

PYTHON



Que es python

Python es un **lenguaje de programación de alto nivel**, interpretado, de propósito general y con una sintaxis clara y sencilla. Fue creado por **Guido van Rossum** y lanzado en 1991 con el objetivo de facilitar la escritura y la lectura del código, promoviendo la productividad y la simplicidad en el desarrollo de software.



Características Principales de Python

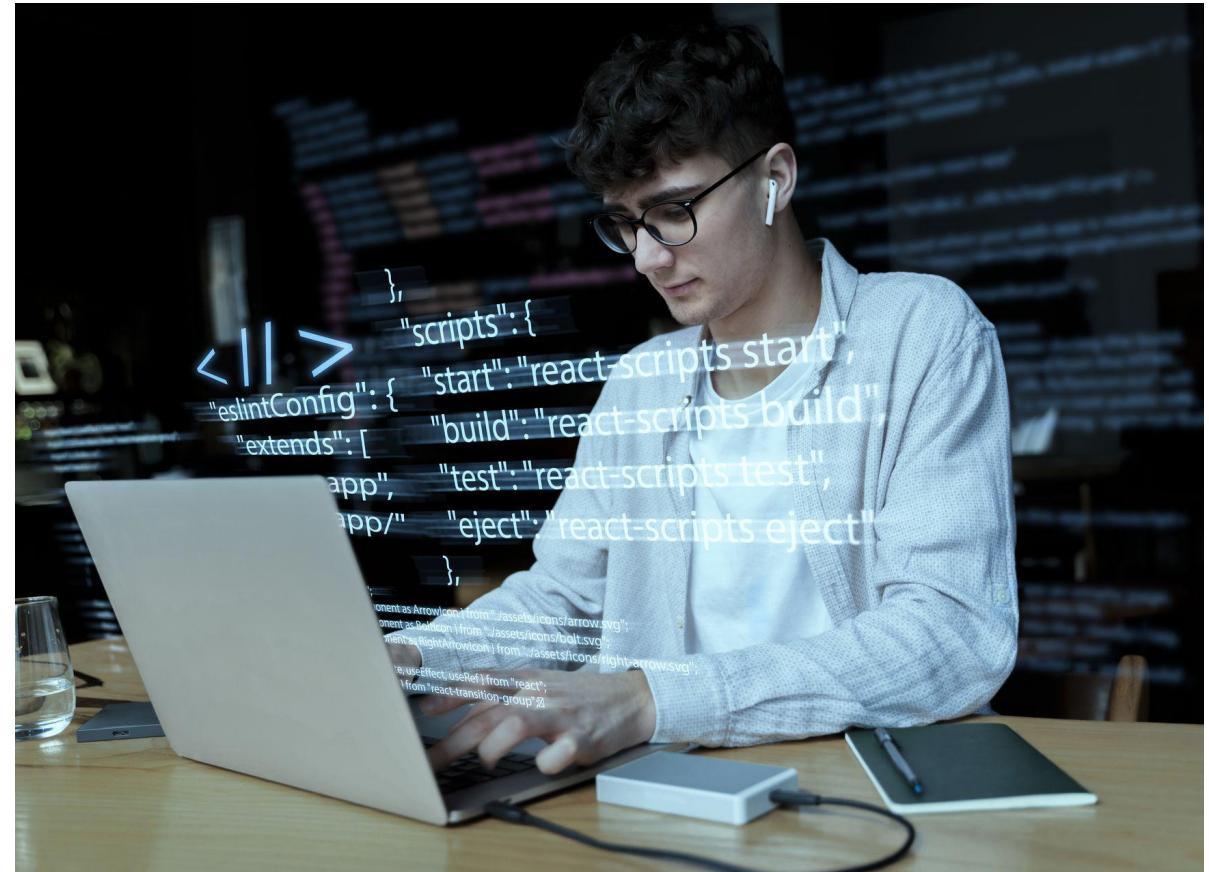
1. Sencillo y Fácil de Aprender

Su sintaxis es clara y concisa, lo que permite escribir código de manera más intuitiva en comparación con otros lenguajes como C++ o Java.

2. Interpretado y Multiplataforma

Python es un lenguaje **interpretado**, lo que significa que no necesita ser compilado antes de ejecutarse.

Es **multiplataforma**, lo que permite que un mismo código funcione en **Windows**, **macOS** y **Linux** sin modificaciones.

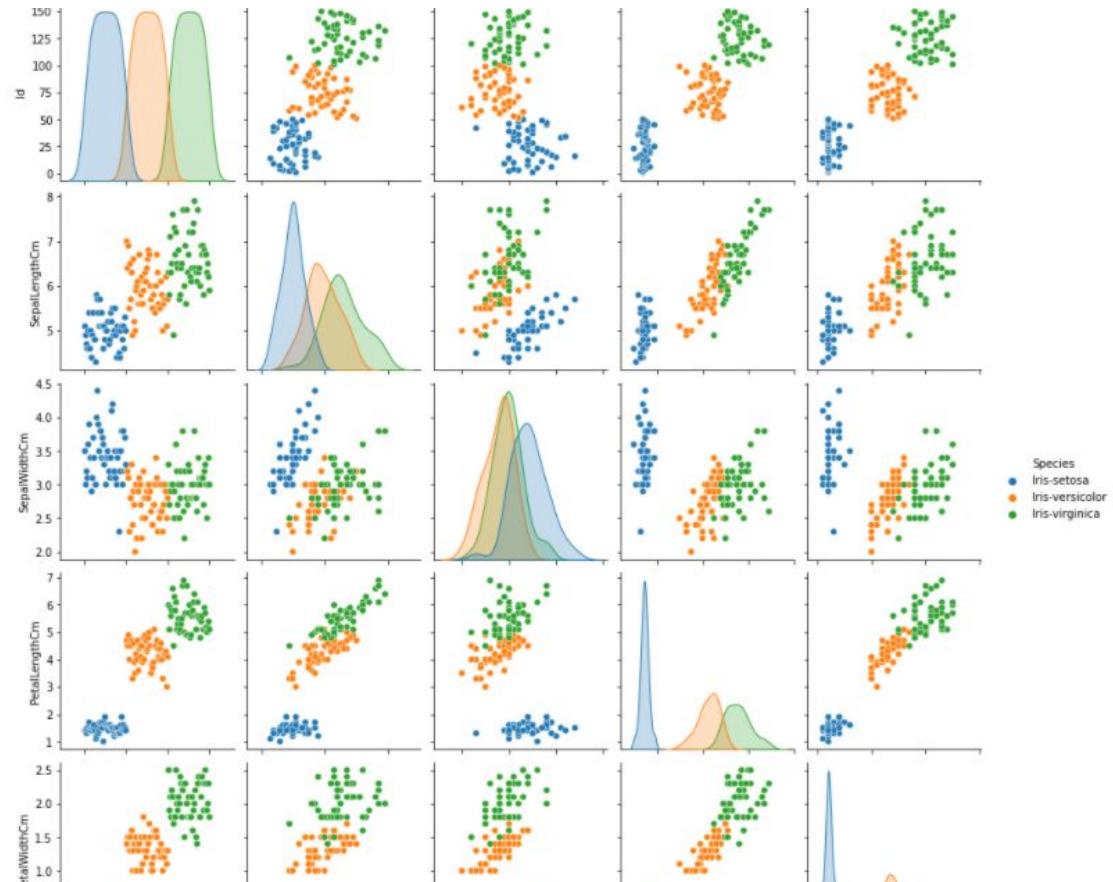


Características Principales de Python

3. Lenguaje de Propósito General

Se puede utilizar en una amplia variedad de aplicaciones, incluyendo:

- Ciencia de datos y análisis estadístico
- Desarrollo web y backend
- Automatización de tareas y scripting
- Inteligencia artificial y machine learning
- Ciberseguridad y hacking ético



Características Principales de Python

4. Gran Comunidad y Bibliotecas Extensas

Tiene una enorme comunidad de desarrolladores y científicos que contribuyen con librerías especializadas.

Algunas bibliotecas populares incluyen:

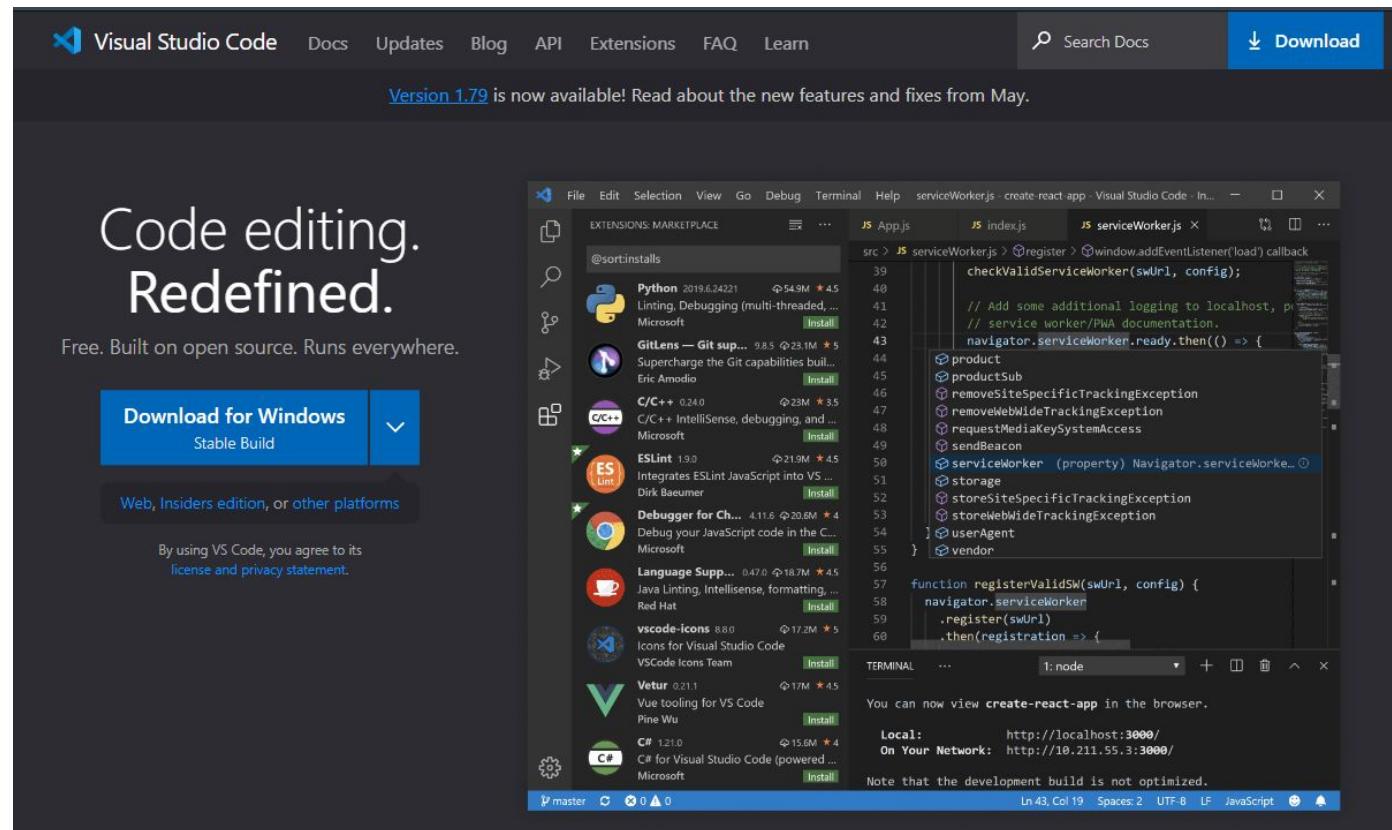
- NumPy, Pandas (análisis de datos)
- Matplotlib, Seaborn (visualización de datos)
- Scikit-learn, TensorFlow, PyTorch (machine learning)
- Flask, Django (desarrollo web)
- OpenCV (procesamiento de imágenes)



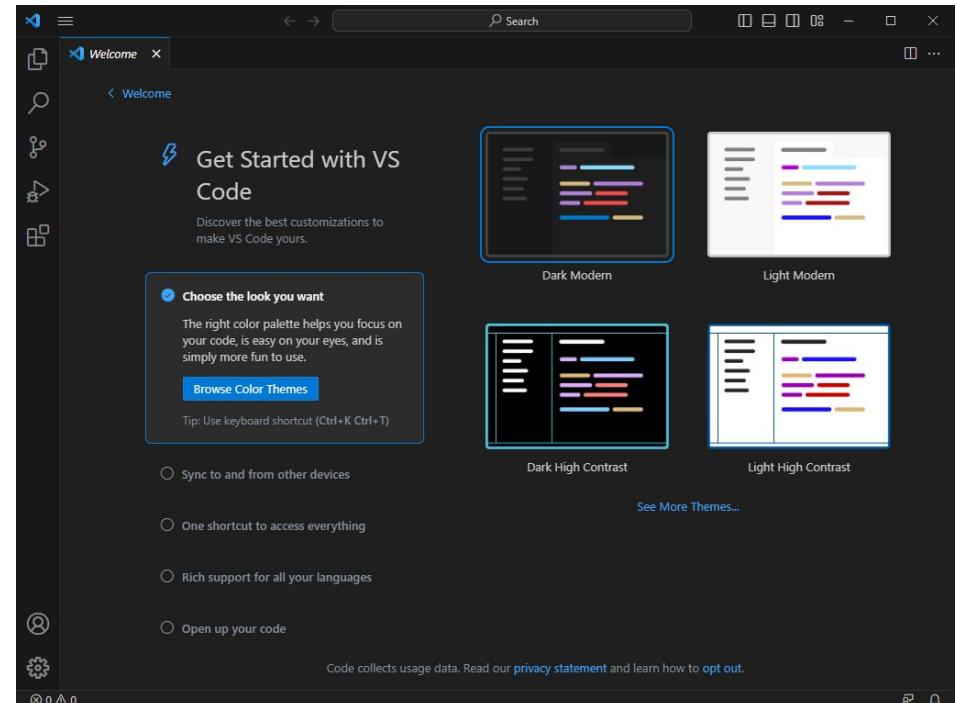
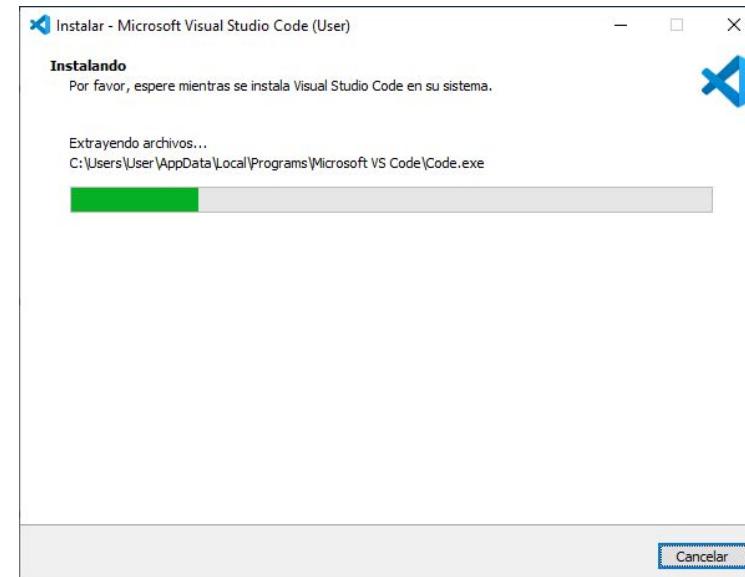
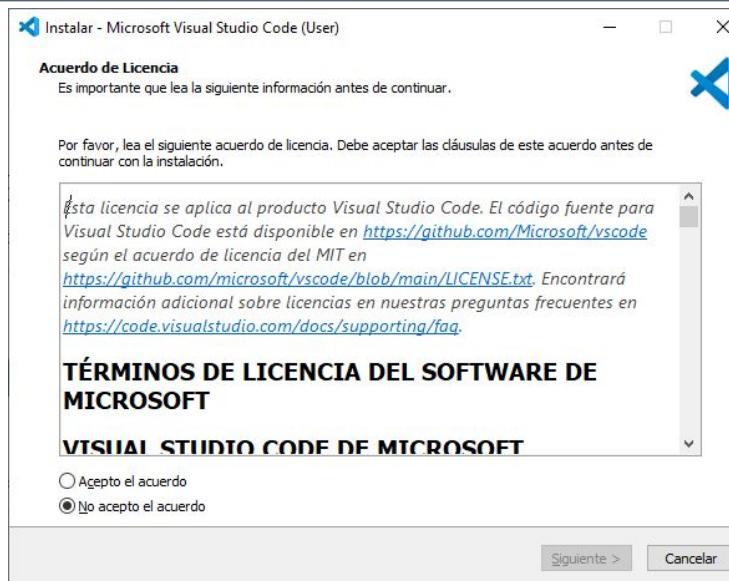
Visual Studio Code

Descargarlo de la pagina oficial e instalarlo:

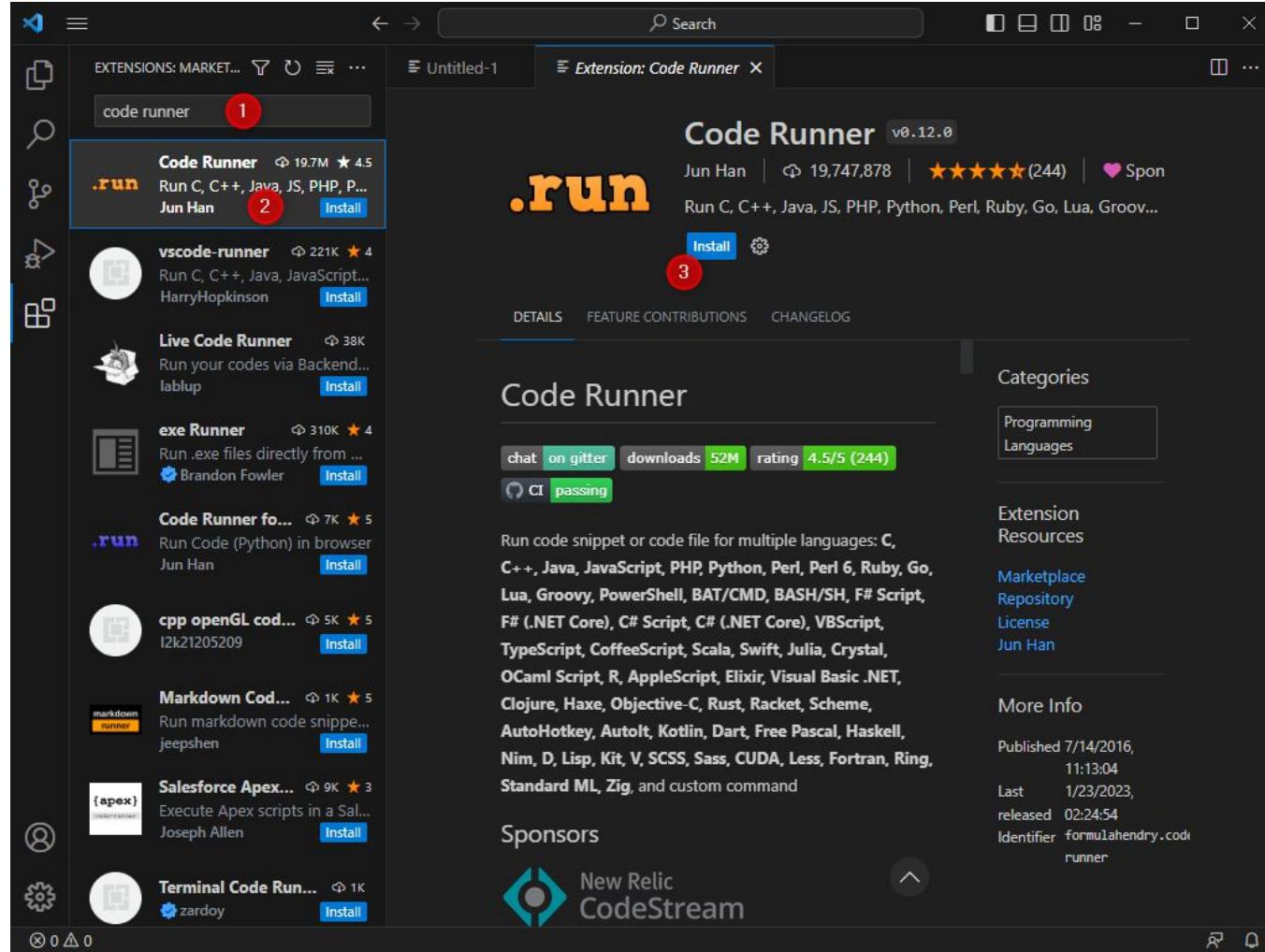
<https://code.visualstudio.com>



Visual Studio Code

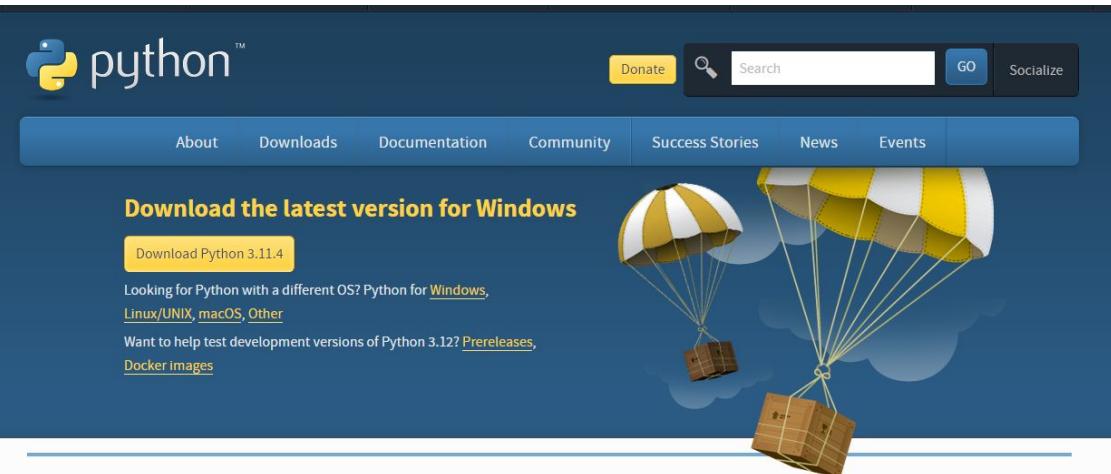


Visual Studio Code



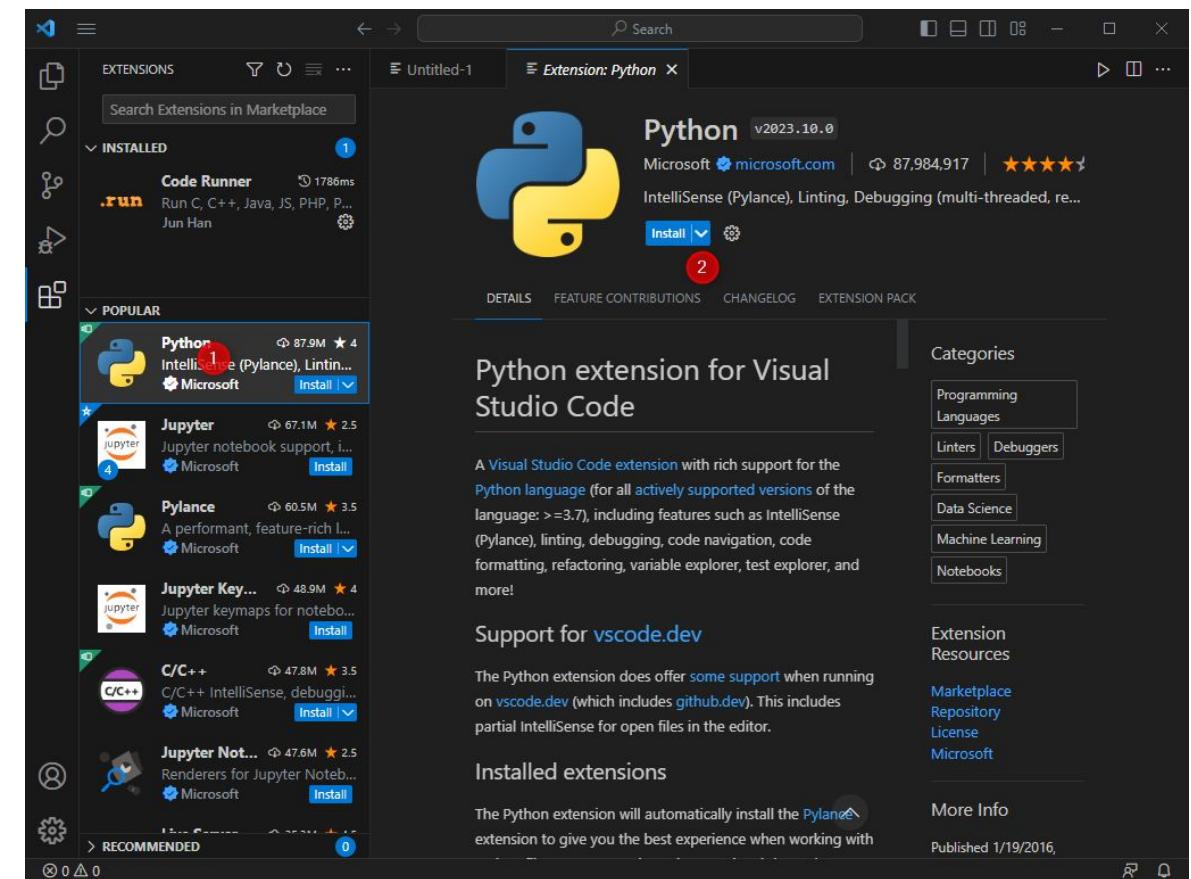
CAFAM

Visual Studio Code



Para mas información de como instalar Python, en vs code Vea el siguiente video

<https://www.youtube.com/watch?v=eFTThEXvuZaM>



Python



```
prueba.py
prueba.py > ...
1 saludo = "Hola Mundo"
2 print(saludo)
3 nombre = input("dime tu nombre:")
4 print("tu nombre es:" + nombre)
```

Ejecuta el programa

Input:

Función que permite al usuario introducir información por medio del teclado al ejecutarse, otorgándole una instrucción acerca del ingreso solicitado. El código continuará ejecutándose luego de que el usuario realice la acción.

```
print("Tu nombre es " + input("Dime tu nombre: "))
```

Print:

Declaración que al ejecutarse muestra (o imprime) en pantalla el argumento que se introduce dentro de los paréntesis.
mostrar texto

Ingresamos entre comillas simples o dobles los caracteres de texto que deben mostrarse en pantalla.

```
print("Hola mundo")
```

```
>> Hola mundo
```

mostrar números

Podemos entregarle a print() el número que debe mostrar, o una operación matemática a resolver. No empleamos comillas en estos casos.

```
print(150 + 50)
```

```
1 print("El nombre de tu cerveza\nes " + input("Que ciudad te gustaria visitar?: ") + " " + input("Cual es tu color favorito?: ") + "\nFelicitaciones!")
2
```

strings

Los strings en Python son un tipo de dato formado por cadenas (o secuencias) de caracteres de cualquier tipo, formando un texto.

concatenación

Unificación de cadenas de texto:

```
print("Hola" + " " + "mundo")
```

```
>> Hola mundo
```

caracteres especiales

Indicamos a la consola que el carácter a continuación del símbolo \ debe ser tratado como un carácter especial.

\\" > Imprime comillas

\n > Separa texto en una nueva linea

\t > Imprime un tabulador

\\" > Imprime la barra invertida textualmente

```
print("Me llamo \"Federico\"")
print("Esta es una linea\nY esta es otra linea")
print("\tEsta es la tercera linea")
print('Este signo \\ es una barra invertida')
```

Tipos de datos

Tipos de Datos Numéricos:

Enteros (int)

Representan números enteros, positivos o negativos, sin decimales.

```
x = 10  
y = -25  
print(type(x)) # Salida: <class 'int'>
```

Flotantes (float)

Representan números con decimales.

```
pi = 3.1416  
salario = 52000.75  
print(type(pi)) # Salida: <class 'float'>
```

Complejos (complex)

Representan números complejos en la forma $a+bj$, donde j es la unidad imaginaria.

```
z = 2 + 3j  
print(type(z)) # Salida: <class 'complex'>
```

Tipos de Números

int

Int, o integer, es un número entero, positivo o negativo, sin decimales, de un largo indeterminado.

```
num1 = 7
```

```
print(type(num1))
```

float

Float, o "número de punto flotante" es un número que puede ser positivo o negativo, que a su vez contiene una o más posiciones decimales.

```
num2 = 7.525587
```

```
print(type(num2))
```

```
edad = input("Dime tu edad: ")  
edad = int(edad)  
nueva_edad = 1 + edad
```

```
int(var)  
>> <class 'int'>
```

```
float(var)  
>> <class 'float'>
```



Convierte el dato en integer



Convierte el dato en float

Tipos de datos

Cadenas de Caracteres (str)

Se representan con comillas simples ' o dobles "

```
nombre = "Python"
mensaje = 'Hola, mundo'
print(type(nombre)) # Salida: <class 'str'>
```

Concatenación de cadenas:

```
saludo = "Hola" + " " + "Mundo"
print(saludo) # Salida: Hola Mundo

print("Python " * 3) # Salida: Python Python Python
```

Tipos de Datos Booleanos.

Solo tienen dos valores: True y False.

```
es_mayor = True
es_menor = False
print(type(es_mayor)) # Salida: <class 'bool'>
```

```
edad = 20
print(edad > 18) # Salida: True
```

Tipos de Datos Estructurados (Colecciones)

Listas (list)

Son **mutables** (pueden modificarse).

Pueden contener distintos tipos de datos.

```
numeros = [1, 2, 3, 4, 5]
frutas = ["manzana", "banana", "cereza"]
print(type(numeros)) # Salida: <class 'list'>
```

Tuplas (tuple)

Son **inmutables** (no pueden modificarse después de su creación).

Pueden ser anidadas (igual que list)

```
coordenadas = (10, 20)
colores = ("rojo", "verde", "azul")
print(type(coordenadas)) # Salida: <class 'tuple'>
```

Conjuntos (set)

No permiten elementos duplicados.

Son **desordenados** (no tienen índice).

```
conjunto = {1, 2, 3, 3, 4, 5}
print(conjunto) # Salida: {1, 2, 3, 4, 5}
```

Tipos de Datos Estructurados (Colecciones)

Diccionarios (dict)

Estructura clave-valor.

Las claves son únicas, es decir, no están repetidas, y nos permiten acceder al objeto almacenado.

```
diccionario = {  
    'clave1': 'Mi primer valor',  
    'clave3': 'Y, como no, mi tercer valor',  
    'clave2': 'Este es mi segundo valor'  
}  
  
diccionario['clave1'] # Devolverá 'Mi primer valor'
```

PYTHON

En Python tenemos varios tipos o estructuras de datos, que son fundamentales en programación ya que almacenan información, y nos permiten manipularla.

string (str)	"hola", "%\$&", " ", "123"
integer (int)	150, 1, 555, -15, 0
floating (float)	1.25, 25.0, 500.50, -95.1
listas (lst)	["sal", 1, -3, 4.5, "marte", 0]
diccionarios (dic)	{'color':'rojo', 'arte':'cine' }
tuples (tup)	('lun','mar','mie','jue','vie')
sets (set)	{'a', 'b', 'c', 'd', 'e'}
booleanos (bool)	True, False

estructuras	mutable	ordenado	duplicados
listas []	✓	✓	✓
tuplas ()	✗	✓	✓
sets { }	✓	✗	✗
diccionarios { }	✓	✗ *	✗ : ✓ **

CADENAS LITERALES

Para facilitar la concatenación de variables y texto en Python, contamos con dos herramientas que nos evitan manipular las variables, para incorporarlas directamente al texto:

Función format: se encierra las posiciones de las variables entre corchetes {}, y a continuación del string llamamos alas variables con la función format

```
color_auto="rojo"
matricula="MHX 007"
print("Mi auto es {} y de matrícula {}".format(color_auto,matricula))
```

Cadenas literales (f-strings): a partir de Python 3.8, podemos anticipar la concatenación de variables anteponiendo f al string

```
print(f"Mi auto es {color_auto} y de matrícula {matricula}")
```

Operadores Matemáticos

Suma: +

Resta: -

Multiplicación: *

División: /

Cociente (división "al piso"): //

Resto (módulo): %

Potencia: **

```
x = 6
y = 2
z = 7

print(f"{x} mas {y} es igual a {x+y}")
print(f"{x} menos {y} es igual a {x-y}")
print(f"{x} por {y} es igual a {x*y}")
print(f"{x} dividido {y} es igual a {x/y}")

print(f"{z} dividido al piso de {y} es igual a {z//y}")
print(f"{z} modulo de {y} es igual {z%y}")
print(f"{x} elevado a la {y} es igual a {x**y}")
print(f"{x} elevado a la {3} es igual a {x**3}")
print(f"La raiz cuadrada de {x} es {x**0.5}")
```

Redondeo

El redondeo facilita la interpretación de los valores calculados al limitar la cantidad de decimales que se muestran en pantalla.

También, nos permite aproximar valores decimales al entero más próximo.

round(numero, #dígitos a redondear)

```
print(round(100/3))
>> 33
print(round(12/7,2))
>> 1.71
```

If - else

El control de flujo determina el orden en que el código de un programa se va ejecutando. En Python, el flujo está controlado por estructuras condicionales, loops y funciones.

estructuras condicionales (if)

*Expresión de resultado booleano
(True/False)*

*la indentación
es obligatoria
en Python*

```
if expresión:  
    código a ejecutarse  
elif expresión:  
    código a ejecutarse  
elif expresión:  
    código a ejecutarse  
...  
else:  
    código a ejecutarse
```

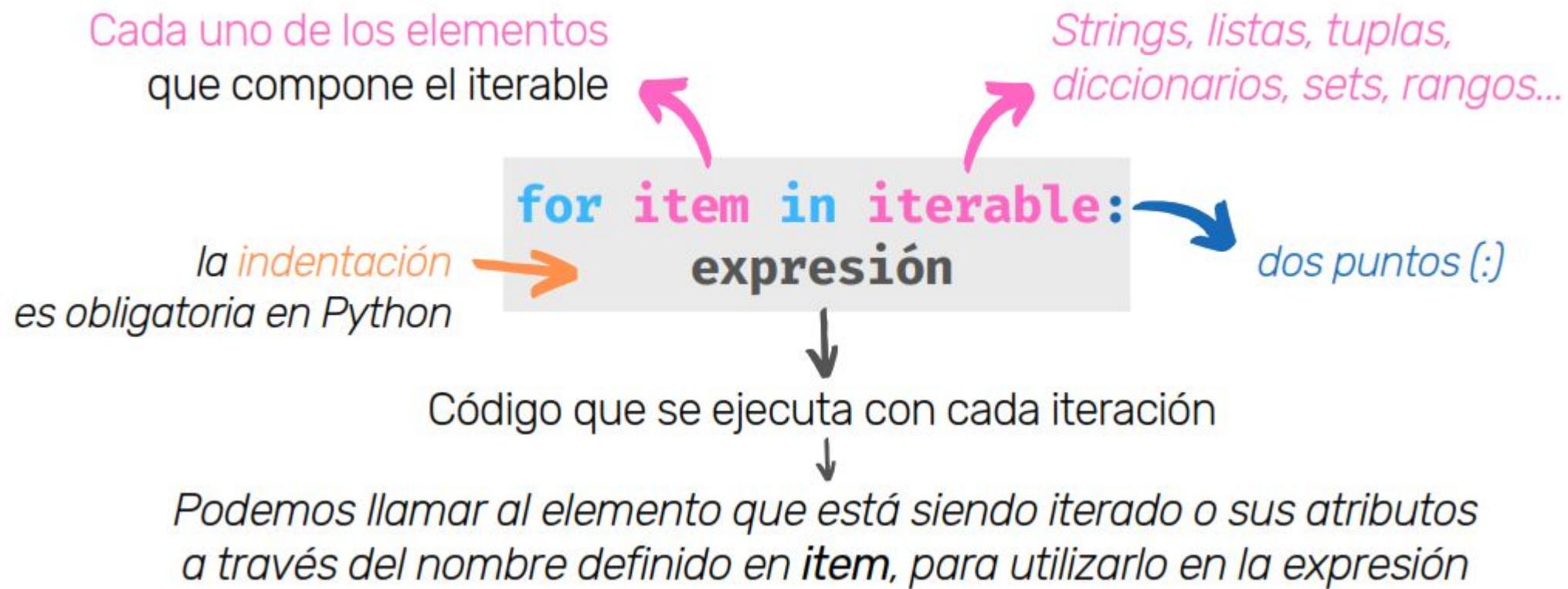
*Los dos puntos (:) dan paso al código
que se ejecuta si expresión = True*

*else & elif son
opcionales*

*pueden incluirse
varias cláusulas elif*

for

A diferencia de otros lenguajes de programación, los loops for en Python tienen la capacidad de iterar a lo largo de los elementos de cualquier secuencia (listas, strings, entre otros), en el orden en que dichos elementos aparecen.



range

La función range() devuelve una secuencia de números dados 3 parámetros. Se utiliza fundamentalmente para controlar el número de ejecuciones de un loop o para crear rápidamente una serie de valores.

El único parámetro obligatorio es valor_final. Los valores predeterminados para valor_inicio y paso son 0 y 1 respectivamente.

número a partir del cual inicia el rango (incluido)

diferencia entre cada valor consecutivo de la secuencia

```
range(valor_inicio, valor_final, paso)
```

número antes del cual el rango finaliza (no incluido)

```
print(list(range(1,13,3)))  
>> [1,4,7,10]
```

while

Si bien los loops while son otro tipo de bucles, resultan más parecidos a los condicionales if que a los loops for. Podemos pensar a los loops while como una estructura condicional que se ejecuta en repetición, hasta que se convierte en falsa.



- Si el código llega a una instrucción **break**, se produce la salida del bucle.
- La instrucción **continue** interrumpe la iteración actual dentro del bucle, llevando al programa a la parte superior del bucle.
- La instrucción **pass** no altera el programa: ocupa un lugar donde se espera una declaración, pero no se desea realizar una acción.

match

En Python 3.10, se incorpora la coincidencia de patrones estructurales mediante las declaraciones `match` y `case`. Esto permite asociar acciones específicas basadas en las formas o patrones de tipos de datos complejos.

```
match objeto:  
    case <patron_1>:  
        <accion_1>  
    case <patron_2>:  
        <accion_2>  
    case <patron_3>:  
        <accion_3>  
    case _:  
        <accion_comodin>
```



El carácter `_` es un comodín que actúa como coincidencia si la misma no se produce en los casos anteriores.

zip

La función zip() crea un iterador formado por los elementos agrupados del mismo índice provenientes de dos o más iterables. Zip deriva de zipper (cremallera o cierre), de modo que es una analogía muy útil para recordar.



La función se detiene al cuando se agota el iterable con menor cantidad de elementos.

```
letras = ['w', 'x', 'c']
numeros = [50, 65, 90, 110, 135]
for letra, num in zip(letras, numeros):
    print(f'Letra: {letra}, y Número: {num}')
```

```
>> Letra: w, y Número: 50
>> Letra: x, y Número: 65
>> Letra: c, y Número: 90
```

Min y max

La función min() retorna el item con el valor más bajo dentro de un iterable. La función max() funciona del mismo modo, devolviendo el valor más alto del iterable. Si el iterable contiene strings, la comparación se realiza alfabéticamente.

```
ciudades_habitantes = {"Tijuana":1810645, "León":1579803}  
lista_valores = [5**5, 12**2, 3050, 475*2]
```

```
print(min(ciudades_habitantes.keys()))  
>> León
```

```
print(max(ciudades_habitantes.values()))  
>> 1810645
```

```
print(max(lista_valores))  
>> 3125
```

Random

Python nos facilita un módulo (un conjunto de funciones disponibles para su uso) que nos permite generar selecciones pseudo-aleatorias* entre valores o secuencias.

Nombre del módulo

```
from random import *
```

* = Todos los métodos

También pueden importarse de manera independiente aquellos a utilizar.

`randint(min, max)`: devuelve un `integer` entre dos valores dados (ambos límites incluidos)

`uniform(min, max)`: devuelve un `float` entre un `valor mínimo` y `uno máximo`

`random(sin parámetros)`: devuelve un `float` entre 0 y 1

`choice(secuencia)`: devuelve un `elemento al azar` de una `secuencia` de valores (listas, tuples, rangos, etc.)

`shuffle(secuencia)`: toma una `secuencia` de valores mutable (como una lista), y la retorna cambiando el orden de sus elementos aleatoriamente.

* La mecánica en cómo se generan dichos valores aleatorios viene en realidad predefinida en "semillas". Si bien sirve para todos los usos habituales, no debe emplearse con fines de seguridad o criptográficos, ya que son vulnerables.

Compresión de listas

La comprensión de listas ofrece una sintaxis más breve en la creación de una nueva lista basada en valores disponibles en otra secuencia. Vale la pena mencionar que la brevedad se logra a costo de una menor interpretabilidad.

Diagrama que muestra la estructura de una lista comprehension en Python:

```
nueva_lista = [expresion for item in iterable if condicion == True]
```

- cada elemento del iterable*: Apunta a la palabra "item" dentro del bucle "for".
- tuplas, sets, otras listas...*: Apunta al clausula "if condicion == True".
- fórmula matemática*: Un brace que abarca la expresión "expresion" y la clausula "for item in iterable".
- operación lógica*: Un brace que abarca la clausula "if condicion == True".

Caso especial con else:

```
nueva_lista = [expresion if condicion == True else otra_expresion  
               for item in iterable]
```

Ejemplo:

```
nueva_lista = [num**2 for num in range(10) if num < 5]  
print(nueva_lista)  
>> [0, 1, 4, 9, 16]
```

Funciones en python

Una función es un bloque de código que solamente se ejecuta cuando es llamada. Puede recibir información (en forma de parámetros), y devolver datos una vez procesados como resultado.

una función es definida mediante la palabra clave def



```
def mi_funcion(argumento):
```



Los argumentos contienen información que la función utiliza o transforma para devolver un resultado

```
mi_funcion(mi_argumento)
```



Para llamar a una función, basta utilizar su nombre, entregando los argumentos que la misma requiere entre paréntesis.

return

Para que una función pueda devolver un valor (y que pueda ser almacenado en una variable, por ejemplo), utilizamos la declaración return.

```
def mi_funcion():
    return [algo]
```



La declaración return provoca la salida de la función
Cualquier código que se encuentre después en el bloque de la función, no se ejecutará

```
resultado = mi_funcion()
```



La variable resultado almacenará el valor devuelto por la función mi_funcion()

Funciones dinamicas

La integración de diferentes herramientas de control de flujo, nos permite crear funciones dinámicas y flexibles. Si debemos utilizarlas varias veces, lograremos un programa más limpio y sencillo de mantener, evitando repeticiones de código.

- Funciones
- Loops (for/while)
- Estructuras condicionales
- Palabras clave (return, break, continue, pass)

```
def mi_funcion(argumento):
    for item in ...:
        if a == b ...:
            ...
    else:
        return ...
    return ...
```

Interacción entre funciones

Las salidas de una determinada función pueden convertirse en entradas de otras funciones. De esa manera, cada función realiza una tarea definida, y el programa se construye a partir de la interacción entre funciones.

```
def funcion_1():
    ...
    return a

def funcion_2(a):
    ...
    return b

def funcion_3(b):
    ...
    return c

def funcion_4(a,c):
    ...
    return d
```

Librería PANDAS



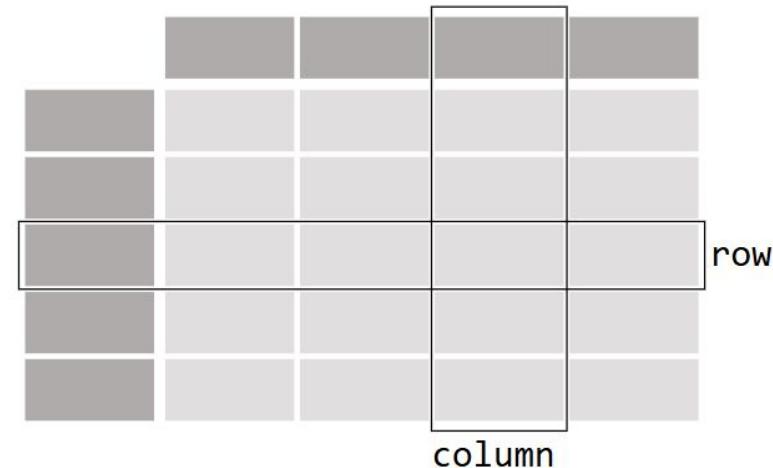
Quiero empezar a usar pandas

```
In [1]: import pandas as pd
```

Para cargar el paquete pandas y comenzar a trabajar con él, importe el archivo paquete. El alias acordado por la comunidad para los pandas es , por lo que cargando pandas, como se supone que es una práctica estándar para todos los pandas, documentación. [pd](#) [pd](#)

Representación de la tabla de datos de Pandas

DataFrame



Librería PANDAS



Quiero almacenar los datos de los pasajeros del Titanic. Para varios pasajeros, conozco los datos de nombre (caracteres), edad (enteros) y sexo (hombre/mujer).

```
In [2]: df = pd.DataFrame(  
...:     {  
...:         "Name": [  
...:             "Braund, Mr. Owen Harris",  
...:             "Allen, Mr. William Henry",  
...:             "Bonell, Miss. Elizabeth",  
...:         ],  
...:         "Age": [22, 35, 58],  
...:         "Sex": ["male", "male", "female"],  
...:     }  
...: )  
...:  
  
In [3]: df  
Out[3]:  
      Name  Age   Sex  
0 Braund, Mr. Owen Harris  22  male  
1 Allen, Mr. William Henry  35  male  
2 Bonell, Miss. Elizabeth  58 female
```

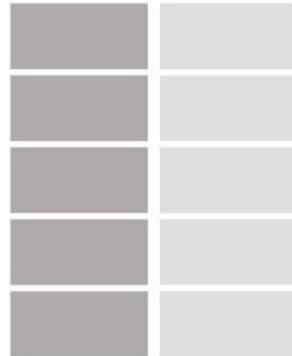
	B	C	D	E	F
1	Name	Age	Sex		
2	0 Braund, Mr. Owen Harris	22	male		
3	1 Allen, Mr. William Henry	35	male		
4	2 Bonell, Miss. Elizabeth	58	female		
5					
6					
7					
8					

Para almacenar datos manualmente en una tabla, cree un archivo . Cuando se utiliza un diccionario de listas de Python, las claves del diccionario se utilizarán como encabezados de columna y Los valores de cada lista como columnas de la extensión .[DataFrame](#) [DataFrame](#)

Librería PANDAS

Cada columna de un es un `DataFrame` `Series`

Series



Solo me interesa trabajar con los datos de la columna `Age`

```
In [4]: df["Age"]
Out[4]:
0    22
1    35
2    58
Name: Age, dtype: int64
```

Al seleccionar una sola columna de un pandas, el resultado es a los pandas. Para seleccionar la columna, use la etiqueta de columna en entre corchetes .`[]`

Librería PANDAS

Hacer algo con un DataFrame o Series



Quiero saber la edad máxima de los pasajeros

Podemos hacer esto en el seleccionando la columna y Aplicar: DataFrame Age max()

```
In [7]: df["Age"].max()  
Out[7]: 58
```

O a los : Series

```
In [8]: ages.max()  
Out[8]: 58
```

Como se ilustra en el método, puede *hacer* cosas con un o . Pandas proporciona una gran cantidad de funcionalidades, cada uno de ellos un *método* que se puede aplicar a un o . Como los métodos son funciones, no olvide usar paréntesis. max() DataFrame Series DataFrame Series ()

Librería PANDAS



I'm interested in some basic statistics of the numerical data of my data table

```
In [9]: df.describe()  
Out[9]:  
Age  
count    3.000000  
mean     38.333333  
std      18.230012  
min      22.000000  
25%     28.500000  
50%     35.000000  
75%     46.500000  
max      58.000000
```

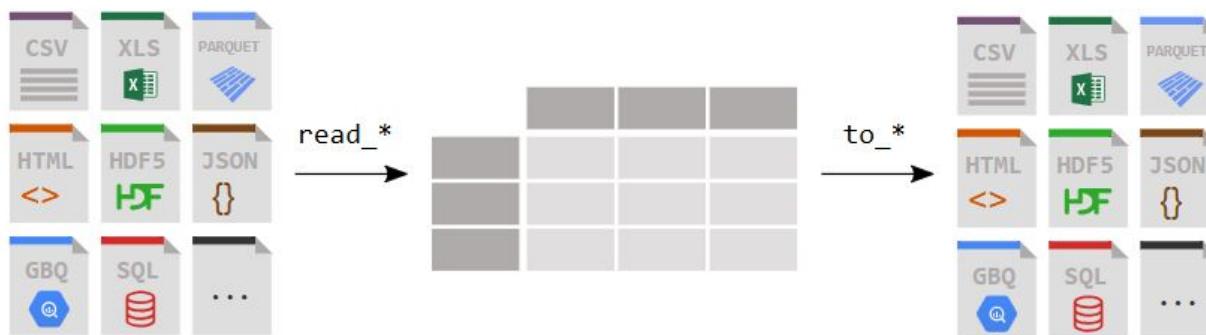


The [describe\(\)](#) method provides a quick overview of the numerical data in a . As the and columns are textual data, these are by default not taken into account by the [describe\(\)](#) method. [DataFrame](#) [Name](#) [Sex](#)

Many pandas operations return a or a . The [describe\(\)](#) method is an example of a pandas operation returning a pandas or a pandas . [DataFrame](#) [Series](#) [Series](#) [DataFrame](#)

Librería PANDAS

How do I read and write tabular data?



Librería PANDAS



I want to analyze the Titanic passenger data, available as a CSV file.

```
In [2]: titanic = pd.read_csv("data/titanic.csv")
```

pandas provides the `read_csv()` function to read data stored as a csv file into a pandas `DataFrame`. pandas supports many different file formats or data sources out of the box (csv, excel, sql, json, parquet, ...), each of them with the prefix `read_*`.

Make sure to always have a check on the data after reading in the data. When displaying a `DataFrame`, the first and last 5 rows will be shown by default:

```
In [3]: titanic
Out[3]:
   PassengerId  Survived  Pclass  ...      Fare Cabin Embarked
0            1         0       3  ...    7.2500   NaN        S
1            2         1       1  ...  71.2833   C85        C
2            3         1       3  ...    7.9250   NaN        S
3            4         1       1  ...  53.1000  C123        S
4            5         0       3  ...    8.0500   NaN        S
..          ...
886          887        0       2  ...  13.0000   NaN        S
887          888        1       1  ...  30.0000   B42        S
888          889        0       3  ...  23.4500   NaN        S
889          890        1       1  ...  30.0000  C148        C
890          891        0       3  ...   7.7500   NaN        Q
```

[891 rows x 12 columns]



I want to see the first 8 rows of a pandas DataFrame.

```
In [4]: titanic.head(8)
```

```
Out[4]:
```

	PassengerId	Survived	Pclass	...	Fare	Cabin	Embarked
0	1	0	3	...	7.2500	NaN	S
1	2	1	1	...	71.2833	C85	C
2	3	1	3	...	7.9250	NaN	S
3	4	1	1	...	53.1000	C123	S
4	5	0	3	...	8.0500	NaN	S
5	6	0	3	...	8.4583	NaN	Q
6	7	0	1	...	51.8625	E46	S
7	8	0	3	...	21.0750	NaN	S

[8 rows x 12 columns]

To see the first N rows of a `DataFrame`, use the `head()` method with the required number of rows (in this case 8) as argument.



Note

Interested in the last N rows instead? pandas also provides a `tail()` method. For example, `titanic.tail(10)` will return the last 10 rows of the DataFrame.

Librería PANDAS

A check on how pandas interpreted each of the column data types can be done by requesting the pandas `dtypes` attribute:

```
In [5]: titanic.dtypes
Out[5]:
PassengerId      int64
Survived         int64
Pclass           int64
Name             object
Sex              object
Age              float64
SibSp            int64
Parch            int64
Ticket           object
Fare              float64
Cabin            object
Embarked         object
dtype: object
```

For each of the columns, the used data type is enlisted. The data types in this `DataFrame` are integers (`int64`), floats (`float64`) and strings (`object`).

Librería PANDAS



My colleague requested the Titanic data as a spreadsheet.

```
In [6]: titanic.to_excel("titanic.xlsx", sheet_name="passengers", index=False)
```

Whereas `read_*` functions are used to read data to pandas, the `to_*` methods are used to store data. The `to_excel()` method stores the data as an excel file. In the example here, the `sheet_name` is named *passengers* instead of the default *Sheet1*. By setting `index=False` the row index labels are not saved in the spreadsheet.

The equivalent read function `read_excel()` will reload the data to a `DataFrame`:

Librería PANDAS



I'm interested in a technical summary of a `DataFrame`

```
In [9]: titanic.info()  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 891 entries, 0 to 890  
Data columns (total 12 columns):  
 #   Column      Non-Null Count  Dtype     
---  --          --           --         
 0   PassengerId 891 non-null    int64    
 1   Survived     891 non-null    int64    
 2   Pclass       891 non-null    int64    
 3   Name         891 non-null    object    
 4   Sex          891 non-null    object    
 5   Age          714 non-null    float64   
 6   SibSp        891 non-null    int64    
 7   Parch        891 non-null    int64    
 8   Ticket       891 non-null    object    
 9   Fare          891 non-null    float64   
 10  Cabin         204 non-null    object    
 11  Embarked      889 non-null    object    
dtypes: float64(2), int64(5), object(5)  
memory usage: 83.7+ KB
```

The method `info()` provides technical information about a `DataFrame`, so let's explain the output in more detail:

- It is indeed a `DataFrame`.
- There are 891 entries, i.e. 891 rows.
- Each row has a row label (aka the `index`) with values ranging from 0 to 890.
- The table has 12 columns. Most columns have a value for each of the rows (all 891 values are `non-null`). Some columns do have missing values and less than 891 `non-null` values.
- The columns `Name`, `Sex`, `Cabin` and `Embarked` consists of textual data (strings, aka `object`). The other columns are numerical data with some of them whole numbers (aka `integer`) and others are real numbers (aka `float`).
- The kind of data (characters, integers,...) in the different columns are summarized by listing the `dtypes`.
- The approximate amount of RAM used to hold the DataFrame is provided as well.

Matplotlib



Matplotlib

```
pip install matplotlib
```

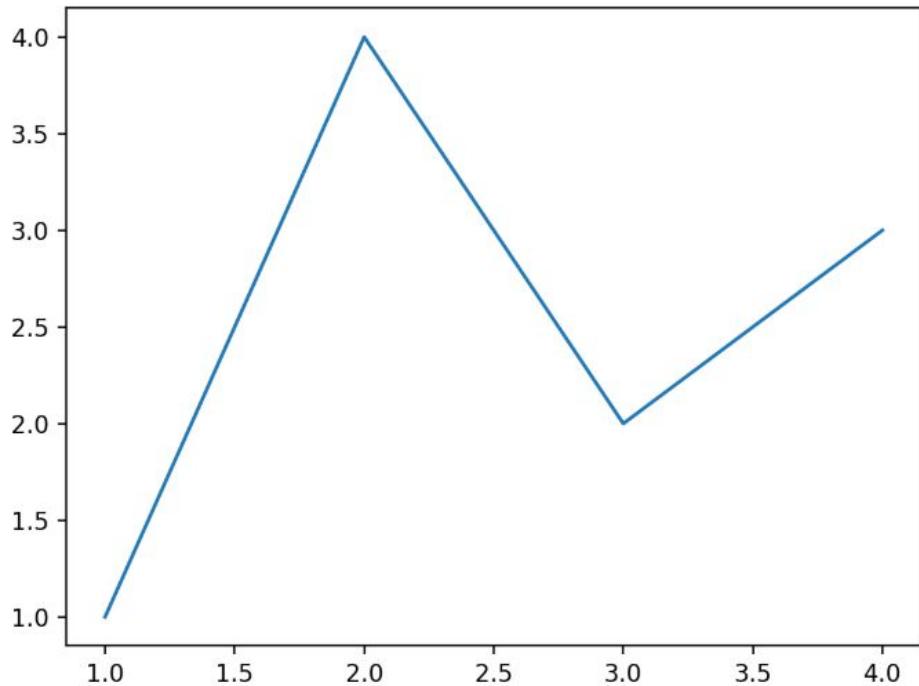
```
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-packages (3.10.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (4.59.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (1.4.9)
Requirement already satisfied: numpy>=1.23 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (25.0)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib) (1.17.0)
```

Matplotlib

```
import matplotlib.pyplot as plt  
import numpy as np
```



```
fig, ax = plt.subplots() # Create a figure containing a single Axes.  
ax.plot([1, 2, 3, 4], [1, 4, 2, 3]) # Plot some data on the Axes.  
plt.show() # Show the figure.
```



Matplotlib representa gráficamente sus datos en Figuras (por ejemplo, ventanas, widgets de Jupyter, etc.), cada una de las cuales puede contener uno o más Ejes, un área donde los puntos pueden especificarse en términos de coordenadas x-y (o theta-r en un gráfico polar, x-y-z en un gráfico 3D, etc.). La forma más sencilla de crear una Figura con un Eje es utilizando `pyplot.subplots`. Luego podemos usar `Axes.plot` para dibujar algunos datos en los Ejes, y `show` para mostrar la figura:

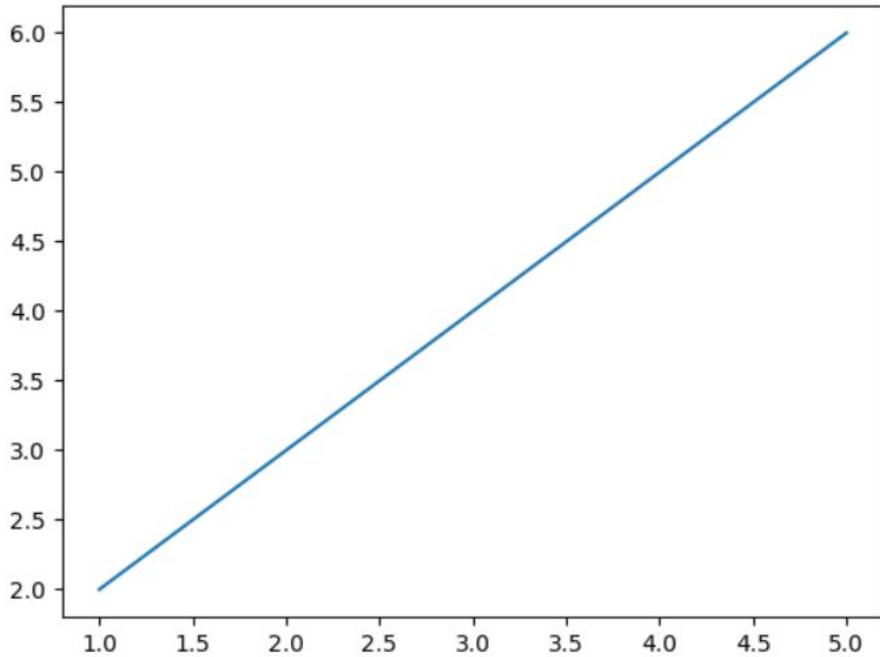
Matplotlib

```
X = [1,2,3,4,5]  
Y = [2,3,4,5,6]
```

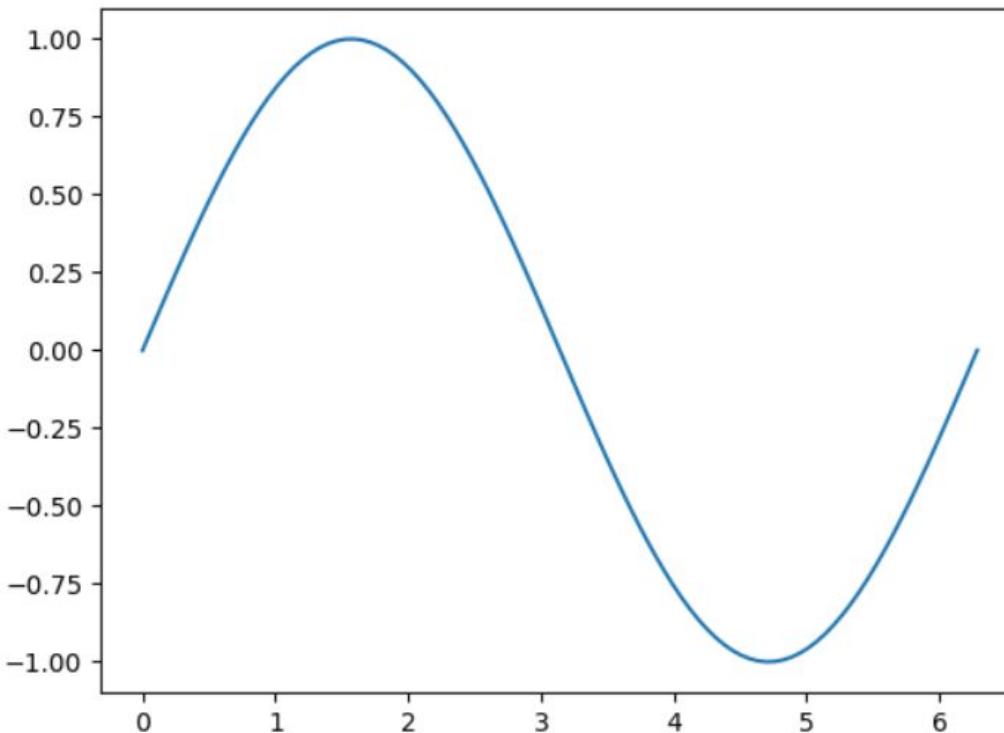
[3]

```
> plt.plot(X,Y)  
plt.show()
```

[6]



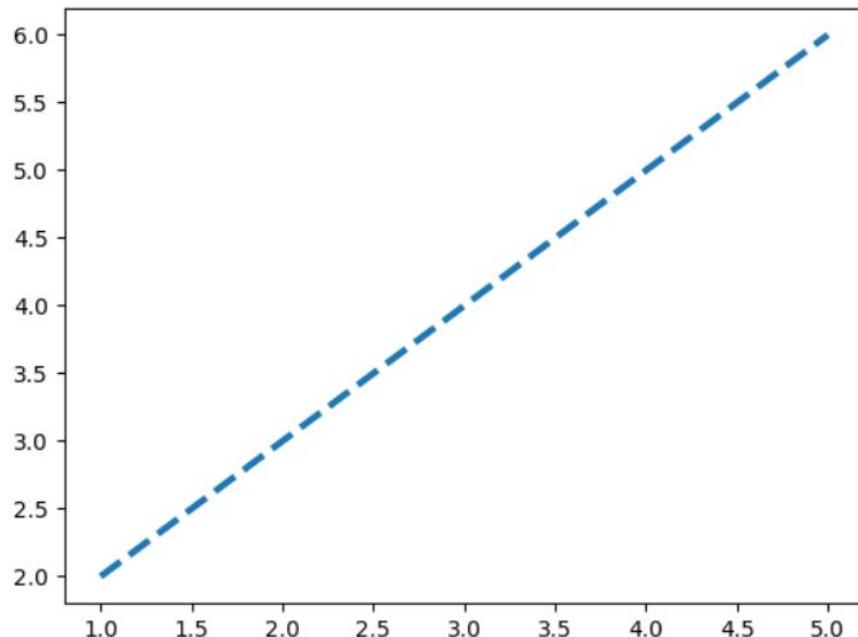
```
x = np.linspace(0, 2*np.pi, 100)  
y = np.sin(x)  
plt.plot(x,y)  
plt.show()
```



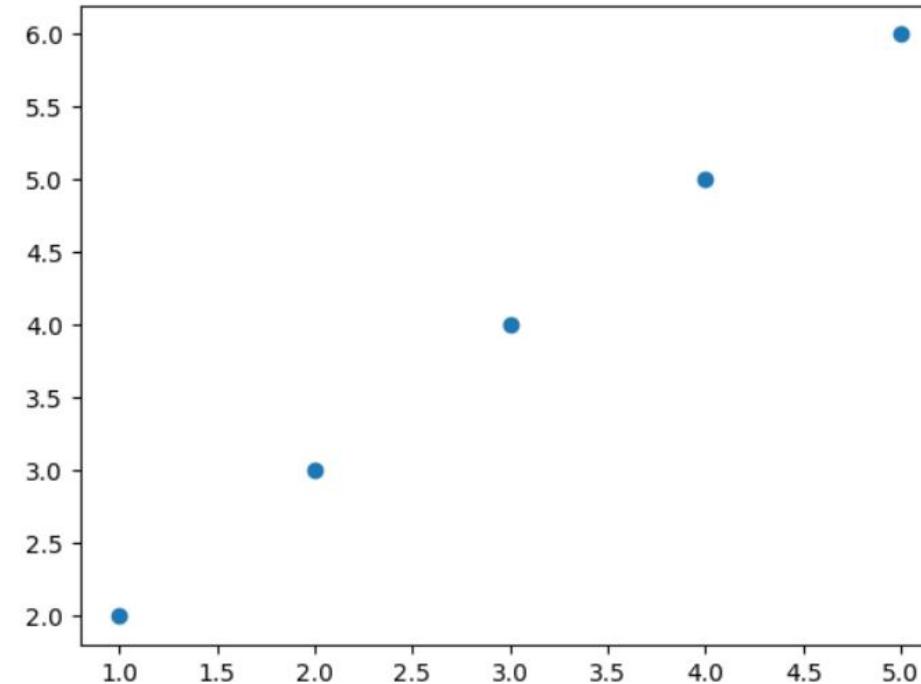
Matplotlib

Formato de grafico

```
X = [1,2,3,4,5]  
Y = [2,3,4,5,6]  
  
plt.plot(X,Y,'--',linewidth=3)  
plt.show()
```

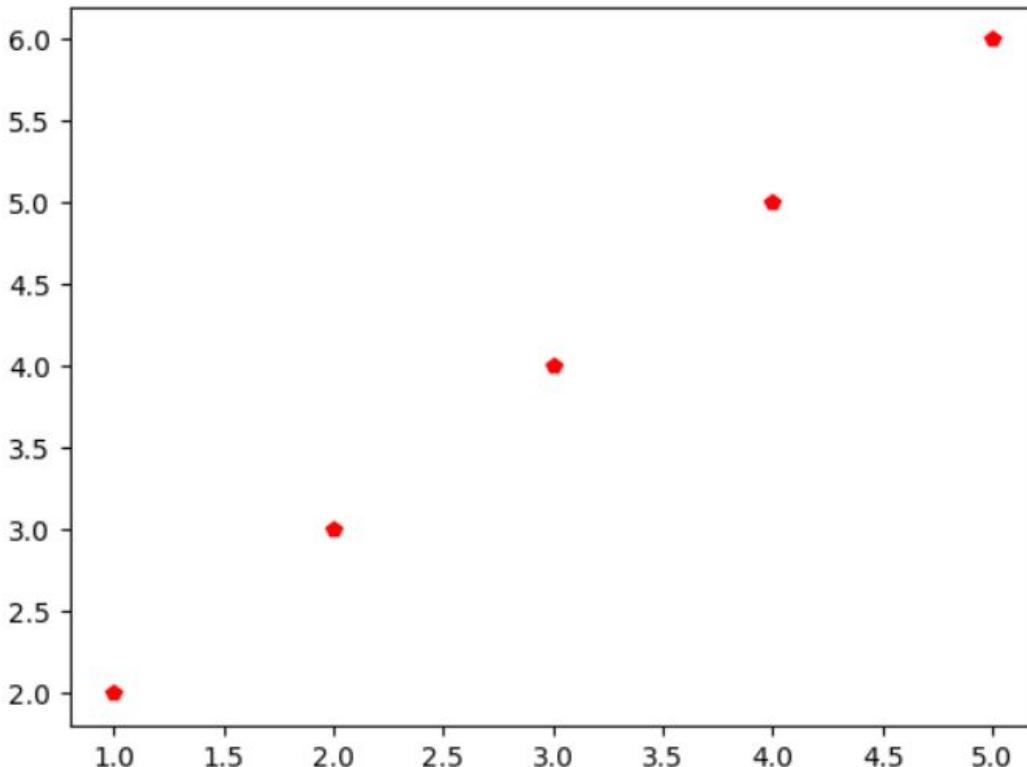


```
X = [1,2,3,4,5]  
Y = [2,3,4,5,6]  
  
plt.plot(X,Y,'o',linewidth=3)  
plt.show()
```

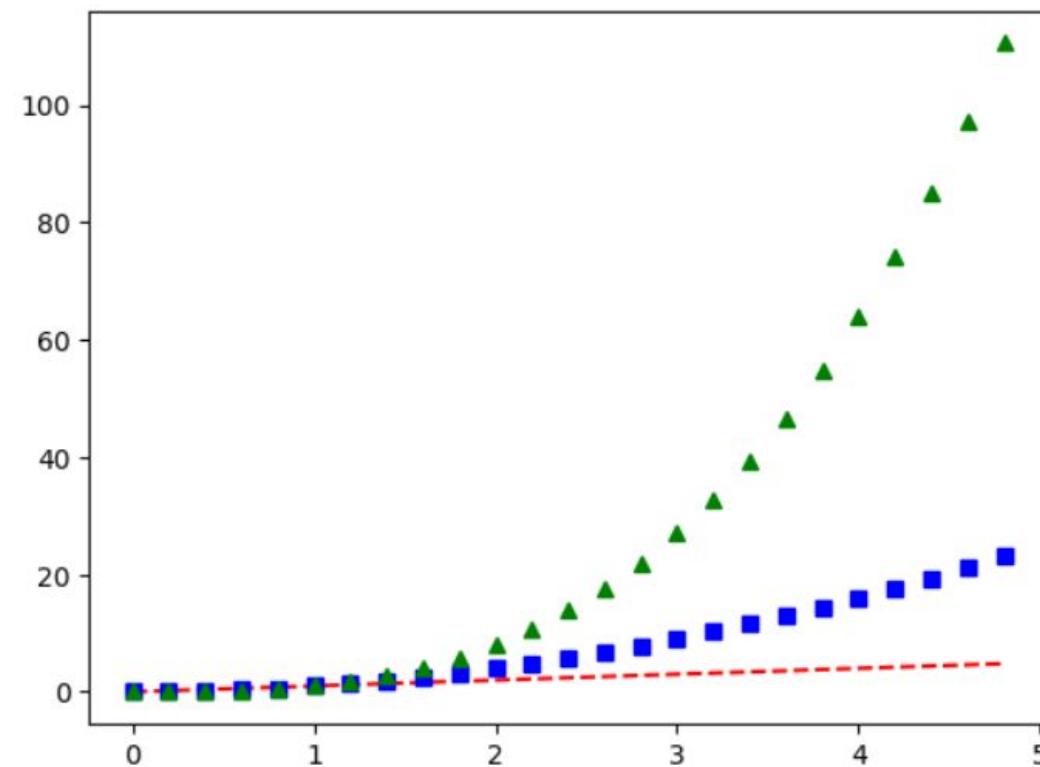


Matplotlib

```
X = [1,2,3,4,5]  
Y = [2,3,4,5,6]  
  
plt.plot(X,Y,'pr',linewidth=3)  
plt.show()
```



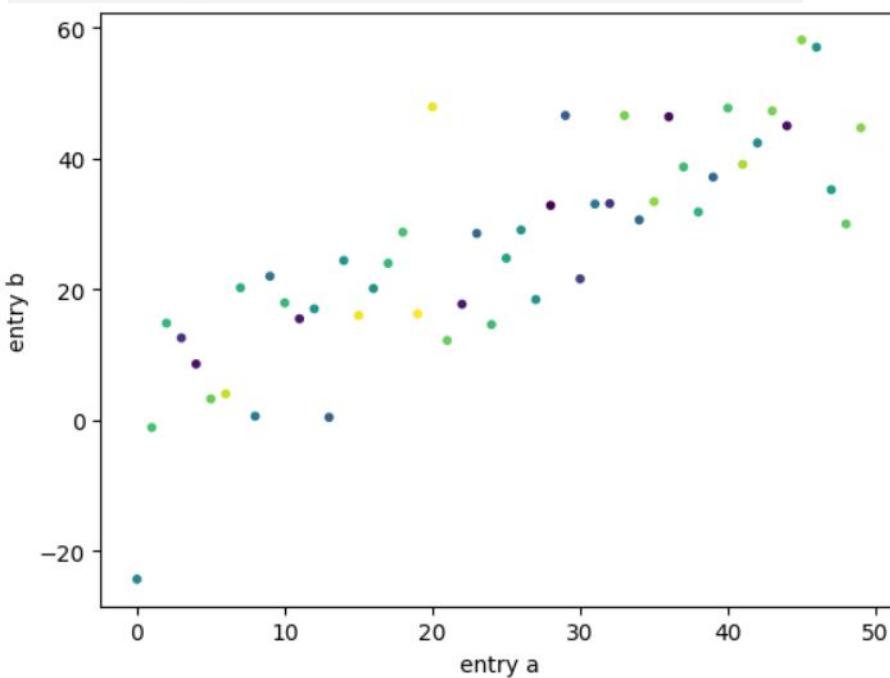
```
t = np.arange(0,5,0.2)  
t2 = t**2  
t3 = t**3  
plt.plot(t,t,"r--",t,t2,"bs",t,t3,"g^" )  
plt.show()
```



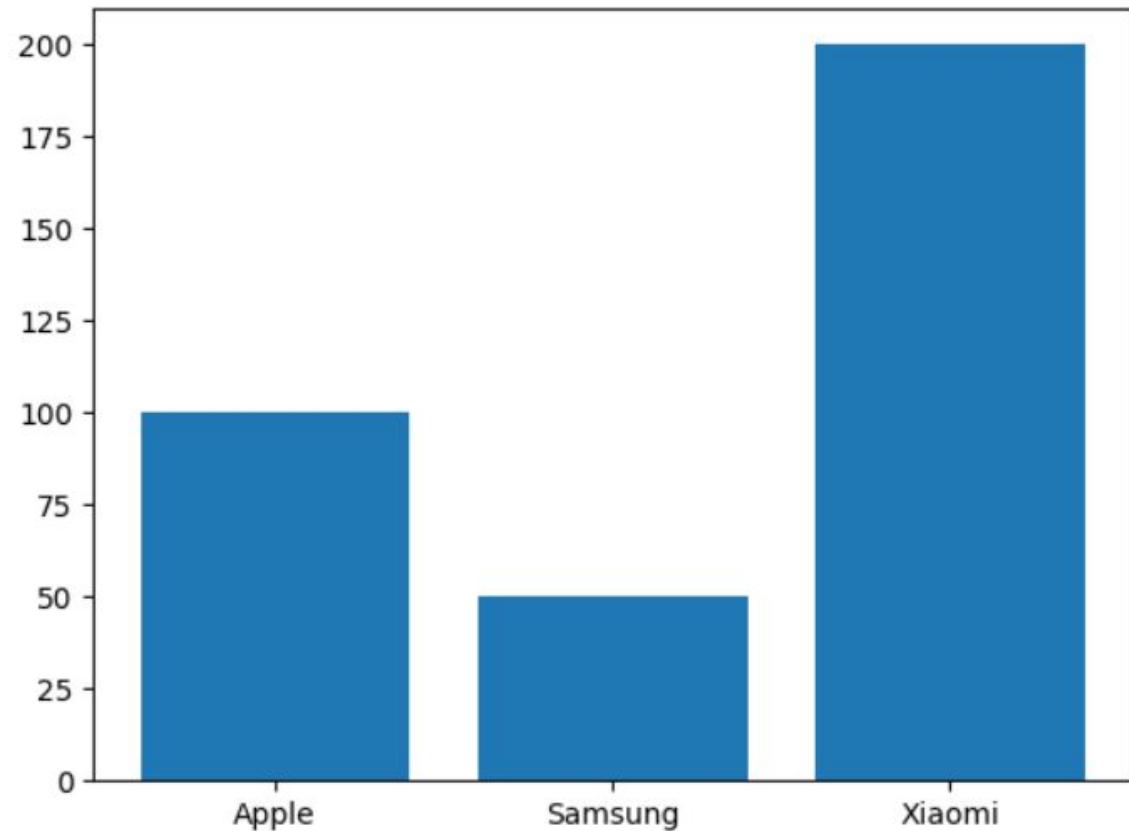
Matplotlib

Grafico de dispersion

```
datos = {  
    'x' : np.arange(50),  
    'c' : np.random.randint(0,50,50),  
    'd' : np.random.randint(0,50)}  
  
datos['b'] = datos['x'] + 10 * np.random.randn(50)  
datos['d'] = np.abs(datos['d'])  
  
plt.scatter('x','b',c='c',s='d',data=datos)  
plt.xlabel('entry a')  
plt.ylabel('entry b')  
plt.show()
```



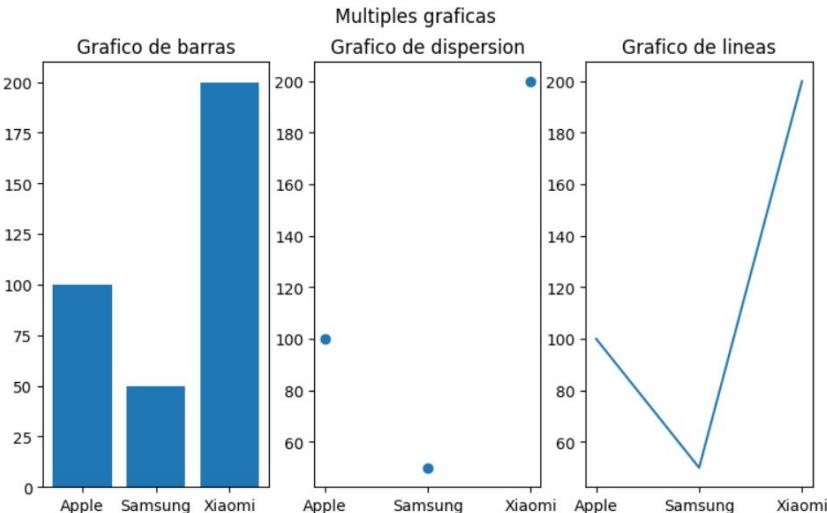
```
Nombres = ["Apple","Samsung","Xiaomi"]  
valores = [100,50,200]  
plt.bar(Nombres,valores)  
plt.show()
```



Matplotlib

Multiples figuras en una sola grafica

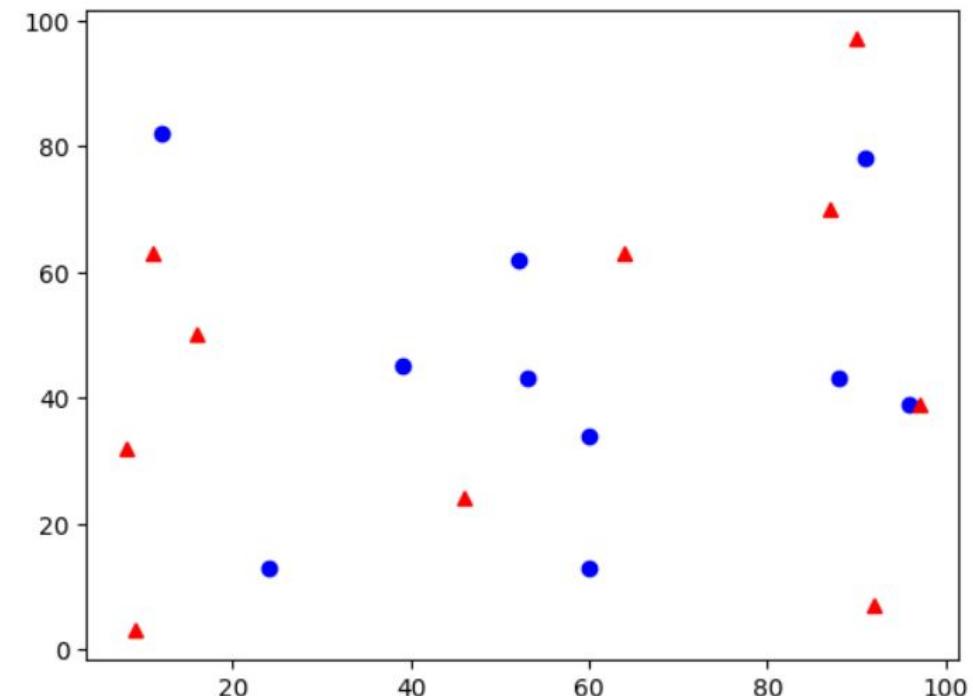
```
Nombres = ["Apple","Samsung","Xiaomi"]
valores = [100,50,200]
plt.figure(figsize=(9,5))
plt.subplot(131)
plt.bar(Nombres,valores)
plt.title("Grafico de barras")
plt.subplot(132)
plt.scatter(Nombres,valores)
plt.title("Grafico de dispersion")
plt.subplot(133)
plt.plot(Nombres,valores)
plt.title("Grafico de lineas")
plt.suptitle('Multiples graficas')
plt.show()
```



```
x1 = np.random.randint(low=1,high=100,size=10)
y1 = np.random.randint(low=1,high=100,size=10)
plt.plot(x1,y1,'bo')
```

```
x2 = np.random.randint(low=1,high=100,size=10)
y2 = np.random.randint(low=1,high=100,size=10)
plt.plot(x2,y2,'r^')
```

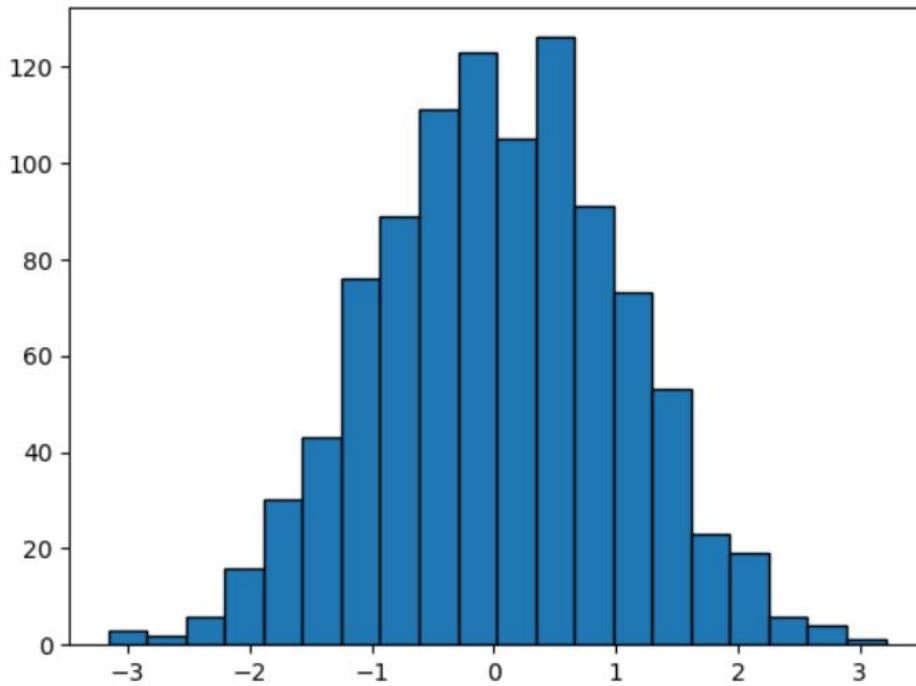
```
plt.show()
```



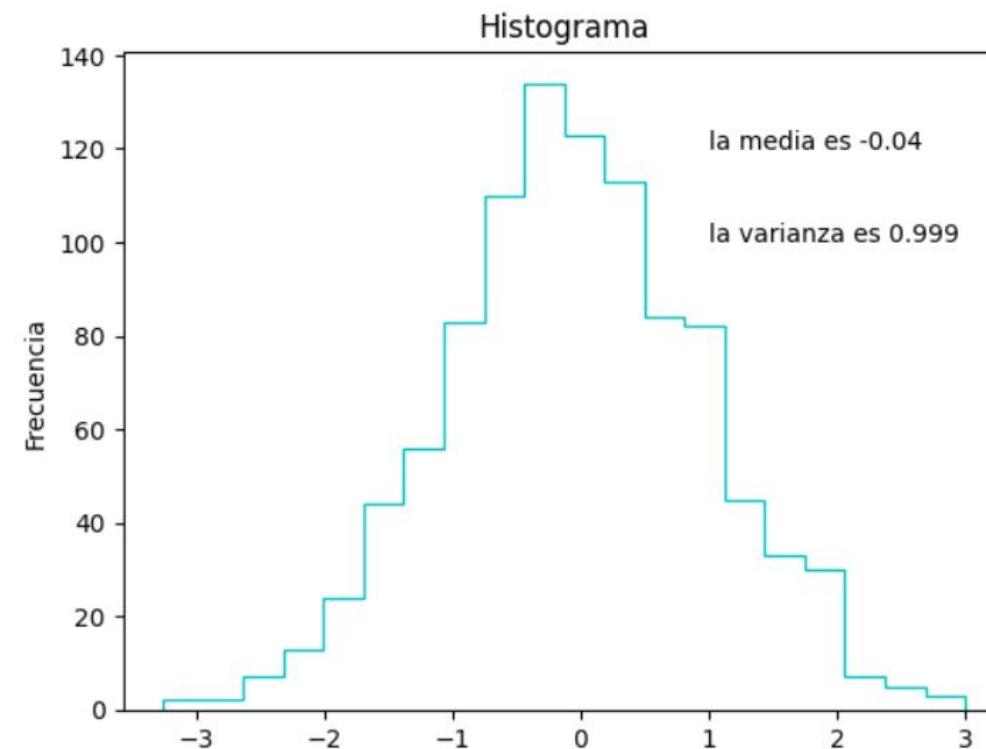
Matplotlib

Histogramas

```
x = np.random.randn(1000)
plt.hist(x,bins=20,edgecolor='black')
plt.show()
```

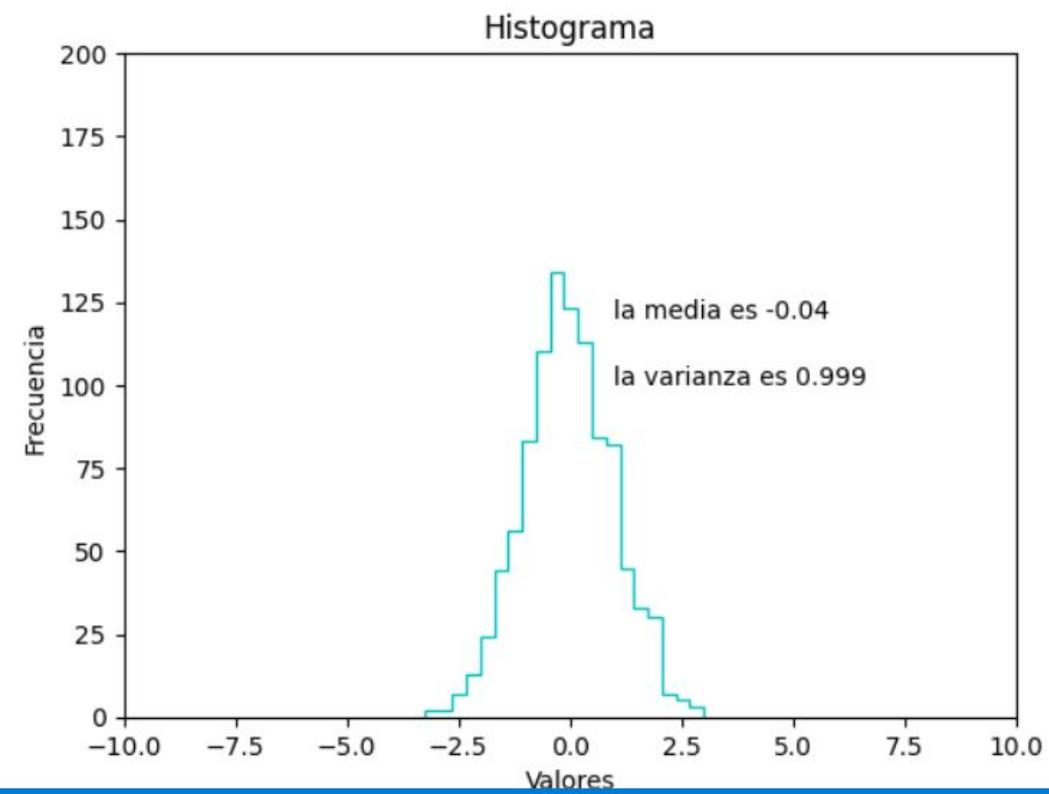


```
media = np.round(np.mean(x),2)
varianza = np.round(np.var(x),3)
plt.hist(x,bins=20,histtype='step',color='c')
plt.xlabel('Valores')
plt.ylabel('Frecuencia')
plt.title('Histograma')
plt.text(1,120,f"la media es {media}")
plt.text(1,100,f"la varianza es {varianza}")
plt.show()
```

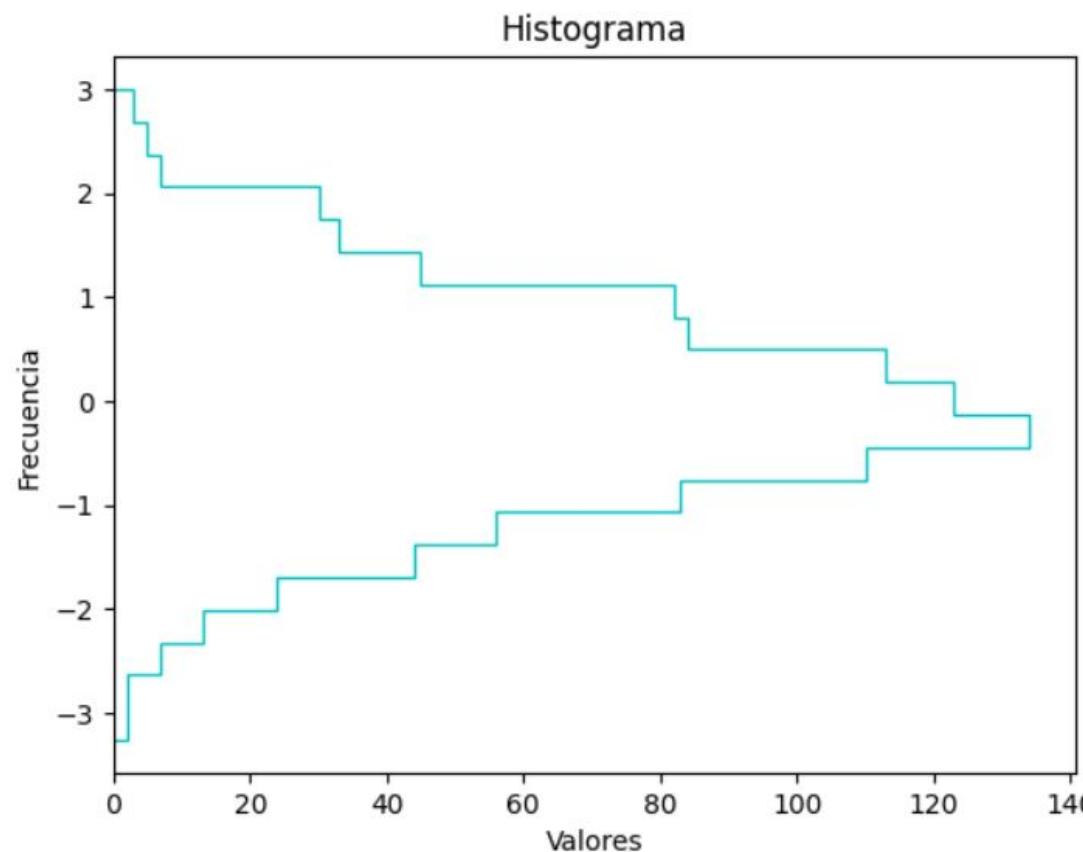


Matplotlib

```
media = np.round(np.mean(x),2)
varianza = np.round(np.var(x),3)
plt.hist(x,bins=20,histtype='step',color='c')
plt.xlabel('Valores')
plt.ylabel('Frecuencia')
plt.title('Histograma')
plt.text(1,120,f"la media es {media}")
plt.text(1,100,f"la varianza es {varianza}")
plt.axis([-10,10,0,200])
plt.show()
```



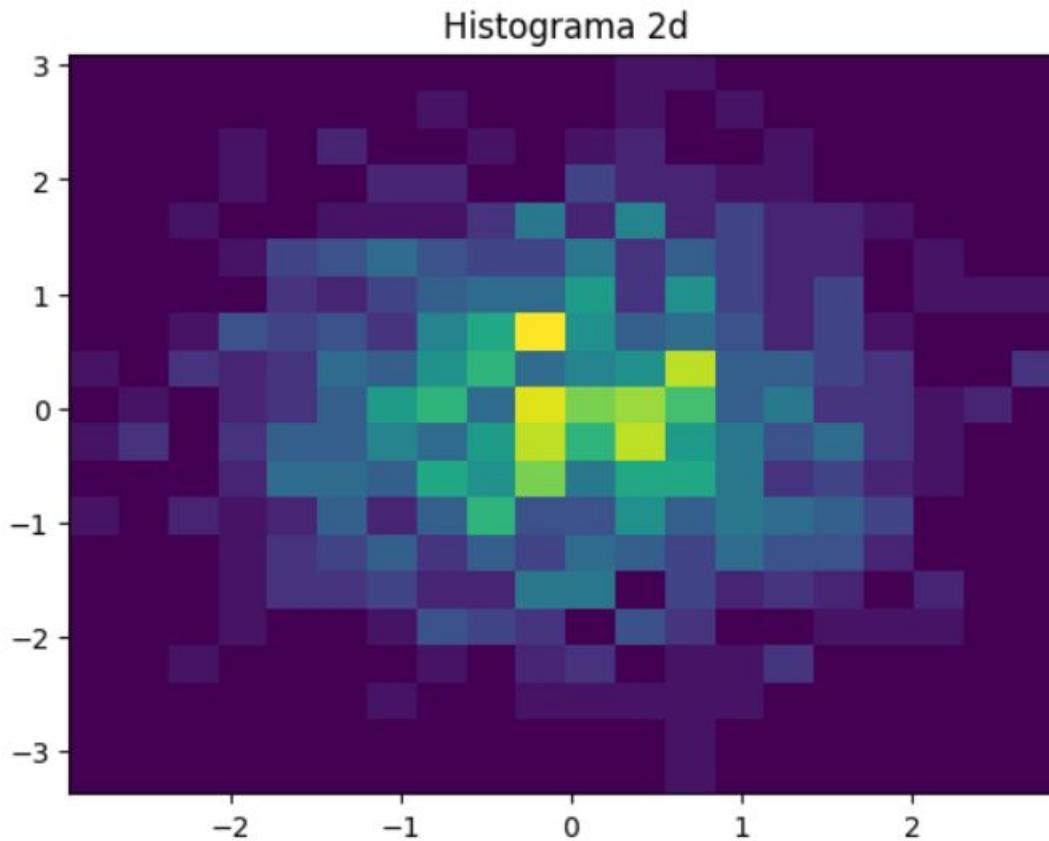
```
media = np.round(np.mean(x),2)
varianza = np.round(np.var(x),3)
plt.hist(x,bins=20,histtype='step',color='c',orientation='horizontal')
plt.xlabel('Valores')
plt.ylabel('Frecuencia')
plt.title('Histograma')
plt.show()
```



Matplotlib

Histograma en 2 dimensiones

```
valores_x = np.random.randn(1000)
valores_y = np.random.randn(1000)
plt.hist2d(valores_x,valores_y,bins=20)
plt.title('Histograma 2d')
plt.show()
#
```



Matplotlib

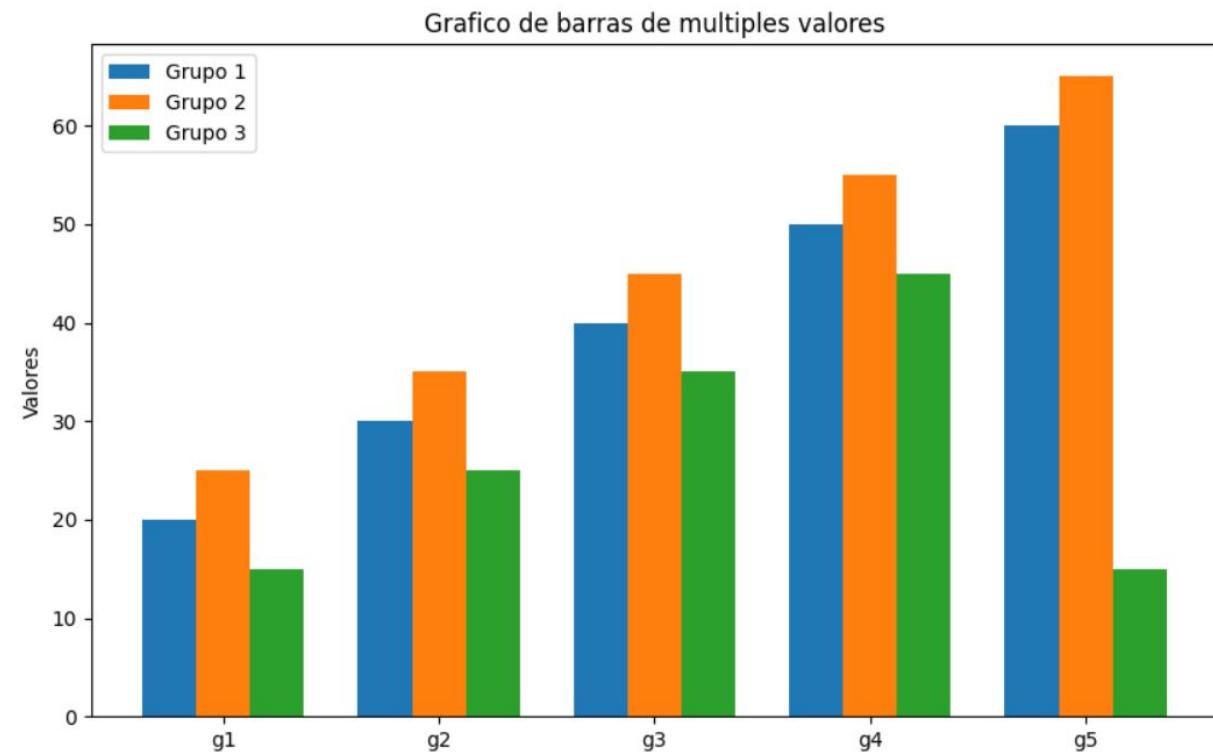
```
etiquetas = ['g1','g2','g3','g4','g5']
valores_1 = [20,30,40,50,60]
valores_2 = [25,35,45,55,65]
valores_3 = [15,25,35,45,15]

ancho = 0.25
posiciones = np.arange(len(etiquetas)) # crear las posiciones de las barras

plt.figure(figsize=(10,6))

plt.bar(posiciones-ancho,valores_1,width=ancho,label='Grupo 1')
plt.bar(posiciones,valores_2,width=ancho,label='Grupo 2')
plt.bar(posiciones+ancho,valores_3,width=ancho,label='Grupo 3')

plt.ylabel('Valores')
plt.title('Grafico de barras de multiples valores')
plt.xticks(posiciones,etiquetas)
plt.legend()
plt.show()
```



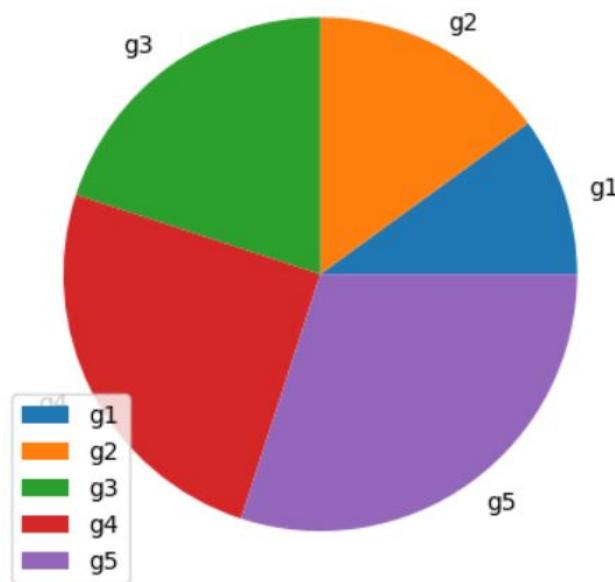
Graficos Circulares

▷ ▾

```
valores = [20,30,40,50,60]
etiquetas = ['g1','g2','g3','g4','g5']
plt.pie(valores,labels=etiquetas)
plt.legend()
plt.show()
```

[94]

...



```
x = [0.2,0.1,0.5,0.2]
x1 = [0.2,0.1,0.5]
plt.pie(x)
plt.show()

plt.pie(x1)
plt.show()
```

[94]

...



Matplotlib

```
fig,axes = plt.subplots(2,2,figsize=(8,8),layout='constrained')

t = np.arange(200)
x= np.cumsum(np.random.randn(200))

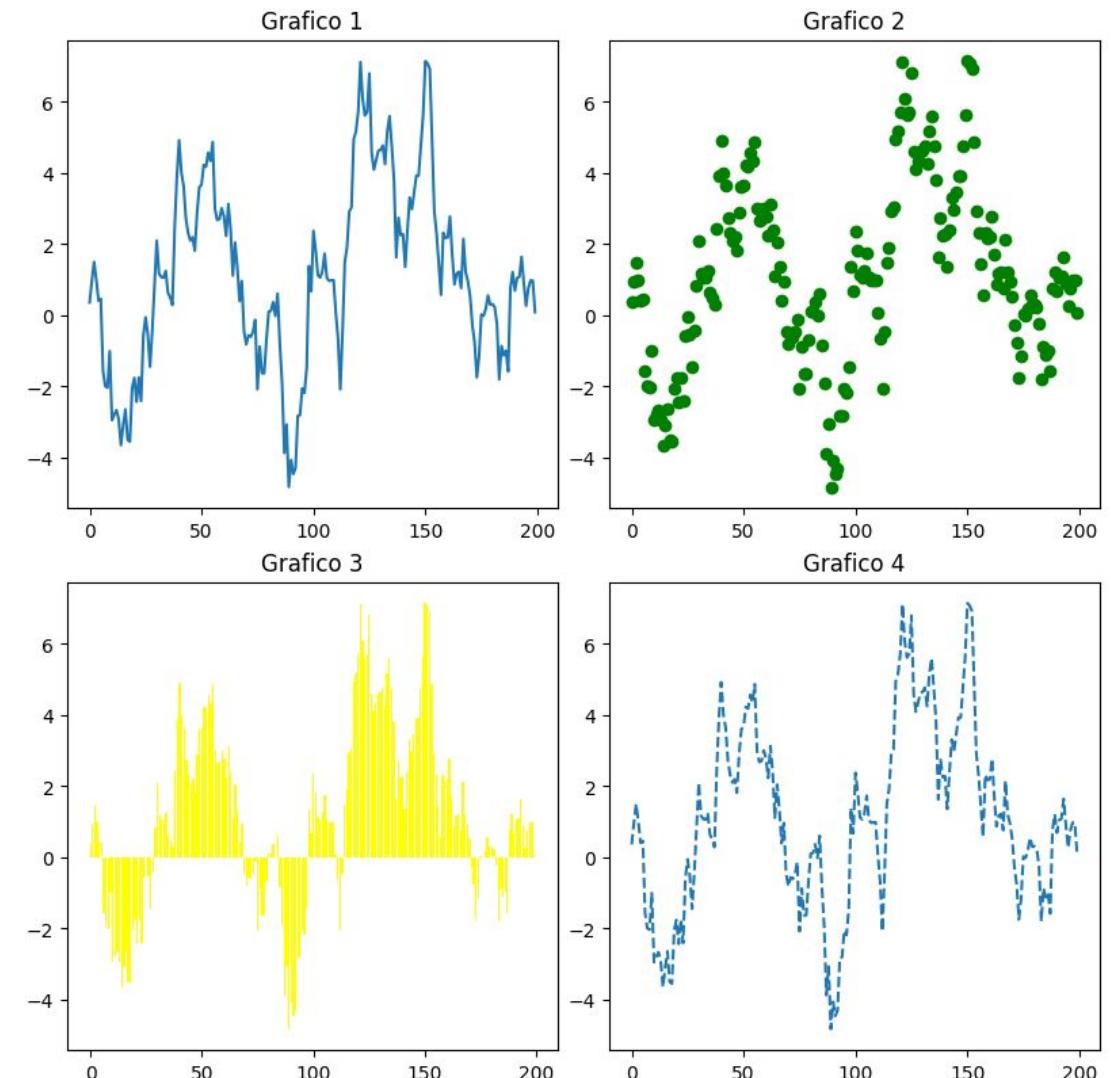
ejes = axes.flatten()

ejes[0].plot(t,x)
ejes[0].set_title('Grafico 1')

ejes[1].plot(t,x,'o',color='green')
ejes[1].set_title('Grafico 2')

ejes[2].bar(t,x,color='yellow')
ejes[2].set_title('Grafico 3')

ejes[3].plot(t,x,'--')
ejes[3].set_title('Grafico 4')
plt.show()
```



Matplotlib

Ejercicio graficar las funciones seno coseno y tangente en una sola figura

```

# --- 1. Preparación de los datos ---
# Crear un rango de valores para el eje x
x = np.linspace(-2 * np.pi, 2 * np.pi, 400)

# Calcular los valores de las funciones
y_sin = np.sin(x)
y_cos = np.cos(x)
y_tan = np.tan(x)

# --- 2. Creación de subgráficos ---
# Crear una figura con 3 filas y 1 columna de subgráficos
fig, axs = plt.subplots(nrows=3, ncols=1, figsize=(8, 10), sharex=True)
fig.suptitle('Funciones Trigonométricas en Subgráficos', fontsize=16)

# --- 3. Graficar en cada subgráfico ---
# Graficar la función seno en el primer subgráfico (eje superior)
axs[0].plot(x, y_sin, color='blue')
axs[0].set_title('sin(x)')
axs[0].grid(True, linestyle='--', alpha=0.6)
axs[0].axhline(0, color='black', linewidth=0.8, linestyle='--')

# Graficar la función coseno en el segundo subgráfico (eje central)
axs[1].plot(x, y_cos, color='red')
axs[1].set_title('cos(x)')
axs[1].grid(True, linestyle='--', alpha=0.6)
axs[1].axhline(0, color='black', linewidth=0.8, linestyle='--')

# Graficar la función tangente en el tercer subgráfico (eje inferior)
axs[2].plot(x, y_tan, color='green')
axs[2].set_title('tan(x)')
axs[2].set_ylim(-2.5, 2.5) # Limitar el eje Y para la tangente
axs[2].grid(True, linestyle='--', alpha=0.6)
axs[2].axhline(0, color='black', linewidth=0.8, linestyle='--')

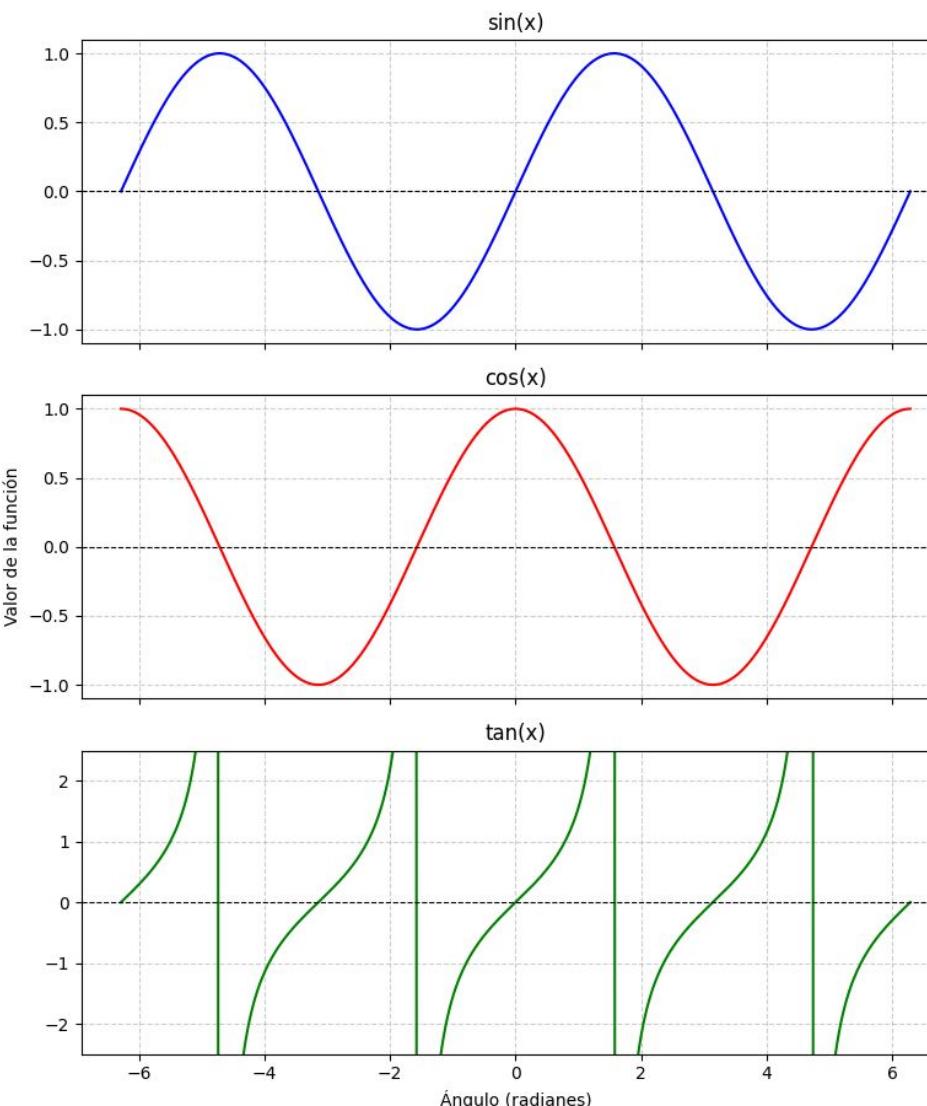
# Añadir etiquetas de eje Y
axs[1].set_ylabel('Valor de la función')
axs[2].set_xlabel('Ángulo (radianes)')

# Ajustar el diseño para evitar que los títulos se superpongan
plt.tight_layout(rect=[0, 0, 1, 0.96])

# Mostrar el gráfico
plt.show()

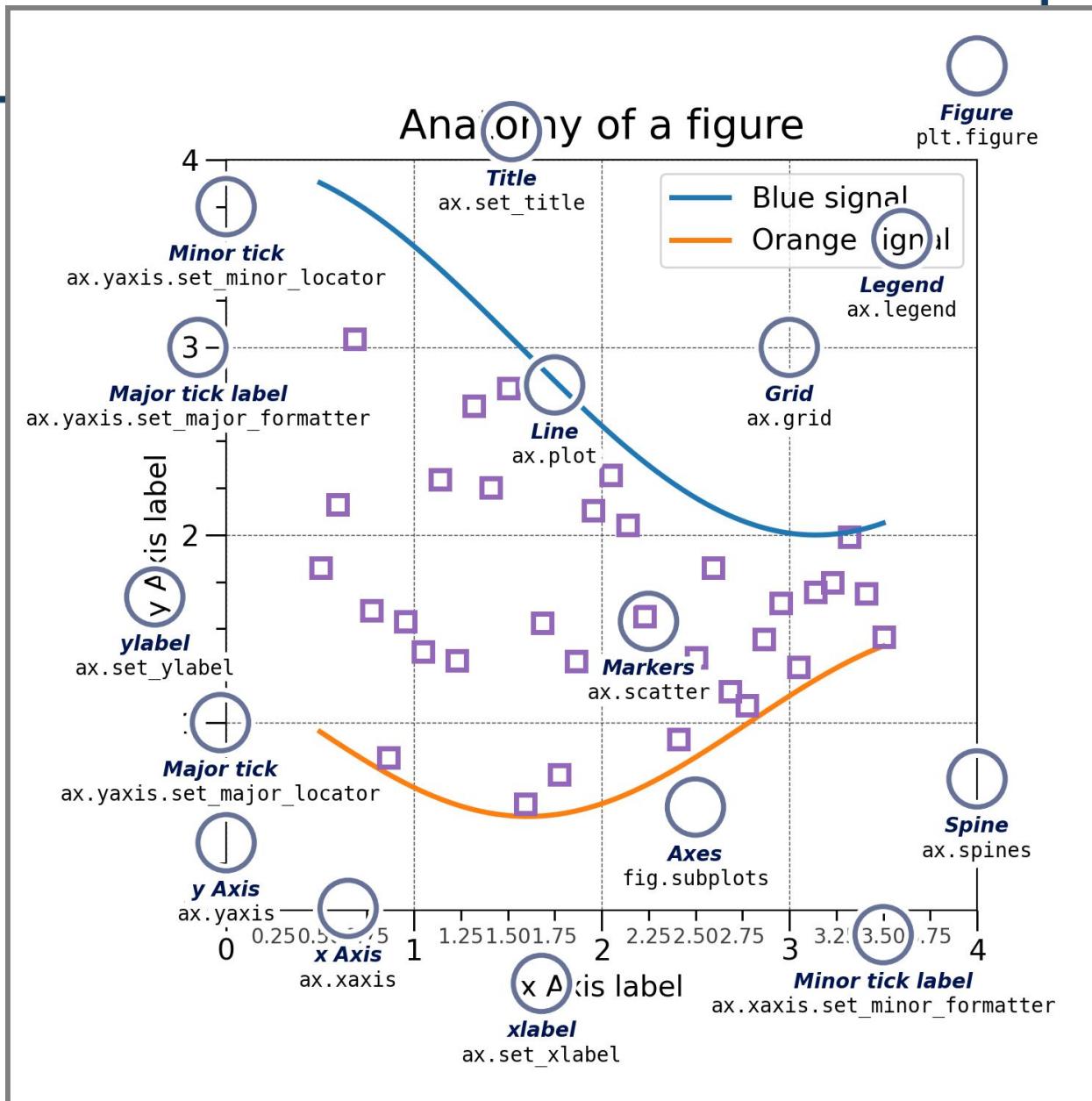
```

Funciones Trigonométricas en Subgráficos



Matplotlib

PARTES DE UNA FIGURA EN MATPLOTLIB



Matplotlib

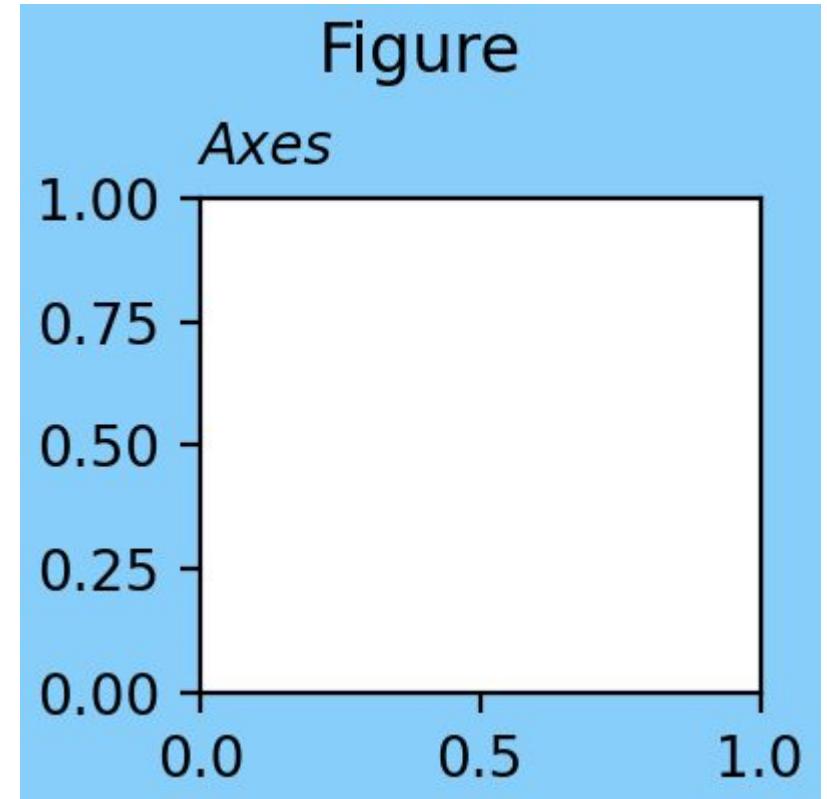
La figura completa. La Figura mantiene un registro de todos los Ejes hijos, un grupo de Artistas "especiales" (títulos, leyendas de figuras, barras de color, etc.), e incluso subfiguras anidadas. Normalmente, creará una nueva Figura a través de una de las siguientes funciones:

```
fig = plt.figure()          # an empty figure with no Axes
fig, ax = plt.subplots()    # a figure with a single Axes
fig, axs = plt.subplots(2, 2) # a figure with a 2x2 grid of Axes
# a figure with one Axes on the left, and two on the right:
fig, axs = plt.subplot_mosaic([('left', 'right_top'),
                               ['left', 'right_bottom'])]
```

Matplotlib Figuras

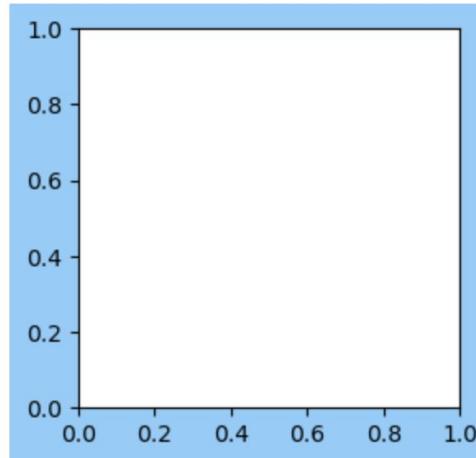
```
fig = plt.figure(figsize=(2, 2), facecolor='lightskyblue',
                 layout='constrained')
fig.suptitle('Figure')
ax = fig.add_subplot()
ax.set_title('Axes', loc='left', fontstyle='oblique', fontsize='medium')
```

Cuando se mira una visualización Matplotlib, casi siempre se está mirando a los **Artistas** colocados en una Figura. En el ejemplo anterior, la figura es la región azul y `add_subplot` ha añadido un Artista de Ejes a la Figura (ver Partes de una Figura). Una visualización más complicada puede añadir múltiples Ejes a la Figura, barras de color, leyendas, anotaciones, y los propios Ejes pueden tener múltiples Artistas añadidos a ellos (por ejemplo `ax.plot` o `ax.imshow`).



Matplotlib Figuras

```
In [1]: import matplotlib.pyplot as plt  
In [2]: fig, ax = plt.subplots(facecolor='lightskyblue',  
                           figsize=(3,3))
```



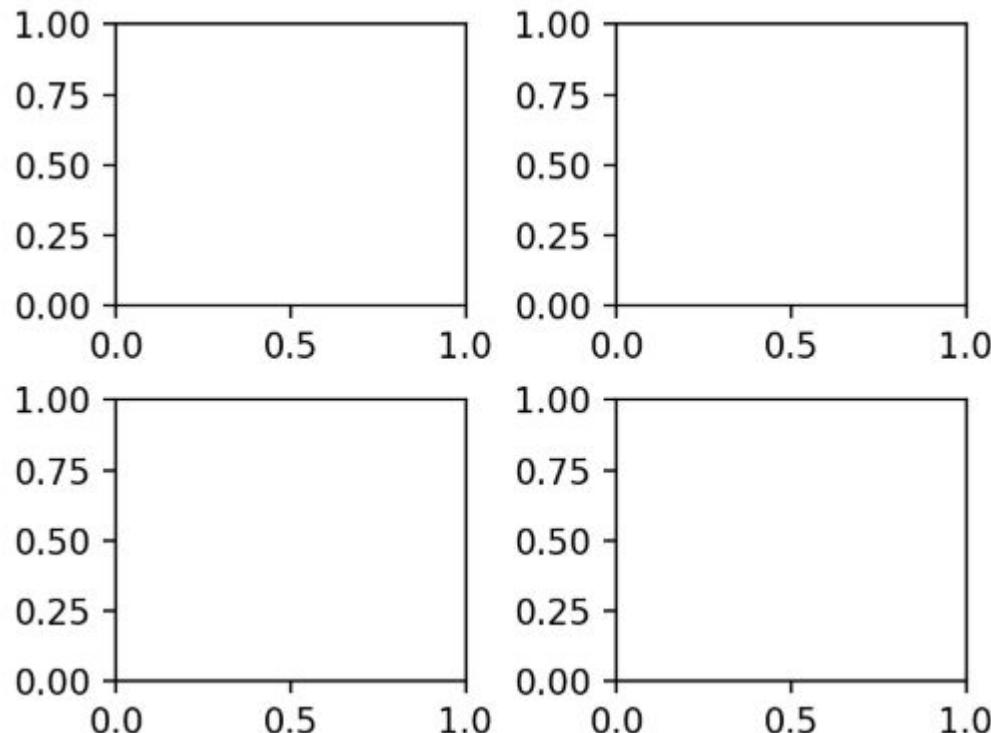
```
In [ ]:
```

la forma más común de crear una figura es utilizando la interfaz pyplot. Como se indica en Matplotlib Application Interfaces (APIs), la interfaz pyplot sirve para dos propósitos.

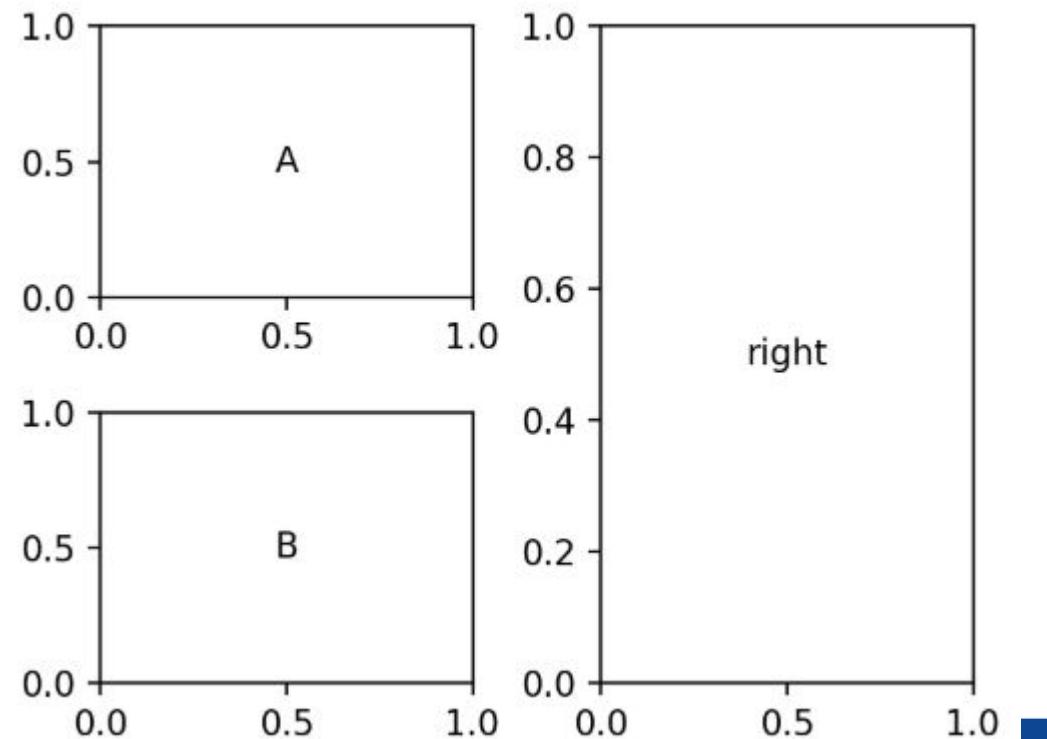
En el ejemplo anterior, usamos pyplot para el primer propósito, y creamos el objeto **Figura, fig**. Como efecto secundario, fig también se añade al estado global de pyplot, y se puede acceder a él a través de gcf. Los usuarios normalmente quieren un Eje o una rejilla de Ejes cuando crean una Figura, así que además de figure, hay métodos de conveniencia que devuelven tanto una Figura como algunos Ejes. Una simple rejilla de Ejes se puede conseguir con pyplot.subplots (que simplemente envuelve a Figure.subplots):

Matplotlib Figuras

```
fig, axs = plt.subplots(2, 2, figsize=(4, 3), layout='constrained')
```



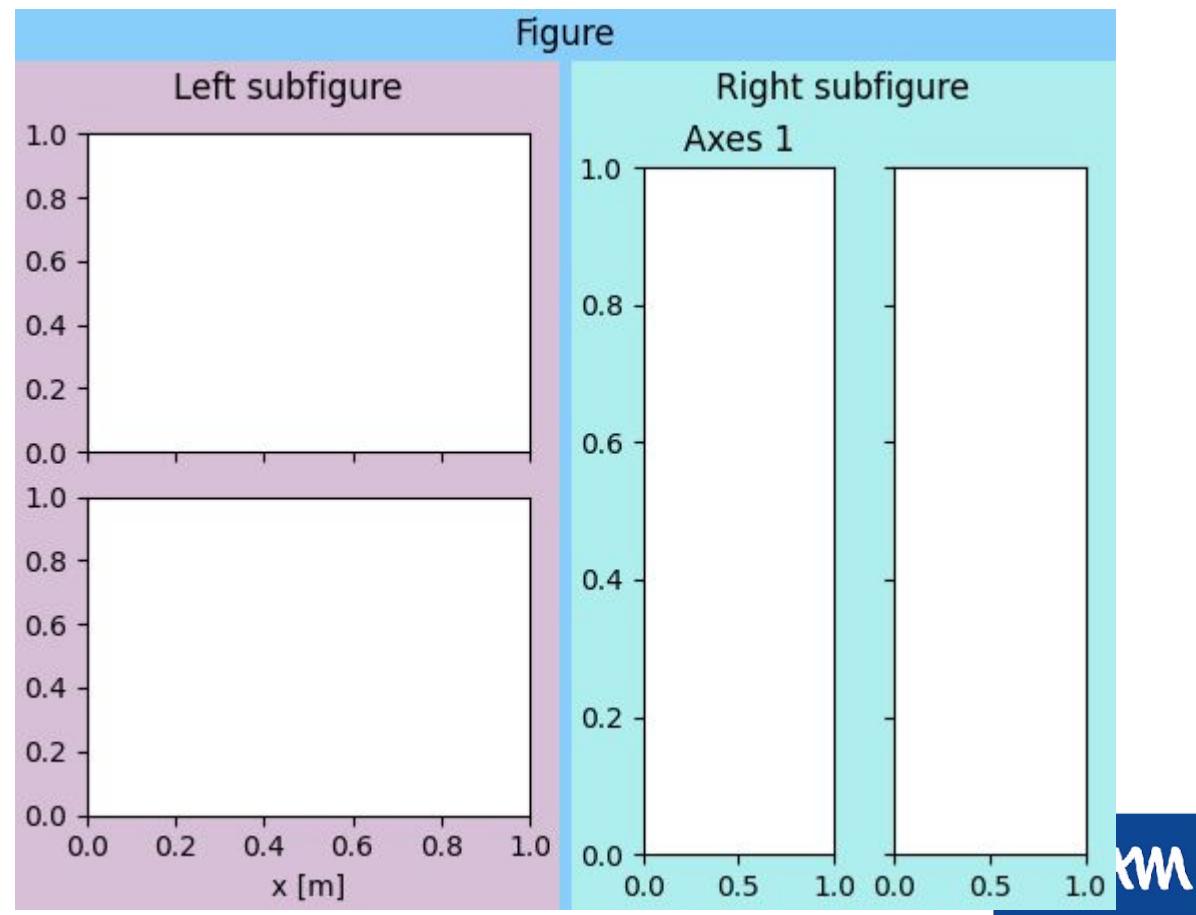
```
fig, axs = plt.subplot_mosaic([('A', 'right'), ('B', 'right')],  
                             figsize=(4, 3), layout='constrained')  
for ax_name, ax in axs.items():  
    ax.text(0.5, 0.5, ax_name, ha='center', va='center')
```



Matplotlib Figuras

A veces queremos tener un diseño anidado en una Figura, con dos o más conjuntos de Ejes que no comparten la misma rejilla de subparcela. Podemos **usar add_subfigure o subfigures** para crear figuras virtuales dentro de una Figura padre; ver Subfigures de Figura para más detalles.

```
fig = plt.figure(layout='constrained', facecolor='lightskyblue')
fig.suptitle('Figure')
figL, figR = fig.subfigures(1, 2)
figL.set_facecolor('thistle')
axL = figL.subplots(2, 1, sharex=True)
axL[1].set_xlabel('x [m]')
figL.suptitle('Left subfigure')
figR.set_facecolor('paleturquoise')
axR = figR.subplots(1, 2, sharey=True)
axR[0].set_title('Axes 1')
figR.suptitle('Right subfigure')
```



Matplotlib Figuras

Añadir Artistas

La clase Figura tiene varios métodos para añadir artistas a una Figura o a una SubFigura.

Con mucho, los más comunes son para añadir Ejes de varias configuraciones (**add_axes**, **add_subplot**, **subplots**, **subplot_mosaic**) y subfiguras (**subfigures**).

Las barras de color se añaden a los Ejes o grupos de Ejes a nivel de Figura (**colorbar**). También es posible tener una leyenda a nivel de Figura (**legend**).

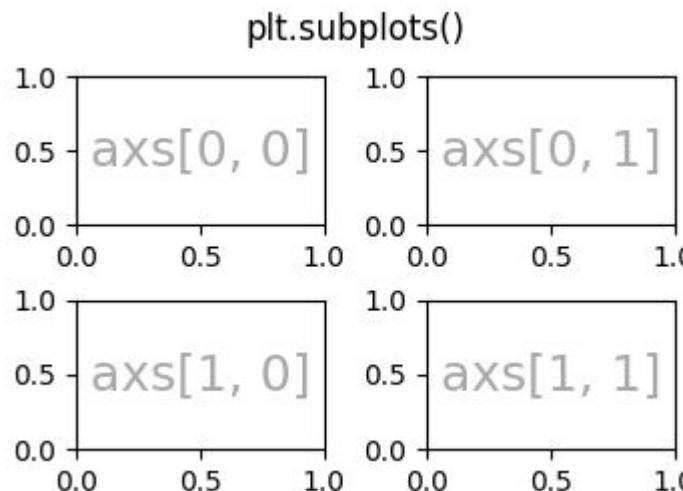
Otros Artistas incluyen etiquetas a nivel de figura (**suptitle**, **supxlabel**, **supylabel**) y texto (**text**).

Por último, los Artistas de bajo nivel pueden añadirse directamente utilizando **add_artist**, normalmente teniendo cuidado de utilizar la transformación adecuada. Normalmente, éstas incluyen **Figure.transFigure** que va de 0 a 1 en cada dirección, y representa la fracción del tamaño actual de la Figura, o **Figure.dpi_scale_trans** que estará en unidades físicas de pulgadas desde la esquina inferior izquierda de la Figura (ver Tutorial de Transformaciones para más detalles).

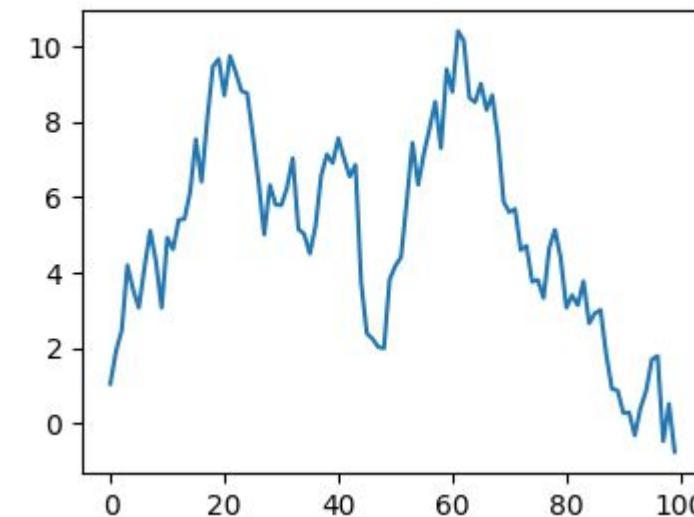
Matplotlib

```
import matplotlib.pyplot as plt
import numpy as np

fig, axs = plt.subplots(ncols=2, nrows=2, figsize=(3.5, 2.5),
                      layout="constrained")
# for each Axes, add an artist, in this case a nice label in the middle...
for row in range(2):
    for col in range(2):
        axs[row, col].annotate(f'axs[{row}, {col}]', (0.5, 0.5),
                               transform=axs[row, col].transAxes,
                               ha='center', va='center', fontsize=18,
                               color='darkgrey')
fig.suptitle('plt.subplots()')
```



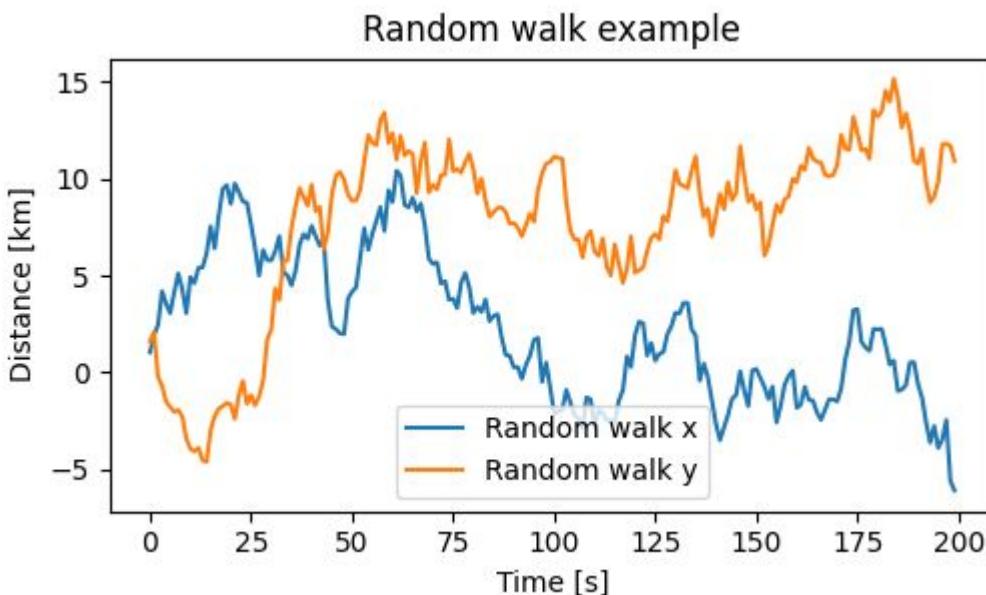
```
fig, ax = plt.subplots(figsize=(4, 3))
np.random.seed(19680801)
t = np.arange(100)
x = np.cumsum(np.random.randn(100))
lines = ax.plot(t, x)
```



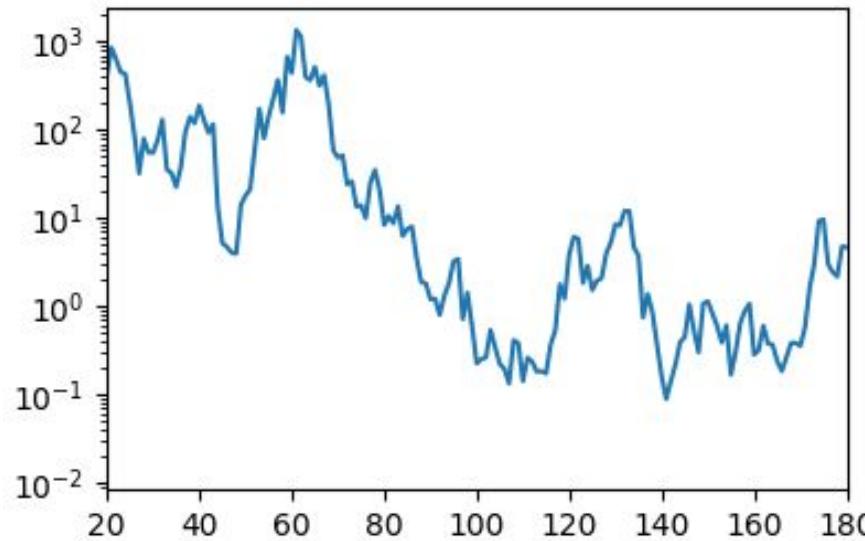
Matplotlib - Etiquetado y anotación de ejes

```
fig, ax = plt.subplots(figsize=(5, 3), layout='constrained')
np.random.seed(19680801)
t = np.arange(200)
x = np.cumsum(np.random.randn(200))
y = np.cumsum(np.random.randn(200))
linesx = ax.plot(t, x, label='Random walk x')
linesy = ax.plot(t, y, label='Random walk y')

ax.set_xlabel('Time [s]')
ax.set_ylabel('Distance [km]')
ax.set_title('Random walk example')
ax.legend()
```

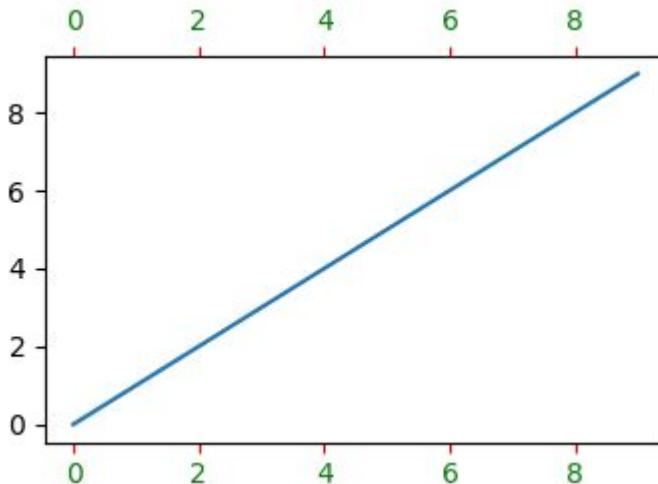


```
fig, ax = plt.subplots(figsize=(4, 2.5), layout='constrained')
np.random.seed(19680801)
t = np.arange(200)
x = 2**np.cumsum(np.random.randn(200))
linesx = ax.plot(t, x)
ax.set_yscale('log')
ax.set_xlim([20, 180])
```



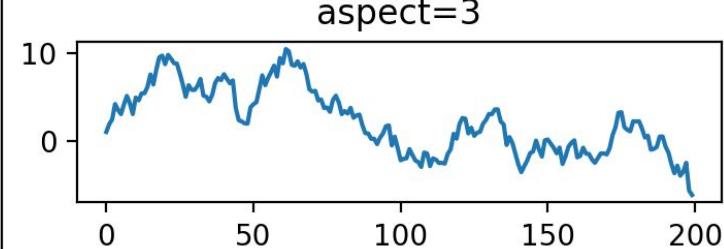
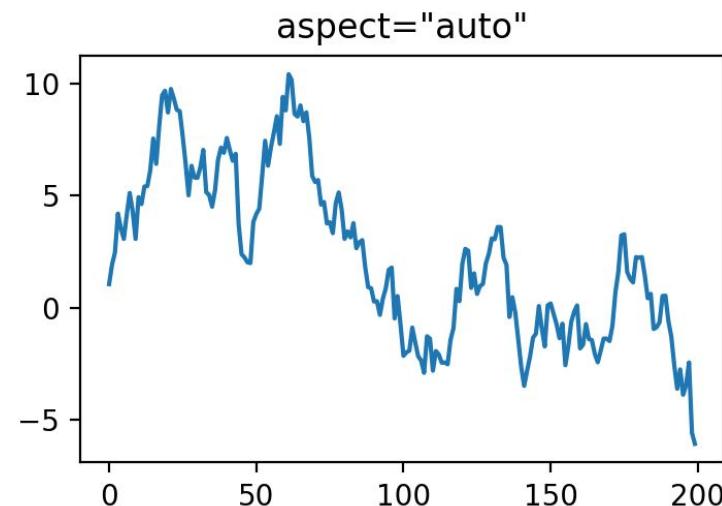
Matplotlib - Etiquetado y anotación de ejes

```
fig, ax = plt.subplots(figsize=(4, 2.5))
ax.plot(np.arange(10))
ax.tick_params(top=True, labeltop=True, color='red', axis='x',
               labelcolor='green')
```



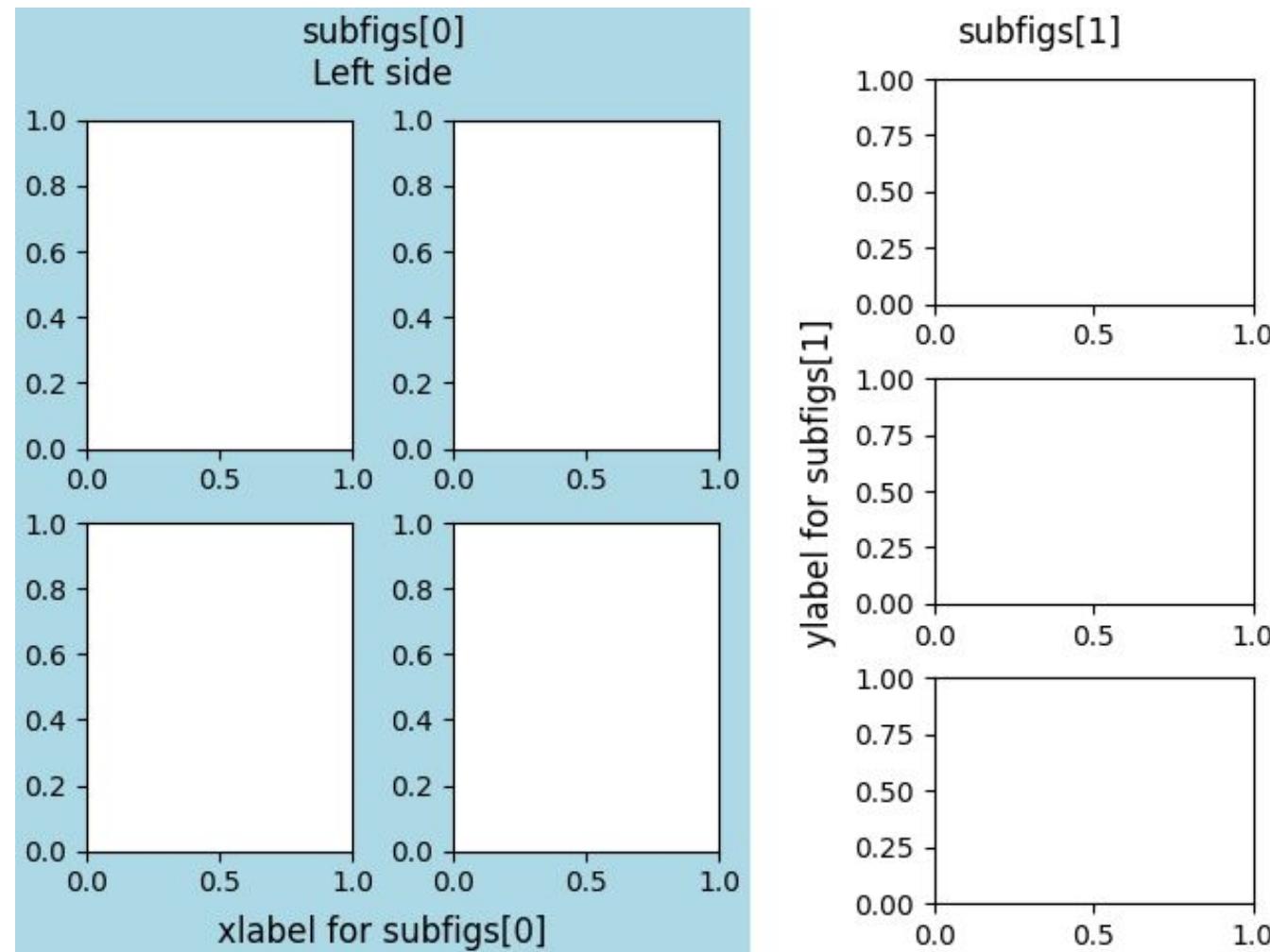
```
fig, axes = plt.subplots(ncols=2, figsize=(7, 2.5), layout='constrained')
np.random.seed(19680801)
t = np.arange(200)
x = np.cumsum(np.random.randn(200))
axes[0].plot(t, x)
axes[0].set_title('aspect="auto"')

axes[1].plot(t, x)
axes[1].set_aspect(3)
axes[1].set_title('aspect=3')
```



Matplotlib - Ejercicio

Crear la siguiente figura

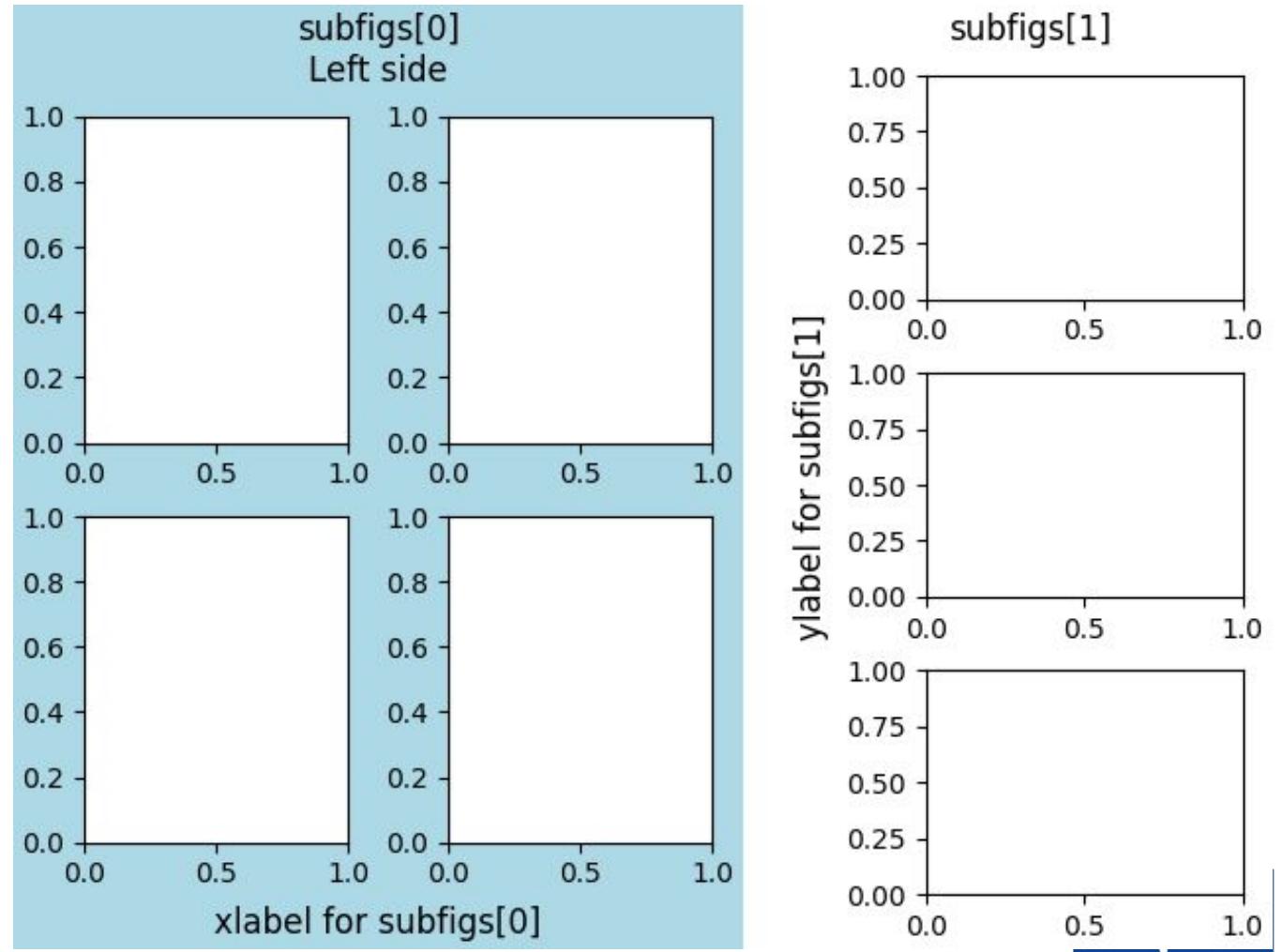


Matplotlib - Ejercicio

Crear la siguiente figura

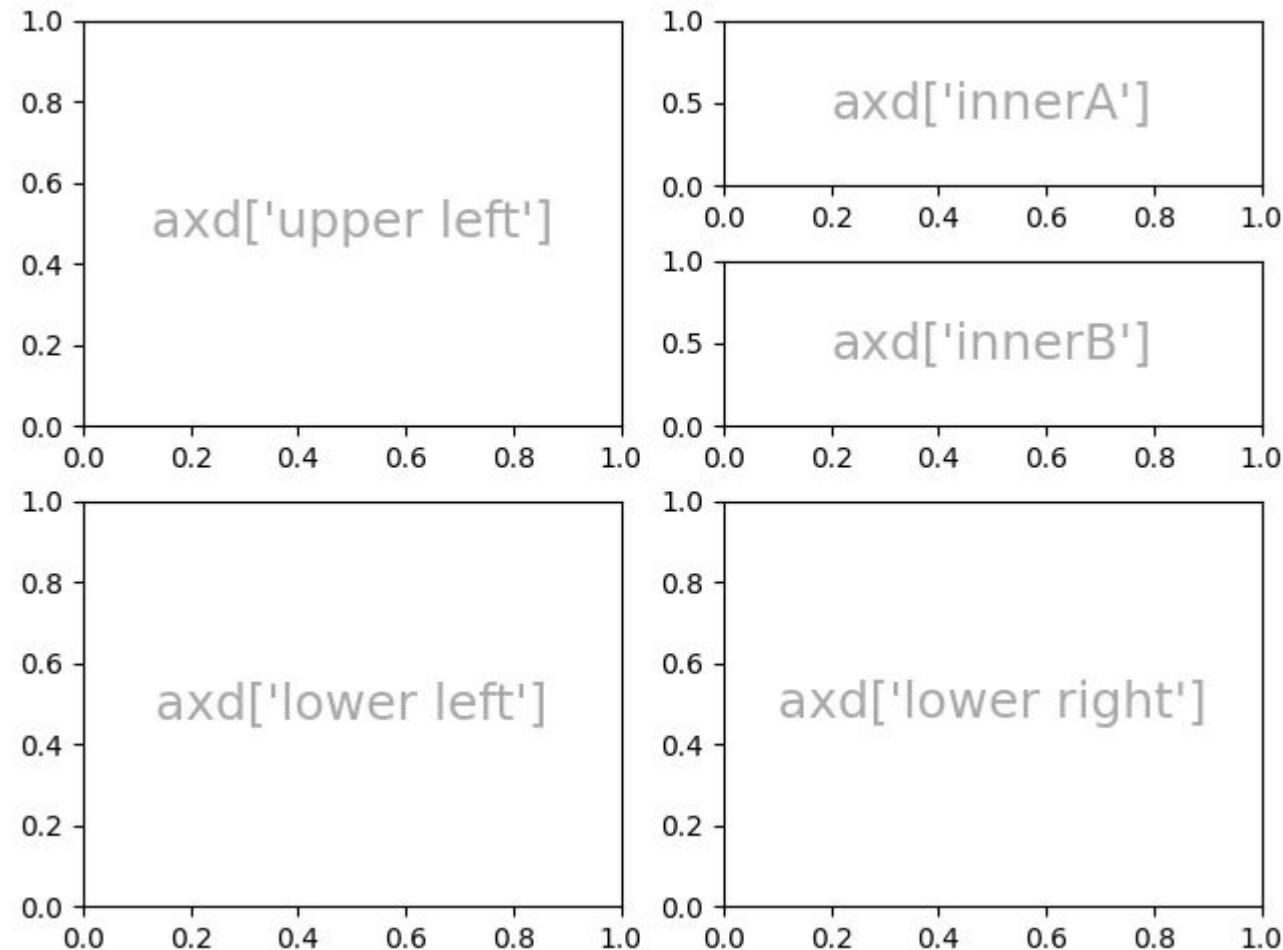
```
fig = plt.figure(layout="constrained")
subfigs = fig.subfigures(1, 2, wspace=0.07, width_ratios=[1.5, 1.])
axs0 = subfigs[0].subplots(2, 2)
subfigs[0].set_facecolor('lightblue')
subfigs[0].suptitle('subfigs[0]\nLeft side')
subfigs[0].supxlabel('xlabel for subfigs[0]')

axs1 = subfigs[1].subplots(3, 1)
subfigs[1].suptitle('subfigs[1]')
subfigs[1].supylabel('ylabel for subfigs[1]')
```



Matplotlib - Ejercicio

Crear la siguiente figura

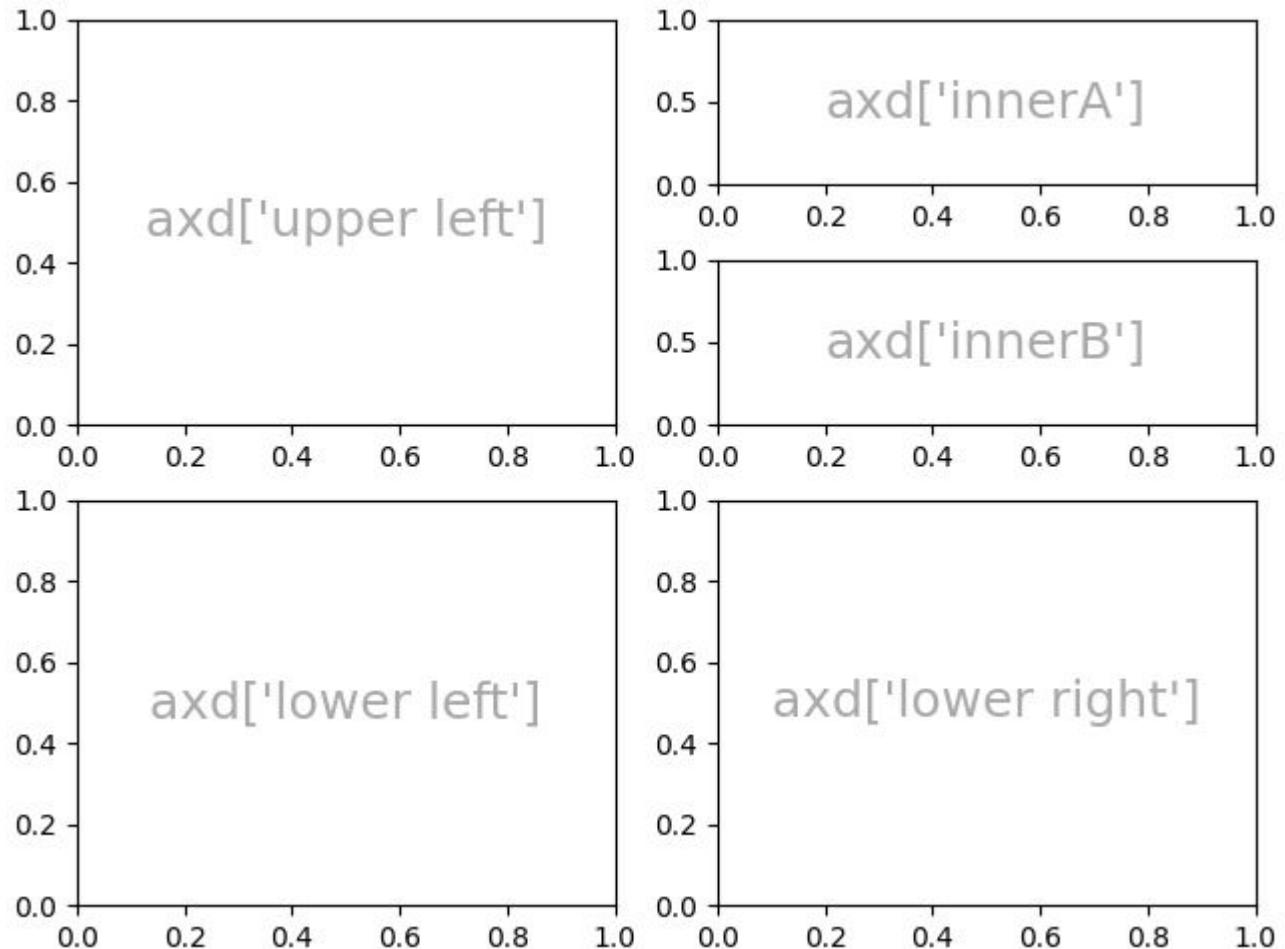


Matplotlib - Ejercicio

Crear la siguiente figura

```
inner = [['innerA'],
          ['innerB']]
outer = [['upper left', inner],
          ['lower left', 'lower right']]

fig, axd = plt.subplot_mosaic(outer, layout="constrained")
for k, ax in axd.items():
    annotate_axes(ax, f'axd[{k!r}]')
```



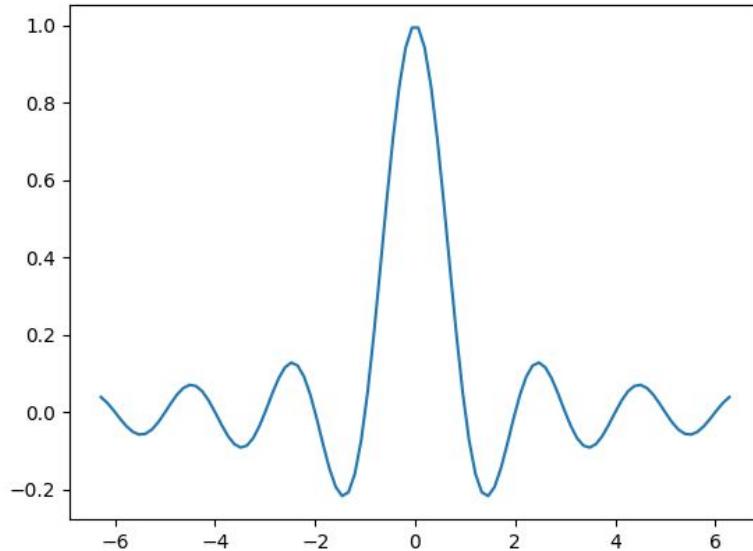
Matplotlib – Autoescalado de ejes

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl

x = np.linspace(-2 * np.pi, 2 * np.pi, 100)
y = np.sinc(x)

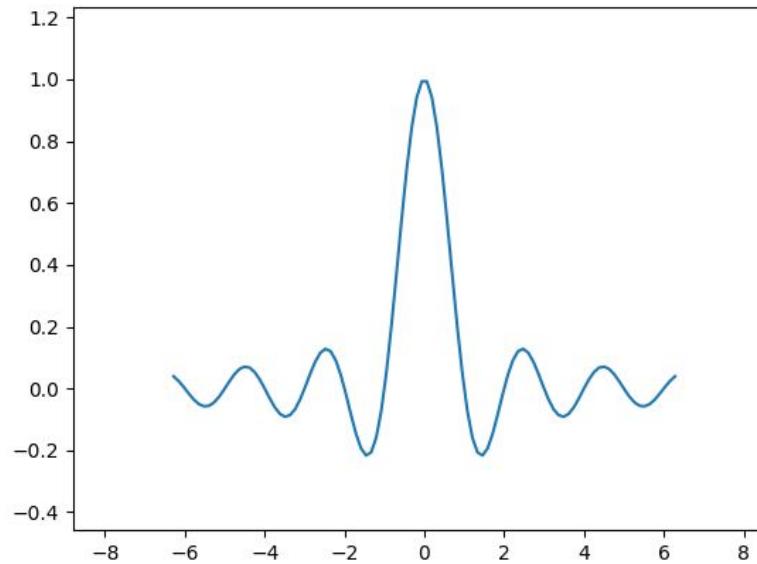
fig, ax = plt.subplots()
ax.plot(x, y)
```



Márgenes

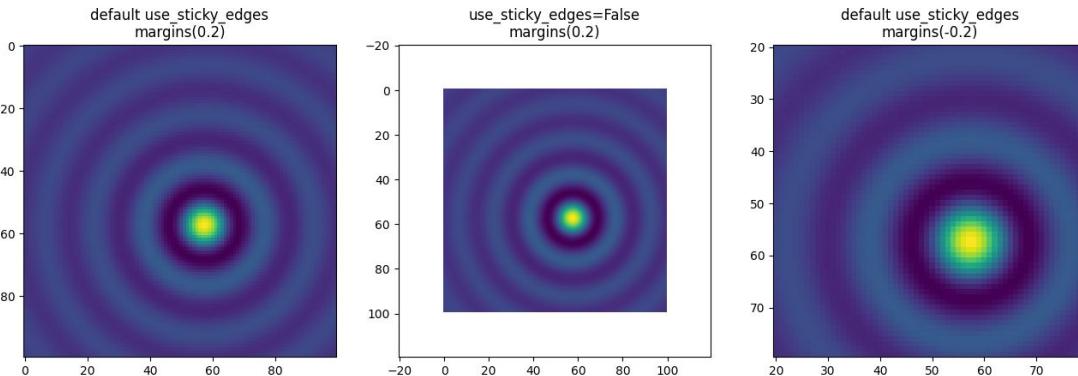
El margen por defecto alrededor de los límites de datos es del 5%

```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.margins(0.2, 0.2)
```

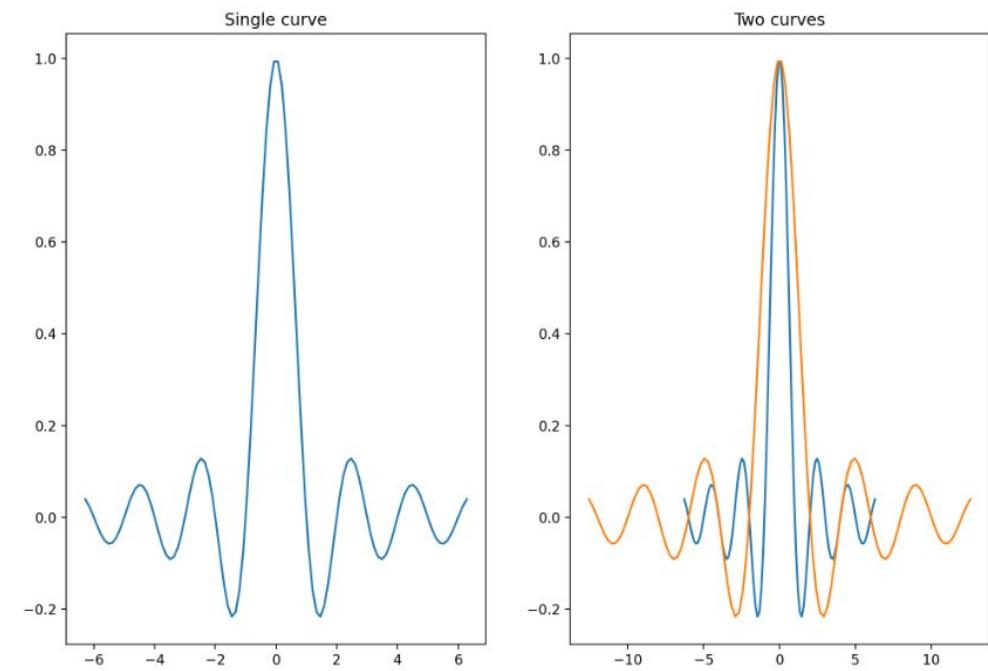


Matplotlib

```
fig, ax = plt.subplots(ncols=3, figsize=(16, 10))
ax[0].imshow(zz)
ax[0].margins(0.2)
ax[0].set_title("default use_sticky_edges\nmargins(0.2)")
ax[1].imshow(zz)
ax[1].margins(0.2)
ax[1].use_sticky_edges = False
ax[1].set_title("use_sticky_edges=False\nmargins(0.2)")
ax[2].imshow(zz)
ax[2].margins(-0.2)
ax[2].set_title("default use_sticky_edges\nmargins(-0.2)")
```



```
fig, ax = plt.subplots(ncols=2, figsize=(12, 8))
ax[0].plot(x, y)
ax[0].set_title("single curve")
ax[1].plot(x, y)
ax[1].plot(x * 2.0, y)
ax[1].set_title("Two curves")
```



Matplotlib

```
fig, axs = plt.subplot_mosaic([['linear', 'linear-log'],
                               ['log-linear', 'log-log']], layout='constrained')

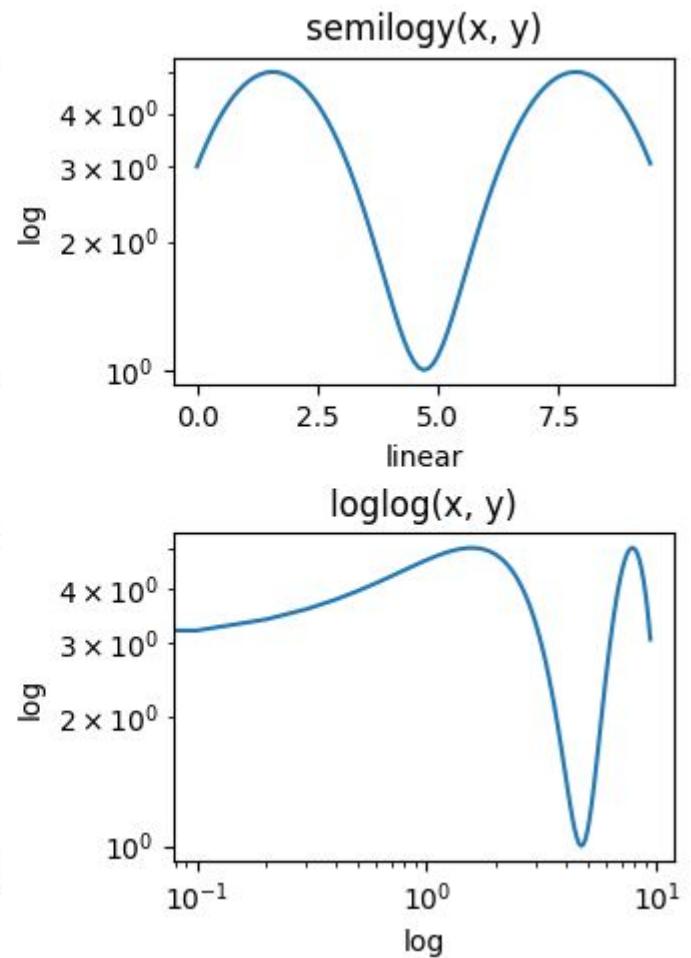
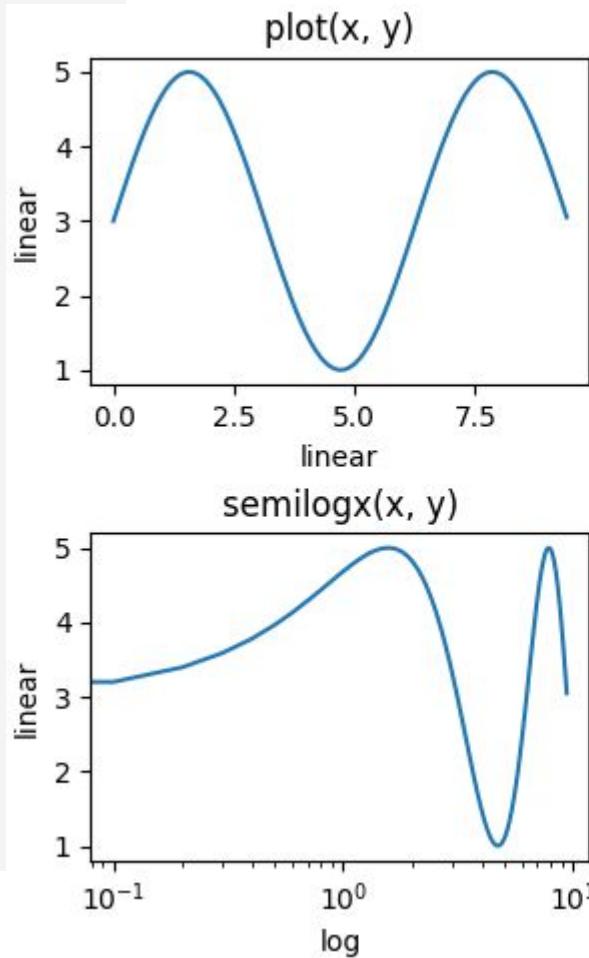
x = np.arange(0, 3*np.pi, 0.1)
y = 2 * np.sin(x) + 3

ax = axs['linear']
ax.plot(x, y)
ax.set_xlabel('linear')
ax.set_ylabel('linear')
ax.set_title('plot(x, y)')

ax = axs['linear-log']
ax.semilogy(x, y)
ax.set_xlabel('linear')
ax.set_ylabel('log')
ax.set_title('semilogy(x, y)')

ax = axs['log-linear']
ax.semilogx(x, y)
ax.set_xlabel('log')
ax.set_ylabel('linear')
ax.set_title('semilogx(x, y)')

ax = axs['log-log']
ax.loglog(x, y)
ax.set_xlabel('log')
ax.set_ylabel('log')
ax.set_title('loglog(x, y)')
```



Librería Seaborn

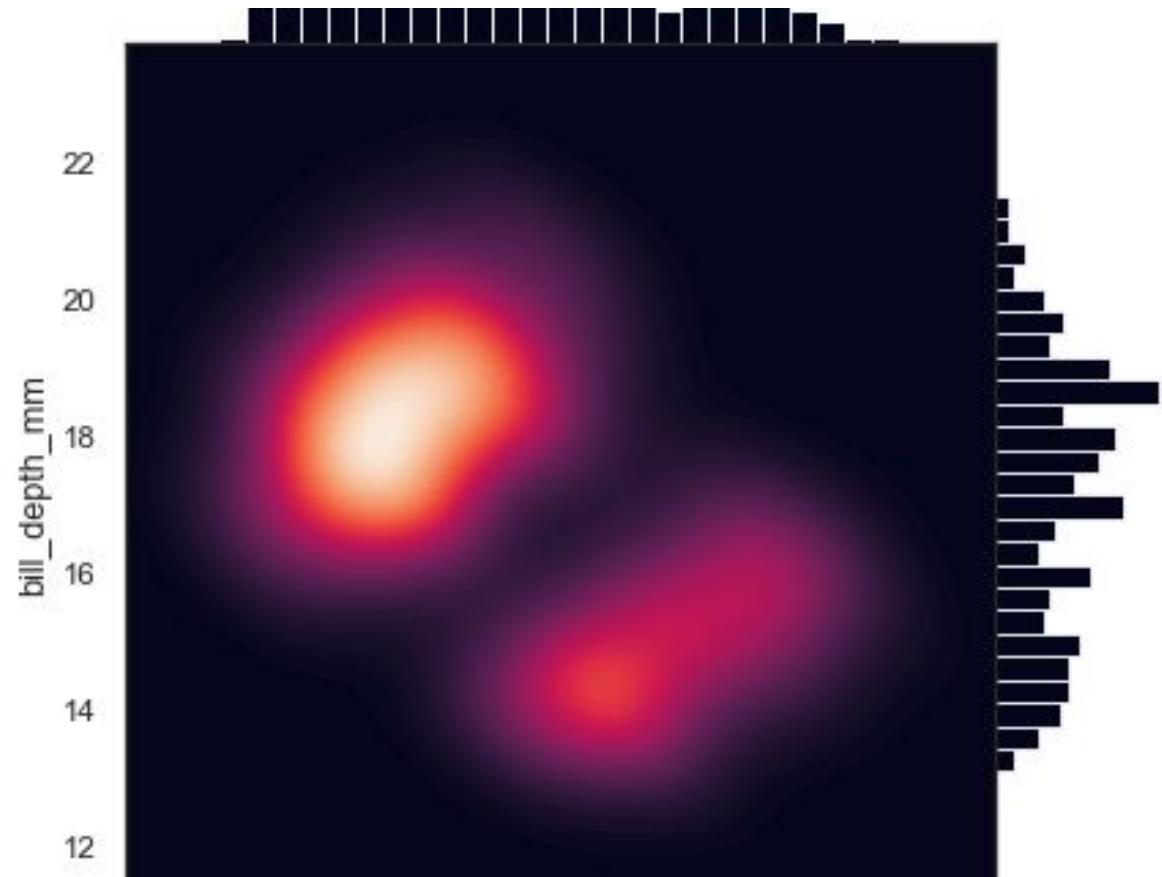


seaborn

Seaborn es una biblioteca para hacer gráficos estadísticos en Python. Se basa en matplotlib y se integra estrechamente con las estructuras de datos de pandas.

Seaborn le ayuda a explorar y comprender sus datos.

Sus funciones de trazado operan en marcos de datos y matrices que contienen conjuntos de datos completos y realizan internamente el mapeo semántico y la agregación estadística necesarios para producir gráficos informativos.



seaborn

```
pip install seaborn
```

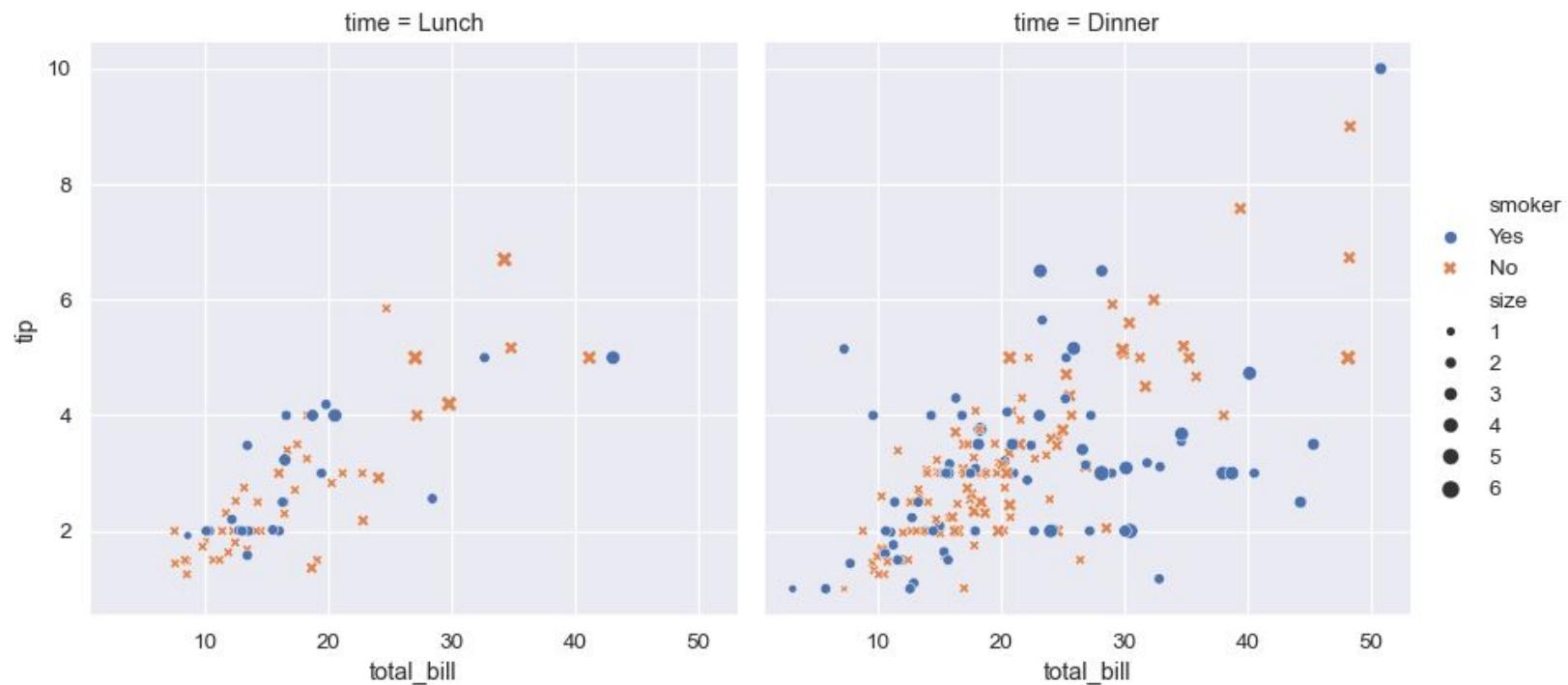
```
import seaborn as sns
```

```
# Import seaborn
import seaborn as sns

# Apply the default theme
sns.set_theme()

# Load an example dataset
tips = sns.load_dataset("tips")

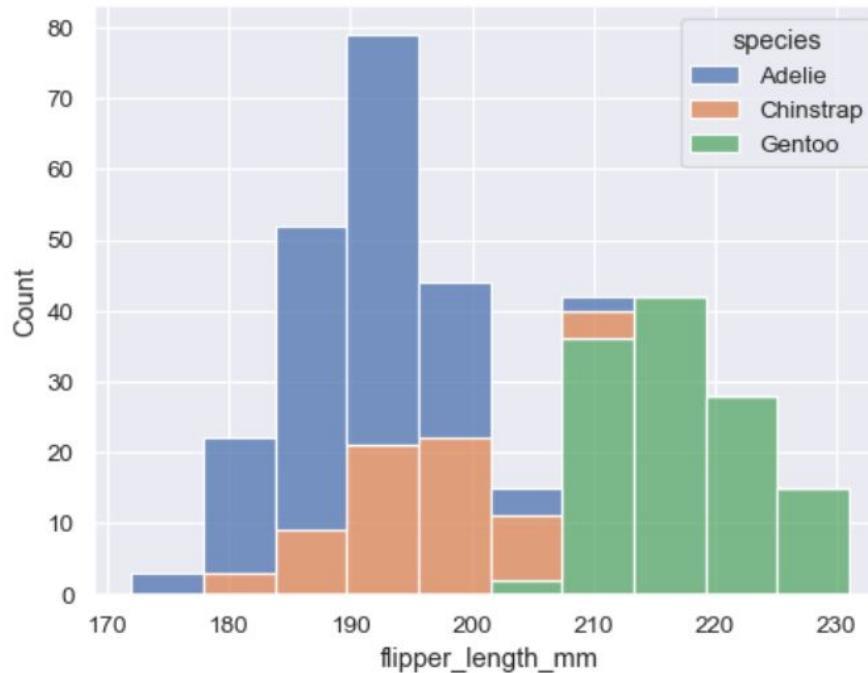
# Create a visualization
sns.relplot(
    data=tips,
    x="total_bill", y="tip", col="time",
    hue="smoker", style="smoker", size="size",
)
```



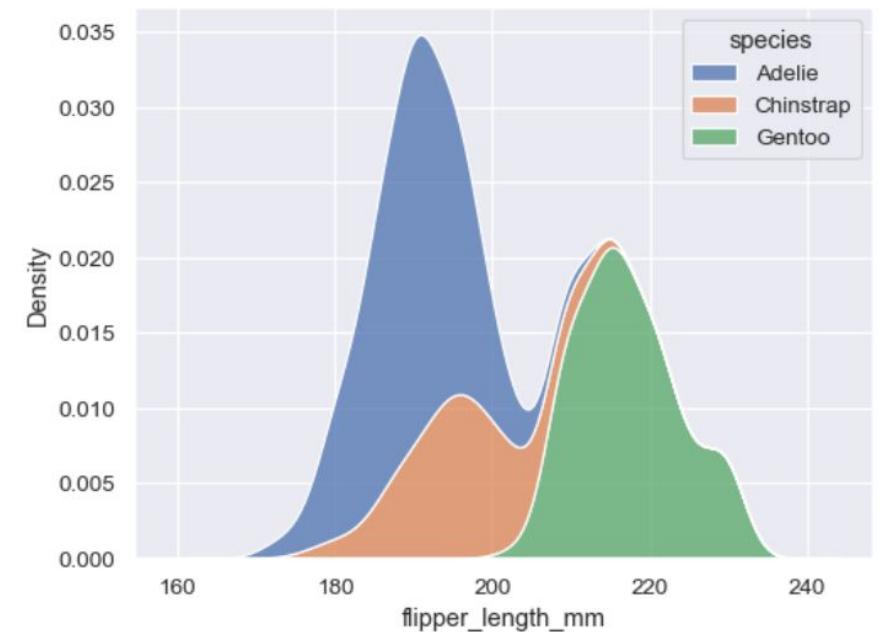
seaborn

El espacio de nombres seaborn es plano; toda la funcionalidad es accesible en el nivel superior. Pero el código en sí está estructurado jerárquicamente, con módulos de funciones que logran objetivos de visualización similares a través de diferentes medios. La mayor parte de la documentación está estructurada en torno a estos módulos: encontrará nombres como "relacional", "distribucional" y "categórico".

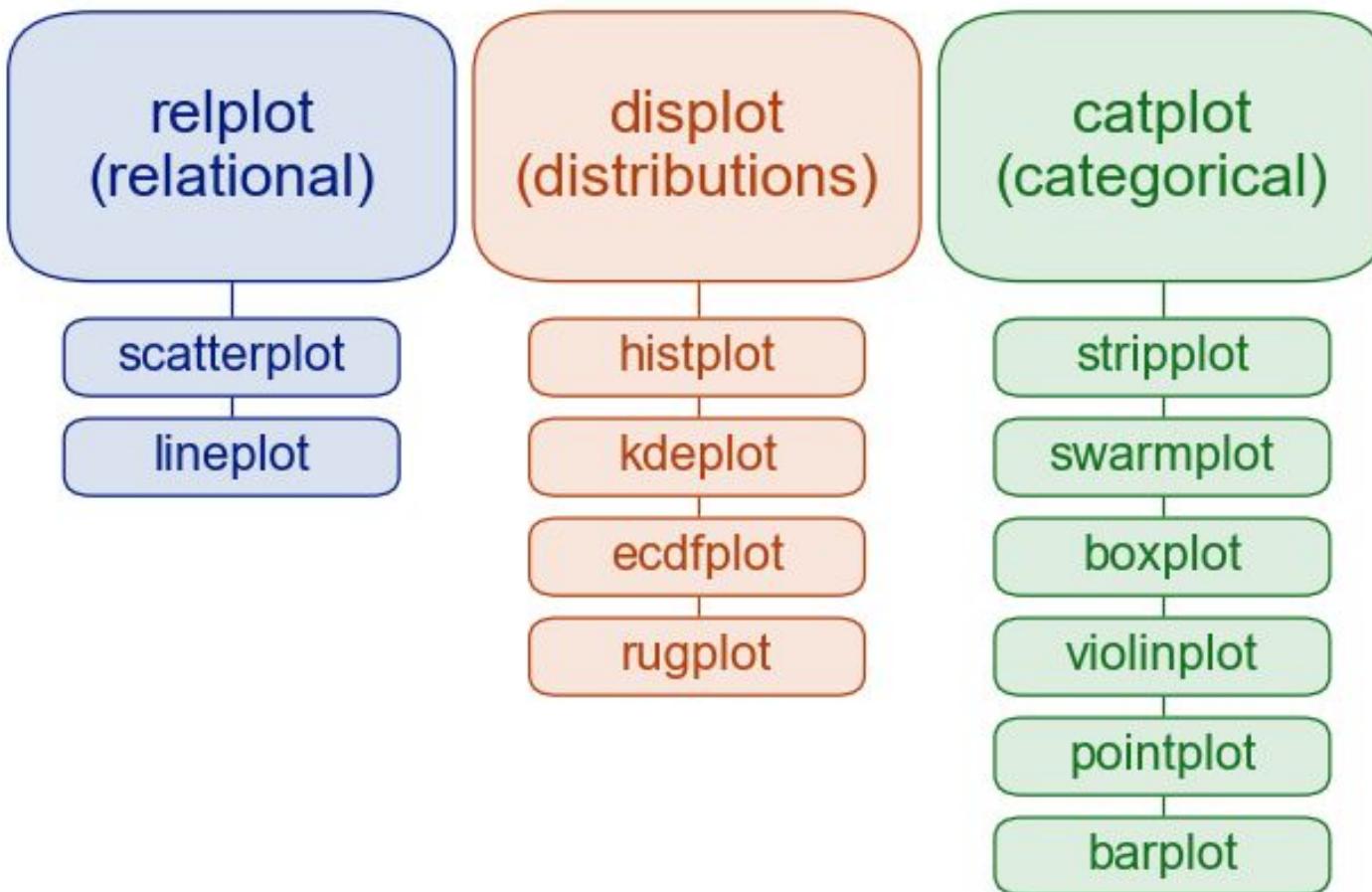
```
penguins = sns.load_dataset("penguins")
sns.histplot(data=penguins, x="flipper_length_mm", hue="species", multiple="stack")
```



```
sns.kdeplot(data=penguins, x="flipper_length_mm", hue="species", multiple="stack")
```



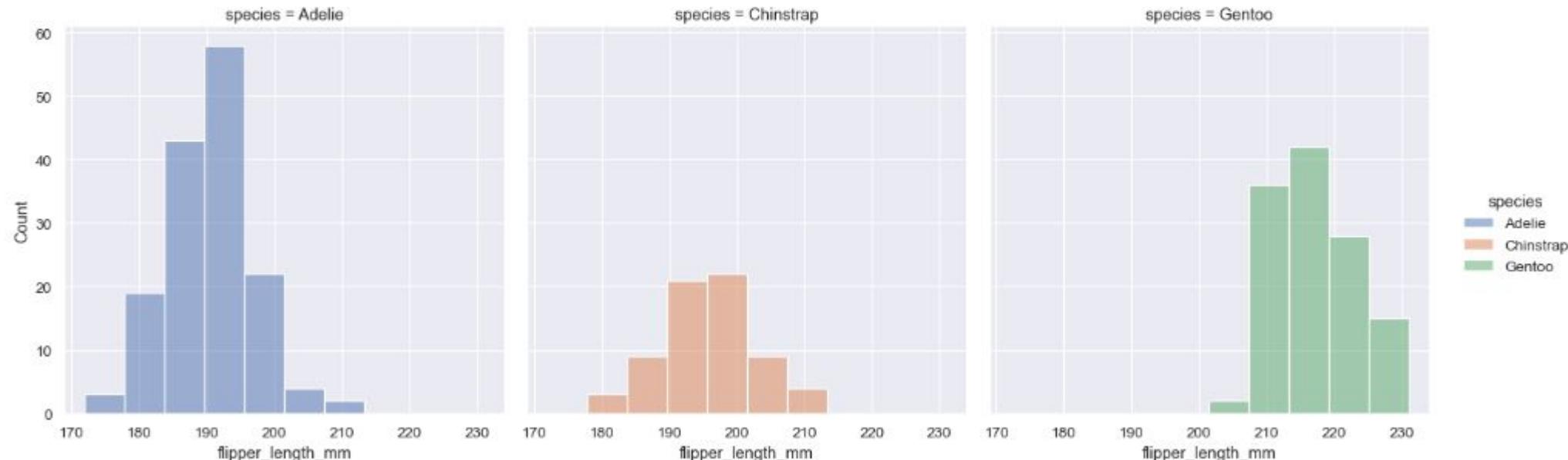
seaborn



seaborn

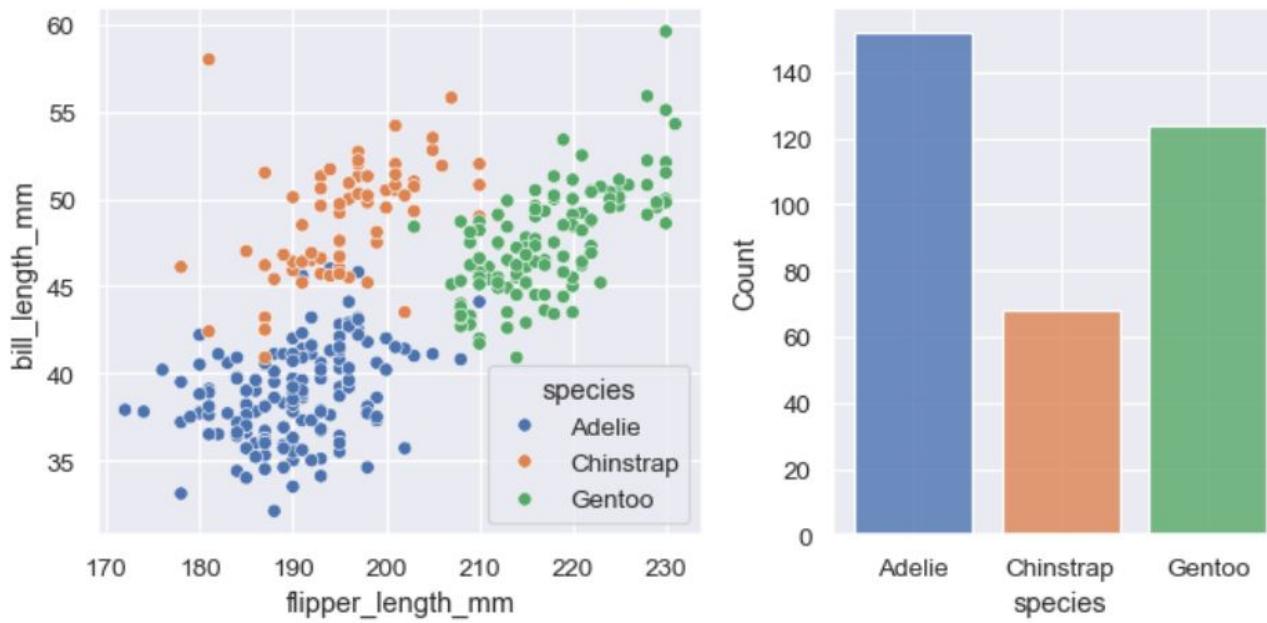
a característica más útil que ofrecen las funciones a nivel de figura es que permiten crear fácilmente figuras con múltiples subparcelas. Por ejemplo, en lugar de apilar las tres distribuciones de cada especie de pingüinos en los mismos ejes, podemos "facetarlas" trazando cada distribución a lo largo de las columnas de la figura:

```
sns.displot(data=penguins, x="flipper_length_mm", hue="species", col="species")
```



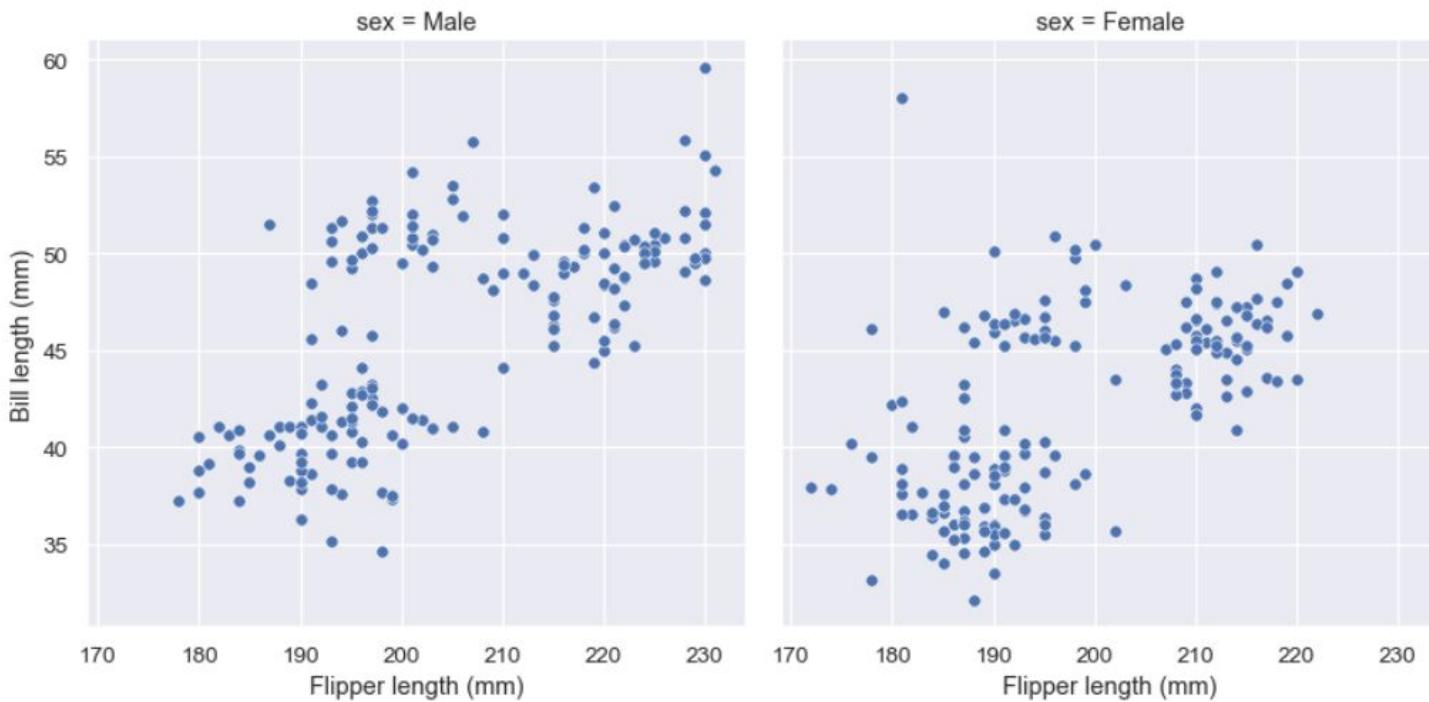
seaborn

```
f, axs = plt.subplots(1, 2, figsize=(8, 4), gridspec_kw=dict(width_ratios=[4, 3]))
sns.scatterplot(data=penguins, x="flipper_length_mm", y="bill_length_mm", hue="species", ax=axs[0])
sns.histplot(data=penguins, x="species", hue="species", shrink=.8, alpha=.8, legend=False, ax=axs[1])
f.tight_layout()
```



seaborn

```
g = sns.relplot(data=penguins, x="flipper_length_mm", y="bill_length_mm", col="sex")
g.set_axis_labels("Flipper length (mm)", "Bill length (mm)")
```

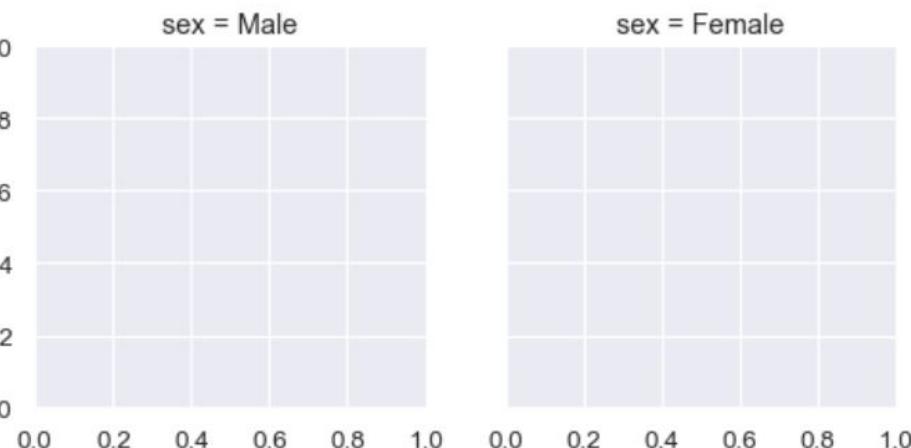
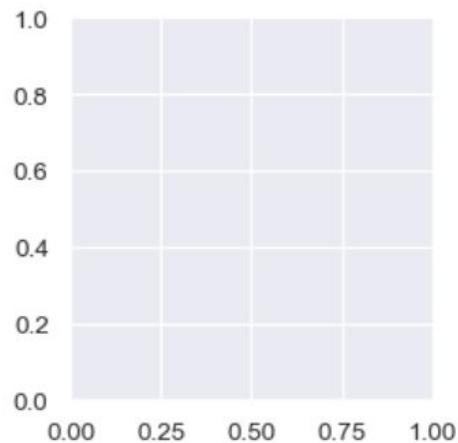


Las funciones a nivel de figura devuelven una instancia de FacetGrid, que tiene algunos métodos para personalizar los atributos del gráfico de forma "inteligente" respecto a la organización de la subparcela. Por ejemplo, puede cambiar las etiquetas de los ejes externos con una sola línea de código

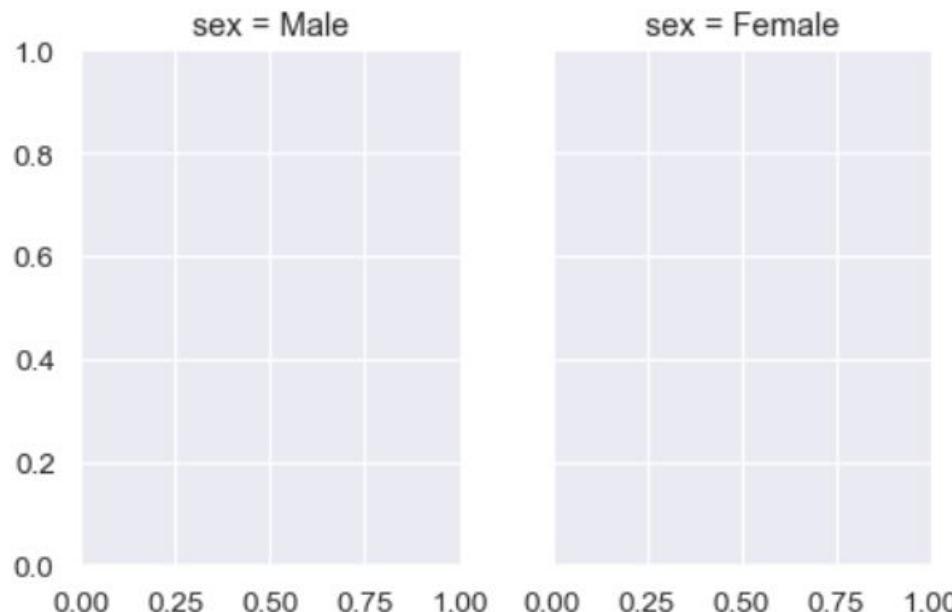
seaborn

```
g = sns.FacetGrid(penguins, col="sex")
```

```
g = sns.FacetGrid(penguins)
```

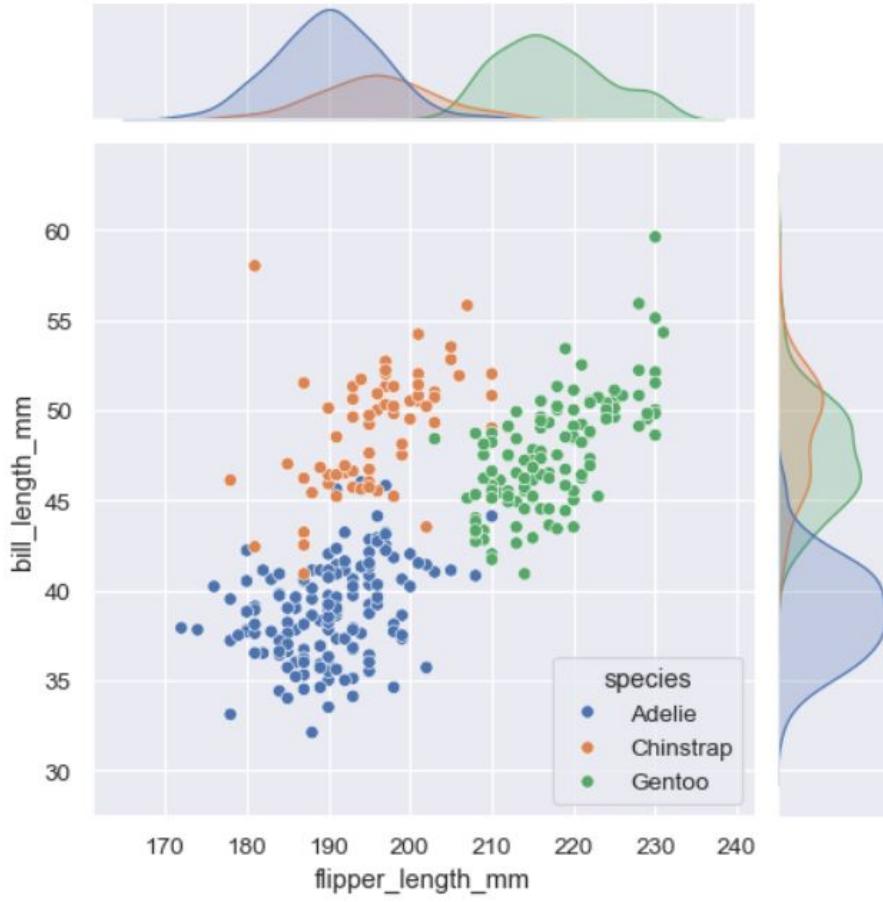


```
g = sns.FacetGrid(penguins, col="sex", height=3.5, aspect=.75)
```

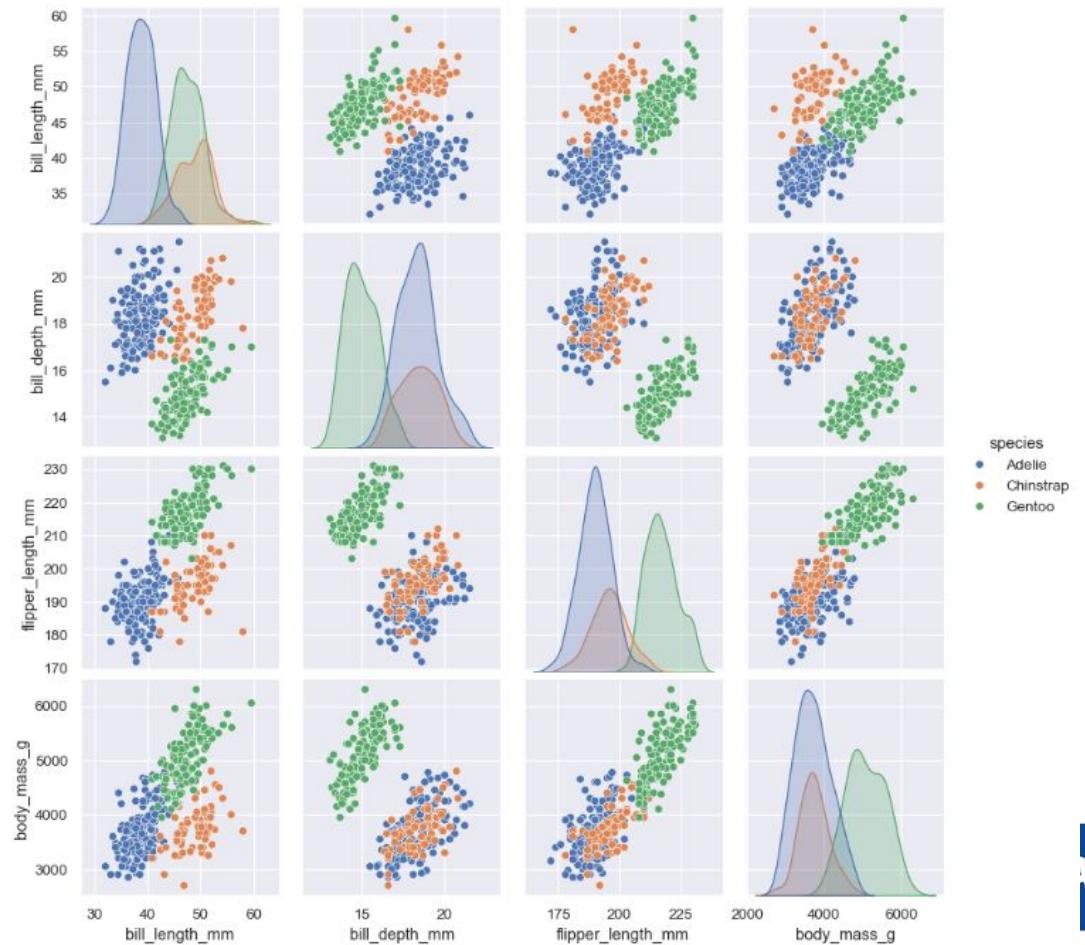


Seaborn - Combinar múltiples puntos de vista sobre los datos

```
sns.jointplot(data=penguins, x="flipper_length_mm", y="bill_length_mm", hue="species")
```

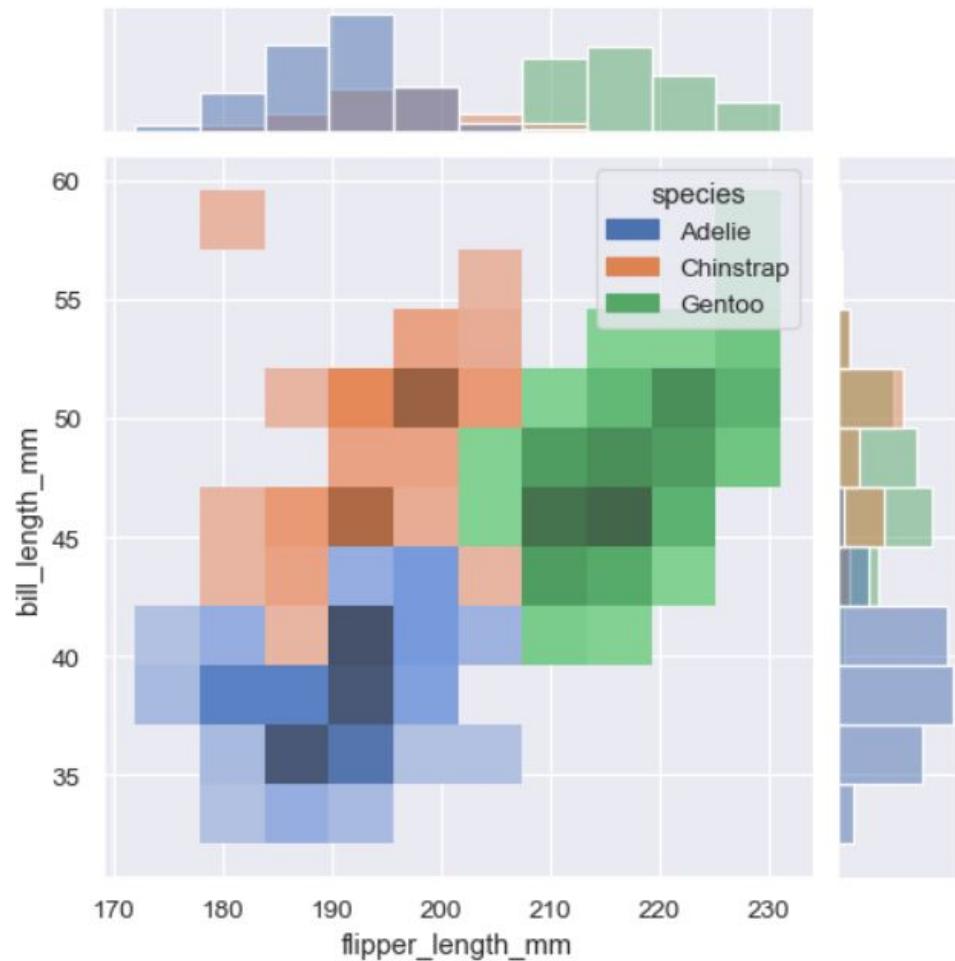


```
sns.pairplot(data=penguins, hue="species")
```



Seaborn - Combinar múltiples puntos de vista sobre los datos

```
sns.jointplot(data=penguins, x="flipper_length_mm", y="bill_length_mm", hue="species", kind="hist")
```



CAFAM