vuela

andaluciavuela.es

CONECTA CON ANDALUCÍA

EXPERTO EN PYTHON

PLATAFORMA DIGITAL DE ANDALUCÍA

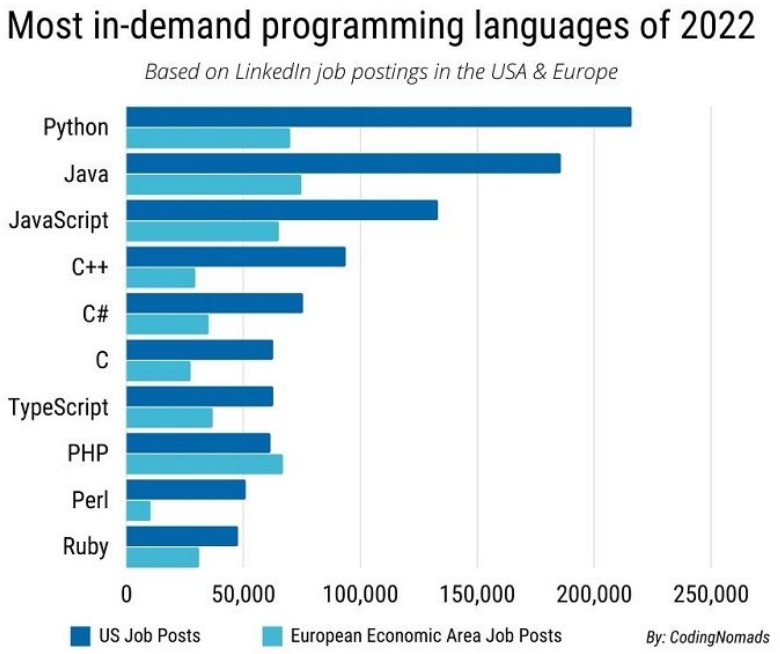
Junta de Andalucía

ÍNDICE DE CONTENIDO

1. Python y entornos de desarrollo

Python es un lenguaje de programación interpretado; es decir, los programas escritos en Python se ejecutan directamente, instrucción a instrucción, por un **gestor** o **intérprete de programa.** Aunque los lenguajes interpretados tradicionalmente suelen ser más lentos que los compilados para una arquitectura de computadores específica, en los últimos tiempos han tomado gran relevancia a nivel de desarrollo, principalmente por las facilidades para ser incluidos en distintas plataformas (Windows, Linux, MAC, sistemas empotrados, etc.).

En particular, Python es uno de los lenguajes más demandados por la industria del software y de las Ciencias de Datos, como se puede ver en la siguiente figura:

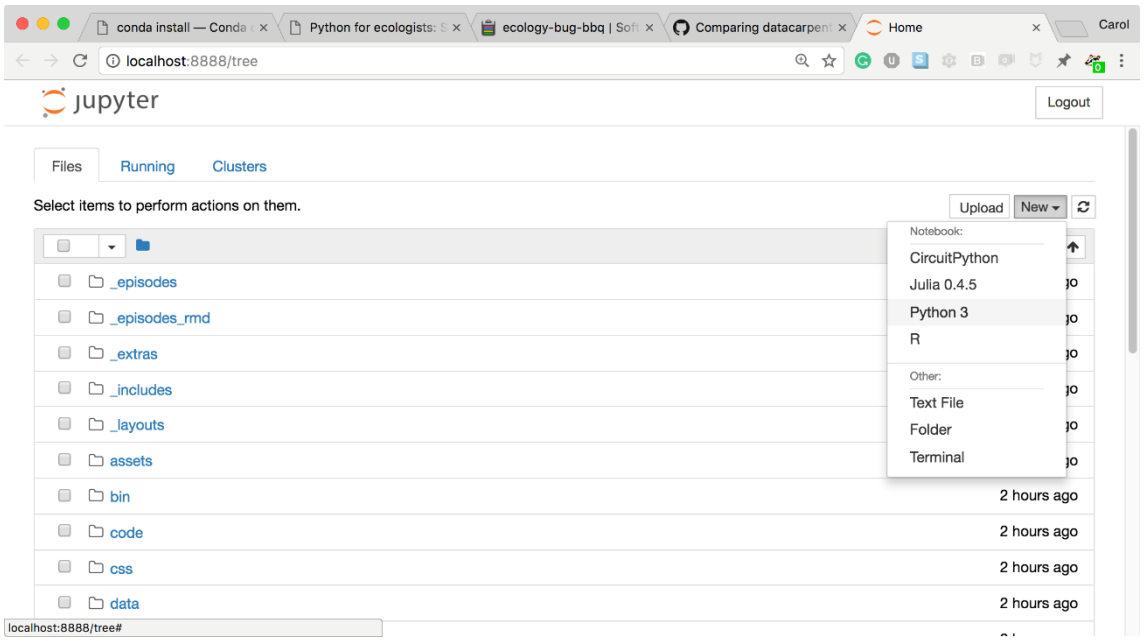


Fuente de la figura: https://ubiqum.com/es/blog/curso-de-python/

Una de las principales ventajas de Python es que posee una gran cantidad de entornos de desarrollo (IDEs), muchos de ellos de libre acceso. Entre ellos encontramos (sólo algunos ejemplos):

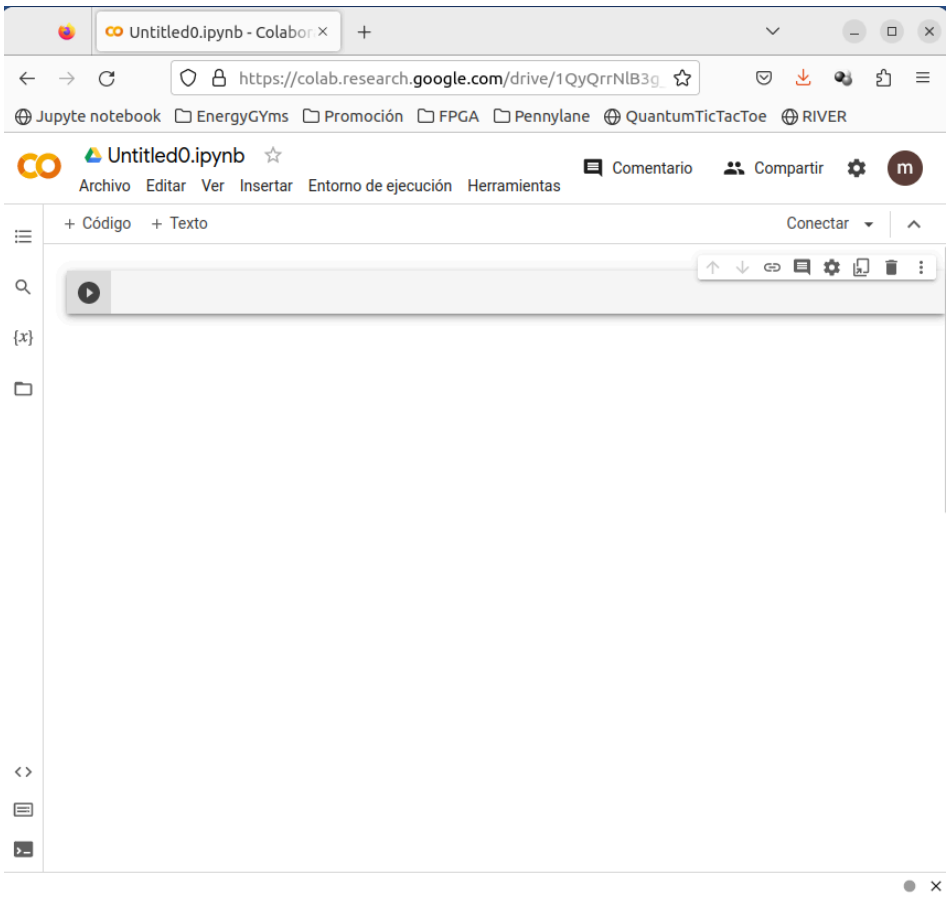
Jupyter Notebook

Es un entorno de desarrollo versátil e interactivo, que organiza el código en *notebooks* o conjuntos de celdas que pueden ejecutarse secuencialmente o por separado. La ejecución es a través del navegador web, por lo que puede ejecutarse en cualquier tipo de plataforma que cuente con este tipo de software. No obstante, su ejecución no requiere conexión a Internet.



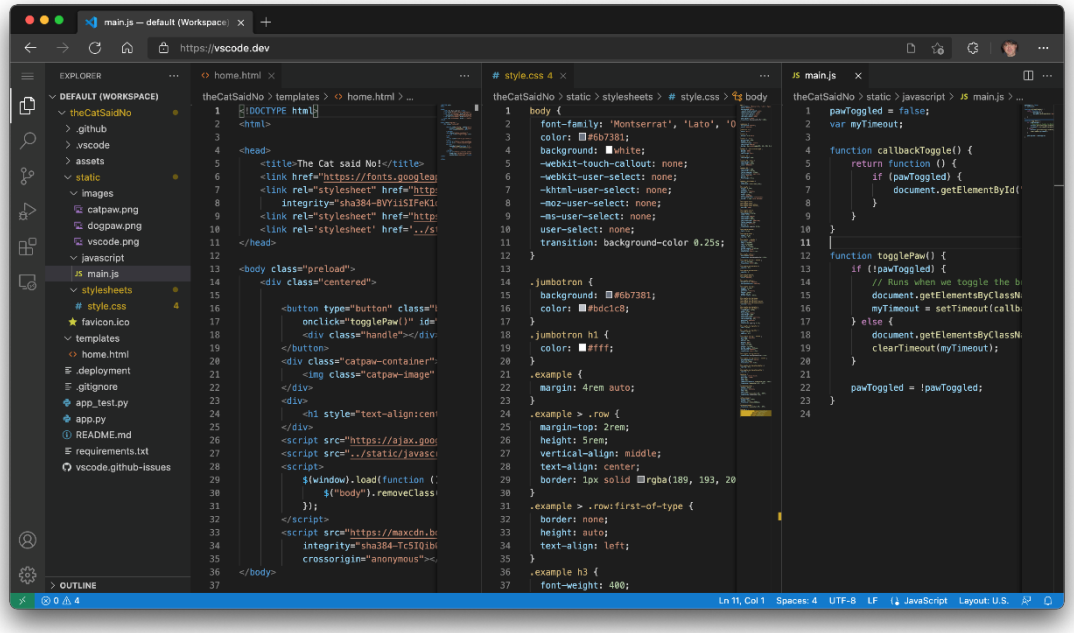
Google Colab

Es un entorno cloud, propiedad de Google, que se ejecuta en un navegador a través de Internet. De características muy similares a Jupyter Notebook, posee adicionalmente la comodidad de ubicuidad, dado que su ejecución se realiza en Cloud a través de cualquier plataforma que disponga de un navegador web. No requiere tener instalado Python ni cualquier otro tipo de software en el PC.



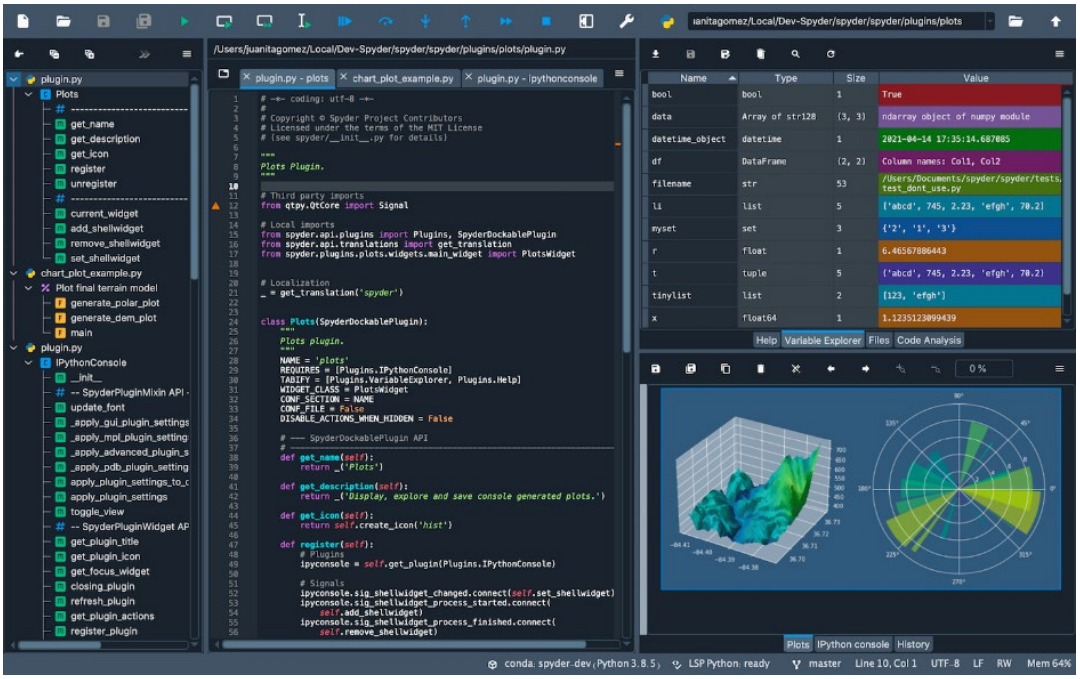
VSCode

Es un entorno simple de programación para cualquier tipo de lenguajes de programación. El IDE VSCode es muy cómodo de usar, sobre todo si se programa en varios lenguajes de programación de forma simultánea, además de tener integradas posibilidades de virtualización y ejecución en distintas plataformas.



Spyder

Es otro IDE de programación en Python, con capacidad para facilitar tareas de análisis de datos y mostrar gráficas. Tiene una alta integración con diferentes sistemas de virtualización como **Anaconda**, por lo que será la opción a usar en este curso.

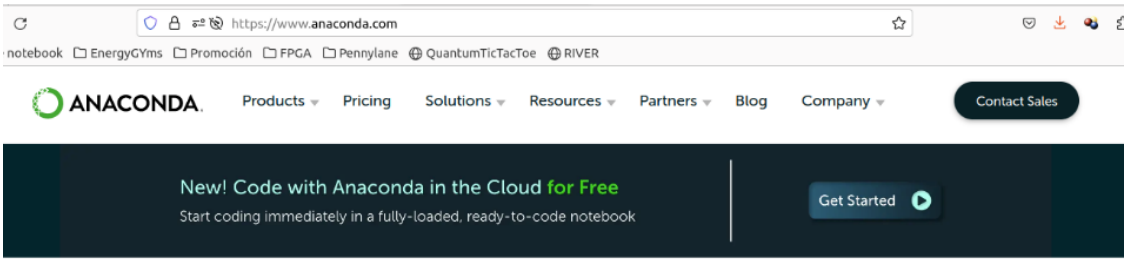


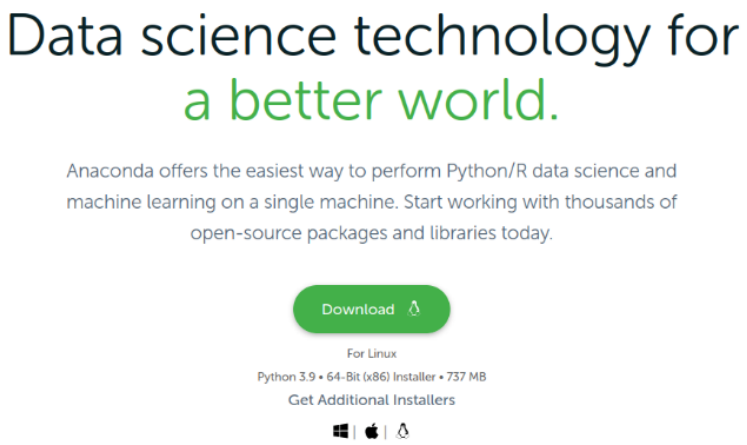
Python también posee amplias posibilidades de generación de entornos virtuales y virtualización para ejecutar programas con los requerimientos necesarios en cada caso (por ejemplo: Docker, Anaconda…). Entre ellos, en el curso trabajaremos con el gestor de entornos Python **Anaconda**, por su simplicidad de uso.

1.1.

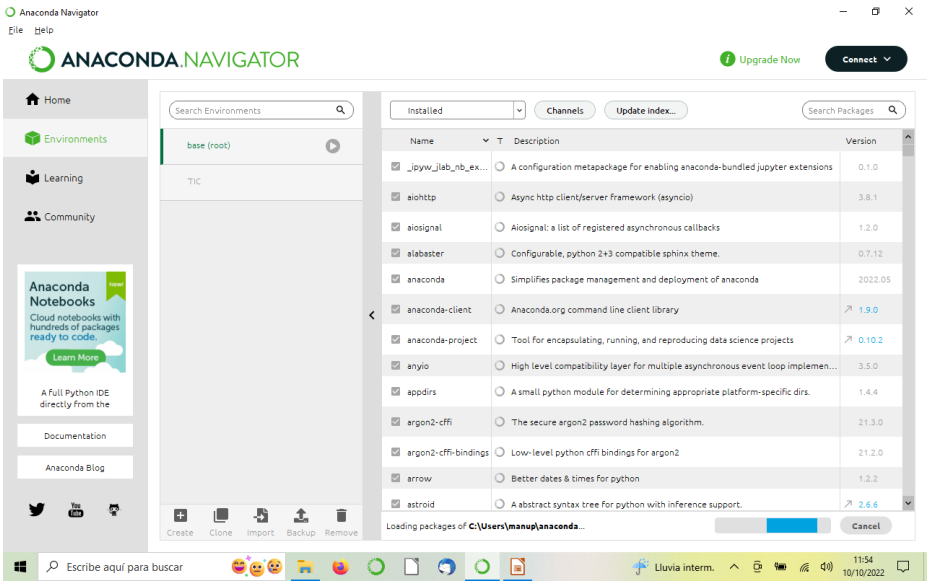
1.2. Instalación de Anaconda

Anaconda es uno de los entornos de virtualización enfocados para lenguajes específicos como Python o R. Permite crear y ejecutar diferentes entornos con bibliotecas y versiones de Python diferentes, facilitando de este modo el desarrollo de aplicaciones para diferentes tipos de proyectos. La instalación de Anaconda se debe realizar introduciendo en el navegador la URL https://www.anaconda.com , descargando el instalador y siguiendo las instrucciones de instalación.

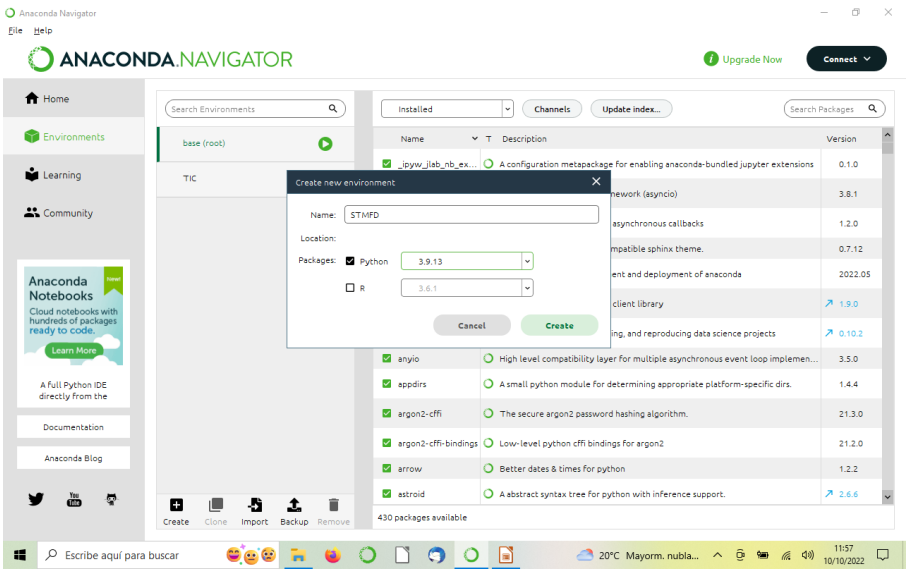




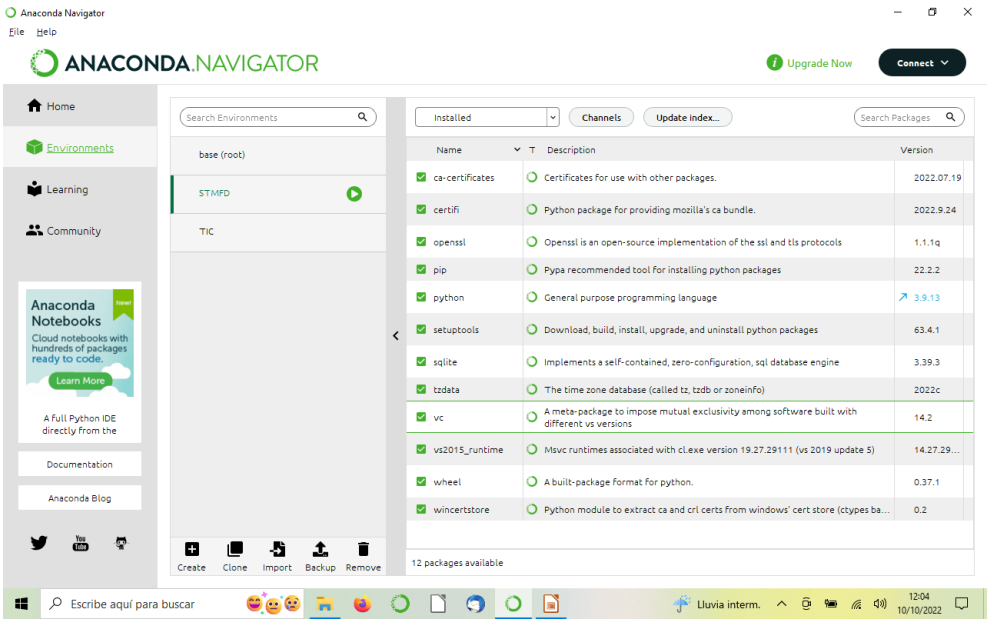
Una vez instalado, al ejecutar **Anaconda** vemos diferentes opciones y, entre ellas, nos fijamos en **environments**, donde se nos permitirá crear un entorno para ejecutar los programas del curso:



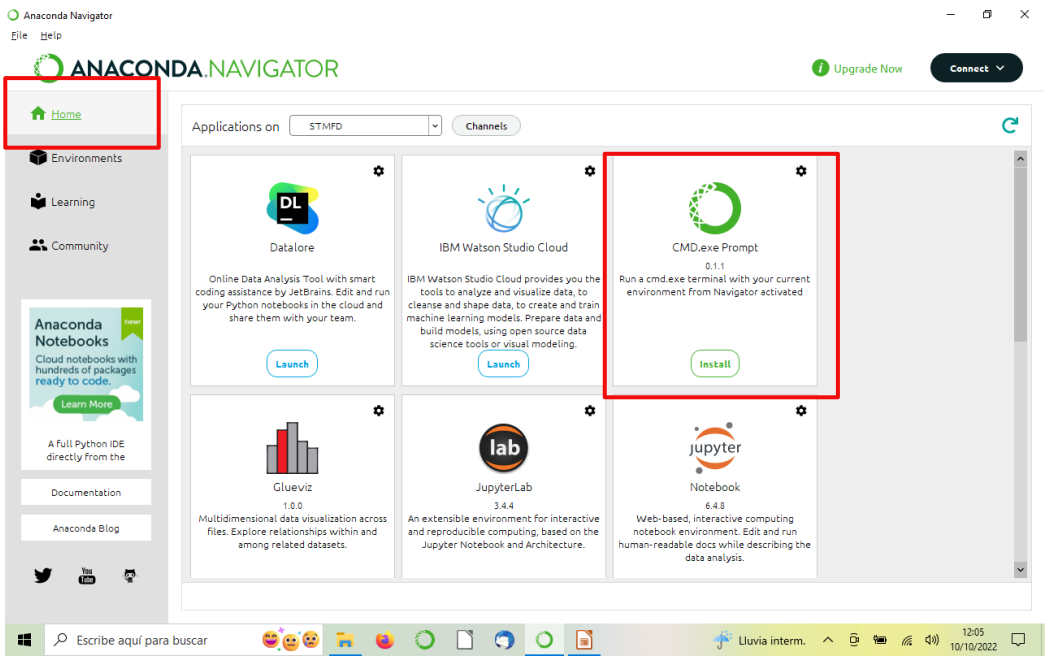
Crearemos un nuevo entorno, dándole el nombre que deseemos (por ejemplo: **ExpertoIA**). Seleccionaremos también una versión de Python a usar en el entorno; en nuestro caso, la última actualización de la **versión 3.9**.



Cuando haya finalizado la instalación, y también **cada vez que iniciemos Anaconda**, debemos asegurarnos de que el entorno deseado está activo:

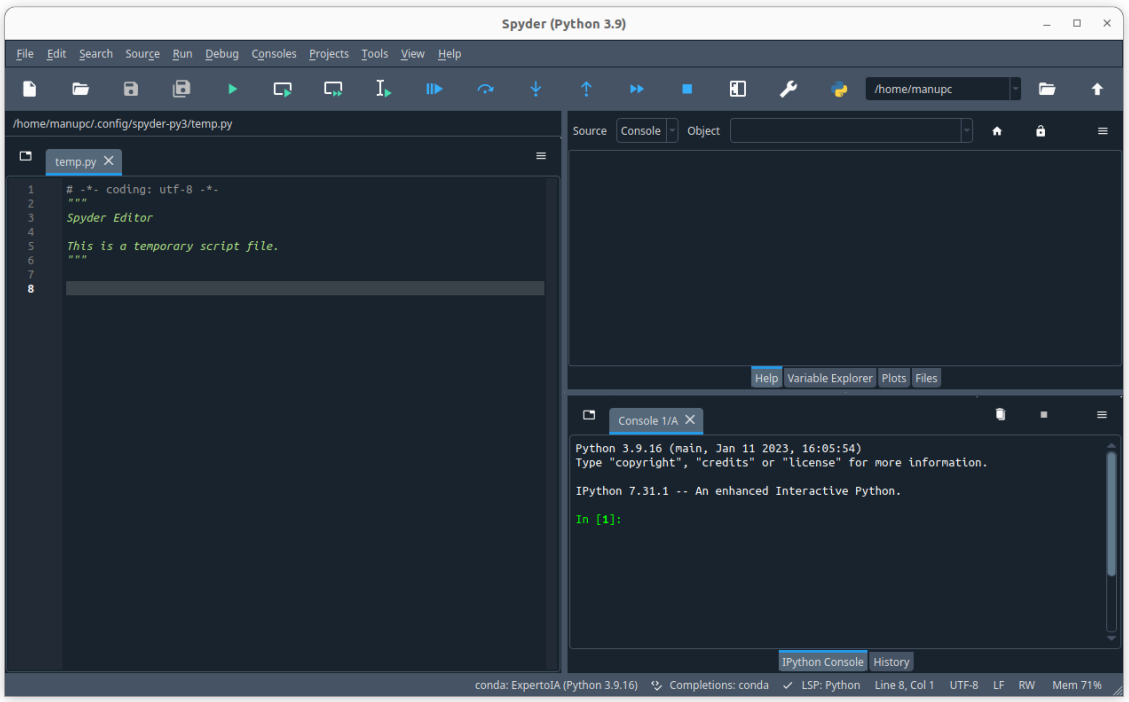


Con el entorno activo deseado, podemos ir a la opción **Home** para poder instalar software (bibliotecas), IDEs de desarrollo (**Instalar Spyder** en nuestro caso), o incluso ejecutar el entorno por línea de comandos (opción en la siguiente figura):



1.3. Primera impresión de Spyder

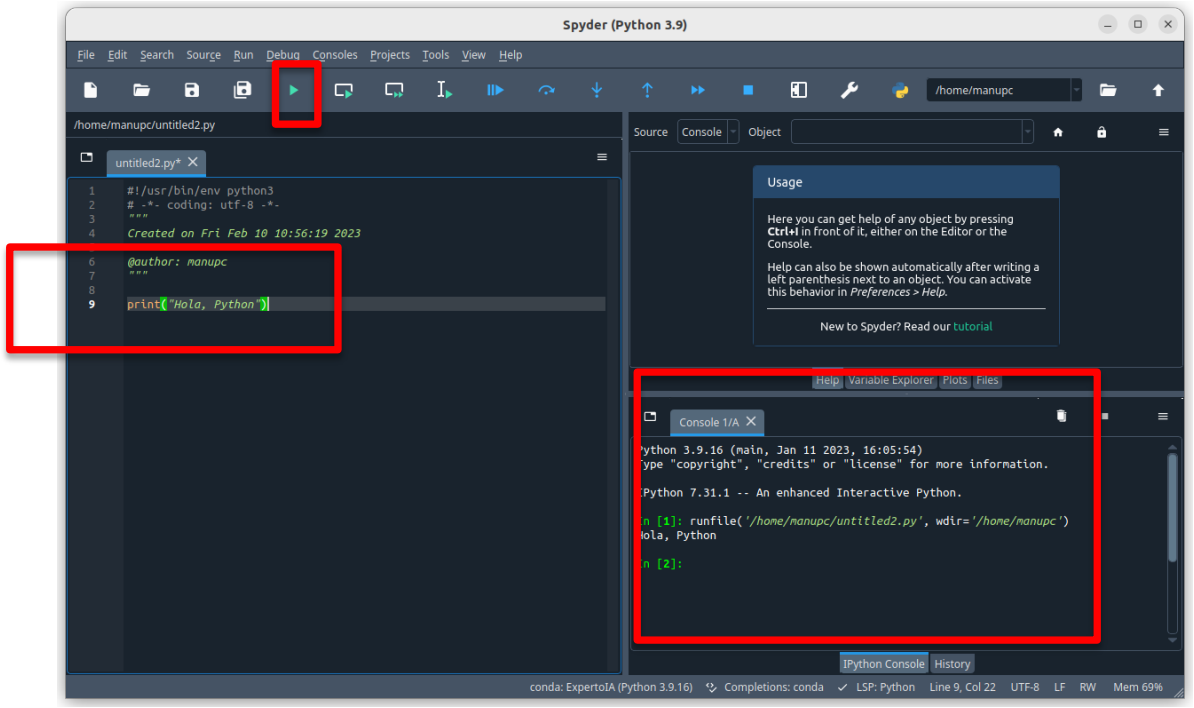
Spyder es un IDE Python fundamentalmente orientado a la depuración de programas y Ciencia de Datos. Permite, en la misma ventana, visualizar el programa, valores de variables, gráficos, consola, etc.



En la ventana del editor, escribiremos bajo todo el texto existente lo siguiente:

print("Hola, Python")

Seguidamente, pulsaremos el botón ***Run*** (triángulo verde de la barra de herramientas). Veremos el resultado en la consola (parte inferior derecha). Hemos construido nuestro primer programa Python.



2.Conceptos básicos de programación

Python es un lenguaje interpretado, pero también es un lenguaje ***débilmente tipado***. Esto significa, en términos prácticos, que en Python existen los tipos de datos (enteros, reales, cadenas de caracteres…), pero que:

- No se tienen porqué definir a priori las variables y constantes a utilizar en un programa.
- Las variables pueden tener tipos de datos distintos a lo largo del programa, dependiendo del dato al que referencien.
- Existe un **valor nulo**, sin tipo, cuya notación es **None**.

2.1. Operadores básicos

Al igual que en el resto de lenguajes de programación, el término **variable** se corresponde con un "*almacén donde se guarda un dato, estructurado o no*". Los operadores más básicos, al igual que en otros lenguajes, son:

- **Operador de asignación =.** Asigna un dato a una variable
- **Operadores aritméticos:** + (suma), - (resta), * (multiplicación), / (división), // (división entera –sólo la parte entera de la división entre dos números-), % (módulo – resto de dividir un número entre otro-), ** (exponenciación – elevar un número a otro).

La **sintaxis para escribir números** se corresponde con la escritura del número literalmente y, si este tiene decimales, utilizando el punto (.) para identificar la parte entera de la decimal. Por otra parte, la **sintaxis para escribir cadenas de caracteres** se corresponde con dos posibles formatos: La escritura del texto encerrada entre comillas dobles, o la escritura del texto encerrada entre comillas simples (carácter ', situado al lado del dígito 0 en el teclado español). También existe otra opción para modelar cadenas de caracteres multilínea, encerrando el texto entre triples comillas dobles (""").

La interacción con un programa escrito en Python, en su versión más simple, se realiza a través de la línea de comandos. Existen dos funciones básicas para ello:

- **Función print:** Permite escribir a consola los datos incluidos como parámetros.
- **Función input:** Permite obtener desde teclado un valor introducido por el programador en consola. El dato, a priori, se considera cadena de caracteres. Si queremos introducir números, se debe transformar a entero (función **int**) o a real (función **float**).

Los comentarios en Python pueden escribirse de dos formas:

- **Comentarios de una línea:** Comienzan con el carácter #
- **Comentarios de varias líneas:** Definiendo una cadena de caracteres multilínea con la triple dobles comillas (“””)

El siguiente código de ejemplo muestra estas características (fichero **Codigo1.py**):

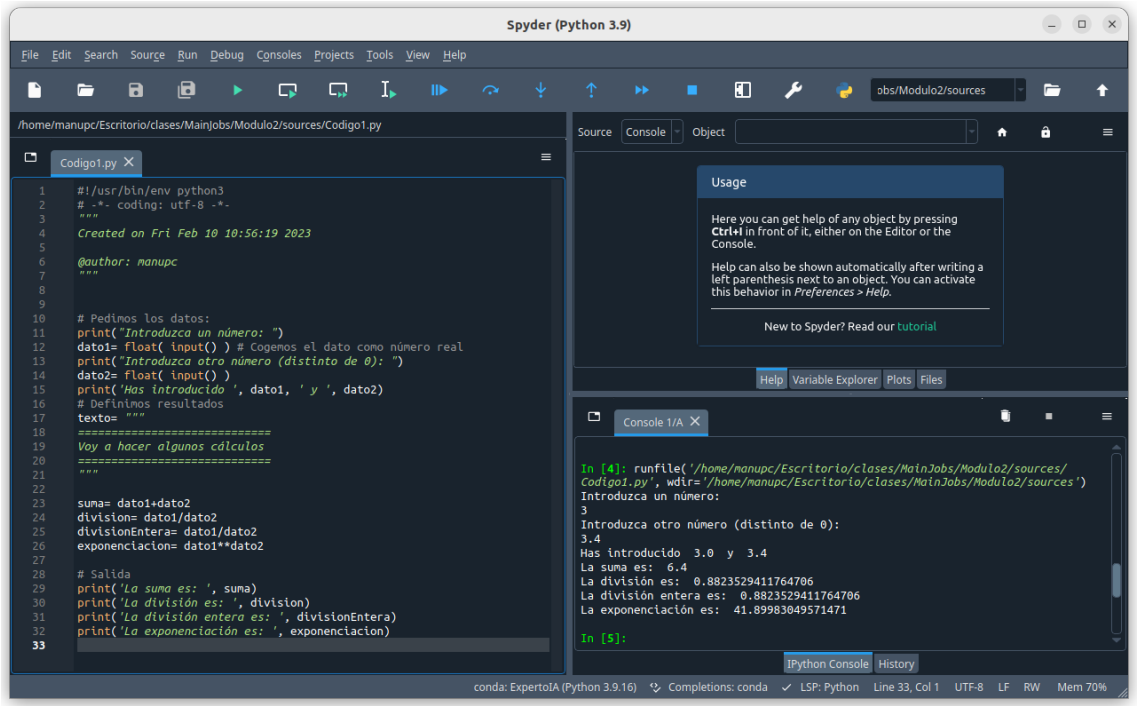
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 10 10:56:19 2023

@author: manupc
"""


# Pedimos los datos:
print("Introduzca un número: ")
dato1= float( input() ) # Cogemos el dato como número real
print("Introduzca otro número (distinto de 0): ")
dato2= float( input() )
print('Has introducido ', dato1, ' y ', dato2)
# Definimos resultados
texto= """
============================
Voy a hacer algunos cálculos
============================
"""

suma= dato1+dato2
division= dato1/dato2
divisionEntera= dato1/dato2
exponenciacion= dato1**dato2

# Salida
print('La suma es: ', suma)
print('La división es: ', division)
print('La división entera es: ', divisionEntera)
print('La exponenciación es: ', exponenciacion)
```



2.2. Sentencias condicionales

Lo normal es que un programa se ejecute sentencia a sentencia de principio a fin. No obstante, hay ocasiones en las que el comportamiento debe variar según alguna condición. Por ejemplo, supongamos que queremos comprobar si la entrada de un usuario por consola es un número positivo (no se permiten números negativos o 0). En este caso, tendremos que modificar el comportamiento del programa para que actúe según el dato introducido.

Valores y operadores lógicos

Una condición de ejecución de una sentencia de un programa puede darse (ser cierta) o no darse (ser falsa). En Python, estos valores se notan mediante los literales **true** y **false**.

Tradicionalmente, al tipo de dato de valor **true/false** se le conoce como tipo ***booleano***. Las operaciones lógicas sobre datos booleanos son:

- **Y:** notada como **and**. El operador se usa como **x and y**, donde **x,y** son datos de tipo booleano. Devuelve **true** si tanto **x** como **y** son ciertos, y **false** en caso contrario.
- **O:** notada como **or**. El operador se usa como **x or y**, donde **x,y** son datos de tipo booleano. Devuelve **true** si alguno de los datos **x** o **y** son ciertos, y **false** en caso contrario (los dos son falsos).
- **NO:** Notada como **not**. El operador se usa como **not x**, donde **x** es un dato de tipo booleano. Devuelve el valor contrario de **x (true** si **x** es falso, **false** si **x** es verdadero).

Todas estas operaciones se pueden agrupar en expresiones se mayor tamaño, a veces usando uso de paréntesis, como por ejemplo:

(x or y) and not (x and y)

Esta expresión devuelve **true** sólo cuando **x** o **y**, de forma exclusiva, valen **true** (es decir cuando **x** o **y** valen **true**, pero no ambos a la vez).

Operaciones de relación

Durante la ejecución de un programa, es común realizar tomas de decisiones en función de los valores de los datos usados. Los datos deben compararse, y tenemos en Python las siguientes operaciones de relación básicas:

- **Relación de igualdad:** Notada como == (dos iguales seguidos). Se usa como **x == y**. Devuelve **true** si **x** e **y** son iguales, y **false** en otro caso.
- **Relación de desigualdad:** Notada como **!=** (cierre de exclamación e igual seguidos). Se usa como **x != y**. Devuelve **true** si **x** e **y** son distintos, y **false** en otro caso. Se tiene que **(x==y) == not (x!=y)** vale cierto siempre.
- **Relaciones de orden**: < (menor), <= (menor o igual), > (mayor), >= (mayor o igual). Se usan como **x<y**, **x<=y**, **x>y**, **x>=y**. Devuelven **true** si se cumple la condición asociada al operador, y **false** en caso contrario.
- **Relación de pertenencia**: Se nota como **in**. Sirve para conocer si un dato se encuentra en un conjunto de datos dado.
- **Relación de ser**: Se nota como **is**. Sirve para conocer si un dato es algún valor o no. Se usa con frecuencia para comprobar si los datos son nulos (**dato is None**) o no (**dato is not None**).

Ejemplo: Operadores relacionales y lógicos

Sean las variables definidas con los valores:

A= 3
B= 2
C= 7

Entonces:
A<B and B<C es **true**
A+B>=C es **false**
not (A<C or B>C) es **false**

La sentencia *if*

Su sintaxis es:

```
1. if condicion:
2.     sentencias
3. sentenciaFinal
```

donde **condición** representa una expresión booleana, usando tipos lógicos, u operadores relacionales. Las **sentencias** sólo se ejecutan si la condición es cierta. La **sentenciaFinal** se ejecuta siempre (está fuera del bloque ***if***).

IMPORTANTE: Prestar atención a los dos puntos (:) al final de la condición, y a la tabulación (espacios) en las sentencias de la condición.

Ejemplo: Sentencia if
Ver fichero Python asociado: **Codigo2.py**

La sentencia *if..else*

Su sintaxis es:

```
1. if condicion:
2.     sentencias
3. else:
4.     OtrasSentencias
5. sentenciaFinal
```

Si se cumple la condición, se ejecutan las sentencias **sentencias**. En caso contrario, si no se cumple se ejecuta **OtrasSentencias.**

Ejemplo: Sentencia if..else

Ver fichero Python asociado: **Codigo3.py**

La sentencia **if..else** se puede combinar con el operador de asignación para realizar asignaciones de valores condicionados de una forma más compacta:

```
A= 8.5
B= 'Aprobado' if A>=5 else 'Suspenso'
Print('El alumno está: ', B)
```

La sentencia *if..elif..*

Su sintaxis es:

```
if condicion1:
    sentencias1
elif condicion2:
    sentencias2
else: # Opcional
    sentenciasElse # Opcional

sentenciaFinal
```

Si se cumple la **condicion1**, se ejecutan las sentencias **sentencias1** y se sigue por **sentenciaFinal**. En caso contrario, si no se cumple se comprueba si se cumple **condicion2** y, si se cumple se ejecuta **sentencias2** y se sigue por **sentenciaFinal.** En caso de que no se cumplan ni **condicion1** ni **condicion2**, se ejecuta **sentenciasElse** y después **sentenciaFinal.**

Ejemplo: Sentencia if..elif..

Ver fichero Python asociado: **Codigo4.py**

2.3. Sentencias repetitivas

Las sentencias repetitivas, o **bucles,** e utilizan para repetir un mismo bloque de sentencias múltiples veces (un número definido, o hasta que se cumpla una condición).

La sentencia *while*

Su sintaxis es:

```
1. while condicion:
2.     sentencias1
3. sentenciaFinal
```

El bloque de sentencias **sentencias1** se ejecuta mientras se cumpla **condicion**. Posteriormente, pasa a ejecutarse **sentenciaFinal**. **IMPORTANTE: Nótese también los dos puntos (:) al final de la condición, y la tabulación (espacios) del bloque de sentencias incluido dentro del while**.

Ejemplo: Sentencia while

Ver fichero Python asociado: **Codigo5.py**

La sentencia *for*

Su sintaxis es:

```
1. for variable in coleccion:
2.     sentencias1
3. sentenciaFinal
```

El bloque itera dando valor a la variable **variable** con cada uno de los elementos existentes en el conjunto de elementos **coleccion**. El bloque **sentencias1** se ejecuta una vez por cada valor en **colección**, dando este valor a **variable**.

Cuando **coleccion** no es un valor numérico, se puede también usar una variante de la sentencia **for** capaz de devolver tanto el valor de cada elemento de la colección como un índice que cuenta a partir de 0. Esto se consigue gracias a la función **enumerate**:

```
1. for índice, variable in enumerate( coleccion ):
2.     sentencias1
3. sentenciaFinal
```

Ejemplo: Sentencia for

Ver fichero Python asociado: **Codigo6.py**

La función *range*

Una forma común de generar colecciones numéricas en Python es usando la función **range**. Tiene varias posibles sintaxis:

- **range(entero)**: Genera una secuencia de números enteros desde 0 hasta entero-1 inclusive. Por ejemplo, **range(5)** genera la secuencia {0, 1, 2, 3, 4}.
- **range(inicio, fin)**: Genera una secuencia de números enteros desde **inicio** hasta **fin-1** inclusive. Por ejemplo, **range(5, 7)** genera la secuencia {5, 6}.
- **range(inicio, fin, pasos)**: Genera una secuencia de números enteros desde **inicio** hasta **fin-1** inclusive, en saldos de **pasos** en **pasos**. Por ejemplo, **range(0,10,2)** genera la secuencia {0, 2, 4, 6, 8}.

Ejemplo: Sentencia for con función range

Ver fichero Python asociado: **Codigo7.py**

Las sentencias *break* y *continue*

En ocasiones, es posible que se dé alguna condición que nos obligue a terminar un bucle antes de lo esperado, o también pasar a la siguiente iteración. La sentencia **break** permite terminar completamente la ejecución del código del bucle, continuando por la siguiente sentencia al mismo. Por su parte, la sentencia **continue** permite terminar la iteración actual sin ejecutar el resto de sentencias del bucle, pasando directamente a la siguiente iteración.

Ejemplo: Uso de break y continue

Ver fichero Python asociado: **Codigo8.py**

2.4. Modularización y funciones

El código fuente de un programa se suele estructuras en **funciones** más pequeñas que realizan parte de código. A su vez, estas funciones pueden formar parte de **bibliotecas**. Una biblioteca se puede ver como un paquete adicional de Python que incluye constantes, funciones y estructuras de programación relacionadas con una temática (por ejemplo: cálculos matemáticos, visualización de datos, control de conexiones a Internet, etc.).

Uso de funciones

Una función es un subprograma que puede tener ninguna o varias entradas, y proporcionar ninguna o varias salidas. Una llamada a función se realiza escribiendo el nombre de la función seguido de paréntesis (abrir paréntesis "(" y cerrar paréntesis ")"). En caso de que la función tenga datos de entrada, estos van separados por comas dentro de los paréntesis.

Una función puede no devolver ningún valor (por ejemplo, la función **print** ya utilizada, o devolver uno o múltiples valores (por ejemplo, la función **input** ya utilizada). Si una función devuelve más de un valor, estos se pueden recoger separados por comas y asignando los valores con el operador de asignación.

Existen funciones incluidas por defecto en Python (por ejemplo, la función **len**, que tiene como entrada una secuencia y devuelve el número de elementos de la misma), aunque lo más normal es usar también funciones incluidas en bibliotecas externas.

Ejemplo: Llamadas a funciones
Ver fichero Python asociado: **Codigo9.py**

Existen también variantes en las que es posible hacer una llamada a función nombrando explícitamente parámetros y valores asociados, que estudiaremos más adelante.

Definición de funciones

La definición de funciones nos permite poder encapsular código en módulos más pequeños, para reutilizarlo en el futuro y para que el código de nuestro programa quede más estructurado y limpio. La definición de funciones tiene la siguiente sintaxis:

```
def nombreFuncion([variable1[, otraVariable, …] ]):
    Código de la función
```

Los términos encerrados entre corchetes son opcionales. Se usan para indicar que una función puede no tener datos de entrada, tener sólo uno, o tener más de uno (en este caso, separados por comas). Un ejemplo de definición de función simple que muestra un saludo:

```
def Saludo(nombre= None):
    if nombre is not None:
        print("Hola, amigo ", nombre)
    else:
        print("No me has dicho tu nombre")
```

Su uso en un programa sería:

```
Saludo() -> Muestra: No me has dicho tu nombre
Saludo("Manuel") -> Muestra: Hola, amigo Manuel
Saludo(nombre="Lolo") -> Muestra: Hola, amigo Lolo
```

Nótese que las variables definidas como entradas a la función pueden tener valores por defecto (en este caso el valor "None"). Ante la ausencia del parámetro como entrada a la función (línea 1) se toma el valor por defecto.

Además, observamos que es posible asignar valores a parámetros concretos de la función (línea 3). En este caso, se asigna el valor "Lolo" al parámetro de entrada **nombre**.

Ejemplo: Código del ejemplo anterior

Ver fichero Python asociado: **Codigo10.py**

Definición de funciones que devuelven valores

La definición e implementación de funciones que realizan algún tipo de cálculo y devuelven uno o más valores no difiere de lo estudiado hasta ahora. No obstante, si se desea devolver algún valor, Python nos aporta dos posibilidades:

- **Palabra reservada *return*:** Permite devolver un cálculo realizado en la función (o varios), y devolverlos. Tras la ejecución de un **return**, la función termina y las sentencias siguientes **no siguen ejecutándose**. Además, si se desea devolver más de un valor, estos deben estar separados por comas. Ejemplo:

```
1. def MaximoMinimo(dato1, dato2):
2.     if dato1 > dato2:
3.         Maximo= dato1
4.         Minimo= dato2
5.     else:
6.         Maximo= dato2
7.         Minimo= dato1
8.     return minimo, máximo # Se devuelven dos valores
```

El uso de la función en un programa sería:

```
A= 32
B= 27
ValorMinimo, ValorMaximo= MaximoMinimo(A, B)
```

Palabra reservada *yield*: la sintaxis es similar a **return**, aunque el comportamiento de **yield** es muy diferente. Mientras que **return** finaliza la llamada a la función, **yield** "deja en suspenso" la ejecución de la función, y la reanuda desde el punto donde se quedó la siguiente vez que se llame. Es útil, sobre todo, cuando se generan datos iterativamente. Un ejemplo de su uso:

```
def Contador(ValorFin):
    contador= 0
    while contador<ValorFin:
        yield contador
        contador= contador+1
```

El uso de la función en un programa sería:

```
for dato in Contador(7):
    print(dato)
```

El funcionamiento de la función **Contador** es similar a **range** con un único parámetro.

Ejemplo: Código del ejemplo anterior

Ver fichero Python asociado: **Codigo11.py**

Cabe destacar que, al contrario que otros lenguajes, **Python no incorpora la devolución de parámetros por referencia**, aunque los parámetros de entrada que sean tipos compuestos sí es posible modificarlos dentro de una función.

Las palabras reservadas pass y raise

Es frecuente que, cuando escribimos un programa, necesitemos funciones que aún no están implementadas (bien porque otras personas se encargan de ello, bien porque hemos planificado su construcción para el futuro). En este caso, sí resulta necesario conocer la cabecera de la función (aunque esta no esté implementada aún). Para que la ejecución de nuestro programa no falle, podemos usar las palabras reservadas **pass** (sentencia nula) o **raise** (lanzamiento de una excepción).

```
1. def UnaFuncion(Parametro= None):
2.     pass # Sentencia nula. No hace nada
3.
4. UnaFuncion() # Llamada a la función. No da error
```

```
1. def OtraFuncion(Parametro= None):
2.     raise Exception('Error: Función aún no implementada')
3.
4. OtraFuncion() # Lanza un error de excepción definido
```

Ejemplo: Código del ejemplo anterior

Ver fichero Python asociado: **Codigo11_2.py**

Uso de bibliotecas externas: import

La palabra clave **import** en Python se utiliza para incluir en el programa bibliotecas, o parte de ellas, de modo que podamos usar su funcionalidad dentro de nuestro programa. Existen diferentes formas de usar un **import**:

- **Importación directa: import *biblioteca***. Permite cargar el código de la biblioteca ***biblioteca*** dentro de nuestro programa. La llamada a funciones y características de esta biblioteca se realiza escribiendo el nombre de la biblioteca, seguido de un punto (.) y el nombre de la característica (función, variable, etc.) que queramos usar. Ejemplo para usar la función raíz cuadrada de la biblioteca **math**:

```
1. import math
2. print(math.sqrt(9)) # Muestra el valor 3.0 por consola
```

- **Importación con alias: import *biblioteca* as *alias***. Permite cargar el código de la biblioteca ***biblioteca*** dentro de nuestro programa, pero renombrándolo con el alias seleccionado. Ejemplo para usar la función raíz cuadrada de la biblioteca **math**:

```
1. import math as Mates
2. print(Mates.sqrt(9)) # Muestra el valor 3.0 por consola
```

- **Importación de partes de bibliotecas: from *biblioteca* import *caracteristica1, caracteristica2, ….*** Permite cargar el código de la biblioteca ***biblioteca***, pero sólo para las características indicadas (separadas por comas). Si se desea incorporar todas las características, en lugar de enumerarlas todas separadas por comas, se puede poner un único símbolo asterisco (*). En estos casos no es necesario indicar el nombre de la biblioteca al usar la característica. Ejemplo para usar la función raíz cuadrada de la biblioteca **math**:

```
1. from math import import sqrt
2. print(sqrt(9)) # Muestra el valor 3.0 por consola
```

También es posible combinar las diferentes alternativas de **import** dentro de una o varias líneas de código.

Ejemplo: Uso de imports

Ver fichero Python asociado: **Codigo12.py**

3.Estructuras de datos en Python

Python es uno de los pocos lenguajes de programación que incorpora estructuras de datos complejas de forma nativa. En particular, permite definir estructuras de tipo lista, conjunto, tuplas y diccionarios.

Es importante diferenciar entre dos clases de tipos de datos complejos:

- **Objetos inmutables:** Son aquellos que, una vez creados, no se pueden modificar. Entre los tipos nativos de Python inmutables encontramos las tuplas.
- **Objetos mutables**: Son aquellos que sí se pueden modificar con posterioridad a su creación. Entre los tipos nativos de Python mutables encontramos las listas, conjuntos y los diccionarios (se pueden insertar o modificar elementos ya existentes en este tipo de estructuras).

Listas

Las listas son secuencias de elementos. Dado que Python es un lenguaje débilmente tipado, es posible crear listas de datos de distinto tipo.

El operador que permite la definición de listas es el corchete. Se puede crear una lista explícitamente usando este operador:

```
1. UnaLista= ['Leche', 'Pan', 'Huevos']
2. UnaLista= ['Leche', 'Pan', 'Huevos']
3. UnaListaVacia= [] # Lista que no contiene ningún elemento
```

El **acceso a los elementos** de una lista se realiza también a través del operador corchete, indicando el índice del elemento al que se desea acceder (desde el 0 hasta el número de elementos -1). También es posible acceder a los elementos de una lista desde el final, comenzando por "-1" (último elemento de la lista), "-2" (penúltimo elemento de la lista), etc.:

```
1. print( UnaLista[0] ) # Muestra "Leche"
2. print( UnaLista[2] ) # Muestra "Huevos"
3. print( UnaLista[-1] ) # Muestra "Huevos"
```

Se puede conocer el número de elementos de la lista mediante la función **len**:

```
1. print(len(UnaLista)) # Muestra 3, el número de elementos
```

Si lo que deseamos es conocer si la lista está vacía o no, mediante una sentencia condicional, el mismo nombre de la lista se puede usar dentro de la condición (devolverá **true** si contiene elementos y **false** en caso contrario):

```
1. if UnaListaVacia:
      1. print( "Esto no se ejecuta" )
2. else:
      1. print( "La lista está vacía" )
```

Algunas operaciones comunes sobre listas son:

- **Lista.append(x):** Añade el elemento **x** al final de la lista **Lista**.
- **Lista.insert(i, x):** Añade el elemento **x** a la lista **Lista** en la posición **i**.
- **Lista.remove(x):** Elimina el elemento **x** de la lista **Lista** (si está repetido, sólo elimina la primera ocurrencia).
- **Lista.pop([i]):** Los corchetes en este caso significa que el parámetro es opcional. Elimina el elemento que se encuentra en la **i**-ésima posición de la lista, y lo devuelve como resultado. Si no se indica, elimina el último elemento de la lista.

- **Lista.clear():** Elimina todos los elementos de la lista **Lista**, dejándola vacía.
- **Lista.reverse():** Devuelve una nueva lista, con los mismos elementos que **Lista** pero en orden inverso.
- **Lista.copy():** Devuelve una copia de la lista.
- **Lista.sort(key=None, reverse=False):** Ordena la lista. Los argumentos *key* y *reverse* se pueden usar para personalizar el criterio de ordenación y el valor sobre el que se ordena, en caso de que los elementos de la lista sean objetos complejos.

Además, se puede combinar el operador de iteración **for** con listas para creación de una secuencia de elementos en tiempo de ejecución:

```
1. L= [x for x in range(5)] # Lista con los elementos [0,1,2,3,4]
```

Las listas son **iterables**; es decir, podemos crear un bucle que recorra cada uno de los elementos de la lista:

```
1. L2= ['Perro', 'Gato', 'Coballa']
2. for animal in L2:
      1. print(animal)
```

Listas de listas

Como un elemento de una lista puede ser cualquier cosa, ¿qué impode que también sea una lista? Nada. Así podemos crear listas anidadas y hacer algo más complejo nuestro programa:

```
1. L= []
2. for fila in range(3):
      1. L.append([]) # Insertamos lista vacía al final


      2. for columna in range(3):
            2  L[fila].append(3*fila+columna)
3. print(L)
```

El código anterior equivale a crear manualmente la siguiente lista:

```
1. L= [ [0, 1, 2], [3, 4, 5], [6, 7, 8] ]
2. print(L)
```

El siguiente código fuente muestra un ejemplo del uso de listas en Python:

Ejemplo: Uso de listas en Python
Ver fichero Python asociado: **Codigo13.py**

CUIDADO: Las listas son **objetos referenciados**. Esto quiere decir que si, por ejemplo, hacemos una asignación de una variable a otra que contiene una lista, **ambas variables referenciarán a la misma lista, no se crea copia**. Si se desea hacer una copia, se debe usar la opción **copy()** de la lista.

Tuplas

Las tuplas son objetos inmutables. Pueden parecer similares a simple vista, pero se diferencian principalmente en que no se pueden modificar, ni tampoco añadir/eliminar elementos de una tupla ya creada. El operador de definición de tuplas es el paréntesis "(" , ")".

Un ejemplo:

```
1. L= ('Mariano', 'López', 'Pérez', 8.5, 'Aprobado')
2. print(L)
```

Las tuplas no disponen de operaciones especiales como append, remove, etc., dado que son estructuras inmutables. No obstante, se puede acceder a cada elemento de una tupla, conocer el número de elementos, etc., como si fueran listas.

Ejemplo: Uso de tuplas en Python

Ver fichero Python asociado: **Codigo14.py**

Conjuntos

Los conjuntos son colecciones no ordenadas de **elementos sin repetir**. Un conjunto es también, al igual que las listas, mutable, y se puede realizar operaciones entre conjuntos. El operador de definición de conjuntos es la llave "{", "}". No obstante, los **diccionarios** también utilizan la llave para su definición, por lo que **Python** incluye el tipo **set() para crear conjuntos**. En particular, la instrucción **set()** sin parámetros genera un conjunto vacío:

```
1. C= {x for x in 'Mañana'}
2. print( C ) # Muestra {'a', 'n', 'ñ', 'M'}
```

Algunas operaciones entre conjuntos son:

- **Set.add(x):** Añade el elemento **x** al final del conjunto **Set**.
- **Set.remove(x):** Elimina el elemento **x** de la lista **Lista** (si está repetido, sólo elimina la primera ocurrencia).

- **Set.pop([x]):** Los corchetes en este caso significa que el parámetro es opcional. Elimina el elemento **x** del conjunto y lo devuelve. Sin parámetros elimina un elemento cualquiera del conjunto y lo devuelve.
- **Set.clear():** Elimina todos los elementos del conjunto **Set**, dejándolo vacío.
- **Set.copy():** Devuelve una copia del conjunto.
- **Set.union(s)**: Realiza la unión del conjunto **Set** con el conjunto **s** pasado por argumento. Devuelve el conjunto resultante como un nuevo conjunto.
- **Set.intersection(s)**: Realiza la intersección del conjunto **Set** con el conjunto **s** pasado por argumento. Devuelve el conjunto resultante como un nuevo conjunto.
- **Set.difference(s)**: Realiza la diferencia entre conjuntos **Set\s**. Devuelve el conjunto resultante como un nuevo conjunto.
- **Set.issubset(s)**: Comprueba si **Set** es subconjunto de **s**. En tal caso, devuelve **true** (**false** en caso contrario).
- **x in Set:** Comprueba si un elemento **x** se encuentra dentro del conjunto **Set**. Devuelve **true** si lo contiene, y **false** en caso contrario.

Ejemplo: Uso de conjuntos en Python

Ver fichero Python asociado: **Codigo15.py**

Diccionarios

Los conjuntos son colecciones no ordenadas de pares (clave, valor). En otros lenguajes, se conocen también como *memoria asociativa* o, más comúnmente, implementaciones de *tablas hash*. Un diccionario asigna un valor a una clave (única), de modo que los datos puedan ser recuperados fácilmente a partir del valor **clave/key**. Por ejemplo, la recuperación de datos de una persona (**value/value**) de una Base de Datos en función del DNI (**key**). El operador de generación de diccionarios en Python es también la llave "{", "}". Se puede crear un diccionario vacío como "{}". Algunos ejemplos:

```
D= {} # Diccionario vacío
D= {'11223344A': ('Perico', 8.5), '22334455B': ('María', 5.1) }
```

El acceso a los valores de un diccionario se realiza a través de su clave:

```
print(D['11223344A') # muestra ('Perico', 8.5)
```

Los diccionarios, al igual que todos los tipos de datos complejos que hemos estudiado anteriormente, también son iterables. La diferencia está en que se itera sobre **las claves del diccionario**:

```
for clave in D:
   print(clave) # 11223344A, 22334455B, ...
```

Las claves de un diccionario son accesibles como un conjunto, a través de la funcionalidad **keys()**:

```
print(D.keys()) #{'11223344A', '22334455B', …}
```

También se puede iterar sobre los pares (clave, valor) mediante la funcionalidad **items()**:

```
for clave, valor in D.items():
  print('A', clave, 'le corresponde el valor', valor)
```

Se puede conocer si un valor de clave está dentro de un diccionario mediante la pertenencia al conjunto de claves. Por ejemplo:

```
print('11223344A' in D.keys()) # Muestra True
```

La inclusión o modificación de ítems dentro de un diccionario se puede realizar mediante el acceso a los elementos por clave (operador de corchete):

```
D['33445566C']= ('Juan', 8.9)
```

Ejemplo: Uso de diccionarios en Python

Ver fichero Python asociado: **Codigo16.py**

4. Algoritmos

Un **algoritmo** es una secuencia finita ordenada de pasos, libres de ambigüedad que, cuando se ejecutan fielmente sobre unos datos de entrada, siempre proporciona las mismas salidas. Usualmente, en Python implementaremos algoritmos dentro de funciones que, a su vez, serán utilizadas desde el cuerpo principal del programa. Estas funciones podrán estar dentro del mismo fichero de código fuente, o en ficheros externos que podremos importar.

Modularización en ficheros

Un fichero Python que contiene funciones o recursos de programación puede importarse desde otro programa Python, tanto completa como parcialmente.

Como ejemplo, implementaremos funciones que calculen la media y la desviación típica de un conjunto de datos numéricos, dados como entrada al algoritmo como una lista. Ambas funciones las implementaremos dentro del fichero **funciones.py**, y tendrán la siguiente cabecera:

```
1. def Media(ListaValores : list[float]) -> float
2. def DesvTipica(ListaValores : list[float]) -> float
```

NOTA: Fijémonos en que, aunque en Python no es obligatorio, en el código anterior estamos forzando a que la entrada sea una lista, cuyos elementos sean número reales. También se fuerza a que la salida sea un número real

Conociendo la ruta del fichero (o bien colocándolo en la misma carpeta del programa principal), se podrá usar el fichero como otra biblioteca más:

```
import funciones as Stat

print('Media: ', Stat.Media([1, 2, 3]))

print('Dev. Estándar: ', Stat.DesvTipica([1, 2, 3]))
```

Ejemplo: Modularización en ficheros

Ver fichero Python asociado: **Codigo17.py**

5. Programación orientada a objetos

La **programación orientada a objetos (POO)** nos permite abstraer mejor el proceso para modelar datos estructurados y no estructurados, creando nuevos tipos de datos que posteriormente podrán ser reutilizados para especificar o especializar otros, mediante mecanismos como la **herencia**.

Clases

El concepto básico en POO es el de clase y objeto. Una **clase** se compone por la definición de datos y operaciones que se pueden dar sobre un tipo de datos que deseamos crear. Por su parte **un objeto de una clase** en nuestro caso será una variable cuyo valor es un dato específico de la clase creada. Por ejemplo, si queremos crear un nuevo tipo de dato **Complejo** que defina a un número completo, este vendría dado por la definición de:

- **Atributos:** Aquellas variables que ayuden a representar un número complejo.
- **Métodos:** Aquellas funciones que permiten operar sobre un objeto de tipo complejo.

Para crear un objeto de tipo **Complejo**, se deberá definir una función especial denominada **constructor**, que deberá tener los parámetros necesarios para crearnos nuestro objeto. Esta función se denomina **__init__()**. Un ejemplo que genera un número complejo con dos valores (uno para la parte real, otro para la imaginaria):

```
class Complejo: # Definición del tipo de dato

  def __init__(self, real= 0, imag= 0):

    self.parteReal= real

    self.parteImag= imag
```

Notemos que:

- Los **métodos** de la clase siempre tienen como primer parámetro **self** (referencia al objeto actual).
- **Los atributos del objeto** (parteReal, parteImag) **también van precedidos por self.**, para indicar que son atributos definidos en la clase.

En el fichero **CodigoComplejo.py** crearemos la clase para modelar un número complejo, incorporando además funcionalidades para:

- Calcular el módulo (valor absoluto)
- Calcular el conjugado
- Sumar/restar complejos

Aunque las operaciones sobre complejos son más extensas, nos limitamos a estas simplemente a objeto ilustrativo.

Ejemplo: Creación de la clase complejo

Ver ficheros Python asociados: **CodigoComplejo.py y Codigo18.py**

Herencia

La herencia, como mecanismo de abstracción para especialización/especificación, se encuentra disponible en Python. Se realiza de la siguiente forma:

```
class NuevaClase : ClasePadre:
   pass
```

Al contrario que en otros lenguajes, a la hora de generar un objeto de la nueva clase con el constructor, no se llama de forma automática a los constructores de las clases padre. En su lugar, hay que hacerlo de forma manual mediante el operador **super**.

```
class NuevaClase( ClasePadre ):

   def __init__(self, param1, param2):

      super().__init__(ClasePadre)

      ...
```

El siguiente código de ejemplo muestra cómo realizar herencia en Python y cómo usar las clases heredadas. Adicionalmente, también es posible realizar **herencia múltiple**, aunque dicha característica queda fuera de los límites de este curso.

Ejemplo: Herencia en Python

Ver ficheros Python asociados: **CodigoAnimales.py y Codigo19.py**

6. La biblioteca NumPy

La biblioteca **NumPy** es un referente en Python para la realización de cálculos matemáticos. Es una biblioteca muy completa que introduce, además de los tipos de datos nativos de Python, el array/array multidimensional. Pese a no ser nativa, es una biblioteca muy eficiente dado que está implementada a bajo nivel en lenguaje C/C++. Se puede instalar a través del gestor de Anaconda, o bien por línea de comandos desde el entorno creado para el curso, escribirendo:

pip install numpy, o bien:

conda install numpy

La API de NumPy es muy extensa, aunque se puede consultar de forma interactiva por web en la siguiente URL:

https://numpy.org/doc/stable/reference/index.html

En esta sección, realizaremos un repaso de las capacidades básicas de esta biblioteca. Normalmente, importaremos la biblioteca con un alias reducido, que usualmente es:

import numpy as np

Tipos de datos

El tipo de dato fundamental en NumPy es el array (**ndarray**). Dentro de NumPy, todo es tratado como tipo **ndarray**, que se corresponde con el concepto clásico de array (unidimensional, o multidimensional). Los **ndarrays** de NumPy son colecciones de tipo array donde todos los elementos deben tener el mismo tipo (normalmente numérico). Además de los tipos clásicos incluidos nativamente (**int** o **float**), numpy introduce los siguientes:

- **np.int64**: Equivale al nativo **int**. Entero de 64 bits.
- **np.float64**: Equivale al nativo **float**. Real de 64 bits.
- **np.int32 / np.float32**: Entero/real de 32 bits
- **np.int16 / np.float16**: Entero/real de 16 bits
- **np.uint8 / np.int8**: Entero (sin signo/con signo) de 8 bits
- **np.complex128**: Complejo de 128 bits (64 parte real, 64 parte imaginaria).
- **Etc**.

Creación de arrays

La generación de arrays puede realizarse de distintas formas, y permite la transformación de tipos nativos Python (listas, tuplas) en un **ndarray**:

- **np.array(lista)**: Crea un array a partir de una lista. Ejemplo: **a= np.array([1.1, 2.2, 3.3, 4.4])**.
- **np.zeros((dim1, dim2, ...))**: Crea un array de 0's, multidimensional (tantas dimensiones como aparezcan en la tupla de entrada, tantas componentes por dimensión como valor tenga **dimx**. Ejemplo: **a= np.zeros((2, 3))**. Crea un array 2D (matriz) con 2 filas y 3 columnas, y todos los valores a 0.
- **np.ones((dim1, dim2, ...))**: De funcionamiento similar a **np.zeros**, pero inicializando todas las componentes a valor 1. Ejemplo: **a= np.ones(3)** crea un array 1D con 3 componentes, todas igual a 1.
- **np.empty((dim1, dim2, ...))**: Crea un array multidimensional, sin valores inicializados por defecto, Ejemplo: **a= np.empty(2, 3, 4)** crea una matriz 3D con 3 dimensiones: 2 celdas en la primera, 3 en la segunda y 4 en la tercera.

- **np.arange(start, end, step)**: Similar al funcionamiento de la función **range** de Python.

- **np.linspace(start, end, n)**: Crea un array con **n** componentes, igualmente espaciadas entre los valores **start** y **end** (inclusive). Ejemplo: **a= np.linspace(10, 30, 3)** devuelve el array **[10, 20, 30]**.

Se puede especificar el tipo de dato de cada componente a la hora de crear un array, mediante el parámetro **dtype**. Por ejemplo:

A= np.zeros(3, dtype=np.int32) -> Crea un array de 3 componentes todas iguales a 0 y de tipo int32.

Los arrays son objetos de tipo **ndarray**. Tienen las siguientes propiedades accesibles:

- **array.size:** Número total de componentes del array. Por ejemplo: **a= np.zeros((2, 3))** , **a.size** vale 6.

- **array.ndim:** Número total de dimensiones del array. Por ejemplo: **a= np.zeros((4, 3))** , **a.ndim** vale 2.

- **array.shape:** Número total de componentes por cada dimensión del array. Por ejemplo: **a= np.zeros((2, 3))** , **a.shape** vale **(2, 3)**. Al tratarse de una tupla, se puede también acceder al número de componentes de una dimensión particular. Por ejemplo: **a.shape[0]** vale **2** y **a.shape[1]** vale 3.

Ejemplo: Creación de arrays

Ver ficheros Python asociados: **Codigo20.py**

Indexación de arrays

Una ventaja importante de NumPy reside en las facilidades para indexar componentes y sub-arrays. De forma básica, se puede acceder a cualquier componente con el operador corchete:

```
A= np.array([ 10, 20, 30])

print(A[1]) # muestra 20

B= np.array( [ [10, 20, 30], [40, 50, 60] ] )

print(B[1, 2]) # muestra 60
```

Se pueden también seleccionar múltiples sub-arrays, usando el operador dos puntos "**:**" :

```
A= np.arange(5)

Print(A[1:-1]) # Muestra 1, 2, 3. Se selecciona el subarray desde 1
hasta la penúltima componente

B= np.array([ [1, 2, 3], [4, 5, 6] ])

print(B[1, :]) # Muestra todas las componentes de la segunda fila (4,
5, 6), array de 1x3

print(B[:, 0]) # Muestra todas las componentes de la primera columna (1,
4), array de 2x1
```

Otra posibilidad que añade NumPy es la indexación lógica, o la selección de celdas conforme a una condición dada:

```
A= np.array([1, 2, 3, 4, 5, 6])

A>=3.5 -> Devuelve el array [False, False, False, True, True, True]

A[A>=3.5] -> Devuelve [4, 5, 6]
```

Ejemplo: Indexación de arrays

Ver ficheros Python asociados: **Codigo21.py**

Operaciones básicas de arrays

Los arrays tienen por defecto implementadas las operaciones aritméticas básicas (+, -, *, /, %). **OJO: el * es multiplicación elemento a elemento (no multiplicación matricial)**.

Ejemplos:

```
A= np.array([1, 2, 3])
B= np.array([4, 5, 6])
C= A+B # [5, 7, 9]
C= A*B # [4, 10, 18]
```

La operación de multiplicación matricial sobre **ndarrays** de 2 dimensiones (matrices), se realiza mediante el operador @:

C= A@B: No es posible, no son arrays 2D.

En ocasiones, tenemos arrays de una dimensión y queremos transformarlos a otra dimensión (por ejemplo, array 1D a array 2D). En este sentido, la función **reshape** permite distribuir la forma del array:

```
A= A.reshape(1, 3) # Cambia el array 1D a array 2D de tamaño 1x3
B= B.reshape(3, 1) # Cambia el array 1D a array 2D de tamaño 3x1
```

Ahora **C=A@B sí es posible**, dado que multiplicamos una matriz de 1x3 por otra de 3x1. Da como resultado una matrix de 1x1.

Además, implementa todas las funciones de la biblioteca estándar de matemáticas **math**, ampliada para arrays: **np.sqrt, np.pow, np.exp, np.sin, np.tan,** etc.

También conviene destacar que es posible agrupar diferentes arrays, haciendo uso de:

- **np.concatenate((a, b))**: Concatena los arrays multidimensionales a, b, supuestos de la misma dimensión.
- **np.hstack((a, b))**: Apila los arrays multidimensionales a, b, horizontalmente (deben tener la misma dimensión en vertical).
- **np.vstack((a, b))**: Apila los arrays multidimensionales a, b, verticalmente (deben tener la misma dimensión en horizontal).
- **np.copy(a)**: Devuelve una copia de a, como un objeto diferente.

Operaciones sobre matrices

Supongamos que tenemos dos arrays **a= np.array([1, 2, 3]), b=np.array([4, 5, 6])**, y dos matrices **A= np.array([[1, 2], [3, 4]]), B= np.array([[5, 6], [7, 8]])**

- **a.dot(b)** devuelve el producto escalar de a*b
- **A@B** devuelve la multiplicación de matrices A*B
- **A.cross(B)** devuelve el producto vectorial de A por B
- **A.T:** Devuelve la traspuesta de A
- **np.linalg.inv(A):** Devuelve la inversa de A
- **np.linalg.det(A):** Devuelve el determinante de A
- **np.linalg.norm(a)**: Devuelve la norma de a

En general, el subpaquete np.linalg contiene operaciones de álgebra lineal (sobre matrices).

Ejemplo: Operaciones básicas de arrays

Ver ficheros Python asociados: **Codigo22.py**

Estadística básica

Supongamos un array A de cualquier dimensión. NumPy incluye algunas herramientas de cálculo estadístico básico:

- **np.mean(A)**: Calcula la media de todos los elementos de A.
- **np.std(A)/np.var(A)**: Calcula la desviación típica/varianza de todos los elementos de A
- **np.sum(A) / np.prod(A)**: Calcula la suma/producto de todos los elementos de A.
- **np.min(A), np.max(A)**: Calcula los valores mínimo/máximo de todos los elementos de A
- **np.argmin(A), np.argmax(A)**: Calcula las coordenadas de la componente que tiene los valores mínimo/máximo de todos los elementos de A
- **np.cumsum(A):** Devuelve un nuevo array, conteniendo la suma acumulada de todos los elementos de A

Ejemplo: Operaciones estadísticas con arrays

Ver ficheros Python asociados: **Codigo23.py**

Constantes y valores universales

Al igual que **math**, NumPy también tiene incluidas algunas constantes matemáticas usuales:

- **np.pi**: El número Pi
- **np.inf, -np.Inf/np.NINF**: Infinito positivo y negativo
- **np.nan**: Valor no definido (equivalente a None)
- **np.e**: El número e

Valores aleatorios

El paquete **np.random** permite generar valores aleatorios procedentes de varias distribuciones de probabilidad, así como valores aleatorios uniformes y permutaciones:

- **np.random.seed(x)**: Establece el valor **x** como semilla para generar números aleatorios (por reproducibilidad de resultados).
- **np.random.rand((tam))**: Devuelve una matriz de tamaño **tam** de números aleatorios uniformes en **U(0,1)**.
- **np.random.randn((tam))** : Devuelve una matriz de tamaño **tam** de números aleatorios de una distribución normal en **N(0,1)**.
- **np.random.randint(low, high, tam)**: Devuelve una matriz de tamaño **tam** de números enteros aleatorios entre **low** y **high-1** (inclusive).
- **Np**.random.randperm(N): Devuelve un array de enteros 1D, con una permutación aleatoria de tamaño N. Ejemplo: **np.random.permutation(3)**-> Una permutación de 3 elementos comenzando desde el 0, por ejemplo [2, 0, 1].

Ejemplo: Valores aleatorios

Ver ficheros Python asociados: **Codigo24.py**

7.La biblioteca Pandas

La biblioteca Pandas de Python se ha convertido en los últimos años en una herramienta indispensable para el tratamiento de datos, principalmente en las áreas de la Ciencia de Datos y de la Inteligencia Artificial. Permite, de forma cómoda, cargar volúmenes de datos desde diferentes formatos y procesarlos, para extraer información relevante que posteriormente se use en la construcción de sistemas de IA y de Machine Learning.

La instalación de la biblioteca **Pandas** se realiza desde la línea de comandos, con el entorno **Anaconda** creado para el curso activo, escribiendo la siguiente orden:

conda install pandas

O bien:

pip install pandas

DataFrames y carga de datos

Uno de los principales tipos de datos que más se usan de Pandas es el **Data Frame**. Un **data frame** es un tipo que permite representar tablas de datos, donde cada columna se denomina ***Series***. Usaremos **Pandas** desde Python importando la biblioteca con un alias, Normalmente, este alias es el siguiente:

import pandas as pd

Los data frames pueden crearse manualmente, o pueden ser cargados desde ficheros externos. La creación manual permite generar una tabla bien por filas, bien por columnas, usando como estructura de datos auxiliar el diccionario de Python. Aunque existen distintos formatos de datos para la carga de información desde fichero, en IA y Ciencia de Datos es común usar el formato de Valores Separados por Comas (**CSV**). Otros formatos permiten leer ficheros Excel, HDF, Pickle, SPSS, o incluso acceder a Bases de Datos con consultas SQL.

Ejemplo: Creación de data frames con Pandas

Ver ficheros Python asociados: **Codigo25.py**

Inspección de datos

Una de las primeras acciones a realizar sobre un conjunto de datos es inspeccionar su contenido. Para ello, Pandas posee diferentes herramientas. Entre ellas se destacan:

- **df.head(n)**: Muestra las **n** primeras filas del dataframe **df**.
- **df.tail(n)**: Muestra las **n** últimas filas del dataframe **df**.
- **df.sample(n)**: Muestra **n** filas aleatorias del dataframe **df**.
- **df.shape**: Tupla conteniendo la forma (filas, columnas) del dataframe **df**.
- **df.info()**: Muestra un resumen de los tipos de datos de cada columnas, y los valores perdidos/NaN que hay en el dataframe **df**.
- **df.describe(n)**: Muestra estadísticos básicos sobre las variables numéricas del dataframe **df** (media, desviación estándar, índices intercuartílicos…).

df.sort_valores('nombre de columna'): Reordena las filas del data frame según la columna especificada. Se puede incluir otro parámetro opcional ***ascending=False***, para que la ordenación sea de mayor a menor, o incluso establecer una lista de nombres de columnas para realizar ordenación por varios criterios.

En esta sección, trabajaremos sobre datasets existentes en internet en la biblioteca **Seaborn** (se estudiará posteriormente para visualización de datos).

Ejemplo: Inspección inicial de data frames con Pandas

Ver ficheros Python asociados: **Codigo26.py**

Indexación y filtrado

La indexación y el filtrado pueden ayudar a centrarnos en un subconjunto particular de los datos en el que estemos interesados. La selección de filas y columnas se puede realizar mediante el acceso a la tabla con el localizador **loc.** No obstante, también es posible establecer condiciones de filtrado con operadores relacionales ==, >, <=, etc.

Otra forma de filtrado, principalmente utilizada en el análisis de datos estratificado, consiste en agrupar los datos por categorías o valores concretos de columnas. En particular, eso se consigue en Pandas mediante la opción **groupby()**.

Ejemplo: Indexación con loc y filtrado

Ver ficheros Python asociados: **Codigo27.py**

Operaciones sobre columnas

Pandas permite también realizar operaciones básicas sobre los datos de una columna, a fin de agregar información:

- **df.columna.sum()**: Suma todos los valores de la columna determinada del dataframe **df**.
- **df.columna.mean()**: Calcula la media de todos los valores de la columna determinada del dataframe **df**.
- **df.columna.max()/df.columna.min()**: Calcula el valor máximo/mínimo de todos los valores de la columna determinada del dataframe **df**.
- **df.columna.cumsum()**: Calcula la suma acumulada de los valores de la columna determinada del dataframe **df**. El resultado es una columna (***Series***).

También es posible añadir nuevas columnas, simplemente definiéndolas en el dataframe como **df['nuevaColumna']= valores**.

Ejemplo: Indexación con loc y filtrado

Ver ficheros Python asociados: **Codigo28.py**

Tratamiento de valores perdidos

Cuando tenemos un conjunto de datos, es habitual tener valores sin rellenar dentro de la tabla. Esto puede deberse a errores de medición, haber descartado datos previamente por considerarlos poco fiables, imposibilidad de recabar toda la información necesaria, etc.. Esto se conoce como **valores perdidos**, y en Pandas se modelan mendiante **NaN** (el equivalente a **None** de Python).

Existen dos estrategias fundamentales para el tratamiento de valores perdidos:

- Descartar filas o columnas que contengan valores perdidos.
- Tratar de aproximar los valores perdidos con los valores conocidos. En esta opción, existen multitud de técnicas, menos y más sofisticadas, para tratar valores perdidos, entre las que se destacan: Rellenar con el último valor previo conocido de la tabla, aproximar el valor mediante regresión o dependencia funcional con otras columnas, usar un valor agregado como la media o la mediana, etc.

Pandas ofrece varias técnicas para tratar valores perdidos, en ambas estrategias. En particular, la primera es la más fácil de aplicar mediante el método **dataframe.dropna()**, que elimina todas las filas del data frame que contengan valores perdidos.

Otras opciones que se incluyen en Pandas, relacionadas con la segunda estrategia, consisten en rellenar los valores perdidos con un valor por defecto (método **fillna(valor)**), y rellenar los valores con los anteriores no nulos en la tabla (método **fillna(method='pad')**) o con los siguientes (método **fillna(method='hfill')**. Otras bibliotecas más especializadas en Machine Learning (por ejemplo, **sk-learn**), incluyen métodos de tratamiento de valores perdidos más sofisticados, como estudiaremos en el módulo próximo.

Ejemplo: Tratamiento de valores perdidos con pandas

Ver ficheros Python asociados: **Codigo29.py**

8. Visualización de datos

La visualización de datos es un área de estudio especialmente relevante en la Ciencia de Datos y en la Inteligencia Artificial, pues nos permite analizar el comportamiento de la información que tenemos sobre un problema a resolver, y también a analizar los resultados de los modelos de IA que generemos para resolverlo. Existen numerosas alternativas para visualización de datos en Python, algunas de ellas específicas para problemas concretos (por ejemplo, geolocalización), pero también encontramos bastantes bibliotecas de carácter genérico que permiten generar diversos tipos de gráficos.

En particular, **Pandas**, como herramienta de Ciencia de Datos básicas, permite visualizar datos, aunque no es la especialización del paquete. Otras bibliotecas como **MatPlotLib** o **Seaborn** son herramientas creadas específicamente para este propósito.

8.1. Algunas bibliotecas de visualización de datos

En esta sección revisamos algunas de las bibliotecas para visualización de datos existentes, a fin de poder exponer distintas alternativas que podamos usar en los proyectos de IA del curso.

Pandas

No es una biblioteca especializada en visualización pero, al tratarse de una herramienta pensada para Ciencia de Datos, dispone de herramientas para generación de gráficos. Con **Pandas** es posibles generar gráficas 2D de forma rápida, lo que ayuda al usuario a estudiar los conjuntos de datos de forma rápida.

Matplotlib

Es una biblioteca de generación de gráficos estáticos en 2D. Su funcionalidad es muy parecida a la que podemos encontrar en otros paquetes orientados a la matemática para ingeniería, como MatLab u Octave. Los usuarios que proceden de este tipo de software tendrán una curva de aprendizaje inferior que el resto, pues las mecánicas de generación de gráficas son muy similares. Además, **Matplotlib** permite generar gráficas con una muy alta definición, lo que hace que sea la herramienta preferida para generación de visualizaciones de datos en el ámbito científico. No obstante, el aspecto visual en términos de atractivo, aunque puede alcanzarse con la biblioteca, tiene como precio un alto incremento en la dificultad de uso.

Seaborn

Es una biblioteca similar, en cuanto a características de uso, a **Matplotlib**. Sin embargo, pone especial esfuerzo en que las gráficas resulten más atractivas y estéticamente modernas. Es una buena alternativa a la anterior.

ggplot

La biblioteca **ggplot** está basada en el sistema de generación de imágenes del lenguaje **R**. Aquellos usuarios que proceden de este lenguaje tendrán una curva de aprendizaje menor que el resto. Con **ggplot** es posible generar una gran cantidad de gráficos, tanto 2D como 3D. Pese a que el abanico de posibilidades de generación de gráficos diferentes es elevado, la calidad visual se puede asemejar a **Matplotlib** a nivel básico.

Pygal

Es una biblioteca que aporta un valor añadido en tanto que permite la generación de gráficos interactivos, permitiendo modificar las figuras y realizar operaciones como traslaciones o zoom sin necesidad de generar la gráfica de nuevo.

Plotly

Es otra biblioteca que permite generar gráficos interactivos. Como valor añadido, permite la generación de un mayor número de gráficas que la competencia habitual, como por ejemplo dendogramas, gráficos 3D o gráficas de contornos.

8.2. Visualización de datos básica con *Matplotlib*

La instalación de la biblioteca **Matplotlib** se realiza desde la línea de comandos, con el entorno **Anaconda** creado para el curso activo, escribiendo la siguiente orden:

conda install matplotlib

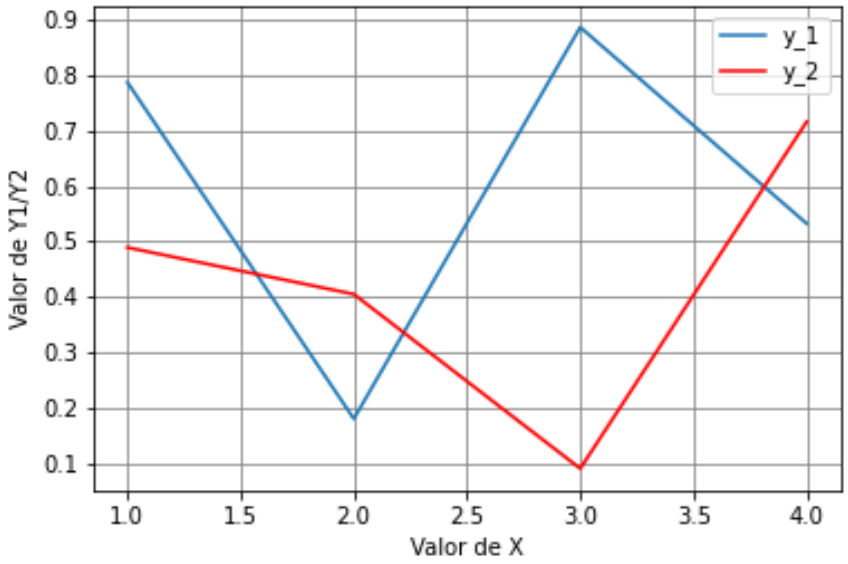
O bien:

pip install matplotlib

La visualización de datos con **Matplotlib** puede ser muy sencilla (generación de gráficas básicas) o muy compleja (personalización de todos los elementos visibles y no visibles de cada figura). En este apartado, para evitar una curva de aprendizaje elevada, nos centraremos en la generación básica de figuras, necesaria para el análisis de datos de los próximos módulos del curso.

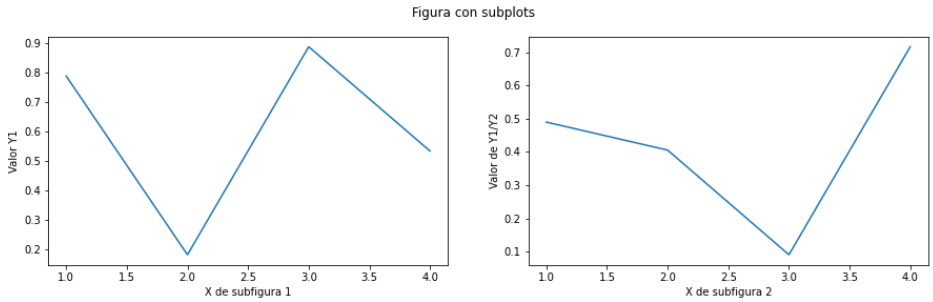
Gráficos de líneas

Es la forma más básica de mostrar datos numéricos. Consiste en dos series de datos, una asociada al eje X y otra asociada al eje Y. Aunque se puede personalizar prácticamente todo, nos centramos en la personalización de las etiquetas de los ejes y, en caso de múltiples gráficas por figura, las leyendas, con opción de incluir rejillas para facilitar la lectura.



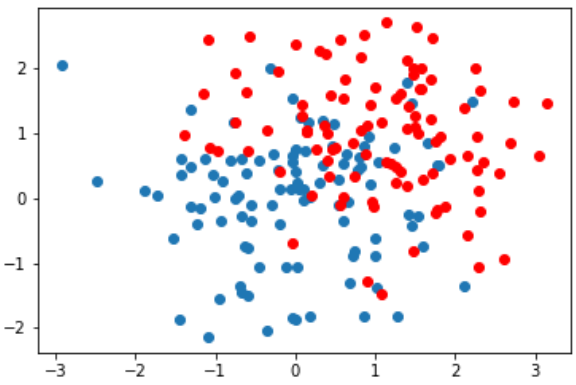
Posibilidad de figuras con múltiples gráficas (subplots)

En ocasiones es recomendable crear una matriz de figuras, para mostrar un tipo de gráfica u otro. Esto se lleva a cabo mediante **subplots**.



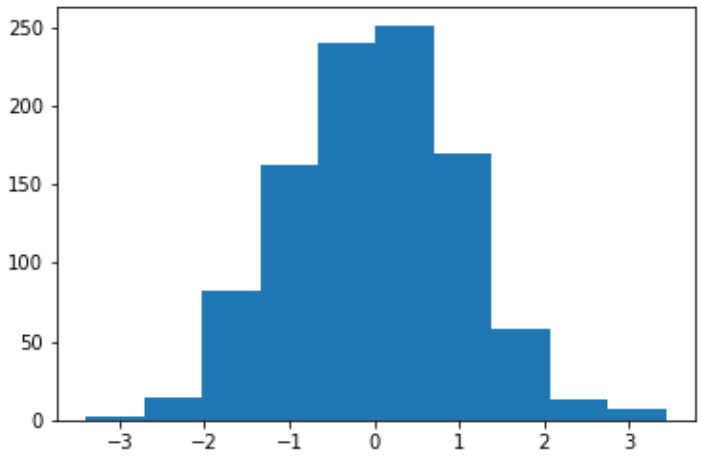
Gráficos de dispersión

Permiten generar figuras que muestren conjuntos de nubes de puntos. Es de especial interés cuando se está tratando con problemas de regresión, o de clasificación.



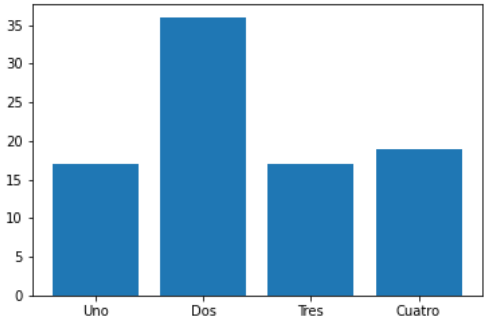
Histogramas

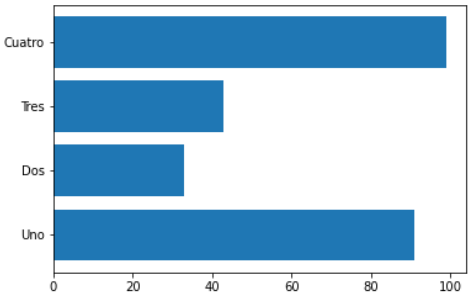
Permiten conocer cómo se distribuyen los datos (en términos de frecuencia) a lo largo de todo el recorrido de una variable.



Gráficos de barras

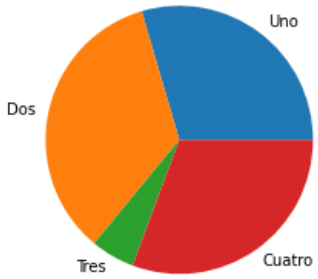
Son útiles para mostrar información numérica sobre variables con valores categóricos. Existen múltiples posibilidades de generación de gráficos de barras, aunque nos centramos (a modo de muestra) en gráficos verticales y horizontales.





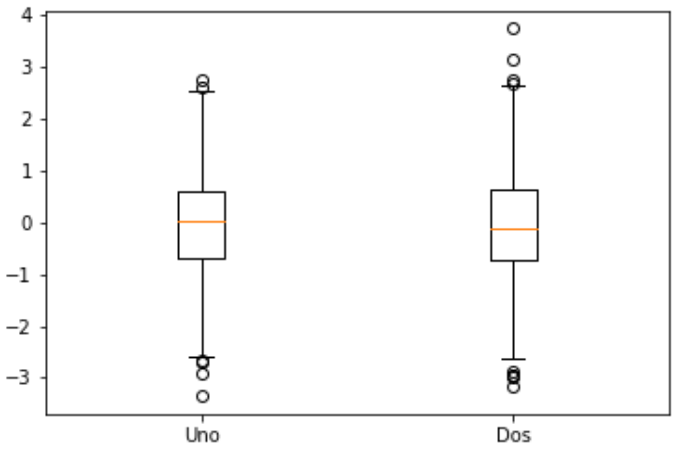
Gráficos de sectores

Es otra forma útil de representar frecuencias de los valores de una variable, o también para ilustrar porcentajes.



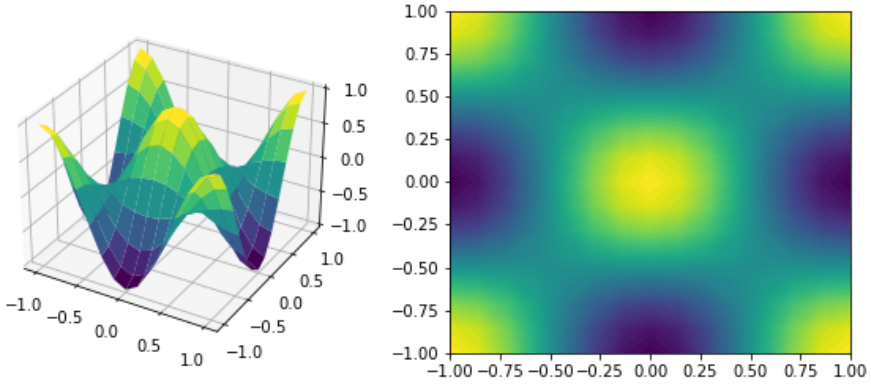
Gráficos de cajas y bigotes

Permiten conocer visualmente información estadística sobre una variable, remarcando los valores medianos y los índices intercuartílicos.



Otras opciones: Contornos, superficies y mapas de calor

Permiten conocer la relación entre 2 variables independientes y otra tercera o dependiente, a través de la ilustración 3D del contorno de la tercera en función de las dos primeras.



Los ejemplos para visualización de las imágenes anteriores se encuentran en el siguiente fichero de código:

Ejemplo: Visualización de datos con Matplotlib

Ver ficheros Python asociados: **Codigo30.py**

8.3. Visualización de datos con *pandas*

La visualización de datos con **pandas** tiene la ventaja de que se encuentra perfectamente integrada con los tipos de datos básicos de la biblioteca, como los **data frames**. Las herramientas de visualización se encuentran dentro del paquete **pandas.plotting**, y están basadas en la biblioteca **Matplotlib**, que deberá estar instalada en el sistema con carácter previo (ver apartado anterior). Los tipos de gráficas a mostrar son similares a los usados en **Matplotlib**. De hecho, el número es un poco más reducido, aunque las características de visualización pueden gestionarse de forma más simple.

Ejemplo: Visualización de datos con pandas

Ver ficheros Python asociados: **Codigo31.py**

9. Otras bibliotecas

Python ha evolucionado enormemente en los últimos tiempos gracias, en mayor medida, a la simplicidad del lenguaje y a la proliferación de multitud de frameworks y bibliotecas específicas que facilitan el desarrollo de diversos tipos de software. A continuación, realizamos una revisión de algunos de los más comunes escenarios donde podemos encontrar Python con gran demanda.

9.1. Aplicaciones web y extracción de información

Python se ha convertido, poco a poco, en un lenguaje a tener muy en cuenta a la hora de desarrollar aplicaciones web, tanto a nivel de codificación back-end como front-end. Contabilizar y estudiar todos los paquetes y entornos web para Python podría llevar a escribir un libro entero, aunque algunos de los frameworks que más se escuchan a nivel profesional hoy día, y que incorporan Python para desarrollo web son:

- **Django**: Es un framework de desarrollo web especializado en desarrollo y prototipado rápido. Aunque tiene múltiples características, su curva de aprendizaje es leve y resulta fácil de usar.
- **Flask**: Centrado en el front-end, permite el desarrollo de interfaces web amigables y de calidad, con bajo conocimiento de otros lenguajes como JavaScript o HTML. Permite realizar webs interactivas de una forma rápida y cómoda, así como integrarse con otros frameworks como generadores de gráficos interactivos, acceso a bases de datos, etc.

- **FastAPI**: Centrado en la generación de interfaces de programación de aplicación (API), centrada en servicios (por ejemplo **RESTful**. Proporciona un entorno intuitivo para la generación de códigos, con gestión de errores y fácil depuración.
- **Falcon**: Centrado en la generación de API mediante servicios **RESTful**. Su característica principal es el bajo peso y óptimos tiempos de acceso a la información, frente a otras alternativas.
- **Scrapy**: Biblioteca centrada en la extracción de información web mediante técnicas de *web scraping*. Es de acceso libre, y tiene una comunidad de desarrollo muy activa.
- **Beautifulsoup**: Se utiliza para cargar y formatear páginas web, y particularmente código HTML. La API de la biblioteca es muy completa y, aunque no está centrada en el prototipado rápido, sí permite al programador personalizar las búsquedas y extracción de información de forma más detallada que otras plataformas.

9.2. Bases de Datos SQL y NoSQL

Con respecto al acceso y manipulación de Bases de Datos, Python se encuentra al mismo nivel que otros lenguajes de programación de propósito general clásicos, tales como Java o C/C++. Existen drivers para Python para la mayoría de gestores de Bases de Datos más utilizadas actualmente a nivel micro y macro, tales como Oracle, MySQL y MariaDB, SQLite, JSON o MongoDB, entre otros. El uso de las APIs para acceso a Bases de Datos tampoco difiere sustancialmente de las de otros lenguajes como Java, siendo herramientas potentes que facilitan el prototipado rápido de aplicaciones software de sistemas de información.

9.3. Minería de datos y Big Data

Como se ha comentado varias veces a lo largo de este documento, Python es uno de los lenguajes más usados en la Ciencia de Datos y, por tanto, dispone de múltiples bibliotecas para Machine Learning y Data Mining. En los últimos años, las herramientas de Data Mining suelen ir ligadas también a aprendizaje automático, aunque a continuación destacamos algunas que se centran más en el análisis de la información, dejando las que se enfocan en el desarrollo de modelos (TensorFlow, Pytorch, etc.) para el siguiente módulo del curso:

- **Scipy:** Biblioteca enfocada al cálculo científico. Dispone de herramientas de minería de datos para regresión, clasificación y clustering.
- **Scikit-Learn**: Un paquete de herramientas basado en **scipy**. Aunque está enfocado más hacia el Machine Learning, dispone de herramientas propias de minería de datos para preprocesamiento de conjuntos de datos, así como herramientas estadísticas para análisis y creación de modelos de regresión, clasificación, o agrupamientos, entre otros.
- **River**: Enfocado en el análisis de flujos de datos, es una de las pocas bibliotecas existentes para análisis de datos masivos procedentes de flujos, como por ejemplo seguimiento de tuits, análisis continuo de flujos eléctricos, etc.
- **PyCaret**: Es un paquete enfocado en el procesamiento de datos orientado a Machine Learning. Su funcionamiento es simple, y gana popularidad por su capacidad para prototipado rápido.

9.4. Computación en la nube

Python también es un lenguaje muyusado para el desarrollo de aplicaciones que tienen relación con la nube. En particular, encontramos las siguientes como más destacadas:

- **Google Cloud Client**: Es un framework que incorpora diferentes bibliotecas para el acceso a los servicios cloud de Google. Entre las más destacadas, encontramos APIs para BigQuery, Cloud Billing, Cloud Dataflow, Cloud Datastore, Cloud Natural Language, etc.
- **Apache Libcloud**: Permite acceso a muchos de los servicios cloud ofertados por múltiples empresas, tales como rackspace, openstack, CloudSigna, CloudFlare, Amazon, etc.
- **CloudBridge**: API que permite integrar diferentes proveedores de servicios Cloud. Gracias a esta biblioteca, es posible gestionar servicios de distintas compañías (google, Microsoft, Amazon, etc.), desde una misma interfaz.
- **EdgeST SDK**: Biblioteca para gestionar información en dispositivos de Edge y Cloud Computing, muy orientada al Internet de las Cosas (IoT).

9.5. Computación cuántica

Python es el principal lenguaje utilizado en Computación Cuántica. Debido a la escasez de acceso a computadores cuánticos reales, diversas compañías han desarrollado simuladores de computadores cuánticos que, a su vez, incorporan una API para acceder a dispositivos cuánticos reales. Entre estas bibliotecas destacamos:

- **Qiskit:** El framework de computación cuántica de IBM. Permite ejecutar programas cuánticos en el PC mediante el uso de distintos simuladores, e incluye también acceso gratuito a sus computadores cuánticos mediante servicios en red. La biblioteca de **Qiskit** está muy bien documentada, y proporciona ejemplos del uso de la computación cuántica en diferentes problemas: finanzas, biocomputación, inteligencia artificial, etc.

- **Pennylane**: Es una biblioteca muy centrada en el aprendizaje automático cuántico. Su uso, al ser más específica, tiene una curva de aprendizaje inferior que la anterior. Además, tiene la posibilidad de creación de modelos híbridos de inteligencia artificial, fusionando las características de **PennyLane** con sistemas como Pytorch o Tensorflow.

- **Strawberry Fields**. Biblioteca de generación de programas de computación cuántica, especializada en dispositivos hardware cuánticos mediante fotónica. Además del acceso a computadores cuánticos reales, proporciona un simulador para ejecutar el código en el PC de sobremesa.

- **Cirq**. Es la apuesta de Google para programación de computadores cuánticos. Al tratarse de esta compañía, tiene un alto soporte y una velocidad de desarrollo muy superior al resto, por lo que la API cambia con frecuencia y es difícil mantener un programa funcional entre versiones. Pese a este inconveniente, es una de las principales bibliotecas de computación cuántica, con una capacidad similar a **Qiskit** con respecto a posibilidades de desarrollo y tutoriales.

- **Tensorflow Quantum**. Es una biblioteca que surge de la unión de **Tensorflow** y de **cirq**, de modo que se puedan crear modelos de aprendizaje automático e inteligencia artificial híbridos (parte clásica y parte cuántica dentro del mismo modelo).

Pese a que la computación cuántica ha surgido con fuerza en los pocos últimos años, se destaca el gran esfuerzo por compañías como Google o IBM en desarrollar APIs que permitan la construcción de programas cuánticos. Cabe destacar, a su vez, la elección del lenguaje Python como primera alternativa para la construcción de estos programas, gracias a la gran popularidad y crecimiento que el lenguaje de programación ha experimentado en las dos últimas décadas.

vuela

PLATAFORMA
DIGITAL DE
ANDALUCÍA

Junta de Andalucía