Astrometría 2020

Release 0.0.1

Aug 19, 2021

Contents

1 Teoría **3**
- 1.1 Conceptos teóricos . . . 3
- 1.2 Recursos . . . 3

2 Ayuda para python **5**
- 2.1 Empezando con python . . . 5
- 2.2 Interactuando con python . . . 8
- 2.3 El intérprete de python . . . 9
- 2.4 usando la ayuda . . . 9
- 2.5 Módulos . . . 10
- 2.6 Gráficos con python . . . 10

3 Herramientas **33**
- 3.1 GIT . . . 33

4 Ayuda para las guias **37**
- 4.1 Ayuda para la guia 1 . . . 37

5 Búsqueda **39**

Aquí encontrarás material para el cursado de la materia Astrometría 2021 de Famaf. Estos recursos corresponden al cursado virtual de la materia, de acuerdo a los lineamientos establecidos por la Res. Rectoral 387/2020.

CHAPTER 1

Teoría

1.1 Conceptos teóricos

1.1.1 Nociones de programación

1.1.2 Algoritmos

1.1.3 Métodos numéricos

1.2 Recursos

1.2.1 Nociones de programación

1.2.2 Algoritmos

1.2.3 Métodos numéricos

CHAPTER 2

Ayuda para python

2.1 Empezando con python

Para trabajar con python es recomendable usar entornos virtuales. Las instalaciones de Anaconda tienen también asociados entornos virtuales, aunque la sintaxis es algo diferente.

2.1.1 Instalación de python

Hay numerosos tutoriales que explican como instalar python.

Entre las opciones disponibles, las más populares son instalar python desde el repositorio oficial o instalar una distribución de Anaconda.

Python es un lenguaje de alto nivel y de propósito general.

Anaconda es una distribución de python y de R que incluye muchas herramientas para trabajar en ciencias de datos y en aprendizaje automático. Las distribuciones de Anaconda incluyen muchos paquetes que posiblemente no necesitamos.

Todas las herramientas de Anaconda se pueden instalar desde python.

2.1.2 Manejadores de paquetes

El manejador de paquetes de Anaconda se llama *conda*, mientras que el de python se llama *pip*.

Para instalar un paquete, por ejemplo numpy:

```
pip install numpy
```

Los paquetes de python forman parte de un repositorio de paquetes llamado **'pypi (Python Package Index)<https://pypi.org/>'_**. Allí se pueden encontrar cientos de miles de paquetes par diferentes propósitos.

2.1.3 Cómo preparar un entorno virtual

Los entornos virtuales permiten experimentar con paquetes de python sin riesgo de afectar al sistema en el que se está trabajando. También permite tener diferentes entornos para trabajar con diferentes versiones de python o de los paquetes.

Pare crear un entorno virtual, llamado "MyVE":

```
virtualenv MyVE
```

o bien, con las versiones más nuevas de python:

```
python -m venv MyVE
```

Una vez que está creado, se puede acceder al mismo ejecutando el script "activate", que está ubicado en la carpeta creada:

```
source MyVE/bin/activate
```

Muchos proyectos tienen asociado un archivo que indica los paquetes requeridos, usualmente se llama "requirements.txt":

```
pip install -r requirements.txt
```

2.1.4 Entornos virtuales con Anaconda

```
conda create --name MyVE
```

y para activarlo:

```
conda activate MyVE
```

Usando Anaconda se puede trabajar con una colección de entornos virtuales, sin tener que acceder a las carpetas creadas para cada entorno. Por ejemplo, si tenemos dos entornos virtuales, podemos pasar de uno a otro desde cualquier lugar:

```
conda activate MyVE_1
```

```
conda activate MyVE_2
```

Más información sobre los entornos virtuales de anaconda

Esto también es posible con las instalaciones de python en el sistema (sin anaconda) usando la herramienta virtualenvwrapper.

2.1.5 Virtualenvwrapper

virtualenvwrapper es un conjunto de extensiones de virtualenv que permiten administrar entornos virtuales y mejorar el flujo de trabajo.

Los pasos a seguir son:

1. Crear una carpeta (oculta) donde se almacenarán las configuraciones de los EV:

```
mkdir ~/.virtualenvs
```

2. Instalar los paquetes necesarios. Por ejemplo en sistemas basados en Debian:

```
apt install virtualenvwrapper
pip install virtualenvswrapper
```

si no está el pip, instalar: (virtualenv y python-setuptools):

```
sudo apt install python3-pip
```

3. Editar el archivo del shell bash:

```
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_VIRTUALENV=/usr/local/bin/virtualenv
export VIRTUALENVWRAPPER_VIRTUALENV_ARGS='--no-site-packages'
export VIRTUALENVWRAPPER_PYTHON=$(which python3)
```

Hay varias cosas que pueden fallar. Aquí una breve lista de resolución de problemas:

Que puede fallar:

1) virtualenvwrapper

   La ubicacion del script virtualenvwrapper.sh depende de la distribucion de linux:

   Mint: source /usr/local/bin/virtualenvwrapper.sh

   CBPP: source $HOME/.local/bin/virtualenvwrapper.sh

2) virtualenv

   en la variable VIRTUALENVWRAPPER_VIRTUALENV poner la ubicación de virtualenv:

   $ which virtualenv

3) python

   Si da este error:

   source $HOME/.local/bin/virtualenvwrapper.sh

   /usr/local/bin/python3: Error while finding module specification for 'virtualenvwrapper.hook_loader' (ModuleNotFoundError: No module named 'virtualenvwrapper') virtualenvwrapper.sh: There was a problem running the initialization hooks.

   If Python could not import the module virtualenvwrapper.hook_loader, check that virtualenvwrapper has been installed for VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3 and that PATH is set properly.

   probar hacer esto:

   reemplazar: export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3 por: export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python

Más info:

- Si da errores.
- Command reference for wirtualenvwrapper
- Virtualenvwrapper readthedocs (Documentación)

Cómo se usa:

- mkvirtualenv –python=$(which python3) MyVE
- lsvirtualenv
- workon MyVE
- rmvirtualenv MyVE

2.1.6 Repositorios y control de versión

Dos de las acciones más útiles a la hora de desarrollar códigos son las de compartir y llevar un registro de versiones.

Para ello existen los llamados repositorios, que además de brindar una ubicación en la nuba donde compartir códigos brindan distintas herramientas.

Algunos de los repositorios más usados en la actualidad son:

- GitHub
- GitLab
- Assembla
- Anaconda Cloud

Estos repositorios trabajan con sistemas de control de versión, por ejemplo:

- GIT
- SVN

En esta materia usaremos la combinación GitHub, que trabaja con GIT.

2.1.7 Usando git con GitHub

Un buen tutorial sobre GitHub.

En la version simple, cuando solo un usuario edita:

1. git clone (one time only)
2. git pull
3. edit files
4. add files to the version control stack
5. commit changes
6. push changes
7. go to 2.

Varios usuarios:

1. git clone (one time only)
2. git pull
3. edit files
4. add files to the version control stack

5. before commit, git pull and resolve conflicts (if any) 5. commit changes 6. push changes 7. go to 2.

2.2 Interactuando con python

2.2.1 Cómo correr un script de python

Hay muchas formas de correr los scripts de python. En lo que sigue vamos a usar el simbolo $ para el prompt de linux y el símbolo >>> para el prompt de python.

La primera forma es correr las sentencias de manera interactiva. Para ello, entrar a python y escribir:

Para practicar las distintas formas de correr esto desde un archivo, vamos a escribir un script muy simple, y guardarlo en un archivo que se llame simple.py:

```
# contenidos del script simple.py
x = 1
y = 2
print(x+y)
```

Para ejecutarlo se puede hacer desde una terminal:

```
python simple.py
```

Otras opciones son:

```
# python3
exec(open('simple.py').read())

# python2
execfile('simple.py')

# ipython3
load simple.py
run simple.py
```

2.3 El intérprete de python

El comando de linux "ipython" abre una línea de comandos interactiva de python. Ofrece algunas funcionalidades adicionales al intérprete de python, por ejemplo:

- colorear la sintaxis
- recorrer fácilmente comandos anteriores
- facilidades para edición
- ayuda

entre otras.

Por ejemplo, los siguientes comandos funcionan en ipython pero no en el intérprete de python:

```
?
?object
object?
*pattern*?
%shell like --syntax
!ls
```

2.4 usando la ayuda

Hay dos formas de usar la ayuda:

```
a = 1.
help(a)
```

o bien, usando ipython,

```
a = 1.
a?
```

Las dos formas a veces devuelven lo mismo, pero a veces son diferentes.

Probemos por ejemplo crear un objeto llamado "a", de type _int_ (entero), y preguntarle a python sobre lo que podemos hacer con él:

La ayuda dice muchas cosas y puede ser difícil de entender, pero siempre es útil y con la experiencia de uso del lenguaje va adquiriendo más relevancia.

2.5 Módulos

Los módulos permiten organizar la forma de escribir código, contribuyendo a:

Simplicidad: Los modulos resuelven problemas simples y cortos, y se pueden usar luego en proyectos más complejos.

Mantenibilidad: Permiten limitar cada módulo a un tipo de problemas.

Reusabilidad: Un módulo se puede usar en muchos proyectos.

Contexto: Permite evitar conflictos con otras partes del programa por los nombres de las variables.

Para cargar los objetos de un módulo se pueden usar varias estrategias. Por ejemplo, para cargar el módulo pyplot, se puede hacer:

```
from matplotlib import pyplot as plt
import matplotlib.pyplot
```

```
import math
mypi = math.pi

from math import pi
mypi = pi

from math import *
mypi = pi

import math as m
mypi = m.pi
```

2.6 Gráficos con python

El lenguaje python permite realizar una amplia variedad de visualizaciones de datos. La misma se realiza mediante la ayuda de módulos (como math o numpy) que están destinados a brindar herramientas de visualización.

En la lista que sigue se presentan algunos de los módulos más usados. El objetivo no es aprenderlos, sino saber que existen varias opciones y disminuir un poco la confusión que se puede producir al principio al buscar ayuda y bibliografía:

- Matplotlib
- Seaborn
- Plotly

- Bokeh
- Altair
- Pygal
- Pandas
- Plotnine

La elección de una de estas herramientas para hacer un gráfico depende del contexto, para qué se quiere hacer el gráfico, dónde se va a mostrar y a partir de qué datos. Así, por ejemplo, Plotly, Bokeh y Altair devuelven gráficos en HTML para ser mostrados en páginas web, Pygal genera gráficos vectoriales y Pandas grafica datos guardados en cierto tipo especial de estructura.

En este tutorial (y en la materia) usaremos solamente Matplotlib.

2.6.1 El módulo Matplotlib

Matplotlib es un módulo de python que ofrece una librería para graficar.

En general viene instalado con la distribución de python Anaconda, y si no se puede instalar con el comando `pip`:

```
pip install -U matplotlib
```

Si eso no funciona, consultar los detalles de la documentación.

Una vez instalado, se puede acceder al módulo desde un entorno de python (es decir, luego de "entrar" a python) con el comando import:

```
import matplotlib
```

aunque no es muy usual utilizarlo así.

2.6.2 Interface pyplot

El módulo Matplotlib incluye una interface denominada `pyplot` que tiene todas las herramientas para hacer gráficos sencillos. Es posible que resulte un poco confuso la utilización de términos como módulo, librería o interface, pero no es necesario tener un conocimiento acabado de estos conceptos para producir gráficos, ni está en el alcance de esta materia. En este breve tutorial aprenderemos lo básico para realizar gráficos aprendiendo a partir de ejemplos.

Para cargar esta interface, simplemente usamos el comando import:

```
import matplotlib.pyplot
```

Hay otras formas de hacer lo mismo, que se pueden encontrar en los numerosos tutoriales que hay disponibles en internet. Algunas de estas formas son:

```
from matplotlib import pyplot        #  (alternativa 1)
from matplotlib import pyplot as plt #  (alternativa 2)
import matplotlib.pyplot as plt      #  (alternativa 3)...
```

Esas tres líneas son equivalentes (sólo hay que usar una). Notar que se introdujo el alias `plt`, que es una costumbre muy arraigada en la comunidad de ciencias de datos. Se puede reemplazar `plt` por cualquier otra cosa. En lo que sigue usaremos el alias `plt` como es usual.

2.6.3 Estilos de uso para pyplot

Antes de empezar a hacer gráficos, conviene aclarar que hay dos formas de usar `Pyplot`. Puede ser confuso leer la documentación disponible si no se tiene en cuenta esto, ya que las mismas cosas sencillas se encuentran realizadas de diferentes formas. Esta sección está destinada a evitar esas confusiones al mostrar las dos formas de trabajo, que son:

- estilo matlab
- estilo con orientación a objetos

Para el práctico se puede usar cualquiera de las dos.

Pyplot al estilo Matlab

Esta forma de usar Pyplot se llama "sintaxis imperativa", y fue diseñada para parecerse a Matlab, que es otro lenguaje pensado para trabajar con matrices que permite también hacer gráficos.

El gráfico de la función seno en Matlab se puede hacer así:

```
x = linspace(0,2*pi,100);
y = sin(x);
plot(x,y)
xlabel('x')
ylabel('sin(x)')
title('Grafico de la funcion seno')
```

Ahora hacemos el mismo gráfico desde python:

```
1 import numpy as np
2 from matplotlib import pyplot as plt
3 x = np.linspace(0, 2*np.pi, 100)
4 y = np.sin(x)
5 plt.plot(x, y)
6 plt.xlabel('x')
7 plt.ylabel('sin(x)')
8 plt.title('Grafico de la funcion seno')
9 plt.show()
```

En la línea 6 estamos creando un gráfico a partir de los arrays x e y, y a partir de allí todo lo que hacemos con `plt` se aplica a ese gráfico.

Al usar el método `pylab` (ver más adelante), es posible modificar los atributos de los gráficos de manera interactiva e ir visualizando los cambios. En ese caso no se usa `plt.` sino que se escribe directamente la función, de manera similar a Matlab:

```
# luego de entrar al interprete usando ipython --pylab:

import numpy as np
from matplotlib import pyplot as plt
x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)
plot(x, y)
xlabel('x')
ylabel('sin(x)')
title('Grafico de la funcion seno')
```

alternativamente,

```
# luego de entrar al interprete usando ipython --pylab:

import numpy as np
from pylab import *

x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)
plot(x, y)
xlabel('x')
ylabel('sin(x)')
title('Grafico de la funcion seno')
show()
```

notar que en este último ejemplo importamos todas las funciones del módulo `pylab`. Aquí el gráfico no será interactivo.

Pyplot al estilo Orientación a Objetos

La orientación a objetos es un paradigma de programación (es decir, una forma de programar justificada teóricamente) que permite estructurar el código utilizando objetos que tienen propiedades o comportamientos. Por ejemplo, un objeto de tipo "animal" puede moverse de cierta forma, como caminar o nadar (comportamiento o método) o tener cierta cantidad de patas (propiedad). Los comportamientos se implementan mediante funciones y se llaman "métodos".

Para ilustrar de modo genérico y sin formalidad cómo funciona esto, pensemos en definir un objeto de tipo animal que tiene la propiedad de moverse:

```
oveja = animal()
movimiento = oveja.movimiento()
```

Un programa puede tener varios objetos de tipo "animal" y no hace falta programar cada uno, sino que basta con decir que "es un animal" y fácilmente adquiere la propiedad de "número de patas" o el comportamiento de "forma de moverse".

Para hacer gráficos usando este concepto, trabajamos con dos objetos:

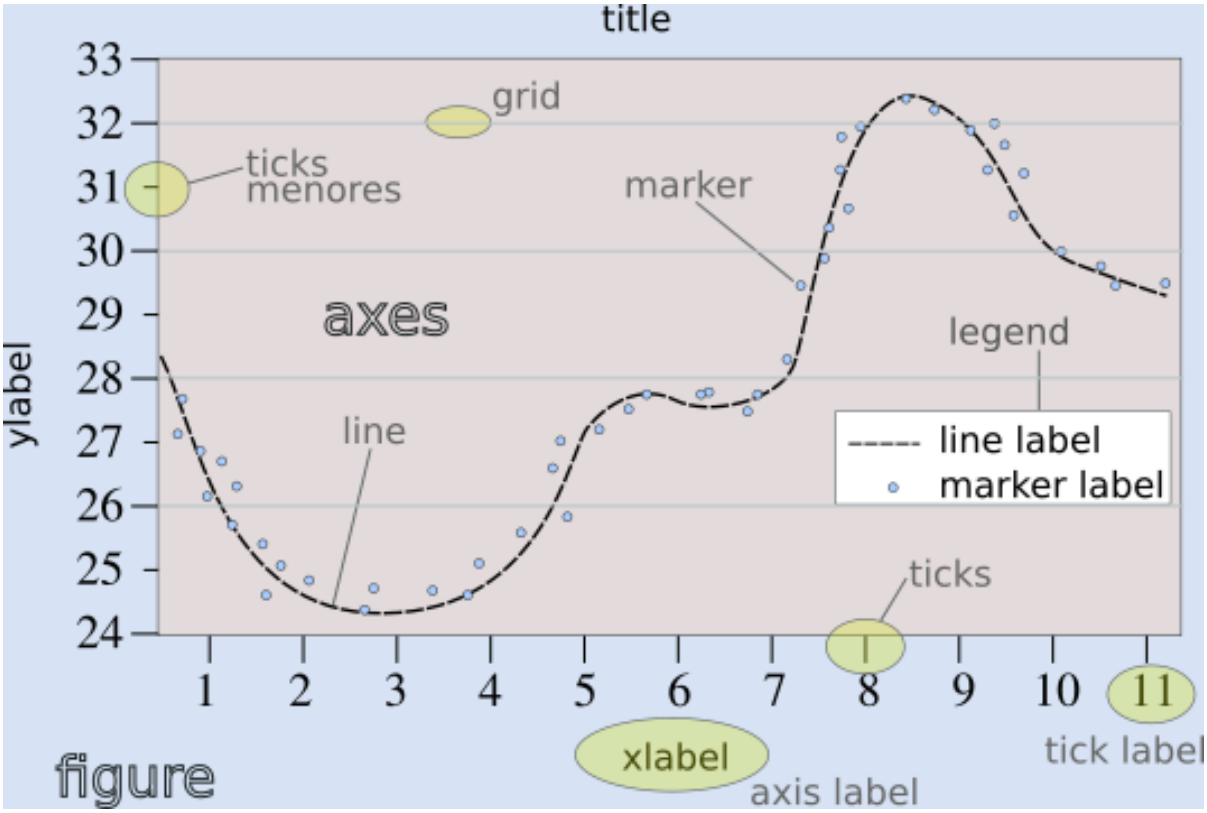
1. el objeto `figure`, que es la figura y puede contener varios gráficos (o `axes`)
2. el objeto `axes`, que es la región que contiene un gráfico individual. No es lo mismo que los ejes (x/y axis).

Así, por ejemplo, siguiendo la idea del ejemplo anterior, podremos hacer cosas como esta:

```
# plt puede crear una figura
figura = plt.figure()

# el objeto figura puede crear un area de trazado
ejes = figura.add_subplot()

# los ejes pueden adoptar nombres
ejes.set_xlabel('eje X')
ejes.set_ylabel('eje Y')
# o pueden pasarse a escala logaritmica
ejes.set_xscale('log')
```

Allí por ejemplo el método del objeto `figura` que crea los ejes se llama `add_subplot` y el método del objeto `ejes` que le permite da un nombre al eje X se llama `set_xlabel`. Es costumbre en la comunidad de python llamarle `fig` a una figura y `ax` (o `axes`) al area de trazado.

En la siguiente figura se muestran estos dos elementos, además de otros que usaremos para personalizar el aspecto visual del gráfico. Figure se refiere a toda la figura, y axes a la parte interior del sistema de ejes.



Es posible encontrar más detalles en esta otra versión.

Para generar una gráfico usando objetos, hay que crear un objeto de tipo `figure`, y luego un objeto de tipo `axes`, que es donde se realizará el gráfico.

```
1  import numpy as np
2  from matplotlib import pyplot as plt
3  x = np.linspace(0, 2*np.pi, 100)
4  y = np.sin(x)
5
6  fig = plt.figure()
7  fig.clf()
8  ax = fig.add_subplot(1,1,1)
9  ax.clear()
10 ax.plot(x, y)
11 ax.set_xlabel('x')
12 ax.set_ylabel('sin(x)')
13 ax.set_title('Grafico de la funcion seno')
14 fig.show()
```

aquí la función `figure` de pyplot crea una nueva figura, que está almacenada en el objeto `fig`. Este objeto, que es de tipo figura, puede hacer ciertas cosas, por ejemplo limpiar (`.clear()`) o mostrar (`.show()`) la figura. Otra cosa que se puede hacer es crear un objeto de tipo `axes`, lo cual se hace en la línea 9 con la función `add_subplot`.

Notar que add_subplot tiene 3 argumentos, para saber qué son podemos acceder a la ayuda en la documentación, por ejemplo desde el intérprete de `ipython`, haciendo:

```
from matplotlib import pyplot as plt
fig = plt.figure()
fig.add_subplot?
```

Hay otras formas de usar los objetos `figure` y `axes`, por ejemplo usando la función `subplots` de Pyplot, que devuelve tanto la figura como los gráficos (`axes`) que contiene:

```
import numpy as np
from matplotlib import pyplot as plt
x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)

fig, ax = plt.subplots()

ax.plot(x, y)
ax.set_xlabel('x')
ax.set_ylabel('sin(x)')
ax.set_title('Grafico de la funcion seno')
fig.show()
```

Si quisiéramos hacer una figura con más de un gráfico, se usan los parámetros de `add_subplot` o de `subplots` (de nuevo, ver la ayuda). Por ejemplo, para hacer los gráficos de las funciones seno y coseno, uno al lado del otro:

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

x = np.linspace(0, 2*np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

fig, ax = plt.subplots(1, 2)

ax[0].plot(x, y1)
ax[0].set_xlabel('x')
ax[0].set_ylabel('sin(x)')
ax[0].set_title('Grafico de la funcion seno')

ax[1].plot(x, y2)
ax[1].set_xlabel('x')
ax[1].set_ylabel('cos(x)')
ax[1].set_title('Grafico de la funcion coseno')

fig.show()
```

Notar que `subplots` devuelve un objeto axes que es una lista, donde cada elemento es un gráfico, es decir, `ax[0]` es el gráfico de la izquierda y `ax[1]` es el gráfico de la derecha. Al graficar, hay que decir en cuál de esos dos gráficos estamos trabajando.

Para hacer los dos gráficos, pero uno arriba del otro, sólo hay que cambiar los parámetros de `plt.subplots` (queda como ejercicio).

Otra forma de trabajar que se puede encontrar en los recursos destinados a este tema y que también es orientada a objetos, consiste en guardar el gráfico como un objeto. Por ejemplo, al graficar una línea guardamos el objeto de tipo "línea" (`matplotlib.lines.Line2D`). Luego a este objeto lo podemos modificar usando las funciones "set", por ejemplo:

```
import matplotlib.pyplot as plt
f = plt.figure()
ax = f.add_subplot()
l,  = ax.plot([1,2,3],[5,3,5])
l.set_color('tomato')
l.set_linestyle('--')
l.set_linewidth(3)
l.set_marker('o')
l.set_markeredgecolor('o')
l.set_markeredgewidth('o')
l.set_markerfacecolor('o')
plt.show()
```

Obteniendo el gráfico

Dependiendo de la forma de trabajar, necesitaremos hacer distintas cosas para obtener o visualizar el gráfico.

Se pueden mencionar las siguientes alternativas para trabajar en un entorno de python y visualizar el resultado de un gráfico con matplotlib:

1. Visualización en pantalla

   Para visualizar un gráfico en pantalla hay que pedirlo explícitamente con el método `show` de `pyplot`.

   ```
   plt.show()
   ```

2. Gráficos interactivos

   Se puede interactuar con un gráfico entrando al intérprete de ipython con la opción –pylab:

   ```
   $ ipython --pylab
   ```

   Notar que aquí el símbolo "$" corresponde al prompt del sistema.

3. Utilizando Notebooks

   Los notebooks son herramientas interactivas que corren en un navegador y que permiten combinar elementos de varios tipos, tales como gráficos, markdown, código y latex.

   Para ver los gráficos, en una celda del notebook hay que escribir el comando:

   ```
   %matplotlib inline
   ```

   El programa para trabajar con notebooks más usado en Jupyter.

4. Salida a un archivo

   Hay que guardar el gráfico en un archivo, con el método `savefig` de una figura.

   ```
   fig.savefig('MiFigura.png')
   ```

   Más detalles se pueden encontrar por ejemplo aquí

   Los formatos más usados son PNG, PDF y SVG. Para más información se puede consultar la documentación de savefig.

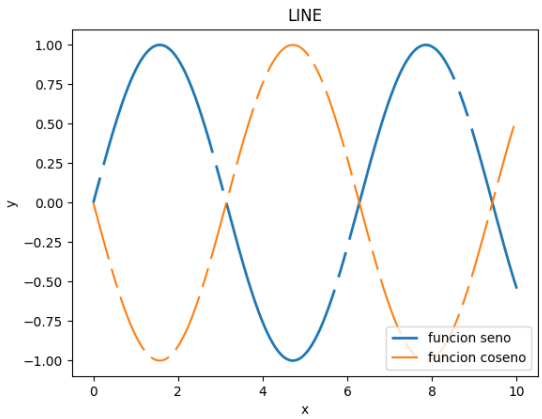
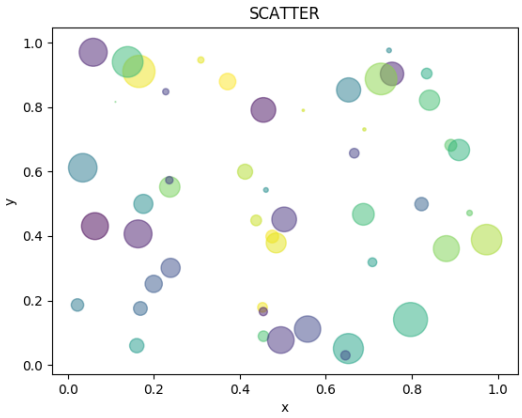
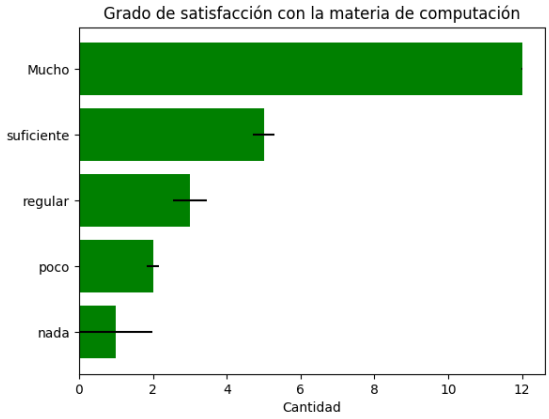
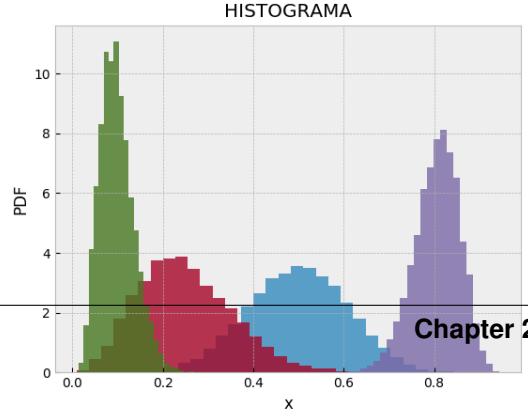
5. Entornos de desarrollo integrado

   Existen muchos programas que permiten desarrollar códigos y graficar en el mismo entorno. Algunos de los que se pueden mencionar son:

- Spyder
- PyCharm
- VSC

2.6.4 Gráficos simples

Ahora veremos ejemplos simples de cómo hacer gráficos en python usando matplotlib. Existen varios tipos de gráficos que se pueden hacer, los más simples son:

Tipo de gráfico	ejemplo
lineas	
puntos (scatter)	
barras	
histograma	

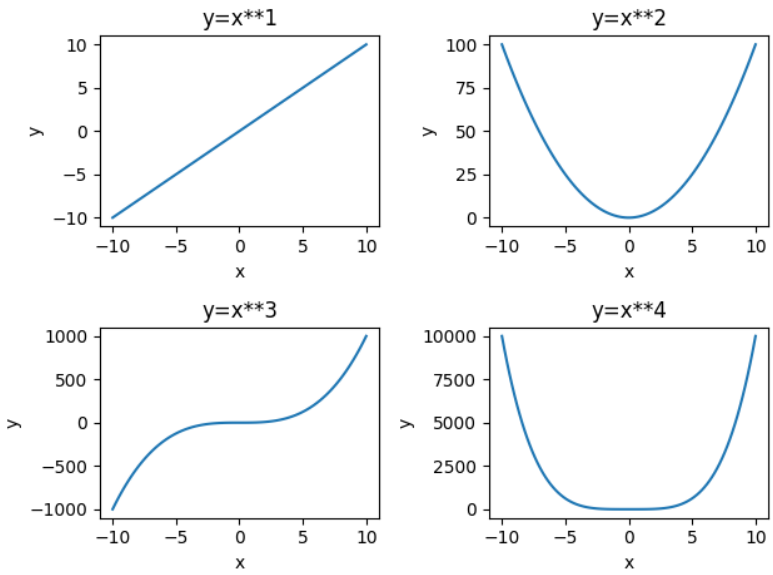
Hay muchos otros, pero entendiendo estos pocos se puede fácilmente incursionar en otros tipos de gráfico usando la documentación.

Varios gráficos en la misma figura

Para hacer varios gráficos en la misma figura se puede usar, como vimos, las funciones `subplots` o `add_subplot`.

```
from matplotlib import pyplot as plt
x = np.linspace(-10, 10, 100)
y1 = x
y2 = x**2
y3 = x**3
y4 = x**4

fig = plt.figure()
fig.clf()
ax = fig.subplots(2,2)

ax[0,0].plot(x, y1)
ax[0,0].set_xlabel('x')
ax[0,0].set_ylabel('y')
ax[0,0].set_title('y=x**1')

ax[0,1].plot(x, y2)
ax[0,1].set_xlabel('x')
ax[0,1].set_ylabel('y')
ax[0,1].set_title('y=x**2')

ax[1,0].plot(x, y3)
ax[1,0].set_xlabel('x')
ax[1,0].set_ylabel('y')
ax[1,0].set_title('y=x**3')

ax[1,1].plot(x, y4)
ax[1,1].set_xlabel('x')
ax[1,1].set_ylabel('y')
ax[1,1].set_title('y=x**4')

fig.tight_layout()
fig.show()
```

Que da algo así:



También se puede usar la función `add_axes`, que hace lo mismo pero tiene una sintaxis un poco diferente, ya que permite elegir explícitamente el tamaño y la ubicación de los gráficos.

Los argumentos de add_axes son las coordenadas de la esquina inferior izquierda del gráfico, y los tamaños de los ejes en el gráfico.

Ver por ejemplo qué produce el siguiente código:

```
f = plt.figure()
ax1 = f.add_axes([.1, .1, .85, .6])
ax2 = f.add_axes([.8, .8, .18, .18])
ax3 = f.add_axes([.2, .2, .5, .1])
ax1.set_xlim([0, 1000])
ax3.set_ylim([0.1, 0.3])
plt.show()
```

Varias lineas en el mismo grafico

Para graficar varias series de datos en el mismo gráfico se puede llamar a una función que grafique varias veces. Por ejemplo, si queremos graficar las funciones seno y coseno en el mismo gráfico, podemos proceder así:

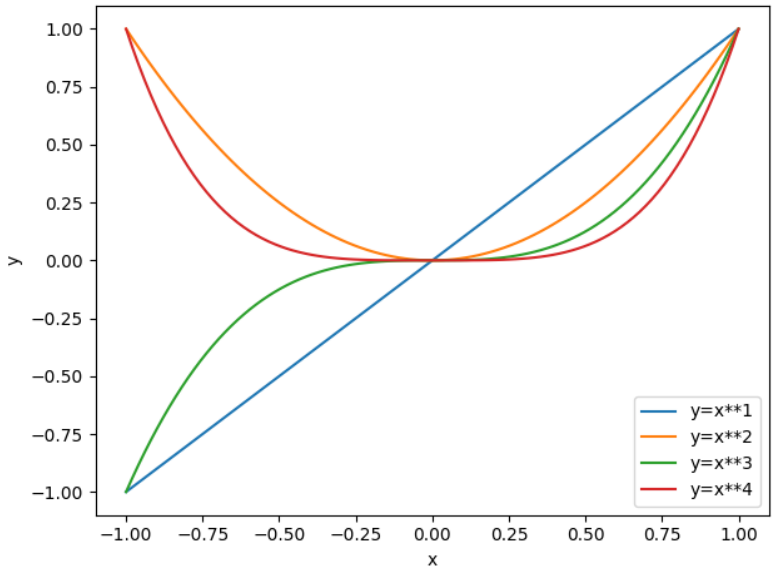
```
from matplotlib import pyplot as plt
x = np.linspace(-1, 1, 100)
y1 = x
y2 = x**2
y3 = x**3
y4 = x**4

fig = plt.figure()
fig.clf()
ax = fig.subplots(1,1)

ax.plot(x, y1, label='y=x**1')
ax.plot(x, y2, label='y=x**2')
ax.plot(x, y3, label='y=x**3')
ax.plot(x, y4, label='y=x**4')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('x')
ax.set_ylabel('y')

ax.legend()
fig.tight_layout()
fig.show()
```

Que da algo así:

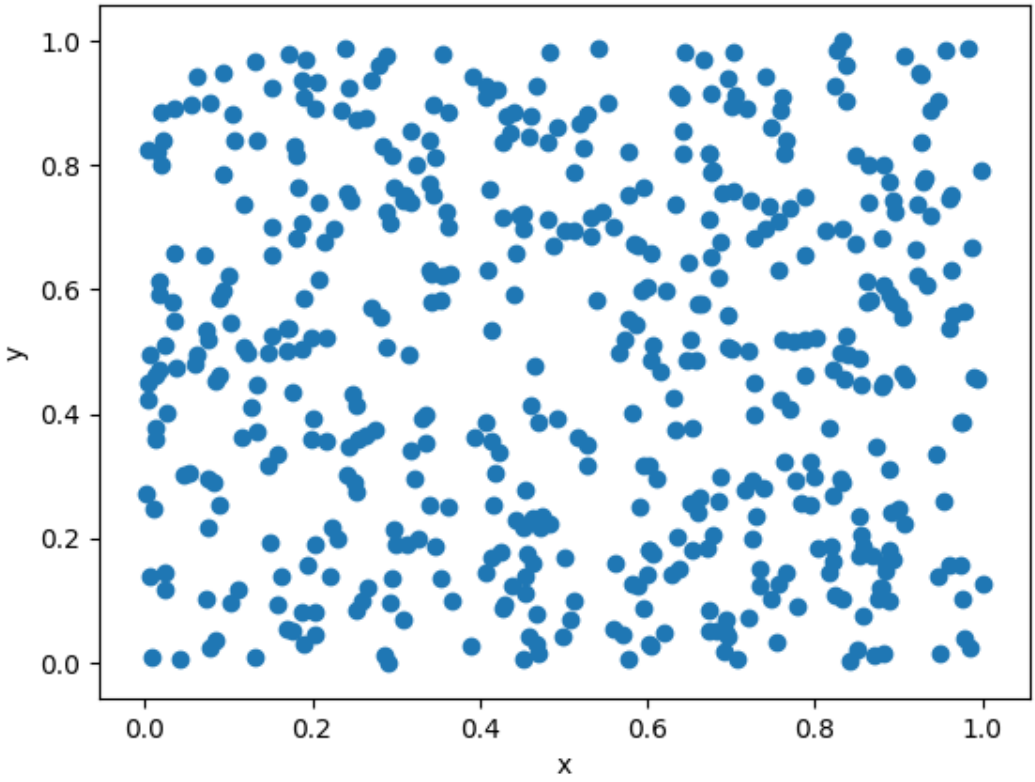


Atributos de los ejes

Se pueden modificar los atributos de los ejes, para lo cual primero hay que identificar los diferentes elementos. Las líneas de los ejes que marcan los valores de la escala se llaman `ticks`, cada tick tiene una valor, que está dentro de un rango determinado.

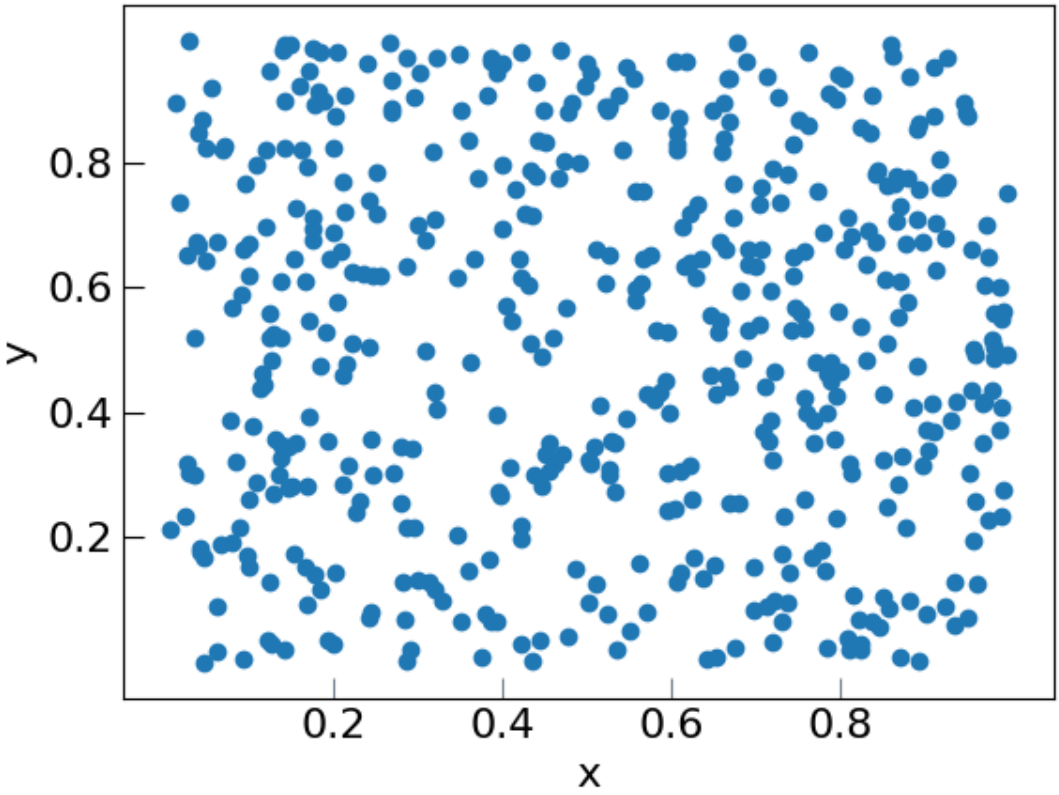
Comencemos con el siguiente gráfico simple y tratemos de mejorarlo un poco:

```
fig, ax = plt.subplots()
N = 500
x = np.random.rand(N)
y = np.random.rand(N)
plt.scatter(x, y)
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```



Queremos cambiar la apariencia del texto usado para etiquetar (`labels`) las lineas que marcan la escala (`ticks`). Esto es común porque en general hace falta agrandar la fuente del texto para que el gráfico sea legible al ser mostrado en distintos medios (por ej. una presentación). Usaremos las siguientes funciones de

```
fig, ax = plt.subplots()
N = 500
x = np.random.rand(N)
y = np.random.rand(N)
plt.scatter(x, y)
ax.set_xlabel('x', fontsize=16)
ax.set_ylabel('y', fontsize=16)

ticks = [.2, .4, .6, .8]
labels = ['0.2', '0.4', '0.6', '0.8']
ax.set_xticks(ticks=ticks)
ax.set_xticklabels(labels=labels, fontsize=16)
ax.set_yticks(ticks=ticks)
ax.set_yticklabels(labels=labels, fontsize=16)
ax.tick_params(axis='x', direction='in', length=8, color='slategrey')
ax.tick_params(axis='y', direction='in', length=8)
plt.show()
```

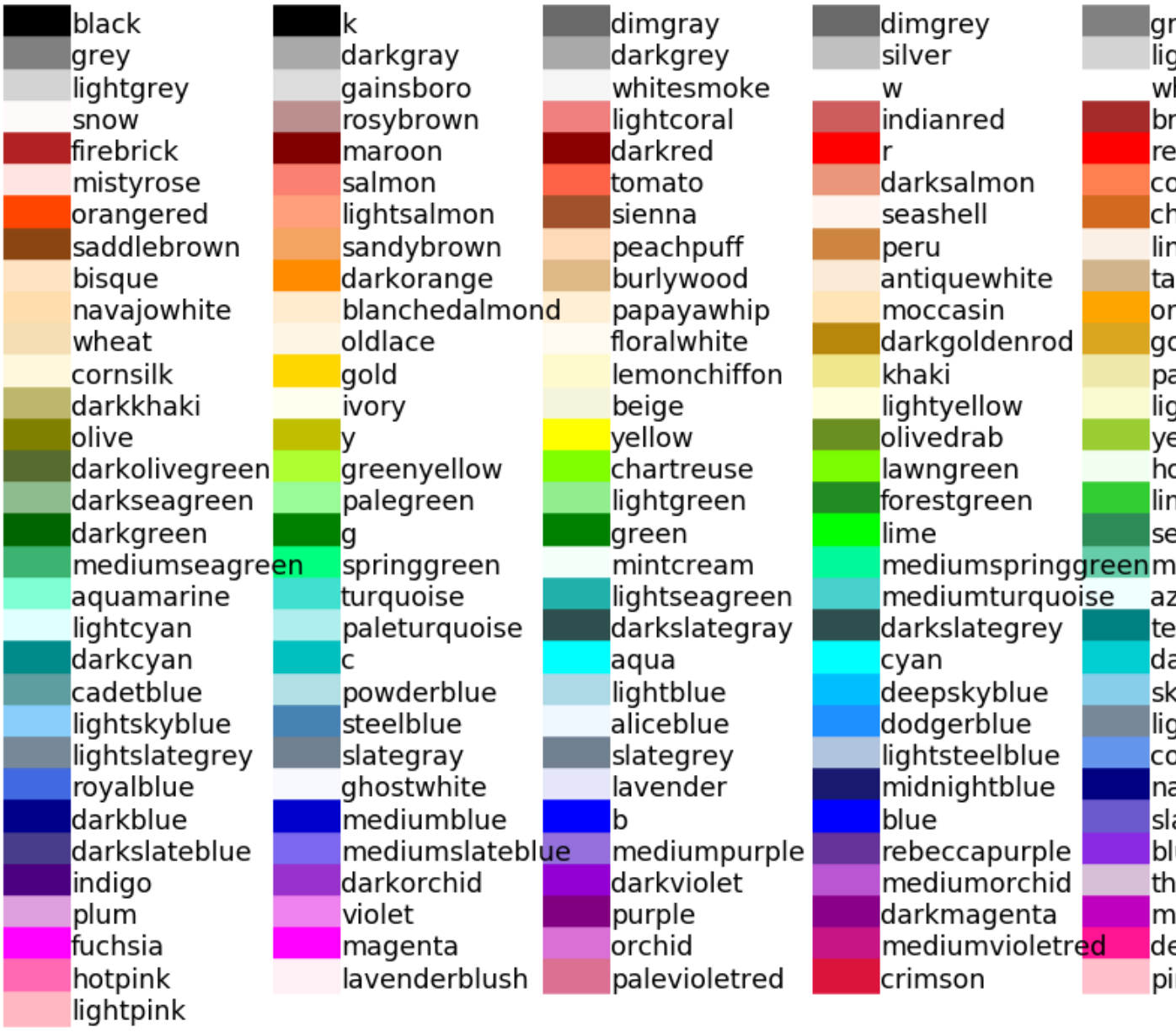


Atributos de las series de datos

Ahora tratemos de mejorar el contenido de los plots. Hay muchos atributos para trabajar, los más comunes son:

atributo	modifica	opciones
alpha	transparencia	escalar
color or c	color	color de matplotlib
label	etiqueta	cadena de carateres
linestyle or ls	tipo de linea	`[ '-' | '--' | '-.' | ':' | 'steps' | ...]`
linewidth or lw	ancho de linea	escalar
marker	marcador	`[ '+' | ',' | '.' | '1' | '2' | '3' | '4' ]`
markeredgecolor or mec	color de borde de marcador	color de matplotlib
markeredgewidth or mew	grosor del marcador	escalar
markerfacecolor or mfc	color de relloeno marcador	color de matplotlib
markersize or ms	tamaño del marcador	escalar
markevery	un marcador cada...	entero

Entre los colores de Matplotlib, los más comunes se pueden usar con nombre:



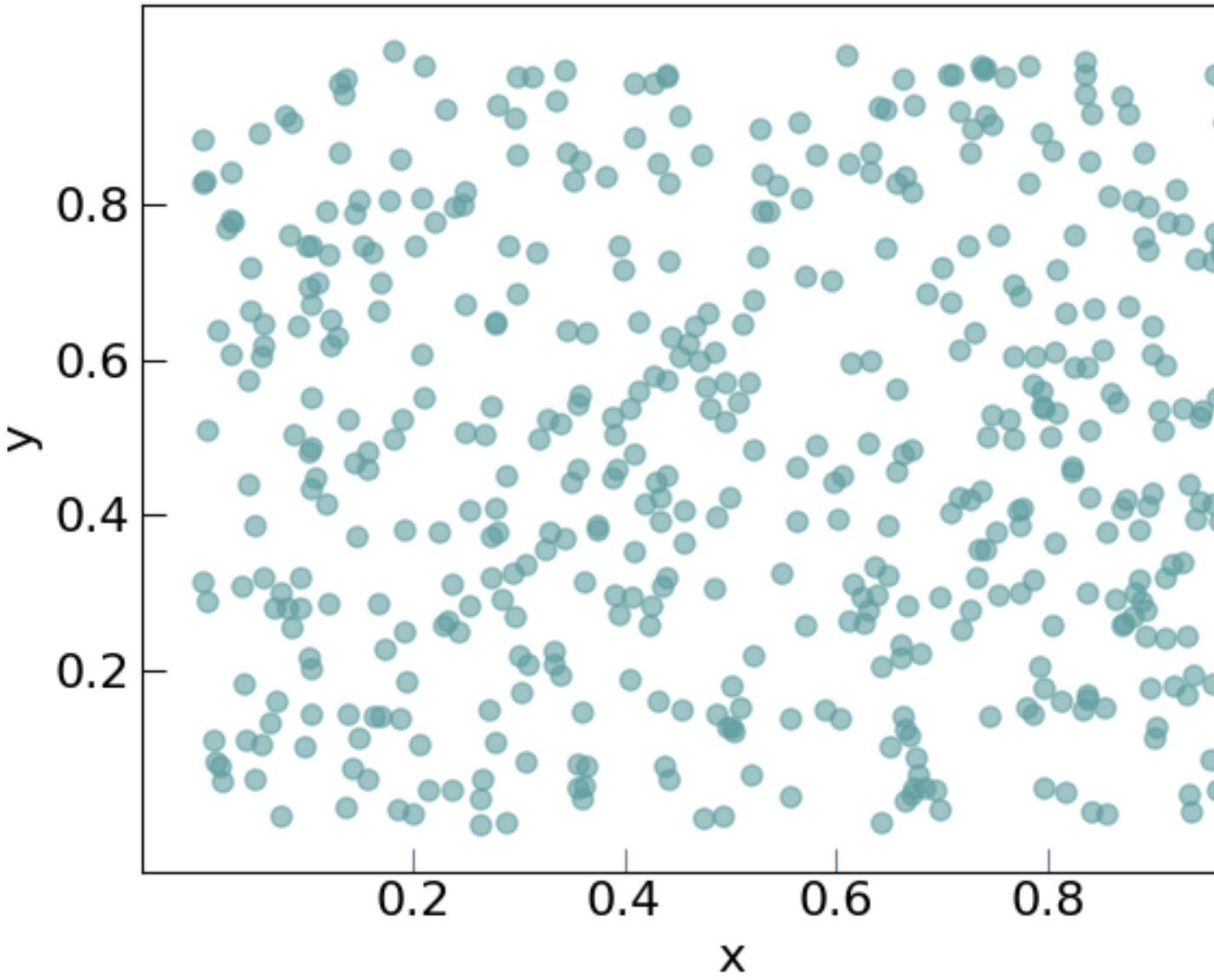
Veamos ahora algunos gráficos donde hemos cambiado varios atributos. La sintaxis es bastante simple y es posible entender cómo funciona leyendo el código:

```
fig, ax = plt.subplots()
N = 500
x = np.random.rand(N)
y = np.random.rand(N)
plt.scatter(x, y, s=44, color='cadetblue', alpha=0.6)
ax.set_xlabel('x', fontsize=16)
ax.set_ylabel('y', fontsize=16)

ticks = [.2, .4, .6, .8]
labels = ['0.2', '0.4', '0.6', '0.8']
ax.set_xticks(ticks=ticks)
ax.set_xticklabels(labels=labels, fontsize=16)
```

(continues on next page)

(continued from previous page)

```
ax.set_yticks(ticks=ticks)
ax.set_yticklabels(labels=labels, fontsize=16)
ax.tick_params(axis='x', direction='in', length=8, color='slategrey')
ax.tick_params(axis='y', direction='in', length=8)
plt.tight_layout()
plt.show()
```



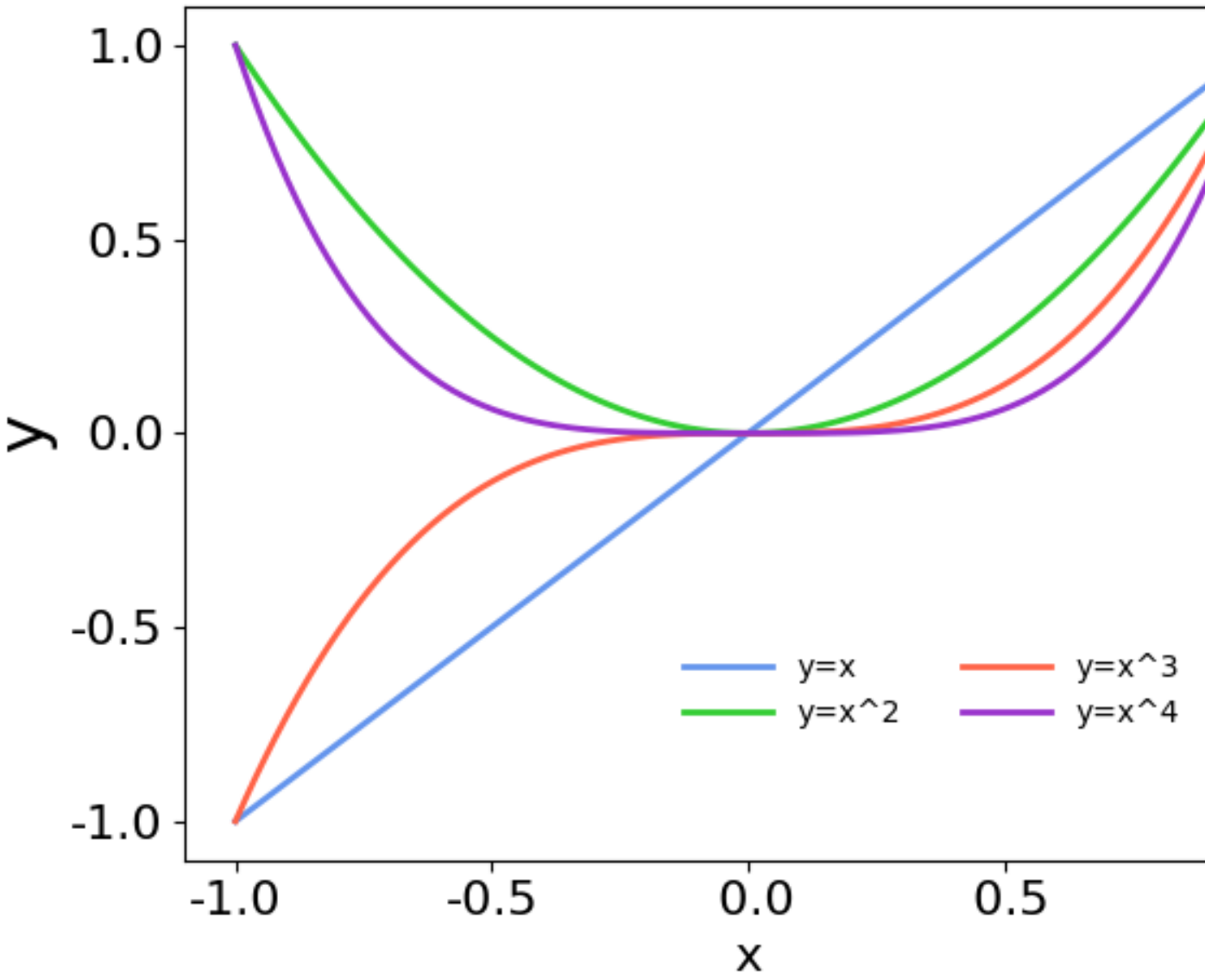
o con líneas:

```
from matplotlib import pyplot as plt
x = np.linspace(-1, 1, 100)
y1 = x
y2 = x**2
y3 = x**3
y4 = x**4
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure()
fig.clf()
ax = fig.subplots(1,1)

ax.plot(x, y1, color='cornflowerblue', linewidth=2, label='y=x')
ax.plot(x, y2, color='limegreen', linewidth=2, label='y=x^2')
ax.plot(x, y3, color='tomato', linewidth=2, label='y=x^3')
ax.plot(x, y4, color='darkorchid', linewidth=2, label='y=x^4')


ticks = [(-1.0 + 0.5*i) for i in range(5)]
labels = [f"{s: 2.1f}" for s in ticks]

ax.set_xticks(ticks=ticks)
ax.set_xticklabels(labels=labels, fontsize=16)
ax.set_yticks(ticks=ticks)
ax.set_yticklabels(labels=labels, fontsize=16)
ax.set_xlabel('x', fontsize=22)
ax.set_ylabel('y', fontsize=22)
ax.legend(loc='lower right', frameon=false,
          borderaxespad=4,
          ncol=2, handlelength=3)
ax.xaxis.label.set_size(16)
fig.tight_layout()
fig.show()
```



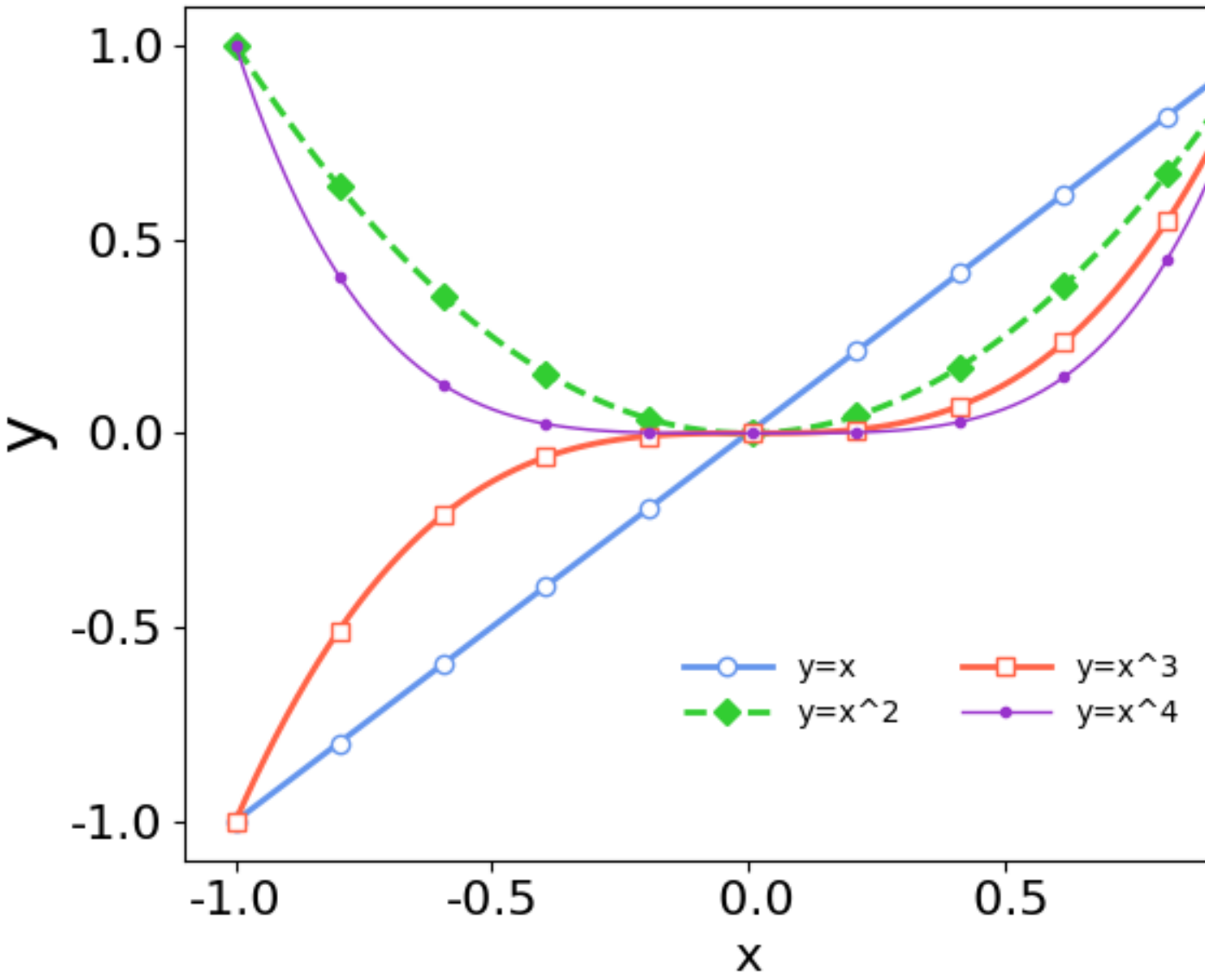
Por último veamos cómo modificar las líneas incluyendo marcadores. El siguiente código implementa varios tipos de marcadores para mostrar cómo se usa. No están explicados en detalles, pero habiendo seguido este tutorial es fácil buscar cómo se usan e incluso explorar muchas más opciones para graficar.

```
from matplotlib import pyplot as plt
import numpy as np
x = np.linspace(-1, 1, 100)
y1 = x
y2 = x**2
y3 = x**3
y4 = x**4

fig = plt.figure()
fig.clf()
ax = fig.subplots(1,1)
```

(continues on next page)

(continued from previous page)

```
ax.plot(x, y1, color='cornflowerblue', linewidth=2, label='y=x',
        linestyle='-', marker='o', markerfacecolor='white',
        markeredgewidth=1, markersize=6, markevery=10, alpha=1)

ax.plot(x, y2, color='limegreen', linewidth=2, label='y=x^2',
        linestyle='--', marker='D', markerfacecolor='limegreen',
        markeredgewidth=1, markersize=6, markevery=10, alpha=1)

ax.plot(x, y3, color='tomato', linewidth=2, label='y=x^3',
        linestyle='-', marker='s', markerfacecolor='white',
        markeredgewidth=1, markersize=6, markevery=10, alpha=1)

ax.plot(x, y4, color='darkorchid', linewidth=1, label='y=x^4',
        linestyle='-', marker='o', markerfacecolor='darkorchid',
        markeredgewidth=1, markersize=3, markevery=10, alpha=1)


ticks = [(-1.0 + 0.5*i) for i in range(5)]
labels = [f"{s: 2.1f}" for s in ticks]

ax.set_xticks(ticks=ticks)
ax.set_xticklabels(labels=labels, fontsize=16)
ax.set_yticks(ticks=ticks)
ax.set_yticklabels(labels=labels, fontsize=16)
ax.set_xlabel('x', fontsize=22)
ax.set_ylabel('y', fontsize=22)
ax.legend(loc='lower right', frameon=False,
          borderaxespad=4,
          ncol=2, handlelength=3)
ax.xaxis.label.set_size(16)
fig.tight_layout()
fig.show()
```



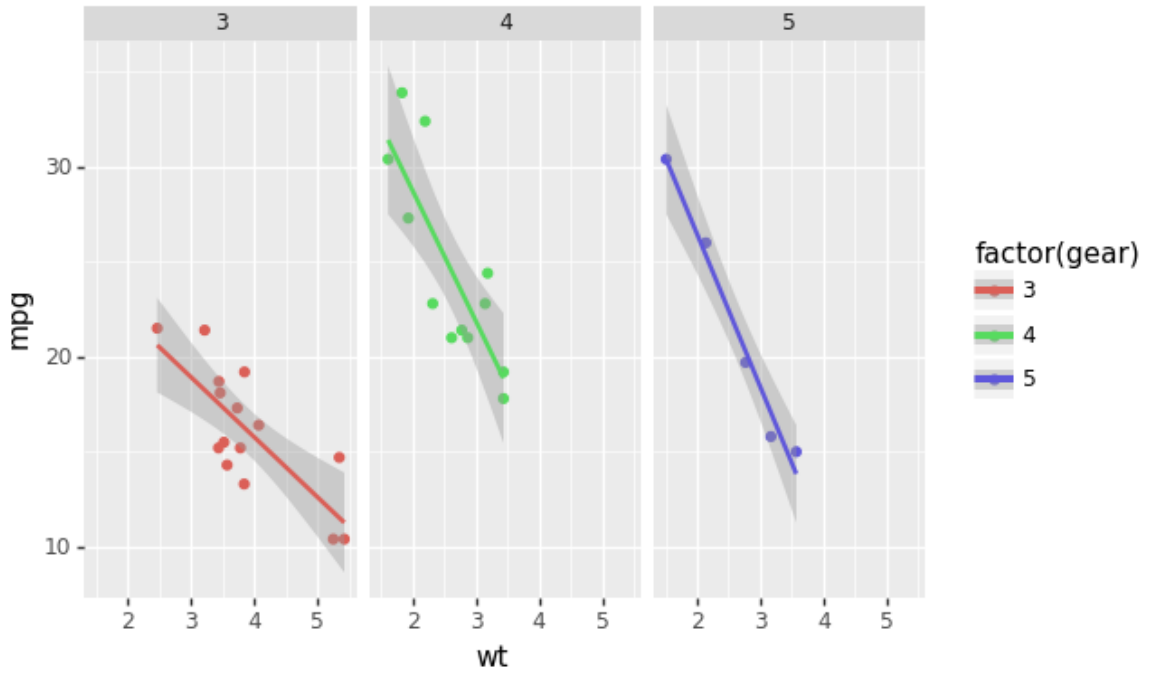
2.6.5 Gramática de gráficos con plotnine

Una forma natural de hacer gráficos es mediante el "diseño orientado a objetos", que se describe en el libro "The grammar of graphics", de L. Wilkinson, 2005, Springer-Verlag, New York.

Las ideas de este libro están implementadas en una librería de R muy popular, que se llama ggplot2.

Existe una versión para python, denominada plotnine. Este paquete permite hacer gráficos complejos con muchas menos líneas de código comparado por ejemplo con matplotlib.

Supongamos por ejemplo que tenemos datos de vehículos, y queremos graficar el rendimiento del combustible en función del peso, diferenciando según la cantidad de marchas:



Este gráfico se puede hacer con la siguiente idea:

- tengo los todos datos en una tabla
- elijo los datos que quiero graficar
- elijo la estética
- agrego un ajuste lineal con su error

la implementación es la siguiente:

```
from plotnine import ggplot, geom_point, aes, stat_smooth, facet_wrap
from plotnine.data import mtcars

(ggplot(mtcars, aes('wt', 'mpg', color='factor(gear)'))
 + geom_point()
 + stat_smooth(method='lm')
 + facet_wrap('~gear'))
```

De esta forma el gráfico está hecho en "capas". Veamos esto más en detalle:

1. Con ggplot se crea una instancia de un gráfico. Los argumentos de ggplot con la fuente de datos y la selección de los datos a graficar:

   ```
   ggplot(mtcars, aes('wt', 'mpg', color='factor(gear)'))
   ```

2. Una vez que se seleccionaron los datos, hay que elegir un atributo gráfico con el cual se quieren representar. Por ejemplo, con puntos:

   ```
   estilo = geom_point()
   ```

3. Se puede agregar algún análisis estadístico. Por ejemplo un ajuste lineal:

   ```
   stats = stat_smooth(method='lm')
   ```

4. Y finalmente, elegimos separar los gráficos según la cantidad de marchas:

```
paneles = facet_wrap('~gear')
```

Esto ultimo se llama "faceting". Ahora se puede rehacer el gráfico de una forma más evidente:

```
from plotnine import ggplot, geom_point, aes, stat_smooth, facet_wrap
from plotnine.data import mtcars

plot = ggplot(mtcars, aes('wt', 'mpg', color='factor(gear)'))
estilo = geom_point()
stats = stat_smooth(method='lm')
paneles = facet_wrap('~gear')

plot + estilo + stats + paneles
```

Se puede probar agregar o sacar capas, por ejemplo:

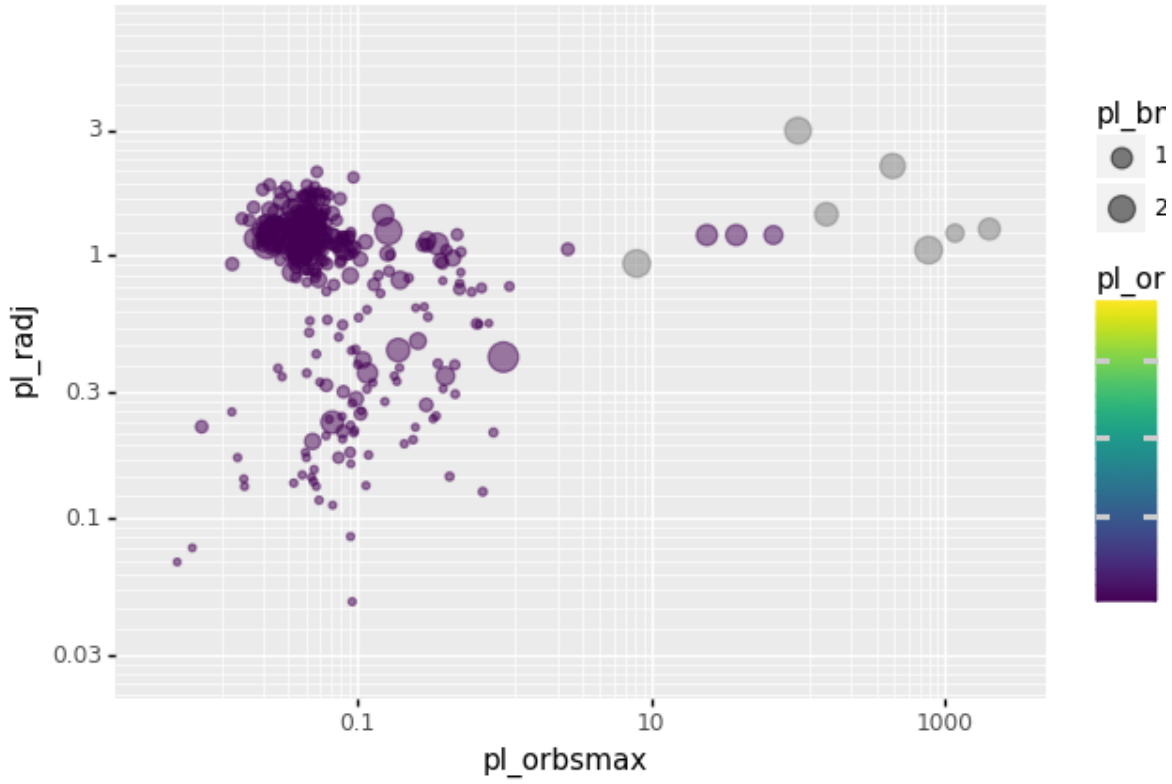
```
plot + estilo
```

hace el gráfico sin las estadísticas y en un solo panel. Se puede agregar o sacar el ajuste de la recta y poner en paneles o no:

```
plot + estilo + stats
# o bien:
plot + estilo + paneles
```

Para dar otro ejemplo, supongamos que queremos estudiar relaciones en los parámetros de una base de datos de planetas extrasolares. El siguiente gráfico es un gráfico de dispersión del radio vs. el semieje mayor de la órbita de un conjunto de exoplanetas, donde el color indica el periodo y el tamaño de los puntos la masa*sin(i).

```
D=pd.read_csv('planets.csv')

axes = aes('pl_orbsmax','pl_radj',color='pl_orbper')
points = geom_point(aes(size='pl_bmassj'), alpha=0.5)

plt = ggplot(D, axes) + points
plt + scales.scale_x_log10() + scales.scale_y_log10()
```



Resources:

'Clearing the confusion once and for all: fig, ax = plt.subplots()<https://towardsdatascience.com/clearing-the-confusion-once-and-for-all-fig-ax-plt-subplots-b122bb7783ca>'_

Python for astronomers: plotting

CHAPTER 3

Herramientas

3.1 GIT

Existen herramientas que permiten tener un registro de las distintas versiones de un conjunto de archios que se va modificando con el tiempo. Esto es muy útil para el desarrollo de códigos o de proyectos de software. Además de que permiten volver a versiones anteriores, estos sistemas son muy útiles para tener una copia de respaldo usando servicios en la nube y para compartir código fácilmente.

Los sistemas de control de versión son:

- Locales: sólo accesibles desde una PC local
- Centralizados: varias PC se conectan a un servidor central que guarda el historial completo
- Compartidos: varias PC guardan el historial completo

Para cada uno de esos sistemas hay herramientas o programas que lo implementan:

- Locales: copias locales “a mano”, RCS
- Centralizados: CVS, Subversion (SVN), Performance
- Compartidos: Git, Mercurial, Bazaar, Darcs

Usaremos GIT, que es uno de los más populares en la actualidad (por ejemplo se usa para el desarrollo del kernel de Linux).

Existen además plataformas web o servicios de intranet para administrar o almacenar los repositorios. Ejemplos de servicios web para GIT son: GitHub, GitLab, Assembla. Usaremos GitHub.

3.1.1 Estados de los archivos en Git

En un área de trabajo (e.g., un directorio) que incluye control de versión con Git, hay archivos en distintos estados:

- no incluidos en el control de version (untracked): forman parte del área de trabajo pero no del repositorio de Git.

- modificados: los archivos del repositorio que hay sido modificados
- marcados (staged): los archivos modificados que hay sido marcados para ser luego agregados a una nueva versión.
- sometidos (comitted): los archivos agregados a la base de datos en alguna versión.

3.1.2 Uso de Git

Se puede usar Git de dos formas:

- Línea de comandos
- GUIs (incluyendo varios IDEs, como VSCODE o Spyder)

La línea de comandos ofrece todas las funcionalidades de Git, mientras que en general las GUIs ofrecen algunas.

3.1.3 Instalación y configuración inicial

La instalación se realiza según el sistema (Linux, Windows, MacOS).

La configuración inicial se hace con el comando:

```
$ git config
```

Las operaciones de red de Git usan un identificador de cada usuario, por lo que lo pimero que hay que hacer es definir la identidad:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Otra cosa importante es el editor para interactuar con diferentes pasos del proceso:

```
$ git config --global core.editor vim
```

Se puede elegir el editor favorito, como vim, nano, emacs, gedit, etc.

Los parámetros de configuración se pueden ver con:

```
$ git config --list
```

3.1.4 Git en línea de comandos

Aquí veremos los comandos básicos para trabajar con Git.

Para obtener la ayuda:

```
$ git help
```

y para un comando en particular:

```
$ git help add
$ git add -h
```

En general se inicia un repositorio clonando uno existente o creando uno nuevo de manera local. Lo más fácil es crear un reposotorio en GitHub y luego clonarlo. Para ello:

```
$ git clone URL
```

donde el URL es el correspondiente al proyecto en GitHub.

Para agregar archivos (pasar el área "staged"):

```
$ git add archivo
```

Para agregar esos archivos a la base de datos del control de versión:

```
$ git commit -m "mensaje"
```

Para saber el estado de los archivos:

```
$ git status
```

Para obtener una lista de las versiones a las que se puede acceder:

```
$ git log
```

Este comando devuelve los nombres de los commits (generados automáticamente, mediante una HASH del tipo SHA1), como así también el autor, fecha y comentarios. Una versión resumida que da solamente el nombre:

```
$ git log --oneline
```

Se puede volver a una de las versiones identificadas con el ID del commit:

```
$ git checkout 833372b
```

Luego de revisar una versión anterior, se puede volver a la última versión con:

```
$ git checkout master
```

donde "master" es el nombre de la rama principal. Si se quiere desarrollar una versión "alternativa" del código, se puede hacer otra rama. Por ejemplo, supongamos que la lista de commits es la siguiente (basado en el video de la clase):

```
46e89c8 (HEAD -> master, origin/master, origin/HEAD) Corregi errores de ortografia
959f79a agrego prueba.py
833372b Initial commit
```

queremos volver al segundo commit y hacer un desarrollo en paralelo:

```
$ git checkout 959f79a
$ git checkout -b remix
```

Una revisión completa de las funcionalidades de Git se puede encontrar en el libro Pro Git.

CHAPTER 4

Ayuda para las guias

4.1 Ayuda para la guia 1

CHAPTER 5

Búsqueda

- search