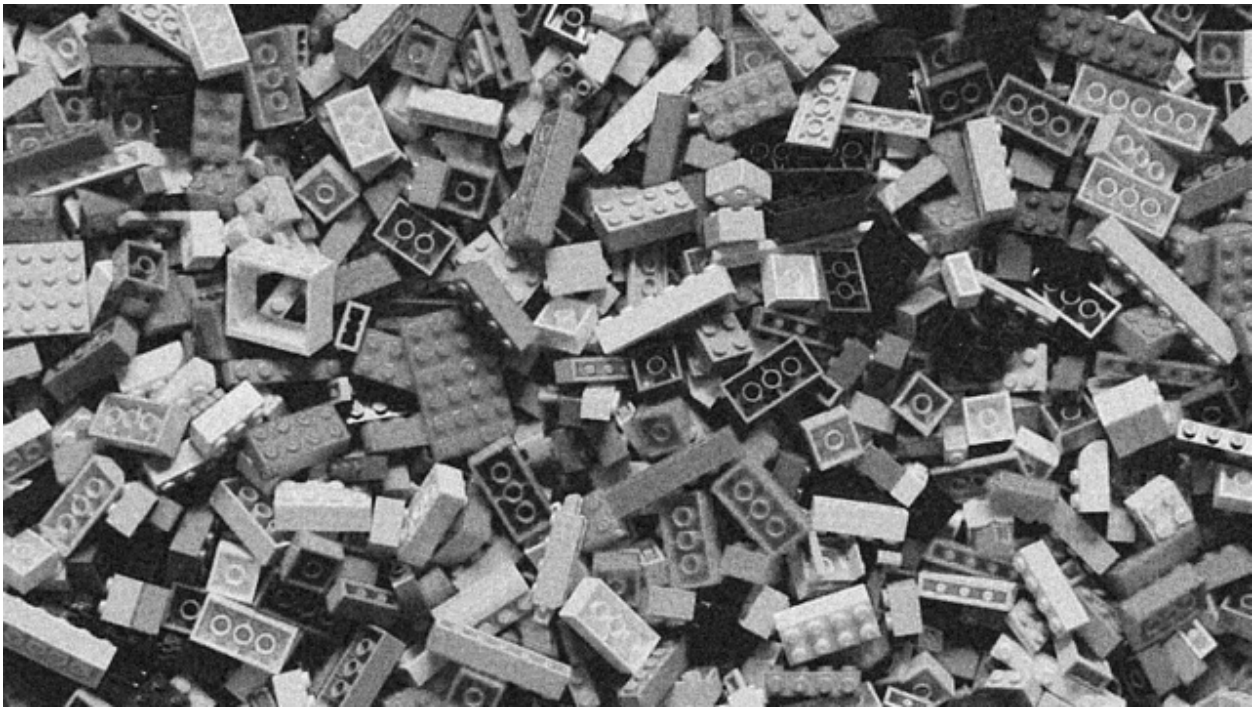
6.4 Módulos



Escribir pequeños trozos de código puede resultar interesante para realizar determinadas pruebas. Pero a la larga, nuestros programas tenderán a crecer y será necesario agrupar el código en unidades manejables.

Los **módulos** son simplemente ficheros de texto que contienen código Python y representan unidades con las que *evitar la repetición* y *favorecer la reutilización*.[1]

6.4.1 Importar un módulo

Para hacer uso del código de otros módulos usaremos la sentencia `import`. Esto permite importar el código y las variables de dicho módulo para que estén disponibles en nuestro programa.

La forma más sencilla de importar un módulo es `import <module>` donde `module` es el nombre de otro fichero Python, sin la extensión `.py`.

Supongamos que partimos del siguiente fichero (*módulo*):

`arith.py`

```
1 def addere(a, b):
2     '''Sum of input values'''
```

(continué en la próxima página)

[1] Foto original por Xavi Cabrera en Unsplash.

(proviene de la página anterior)

```
3       return a + b
4
5
6   def minuas(a, b):
7       '''Substract of input values'''
8       return a - b
9
10
11  def pullulate(a, b):
12      '''Product of input values'''
13      return a * b
14
15
16  def partitus(a, b):
17      '''Division of input values'''
18      return a / b
```

Desde otro fichero - en principio en la misma carpeta - podríamos hacer uso de las funciones definidas en `arith.py`.

Importar módulo completo

Desde otro fichero haríamos lo siguiente para importar todo el contenido del módulo `arith.py`:

```
1   >>> import arith
2
3   >>> arith.addere(3, 7)
4   10
```

Nota: Nótese que en la **línea 3** debemos anteponer a la función `addere()` el *espacio de nombres* que define el módulo `arith`.

Ruta de búsqueda de módulos

Python tiene 2 formas de encontrar un módulo:

1. En la carpeta actual de trabajo.
2. En las rutas definidas en la variable de entorno `PYTHONPATH`.

Para ver las rutas de búsqueda establecidas, podemos ejecutar lo siguiente en un intérprete de Python:

```
>>> import sys

>>> sys.path
['/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',
'/path/to/.pyenv/versions/3.9.1/lib/python3.9',
'/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',
'']
```

La cadena vacía que existe al final de la lista hace referencia a la **carpeta actual**.

Modificando la ruta de búsqueda

Si queremos modificar la ruta de búsqueda, existen dos opciones:

Modificando directamente la variable `PYTHONPATH` Para ello exportamos dicha variable de entorno desde una terminal:

```
$ export PYTHONPATH=/tmp
```

Y comprobamos que se ha modificado en `sys.path`:

```
>>> sys.path
['/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',
 '/tmp',
'/path/to/.pyenv/versions/3.9.1/lib/python3.9',
'/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',
'']
```

Modificando directamente la lista `sys.path` Para ello accedemos a lista que está en el módulo `sys` de la librería estandar:

```
>>> sys.path.append('/tmp')  # añadimos al final

>>> sys.path
['/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',
'/path/to/.pyenv/versions/3.9.1/lib/python3.9',
'/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',
'',
'/tmp']
```

```
>>> sys.path.insert(0, '/tmp')  # insertamos por el principio

>>> sys.path
['/tmp',
'/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'/path/to/.pyenv/versions/3.9.1/lib/python3.9',
'/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',
'']
```

Truco: El hecho de poner nuestra ruta al principio o al final de `sys.path` influye en la búsqueda, ya que si existen dos (o más módulos) que se llaman igual en nuestra ruta de búsqueda, Python usará el primero que encuentre.

Importar partes de un módulo

Es posible que no necesitemos todo aquello que está definido en `arith.py`. Supongamos que sólo vamos a realizar divisiones. Para ello haremos lo siguiente:

```
1 >>> from arith import partitus
2
3 >>> partitus(5, 2)
4 2.5
```

Nota: Nótese que en la **línea 3** ya podemos hacer uso directamente de la función `partitus()` porque la hemos importado directamente. Este esquema tiene el inconveniente de la posible **colisión de nombres**, en aquellos casos en los que tuviéramos algún objeto con el mismo nombre que el objeto que estamos importando.

Importar usando un alias

Hay ocasiones en las que interesa, por colisión de otros nombres o por mejorar la legibilidad, usar un nombre diferente del módulo (u objeto) que estamos importando. Python nos ofrece esta posibilidad a través de la sentencia `as`.

Supongamos que queremos importar la función del ejemplo anterior pero con otro nombre:

```
>>> from arith import partitus as mydivision

>>> mydivision(5, 2)
2.5
```

6.4.2 Paquetes

Un **paquete** es simplemente una **carpeta** que contiene ficheros `.py`. Además permite tener una jerarquía con más de un nivel de subcarpetas anidadas.

Para ejemplificar este modelo vamos a crear un paquete llamado `mymath` que contendrá 2 módulos:

- `arith.py` para operaciones aritméticas (ya visto *anteriormente*).
- `logic.py` para operaciones lógicas.

El código del módulo de operaciones lógicas es el siguiente:

`logic.py`

```
1  def et(a, b):
2      '''Logic "and" of input values'''
3      return a & b
4
5
6  def uel(a, b):
7      '''Logic "or" of input values'''
8      return a | b
9
10
11 def vel(a, b):
12     '''Logic "xor" of input values'''
13     return a ^ b
```

Si nuestro código principal va a estar en un fichero `main.py` (*a primer nivel*), la estructura de ficheros nos quedaría tal que así:

```
1  .
2  □□□ main.py
3  □□□ mymath
4      □□□ arith.py
5      □□□ logic.py
6
7  1 directory, 3 files
```

Línea 2 Punto de entrada de nuestro programa a partir del fichero `main.py`

Línea 3 Carpeta que define el paquete `mymath`.

Línea 4 Módulo para operaciones aritméticas.

Línea 5 Módulo para operaciones lógicas.

Importar desde un paquete

Si ya estamos en el fichero `main.py` (o a ese nivel) podremos hacer uso de nuestro paquete de la siguiente forma:

```
1 >>> from mymath import arith, logic
2
3 >>> arith.pullulate(4, 7)
4 28
5
6 >>> logic.et(1, 0)
7 0
```

Línea 1 Importar los módulos `arith` y `logic` del paquete `mymath`

Línea 3 Uso de la función `pullulate` que está definida en el módulo `arith`

Línea 5 Uso de la función `et` que está definida en el módulo `logic`

6.4.3 Programa principal

Cuando decidimos desarrollar una pieza de software en Python, normalmente usamos distintos ficheros para ello. Algunos de esos ficheros se convertirán en *módulos*, otros se englobarán en *paquetes* y existirá uno en concreto que será nuestro **punto de entrada**, también llamado **programa principal**.

Consejo: Suele ser una buena práctica llamar `main.py` al fichero que contiene nuestro programa principal.

La estructura que suele tener este *programa principal* es la siguiente:

```
# imports de la librería estándar
# imports de librerías de terceros
# imports de módulos propios

# CÓDIGO PROPIO
# ...
# CÓDIGO PROPIO

if __name__ == '__main__':
    # punto de entrada real
```

Importante: Si queremos ejecutar este fichero `main.py` desde línea de comandos, tendríamos que hacer:

```
$ python3 main.py
```

`if __name__ == '__main__'`

Esta condición permite, en el programa principal, diferenciar qué codigo se lanzará cuando el fichero se ejecuta directamente o cuando el fichero se importa desde otro lugar.

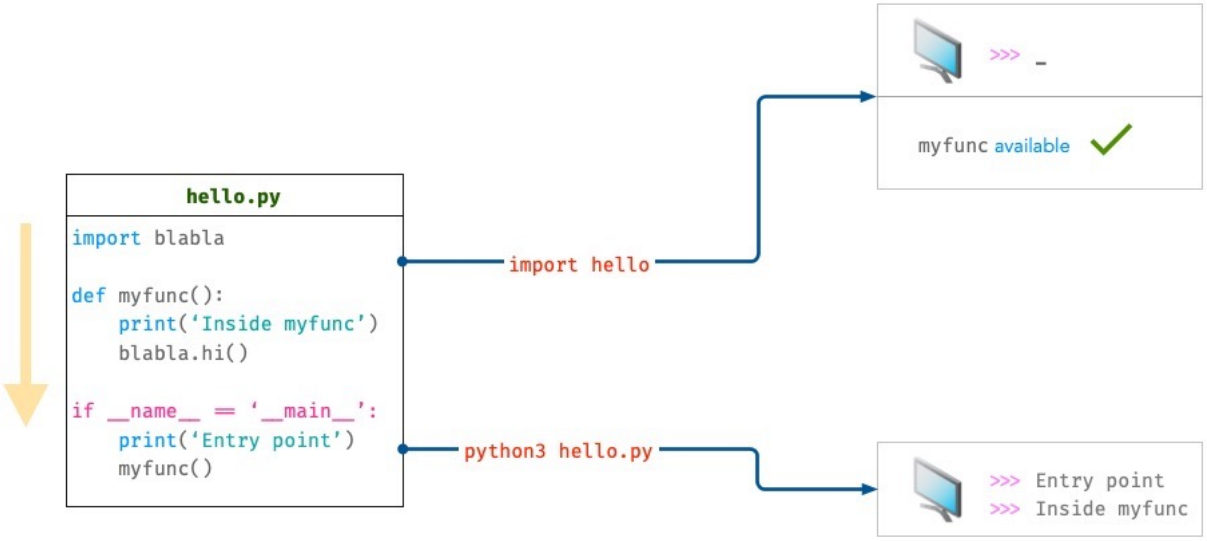


Figura 16: Comportamiento de un programa principal al importarlo o ejecutarlo

`hello.py`

```
1  import blabla
2
3
4  def myfunc():
5      print('Inside myfunc')
6      blabla.hi()
7
8
9  if __name__ == '__main__':
10     print('Entry point')
11     myfunc()
```

`import hello` El código se ejecuta siempre desde la primera instrucción a la última:

- **Línea 1**: se importa el módulo `blabla`.
- **Línea 4**: se define la función `myfunc()` y estará disponible para usarse.

- **Línea 9**: esta condición **no** se cumple, ya que estamos importando y la variable especial `__name__` no toma ese valor. Con lo cual finaliza la ejecución.
- *No hay salida por pantalla.*

`$ python3 hello.py` El código se ejecuta siempre desde la primera instrucción a la última:

- **Línea 1**: se importa el módulo `blabla`.
- **Línea 4**: se define la función `myfunc()` y estará disponible para usarse.
- **Línea 9**: esta condición **sí** se cumple, ya que estamos ejecutando directamente el fichero (*como programa principal*) y la variable especial `__name__` toma el valor `__main__`.
- **Línea 10**: salida por pantalla de la cadena de texto `Entry point`.
- **Línea 11**: llamada a la función `myfunc()` que muestra por pantalla `Inside myfunc`, además de invocar a la función `hi()` del módulo `blabla`.

AMPLIAR CONOCIMIENTOS

- Defining Main Functions in Python
- Python Modules and Packages: An Introduction
- Absolute vs Relative Imports in Python
- Running Python Scripts
- Writing Beautiful Pythonic Code With PEP 8
- Python Imports 101
- Clean Code in Python