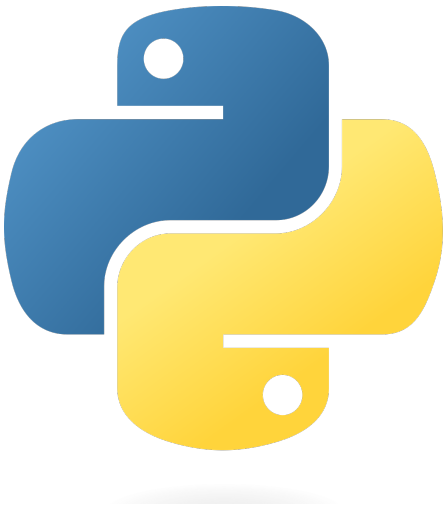
Manual Python

Guía introductoria a python
y sus aplicaciones a la física



Índice

Conceptos básicos de Python

El lenguaje de programador Python fue creado por Guido van Rossum, un científico y programador holandés, a principio de los años 90 del siglo XX. Entre otras cosas, alcanzó relevancia al ser utilizado en el desarrollo del buscador Google, siendo en la actualidad uno de los lenguajes más utilizados. La versión 3 es actualmente la última del lenguaje Python.

Python es un lenguaje de alto nivel, esto implica que su sintaxis es fácilmente comprensible para una persona (aunque requiera formación) y que la escritura y lectura de los programas escritos en él no es demasiado costosa.

Python trabaja en el marco de la programación orientada a objetos. Este es un modelo de programación en el que el diseño de software se organiza alrededor de datos u objetos, en vez de usar funciones y lógica. Se enfoca en los objetos que los programadores necesitan manipular, en lugar de centrarse en la lógica necesaria para esa manipulación

1. Lenguaje básico de Python

1.1. Variables y tipos de datos

Podemos asignar valor a una variable cualquiera simplemente con

```
"nombre  variable" = "valor  variable"
```

Nunca empezar el nombre de una variable por un símbolo o un número.

Existen diferentes tipos de datos:
-Int: número enteros
-Float: números con decimales
-Str(String): cadena de carácteres
-Bool: valores booleanos (verdadero o falso)

Podemos conocer el tipo de dato con el comando *type*("*variable*"). Otros comandos relacionados son:

1. *dir*(), para obtener información de las variables que se encuentran en memoria.

2. *isinstance*("*variable*", *tipo*), para comprobar si una variable es de un tipo determinado.

3. *del*("*variable*"), para eliminar variables de memoria.

También se puede convertir un tipo de variable en otra usando $int()$, $str()$, $float()$

1.2. Usar las variables

Para pedir un valor al usuario y asociarlo a una variable al ejecutar el programa usamos:

$"variable" = input("Introduzca\ \ una\ \ variable:")$

para imprimir una variable en la consola

$print("variable","otra\ \ variable",...)$

También podremos jugar con el valor de las variables numéricas usando los operadores aritméticos más básicos:

+ (Incremento y Suma)
- (Decremento y Resta)
* (Multiplicación)
/ (División) (No se admite división entre 0)
// (División entera)
% (módulo)
** (exponencial)

1.3. Estructuras de control

1.3.1. If

La sentencia if permite determinar si se debe ejecutar un bloque de código o no. Va seguida de una expresión lógica (devuelve true o false). Puede contener alternativas condicionales mediante la palabra reservada elif.

Ejemplo de uso:

```
almendras=15

if almendras >= 10

        print("Tienes muchas almendras")

elif almendras <= 4

        print("Tienes poquitas almendras")

else:
```

```
    print("Tienes almendras")
```

1.3.2. While

La estructura de control while permite repetir un bloque de código mientras se cumple una condición. La condición se indica a continuación de la palabra reservada while y se va a evaluar siempre antes de ejecutar el bloque de código, por lo que dicho bloque puede no ejecutarse nunca.

Ejemplo de uso:

```
i = 1

while i <= 10:

    print(i)

    i = i + 1

print("Ya sabes contar hasta diez")
```

1.3.3. For

El bucle for permite recorrer una colección de elementos, ejecutando un bloque de código por cada uno de ellos.

Ejemplo de uso:

```
for numero in range(5):

        if (numero == 3)

        print("aqui esta el tres")

        continue   ...(o break para detenerlo)

    print(numero)
```

1.3.4. Try

Se utiliza la palabra reservada try para indicar que el código que se va a ejecutar a continuación puede producir errores. Esta detección tiene como objetivo el poder hacer un tratamiento de dichos errores, con el fin de corregirlos o notificarlos.

Se usa tal que:

try:

    código que puede dar error

except:

    código a ejecutar si da el error

1.4. Estructuras de datos

Una estructura de datos es una manera de almacenar y organizar información para permitir su tratamiento de manera eficiente.

Nombre	Sintáxis	Inmutable	Slicing	Duplicados
Lista	[a,b,c,...]	No	Si	Si
Tupla	(a,b,c,...)	Si	Si	Si
Conjunto	{a,b,c,...}	Si	No	No
Diccionario	{a=1,b=2,...}	No	No	No

El slicing es una técnica que permite obtener una sección de una colección de datos a partir de la posición de sus elementos, ya sea esta una cadena de caracteres (colección de caracteres) o una estructura de datos como es la lista o la tupla. El slicing se realiza indicando entre corchetes la posición del elemento de inicio, dos puntos y la posición del elemento final.

Ejemplo:

numeros = ”123456789”
pequeños = numeros[0:3] ——> ”123”
grandes = numeros [6:8] ——> ”789”
todos = numeros[:] ——> ”123456789”

También aplicable a listas con las mismas reglas.
Se pueden coger de forma alterna como ïmpares = numeros[0::2]

1.4.1. Listas

La lista es una relación no inmutable (puede modificarse) de elementos que pueden estar duplicados y se encuentran ordenados. Se compone de una serie de elementos heterogéneos agrupados entre corchetes y separados por comas.

Con el comando "*lista*".*append*("*variable*") puedes meter nuevos valores en la lista.

- *len*("*lista*") vemos el número de elementos de una lista.

- "*variable*" *in* (*not in*)"*lista*" compruebas si el elemento está o no en la lista.

- "*lista*".*remove*("*variable*") eliminamos dicha variable

- "*lista*".*insert*("*posicion*","*variable*") introducimos una variable en la posición numérica dada.

- "*lista*".*reverse*() invertimos el orden de la lista

- "*lista*".*extend*("*otralista*") añade los elementos de otra lista en la esta

- "*lista*".*count*("*variable*") cuenta el número de veces que aparece la variable en la lista

- "*lista*".*index*("*variable*") encuentra la posición de la variable en la lista

1.4.2. Tuplas

La tupla es una relación inmutable (no se puede modificar) de elementos que pueden estar duplicados y se encuentran ordenados. Se compone de una serie de elementos heterogéneos agrupados entre paréntesis y separados por comas.

1.4.3. Conjuntos

El conjunto es una relación inmutable (no se puede modificar) de elementos que no pueden estar duplicados y no se encuentran ordenados. Se compone de una serie de elementos heterogéneos agrupados entre llaves y separados por comas. En los conjuntos no importa el orden, solo saber si un elemento está o no está dentro de él.

1.4.4. Diccionarios

El término diccionario (en inglés dict), en informática, es una relación no inmutable (se puede modificar) de elementos compuestos por pares de clave-valor cuyas claves no pueden estar duplicadas y no se encuentran ordenados. Se compone de una serie de elementos de tipo clave-valor agrupados entre llaves y separados por comas.

2. Módulos, paquetes y funciones

Un módulo es un fichero con extensión .py, que puede contener funciones, clases o variables y cuyo objetivo es facilitar la reutilización del código en otros módulos. Para usarlo solo es necesario tenerlo en una carpeta accesible desde el programa a usar e importarlo.

Por ejemplo, si un módulo denominado calculadora.py dispone de las funciones sumar() y restar(), al importar dicho módulo desde otro programa, este va a poder utilizar ambas funciones.

Una función es un bloque de código reutilizable que puede recibir un conjunto de parámetros (información proporcionada desde el exterior a la función), ejecutar una serie de sentencias y, opcionalmente, devolver un resultado.

Para importar usamos
import "nombre modulo" (con *as "nuevo nombre"* cambiamos su nombre)

Y para llamar a las variables y funciones las usamos
"funcion" = "nombre módulo"."funcion"

Y si queremos importar algo en concreto
from "nombre modulo" import "nombre función"

Para evitar ejecuciones no deseadas de código, se puede incluir una estructura condicional cuyo contenido se ejecute solo en el caso en el que el módulo al que pertenece sea el módulo que se ha ejecutado.

Un paquete en Python es un conjunto de módulos ubicados en una misma carpeta o directorio. Para convertir una carpeta en un paquete hay que incluir en la misma un fichero con el nombre *__init__.py*. De esta manera, Python va a reconocer el contenido de la carpeta como parte de un paquete y va a hacer que sus módulos sean visibles por parte del resto de programas de la aplicación.

Las funciones se formulan con

```
def my_function() :
    print("Hello from a function")
```

También podemos añadir argumentos en paréntesis

```
def suma(num1, num2) :
    suma = num1 + num2
    print('La suma da :', suma)
    returnsuma
```

3. Entradas y salidas

La forma por defecto para introducir información es la función *input*(). Si queremos ocultar esa información como una contraseña, usamos getpass, que importamos con import getpass y luego *getpass.getpass*().

Las comillas por ejemplo dan problemas, por lo general \ sirve para salvar estos errores, en este caso pues poniendo \". Otros como \n, salto de línea, \t para tabular.

También podemos crear f-strings con *print*(*F*"*texto*").

Además podremos usar el método .format tal que

print((.El hermano de 0 se llama 1").format("Lucía","Manuel")

También se pueden usar claves.

3.1. Archivos

Para abrir archivos usamos $fichero = open("datos.txt","w")$ y para cerrar $fichero.close()$. La opción "*w*" es para escribir, "*r*" para leer, "*x*" para creación, "*b*" para binario o " + " para leer y escribir.

Un ejemplo de escritura con el comando with es:

```
with open ("datos.txt","w", encoding = "utf-8") as fichero:
fichero.write("Me gustan las patatas")
fichero.write("\n")
fichero.write("Y tambien  las  tortillas")
```

También podemos escribir el contenido de una lista usando

$fichero.writelines("nombre\ lista")$ Con el comando $fichero.read()$ podemos leer y guardar en una variable str toda la información de un archivo. Introduciendo un número n entre los paréntisis indicamos la cantidad de caracteres a leer. Con $fichero.readlines()$ almacenamos cada linea de texto.

3.2. JSON

El formato JSON (JavaScript Object Notation) es el formato en el que JavaScript (un lenguaje de programación de las páginas web) que representa los objetos,

así como uno de los formatos de intercambio de datos (un lenguaje de representación de información común para los dos extremos de una comunicación) más utilizados.

Un documento JSON está compuesto de objetos y de arrays de objetos. Los objetos están delimitados por llaves y formados por elementos compuestos por pares de clave-valor. Las claves siempre son cadenas de caracteres. Los valores pueden ser otros objetos JSON, arrays de objetos JSON, cadenas de caracteres, números, los valores lógicos true y false y el valor null. Mientras que, un array, por su parte, está delimitado por corchetes y compuesto de una relación de objetos separados por comas.

Los ficheros JSON se almacenan en texto plano y tienen la extensión .json.

Existe un módulo JSON en Python para tratar estos ficheros de forma sencilla.

4. Programación orientada a objetos

La programación orientación a objetos (POO) es uno de los diversos paradigmas de programación que existen. Los paradigmas de programación definen la manera en la que se diseñan y programas las aplicaciones. La programación orientada a objetos pone el foco del diseño del software en los conceptos de clases y objetos, abstracciones de los elementos del mundo real.

Una clase es un componte software que representa a un elemento que participa en un proceso. Una factura, un producto, la cesta de la compra, un alumno... Las clases se componen de atributos y métodos. Los atributos definen el estado de los objetos y los métodos su comportamiento.

En una clase se detallan los atributos que caracteriza a un elemento. Por ejemplo, un alumno matriculado en una carrera universitaria puede caracterizarse mediante los atributos NIA, nombre, apellidos, número de teléfono... Los métodos, por su parte, representan las acciones que puede realizar una clase.

Los objetos, por su parte, son las instancias de las clases. Una clase define cómo ha de ser un elemento de software, mientras que un objeto es un elemento concreto, construido a partir de la definición contenida en la clase.

Para declarar una clase usamos *class "nombre de la clase"* :

Ejemplo de uso:

```
class Coche:

    def __init(self)__ :
```

```
        self.motor_arrancado=False

    def  arrancar(self):

        self.motor_ arrancado=True

    def parar(self):

        self.motor_arrancado=False
```

El concepto de encapsulación en programación orientada a objetos hace referencia a la capacidad de impedir el acceso directo a atributos y a determinados métodos con el objetivo de garantizar la consistencia de los objetos.

Un constructor es un método especial que sirve para crear objetos y que estos sean consistentes. Una clase en Python solo puede tener un único constructor, debe tener el nombre *__init__* y debe recibir al menos un atributo que por convención se denomina self y que hace referencia al propio objeto. Puede incluir más parámetros y estos deben cumplir las reglas de los parámetros de las funciones en Python.

Una instancia es el resultado de convertir un elemento abstracto conocido como clase en otro concreto conocido como objeto. Instanciar es la acción de construir un objeto (instancia) de una clase.

A continuación mostramos un ejemplo mucho más completo

```
class Coche:
def __init__(self, marca, modelo, color,velocidad):
    self.marca = marca
    self.modelo = modelo
    self.color = color
    self.velocidad = 0

def acelerar(self, incremento):
    self.velocidad += incremento

def frenar(self, decrement):
    self.velocidad -= disminución
    if self.velocidad < 0:
        self.velocidad = 0

def display_info(self):
    printf("marca: {self.marca}, Modelo: {self.modelo},
```

```
        Color: {self.color}, velocidad: {self.velocidad} km/h")


# Creamos las instancias (objetos) de la clase coche
coche1 = coche("Opel", "Corsa", "Gris")
coche2 = coche("Suzuki", "Swift", "Rojo")

# Accedemos a los atributos y llamamos a los métodos
coche1.acelerar(50)
coche2.acelerar(70)
coche1.display_info()
coche2.display_info()

coche1.frenar(20)
coche2.frenar(30)
coche1.display_info()
coche2.display_info()
```

A continuación vamos a hablar de otra característica clave de los objetos, la herencia. La herencia es la capacidad que tiene una clase de ampliar a otra, tanto en estado como en funcionalidad. Cuando esto ocurre, se dice que es una clase (llamada entonces clase hija, clase derivada o subclase) hereda de otra (clase base, clase padre o superclase).

Para definir una clase hija simplemente hay que ponerla entre paréntesis, por ejemplo siguiendo con el ejemplo de la clase coche:

class coche(*descapotable*) :

Si la clase padre tiene un constructor, será obligatorio que el hijo también lo posea usando la función super(). También es obligatorio inicializar la clase padre antes que la hija:

```
class coche(deportivo):
        def __init__(self,marca,modelo,color,velocidad):
            super().__init__(marca, modelo, color)
            self.velocidad=velocidad
```

El lenguaje de Python también admite polimorfismo. Se dice que un lenguaje es polimórfico cuando la ejecución de un mismo método se puede realizar sobre

objetos distintos generando respuestas diferentes. Un concepto asociado al polimorfismo es el de la sobreescritura de métodos. Un método programado en la clase base se puede volver a programar en la clase hija, cambiando su comportamiento.

4.1. Sobrecarga

Otro concepto importante de la orientación a objetos es la sobrecarga, que se puede aplicar tanto a operadores como a métodos. En una clase, un método se puede sobrecargar creando diferentes versiones de este con el mismo nombre, diferenciándose unos de otros en los parámetros que reciben.

En Python, la sobrecarga de métodos se puede obtener utilizando parámetros opcionales, ya que no existe la posibilidad de crear distintos métodos con el mismo nombre.

4.2. Atributo de clase

Existe un tipo de atributo especial que se conoce como atributo de clase (frente al atributo de instancia o de objeto convencional). El atributo de clase tiene dicho nombre ya que no caracteriza a cada objeto de manera individual, sino que afecta a todos por igual, existiendo solamente un valor compartido por todos los objetos.

Por ejemplo, si la vida útil máxima de todos los teléfonos móvil que existan en el sistema va a ser siempre la misma, no es necesario tener un atributo por cada instancia, ya que el valor siempre va a ser igual y se va a estar desperdiciando espacio en memoria. Estos atributos se definen a nivel de clase (no dentro del constructor) y no llevan la referencia a self. En su lugar, para referenciarlos dentro de los métodos, se debe indicar el nombre de la clase.

```
class coche:
    vida_util=10
        def __init__(self):
            self._codigo=12345
            self.motor_arrancado=False
        def __modificar_codigo(slef, nuevo_codigo):
            self.__codigo = nuevo_codigo
        def arrancart(self):
            self.motor_arrancado = True
        def parar(self):
            self.motor_arrancado = False
```

```
    @classmethod
    def incrementar_vida_util(cls):
        cls.vida_util+=1
```

4.3. Método estático

Un método estático, prácticamente, no tiene ninguna relación con la clase a la que pertenece, salvo por el hecho de que se encuentra programado dentro de la misma. No recibe como primer parámetro la referencia a la clase a la que pertenece (el parámetro cls), como sí ocurre en los métodos de clase.

Se identifican mediante el decorador @staticmethod y se ejecutan a través del nombre de la clase y no de la referencia a ningún objeto concreto.

```
@staticmethod
def calcular_peso_coche(gravedad,masa):
    peso=gravedad*masa
    return peso

%%Para invocarla%%

Coche.calcular_peso_coche(9.81,2000)
```

4.4. Clases abstractas

Las clases abstractas y las interfaces son dos técnicas relacionadas con la orientación a objetos utilizadas en la resolución de problemas complejos.

Una clase abstracta es una clase en la que al menos uno de sus métodos no está implementado (no ha sido programado aún). Esto obliga a que las clases derivadas incluyan la implementación de dichos métodos. Las clases abstractas no se pueden instanciar, debido a su naturaleza incompleta.

El módulo abc de Python proporciona la infraestructura necesaria para definir clases abstractas, principalmente a través de la clase ABC. Una clase que se quiera definir como abstracta debe heredar de ABC y añadir al menos uno de sus métodos el decorador @abstractmethod que también debe ser importado del módulo abc.

Por su parte, una interfaz es una especie de clase abstracta en la que todos sus métodos son abstractos. Una interfaz no dispone de métodos implementados

sino únicamente de métodos declarados. Es una declaración de comportamiento esperado, una especie de contrato que deben cumplir las clases que implementen la interfaz.

La técnica más sencilla para construir interfaces en Python consiste en definir todos los métodos de una clase abstracta como abstractos para implementarlos en las clases heredadas.

4.5. Clase interna

Una clase interna es aquella que se define dentro de otra. Se utiliza para encapsular atributos y funcionalidad y organizar el código de una forma más eficaz. Las clases internas se suelen definir cuando los objetos de dicha clase solo tienen sentido dentro de la clase contenedora.

Un ejemplo puede ser la clase Idioma que define el nivel de conocimiento de una lengua extranjera. Si dicha clase solo tiene sentido dentro del contexto de un currículum (representado por la clase CV), puede declararse como una clase interna de este.

```
class CV(object):
    def __init__(self,nombre,direccion,idioma,nivel):
        self.nombre = nombre
        self.direccion = direccion
        self.experiencias=[]
        self.idioma = CV.Idioma(idioma,nivel)

    class Idioma():
        def __init__(self,idioma,nivel) -> None:
            self.idioma = idioma
            self.nivel = nivel

miCV=CV("Juanito","Guadalete","Serbio","8")

print("Idioma:",miCV.idioma.idioma)
```

---

5. Expresiones regulares y aspectos avanzados

Una expresión regular es un patrón utilizado para encontrar cadenas de caracteres que cumplan con determinadas condiciones o determinar si las cumplen. O en otras palabras, una expresión regular se basa en una sintaxis formal, precisa y muy versátil, capaz de determinar un patrón carente de ambigüedad. Las expresiones regulares también se conocen por los nombres regex o regexp. Se compone de una combinación de símbolos que definen un patrón. Dicho patrón se puede utilizar para verificar que una cadena de caracteres cumple con determinadas reglas o que contiene una combinación concreta de caracteres.

Algunos ejemplos de expresiones regulares son:

Expresión	Significado
[a-z]	Una letra minúscula.
[a-zA-Z]	Una letra minúscula o mayúscula
[a-zA-Z0-9]	Una letra minúscula o mayúscula o un dígito numérico.
[0-3]	Un dígito numérico entre 0 y 3.
[a-z]+	Una o más letras minúsculas.
[a-z]{3}	Tres letras minúsculas
[^a-z]	Un carácter que no sea una letra minúscula
(diurno|nocturno)	Las palabras diurno o nocturno

Python permite realizar operaciones con expresiones regulares a través del módulo re. Las funciones más importantes del módulo re de Python son las siguientes:

1. *match*: Determina si la expresión regular (RE) coincide con el comienzo de la cadena de caracteres. Devuelve None si no encuentra la cadena.

2. $fullmatch$: Determina si la expresión regular (RE) coincide con la cadena de caracteres completa. Devuelve None si no encuentra la cadena.

3. *search*: Escanea una cadena, buscando cualquier ubicación donde coincida este RE. Devuelve None si no encuentra la cadena.

4. $findall$: Encuentra todas las subcadenas de caracteres donde coincide la RE y las retorna como una lista. Devuelve una lista vacía si no hay ninguna concordancia.

5. $finditer$: Encuentra todas las subcadenas donde la RE coincide y retorna un iterador.

La diferencia entre las funciones match y search es que la primera busca la concordancia al principio de la cadena y search lo hace en cualquier posición.

En el siguiente ejemplo, se diseña un sistema que solicite al usuario un nombre de fichero y que valide que esté formado por uno o más caracteres alfanuméricos y tenga la extensión py.

```
import re

nombre_fichero=input("Introduce un nombre de fichero:")

expresión_regular="[a-zA-Z0-9]+\.py"

if (re.fullmatch(expresion_regular,bombre_fichero)):
    print("El nombre del fichero es correcto")

else

    print("El nombre del fichero es erróneo")
```

5.1. Otros conceptos avanzados

5.1.1. Compresión de listas

La comprensión de listas o list comprehension es una construcción sintáctica para la construcción de listas de manera compacta.

Una fomra básica de escribirla es:

```
nombre_lista=[ "expresion" for "miembro" in "iterable" ]

%% Como ejemplo %%

frutas = ["manzana", "melon", "cereza", "kiwi", "mango"]

nueva_lista = [x for x in frutas if x != "cereza"]
```

5.1.2. Funciones lambda

Una función o expresión lambda es una función potencialmente anónima (sin nombre) de un solo uso. Son una alternativa compacta a las funciones tradicionales. Comienzan por la palabra reservada lambda, seguida de la lista de parámetros y, separados por dos puntos.

Si tenemos por ejemplo una función sencilla que suma dos parámetros como:

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

resultado = suma(15,18)
```

Si lo escribimos como función lambda sería:

```
sumador = lambda a,b: a+b
resultado = sumador(15,18)
```

Uno de los usos habituales de las funciones lamdba consiste en incluirlas como parámetro de la función filter. La función filter recibe como parámetro el nombre de una función y una colección de datos (una lista o una tupla, por ejemplo).

La función se ejecuta por cada elemento de la colección devolviendo True o False en función de la lógica programada en su interior.

Con los elementos a los que la aplicación de la función ha devuelto True (los que pasan el filtro) se obtiene un objeto de la clase filter que habrá que convertir al tipo deseado (normalmente lista o tupla).

Como ejemplo, para filtrar ciertos elementos de una lista, en este ejemplo separamos las temperaturas demasiado altas:

```
temperaturas=(20,47,14,33,41,36,50,11)

temperaturas_calurosas= filter(lambda t : t>40, temperaturas)

temperaturas_calurosas=list(temperaturas_calurosas)
```

5.1.3. Closures

Las closures son funciones anidadas que permiten encapsular funcionalidad. Una de sus características principales es que las funciones internas tienen acceso a las variables de las externas. Se pueden generar diferentes versiones de funciones configuradas para su posterior ejecución.

Lo más sencillo es verlo con un ejemplo, en esta caso de diferentes multiplicadores:

```
def generador(multiplicando):
    def multiplicar(numero):
        return numero*multiplicando
    return multiplicar


duplicador=generador(2)
triplicador=generador(3)

print(duplicador(7))    % Resultado = 14 %
print(triplicador(11))  % Resultado = 33 %
```

5.1.4. Generadores

Un generador es una alternativa eficiente a estructuras de datos como listas y tuplas. La diferencia fundamental es que el generador no almacena todos los datos en memoria, sino que los va generando bajo demanda.

Como funciones, los generadores no retornan los datos con la sentencia return, sino con la sentencia yield. El uso de return provoca necesariamente la finalización de la función, mientras que yield realiza una devolución de resultados, pero continua con la ejecución, pudiendo de esta manera proporcionar los datos en un flujo y no en una única entrega. Veamos un ejemplo de su uso:

```
def obtener_provincias():
    provincias=("Zamora","Alicante","Lugo","Huelva","Murcia",
    "Madrid","Guadalajara")

    for provincia in provincias:
```

```
        yield provincia

for provincia in obtener_provincias():
        print(provincia)
```

5.1.5. Decoradores

Un decorador es una función que permite "envolver" (decorar) a otras funciones. Constan de dos partes, la dedicada a definir y programar el decorador y la referente al uso de este. Vamos a verlo con el siguiente ejemplo:

```
def titulo_libro(funcion):
    def envoltorio(titulo):
        titulo="Titulo:" + titulo.upper()
        funcion(titulo)
    return envoltorio

@titulo_libro
def mostrar_titulo(titulo):
    print(titulo)

mostrar_titulo("Upanishad")


%% Resultado:

    Titulo: UPANISHAD

%%
```

Numpy

Numpy es una librería de Python que permite trabajar con vectores y matrices (arrays) de gran tamañan y multidimensionales, además de muchas funciones matemáticas de alto nivel.

Un array se diferencia de una lista en que esta puede tener distintos tipos de datos dentro de ella, mientras que en los arrays estos tienen que ser homogéneos, consiguiendo así mucha más eficiencia.

Un vector se escribe tal que:

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

Y podemos escribir una matriz (o tensor de orden 2...) como:

```
M = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Para acceder a los elementros de un array usamos

```
print(M[0])          %% Resulta en: [1 2 3 4]  %%
```

Como es de esperar, también tenemos arrays de dimensión tres, un tensor de tercer orden:

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

Para comprobar la dimensión de nuestro array podemos usar:
print(T.ndim)

Tras esta pequeña introducción vamos a entrar en profundidad en el uso de los arrays, que es la base de Numpy.

Todo sobre los arrays

A parte de las formas de crear vectores y tensores que hemos visto anteriormente podemos también por ejemplo crear un array lleno de ceros:

```
ceros = np.zeros(n)
```

Siendo n el número de elementos que queremos. Podemos hacer lo mismo con unos usando np.ones(n). También es posible crear un array vacío, que es más óptimo que uno de ceros, usando np.empty(n).

Podemos crear un array con un rango tal que:

```
vector = np.arange(5)
```

Que nos dará el vector [0,1,2,3,4].

Podemos hacer que vaya de un número a otro dando pasos como:

```
vector = np.arange(2, 9, 2)
```

Que nos dará el vector [2, 4, 6, 8].

También podemos crear vectores cuyos valores estén linealmente espaciados con:

```
vector = np.linspace(0, 10, num=5)
```

Que nos dará el vector [0. , 2.5, 5. , 7.5, 10.].
Este último es muy útil y se usará bastante de aquí en adelante.

Sacar información de un vector

Vamos a ver con un ejemplo a continuación como obtener información básica de un array:

```
v = np.arange(15).reshape(3, 5)

%% Nos genera lo siguiente
v = ([[ 0,  1,  2,  3,  4],
      [ 5,  6,  7,  8,  9],
      [10, 11, 12, 13, 14]]) %%

v.shape          %% ----> (3, 5)  La forma del vector %%

v.ndim           %% ----> 2       La dimension        %%

v.dtype.name     %% ----> 'int64' El tipo de dato     %%

a.itemsize       %% ----> 8  Los bytes de cada elemento %%

a.size           %% ----> 15  El numero de elementos   %%

type(v)          %% ----> <class 'numpy.ndarray'>     %%
```

Modificar y ordenar un vector

Ahora vamos a ver como ordenar y modificar un array de distintas maneras.

Primero para ordenar de forma ascendente, tenemos el siguiente comando, que de tener nuestro vector v desordenado, nos lo dejaría igual que lo tenemos en el ejemplo anterior.

```
np.sort(v)
```

Puedes concatenar varios vectores de forma que si tienes un vector a = [1,2,3] y otro b = [4,5,6], obtengas c = [1,2,3,4,5,6]:

```
c = np.concatenate((a, b))
```

Con la fucion *reshape* puedes cambiar la forma de los arrays sin cambiar sus datos. Por ejemplo con un vector v=[0,1,2,3,4,5]:

```
vr = np.reshape(v, newshape=(3, 2), order='C')
```

Dando de resultado vr=([0 1],[2 3],[4 5]). Con newshape damos la forma que queremos mientras que con order marcamos la forma de leer y escribir, en este caso C es con los índices como en el lenguaje C, mientras que poniendo F sería el orden de Fortran.

Indexar y sacar datos de un vector

Para indexar simplemente usamos la siguiente forma:

```
v = np.array([1,2,3,4,5,6,7,8])

a = v[1:5]
```

Donde obtenemos [2 3 4 5]. Si queremos el dato de una única posición, usamos el número de dicha posición y tambien podemos omitir el segundo número, como sería en el ejemplo anterior poner [5:] y obtendríamos [6,7,8]. Con número negativo sería la posición empezando por la derecha, por lo que [-3:] nos daría [6,7,8] también.

Con

```
a = v[v<5]
```

Guardamos en el vector a todos los valores menores a 5. Tambien podemos hacer ($a >= 5$) para valores igual o mayores a cinco, por ejemplo. También todos los que sean divisibles por un número, en este caso el dos

---

```
a = v[v%2==0]
```

También podemos pedir que se cumplan varias condiciones como:

```
c = a[(a > 3) & (a < 7)]
```

Para juntar varios vectores tambien podemos hacerlo de la siguiente manera. Con los vectores de ejemplo a=([[1,3],[6,2]]) y b=([[5,1],[3,2]]), los podemos juntar verticalmente tal que:

```
vv = np.vstack((a, b))
```

Obteniendo el vector vv=([[1, 3],[6, 2],[5, 1],[3, 2]])

U horizontalmente tal que:

```
vh = np.hstack((a, b))
```

Obteniendo el vector vh=([[1, 3, 5, 1],[6, 2, 3, 2]])

O separarlos de la misma forma usando np.vsplit/np.hsplit.

Operaciones con vectores

Si tenemos tres vectores distintos tal que a=[3,2], b=[1,5] y c=[2,4,6,8], podemos sumar, restar, multiplicar, etc. sencillamente tal que:

---

a1 = a+b

b1 = a–b

Obteniendo a1=[4,7] y b1=[2,-3]

a2 = a∗b

b2 =a /a

Obteniendo a2=[3,10] y b2=[1,1]

También podemos hace una suma sobre todos los datos del vector tal que:

---

c1 = c.**sum**()

Obteniendo c1 = 20

Multiplicar por un escalar multiplicará todos los componentes del vector por ese mismo escalar, como es de esperar.

Podemos también extraer el valor máximo y mínimo de un vector con:

---

cmin = c.**min**()

cmax = c.**max**()

Obteniendo cmin=2 y cmax=8

Si tenemos varias columnas y queremos encontrar el máximo o mínimo de una en concreto, entre los paréntesis especificamos (axis=) y el índice de la columna escogida.

Trabajar con matrices

Una matriz sería simplemente un array con varias columnas y filas, por ejemplo una matriz 3x3:

```
M = np.array([[1, 2, 3 ], [4, 5, 6], [7, 8, 9]])
```

Se puede indexar y operar con matrices de la misma forma que hemos explicado anteriormente para vectores, añadiendo el (axis=) para escoger que eje cojer, si la columna (axis=0) o la fila (axis=1).

Podemos generar vectores o matrices con datos aleatorios con la siguiente función:

```
A = rng.integers(10, size=(2, 3))
```

Obtendremos una matriz 2x3 cuyos valores sean enteros entre el 0 y el 10.

Podemos obtener los valores únicos con la siguiente función, de forma que si tenemos un vector o una matriz con muchos números repetidos, podemos sacar una lista de cada uno de esos números sin repetirlos.

```
u = np.unique(a)
```

También se puede transponer una matriz M fácilmente que guardamos como t por ejemplo con la siguiente función:s

```
t = M.transpose()
```

La función flip nos da la vuelta al vector (o la matriz, entera o escogiendo un axis= , como anteriormente)

```
r = np.flip(v)
```

Podemos transformar una matriz en un vector unidimensional con

```
v = M.flatten()
```

Para guardar y cargar arrays, podemos usar fácilmente las siguientes funciones. Dado un vector v=[1,2,3,4,5]:

```
np.save('nombre_archivo',v)
```

Y lo cargamos de nuevo con otro nombre por ejemplo:

```
v =np.load('nombre_archivo.npy')
```

Podemos hacer lo mismo pero con un archivo de texto:

```
np.savetxt('nombre_archivo.txt',v)
```

Y lo cargamos de nuevo con otro nombre por ejemplo:

```
v =np.loadtxt('nombre_archivo.txt')
```

Para leer archivos de excel, podemos usar:

```
import pandas as pd

x = pd.read_csv('nombre.csv', header=0).values
```

O seleccionar columnas con

```
x = pd.read_csv('nombre.csv', usecols=['Col1', 'Col2']).values
```

Sistemas lineales

A continuación veremos como aplicar los vectores y matrices que hemos visto para resolver sistemas de ecuaciones lineales.

Primero veremos algunas cosas útiles previas sobre matrices. Como por ejemplo obtener el determinante y la traza:

```
v = np.array([[1,2,-2], [4,5,-2], [7,1,3]])

detV=np.linalg.det(v)

tr = np.trace(v)
```

De forma que detV=27 y tr=9

Dado un sistema con solución única, obtener esta es bastante sencillo. Vamos a usar de ejemplo un problema sencillo de EBAU, cuyas ecuaciones resultan ser:

$$a + b + c = 70$$
$$-2a + 5b + 5c = 0$$
$$5a + 4b + 2c = 320$$

Por lo que solo tenemos que escribir:

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

c = np.array([70,0,320])

a = np.linalg.solve(A,c)
```

De forma que a=[50, 15, 5]

Autovalores y autovectores

Dada la siguiente matriz $\begin{pmatrix} 1 & 1 & -2 \\ -1 & 2 & 1 \\ 0 & 1 & -1 \end{pmatrix}$, para obtener sus autovalores y autovectores, aplicamos la siguiente función tal que:

```
A = np.array([[1,1,-2], [-1,2,1], [0,1,-1]])

v, w = np.linalg.eig(A)
```

Donde en v guardamos los autovalores y en w hemos guardado los autovectores

Es importante notar dos cosas importantes. Primero, que los autovectores correspondientes están en la columna, y no la fila de la matriz w. Es decir, que si el primer autovalor es 2, entonces su autovector correspondiente se obtiene con w1=[:,0].

Lo segundo es que los autovectores vendrán normalizados a la longitud unidad, por lo que es normal que el resultado obtenido a mano sea el autovector obtenido por un escalar.

Distribuciones y números aleatorios

En esta sección vamos a cubrir la generación de números aleatorios con Numpy mediante distintas distribuciones estadísticas.

La función más sencilla para generar un array de números aleatorios es:

```
a = np.random.rand(n,m,...)
```

Con n,m,... siendo las dimensiones del array.

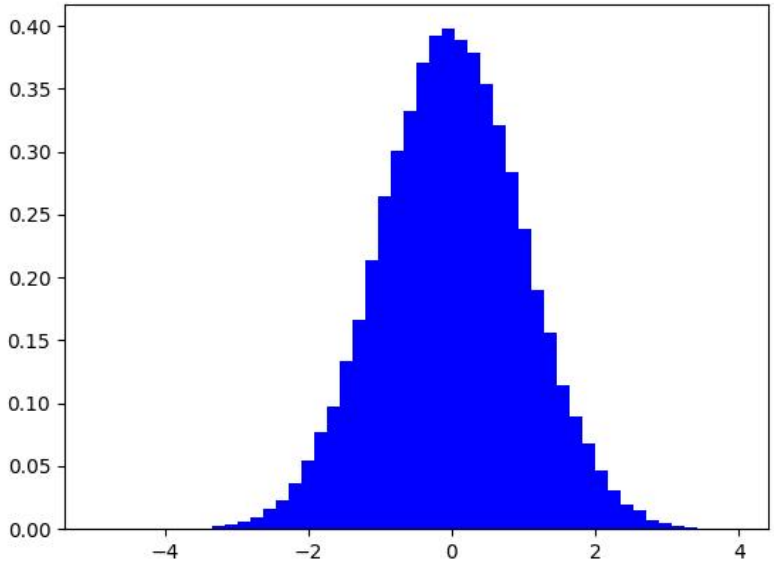
De la misma forma podemos generar un array con números enteros en un intervalo [a,b) tal que:

```
a = np.random.randint(a,b,size=(n,m,...))
```

Distribución normal

La función random.randn() nos da un array de números aleatorios sacados de una distribución normal. Además añadimos la función x para tener en cuenta la media μ (mu) y la raíz de la varianza σ (sigma):

```
x = mu + sigma * np.random.randn(10000)

plt.hist(a, 50, density=True, facecolor='b', alpha=1)
plt.show()
```



Distribución de Poisson

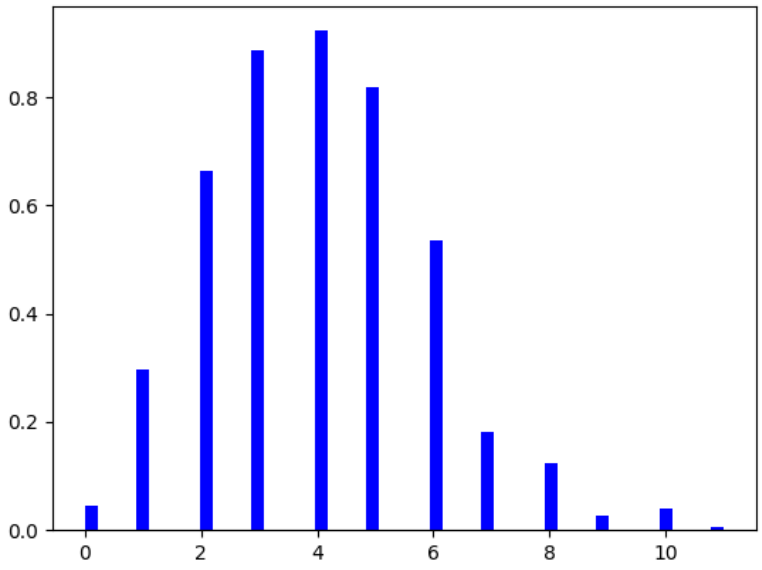
La distribución de Poisson que sigue la forma

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

expresa a partir de una frecuencia de ocurrencia media, la probabilidad de que ocurra un determinado número de eventos durante cierto período de tiempo. En este caso λ es la varianza y la media. k es el número de veces que ocurre un suceso en un intervalo de tiempo, en el cual puede o no ocurrir, de manera independiente a los sucesos anteriores.

```
x = np.random.poisson(k, size=())

plt.hist(a, 50, density=True, facecolor='b', alpha=1)
plt.show()
```

Para un k=4 y un tamaño de 1000 para el array, obtenemos la distribución:



Distribución binomial

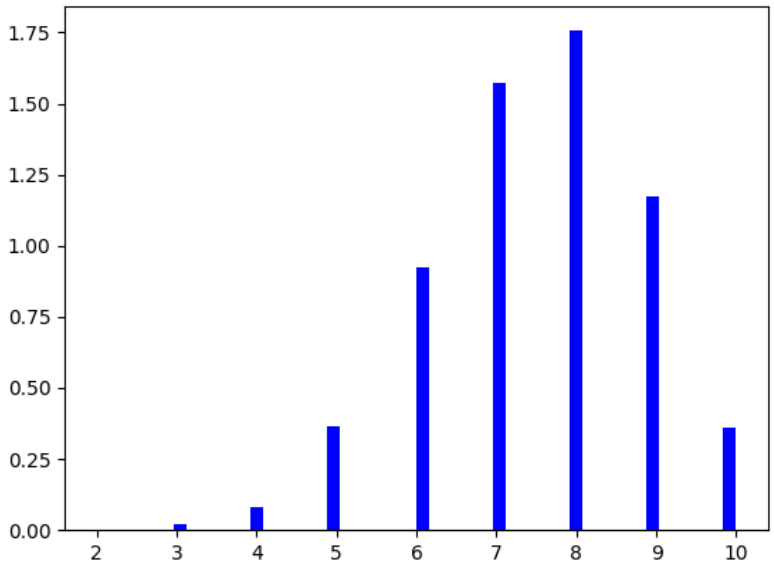
Esta distribución que viene dada por la expresión

$$f(k,n,p) = \frac{n!}{k!(n-k)!}p^k(1-p)^{n-k}$$

donde n es el número de iteraciones de la experiencia, k es el número de sucesos deseados y p es la probabilidad de acertar el suceso deseado. Un ejemplo sencillo es el de una moneda trucada, donde la probabilidad de obtener cara sea 0.75, con la cual queremos acertar al menos 8 veces en 10 tiradas. Entonces p=0.75, $k => 8$ y n=10

```
a = np.random.binomial(10,0.75,10000)

plt.hist(a, 50, density=True, facecolor='b', alpha=1)
plt.show()

ac = sum(a >= 8)/10000
```

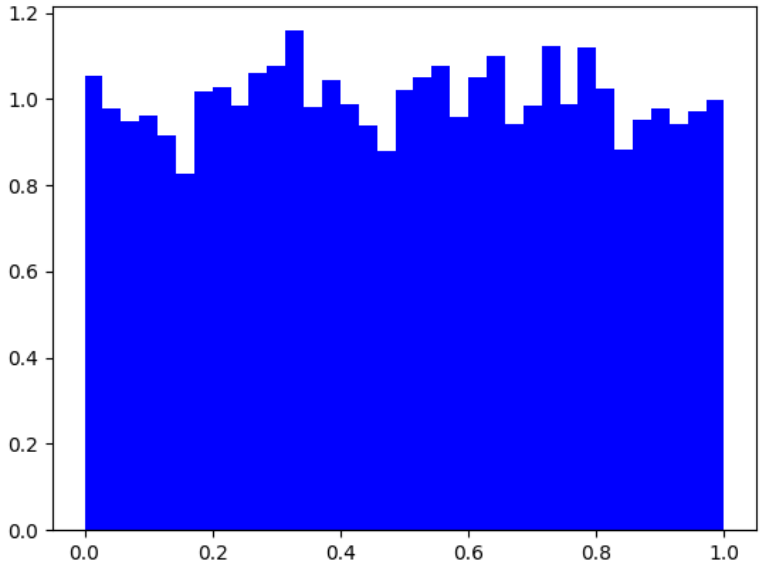
Con ac obtenemos de nuestro modelo la probabilidad de sacar cara 8, 9 o 10 veces al tirar la moneda 10 veces, que en este caso es 0.514.



Distribución uniforme

La función de esta distribución tomará valores en un intervalo [a,b) que en el ejemplo cogeremos como [0,1) y a continuación el size() del array.

```
a = np.random.uniform(0,1,10000)

plt.hist(a, 35, density=True, facecolor='b', alpha=1)
plt.show()
```



Gráficas con Matplotlib

Matplotlib es una librería open source de Python que nos permite elaborar gráficas de todo tipo así como animaciones.

Diagrámas de dispersión

Vamos a explicar lo básico para hacer una gráfica de dispersión, en la que representamos datos como puntos inconexos. Tomaremos de ejemplo los datos medidos del coeficiente de reflexión de una línea de transmisión según variamos la frecuencia de la señal que viaja por ella.

```
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
```

[Escribimos los datos como arrays, donde f
correspondera a x y Gamma a y. También podríamos
tomar los datos desde excel con Pandas según
explicamos en el capítulo anterior]

```
f = np.array([7, 7.7, 7.81, 9.06, 9.89, 11.04, 12])
Gamma = np.array([0.081, 0.103, 0.099, 0.051, 0.006,
0.064, 0.132])


plt.figure(figsize=(7,6))
mpl.pyplot.scatter(f, Gamma, s=35, marker='o',c='r',
linewidths=1.2 ,edgecolors='k', label='Datos' )
plt.xlabel('Frecuencia (MHz)')
plt.ylabel('Gamma')
plt.legend(loc='upper left',fontsize=10)
plt.ylim([0,0.14])
plt.xlim([6.5,12.5])
plt.title('Coeficiente de reflexion frente a la frecuencia')

plt.show()
```

Vamos a explicar por orden cada una de las funciones utilizadas:

− > plt.figure nos permite ajustar el tamaño de la ventana en la que se mostrará la gráfica.

− > mpl.pyplot.scatter(x, y, size=(), m = ”,c = ”, ...) será lo que nos dará la dispersión. Toma primero los arrays como x e y, después el tamaño de los marcadores, luego la forma que puede ser ’o’, ’.’, ’*’ por ejemplo. A continuación tenemos el tamaño de los bordes y su color respectivamente. Por último le damos un nombre a los datos, útil para cuando hay varios datos.

− > plt.x/ylabel() nos permite darle el nombre a los ejes.

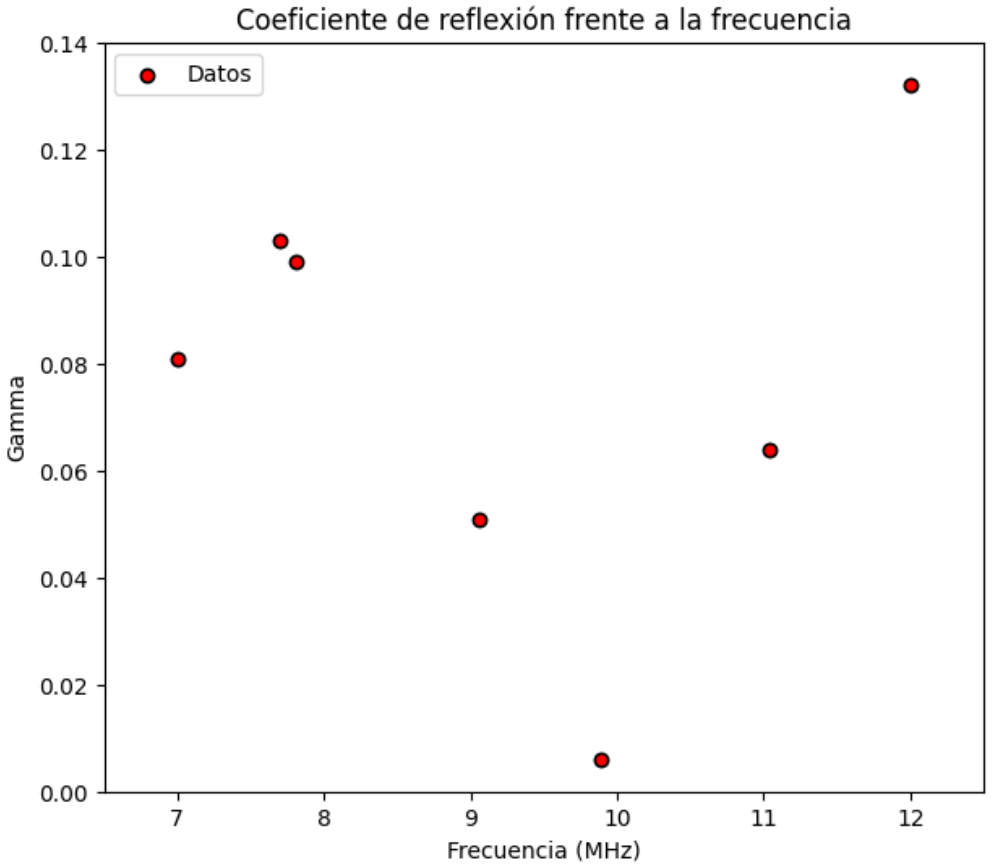
− > plt.legend() genera la leyenda. Con los dos argumentos seleccionamos su posición y tamaño.

− > plt.x/ylim delimita los ejes de la gráfica.

− > plt.title() genera el título de la figura.

− > plt.show() para finalmente mostrar la figura.

Por lo tanto, el código nos producirá la siguiente gráfica:



Histogramas

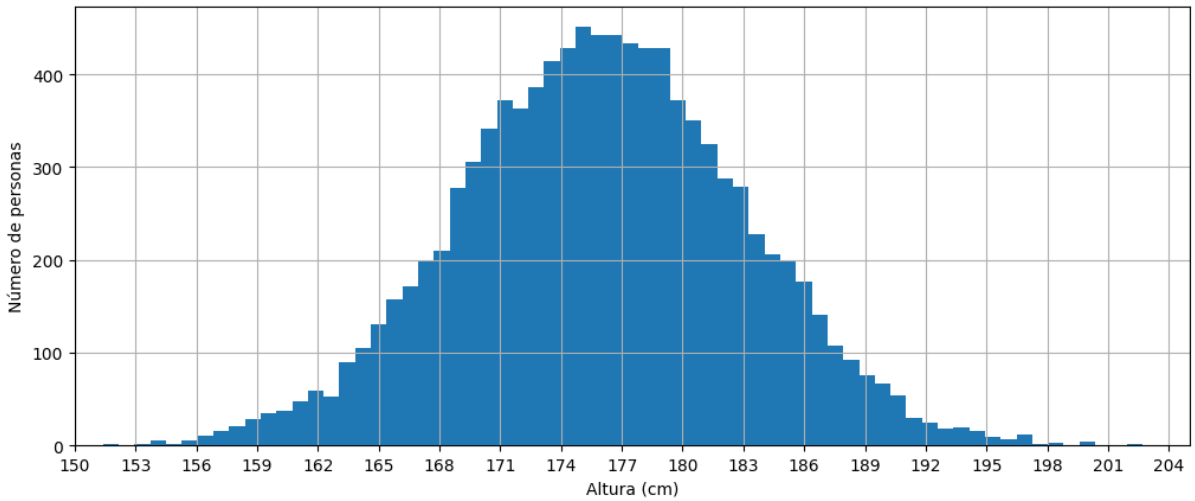
Para ver como hacer histogramas tomaremos de ejemplo una distribución aleatoria de alturas en un grupo de jóvenes en España (10.000 personas), considerando que la media actual ronda los 176 cm.

```
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt

x = 176 + 7*np.random.randn(10000)

plt.hist(x,bins=70)
plt.xlim(150,205)
plt.xticks(np.arange(150, 205, 3))
plt.xlabel(...)
plt.grid()
plt.show()
```

− > plt.hist() genera el histograma con el array (x) de datos dado. El argumento bins nos da la cantidad de divisiones que tendrá el histograma.

− > plt.xticks() nos permite seleccionar el número de divisiones de los ejes, en este caso el x.

− > plt.grid() muestra la rejilla de fondo.



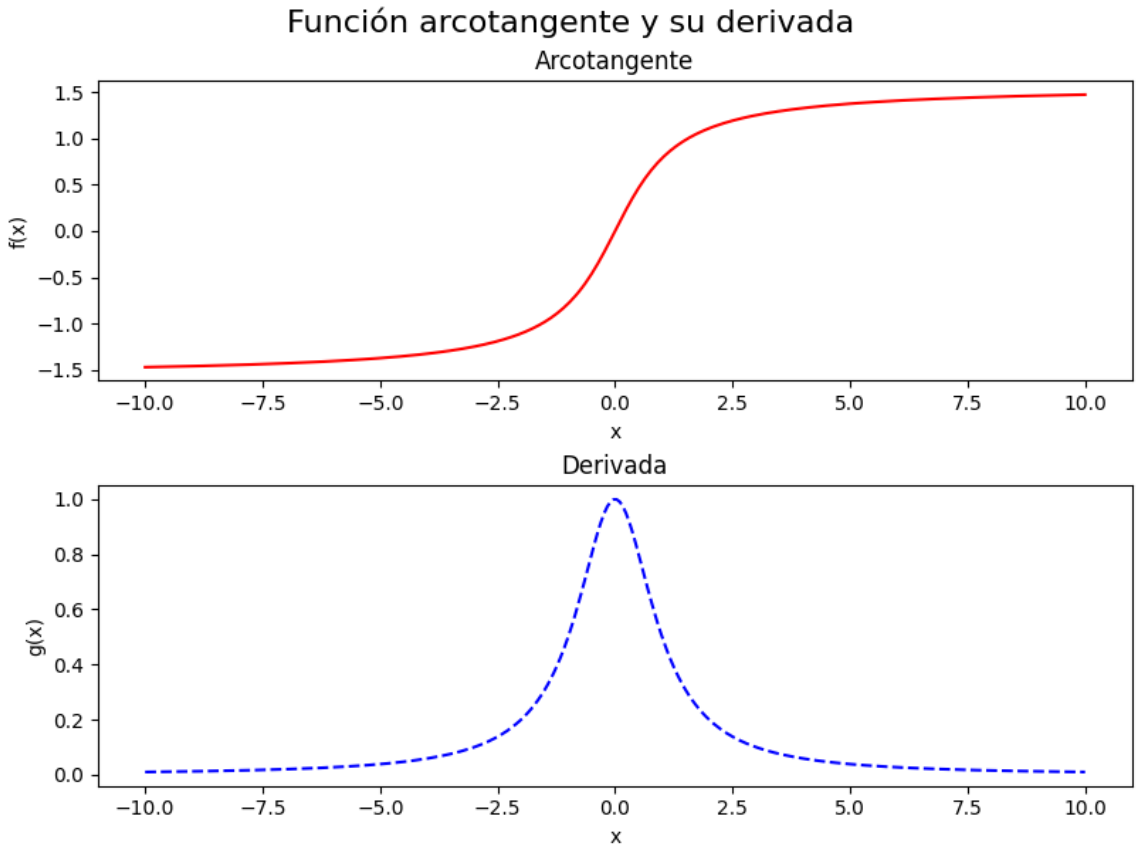
Gráficas de funciones y subplots

Vamos a ver varias cosas en un mismo ejemplo. Primero como usar una función en vez de una serie de datos y como mostrar varias gráficas en una sola figura. Vamos a tomar la función arcotangente y vamos a representar también su derivada. Para ello solo tenemos que definir las funciones, crear un array con np.linspace().

```
def f(x):
    return np.arctan(x)

def g(x):
    return 1/(x**2+1)

x = np.linspace(-10, 10, 300)

fig, (ax1, ax3) = plt.subplots(2, figsize=(8, 6),
layout='constrained')

ax1.plot(x, f(x), '-', c='r')
ax3.plot(x, g(x),'--', c='b')

ax1.set_title('Arcotangente')
ax3.set_title('Derivada')
ax1.set_ylabel('f(x)')
ax1.set_xlabel('x')
ax3.set_ylabel('g(x)')
ax3.set_xlabel('x')
fig.suptitle('Funcion arcotangente y su derivada',
fontsize=16)

plt.show()
```

La clave para representar varias funciones en una misma figura es usar plt.subplots(), donde primero definimos ambas con ax1 y ax3 como ejemplo. Si cambiamos el 2 por 1,2; tendremos las gráficas una al lado de la otra. El resto de código es para respresentarlas y poner títulos y ejes.

Con el código anterior obtenemos pues:



Gráficas de contorno

A continuación vamos a ver como representar gráficas de contorno, en las que tenemos ademas de x e y, el valor z. Es tan sencillo como crear un array con la longitud que queramos, que será el area xy que cubrirá nuestro contorno.

```
h=np.linspace(-1,1,100)

x,y=np.meshgrid(h,h)

z=x**2-y**2

plt.contourf(x,y,z, levels=100)
plt.colorbar(label='Valor de z')
plt.show()
```

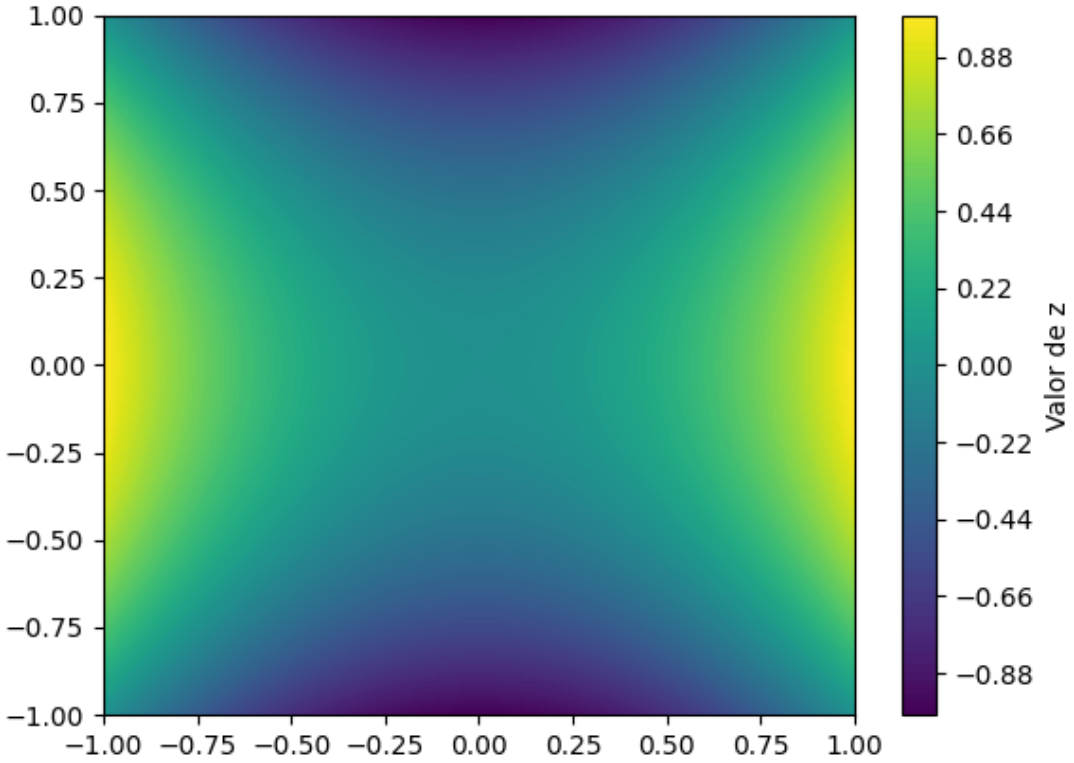
− > x,y=np.meshgrid(h,h) nos crea el contorno según los arrays dados, en este caso son iguales por lo que tenemos un array cuadrado.

− > A continuación definimos el valor que tendrá z en función de x e y.

− > plt.contourf(x,y,z, levels=100) nos crea la representación gráfica. Con levels escogemos la definición de color. Con 100 parece continuo, con menos se vería discreto.

− > plt.colorbar(label='Valor de z') muestra la barra de referencia para los colores y le da un título.

Por lo que con el código anterior obtenemos la siguiente gráfica:



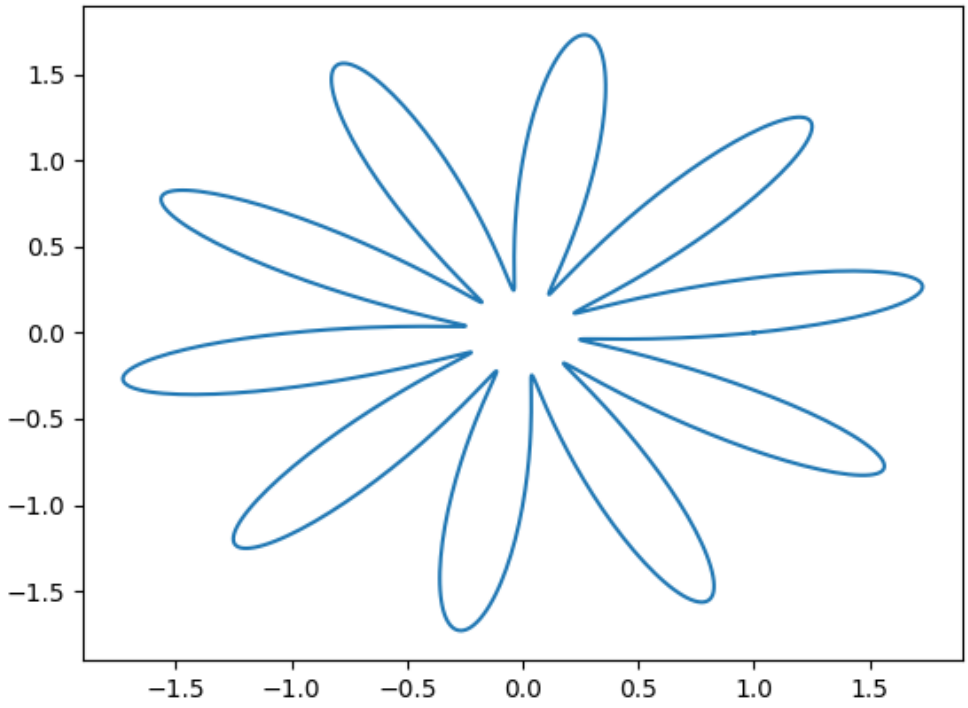
Funciones polares

Para representar funciones en polares no hay que usar ninguna función nueva, solo requiere pasar la función que tengamos en polares a sus cordenadas cartesianas y añadirlo al plt.plot().

```
theta = np.linspace(0, 2*np.pi, 1000)
r = 1 + 3/4*np.sin(10*theta)
x=r*np.cos(theta)
y=r*np.sin(theta)
plt.plot(x,y)
plt.show()
```

[Como añadido, A nos da el área dentro de la curva y L la longitud de arco]

```
A= 0.5*sum(r**2) * (theta[1]-theta[0])
L= sum(np.sqrt(r**2 + np.gradient(r,theta)**2)
* (theta[1]-theta[0]))
```



Animaciones

Matplotlib nos permite también realizar animaciones a partir de funciones. Para el ejemplo vamos a animar un seno desplazándose a medida que pasa el tiempo.

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation
from matplotlib.animation import PillowWriter
```

Primero importamos todo lo necesario. A continuación definimos la función que queremos animar y el array sobre el que se evaluará.

```
def f(x,t):
    return np.sin(x-3*t)

x=np.linspace(0,10*np.pi,1000)
```

Posteriormente definimos la figura, con una línea vacia ln1 y el timetext que nos colocará el contador de tiempo en la figura, que de nuevo dejamos vacío para luego introducir los parámetros. 0.65 y 0.95 corresponden a la posición del contador y el resto el color y borde de la caja. También definimos los límites de la gráfica.

```
fig, ax = plt.subplots(1,1,figsize=(8,4))
ln1,=plt.plot([],[])
time_text = ax.text(0.65, 0.95, '', fontsize=15,
    transform=ax.transAxes, bbox=dict(facecolor='white',
    edgecolor='black'))
ax.set_xlim(0,10*np.pi)
ax.set_ylim(-1.5,1.5)
```

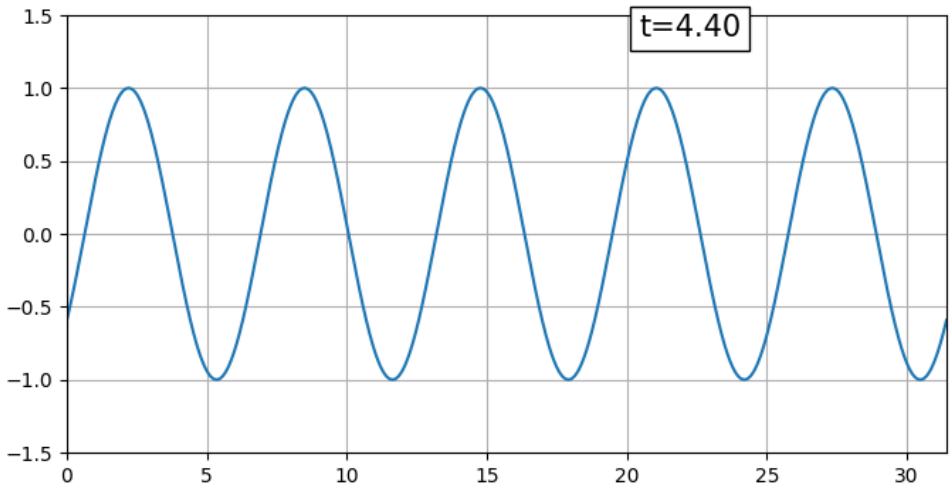
La función animate toma el número del frame i y lo anima paso a paso. Ahora es cuando introducimos los datos a la línea y el contador de tiempo. A ln1 metemos el valor de x y la función f(x,t), donde t vendrá dado por el tiempo y el número de frames, de modo que con i/50 tenemos 50 frames por segundo.

```
def animate(i):
    ln1.set_data(x,f(x,1/50*i))
    time_text.set_text('t={:.2f}'.format(i/50))
```

```
ani = animation.FuncAnimation(fig,animate,frames=240,
    interval=50)
ani.save('ani.gif',writer='pillow', fps=50, dpi=100)
```

Por último generamos la animación introduciendo en animation.FuncAnimation la figura, la función animate, los frames y el intervalo. Usamos ani.save para guardar la animación como un Gif. Importante que coincidan los fps con los introducidos anteriormente con i/50.

No voy a poner un GIF en este documento así que os tendréis que creer que la siguiente figura se mueve. Igualmente copiando el código superior debería poder ejecutarse sin problema.



Scipy

SciPy es una biblioteca libre y de código abierto para Python. Contiene módulos para optimización, álgebra lineal, integración, interpolación, funciones especiales, FFT, procesamiento de señales y de imagen, resolución de ODEs y otras tareas para la ciencia e ingeniería. Se basa en el objeto de matriz NumPy y es parte del conjunto NumPy, que incluye herramientas como Matplotlib, Pandas y SymPy.

Análisis matemático

Optimización

Para los problemas de optimización buscamos los valores de una variable que corresponden a un mínimo local de la función dependiente de esta variable. Vamos a ver el caso más sencillo de una función unidimensional ($f(x) = x^2 - x + 4$):

```
import scipy as sp
from scipy.optimize import minimize

def f(x):
    return x**2-x+4

res = minimize(f,2)

print('The minimal value is on x =', res.x[0])
print()

print(res)
```

No tenemos más que definir la función y aplicar la función minimize, introduciendo la función y una estimación de donde puede estar el mínimo, en este caso 2. Dentro de la variable res se guarda varias partes de la información de la optimización relevantes, como el valor alcanzado por la función en el mínimo y el valor de x en esta.

A continuación vamos a buscar un mínimo de una función en dos dimensiones $f(x,y) = (x-1)^2 + (y-2{,}5)^2$ con unas ciertas condiciones de contorno

$$x > 0, \quad y > 0, \quad x - 2y + 2 > 0, \quad -x - 2y + 6 > 0, \quad -x + 2y + 2 > 0$$

```
g = lambda x: (x[0]-1)**2 + (x[1]-2.5)**2

cons = ({'type': 'ineq', 'fun': lambda x:x[0]-2*x[1]+2},
{'type': 'ineq', 'fun': lambda x:-x[0]-2*x[1]+6},
{'type': 'ineq', 'fun': lambda x:-x[0]+2*x[1]+2})

bnds = ((0, None), (0, None))

res2 = minimize(g, (2,0) , bounds=bnds, constraints=cons)
print(res2)
```

Primero hemos usado la función lambda (5.1.2) para definir la expresión del enunciado, con x como un vector con las dos dimensiones. Luego en la variable cons definimos las condiciones de contorno como una tupla de diccionarios (1.4). Se selecciona el tipo de condición, en este caso ineq para la desigualdad y definimos la función de nuevo con lambda.

Las otras condiciones son más sencillas, solo tenemos que definir una tupla de tuplas, con las condiciones de mínimo y máximo. En este caso los mínimos son 0 pero no tenemos máximos por eso ponemos None.

Por último, introducimos todo en la función minimize y se guarda en res2.

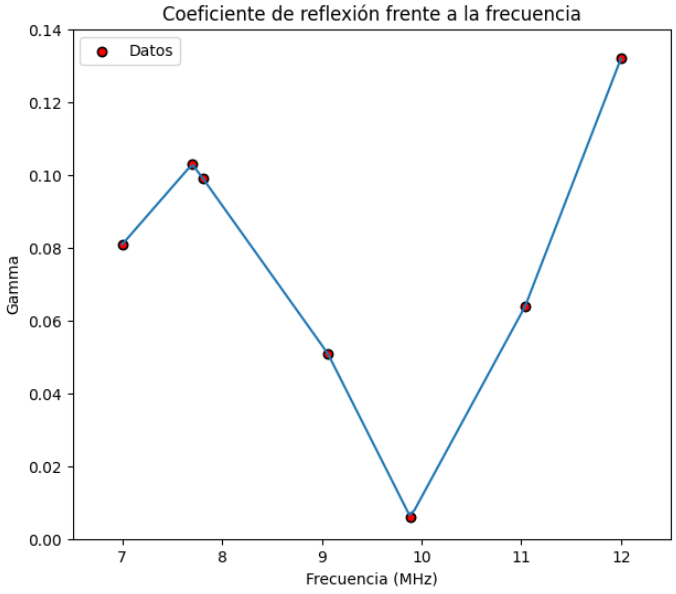
Interpolación

Para ver las distintas formas de interpolar vamos a tomar el mismo ejemplo que usamos para la gráfica de dispersión. Primero vamos a ver algo no demasiado útil pero sirve de ejemplo, una interpolación lineal entre los puntos. Por brevedad recorto las partes que son exactamente iguales.

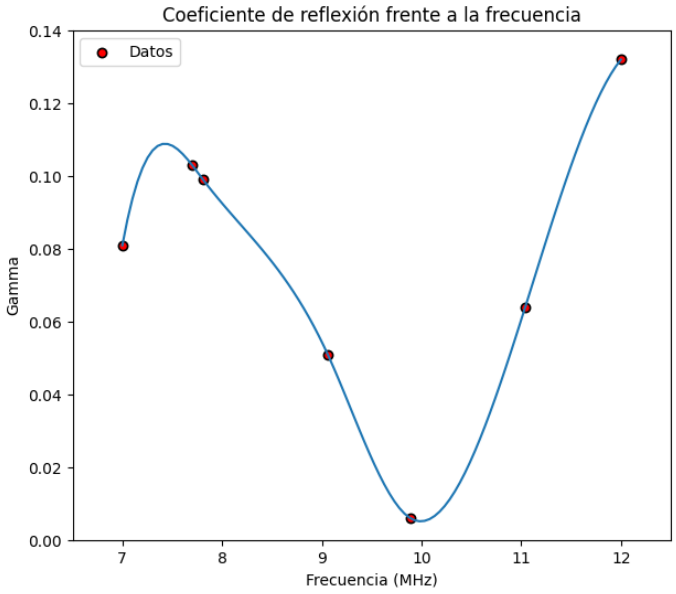
```
import scipy as sp
from scipy.interpolate import  interp1d

f = np.array([7,  ...])
Gamma = np.array([0.081,  ...])

int=interp1d(f,Gamma, kind = 'linear')

x_dense = np.linspace(7,12,100)
y_dense = int(x_dense)

plt.figure(figsize=(7,6))
(...)
plt.title('Coeficiente...')

plt.plot(x_dense, y_dense)

plt.show()
```

Solo tenemos que añadir los arrays de los datos en la función interp1d y escoger el tipo de interpolación, en este caso lineal. A continuación creamos los arrays de densidad, donde x será simplemente el intervalo que abarcan los datos y el y será ese intervalo evaluado en la función int().

Obtenemos la siguiente figura:



Sin embargo, si cambiamos el kind de 'linear' a 'cubic' obtenemos algo más razonable:



Ajuste de datos

A diferencia de la interpolación, en esta caso queremos ajustar nuestros datos a una función dada para extraer algun tipo de información del ajuste. Para ver un ejemplo práctico de ajuste linear, vamos a tomar el caso de una barra de metal que se dilata debido a la temperatura. La expresión del aumento de longitud en función de la temperatura se puede con la siguiente expresión:

$$\Delta L = \bar{\alpha}_L \cdot L_0 \cdot \Delta T$$

Por lo que si queremos obtener el coeficiente de dilatación lineal, solo tenemos que conocer L_0 y obtener una regresión lineal donde $y = \Delta L$ y $x = \Delta T$.

```
from scipy.optimize import curve_fit

x = np.array([7.7, 15.7, 24.0, 30.8, 37.3])
y = np.array([0.11, 0.22, 0.33, 0.42, 0.52])

def func(x,a,b):
    return a*x + b

popt, pcov = curve_fit(func, x, y, (0.01,0.001))

x_fit = np.linspace(7.7,37.3,100)
y_fit = func(x_fit,popt[0],popt[1])

plt.figure(figsize=(7,6))
mpl.pyplot.scatter(x, y, s=45, marker='^',c='orange',
linewidths=0.1 ,edgecolors='k', label='Datos' )
plt.xlabel('Temperatura (C)')
plt.ylabel('Long (mm)')
plt.legend(loc='upper left',fontsize=10)
plt.ylim([0,0.6])
plt.xlim([6.5,40])
plt.title('Dilatacion de un metal')
plt.text(13.4, 0.525, f'y= {popt[0]:.4f}x + {popt[1]:.4f}', size
         ha="center", va="center",
         bbox=dict(boxstyle="round",
                   ec=(1., 0.5, 0.5),
                   fc=(1., 0.8, 0.8),
```

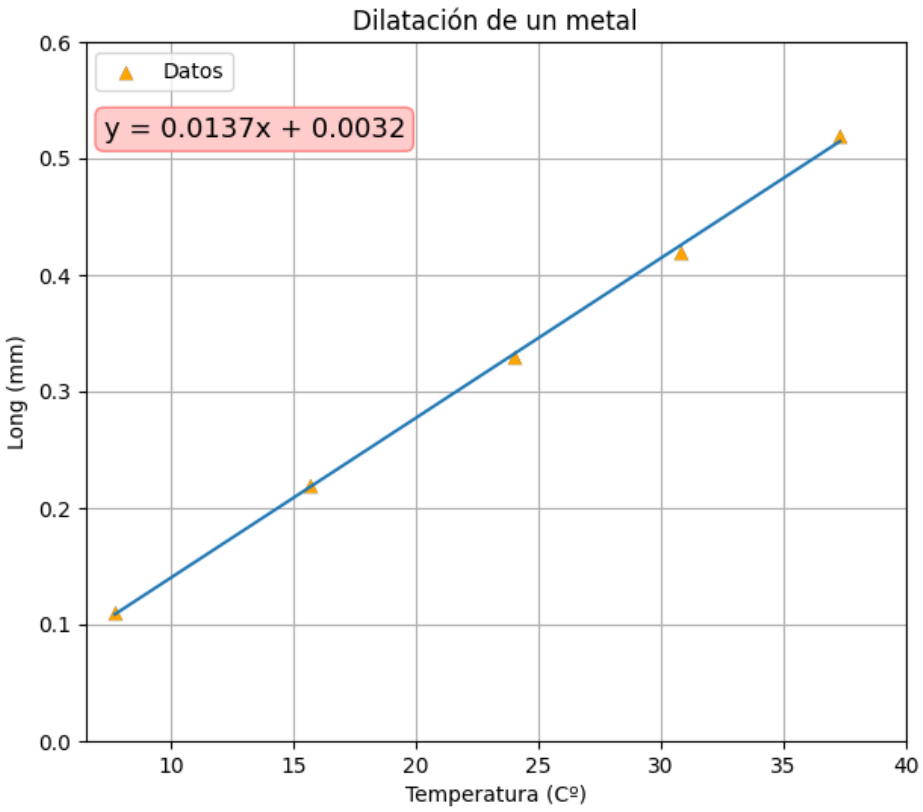
```
                    )
            )

plt.plot(x_fit, y_fit)

plt.grid()
plt.show()
```

Primero hemos introducido los datos experimentales y definido la función lineal para la regresión donde $a = \bar{\alpha}_L \cdot L_0$. A continuación añadimos el método importado, en popt será guardado el valor de a. A este método le damos la función, los datos y una estimación aproximada de lo que podría valer a.

Para poder representar la recta de regresión definimos lo valores de x e y fit.

Por último todos los cosméticos de la figura incluyendo una caja de texto donde aparece la ecuación de la interpolación.

Obtenemos pues la siguiente figura:



Teniendo ya el valor de a, solo nos quedaría dividirlo por L_0 y ya tendríamos el coeficiente de dilatación lineal del metal, en este caso aluminio.

Derivadas

Primero vamos a ver un ejemplo de derivación de una función unidimensional, en este caso una bastante fea como

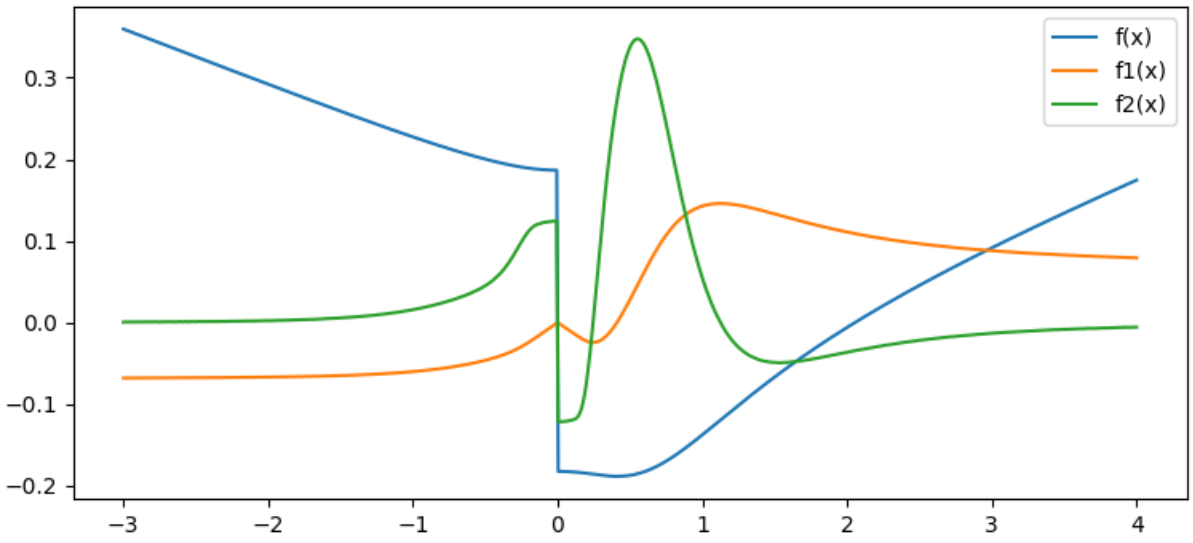
$$f(x) = \frac{x \cdot \cos(e^{\tanh \frac{1}{x}})}{5 \arctan x}$$

y queremos obtener $f'(1{,}0)$ y $f''(2{,}0)$:

```
from scipy.misc import derivative
def f(x) :
    return x*np.cos(np.exp(np.tanh(1/x)))/(5*np.arctan(x))
der = derivative(f, 1.0, dx=1e-6, n=1)
der2 = derivative(f, 2.0, dx=1e-6, n=2)

x=np.linspace(-3,4,700)

plt.figure(figsize=(9,4))
plt.plot(x,f(x), label='f(x)')
plt.plot(x,derivative(f, x, dx=1e-6, n=1), label='f1(x)')
plt.plot(x,derivative(f, x, dx=1e-6, n=2), label='f2(x)')
plt.legend(loc='upper right',fontsize=10)

print(der)
print(der2)

plt.show()
```

Solo tenemos que definir la función e introducirla en derivative() junto con el punto del cual obtener la derivada, el espaciado con dx y el orden de la derivada.

Obtenemos la siguiente figura:



Integrales

Vamos a empezar con una integral unidimensional:

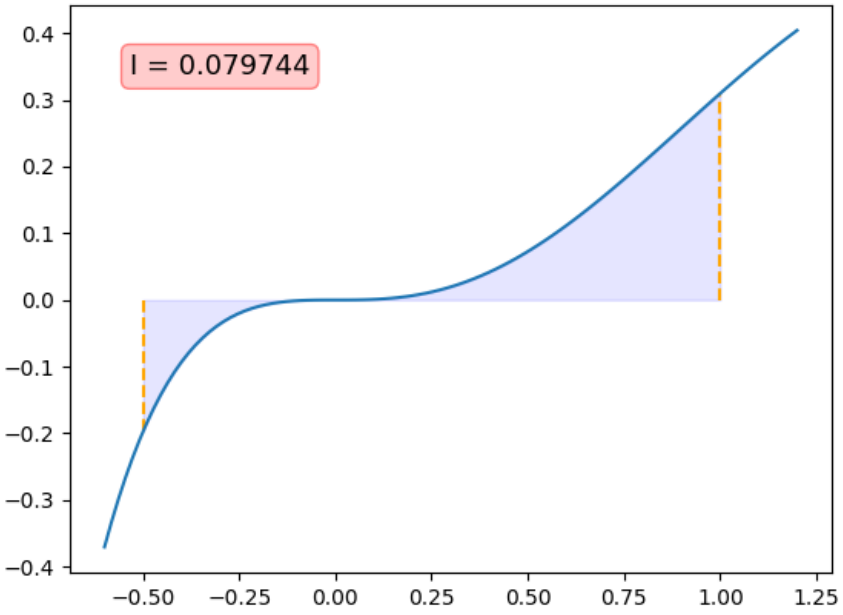
$$I = \int_0^1 x^2 \sin(2x) e^{-x}\, dx$$

Que calcularemos tal que:

```
from scipy.integrate import quad

int = lambda x: x**2*np.sin(x)*np.exp(-x)

I, Ierror = quad(int,-0.5,1)

x=np.linspace(-0.6,1.2,100)
xint=np.linspace(-0.5,1,100)

fig, ax = plt.subplots()
plt.vlines(x = 1, ymin = 0, ymax = int(1),
           colors = 'orange', linestyles='—',
```

```
            label = 'vline multiple - full height')
plt.vlines(x = -0.5, ymin = 0, ymax = int(-0.5),
            colors = 'orange', linestyles='--',
            label = 'vline multiple - full height')
plt.text(-0.3, 0.35, f'I = {I:.6f}', size=13,
         ha="center", va="center",
         bbox=dict(boxstyle="round",
                   ec=(1., 0.5, 0.5),
                   fc=(1., 0.8, 0.8),
                   )
         )

ax.plot(x,int(x))
ax.fill_between(xint, int(xint), 0, color='blue', alpha=.1)

plt.show()
```

Definimos el integrando mediante al función lambda y lo introducimos en la función quad() junto con los límites de integración. En I guardamos el valor de la integral y en Ierror el error de cálculo. El resto son cosméticos para la figura, con plt.vlines() colocamos las líneas verticales de que marcan los límites de integración y con ax.fillbetween() coloreamos el área integrada.



Por último en esta sección vamos a ver como hacer una integral sobre dos variables, que no difiere mucho de el ejemplo anterior. Tomaremos la siguiente integral:

$$I = \int_0^1 \int_{-x}^{x^2} \sin(x^2 + y^2)\, dy\, dx$$

```
from scipy.integrate import dblquad

int = lambda x,y: np.sin(x**2 + y**2)

y_inf = lambda x: x**2
y_sup = lambda x: -x

I, Ierror = dblquad(int,0,1, y_inf, y_sup)

print(I)
print(Ierror)

plt.show()
```

La única diferencia con la integral unidimensional es usar dblquad() en vez de quad() y que luego definimos las funciones para los límites de integración dependientes de x.

Ecuaciones diferenciales

EDOs

Vamos a comenzar con una ecuación diferencial ordinaria de primer orden. Como ejemplo vamos a tomar la ecuación simplificada de un objeto en caida libre con rozamiento, dado por el coeficiente a tal que:

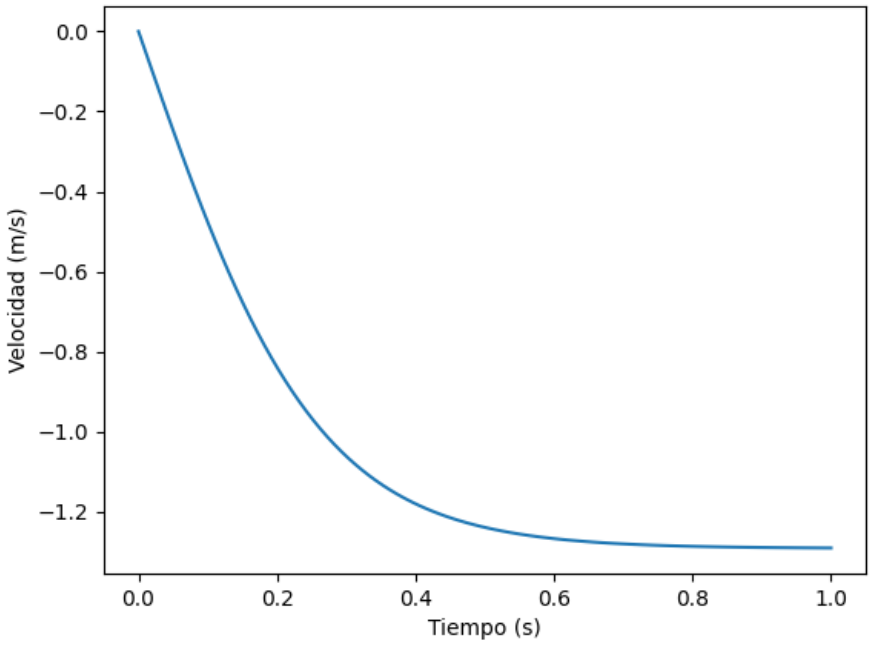
$\dot{v} - av^2 + b = 0 \quad \text{for} \quad v(0) = 0$

Nos interesa definir la función como $\frac{dv}{dt} = av^2 - b$ de froma que:

```
from scipy.integrate import odeint

a=3
b=5

def dvdt(v,t):
    return a*v**2 - b
v0 = 0

t = np.linspace(0,1,100)
sol = odeint(dvdt, v0, t)

sol.T[0]

plt.plot(t, sol.T[0])
plt.show()
print(sol)
```

Una vez definimos la función $\frac{dv}{dt}$ dependiente de la velocidad y del tiempo, solo tenemos que introducirla en odeint() junto con el valor inical y el tiempo de recorrido. Para representar el resultado conviene trasponer el vector de la solución.

Y la figura que obtenemos es la siguiente:



Que es lo que esperaríamos, partiendo del reposo el objeto empieza a adquirir velocidad negativa (hacia el suelo) debido a la gravedad hasta alcanzar una velocidad terminal por entre 1,2 y 1,4 m/s.

EDOs acopladas

Tenemos para esta sección dos ecuaciones diferenciales que dependen la una de la otra, es decir están acopladas entre ellas. Vamos a tomar el siguiente ejemplo:

$$\dot{y}_1 = y_1 + y_2^2 + 3x$$

$$\dot{y}_2 = 3y_1 + y_2^3 - \cos x$$

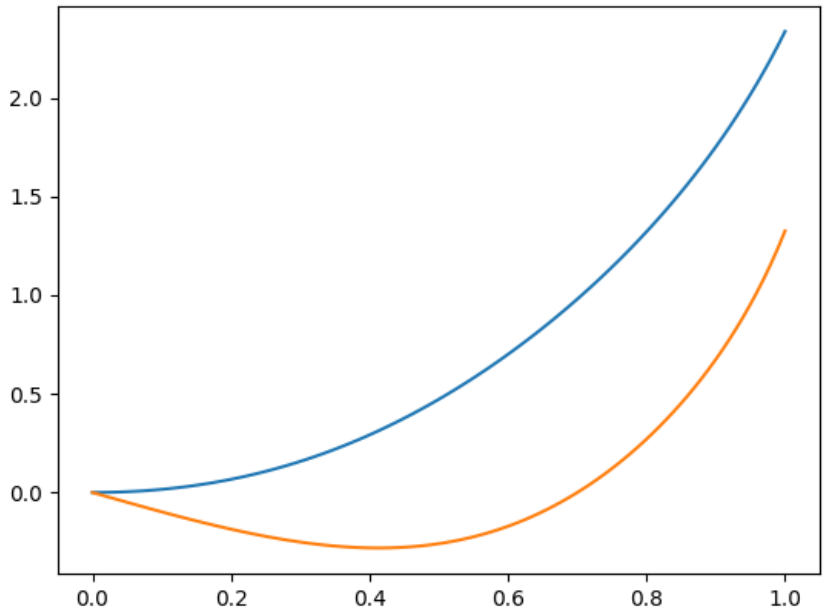
```
def dSdx(S, x):
    y1 , y2 = S
    return [y1+y2**2+3*x, 3*y1+y2**3-np.cos(x)]

y1_0 = 0
y2_0 = 0
```

```
S_0 = (y1_0,   y2_0)

x = np.linspace(0,1,100)
sol = odeint(dSdx, S_0,x)
y1 = sol.T[0]
y2 = sol.T[1]

plt.plot(x,y1)
plt.plot(x,y2)
plt.show()
```

Primero definiremos una función dSdx que nos incluya las dos variables y1 e y2 dentro de S. Posteriormente damos los valores iniciales y los introducimos en una S inicial.

Por último solo tenemos que crear el vector para x e introducir todo en la función odeint. Obtenemos de ahí los resultados y los trasponemos para poder representarlos sin problemas.

Obtenemos la siguiente figura:



EDOs de segundo orden

Para ver la resolución de una EDO de segundo orden vamos a tomar de ejemplo la ecuación de un péndulo tal que:

$$\ddot{\theta} - \sin\theta = 0$$

Python no puede resolver de forma directa una ecuación de segundo orden o superior, por lo que para resolver estas, las tenemos que escribir como una serie de EDOs acopladas. Para hacer esto, vamos a definir una nueva variable $\omega = d\theta/dt$ de forma que nos quedan las ecuaciones:

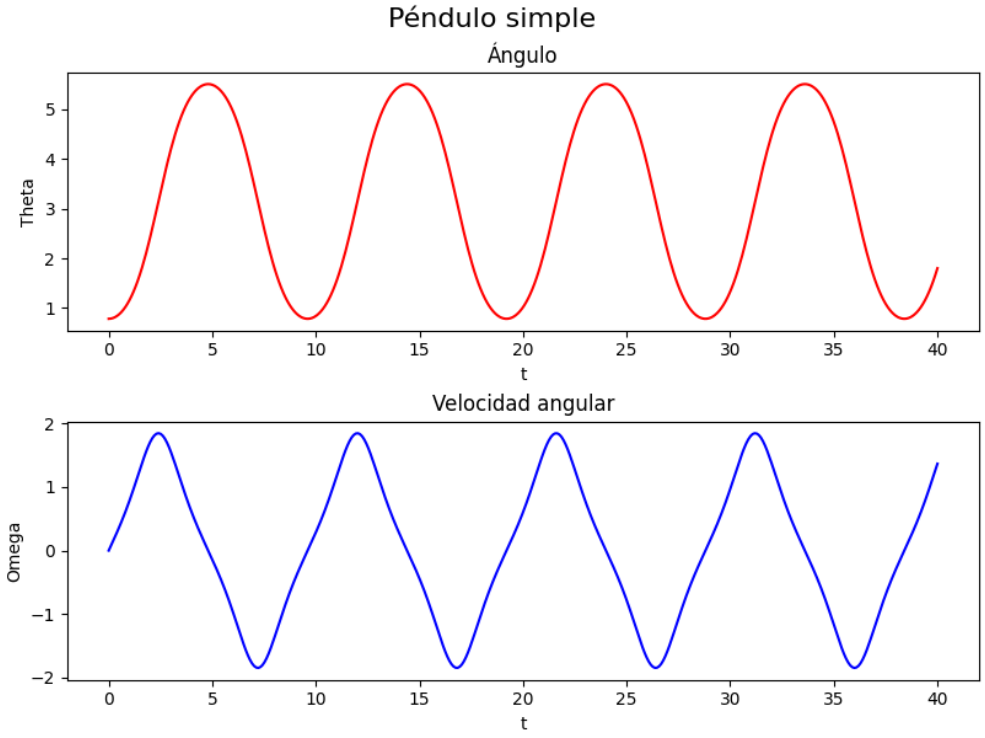
$$d\omega/dt = \sin\theta$$

$$d\theta/dt = \omega$$

```
def dSdt(S,t):
    theta,omega=S
    return [omega,np.sin(theta)]
x0= np.pi/4
z0=0
S0 = (x0, z0)
t=np.linspace(0,40,400)
sol= odeint(dSdt,S0,t)
x,z=sol.T

fig, (ax1, ax3) = plt.subplots(2, figsize=(8, 6),
layout='constrained')
ax1.plot(t, x, '-', c='r')
ax3.plot(t, z, '-', c='b')
ax1.set_title('Angulo')
ax3.set_title('Velocidad angular')
ax1.set_ylabel('Theta')
ax1.set_xlabel('t')
ax3.set_ylabel('Omega')
ax3.set_xlabel('t')
fig.suptitle('Pendulo simple',
fontsize=16)
```

Obramos de igual manera que en el apartado anterior, definimos dSdt donde estará la variable S que nos incluye θ y ω y nos devuelve el resultado del otro lado de la ecuación. Introducimos a continuación los valores iniciales y el rango del tiempo para finalmente dárselo todo a la función odeint y obtener la solución, la cual de nuevo trasponemos para poder usarla posteriormente.

Por último simplemente creamos un subplot para representar la solución, tanto del ángulo del péndulo como de su velocidad angular, obteniendo:



Polinomios de Legendre

Los polinomios de Legendre corresponden a la solución de la ecuación diferencial de Legendre:

$$(1-x^2)\ddot{y} - 2x\dot{y} + p(p+1)y = 0 \qquad para\ p\ (cte)\ > -1$$

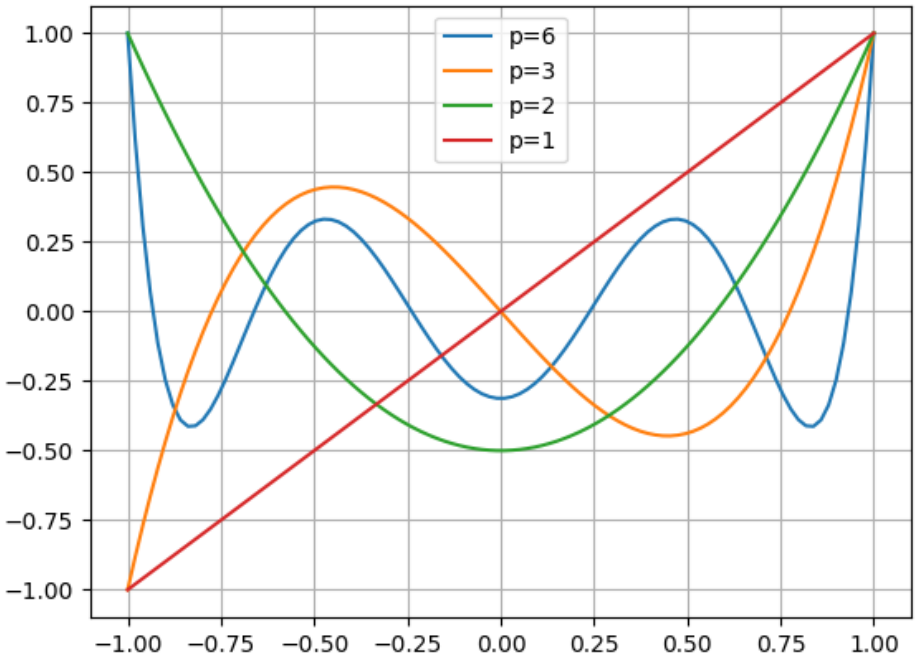
Esta ecuación suele aparecer al resolver problemas en coordenadas esféricas (en cuyo caso, la interpretación geométrica es que $x = \cos\theta$).

Vamos a representar gráficamente varias de las soluciones, por ejemplo para los casos p=1,2,3,6.

```
from scipy.special import legendre

x = np.linspace(-1,1,100)

plt.plot(x,legendre(6)(x),label='p=6')
plt.plot(x,legendre(3)(x),label='p=3')
plt.plot(x,legendre(2)(x),label='p=2')
plt.plot(x,legendre(1)(x),label='p=1')
plt.legend(loc='upper center',fontsize=10)

plt.grid()
plt.show()
```

Para llamar el polinomio solo tenemos que usar legendre()() añadiendo primero el valor de p y luego el vector de x.

Obtenemos por lo tanto la siguiente figura:



Funciones de Bessel

Corresponden a las soluciones de la ecuación de Bessel:

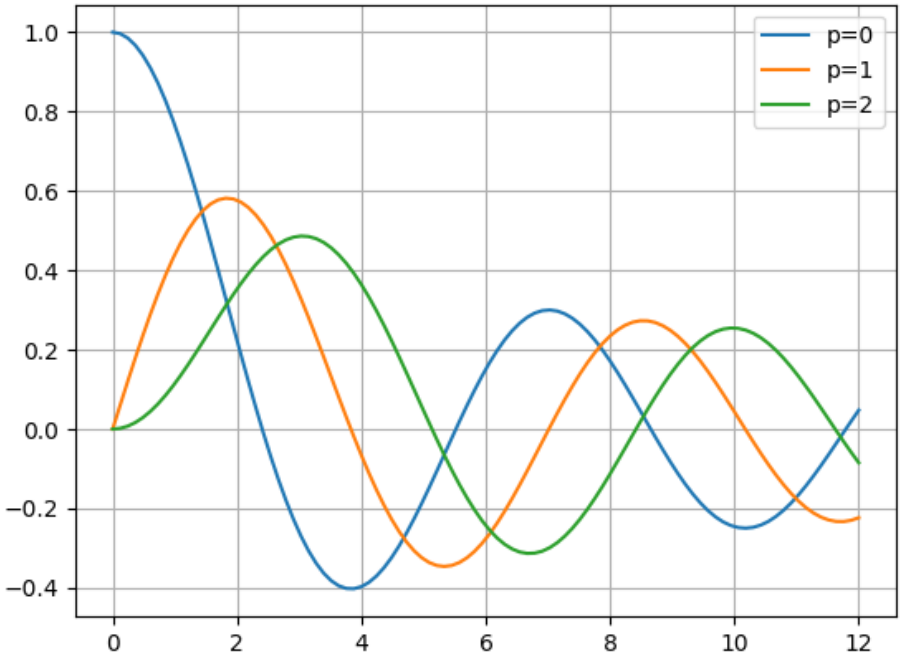
$$x^2\ddot{y} + x\dot{y} + (x^2 - p^2)y = 0 \qquad para\ p\ (cte)\ \geq 0$$

Esta ecuación suele aparecer al resolver problemas en coordenadas cilíndricas (la interpretación geométrica es que $x = \rho$). Algunos ejemplos pueden ser una onda electromagnética en una guía de ondas cilíndrica, soluciones para la ecuación de Schrödinger radial de una partícula libre o la difracción de objetos helicoidales como el ADN.

```
from scipy.special import jv

x = np.linspace(0,12,100)

plt.plot(x,jv(0,x),label='p=0')
plt.plot(x,jv(1,x),label='p=1')
```

```
plt.plot(x,jv(2,x),label='p=2')

plt.legend(loc='upper right',fontsize=10)


plt.grid()
plt.show()
```

Obtenemos así la siguiente figura:



Polinomios de Hermite

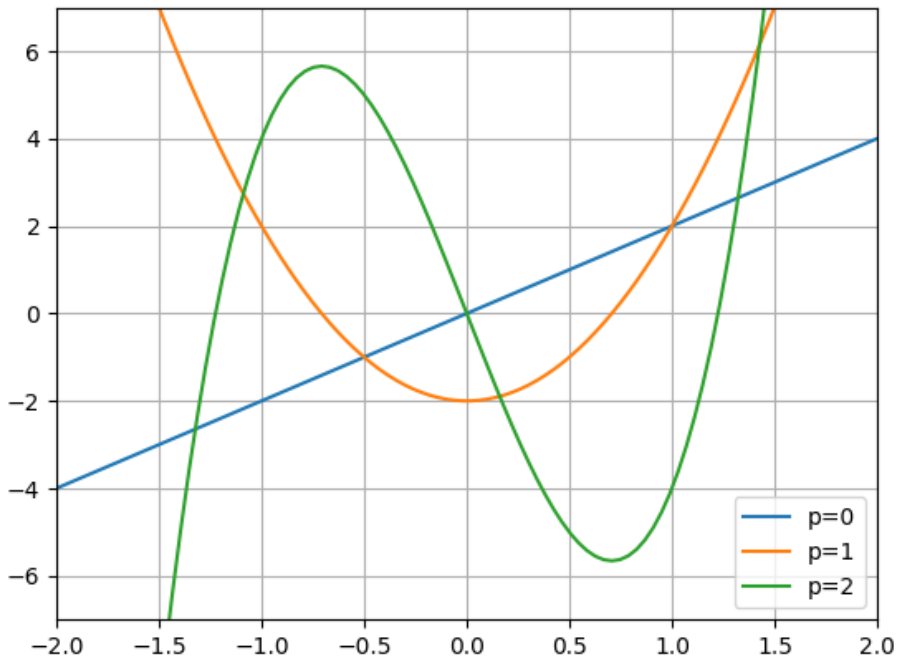
Corresponden a las soluciones de la ecuación de Hermite y son un ejemplo de secuencia de polinomios ortogonales :

$$\ddot{y} - 2x\dot{y} + 2py = 0 \qquad para\ p\ (cte)$$

Esta ecuación suele aparecer al resolver problemas en de mecánica cuántica y el oscilador armónico unidimensional. Otros ejemplos son en procesado de señales, en análisis numérico o probabilidad como en movimiento browniano.

```
from scipy.special import hermite

x = np.linspace(-2,2,100)

plt.plot(x,hermite(1)(x),label='p=0')
plt.plot(x,hermite(2)(x),label='p=1')
plt.plot(x,hermite(3)(x),label='p=2')

plt.legend(loc='lower right',fontsize=10)
plt.ylim([-7,7])
plt.xlim([-2,2])


plt.grid()
plt.show()
```

Obtenemos así la siguiente figura:



Transformadas de Fourier

El análisis de Fourier nos permite expresar funciones o aproximarlas mediante suma de funciones trigonométricas simples. En el caso de cálculos númericos a ordenador se utiliza la transformada de Fourier discreta (DFT) que se puede expresar como:

$$y[k] = \sum_{n=0}^{N-1} \exp -2\pi j \frac{kn}{N} x[n]$$

y cuya inversa es

$$x[n] = \frac{1}{N} \sum_{n=0}^{N-1} \exp 2\pi j \frac{kn}{N} y[k]$$

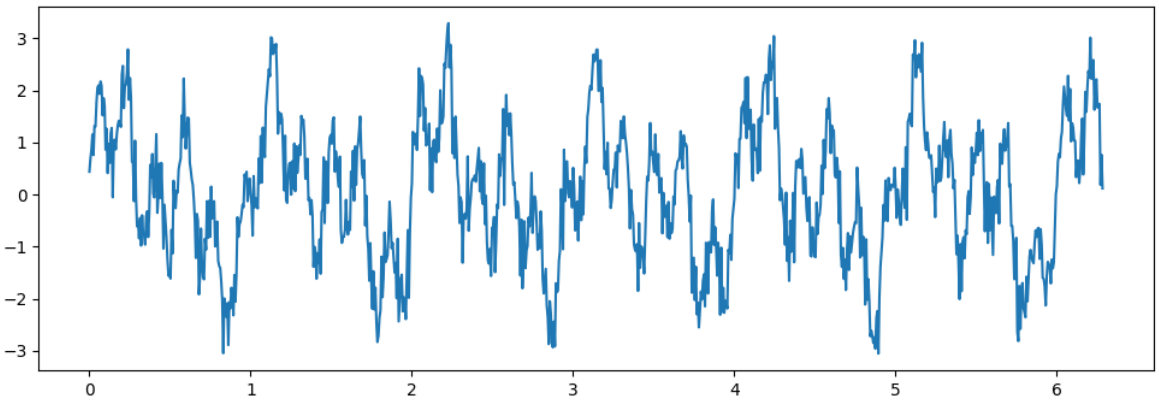
Estas se calculan mediante los algoritmos FFT (Fast Fourier Transform) e IFFT (Inverse Fast Fourier Transform).

Para ver un ejemplo sencillo, vamos a tener una señal compuesta por tres senos de frecuencia 1 y 2 y 5.5 con bastante ruido en toda esta.

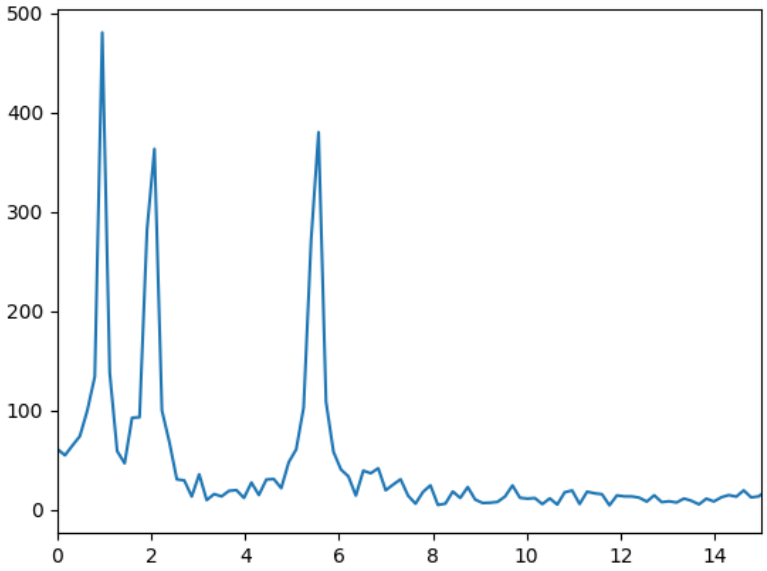
```
from scipy.fft import fft, fftfreq

t = np.linspace(0, 2*np.pi, 1000)

def f(x):
    return np.sin(2*np.pi*x) + np.sin(4*np.pi*x)
    + 0.4*np.random.randn(len(x))

N = len(t)
y = fft(f(t))
ypos=fft(f(t))[:N//2]
frec= fftfreq(N, np.diff(t)[0])[:N//2]

plt.plot(t,f(t))
plt.show()
plt.plot(frec, np.abs(ypos))
plt.xlim(0,15)
plt.show()
```

Definimos el tiempo sobre el que trabajaremos como vector y la función de la que queremos obtener las frecuencias. N será la longitud del vector del tiempo. Llamando fft() realizamos la transformada sobre la función definida en t. Luego con ypos obtenemos solo los valores positivos que son simétricos a los positivos de la transformada. En frec estamos guardando el vector de los valores de las frecuencias usando fftfreq() introduciendo N y la diferencia entre valores del vector tiempo.

Primero obtenemos la representación de la señal con ruido de la cual queremos extraer el espectro de sus frecuencias tal que:



Y su espectro de frecuencias es:



Donde vemos claramente que esta compuesta principalmente por tres frecuencias, de 1, 2 y 5.5 unidades.