

Módulos

En Python, cada uno de nuestros archivos .py se denominan módulos. Estos módulos, a la vez, pueden formar parte de paquetes. Un paquete, es una carpeta que contiene archivos .py.

Para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado `__init__.py`. Este archivo, no necesita contener ninguna instrucción. De hecho, puede estar completamente vacío.

```
└─ paquete
   └─ __init__.py
   └─ modulo1.py
   └─ modulo2.py
   └─ modulo3.py
```

Los paquetes, a la vez, también pueden contener otros sub-paquetes:

```
└─ paquete
   └─ __init__.py
   └─ modulo1.py
   └─ subpaquete
      └─ __init__.py
      └─ modulo1.py
      └─ modulo2.py
```

Y los módulos, no necesariamente, deben pertenecer a un paquete:

```
└─ modulo1.py
└─ paquete
   └─ __init__.py
   └─ modulo1.py
   └─ subpaquete
      └─ __init__.py
      └─ modulo1.py
      └─ modulo2.py
```

Importando módulos enteros

El contenido de cada módulo, podrá ser utilizado a la vez, por otros módulos. Para ello, es necesario importar los módulos que se quieran utilizar. Para importar un módulo, se utiliza la instrucción **import**, seguida del nombre del paquete (si aplica) más el nombre del módulo (sin el .py) que se desee importar.

importar un módulo que no pertenece a un paquete

```
import modulo
```

importar un módulo que está dentro de un paquete

```
import paquete.modulo1
```

```
import paquete.subpaquete.modulo1
```

La instrucción import seguida de nombre_del_paquete.nombre_del_modulo, permite hacer uso de todo el código que el módulo contenga. Python tiene sus propios módulos estándar, que también pueden ser importados.

Namespaces

Para acceder a cualquier elemento del módulo importado, la operación se realiza mediante el **namespace**, seguido de un punto (.) y el nombre del elemento que se desee obtener. En Python, un namespace, es el nombre que se ha indicado luego de la palabra import, es decir la ruta (namespace) del módulo:

Alias

Es posible también, abreviar los namespaces mediante un **alias**. Para ello, durante la importación, se asigna la palabra clave **as**, seguida del alias con el cuál nos referiremos en el futuro a ese namespace importado:

```
import modulo as m

import paquete.modulo1 as pm

import paquete.subpaquete.modulo1 as psm
```

Luego, para acceder a cualquier elemento de los módulos importados, el namespace utilizado será el alias indicado durante la importación:

```
print(m.CONSTANTE_1)

print(pm.CONSTANTE_1)

print(psm.CONSTANTE_1)
```

Importar módulos sin utilizar namespaces

En Python, es posible importar de un módulo solo los elementos que se desee utilizar. Para ello se utiliza la instrucción **from**, seguida del namespace, más la instrucción **import** seguida del elemento que se desee importar:

En este caso, se accederá directamente al elemento, sin recurrir a su namespace:

```
from paquete.modulo1 import CONSTANTE_1
```

Es posible también, importar más de un elemento en la misma instrucción. Para ello, cada elemento irá separado por una coma (,) y un espacio en blanco:

```
from paquete.modulo1 import CONSTANTE_1, CONSTANTE_2
```

¿Qué sucede si los elementos importados desde módulos diferentes tienen los mismos nombres? En estos casos, hay que prevenir fallos, utilizando alias para los elementos:

```
from paquete.modulo1 import CONSTANTE_1 as C1, CONSTANTE_2 as C2

from paquete.subpaquete.modulo1

import CONSTANTE_1 as CS1, CONSTANTE_2 as CS2
```

PEP8 importación:

La importación de módulos debe realizarse al comienzo del documento, en orden alfabético de paquetes y módulos. Primero deben importarse los módulos propios de Python. Luego, los módulos de terceros y finalmente, los módulos propios de la aplicación. Entre cada bloque de imports, debe dejarse una línea en blanco. De forma alternativa (pero muy poco recomendada), también es posible importar todos los elementos de un módulo, sin utilizar su namespace pero tampoco alias. Es decir, que a todos los elementos importados se accederá con su nombre original:

```
from paquete.modulo1 import *

print(CONSTANTE_1)

print(CONSTANTE_2)
```

Definir nuestros propios módulos e importarlos:

Ejemplo: Crear un directorio llamado `modulos`. Dentro construir `mi_primer_modulo.py` con el siguiente código:

```
def suma(a1,a2):
    print("La suma es: ",a1+a2)

def resta(a1,a2):
    print("La resta es: ",a1-a2)

def multiplicacion(a1,a2):
    print("La multiplicacion es: ",a1*a2)

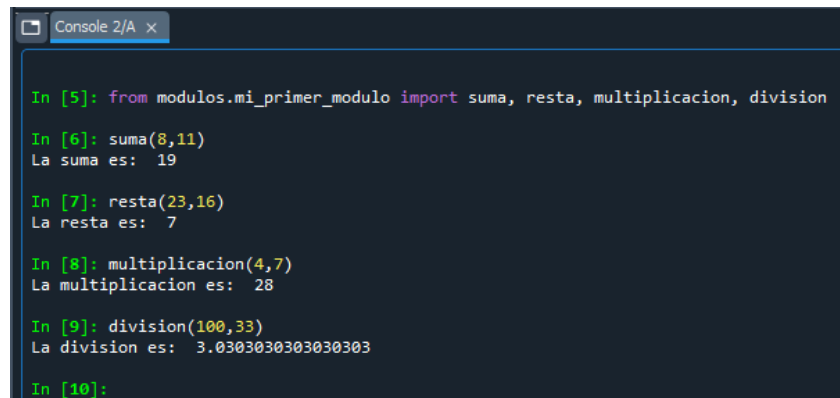
def division(a1,a2):
    print("La division es: ",a1/a2)
```

Uso `mi_primer_modulo.py` desde un programa principal:

```
In [1]: from modulos.mi_primer_modulo import suma, resta, multiplicacion, division
suma(7, 12)
resta(19, 7)
multiplicacion(8, 9)
division(120, 4)
```

La suma es: 19
 La resta es: 12
 La multiplicacion es: 72
 La division es: 30.0

Probar también desde la consola de Spyder (o VSC, Python nativo, etc)



```
Console 2/A x

In [5]: from modulos.mi_primer_modulo import suma, resta, multiplicacion, division

In [6]: suma(8,11)
La suma es: 19

In [7]: resta(23,16)
La resta es: 7

In [8]: multiplicacion(4,7)
La multiplicacion es: 28

In [9]: division(100,33)
La division es: 3.0303030303030303

In [10]:
```

if __name__ == "__main__":

Utilizando `if __name__ == "__main__":` se puede comprobar si el módulo ha sido ejecutado directamente o no, es decir que ha sido importado. Una de las razones para hacerlo es que, a veces, se escribe un módulo (un archivo .py) que se puede ejecutar directamente pero que, alternativamente, también se puede importar y reutilizar sus funciones, clases, métodos, etc en otro módulo.

Con el uso del condicional `if __name__ == "__main__":` conseguimos que la ejecución sea diferente. Todo lo que hay dentro del condicional, cuando el módulo se importa, será completamente ignorado por el intérprete, pero no cuando se ejecute como módulo principal.

Ejemplo: Dentro del directorio modulos, construir `mi_modulo.py` con el siguiente código:

```
print("¡Hola desde mi_modulo.py!")

def hacer_algo():
    print("¡Soy una función!")

if __name__ == "__main__":
    print('Ejecutando como programa principal')
    hacer_algo()

print("¡Adiós desde mi_modulo.py!")
```

En un programa principal ejecutar el siguiente código:

```
In [2]: print("¡Hola!")

def hacer_algo():
    print("¡Soy una función!")

if __name__ == "__main__":
    print('Ejecutando como programa principal')
    hacer_algo()

print("¡Adiós!")

¡Hola!
Ejecutando como programa principal
¡Soy una función!
¡Adiós!
```

En este caso no hay módulo importado, `if __name__ == "__main__":` se ejecutó

En un nuevo archivo principal o en la consola ejecutar:

```
In [3]: ▶ import modulo.mi_modulo
```

```
¡Hola desde mi_modulo.py!  
¡Adiós desde mi_modulo.py!
```

En este caso el condicional no se ejecutó

En el siguiente caso también tenemos un principal que llama a una función del módulo:

```
In [4]: ▶ import modulo.mi_modulo as mimod  
  
print("¡Hola desde principal.py!")  
mimod.hacer_algo()  
print("¡Adiós desde principal.py!")
```

```
¡Hola desde principal.py!  
¡Soy una función de mi_modulo.py!  
¡Adiós desde principal.py!
```

Para el último ejemplo:

- No se cumple la condición `if name == "main"` por lo que solo se ejecuta el código global (el print inicial y se crea el objeto función `hacer_algo` en memoria).
- La función solo se ejecuta cuando la llamamos desde el módulo principal

Ejemplo: Dentro de `modulos`, construir `mi_otro_modulo.py`

```
print("¡Hola desde mi_otro_modulo.py!")  
  
def hacer_algo():  
    print("¡Soy una función en mi_otro_modulo.py!")  
  
if __name__ == "__main__":  
    print('Ejecutando como programa principal')  
  
hacer_algo()  
print("¡Adiós desde mi_otro_modulo.py!")
```

Desde un módulo principal ejecutar:

```
In [5]: ▶ import modulo.mi_otro_modulo as mimod  
  
print("¡Hola desde principal.py!")  
mimod.hacer_algo()  
print("¡Adiós desde principal.py!")
```

```
¡Hola desde mi_otro_modulo.py!  
¡Soy una función en mi_otro_modulo.py!  
¡Adiós desde mi_otro_modulo.py!  
¡Hola desde principal.py!  
¡Soy una función en mi_otro_modulo.py!  
¡Adiós desde principal.py!
```

Para este último ejemplo:

- Al importar ejecutamos todo el código del módulo importado, incluyendo la llamada a la función que ahora no está envuelta en el condicional.
- Una vez importado el módulo principal llama a la función importada de nuevo.

- Cuando el intérprete lee un archivo de código, ejecuta todo el código global que se encuentra en él. Esto implica crear objetos para toda función, clase definida y variables globales.
- Todo módulo (archivo de código) en Python tiene un atributo especial llamado `"__name__"` que define el espacio de nombres en el que se está ejecutando. Es usado para identificar de forma única un módulo en el sistema de importaciones.
- Por su parte `"__main__"` es el nombre del ámbito en el que se ejecuta el código de nivel superior (programa principal).
- El intérprete pasa el valor del atributo `'name'` a la cadena `'main'` si el módulo se está ejecutando como programa principal.
- Si el módulo no es llamado como programa principal, sino que es importado desde otro módulo, el atributo `'name'` pasa a contener el nombre del archivo en sí.

Ejemplo: Dentro del directorio `modulos` construye la siguiente estructura:

```
├─ saludos.py
├─ script.py
└─ test
    └─ script.py
```

Y los siguientes códigos:

saludos.py

```
def saludar():
    print("Hola, te estoy saludando desde la función saludar() del módulo saludos")

class Saludo():
    def __init__(self):
        print("Hola, te estoy saludando desde el __init__ de la clase Saludo")
```

script.py

```
from saludos import Saludo

s = Saludo()
```

test/script.py

```
import sys
sys.path.insert(1, '..')
# print(sys.path)

from saludos import *
s = Saludo()

saludar()
```

sys.path.insert(1, '..'): Python verifica los directorios en orden secuencial comenzando en el primer directorio de la lista *sys.path* hasta que encuentra el archivo .py que busca. El primer elemento de esta lista, es el directorio que contiene el script que se usa para invocar al intérprete de Python. <https://docs.python.org/3/library/sys.html#sys.path> (<https://docs.python.org/3/library/sys.html#sys.path>).

Ejecuta:

```
In [6]:  from saludos import *
```

```
In [7]:  saludar()
```

```
Hola, te estoy saludando desde la función saludar() del módulo saludos
```

```
In [8]:  Saludo()
```

```
Hola, te estoy saludando desde el __init__ de la clase Saludo
```

```
Out[8]: <saludos.Saludo at 0x184ce517d00>
```

Archivos “compilados” de Python

Para acelerar la carga de módulos, Python cachea las versiones compiladas de cada módulo en el directorio **pycache** bajo el nombre "module.version.pyc" dónde la versión codifica el formato del archivo compilado; generalmente contiene el número de versión de Python. Esta convención de nombre permite compilar módulos desde diferentes releases y versiones de Python para coexistir.

Además chequea la fecha de modificación de la fuente contra la versión compilada para ver si esta es obsoleta y necesita ser recompilada. Esto es un proceso completamente automático. Los módulos compilados son independientes de la plataforma, así que la misma librería puede ser compartida a través de sistemas con diferentes arquitecturas.

Paquetes

Un paquete es una colección de módulos de Python, mientras que un módulo es un solo archivo de Python. Un paquete contiene un archivo **__init__.py**, de esta manera se diferencia lo que es un paquete de lo que es un directorio que solo contiene scripts.

Los paquetes se pueden anidar a cualquier profundidad, siempre que los directorios correspondientes contengan su propio archivo **__init__**. Es importante tener en cuenta que todos los paquetes son módulos, pero no todos los módulos son paquetes.

Ejemplo: Crea un directorio llamado packs y dentro construye la siguiente estructura:

```

un_comercio/
├── __init__.py
├── principal.py
├── utilidades/
│   ├── calculos.py
│   ├── impuestos.py
│   └── __init__.py
└── __init__.py

```

En utilidades/calculos.py, escribe el siguiente código:

```

def suma_total(monto=0, suma=0):
    calculo_suma = monto + suma
    return calculo_suma

```

En utilidades/impuestos.py, escribe el siguiente código:

```

def impuesto_iva21(monto=0):
    total = monto * 0.21
    return total

```

En un_comercio/principal.py, escribe el siguiente código:

```

from utilidades import impuestos
from utilidades import calculos

monto = float(input("Ingrese un monto para calcular el iva: "))
print("El IVA es de 21%:", impuestos.impuesto_iva21(monto))

suma = float(input("Ingrese un monto a sumar: "))
print("El IVA es de 21%:", impuestos.impuesto_iva21(suma))

print("Los montos totales suman:{0}".format(calculos.suma_total(monto, suma)))
print("El total de IVA es: {0}".format(impuestos.impuesto_iva21(monto + suma)))

```

En calculo_factura.py, escribe el siguiente código:

```

from un_comercio.utilidades import calculos
from un_comercio.utilidades import impuestos

monto = float(input("Ingrese un monto: "))
monto_suma = float(input("Ingrese un monto a sumar: "))

print("Total a Facturar: {0} pesos".format(calculos.suma_total(monto, monto_suma)))
print("IVA del 21%: {0} pesos".format(impuestos.impuesto_iva21(monto + monto_suma)))

```

Todos los init.py, quedan vacíos.

```

In [9]:  from packs import *
        %run packs/calculo_factura.py

Ingrese un monto: 2000
Ingrese un monto a sumar: 1000
Total a Facturar: 3000.0 pesos
IVA del 21%: 630.0 pesos

```

Los archivos **init.py** se necesitan para hacer que Python trate los directorios como paquetes y se hace para prevenir directorios con un nombre común, que pueden interferir en la búsqueda de módulos. En el caso más simple, **init.py** puede ser solamente un archivo vacío, pero también puede ejecutar código de inicialización para el paquete o configurar la variable **__all__**

Los paquetes son muy útiles, pero si intentamos ejecutarlos desde un directorio distinto a donde se encuentran, no funcionan. Para resolverlo se puede instalar el paquete dentro de Python. Para ello hay que convertir el paquete en un distribuible.

Distribución

Para crear un paquete distribuible hay que crear un archivo setup.py fuera de la raíz. Este archivo incluye la configuración del paquete. Para el ejemplo dado sería:

```

"""Instalador para el paquete "un_comercio."""

from setuptools import setup

long_description = (
    open('README.txt').read()
    + '\n' +
    open('LICENSE').read()
    + '\n')

setup(
    name="mi-paquete",
    version="0.1",
    description="Este es un paquete de ejemplo",
    long_description=long_description,
    # Get more https://pypi.org/pypi?%3Aaction=list_classifiers
    classifiers=[
        # ¿Cuan maduro esta este proyecto? Valores comunes son
        # 3 - Alpha
        # 4 - Beta
        # 5 - Production/Stable
        "Development Status :: 3 - Alpha",
        # Indique a quien va dirigido su proyecto
        "Environment :: Console",
        "Intended Audience :: Developers",
        "Topic :: Software Development :: Libraries",
        # Indique licencia usada (debe coincidir con el "license")
        "License :: OSI Approved :: GNU General Public License (GPL)",
        # Indique versiones soportadas, Python 2, Python 3 o ambos.
        "Programming Language :: Python",
        "Programming Language :: Python :: 3.10",
        "Operating System :: OS Independent",
    ],
    keywords="ejemplo instalador paquete un_comercio",
    author="Nombre Apellido",
    author_email="nombre_apellido@dominio.com",
    url="http://www.un_sitio.net",
    download_url="https://github.com/nombre_apellido/un_comercio",
    license="GPL",
    platforms="Unix",
    packages=['packs', 'packs/un_comercio', 'packs/un_comercio/utilidades' ],
    include_package_data=True,
    scripts=[]
)

```

Los archivos Readme y License hay que crearlos y deben estar en el mismo nivel que setup.py.

- Readme es el archivo donde se define la documentación general del paquete.
- License, es el archivo donde se define los términos de licencia usados en el proyecto. Para seleccionar una licencia, consulte <https://choosealicense.com/> (<https://choosealicense.com/>). Una vez elegida una licencia, abrir el archivo LICENSE e ingresar el texto de la licencia. Por ejemplo, si elige la licencia GPL:

```

License
=====

```

```

PACKAGE-NAME Copyright YEAR, PACKAGE-AUTHOR

```

```

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

```

```

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

```

```

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston,
MA 02111-1307 USA.

```

Luego de construir setup.py ejecutar:

```
!python setup.py sdist --formats=zip
```

Si todo funcionó se habrá creado una nueva carpeta llamada **dist** y otra **.egg-info**. En dist encontraremos un archivo zip o tar.gz (según el S.O. indicarlo en formats). Este archivo es el distribuible y se puede compartir para instalar.

Para instalar, ubicarse en el directorio donde se encuentra el paquete zipeado, ejecutar:

```
pip install mi-paquete-0.1.zip
```

Con el comando

```
pip list
```

se pueden consultar todos los paquetes instalados y ver el propio. Si aparece correctamente puede utilizarse el paquete en cualquier script. Finalmente se utiliza el comando pip uninstall para desinstalarlo:

```
pip uninstall mi-paquete
```

<https://docs.python.org/es/3/tutorial/modules.html#> (<https://docs.python.org/es/3/tutorial/modules.html#>)

<https://docs.python.org/es/3/tutorial/modules.html#packages> (<https://docs.python.org/es/3/tutorial/modules.html#packages>)

<https://docs.python.org/es/3/reference/import.html#packages> (<https://docs.python.org/es/3/reference/import.html#packages>)

<https://docs.python.org/es/3/reference/import.html> (<https://docs.python.org/es/3/reference/import.html>)

<https://docs.python.org/es/3/distutils/sourcedist.html> (<https://docs.python.org/es/3/distutils/sourcedist.html>)