

Python 3

> import

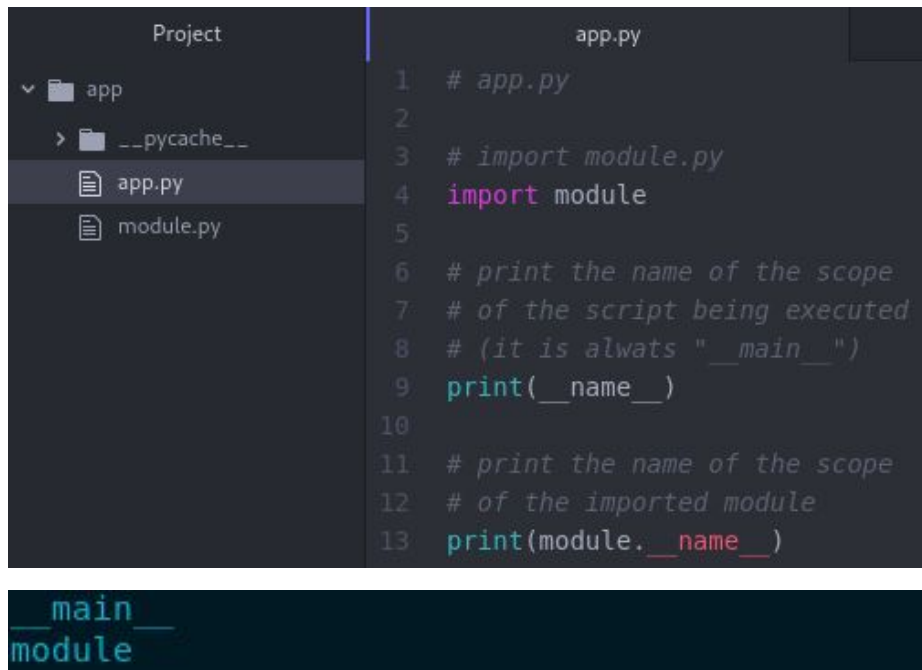
Modules (I)

— — —

Un **módulo** es un archivo que contiene definiciones y sentencias de Python.

El nombre del archivo del **módulo** es el nombre del **módulo** con el sufijo `.py` agregado.

Dentro de un **módulo**, el nombre del **módulo** (como `string`) esta disponible como la variable global `__name__`.



The image shows a code editor interface. On the left, a 'Project' pane displays a file tree with a folder 'app' containing files 'app.py' and 'module.py'. On the right, the 'app.py' file is open, showing Python code. The code includes comments and an import statement. At the bottom, a dark bar displays the value of the `__name__` variable.

```
Project
└─ app
   └─ __pycache__
      └─ app.py
      └─ module.py

app.py
1  # app.py
2
3  # import module.py
4  import module
5
6  # print the name of the scope
7  # of the script being executed
8  # (it is always "__main__")
9  print(__name__)
10
11 # print the name of the scope
12 # of the imported module
13 print(module.__name__)

__main__
module
```

Modules (II)

— — —



```
Project
└─ app
   └─ __pycache__
      └─ a.py
         app.py
         b.py

app.py
1 # app.py
2
3 import a, b
4
5
6

a.py
1 # a.py
2
3 print("I'm module A")
4
5
6

b.py
1 # b.py
2
3 import a
4
5 print("I'm module B")
6
```

Cuando se **importa** un **módulo** por primera vez, el intérprete ejecuta las sentencias del **módulo**.

Si el **módulo** es importado nuevamente sus sentencias no serán ejecutadas.

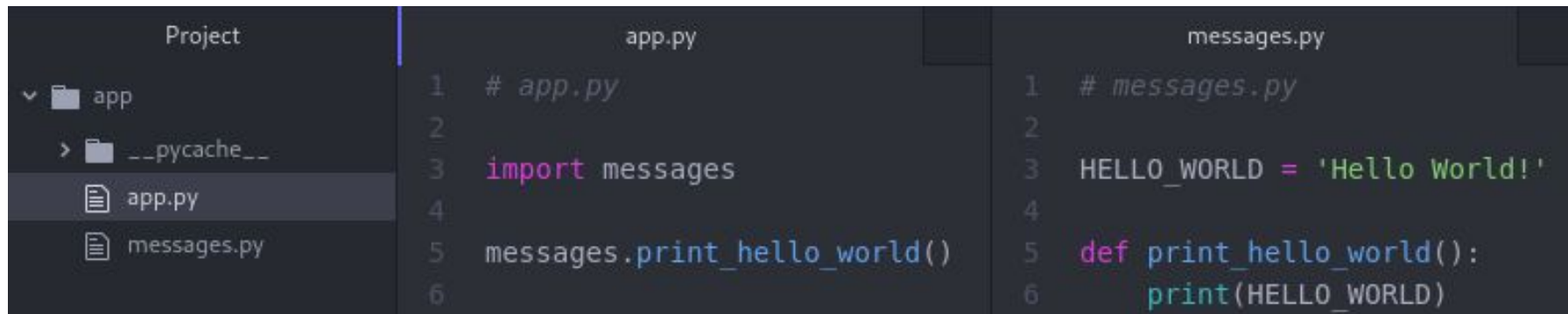
```
I'm module A
I'm module B
```

Scopes (I)

— — —

Cada **módulo** ejecuta sus sentencias en su propio scope (que es visto como global desde el propio **módulo**).

Al **importar** un **módulo** se introduce una referencia en el scope actual. Los símbolos definidos dentro del **módulo** se pueden acceder utilizando la sintaxis `module.symbol`.



```
Project
├── app
│   ├── __pycache__
│   ├── app.py
│   └── messages.py
└── app.py
    1 # app.py
    2
    3 import messages
    4
    5 messages.print_hello_world()
    6

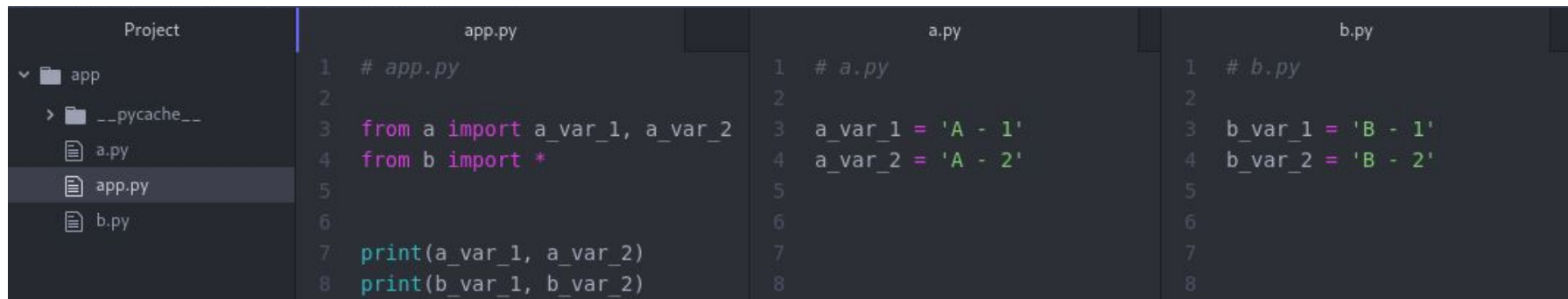
messages.py
    1 # messages.py
    2
    3 HELLO_WORLD = 'Hello World!'
    4
    5 def print_hello_world():
    6     print(HELLO_WORLD)
```

Scopes (II)

— — —

Utilizando la sentencia `from ... import ...` se pueden **importar** los símbolos de un **módulo** directamente en el scope actual.

Utilizando `from ... import *` se van a **importar** todos los símbolos que no comiencen con `_` (o los símbolos definidos por `__all__` si está definido). Esta variante del `import` no se considera buena práctica.



```
Project
└─ app
   └─ __pycache__
      └─ a.py
      └─ app.py
      └─ b.py

# app.py
1
2
3 from a import a_var_1, a_var_2
4 from b import *
5
6
7 print(a_var_1, a_var_2)
8 print(b_var_1, b_var_2)

# a.py
1
2
3 a_var_1 = 'A - 1'
4 a_var_2 = 'A - 2'
5
6
7
8

# b.py
1
2
3 b_var_1 = 'B - 1'
4 b_var_2 = 'B - 2'
5
6
7
8
```

Ejecutando módulos

A veces es útil definir **módulos** que pueden ser ejecutados individualmente como scripts.

Se pueden realizar distintas acciones dependiendo de si el **módulo** fue **importado** o ejecutado directamente.

```
1  # file_ops.py
2
3  import sys
4
5  def read_and_print(path):
6      with open(path) as f:
7          print(f.read())
8
9  # if the module is executed as a script
10 # read and print the file passed as
11 # argument
12 if __name__ == '__main__':
13     read_and_print(sys.argv[1])
```

```
$ python3 file_ops.py hello.txt
Hello!
```

```
>>> import file_ops
>>> file_ops.read_and_print('hello.txt')
Hello!
```

¿Donde se buscan los módulos?

— — —

Cuando se trata de **importar** un **módulo** primero se buscará en los **módulos** built-in de Python.

Luego, si no es encontrado, se buscará en la lista de directorios definidos por `sys.path`.

La lista de `sys.path` está constituida por los siguientes directorios (en orden de precedencia):

- El directorio que contiene el script que se ejecutó (o el directorio actual si se invocó el intérprete sin un script).
- Los directorios definidos por la variable de entorno `PYTHONPATH`.
- Una lista de directorios por default que dependen de la instalación.

Packages (I)

— — —

Los **paquetes** son **módulos** que sirven para agrupar **módulos** en un mismo namespace.

Los archivos `__init__.py` son necesarios para que Python reconozca un directorio como **paquete**. El archivo `__init__.py` será ejecutado al importar el **paquete** y puede estar vacío.

sound/ __init__.py formats/ __init__.py wavread.py wavwrite.py aiffread.py aiffwrite.py auread.py auwrite.py ...	Top-level package Initialize the sound package Subpackage for file format conversions
effects/ __init__.py echo.py surround.py reverse.py ...	Subpackage for sound effects
filters/ __init__.py equalizer.py vocoder.py karaoke.py ...	Subpackage for filters

Project	app_a.py	app_b.py
<div>Project Explorer</div> <ul style="list-style-type: none">app<ul style="list-style-type: none">__pycache__package<ul style="list-style-type: none">__pycache____init__.pymodule.pyapp_a.pyapp_b.py	<pre>1 # app_a.py 2 3 import package.module 4 5 package.module.foo() 6 7 print(package.var) 8</pre> <div>package initializing... module initializing... bar somevar</div>	<pre>1 # app_b.py 2 3 from package import module 4 5 module.foo() 6 7 print(package.var) 8</pre> <div>package initializing... module initializing... bar Traceback (most recent call last): File "app_b.py", line 7, in <module> print(package.var) NameError: name 'package' is not defined</div>
	<pre>__init__.py 1 # package/__init__.py 2 3 print('package initializing...') 4 5 var = 'somevar' 6</pre>	<pre>module.py 1 # package/module.py 2 3 print('module initializing...') 4 5 def foo(): 6 print('bar')</pre>

Packages (II)

Packages (III)

— — —

Dentro de un **módulo** o **subpaquete** perteneciente a un **paquete** se pueden hacer **imports** internos.

Los **imports** pueden tener la forma absoluta o relativa (recomendada).

```
1  # effects/surround.py
2
3  # absolute import
4  from sound.effects import reverse
5
6  # relative imports
7  from . import echo
8  from .. import formats
9  from ..filters import equalizer
10
11 # ...
```

```
sound/
  __init__.py
  formats/
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...
```

Packages (IV)

Importar un **paquete** utilizando la sentencia `from <package> import *` puede que no tenga el resultado esperado.

Uno puede esperar que se importen todos los **módulos** o **subpaquetes** definidos dentro del paquete pero esto sólo se importará los símbolos definidos en el archivo `__init__.py`.

Para que un **paquete** importe sus **módulos** o **subpaquetes** de forma automática al usar `from <package> import *` el autor del **paquete** los puede agregar a la lista `__all__` en `__init__.py`. También puede usar sentencias `import` para lograr esto.

```
1  # __init__.py
2
3  # this will import module_a and module_b
4  # when from <package> import * is used
5  __all__ = ['module_a', 'module_b']
6
7  # this will import module_c when
8  # from <package> import * is used
9  from . import module_c
```

Utilizando alias

Los **módulos** o **paquetes** se pueden importar al scope actual utilizando otro símbolo en vez de su nombre.

A esto se lo llama utilizar un **alias** y sirve para utilizar símbolos más simples o evitar colisiones dentro del scope.

```
1 import package as p
2 from package import module_a as a, module_b as b
3 import package.module_c as c
```

Módulos built-in

Para ver una lista de los **módulos** que vienen incluidos en Python se puede ver el [Python Module Index](#).

La lista de **módulos** built-in disponibles varía de acuerdo a la instalación. A pesar de esto hay algunos **módulos** que están disponibles siempre (como el módulo `sys`).



`code()`

Ejercicios 11

- Crear una aplicación que cada 5 segundos imprima la hora actual con el formato “<hh> horas, <mm> minutos, <ss> segundos”.
- Crear un módulo que defina una función para convertir un archivo CSV a otro de tipo JSON. El archivo JSON producido deberá contener una lista de objetos, donde cada objeto represente una fila usando los nombres de columnas como keys y sus valores como values. Importar el módulo y utilizar la función desde otro script.

Bibliografía

- [Python docs](#)

— — —