Python Essentials 2:
Module 1
Modules, Packages and PIP

In this module, you will learn about:

- importing and using Python modules;
- using some of the most useful Python standard library modules;
- constructing and using Python packages;

PIP (Python Installation Package) and how to use it to install and uninstall ready-to-use packages from PyPI.

What is a module?

Computer code has a tendency to grow. We can say that code that doesn't grow is probably completely unusable or abandoned. A real, wanted, and widely used code develops continuously, as both users' demands and users' expectations develop in their own rhythms.

A code which is not able to respond to users' needs will be forgotten quickly, and instantly replaced with a new, better, and more flexible code. Be prepared for this, and never think that any of your programs is eventually completed. The completion is a transition state and usually passes quickly, after the first bug report. Python itself is a good example how the rule acts.

Growing code is in fact a growing problem. A larger code always means tougher maintenance. Searching for bugs is always easier where the code is smaller (just as finding a mechanical breakage is simpler when the machinery is simpler and smaller).

Moreover, when the code being created is expected to be really big (you can use a total number of source lines as a useful, but not very accurate, measure of a code's size) you may want (or rather, you will be forced) to divide it into many parts, implemented in parallel by a few, a dozen, several dozen, or even several hundred individual developers.

Of course, this cannot be done using one large source file, which is edited by all programmers at the same time. This will surely lead to a spectacular disaster.

If you want such a software project to be completed successfully, you have to have the means allowing you to:

- divide all the tasks among the developers;
- join all the created parts into one working whole.

For example, a certain project can be divided into two main parts:

- the user interface (the part that communicates with the user using widgets and a graphical screen)
- the logic (the part processing data and producing results)

Each of these parts can be (most likely) divided into smaller ones, and so on. Such a process is often called decomposition.

For example, if you were asked to arrange a wedding, you wouldn't do everything yourself - you would find a number of professionals and split the task between them all.

How do you divide a piece of software into separate but cooperating parts? This is the question. Modules are the answer.

How to make use of a module?

So what is a module? The Python Tutorial defines it as a file containing Python definitions and statements, which can be later imported and used when necessary.

The handling of modules consists of two different issues:

- the first (probably the most common) happens when you want to use an already existing module, written by someone else, or created by yourself during your work on some complex project - in this case you are the module's user;
- the second occurs when you want to create a brand new module, either for your own use, or to make other programmers' lives easier - you are the module's supplier.

Let's discuss them separately.

First of all, a module is identified by its name. If you want to use any module, you need to know the name. A (rather large) number of modules is delivered together with Python itself. You can think of them as a kind of "Python extra equipment".

All these modules, along with the built-in functions, form the Python standard library - a special sort of library where modules play the roles of books (we can even say that folders play the roles of shelves). If you want to take a look at the full list of all "volumes" collected in that library, you can find it here:
https://docs.python.org/3/library/index.html.

Each module consists of entities (like a book consists of chapters). These entities can be functions, variables, constants, classes, and objects. If you know how to access a particular module, you can make use of any of the entities it stores.

Let's start the discussion with one of the most frequently used modules, named math. Its name speaks for itself - the module contains a rich collection of entities (not only functions) which enable a programmer to effectively implement calculations demanding the use of mathematical functions, like sin() or log().

Importing a module

To make a module usable, you must import it (think of it like of taking a book off the shelf). Importing a module is done by an instruction named import. Note: import is also a keyword (with all the consequences of this fact).

Let's assume that you want to use two entities provided by the math module:

- a symbol (constant) representing a precise (as precise as possible using double floating-point arithmetic) value of π (although using a Greek letter to name a variable is fully possible in Python, the symbol is named pi - it's a more convenient solution, especially for that part of the world which neither has nor is going to use a Greek keyboard)
- a function named sin() (the computer equivalent of the mathematical sine function)

Both these entities are available through the math module, but the way in which you can use them strongly depends on how the import has been done.

The simplest way to import a particular module is to use the import instruction as follows:

```
import math
```

The clause contains:

- the import keyword;
- the name of the module which is subject to import.

The instruction may be located anywhere in your code, but it must be placed before the first use of any of the module's entities.

If you want to (or have to) import more than one module, you can do it by repeating the import clause (preferred):

```
import math
import sys
```

or by listing the modules after the import keyword, like here:

```
import math, sys
```

The instruction imports two modules, first the one named math and then the second named sys.

The modules' list may be arbitrarily long.

Importing a module: continued

To continue, you need to become familiar with an important term: namespace. Don't worry, we won't go into great detail - this explanation is going to be as short as possible.

A namespace is a space (understood in a non-physical context) in which some names exist and the names don't conflict with each other (i.e., there are not two different objects of the same name). We can say that each social group is a namespace - the group tends to name each of its members in a unique way (e.g., parents won't give their children the same first names).

This uniqueness may be achieved in many ways, e.g., by using nicknames along with the first names (it will work inside a small group like a class in a school) or by assigning special identifiers to all members of the group (the US Social Security Number is a good example of such practice).

Inside a certain namespace, each name must remain unique. This may mean that some names may disappear when any other entity of an already known name enters the namespace. We'll show you how it works and how to control it, but first, let's return to imports.

If the module of a specified name exists and is accessible (a module is in fact a Python source file), Python imports its contents, i.e., all the names defined in the module become known, but they don't enter your code's namespace.

This means that you can have your own entities named sin or pi and they won't be affected by the import in any way.

At this point, you may be wondering how to access the pi coming from the math module.

To do this, you have to qualify the pi with the name of its original module.

Importing a module: continued

Look at the snippet below, this is the way in which you qualify the names of pi and sin with the name of its originating module:

math.pi
math.sin

It's simple, you put:

- the name of the module (e.g., math)
- a dot (i.e., .)
- the name of the entity (e.g., pi)

Such a form clearly indicates the namespace in which the name exists.

Note: using this qualification is compulsory if a module has been imported by the import module instruction. It doesn't matter if any of the names from your code and from the module's namespace are in conflict or not.

This first example won't be very advanced - we just want to print the value of sin(½π).

Look at the code in the editor. This is how we test it.

The code outputs the expected value: 1.0.

Note: removing any of the two qualifications will make the code erroneous. There is no other way to enter math's namespace if you did the following:

import math

editor:
import math
print(math.sin(math.pi/2))

Importing a module: continued

Now we're going to show you how the two namespaces (yours and the module's one) can coexist.

Take a look at the example in the editor window.

```
import math


def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None


pi = 3.14

print(sin(pi/2))
print(math.sin(math.pi/2))
```

We've defined our own pi and sin here.

Run the program. The code should produce the following output:

```
0.99999999
1.0
output
```

As you can see, the entities don't affect each other.

Importing a module: continued

In the second method, the import's syntax precisely points out which module's entity (or entities) are acceptable in the code:

```
from math import pi
```

The instruction consists of the following elements:

- the from keyword;
- the name of the module to be (selectively) imported;
- the import keyword;
- the name or list of names of the entity/entities which are being imported into the namespace.

The instruction has this effect:

- the listed entities (and only those ones) are imported from the indicated module;
- the names of the imported entities are accessible without qualification.

Note: no other entities are imported. Moreover, you cannot import additional entities using a qualification - a line like this one:

```
print(math.e)
```

will cause an error (e is Euler's number: 2.71828...)

Let's rewrite the previous script to incorporate the new technique.

Here it is:

```
from math import sin, pi

print(sin(pi/2))
```

The output should be the same as previously, as in fact we've used the same entities as before: 1.0. Copy the code, paste it in the editor, and run the program.

Does the code look simpler? Maybe, but the look is not the only effect of this kind of import. Let's show you that.

Importing a module: continued

Look at the code in the editor. Analyze it carefully:

- line 1: carry out the selective import;
- line 3: make use of the imported entities and get the expected result (1.0)
- lines 5 through 12: redefine the meaning of pi and sin - in effect, they supersede the original (imported) definitions within the code's namespace;
- line 15: get 0.99999999, which confirms our conclusions.

Let's do another test. Look at the code below:

```
pi = 3.14


def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None


print(sin(pi / 2))

from math import sin, pi

print(sin(pi / 2))
```

Here, we've reversed the sequence of the code's operations:

- lines 1 through 8: define our own pi and sin;
- line 11: make use of them (0.99999999 appears on the screen)
- line 13: carry out the import - the imported symbols supersede their previous definitions within the namespace;
- line 15: get 1.0 as a result.

Importing a module: *

In the third method, the import's syntax is a more aggressive form of the previously presented one:

from module import *

As you can see, the name of an entity (or the list of entities' names) is replaced with a single asterisk (*).

Such an instruction imports all entities from the indicated module.

Is it convenient? Yes, it is, as it relieves you of the duty of enumerating all the names you need.

Is it unsafe? Yes, it is - unless you know all the names provided by the module, you may not be able to avoid name conflicts. Treat this as a temporary solution, and try not to use it in regular code.

Importing a module: the as keyword

If you use the import module variant and you don't like a particular module's name (e.g., it's the same as one of your already defined entities, so qualification becomes troublesome) you can give it any name you like - this is called aliasing.

Aliasing causes the module to be identified under a different name than the original. This may shorten the qualified names, too.

Creating an alias is done together with importing the module, and demands the following form of the import instruction:

import module as alias

The "module" identifies the original module's name while the "alias" is the name you wish to use instead of the original.

Note: as is a keyword.

Importing a module: continued

If you need to change the word math, you can introduce your own name, just like in the example:

```
import math as m

print(m.sin(m.pi/2))
```

Note: after successful execution of an aliased import, the original module name becomes inaccessible and must not be used.

In turn, when you use the from module import name variant and you need to change the entity's name, you make an alias for the entity. This will cause the name to be replaced by the alias you choose.

This is how it can be done:

```
from module import name as alias
```

As previously, the original (unaliased) name becomes inaccessible.

The phrase name as alias can be repeated - use commas to separate the multiplied phrases, like this:

```
from module import n as a, m as b, o as c
```

The example may look a bit weird, but it works:

```
from math import pi as PI, sin as sine

print(sine(PI/2))
```

Now you're familiar with the basics of using modules. Let us show you some modules and some of their useful entities.

Key takeaways

1. If you want to import a module as a whole, you can do it using the import module_name statement. You are allowed to import more than one module at once using a comma-separated list. For example:

```
import mod1
import mod2, mod3, mod4
```

although the latter form is not recommended due to stylistic reasons, and it's better and prettier to express the same intention in more a verbose and explicit form, such as:

```
import mod2
import mod3
import mod4
```

2. If a module is imported in the above manner and you want to access any of its entities, you need to prefix the entity's name using dot notation. For example:

```
import my_module

result = my_module.my_function(my_module.my_data)
```

The snippet makes use of two entities coming from the my_module module: a function named my_function() and a variable named my_data. Both names must be prefixed by my_module. None of the imported entity names conflicts with the identical names existing in your code's namespace.

3. You are allowed not only to import a module as a whole, but to import only individual entities from it. In this case, the imported entities must not be prefixed when used. For example:

```
from module import my_function, my_data

result = my_function(my_data)
```

The above way - despite its attractiveness - is not recommended because of the danger of causing conflicts with names derived from importing the code's namespace.

4. The most general form of the above statement allows you to import all entities offered by a module:

```
from my_module import *

result = my_function(my_data)
```

Note: this import's variant is not recommended due to the same reasons as previously (the threat of a naming conflict is even more dangerous here).

5. You can change the name of the imported entity "on the fly" by using the as phrase of the import. For example:
from module import my_function as fun, my_data as dat

result = fun(dat)

Exercise 1

You want to invoke the function make_money() contained in the module named mint. Your code begins with the following line:

import mint

What is the proper form of the function's invocation?

Check

Exercise 2

You want to invoke the function make_money() contained in the module named mint. Your code begins with the following line:

from mint import make_money

What is the proper form of the function's invocation?

Check

Exercise 3

You've written a function named make_money on your own. You need to import a function of the same name from the mint module and don't want to rename any of your previously defined names. Which variant of the import statement may help you with the issue?

Check

Exercise 4

What form of the make_money function invocation is valid if your code starts with the following line?

from mint import *

Check