



Curso de Especialización de «Desarrollo de Aplicaciones en Lenguaje Python»

Avanza 2024/25

Módulo «Estructuras de control en Python»

Conceptos previos y recomendaciones

Profesor: Ismael Reyes Rodríguez

Tabla de contenido

1	Introducción.....	3
2	Instalando Python.....	4
3	Herramientas para trabajar	5
3.1	Gestión de paquetes	5
3.2	Jupyter Notebook	6
3.3	Spyder	7
4	Tipos de datos.....	9
4.1	Variables.....	9
4.2	Operadores	10
4.2.1	Operadores aritméticos	10
4.2.2	Operadores relacionales.....	11
4.2.3	Operadores lógicos.....	11
5	Otros conceptos básicos.....	12
5.1	Comentarios.....	12
5.2	Mostrar mensajes en pantalla	12
5.3	Introducir datos desde teclado	13
5.4	Estructuras de control.....	13
6	Prueba de los ejercicios con pypas	14
7	Web para explorar	17

1 Introducción

Este documento recoge algunos **conceptos previos** que se deben saber para afrontar este módulo, así como algunas **recomendaciones** sobre instalación de herramientas, entornos y otros aspectos básicos que se deben tener en cuenta.

La mayoría de las indicaciones que se dan en este documento están basadas en entornos Windows, aunque si se tiene soltura con otros Sistemas Operativos, como Ubuntu, no se debería, encontrar problemas para seguir las instrucciones dadas.

Las indicaciones que se realizan sobre características del propio lenguaje Python se irán desarrollando en profundidad a medida que se desarrolle **las 5 Unidades de Trabajo** en las que se divide este módulo.

2 Instalando Python

Python es un lenguaje de programación interpretado, de alto nivel y propósito general. Es un proyecto libre y de código abierto, con una gran comunidad implicada en el desarrollo y mantenimiento de librerías.

Para el desarrollo y seguimiento de este módulo será necesario tener instalado Python en nuestra máquina. Es recomendable instalar la última versión disponible, aunque no indispensable ya que basta con tener instalada la **versión 3.0 o posterior**.

Para el correcto seguimiento del módulo será muy recomendable instalar la distribución **Anaconda**¹ de Python, la cual contiene el intérprete y las librerías del core, que nos permitirán desarrollar sin problemas los ejercicios y, además, incluye la mayoría de librerías utilizadas para el desarrollo de aplicaciones.

Puedes descargar la versión de Anaconda desde el siguiente enlace:

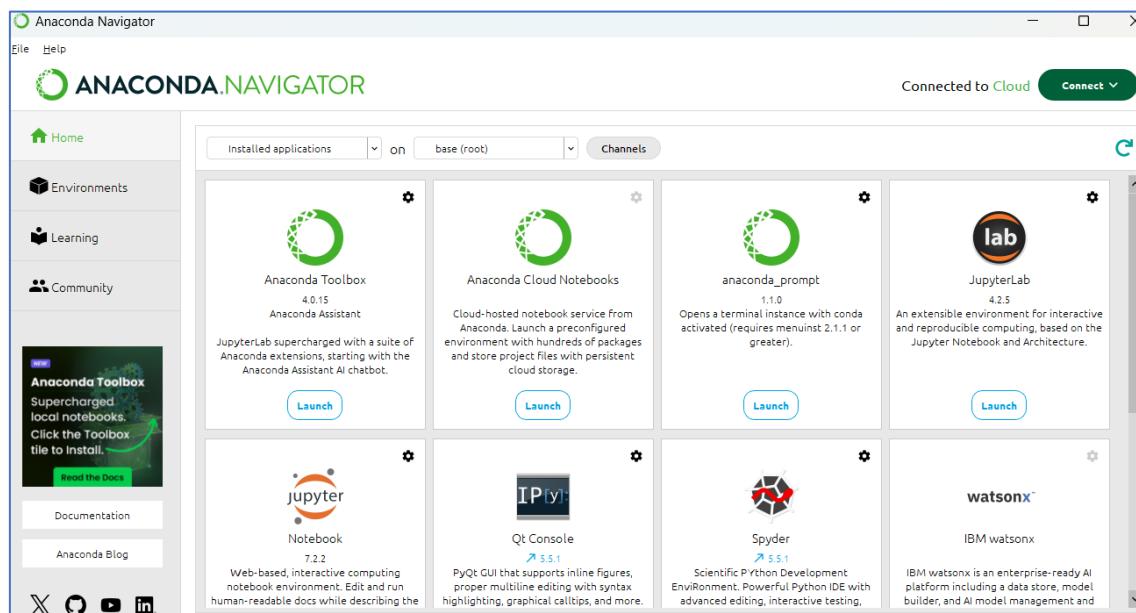
<https://www.anaconda.com/products/individual>

Es más que posible que la última distribución de Anaconda no instale la última versión de Python pero, como ya he comentado, no será necesaria.

La instalación es sencilla, pero ante cualquier duda, recomiendo consultar la siguiente página:

[Installation — Anaconda documentation](#)

Una vez instalado Anaconda, su navegador nos permitirá acceder a prácticamente todas las herramientas que usaremos a lo largo de este curso:



Ejemplo de herramientas instaladas

¹ No es obligatorio trabajar con Anaconda, aunque a lo largo del curso es posible se haga referencia a algún aspecto de dicho entorno.

3 Herramientas para trabajar

Un programa en lenguaje Python no deja de ser un texto normal almacenado en un fichero, por lo que podemos editar y trabajar con este fichero desde cualquier editor de texto plano: Notepad, SublimeText, nano,

A medida que los programas que generamos en Python son más complejos, editar uno o más ficheros de texto con este tipo de programas no resulta útil por lo que se utilizan herramientas especializadas, llamadas **IDE** (Integrated Development Environment), que nos ofrecen diferentes funcionalidades que van desde el coloreado de palabras reservadas a la generación de código con Inteligencia Artificial, pasando por la depuración de programas.

Existen diferentes IDE, como [Thonny](#) o Visual Studio Code (descargables desde Microsoft Store) y diferentes herramientas con las que nos puede resultar útil trabajar. Para intentar mantener un poco estandarizada la forma en la que se trabajará en este curso, los siguientes subapartados muestran una serie de **recomendaciones**² y aplicaciones con las que podemos comenzar a programar en este módulo.

3.1 Gestión de paquetes

La instalación limpia de Python ya ofrece de por sí muchos paquetes y módulos que vienen por defecto, es lo que se llama la [librería estándar](#), aunque no debemos olvidar que una de las características más destacables de Python es su inmenso «ecosistema» de paquetes disponibles en el [Python Package Index \(PyPI\)](#).

Para gestionar los paquetes que tenemos en nuestro sistema se utiliza la herramienta [pip](#), una utilidad que también se incluye en la instalación de Python. Con ella podremos instalar, desinstalar y actualizar paquetes, según nuestras necesidades.

En nuestro caso, podremos también utilizar Anaconda para gestionar paquetes. Para ello, debemos entrar en el directorio (carpeta) de instalación de [Anaconda](#), entrar en el directorio [condabin](#) (en mi caso este directorio es `C:\Users\elmae\anaconda3-2024\condabin`) y ejecutar el programa `conda.bat` con algunos parámetros.

Los siguientes comandos no se tienen que ejecutar pero nos dan una idea de cómo:

- Instalar un paquete

```
C:\Users\elmae\anaconda3-2024\condabin>conda install pandas
```

El paquete *pandas* ya viene instalado por defecto en Anaconda por lo que el resultado de la instalación será una respuesta del tipo: *# All requested packages already installed.*

² Solo son recomendaciones. Si os sentís más a gusto utilizando otras herramientas y podéis afrontar las tareas del módulo sin problema, sentíos libres de utilizar las herramientas que consideréis oportunas.

- Desinstalar un paquete

```
conda uninstall pandas
```

- Actualizar un paquete

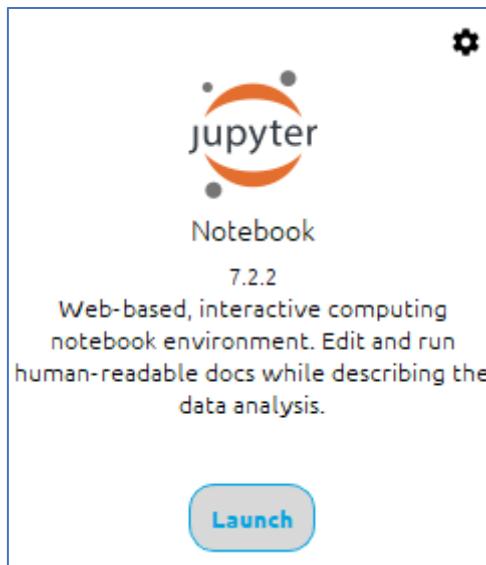
```
conda upgrade pandas
```

3.2 Jupyter Notebook

Jupyter Notebook es una aplicación «open-source» que permite crear y compartir documentos que contienen código, ecuaciones, visualizaciones y texto narrativo. Podemos utilizarlo para propósito general, aunque suele estar más enfocado a *ciencia de datos*: limpieza y transformación de datos, simulación numérica, modelado estadístico, visualización o «machine-learning».

Podemos verlo como un intérprete de Python (contiene un «kernel» que permite ejecutar código) con la capacidad de incluir documentación en formato [Markdown](#), lo que potencia sus funcionalidades y lo hace adecuado para preparar cualquier tipo de material vinculado con lenguajes de programación.

Esta herramienta se instala con la instalación de Anaconda por lo que podemos acceder a ella pulsando el botón Launch de nuestro **Anaconda Navigator**:



Acceso a Jupyter Notebook

Al acceder a la herramienta comprobamos que Jupyter Notebook es una aplicación web local (puerto 8888, en mi caso), en la que como desarrolladores podemos dividir el código en partes y trabajar en ellas sin importar el orden: escribir, probar funciones, cargar un archivo en la memoria y procesar el contenido,

Para probar inicialmente el funcionamiento de esta herramienta, podemos escribir el siguiente código:

```
print('Primeras pruebas del módulo "Estructuras de control"')
```

Ahora, al pulsar sobre el botón *Run* (▶) veremos el resultado de interpretar el código:

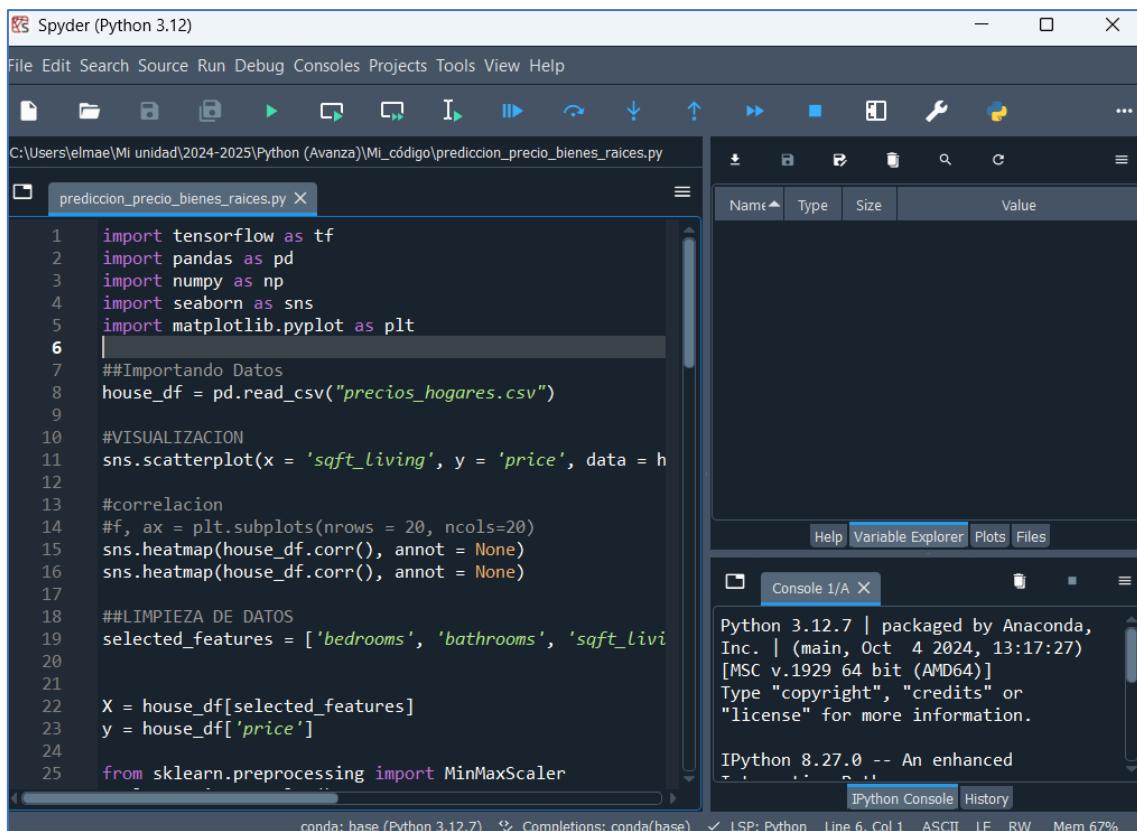


The screenshot shows a Jupyter Notebook interface with a single cell containing the code: `print('Primeras pruebas del módulo "Estructuras de control"')`. The output of the cell is the string `'Primeras pruebas del módulo "Estructuras de control"`. The cell is labeled [3]:

Resultado de ejecutar una línea de Python

3.3 Spyder

El IDE [Spyder](#) se instala automáticamente con Anaconda, por lo que podemos acceder a esta herramienta mediante nuestro Anaconda Navigator.



The screenshot shows the Spyder IDE interface. On the left, a code editor displays a Python script named `prediccion_precio_bienes_raices.py` with the following content:

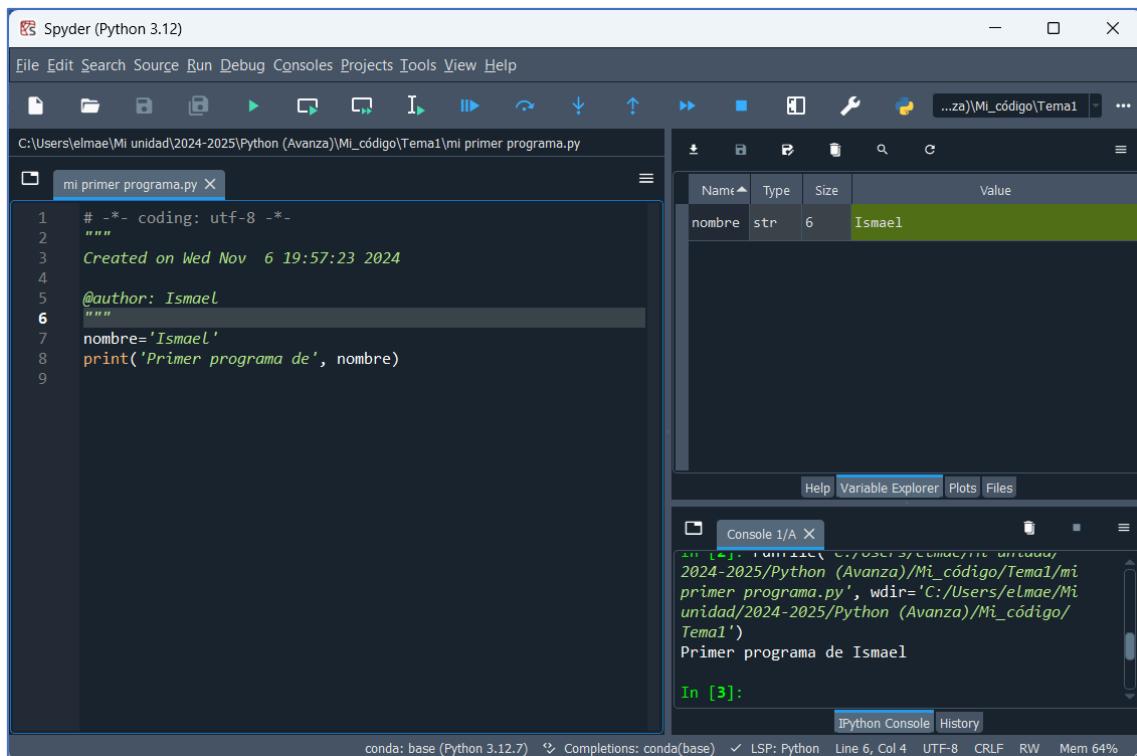
```
1 import tensorflow as tf
2 import pandas as pd
3 import numpy as np
4 import seaborn as sns
5 import matplotlib.pyplot as plt
6
7 ##Importando Datos
8 house_df = pd.read_csv("precios_hogares.csv")
9
10 #VISUALIZACION
11 sns.scatterplot(x = 'sqft_living', y = 'price', data = h
12
13 #correlacion
14 #f, ax = plt.subplots(nrows = 20, ncols=20)
15 sns.heatmap(house_df.corr(), annot = None)
16 sns.heatmap(house_df.corr(), annot = None)
17
18 ##LIMPIEZA DE DATOS
19 selected_features = ['bedrooms', 'bathrooms', 'sqft_living']
20
21 X = house_df[selected_features]
22 y = house_df['price']
23
24 from sklearn.preprocessing import MinMaxScaler
```

On the right, there are several panes: a Variable Explorer showing an empty table, a Plots pane, a Console 1/A pane displaying the Python and IPython environment details, and a Files pane.

Programa Python visualizado en Spyder

Esta herramienta nos permite crear programas en Python, depurarlos y ejecutarlos de una forma sencilla. Para profundizar en su uso, podemos consultar alguno de los tutoriales disponible en internet, por ejemplo, en la propia página de [Spyder](#) (en inglés).

A modo de resumen, en la parte superior vemos los diferentes menús y los iconos para realizar ejecutar acciones, en la parte izquierda del IDE vemos el programa con el que estamos trabajando (pueden ser varios) y, a la derecha, vemos arriba información sobre las variables del programa y abajo, el resultado de la ejecución del mismo.



Ejemplo de programa ejecutado en Spyder

Es muy recomendable, almacenar ordenadamente todas las prácticas y programas con los que estemos trabajando de modo que podamos encontrarlos de forma inmediata. Por ejemplo, seria bueno tener todos los programas con los que trabajemos en el tema 1 de este módulo en una carpeta del tipo “*\Escritorio\Estructuras de control\Tema1*”.

4 Tipos de datos

En este mismo curso de especialización existe un módulo propio para los tipos de datos de Python, pero como todos los módulos se pueden cursar a la vez, es conveniente hacer un resumen de los conocimientos básicos que debemos tener para comenzar a trabajar con estructuras de control sin demasiados problemas.

Los **programas** están formados por **código y datos**, pero a nivel interno de la memoria del ordenador no son más que una secuencia de bits. La interpretación de estos bits depende del lenguaje de programación.

Cada «trozo» de memoria contiene realmente un objeto, de ahí que se diga que en Python **todo son objetos**, y cada objeto tiene, al menos, la siguiente información:

- Un **tipo** del dato almacenado.
- Un **identificador** único para distinguirlo de otros objetos.
- Un **valor** consistente con su tipo.

Los tipos de datos básicos en Python se muestran en la siguiente tabla:

Nombre	Tipo	Ejemplos
Booleano	bool	True, False
Entero	int	62, 13573, 13_573
Flotante	float	3.1416, 1.8e4
Complejo	complex	5 + 66j, 7j
Cadena	str	'hola', "Rodolfo"
Tupla	tuple	(3, 4, 8)
Lista	list	['manzanas', 'peras']
Conjunto	set	set([3, 4, 8])
Diccionario	dict	{'Nombre': 'Juan', 'Edad': '23'}

Además de conocer los tipos de datos, debemos saber como se trabaja con los propios datos en un programa, cosa que se resume brevemente en los siguientes subapartados.

4.1 Variables

Las variables son fundamentales ya que permiten definir nombres para los valores que tenemos en memoria y que vamos a usar en nuestro programa.

Para **asignar un valor** a una variable utilizamos sentencias como las que hemos visto con anterioridad en Spyder:

```
autor="Quevedo"
```

Donde *autor*, es el nombre de la variable e “*Quevedo*” es el valor que le asignamos.

En Python existen una serie de reglas para los nombres de variables:

1. Sólo pueden contener los siguientes caracteres:
 - Letras minúsculas.
 - Letras mayúsculas.
 - Dígitos.
 - Guiones bajos (_).
2. Deben empezar con una letra o un guion bajo, nunca con un dígito.
3. No pueden ser una palabra reservada del lenguaje («keywords»).

4.2 Operadores

Los operadores, como su nombre indica, nos permiten realizar operaciones con datos. Dependiendo del tipo de datos podremos realizar unas operaciones u otras.

4.2.1 Operadores aritméticos

La siguiente tabla muestra los operadores aritméticos existentes:

Operación	Operador
Suma	+
Resta	-
Multiplicación	*
División	/
División entera	//
Módulo	%
Exponenciación	**

El resultado de ejecutar la siguiente línea:

```
print('2 elevado a 4 =', 2**4)
```

Sería:

```
2 elevado a 4 = 16
```

4.2.2 Operadores relacionales

Los operadores relacionales (o de comparación) nos permite efectuar comparaciones entre objetos de Python. El resultado de una comparación es un valor booleano **True** o **False**.

La siguiente tabla muestra los operadores relacionales existentes:

Comparación	Operador	Ejemplo	Resultado del ejemplo
Igual que	<code>==</code>	<code>3 == 3</code>	<code>True</code>
Diferente a	<code>!=</code>	<code>4 != 4</code>	<code>False</code>
Mayor que	<code>></code>	<code>5 > 1</code>	<code>True</code>
Mayor o igual que	<code>>=</code>	<code>3 >= 3</code>	<code>True</code>
Menor que	<code><</code>	<code>5 < 1</code>	<code>False</code>
Menor o igual que	<code><=</code>	<code>5 <= 10</code>	<code>True</code>

4.2.3 Operadores lógicos

Los operadores lógicos sirven para realizar operaciones de lógica booleana entre valores de tipo **bool**. En Python podemos utilizar los operadores lógicos **and**, **or** y **not**.

La siguiente tabla muestra la tabla de verdad para los valores lógicos:

X	Y	X and Y	X or Y	Not X
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

5 Otros conceptos básicos

Veamos cómo realizar algunas tareas comunes y básicas que resultan necesarias al crear un programa en Python.

5.1 Comentarios

Añadir comentario en lenguaje natural en nuestro programa siempre es necesario para hacer indicaciones y aclaraciones al posible lector del mismo, que podríamos ser nosotros mismos o compañero que esté colaborando con nuestro trabajo.

El lenguaje Python usa el símbolo almohadilla (#) para indicar al interprete que, el texto que le sigue es solo un comentario que debe ignorar. Un posible ejemplo sería:

```
# Ejemplo de comentario
```

5.2 Mostrar mensajes en pantalla

Mostrar mensajes por pantalla es necesario para que el usuario comprenda qué está haciendo el programa. Para esta necesidad se utiliza la función **print()** que ya hemos visto en algunos ejemplos de este documento.

Existen distintas formas de pasar argumentos (parámetros) a la función, aunque en estos momentos nos es suficiente con saber las básicas.

Imprimir una cadena literal

```
print("hola mundo")  
o  
print('hola mundo')
```

Imprimir el contenido de una variable

```
print(una_variable)
```

Imprimir varias variables y cadenas, separadas por comas

```
print("Mi edad es:", var_edad, ". Pi vale:", var_pi)
```

5.3 Introducir datos desde teclado

En ocasiones es necesario introducir datos de entrada a nuestro programa y una forma habitual de hacerlo es mediante teclado. Para esta necesidad se utiliza la función **input()**.

La función **input()** retorna la cadena de caracteres introducida, por lo que hay que guardarla en una variable. Veamos algunos ejemplos de su uso.

Función input para introducir datos.

```
entrada = input()

# Muestro la cadena escrita
print(entrada)
```

Función input() admite un parámetro

```
# La función input() admite un parámetro de tipo cadena para
# informar al usuario
entrada = input("Introduce un número: ")
```

Conversión a entero

```
# Si queremos introducir números hay que convertir la entrada a un
# tipo numérico mediante un casting (cambio de tipo)
entrada = int( input("introduce un número: ") )

# La función type() muestra el tipo de la variable
print(type(entrada))
```

5.4 Estructuras de control

Las estructuras de control las veremos detenidamente a lo largo de todo el módulo, ya que son la esencia del mismo.

Por adelantar algo, solo indicar que trabajaremos en profundidad el uso y las características de la **sentencia condicional «if»** y de las **sentencias iterativas «while» y «for»**.

6 Prueba de los ejercicios con pypas

Para la realización y entrega de mucho de los ejercicios de este módulo es **indispensable** trabajar con el paquete [pypas](#).

Para ello, debemos instalarlo en el sistema utilizando **pip**, que es un ejecutable que se encuentra en el **directorio Scripts**, dentro del directorio donde hemos instalado Anaconda, mediante el comando:

```
pip install pypas-cli
```

Una vez instalado, se genera el **ejecutable pypas** en el mismo directorio Script y ya podremos validar muchos de los ejercicios que realicemos mediante los casos de prueba contenidos en esta herramienta.

La herramienta **pypas** servirá para validar la correcta implementación de algunos ejercicios que deberéis entregar como tareas del módulo.

Como os he comentado anteriormente, hay muchas formas de trabajar y de configurar los diferentes entornos, yo simplemente os realizo algunas recomendaciones, de modo que todos trabajemos de forma similar y, en caso de encontrar algún problema en un punto, os pueda dar respuesta a todos a la vez.

Es por este motivo que os realizo las siguientes indicaciones de cómo recomiendo que se trabaje «pypas».

Pasos previos

1. Nos vamos a la carpeta de trabajo de, por ejemplo, el Tema1, bien con la línea de comandos o bien con el Explorador de archivos. Un ejemplo, si tenemos la carpeta de trabajo directamente en el escritorio sería algo así como:

```
cd C:\Users\elmae\Desktop\Mi_código\Tema1
```
2. En este directorio, creamos un fichero por lotes, llamado **pypas.bat** que se encargará de llamar a nuestro programa sin necesidad de modificar el PATH con un contenido similar al siguiente:

```
C:\Users\elmae\anaconda3-2024\Scripts\pypas %1 %2 %33
```

Descarga de un ejercicio

Cuando se proponga la realización de un programa y su validación con pypas, deberemos descargárnoslos mediante la ejecución de **pypas get <ejercicio>**, en nuestra carpeta de trabajo. Ejemplo:

```
C:\Users\elmae\Desktop\Mi_código\Tema1>pypas get add
```

³ La dirección de la carpeta “Scripts” variará con cada usuario e instalación. Los parámetros %1 %2 %3 deben indicarse para poder pasarle los parámetros necesarios a nuestro programa

Esto nos creará una carpeta, en este caso “add”, con diferente contenido entre los que destaca:

3. Fichero “**\doc\README.pdf**” o similar, que contendrá la definición del problema a solucionar con Python.
4. Fichero **main.py**. Es el que tendremos que editar con **Spyder**⁴ para realizar el ejercicio. Dentro de este fichero, encontraremos el texto “**# TODO**” que nos indica el lugar de este programa que deberemos modificar.

En el ejemplo del programa “add”, el problema se define como:

Mi primera suma

Dados dos valores numéricos enteros a y b calcula la suma de los mismos utilizando el lenguaje de programación Python.

Y la resolución del ejercicio, contenido de **main.py**, sería:

```
def run(a: int, b: int) -> int:  
    result = a + b  
    return result  
  
# DO NOT TOUCH THE CODE BELOW  
if __name__ == '__main__':  
    import vendor  
  
    vendor.launch(run)
```

*En **negrita** los cambios realizados.*

Probar el programa

Una vez modificado el fichero *main.py* debemos comprobar que el programa realizado hace lo que se nos pedía y, para ello, utilizaremos la instrucción “**pypas test**”. En mi directorio de trabajo para comprobar el funcionamiento del programa *add*, la ejecución sería:

```
C:\Users\elmae\Desktop\Mi código\Tema1\add>..\pypas test5
```

⁴ Si no queréis utilizar Spyder utilizad otro IDE con el que os veáis más sueltos

⁵ Como tenemos el fichero *pypas.bat* en la carpeta anterior a la de la carpeta “*add*” en la que estamos, debemos indicarlo en la instrucción mediante “*..*” para que lo encuentre.

El resultado de la prueba de un programa aparecerá en pantalla y se mostrará en **verde** si todo ha ido bien. En el ejemplo:

```
===== test session starts
=====
platform win32 -- Python 3.12.7, pytest-7.4.4, pluggy-1.0.0
rootdir: C:\Users\elmae\Desktop\Mi_código\Tema1\add
plugins: anyio-4.2.0, dependency-0.6.0, typeguard-4.3.0
collected 4 items

tests\test_main.py ...
[100%]

=====
4 passed in 0.04s
=====
```

Nota importante: En muchas de las tareas que se realicen de programación, se pedirá que se descargue un **ejercicio con pypas**, se entregue el **código** y, además, el resultado de las **pruebas** validadas con *pypas test*.

Si queremos probar el resultado del programa con algunos argumentos concretos debemos crear un fichero **args.py** que contenga el valor deseado para los argumentos y posteriormente, ejecutar **main.py** en Spyder. Un ejemplo del contenido del fichero **args.py** para nuestro programa *add* podría ser:

```
x = 3
y = 7
```

El programa *add* tomaría estos argumentos, pero no tiene por qué mostrar el resultado de la suma en consola.

7 Web para explorar

En Internet podéis encontrar multitud de vídeos, tutoriales y manuales para profundizar en los temas que se trabajan en este módulo y, podíamos empezar por las propias Web de las herramientas que tenéis instaladas.

El siguiente listado es solo un ejemplo de los múltiples recursos existentes.

[El tutorial de Python — documentación de Python - 3.13.0](#)

[Aprende a Programar en Python Desde Cero — Curso Completo Gratis de 4.5+ Horas](#)

[Aprende Python](#)

[Tutorial de Python completo en español - Aprender Gratis](#)

[pypas | Aprende Python comiendo pipas](#)

[El Manual de Python](#)

[Curso de PYTHON desde CERO para PRINCIPIANTES 😊 Tutorial en Español 🎓 N° 000](#)