Módulos

Python es un lenguaje que se puede (y muchas veces debe) modularizar, es decir, dividir su código en distintos módulos reutilizables. De hecho, el propio núcleo de Python que instalamos en el sistema ya está modularizado, y podemos incorporar algunos de esos módulos a nuestros programas con la instrucción `import`. También podemos instalar módulos adicionales en el sistema (o en el proyecto) e incorporarlos con esta misma instrucción. Veremos cómo hacer esto en este documento.

1. Importando módulos propios de Python

Como decíamos, la instrucción `import` nos permite incorporar a nuestro programa módulos del núcleo de Python para poderlos utilizar. En el siguiente ejemplo importamos el módulo `sys` para acceder a los parámetros que se pasan al programa principal (`sys.argv`).

```
import sys

for i in range(1, len(sys.argv)):
    # Recorremos los parámetros quitando el 0 (ejecutable)
    print(sys.argv[i])
```

En la instrucción `import`, podemos añadir una partícula `as` para dar un nombre alternativo al módulo a la hora de utilizarlo:

```
import sys as sistema

for i in range(1, len(sistema.argv)):
    # Recorremos los parámetros quitando el 0 (ejecutable)
    print(sistema.argv[i])
```

Alternativamente, también podemos elegir importar sólo una parte (o partes, separadas por comas) del módulo en cuestión. Por ejemplo, de este modo importamos únicamente la constante `pi` y la función `sqrt` del módulo `math`:

```
from math import pi, sqrt
print(pi)
print(sqrt(pi * 3))
```

2. Descomponiendo nuestro código en módulos

A medida que el código de nuestro programa crece, puede ser necesario dividirlo en distintos ficheros fuentes. Al hacer esto, podemos emplear la instrucción `import` para "cargar" o utilizar unos módulos dentro de otros. Por ejemplo, supongamos que incluimos en un archivo llamado *modulo.py* el siguiente contenido:

```
constantePi = 3.141592

def sumar(a, b):
    return a + b
```

Si queremos utilizar estos elementos desde otro fichero, basta con importarlo:

```
import modulo

print(modulo.sumar(3, 4))
print(modulo.constantePi * 8)
```

Podemos utilizar esta otra sintaxis alternativa para importar directamente unos elementos seleccionados:

```
from modulo import constantePi, sumar

print(sumar(3, 4))
print(constantePi * 8)
```

En el caso de trabajar con clases, si tenemos cada clase en un archivo, será conveniente utilizar también la cláusula `from` para indicar que de ese archivo se importe la clase, y así no tener que poner ningún prefijo. Por ejemplo, si tenemos la clase `Persona` en el fichero *Persona.py*:

```
class Persona:
    def __init__(self, ... ):
        ...
```

Y queremos utilizarla desde otro archivo, en lugar de hacer `import Persona` (que nos obligaría después a usar `Persona.Persona` para referirnos a la clase), podemos hacer:

```
from Persona import Persona
```

2.1. Trabajando con varios archivos fuente

Si nuestro proyecto empieza a ser complejo y queremos descomponerlo en distintos archivos fuente, necesitamos un IDE que nos permita trabajar cómodamente con todos estos archivos. Por ejemplo, podemos emplear el IDE **PyCharm**, del que ya hemos hablado aquí. Basta con instalarlo, crear un proyecto e ir colocando nuestro código fuente en distintos archivos dentro de ese proyecto.

> **Ejercicio 1:**
>
> Descompón el ejercicio 3 de esta sesión anterior en módulos, de forma que cada clase esté en un módulo, y mediante instrucciones `import` se incluyan donde se necesiten. Utiliza un IDE avanzado que te permita gestionar cómodamente estos archivos.

3. Instalación de módulos adicionales

Como comentábamos al inicio de este documento, podemos enriquecer nuestras aplicaciones Python añadiendo módulos de terceros en nuestro sistema. Esto puede llevarse a cabo con la herramienta `pip` (*Package Installer for Python*). Esta herramienta viene ya incorporada en las últimas versiones de Python. Podemos utilizarla de este modo:

```
pip install <nombre_modulo>
```

> **NOTA**: en algunas versiones de Linux y Mac el comando se llama `pip3` en lugar de `pip`.

Podemos consultar los paquetes o módulos que tenemos instalados con el siguiente comando:

```
pip list
```

3.1. Ejemplo: Pillow

Pillow es una versión simplificada de una librería llamada PIL (*Python Imaging Library*) que permite manipular imágenes desde Python. Por ejemplo, escalarlas, rotarlas, cambiarles el formato, etc.

3.1.1. Instalación y primeros pasos

Para utilizar Pillow en nuestro sistema, deberemos instalar la librería con un comando como éste:

```
pip install Pillow
```

Después, deberemos incorporar (importar) la librería en los archivos fuente que la necesiten. Normalmente basta con incorporar el módulo `Image`:

```
from PIL import Image
```

3.1.2. Algunas operaciones básicas

Una vez importada la librería, podemos abrir imágenes con el método `open` del elemento `Image`:

```
imagen = Image.open("fichero.png")
```

Podemos generar miniaturas de una imagen con el método `thumbnail` indicando el tamaño final como una tupla:

```
tamano = (64, 64)
miniatura = imagen.thumbnail(tamano)
```

Podemos reescalar la imagen a cualquier tamaño con `resize` (también pasando una tupla con el tamaño deseado):

```
tamano2 = (300, 300)
redimensionada = imagen.resize(tamano2)
```

O rotarla con el método `rotate` (indicando el ángulo de rotación en grados):

```
rotada = imagen.rotate(45)
```

También podemos hacer operaciones de transposición, como voltear la imagen horizontal o verticalmente:

```
volteoHoriz = imagen.transpose(Image.Transpose.FLIP_LEFT_RIGHT)
```

Finalmente, podemos guardar los cambios en otro archivo con el método `save`, indicando el nombre del archivo destino y la extensión o formato de salida (opcional):

```
rotada.save("rotada.png")
volteoHoriz.save("volteada.jpg", "JPEG")
```

Puedes encontrar más información sobre esta librería en su página oficial

> **Ejercicio 2:**
>
> Haz un programa que le pida al usuario un nombre de imagen, la cargue y, si existe, le pida continuamente tamaños a los que la quiera escalar (ancho y alto), y guarde una copia de la imagen original con ese tamaño. El nombre de cada imagen escalada tendrá el patrón *ancho_alto_nombreFicheroOriginal*. El proceso terminará cuando el usuario ponga la anchura o altura a 0.
>
> **NOTA**: para saber si un archivo existe o no puedes usar el módulo `os.path` del núcleo de Python, y su método `isfile`:
>
> ```
> import os.path
>
> if (os.path.isfile(nombreFichero)):
>     ...
> ```
>
> **AYUDA:** vídeo con la solución del ejercicio

3.2. Más operaciones con módulos

Además de instalar módulos, existen otras operaciones básicas que podemos necesitar hacer con ellos.

3.2.1. Actualización de módulos

Podemos actualizar módulos previamente instalados, e incluso la propia herramienta *pip*. Para actualizar el propio *pip*, ejecutamos el siguiente comando:

```
pip install pip -U
```

Para actualizar cualquier módulo instalado, basta con añadir el parámetro `-U`:

```
pip install Pillow -U
```

3.2.2. Consultar versión actual

Podemos consultar la versión actual de *pip* con:

```
pip --version
```

También podemos obtener información sobre uno de los módulos instalados con `pip show`:

```
pip show Pillow
```

3.2.3. Eliminar módulos

Para eliminar un módulo instalado ejecutamos el siguiente comando:

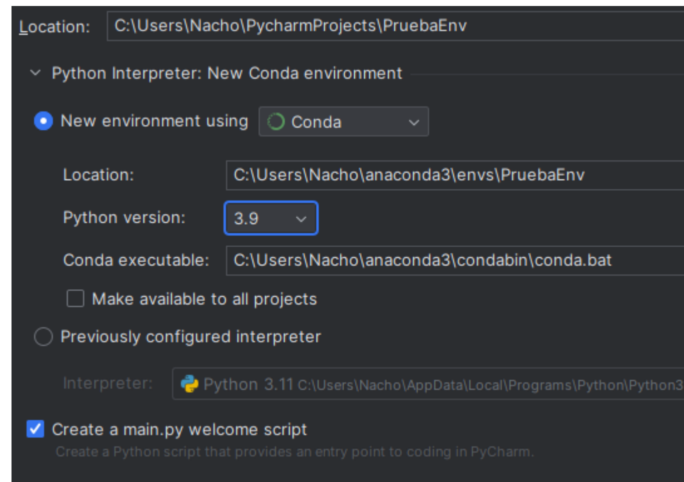
```
pip uninstall nombre_modulo
```

3.3. Gestión de dependencias con entornos virtuales y Conda

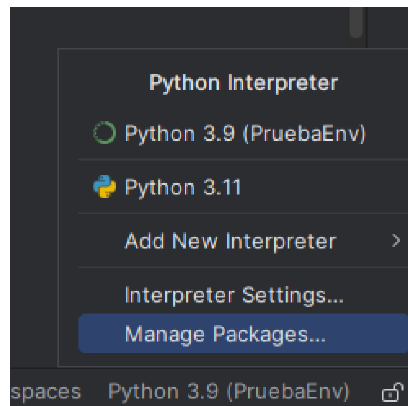
En algunos casos es posible que nos interese instalar ciertas librerías o módulos que no son compatibles con la versión de Python que tenemos actualmente instalada, o que queramos instalar un conjunto de módulos interdependientes entre sí con versiones específicas. Hacer eso en la misma máquina en la que conviven otras instalaciones puede ser algo complicado pero, afortunadamente, podemos crear entornos separados usando la herramienta **Conda**.

Existen varias formas de instalar el gestor *Conda*, aunque una de las más habituales es descargando e instalando la herramienta *Anaconda* que explicamos aquí. Al instalarla, además de todas las herramientas que incorpora, también tendremos disponible el gestor *Conda*.

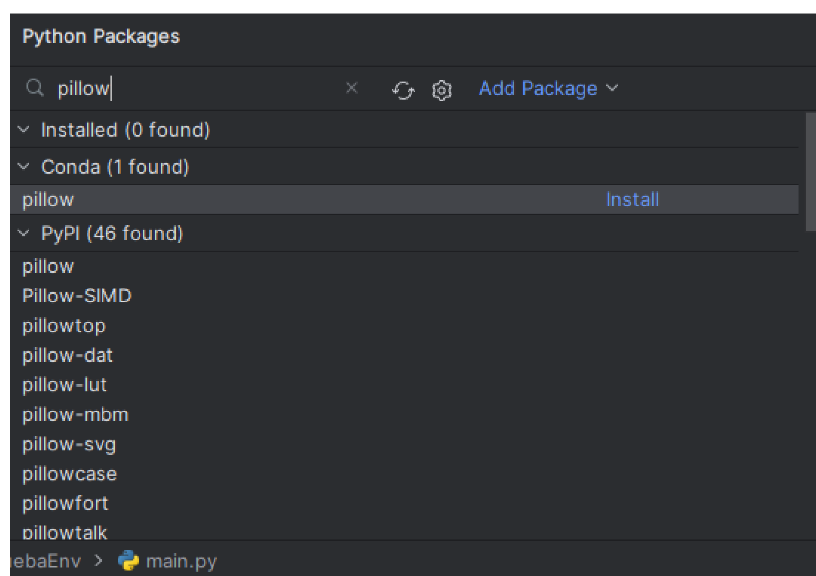
El siguiente paso será crear un **entorno virtual** usando Conda. Podemos emplear el IDE PyCharm para ello. Creamos un nuevo proyecto, y elegimos *New environment*, empleando Conda como gestor del entorno. Notar que también podemos elegir qué versión de Python añadir al entorno (cualquiera, aunque no la tengamos instalada específicamente en nuestro sistema):



Una vez se ha creado el proyecto con el entorno, podremos gestionar los paquetes o módulos que forman parte del entorno desde la esquina inferior derecha de la ventana de PyCharm, eligiendo la opción *Manage Packages*:



Aparecerá un panel donde podremos buscar los paquetes o módulos a instalar, y hacer clic en el enlace *Install*. Podremos elegir también qué versión instalar del módulo en cuestión:



NOTA: es IMPORTANTE recalcar que los módulos que instalemos sólo estarán presentes en el entorno virtual creado para el proyecto específico, y no para otros proyectos (deberemos repetir los mismos

> pasos si queremos tener lo mismo en otros proyectos).