Modules in Python

There are three different ways to define a module in Python:

There are three different ways to define a module in Python:

- **A module can be written in Python itself.**

There are three different ways to define a module in Python:

- **A module can be written in Python itself.**
- A module can be written in C and loaded dynamically at run-time, like the `re` (regular expression) module.

There are three different ways to define a module in Python:

- **A module can be written in Python itself.**
- A module can be written in C and loaded dynamically at run-time, like the `re` (regular expression) module.
- A built-in module is loaded automatically as the interpreter starts and is always available.
  Example: `OS` Module (`import os`). It provides functions for manipulating directories, files, and so on.

There are three different ways to define a module in Python:

- **A module can be written in Python itself.**
- A module can be written in C and loaded dynamically at run-time, like the `re` (regular expression) module.
- A built-in module is loaded automatically as the interpreter starts and is always available.
  Example: `OS` Module (`import os`). It provides functions for manipulating directories, files, and so on.

Our focus will be on modules that are written in Python. All you need to do is create a `.py` file that contains Python code.

Conventions

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability.

Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

https://www.python.org/dev/peps/pep-0008/#package-and-module-names

Example:

A module can contain the functions to build palindrome strings.

The module defines several functions and variables

```
# variables
name
_secret_name

# functions
_is_palindrome1( )
_is_palindrome( )
_split_palindrome_nonpalindrome( )

create_palindrome( )
```

INTERFACE OF A MODULE

Interface

`name, create_palindrome( )`

Secrets of the module

`_secret_name, _is_palindrome( ), _is_palindrome1( ), _split_palindrome_nonpalindrome( )`

The underscore prefix (_) means that a variable or function is intended for **internal use** only. It is merely a **hint to another programmer** that should avoid to use `_secret_name` and `_is_palindrome( )`.
This behavior is generally **not enforced** by the Python interpreter.

`mainprogram` **can** call `mymodule._private_function( )` and `mymodule._var`, but this practice is **strongly discouraged**.

Assuming `mymodule.py` is in an ***appropriate location***, the ***client module*** can access to its objects by importing the module as follows:

```
# mainprogram.py
import mymodule

print (mymodule.var)
print (mymodule.f(2, 3))
```

Modules help avoid collisions between names:

You can define a function `f()` both in in the `mainprogram.py` and in a module, and you can use them with no conflict.

```
# mainprogram.py
import mymodule

def f():
    # ...

a = mymodule.f() # from mymodule.py
b = f()  # from mainprogram.py
```

Don't use direclty 'private' objects - they are a secret of the module.

The programmer could completely **change the implementation of the module**, while leaving the interface unchanged.

If we rely on an implementation detail of the server module, **if the implementation changes**, the client code may no longer work.

Example:

Function `_is_palindrome( )` could be deleted from the server module, and the functionality can be implemented directly by the instruction
`s == s[::-1]`
in another function.

Or the programmer can change the name of `_is_palindrome( )` into other name.
If the client code uses `_is_palindrome( )`, it won't work anymore.

Executing a Module as a Script

Any `.py` file that contains a module is essentially a Python script, so we can run it as a script. **But** we get the same output when we just *import the module*.
The action of importing the module entails the execution of the code.

Module

```
def f(x):
    return x+1

print ('I am a module!')
print (f(1))
```

Output

```
I am a module!
2
```

The special variable `__name__`

You can control what code to execute when you run the module as a script.

When a .py file is **imported as a module**, Python sets the special **dunder**[1] variable `__name__` to the name of the module.

When a file is run as a standalone script, `__name__` is set to the string `__main__`.

---

[1] Dunder here means "Double Underscores"

```python
# my_module.py

print(__name__)
```

```python
# import only
import my_module
```

OUTPUT:

```
my_module
```

```python
# run as a script
python my_module.py
```

OUTPUT:

```
__main__
```

Using this fact, you can discern 'run' actions from 'import' actions.

```
# FILE my_module.py
# ...
if (__name__ == '__main__'):
    print('Executing as standalone script')
    print(' example of use')
    # ...
```

Modules are often designed with the capability to run as a standalone script for **purposes of explain how to use it** or **testing its functionality.**

EXERCISE

- Create a module `fact.py` containing a `factorial()` function.
- When you **import** the module, it prints "I am a module", and it provides the `factorial()` function to the client module.
- When you **run** the module as a standalone script, i.e.
  `python fact.py n`
  it prints "run as a standalone script", and prints the factorial of `n`

`sys.argv` is a list which contains the command-line arguments passed to the script.
`sys.argv[0]` is the name of the script
`sys.argv[1]` is the first argument

Write the module and a program that imports and uses the module.

Solution: `factorial_1.py`

EXERCISE

Write a module that provides the functionality to build a palindrome string ('palindrome' exercise).

Write a module that provides the functionality to store a time slot ('time slot' exercise).

Write a module that contains

- a function `computeRealRoots(a, b, c)` that computes real roots of the quadratic equation ($ax^2 + bx + c = 0$) using the quadratic formula (https://en.wikipedia.org/wiki/Quadratic_formula)
- a function `_computeDelta()` that compute the `discriminant` Δ

Modules must test itself if you run them as a script.

PYTHON PACKAGES

PYTHON PACKAGES

- In a very large application that includes many modules, as the number of modules grows, it becomes difficult to keep track of all of them. Remember that a single `.py` file (a module, in this meaning), can contain several 'public' elements (functions, variables, and so on), and several 'private' elements.

PYTHON PACKAGES

- In a very large application that includes many modules, as the number of modules grows, it becomes difficult to keep track of all of them. Remember that a single `.py` file (a module, in this meaning), can contain several 'public' elements (functions, variables, and so on), and several 'private' elements.

- Packages are merely **directories**. Packages allow you to organize and group modules. They allow for a hierarchical structuring of the module namespace using **dot notation** (i.e. `pkg.mod1`). To create a package, simply create a **directory** and insert the `.py` modules (*files*) in it.

PYTHON PACKAGES

PYTHON PACKAGES

- In the same way that modules help avoid collisions between global variable names, packages help avoid collisions between module names.

PYTHON PACKAGES

- In the same way that modules help avoid collisions between global variable names, packages help avoid collisions between module names.
- Old documentation states that an `__init__.py` file **must be present** in the package directory when creating a package. Starting with Python 3.3, it is possible to create a package without `__init__.py` file.

EXERCISE

1) Write a package `mypkg` that contains `module1`, with a function `bar()`, and `module2`, with a function `foo()`.
Each function must print its name.
Import and use the functions.

2) Open a new instance of the python interpreter
Try to **import the package** `mypkg`. What happens?
After that you import **only** the package `mypkg`, can you use the functions or the modules?

Package Initialization

If a file named **`__init__.py`** is present in a package directory, it is invoked automatically when **the package** or **a module in the package** is imported.

This can be used for execution of **package initialization code**, such as initialization of package-level data. In this way, a package provides not only functions but also data.

`__init__.py` can be used also to **automatic import modules from a package.**

EXERCISE

Write a package **`mypkg`** that contains

`module1`, with a function `bar()`, and `module2`, with a function `_secret()`. Each function must print its name.

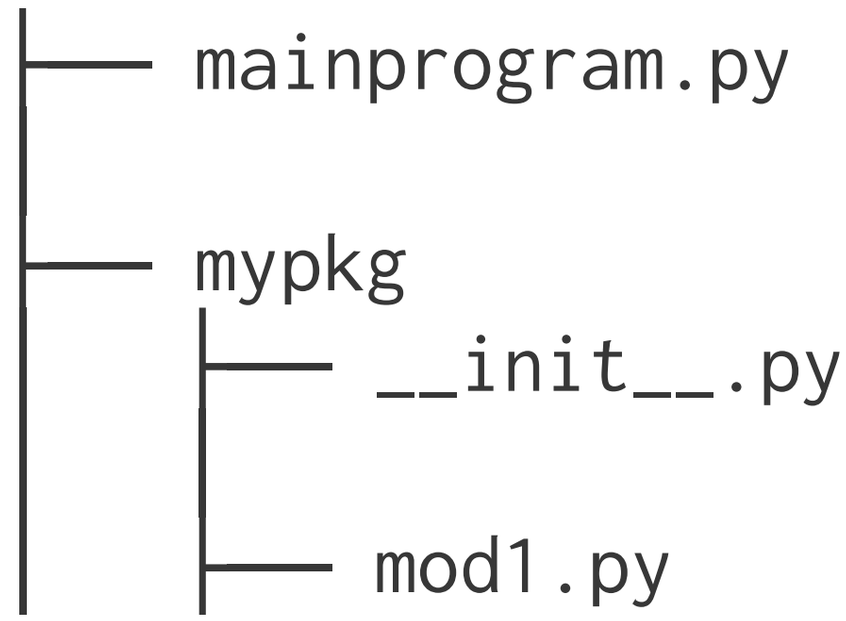
When you import the package **`mypkg`**,

- the package must print a welcome message and it must initialize the list `mypkg.myList=['a','b', 'c']`
- modules `module1` and `module2` are imported automatically.

Write a client (`main.py`) that use the `bar()` and `_secret()`.

Different ways to import a module

Consider the following situation:



`mod1.py` contains `bar1()` and `foo1()` functions.

IMPORT

Instruction	Effect
`import mypkg`	imports the symbol `mypkg`
`import mypkg.mod1`	`mypkg.mod1.bar1()` `mypkg.mod1.foo1()`
`from mypkg import mod1`	`mod1.foo1()` `mod1.bar1()`
`from mypkg.mod1 import foo1`	`foo1()` ***(in the namespace of the client module)***

AS

Instruction	Effect
`import mypkg.mod1 as mymodule`	`mymodule.foo1()` `mymodule.bar1()`
`from mypkg import mod1 as mymodule`	`mymodule.foo1()` `mymodule.bar1()`
`from mypkg.mod1 import foo1 as myfunction`	`myfunction()` ***(in the namespace of the client module)***

ERROR

Instruction	Error
`import mypkg.mod1.foo1`	ERROR: No module named '`mypkg.mod1.foo1`' *foo1 is a function, not a module*
`from mypkg.mod1 import foo1` `bar1()`	ERROR - we imported `foo1()` only

`from mypkg.mod1 import foo1`
`as myfunction`
`foo1()`
	ERROR: `foo1` is not defined we imported `foo1` with the name `myfunction`

It is even possible to indiscriminately import *everything* from a module:

```
from mypkg.mod1 import *
foo1()
bar1()
```

This will place the names of all objects from `mypkg.mod1` into the local symbol table, **except names that begin with the underscore** `(_)` character.

This form is not recommended, because you are entering names into the local symbol table. You could overwriting an existing name inadvertently.

The analogous statement for a package is this:

```
from mypkg import *
```

Despite what we could expect, this instruction **does not import the modules of the package**.

If the `__init__.py` file in the package directory contains a list named `__all__`, this `inport` statement imports all modules in the list.

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; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports the names defined in the package.

The Module Search Path

When the interpreter executes

```
import mod
```

it searches for `mod.py` in a list of directories:

- the current directory (the directory from which the input script was run)
- the directories contained in the `PYTHONPATH` environment variable
- an installation-dependent list of directories

The search path is accessible in the Python variable `sys.path`

```
import sys
sys.path
```

```
['', '/Users/username/anaconda3/lib/python37.zip', '/
Users/username/anaconda3/lib/python3.7']
```

Note: The exact contents of sys.path are installation-dependent

It is possible to modify `sys.path` at run-time so that it contains your module directory.

```
sys.path.append('my_module_dir')
```

The `dir()` function

The built-in function `dir()` returns a list of **defined names** in a namespace.

Try `dir()` **before** and **after** declaring a variable or importing a module.

`dir()` can be useful for identifying what exactly has been added to the namespace by an import statement.

When given an argument that is the name of a module, `dir()` lists the names defined in the module.

The `help()` function

Python `help()` function is used to get the documentation of specified module, class, function, variables etc. This method is generally used with python interpreter console to get details about python objects.

We can define `help()` function output for our custom classes and functions by defining docstring (documentation string). By default, the first comment string in the body of a method is used as its docstring. It’s surrounded by three double quotes.

See also:

https://docs.python.org/3/reference/import.html
https://docs.python.org/3/tutorial/modules.html