are needed, we can import only those:

```
from math import sin, pi
```

The other objects, e.g. cos, are not available after this import. We are capable of accessing sin and pi directly, i.e. without prefixing them with "math."
Instead of explicitly importing certain objects from a module, it's also possible to import everything in the namespace of the importing module. This can be achieved by using an asterisk in the import:

```
>>> from math import *
>>> sin(3.01) + tan(cos(2.1)) + e
2.2968833711382604
>>> e
2.718281828459045
>>>
```

It's not recommended to use the asterisk notation in an import statement, except when working in the interactive Python shell. One reason is that the origin of a name can be quite obscure, because it can't be seen from which module it might have been imported. We will demonstrate another serious complication in the following example:

```
>>> from numpy import *
>>> from math import *
>>> print(sin(3))
0.1411200080598672
>>> sin(3)
0.1411200080598672
>>>
```

Let's slightly change the previous example by changing the order of the imports:

```
>>> from math import *
>>> from numpy import *
>>> print(sin(3))
0.14112000806
>>> sin(3)
0.14112000805986721
>>>
```

People use the asterisk notation, because it is so convenient. It means avoiding a lot of tedious typing. Another way to shrink the typing effort consists in renaming a namespace. A good example for this is the numpy module. You will hardly find an example or a tutorial, in which they will import this module with the statement

```
import numpy
```

It's like an unwritten law to import it with

```
import numpy as np
```

Now you can prefix all the objects of numpy with "np." instead of "numpy.":

```
>>> import numpy as np
>>> np.diag([3, 11, 7, 9])
array([[ 3,  0,  0,  0],
       [ 0, 11,  0,  0],
       [ 0,  0,  7,  0],
       [ 0,  0,  0,  9]])
>>> np.e
2.718281828459045
>>>
```

DESIGNING AND WRITING MODULES

But how do we create modules in Python? A module in Python is just a file containing Python definitions and statements. The module name is moulded out of the file name by removing the suffix .py. For example, if the file name is fibonacci.py, the module name is fibonacci.

Let's turn our Fibonacci functions into a module. There is hardly anything to be done, we just save the following code in the file fibonacci.py:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
def ifib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

The newly created module "fibonacci" is ready for use now. We can import this module like any other module in a program or script. We will demonstrate this in the following interactive Python shell:

```
>>> import fibonacci
>>> fibonacci.fib(7)
13
>>> fibonacci.fib(20)
6765
>>> fibonacci.ifib(42)
267914296
>>> fibonacci.ifib(1000)
434665576869374564356885276750406258025646605173717804024817290895
365554179490518904038798400792551692959225930803226347752096896232
398733224711616429964409065331879382989696499285160037044761377951
66849228875
>>>
```

Don't try to call the recursive version of the Fibonacci function with large arguments like we did with the iterative version. A value like 42 is already too large. You will have to wait for a long time!

As you can easily imagine: It's a pain if you have to use those functions often in your program and you always have to type in the fully qualified name, i.e. fibonacci.fib(7). One solution consists in assigning a local name to a module function to get a shorter name:

```
>>> fib = fibonacci.ifib
>>> fib(10)
55
>>>
```

But it's better, if you import the necessary functions directly into your module, as we will demonstrate further down in this chapter.

MORE ON MODULES

Usually, modules contain functions or classes, but there can be "plain" statements in them as well. These statements can be used to initialize the module. They are only executed when the module is imported.

Let's look at a module, which only consists of just one statement:

```
print("The module is imported now!")
```

We save with the name "one_time.py" and import it two times in an interactive session:

```
>>> import one_time
The module is imported now!
>>> import one_time
>>>
```

We can see that it was only imported once. Each module can only be imported once per interpreter session or in a program or script. If you change a module and if you want to reload it, you must restart the interpreter again. In Python 2.x, it was possible to reimport the module by using the built-in reload, i.e.reload(modulename):

```
$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import one_time
The module is imported now!
>>> reload(one_time)
The module is imported now!

>>>
```

This is not possible anymore in Python 3.x.
You will cause the following error:

```
>>> import one_time
The module is imported now!
>>> reload(one_time)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'reload' is not defined
>>>
```

Since Python 3.0 the reload built-in function has been moved into the imp standard library module. So it's still possible to reload files as before, but the functionality has to be imported. You have to execute an "import imp" and use imp.reload(my_module). Alternatively, you can use "imp import reload" and use reload(my_module).

Example with reloading the Python3 way:

```
$ python3
Python 3.1.2 (r312:79147, Sep 27 2010, 09:57:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> from imp import reload
>>> import one_time
The module is imported now!
>>> reload(one_time)
The module is imported now!

>>>
```

Since version 3.4 you should use the "importlib" module, because imp.reload is marked as deprecated:

```
>>> from importlib import reload
>>> import one_time
The module is imported now!
>>> reload(one_time)
The module is imported now!

>>>
```

A module has a private symbol table, which is used as the global symbol table by all functions defined in the module. This is a way to prevent that a global variable of a module accidentally clashes with a user's global variable with the same name. Global variables of a module can be accessed with the same notation as functions, i.e. *modname.name*
A module can import other modules. It is customary to place all import statements at the beginning of a module or a script.

IMPORTING NAMES FROM A MODULE DIRECTLY

Names from a module can directly be imported into the importing module's symbol table:

```
>>> from fibonacci import fib, ifib
>>> ifib(500)
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 symbol table. It's possible but not recommended to import all names defined in a module, except those beginning with an underscore "_":

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

This shouldn't be done in scripts but it's possible to use it in interactive sessions to save typing.

EXECUTING MODULES AS SCRIPTS

Essentially a Python module is a script, so it can be run as a script:

```
python fibo.py
```

The module which has been started as a script will be executed as if it had been imported, but with one exception: The system variable __name__ is set to "__main__". So it's possible to program different behaviour into a module for the two cases. With the following conditional statement the file can be used as a module or as a script, but only if it is run as a script the method fib will be started with a command line argument:

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

If it is run as a script, we get the following output:

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

If it is imported, the code in the if block will not be executed:

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

RENAMING A NAMESPACE

While importing a module, the name of the namespace can be changed:

```
>>> import math as mathematics
>>> print(mathematics.cos(mathematics.pi))
-1.0
```

After this import there exists a namespace mathematics but no namespace math.
It's possible to import just a few methods from a module:

```
>>> from math import pi,pow as power, sin as sinus
>>> power(2,3)
8.0
>>> sinus(pi)
1.2246467991473532e-16
```

KINDS OF MODULES

There are different kind of modules:

- Those written in Python
  They have the suffix: .py
- Dynamically linked C modules
  Suffixes are: .dll, .pyd, .so, .sl, ...
- C-Modules linked with the Interpreter:
  It's possible to get a complete list of these modules:

  ```
  import sys
  print(sys.builtin_module_names)
  ```

  An error message is returned for Built-in-Modules.

MODULE SEARCH PATH

If you import a module, let's say "import abc", the interpreter searches for this module in the following locations and in the order given:

1. The directory of the top-level file, i.e. the file being executed.
2. The directories of PYTHONPATH, if this global variable is set.
3. standard installation path Linux/Unix e.g. in /usr/lib/python3.5.

It's possible to find out where a module is located after it has been imported:

```
>>> import numpy
>>> numpy.__file__
'/usr/lib/python3/dist-packages/numpy/__init__.py'
>>>
>>> import random
```

```
>>> random.__file__
'/usr/lib/python3.5/random.py'
>>>
```

The __file__ attribute doesn't always exists. This is the case with modules which are statically linked C libraries.

```
>>> import math
>>> math.__file__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__file__'
>>>
```

CONTENT OF A MODULE

With the built-in function dir() and the name of the module as an argument, you can list all valid attributes and methods for that module.

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd',
'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp',
'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Calling dir() without an argument, a list with the names in the current local scope is returned:

```
>>> import math
>>> cities = ["New York", "Toronto", "Berlin", "Washington",
"Amsterdam", "Hamburg"]
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__',
'__spec__', 'cities', 'math']
>>>
```

It's possible to get a list of the Built-in functions, exceptions, and other objects by importing the builtins module:

```
>>> 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__', '__loader__', '__name__',
'__package__', '__spec__',
'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']
>>>
```

PACKAGES

If you have created a lot of modules at some point in time, you may loose the overview about them. You may have dozens or hundreds of modules and they can be categorized into different categories. It is similar to the situation in a file system: Instead of having all files in just one directory, you put them into different ones, being organized according to the topics of the files. We will show in the next chapter of our Python tutorial how to organize modules into packages.