Introducción a Python para cálculo científico

A. Garcimartín

Material del curso
Lenguajes de Programación

Máster en Métodos Computacionales en Ciencias

Introducción a Python para cálculo científico

A. Garcimartín

ISBN 978-84-8081-728-8

Sprinter Verlag, Buztintxuri-Barañain-Heidelberg (2022)

CONTENIDOS

Python básico

1. Tipos de variables
2. Operadores básicos
3. Colecciones: listas y tuplas
4. Métodos. Métodos para listas
5. Control del flujo: ramificación
6. Control del flujo: bucles
7. Modulos. Módulos numpy y matplotlib
8. Strings y texto; métodos
9. Funciones
10. Clases
11. Entrada y salida: ficheros
12. Más allá

NumPy

1. Objetos y atributos
2. Métodos: creación y unión de arrays
3. Funciones universales
4. Rutinas matemáticas

matplotlib

1. El módulo matplotlib. Gráficos sencillos
2. Resumen de los comandos más usuales de matplotlib.pyplot
3. Documentación y ejemplos

Tópicos adicionales

1. Tuplas: *packing & unpacking*
2. Formateo del texto
3. *List comprehension*
4. El módulo random
5. Métodos con funciones lambda
6. El módulo SymPy

Python básico

Este documento contiene un resumen personal del lenguaje Python. Los comandos de Python se muestran con su salida correspondiente tal como aparecen en una consola interactiva, con el input *indicado así:* >>> *, o bien en una "celda" de Jupyter Notebook con el* input *sombreado.*

1. Tipos de variables

Los tipos de variables más usados son: números enteros (`int`), números reales (`float`) y cadenas de caracteres (`str`). Esas palabras clave se emplean además para convertir de un tipo a otro.

```
>>> n1 = 3  # Python elige automáticamente el tipo de variable
>>> type(n1)
<class 'int'>
>>> x1 = 3.1416
>>> type(x1)
<class 'float'>
>>> s1 = '7 de julio' # las comillas simples ' y dobles " son equivalentes
>>> type(s1)
<class 'str'>
```

Para mostrar en pantalla se emplea la función `print`, en cuya sintaxis se pueden incluir referencias a variables que se rellenan según su valor.

```
>>> f1 = float(n1)
>>> s2 = str(n1)
>>> print(f1)
3.0
>>> type(f1)
<class 'float'>
>>> print(s2)
3
>>> type(s2)
<class 'str'>
>>> nombre = "Zendaya"
>>> edad = 25
>>> print(nombre, 'tiene', edad, "años")
Zendaya tiene 25 años
>>> print("{} tiene {} años".format(nombre,edad))
Zendaya tiene 25 años
>>> print(f'{nombre} tiene {edad} años') # nótese la f al principio
Zendaya tiene 25 años
```

2. Operadores básicos

2.1. Operaciones algebraicas

Operador	*Significado*	*Ejemplo*
+	suma	x+y+2
-	resta	x-y-2
*	multiplicación	x*y
/	división	x/y
%	resto (de la división)	x%y
**	potencia	x**y (daría x^y)

Observación.- Python es un lenguaje orientado a objeto. Eso quiere decir que los operadores pueden actuar de manera diferente según el método definido para cada objeto.

```
>>> cad1 = 'a'
>>> cad2 = 'h'
>>> cad3 = 'g'
>>> cad4 = cad1 + (cad2*5) + cad3 + '!'
>>> print(cad4) # para strings: + concatenación, * repetición
ahhhhhg!
```

2.2. Operadores de asignación

Operador	*Ejemplo*	*Equivalente a*
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5

2.3. Comparaciones

Las siguientes operaciones dan como resultado una variable lógica que solo puede tomar los valores `True` o `False`.

Operador	*Significado*	*Ejemplo*
==	igual que	x == y
!=	distinto a	x != y
<	menor que	x < y
<=	menor o igual que	x <= y
>	mayor que	x > y
>=	mayor o igual que	x >= y

2.4. Operadores lógicos

El operador `in` devuelve `True` si el valor está en el objeto; en caso contrario devuelve `False`. Además, se pueden emplear las conjunciones lógicas `and`, `not`, y `or`.

```
>>> 5 >= 6
False
>>> 5 != 6
True
>>> (5 >= 6) or (5 != 6)
```

```
True
>>> v = [1,2,3] # definición de v (una lista; ver más adelante)
>>> 1 in v      # ¿está 1 entre los elementos de v?
True
>>> 5 in v      # ¿está el 5?
False
>>> (5 in v) or (1 in v)  # ¿está el 5 o el 1?
True
>>> (5 in v) and (1 in v) # ¿están el 5 y el 1?
False
>>> (5 in v) or not(6 in v) # ¿está el 5 o falta (no está) el 6?
True
```

3. Colecciones: listas y tuplas

Las colecciones de objetos más usuales en Python son:

- **list** Colección de elementos ordenados, mutable (es decir, los elementos de una lista se pueden cambiar). Admite duplicados (o sea, elementos repetidos). Se crea agrupando los elementos entre corchetes y separados por comas: `[ , , ... ]`.
- **tuple** Colección de elementos ordenado, inmutable. Admite duplicados. Se crea agrupando los elementos separados por comas. Para que facilitar la lectura se suelen añadir paréntesis, pero no son imprescindibles: `( , , ... )`.
- **set** Colección de elementos no ordenada, no indexada, inmutable, sin duplicados. Se crea con llaves `{ , , ... }` o con la función `set( )`, que toma como argumento una lista.
- **dictionary** Colección ordenada, mutable, de elementos agrupados por pares. No admite duplicados. Los pares se agrupan con `:`, y el conjunto con `{ ... }`.

Los elementos de las colecciones pueden ser de cualquier tipo, mezclados en el mismo objeto (una lista, por ejemplo, puede contener números, texto, una tupla, y otra lista).

Los más usuales son las listas y las tuplas. Los *sets* se suelen emplear cuando se trabaja con álgebra de conjuntos, y los diccionarios para referirse a valores específicos por otro nombre. Para acceder a un elemento de una colección se emplean los corchetes. ***Los índices en Python empiezan en cero***. En lo sucesivo se usa una lista; a los elementos de las tuplas se accede de manera similar. Los `sets` y los `dictionaries` no admiten indexación.

```
>>> primos = [2,3,5,7,11,13,17]
>>> type(primos)
<class 'list'>
>>> print(primos[1]) # ¡no es el primero, es el segundo!
3
>>> primos[1]=19 # una lista es mutable (se puede cambiar)
>>> print(primos)
[2, 19, 5, 7, 11, 13, 17]
>>> Letras = ('A','B','C','D')
>>> print(Letras[0])
A
>>> type(Letras)
<class 'tuple'>
>>> Letras[0]='Z' # imposible; la tupla es inmutable
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Los diccionarios son pares *clave / valor*; si se solicita la clave, devuelve el valor.

```
>>> Contactos = {
... "Cara Delevigne":"310-421-0894",
... "Jennifer Lopez":"305-690-0379",
... "Hugh Jackman":"212-596-7998"}
>>> type(Contactos)
<class 'dict'>
>>> Tel_JLo = Contactos["Jennifer Lopez"]
>>> print(Tel_JLo)
305-690-0379

>>> PIGS = {'Portugal','Italy','Greece','Spain'}
>>> type(PIGS)
<class 'set'>
>>> 'Italy' in PIGS
True
>>> 'France' in PIGS
False
```

Indices y secciones

Para obtener un subconjunto contiguo de los elementos de una lista se emplea la sintaxis `lista[inicio:fin:paso]`, donde el índice inicio se incluye pero el fin no. Hay distintas variantes (consúltese Understanding Slicing en stackoverflow):

```
>>> impares = [1,3,5,7,9,11,13,15,17,19]
>>> impares[0] # el primero
1
>>> impares[0:3] # si se omite el paso, es 1
[1, 3, 5]
>>> impares[0:5:2] # de dos en dos hasta el sexto (sin incluir)
[1, 5, 9]
>>> impares[::2] # todos de dos en dos
[1, 5, 9, 13, 17]
>>> impares[:5] # todos hasta el sexto, sin incluirlo
[1, 3, 5, 7, 9]
>>> impares[-1] # índices negativos: empezar por el final
19
>>> impares[-1:-3:-1] # los dos últimos (el tercero por el final ya no)
[19, 17]
>>> impares[-1:0:-1] # desde el último (-1) hasta el primero (0)
[19, 17, 15, 13, 11, 9, 7, 5, 3]
>>> impares[-2:] # los dos últimos
[17, 19]
```

A los elementos de una lista contenida dentro de otra lista se accede mediante dos índices:

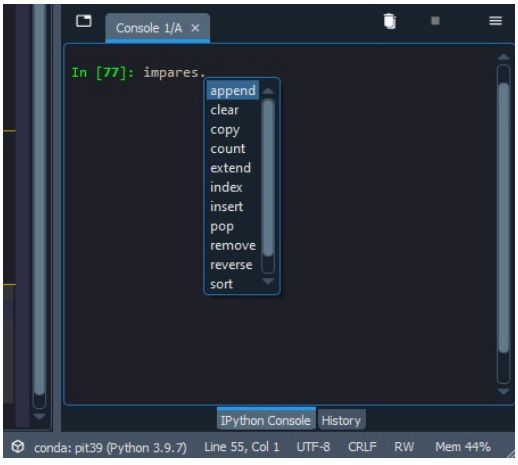
```
>>> pecera1 = ['betta','escalar','guppy']
>>> pecera2 = ['cebra','neón']
>>> acuario = [pecera1,pecera2]

>>> pecera1[0]
'betta'
>>> acuario[0][2]
'guppy'
>>> acuario[1][0]
'cebra'
```

Las cadenas de caracteres admiten indexación:

```
>>> Nombre = 'Julia Roberts' # un nombre con las 5 vocales:
>>> Nombre[1]
'u'
>>> Nombre[3]
'i'
>>> Nombre[4]
'a'
>>> Nombre[7]
'o'
>>> Nombre[9]
'e'
```

4. Métodos. Métodos para listas

Cada objeto en Python tiene definidos una serie de procedimientos anejos que se le pueden aplicar: se les llama **métodos**. Las listas, por el mero hecho de serlo, tienen aparejados unos métodos. Se pueden ver los métodos aplicables a un objeto escribiendo en un intérprete `objeto.[Tab]`, donde `objeto` hay que sustituirlo evidentemente por el nombre de la variable, y [Tab] indica que con el cursor detrás del punto hay que apretar la tecla tabulador. El resultado es el que se muestra en la figura, para el caso de una lista.



Menú desplegable que aparece cuando se escribe el nombre de un objeto, seguido de un punto, al apretar la tecla Tabulador. En este caso, `impares` *es una lista.*

Métodos para listas

Operador	*Significado*
`append()`	añadir elemento
`extend()`	concatenar otra lista
`insert()`	insertar
`remove()`	quitar un *valor* de la lista
`pop()`	quita el último elemento
`count()`	cuántas veces sale un valor
`index()`	lugar donde aparece por primera vez un valor
`sort()`	ordenar (redefine la lista, ordenándola)
`reverse()`	da la vuelta a la lista (y la deja así)
`copy()`	crea una copia de la lista
`clear()`	borra los elementos de la lista (pero no la lista)

Los métodos se ejecutan con la sintaxis `objeto.método()`. Ejemplo:

```
>>> lista = [10,50,30] # definición de una lista
>>> lista2 = lista.copy() # una copia
>>> lista.sort() # este método ordena la lista
>>> print(lista) # y la lista queda cambiada
[10, 30, 50]
>>> print(lista2)
[10, 50, 30]
>>> lista.reverse() # da la vuelta a la lista de atrás adelante
>>> print(lista) # y la deja así, modificada
[50, 30, 10]
>>> lista.append(40) # añade el número 40
>>> lista3 = [20, 40, 60]
>>> lista.extend(lista3) # concatena lista3 a lista
>>> print(lista)
[50, 30, 10, 40, 20, 40, 60]
>>> lista.pop() # quita el último elemento
60
>>> print(lista)
[50, 30, 10, 40, 20, 40]
>>> lista.count(40) # ¿cuántas veces está el 40?
2
>>> animales = ['gallo', 'manzana', 'profesor', 'sardina']
>>> print(animales)
['gallo', 'manzana', 'profesor', 'sardina']
>>> animales.remove('manzana') # quitar de la lista ese elemento
>>> print(animales)
['gallo', 'profesor', 'sardina']
>>> animales.index('sardina') # ¿en qué puesto está 'sardina'?
2
>>> animales.insert(2,'estudiante') # el estudiante es también un animal
>>> print(animales)
['gallo', 'profesor', 'estudiante', 'sardina']
```

5. Control del flujo: ramificación

- la indentación es parte de la sintaxis: lo que se ejecuta condicionalmene es lo que está indentado.
- no se usan paréntesis; las condiciones deben evaluarse en `True` o `False`.
- se acaba la condición con :

5.1. Ramificación simple

Palabra clave: `if`

```
if condición :
    comando 1
    comando 2
    ...
```

donde la *condición* debe ser una expresión lógica que se evalúe como verdadera o falsa. Adviértase que los comandos están indentados; la acción que se ejecuta si la acción es cierta acaba con la indentación.

5.2. Ramificaciones múltiples

Palabras clave: `if`, `elif`, `else` (*elif* es una contracción de *else if*).

```
if condición A :
    comandos I
elif condición B :
    comandos II
elif condición C :
...
else :
    comandos IV
```

Si se cumple la *condición A*, se ejecutan los *comandos I*; en caso contrario (es decir, no se cumple *A*), si se cumple la *condición B*, se ejecutan los *comandos II*; en caso contrario (no se cumple ni *A* ni *B*), si se cumple *C*, etcétera, y si no se cumple ninguna de las condiciones (*A*, *B*, *C*, ...) se ejecuntan los *comandos IV*.

6. Control del flujo: bucles

Valen las mismas advertencias sobre la indentación que se han hecho para las ramificaciones.

6.1. Bucle con iterable

Palabras clave: `for`, `in`

```
for variable in secuencia :
    comandos
```

La *secuencia* debe ser una colección secuenciable: una lista, una tupla, un set, un diccionario o incluso una cadena de caracteres. La *variable* tomará sucesivamente cada uno de los posibles valores de la colección. Ejemplos:

[1]:
```
impares = [1,3,5]
for n in impares:
    print(n)
```

```
1
3
5
```

[2]:
```
Nombre = 'Martin Lutero'
for letra in Nombre:
    if letra in ['a','e','i','o','u']:
        print(letra)
```

```
a
i
u
e
o
```

[3]:
```
pecera1 = ['betta','escalar','guppy']
pecera2 = ['cebra','neón']
acuario = [pecera1,pecera2]
for elemento in acuario:
    print(elemento)
```

```
['betta', 'escalar', 'guppy']
['cebra', 'neón']
```

Adviértase que este bucle no utiliza un contador, sino una variable que toma sucesivamente cada uno de los valores de una colección finita numerable. A esa secuencia se le llama en este contexto un *iterable*.

6.2. Bucle con condición

Palabra clave: `while`

En el siguiente ejemplo se calcula la serie de Fibonacci (cuyo término general es la suma de los dos anteriores: $a_n = a_{n-1} + a_{n-2}$):

[1]:
```
a0 = 1 # por definición:
a1 = 1 # los dos primeros términos son 1
num_terms = 15 # número de términos
contador = 0
while contador < num_terms:
    a2 = a0 + a1
    a0 = a1
    a1 = a2
    contador = contador+1
    print(a2, end=' ') # así no acaba con nueva línea
```

```
2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Aunque el bucle anterior es sencillo, es más *Pythonesco* utilizar métodos para obtener códigos más compactos:

[2]:
```
fibo = [1,1]
num_terms = 17 # número de términos
contador = 2
while contador < num_terms:
    fibo.append(fibo[contador-1]+fibo[contador-2])
    contador = contador+1

for f in fibo:
    print(f,end=' ')
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Como en Python los bucles no gestionan contadores, debe ser el usuario el que se ocupe de esta tarea en caso necesario.

7. Módulos. Módulos numpy y matplotlib

Los **módulos** son paquetes de funciones encaminados a una tarea particular. La filosofía de Python lleva soslayar la pregunta *'¿Cómo se hace X en Python?'* en favor de la cuestión *'¿Qué módulo de Python hace X?'*. Eso hace de Python un lenguaje bastante somero (no posee gran cantidad de comandos o funciones), al que se le pueden añadir módulos para diferentes tareas. Hay módulos para calcular la función de supervivencia, para tratamiento de imágenes, o para tareas más prosaicas, como leer un formato específico de audio. Los módulos con frecuencia dependen unos de otros, por lo que es imperativo usar un instalador que tenga en cuenta tanto la versión de Python como las dependencias entre módulos. Un instalador muy popular es `conda`. Otro es `pip`. Pero no se pueden mezclar instaladores.

Python viene ya con muchos módulos (más de 200). Por ejemplo: `math`, `signal`, `email`, `pickle`, `random` ... La lista de los que vienen con Python 3.9 se puede encontrar en la documentación: Python Module Index. Unos pocos se usarán en este documento. Otros no vienen con la instalación básica, y se llaman módulos *descargables*.

Para el científico, son de obligada referencia los siguientes: SciPy (una colección de algoritmos científicos), NumPy (un módulo para cálculo científico, que contiene en particular muchas funciones matemáticas), matplotlib (un paquete de gráficos para visualización científica), pandas (análisis de datos y manipulación), scikit-image (procesamiento de imágenes), scikit-learn (*machine learning*) y SimPy (cálculo simbólico).

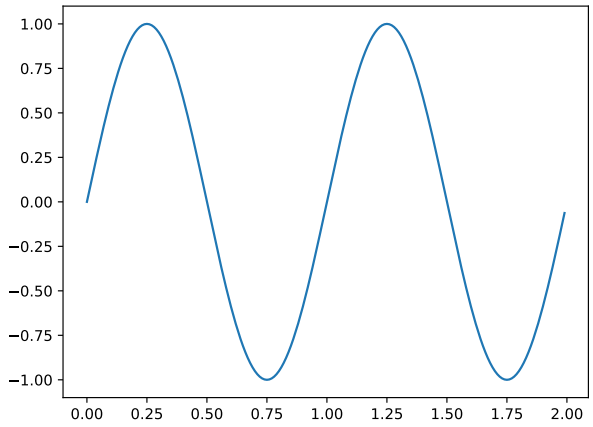
Una vez instalados, los módulos se cargan usualmente con el comando `import módulo as abreviatura`. A partir de ese momento, las funciones del módulo se pueden usar mediante la sintaxis `abreviatura.función`.

El manual de uso y la descripción de estos módulos es de una extensión tal que excede a este documento, pero se ofrece un breve resumen de NumPy y matplotlib en sendos capítulos. El módulo NumPy se basa en unos objetos de clase ***numpy array***. Vienen a ser listas cuyos elementos son todos del mismo tipo. Se comportan de manera muy parecida a los *arrays* de Matlab. El paquete incluye las funciones matemáticas elementales. El módulo matplotlib incluye MUCHAS funciones gráficas para representar datos y objetos matemáticos. La más básica quizá sea `plot`.

Se recomienda aprender a servirse de estas funcionalidades cuanto antes, pues son de gran ayuda.

```
>>> import numpy as np
```

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(0, 2, 0.01)
>>> y = np.sin(2 * np.pi * x)
>>> plt.plot(x, y)
[<matplotlib.lines.Line2D object at 0x00000244DA2B0D60>]
```



8. *Strings* y texto; métodos

Las cadenas de caracteres son secuencias de letras, con ciertos atributos y métodos especiales (por ejemplo: son inmutables, como las tuplas). Una de las ventajas de Python es que emplea la codificación Unicode, por tanto admite todas las letras del español (como las vocales acentuadas y la ñ). Dentro de las cadenas de caracteres se pueden insertar caracteres especiales (‘*escape characters*’) para formatear: `\n` es nueva línea, `\t` es tabulador, etcétera.

Algunos métodos específicos para *strings* son:

método	*acción*
`capitalize()`	Primera letra en mayúscula
`lower()`	pasar a minúsculas
`upper()`	pasar a mayúsculas
`count()`	número de ocurrencias de una subcadena
`find()`	índice donde aparece por primera vez una subcadena
`format()`	ejecutar una operación de formateo
`isnumeric()`	devuelve `True` si son números
`split()`	separa en subcadenas cortando en los espacios
`replace()`	reemplaza una subcadena por otra
`join()`	concatenar subcadenas

Evidentemente, hay muchos más. Puede consultarse la página correspondiente de docs.python.org.

```
>>> t = \
... 'y qué bien por la mañana \ncorrer en las vagonetas \n\
... llenas de nieve salada \nhacia las blancas casetas'
>>> print(t)
y qué bien por la mañana
correr en las vagonetas
llenas de nieve salada
hacia las blancas casetas
```

```
>>> n_i = t.count('i')
>>> print(n_i)
3
>>> print(t.upper())
Y QUÉ BIEN POR LA MAÑANA
CORRER EN LAS VAGONETAS
LLENAS DE NIEVE SALADA
HACIA LAS BLANCAS CASETAS
>>> s = "tres eran tres las hijas del rey"
>>> print(s.find('rey'))
29
>>> L = s.split()
>>> type(L)
<class 'list'>
>>> print(L)
['tres', 'eran', 'tres', 'las', 'hijas', 'del', 'rey']
>>> print(L[4])
hijas
>>> type(L[4])
<class 'str'>
>>> s2=s.replace('tres','dos')
>>> print(s2)
dos eran dos las hijas del rey
```

Observaciones.- La barra invertida (\) se usa para cortar una línea en el editor y seguir más abajo, evitando que desborde por la derecha. `L` es una lista cuyos elementos son *strings* (las palabras también son cadenas de caracteres).

9. Funciones

Si un procedimiento se repite muchas veces, lo más adecuado es crear una función. La primera motivación es, pues, evitar la repetición de código.

[1]:
```
def cubo(x):
    "el cubo de x"
    y = x*x*x
    return y
```

[2]:
```
help(cubo)
```

```
Help on function cubo in module __main__:

cubo(x)
    el cubo de x
```

[3]:
```
dos_3 = cubo(2)
print(dos_3)
```

```
8
```

[4]:
```
cinco_3 = cubo(5)
print(cinco_3)
```

```
125
```

La palabra clave es `def`, seguida de un paréntesis que toma los *argumentos de entrada* (en este caso es solo uno, `x`). La función devuelve uno o varios *argumentos de salida* a través de `return`; en este caso, el argumento de salida es `y`. La indentación es parte de la sintaxis. La cadena de la primera línea (opcional) es la ayuda sobre la función.

Las funciones se pueden definir de manera implícita, más compacta: se llaman *funciones lambda*, o funciones anónimas. La sintaxis es: `nombre = lambda parámetros: comando` (`lambda` es una palabra clave).

```
>>> triple = lambda x: x*3
>>> r = triple(2)
>>> print(r)
6
>>> triple(6)
18
```

10. Clases

Hasta este momento, se ha pasado por alto que Python es un *lenguaje orientado a objeto*. Eso quiere decir que cada elemento (sea una variable, un método, una función, etcétera) pertenece a una **clase**. Los lenguajes orientados a objeto constituyen un paradigma de programación, donde uno de los conceptos fundamentales es que cada objeto es una '*encarnación*', una '*implementación*', un '*caso*', una '*ocurrencia*' (en inglés: *instance*) de la clase, de la cual heredan las propiedades. Por poner un caso, las ventanas en un sistema operativo podrían ser una clase; hacer click en determinado icono podría '*instanciar*' un objeto de la clase ventana, y ese objeto tiene por ese hecho toda una serie de propiedades (**atributos**) aparejados –como el color de la barra–, así como ciertos **métodos** que se le pueden aplicar (por poner un caso: a una ventana se le pueden cambiar las dimensiones estirando con el ratón desde los bordes).

La clase define un tipo de objetos con ciertas propiedades, igual que en biología: los Panthera son un género de la familia de los felinos, o Felidae (por tanto se diría: "han heredado las propiedades de la clase Felidae"); el tigre (Panthera tigris) es un objeto de la clase Panthera; y mi tigre Sultán es una instancia de la clase Panthera tigris. Los padres de Sultán instanciaron un objeto de la clase tigre, y Sultán heredó las propiedades de la clase.

Los objetos pueden contener datos y código. Los datos se llaman *campos* o *atributos*, que pueden ser comunes a toda la clase (de los tigres) o particulares del objeto (de Sultán). El código contenido en el objeto son los *métodos* (los procedimientos) que el objeto puede ejecutar. Es de señalar que los métodos pueden cambiar los propios atributos del objeto: en ese caso hace falta una referencia al objeto `self`.

Por poner un caso, en Python hay una clase que son los números complejos. Los objetos de esta clase tienen un campo que es la parte real y otro que es la parte imaginaria. Además, estos objetos tienen definido el método *complejo conjugado*. (En vez de i, en Python se usa j para denotar $\sqrt{-1}$).

```
>>> z = 2 + 3j
>>> type(z)
<class 'complex'>
```

```
>>> print(z)
(2+3j)
>>> z.imag
3.0
>>> z.real
2.0
>>> z.conjugate()
(2-3j)
```

Una función, un elemento de un gráfico, un método, una variable, son objetos de una determinada clase. En realidad, en Python todo son objetos.

Aparte de todas las clases propias de Python, se pueden definir las que el usuario desee, especificando sus campos y sus métodos.

[1]:
```
class Pato:                    # definición de la clase Pato
    def volar(self):           # los patos pueden implementar este método
        print('Pato volando') # volar hace esto
    def nadar(self):           # otro método
        print('Pato nadando') # nadar hace esto
```

[2]:
```
UnPato = Pato() # instancia de Pato
```

[3]:
```
UnPato.nadar()  # método definido para pato
```

```
Pato nadando
```

[4]:
```
class Tigre:
    def __init__(self,nombre):  # init: se ejecuta al instanciarse
        self.nombre = nombre    # se le da un nombre al tigre
        self.trucos = []        # nace sin saber ningún truco
    def nadar(self):            # los tigres nadan
        print('Tigre nadando')
    def aprender_truco(self,truco): # y pueden aprender trucos
        self.trucos.append(truco)
```

[5]:
```
x = Tigre('Sultán')  # mi tigre se llama Sultán
type(x)
```

[5]:
```
__main__.Tigre
```

[6]:
```
print(x.nombre)
```

```
Sultán
```

[7]:
```
x.trucos   # no sabe nada
```

[7]:
```
[]
```

[8]:
```
x.aprender_truco('dar la pata') # le enseño a dar la pata
```

[9]:
```
x.trucos
```

[9]:
```
['dar la pata']
```

[10]:
```
x.nadar()
```

```
Tigre nadando
```

[11]:
```
x.cantar() # Sultán no sabe cantar: los tigres no cantan
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17768/4110289659.py in <module>
----> 1 x.cantar() # Sultán no sabe cantar: los tigres no cantan

AttributeError: 'Tigre' object has no attribute 'cantar'
```

Compruébese cómo el método `aprender_truco` modifica el atributo `trucos` del propio objeto. Por otra parte, hay que prestar atención porque *el mismo método, con el mismo nombre, puede ejecutar comandos diferentes dependiendo de la clase del objeto*. El método `nadar` hace algo diferente en el tigre que en el pato; por ejemplo, *raíz* podría ser un método que hiciera algo diferente en un número complejo que en un número real, incluso llamándose de la misma manera. Un comando que se llame *plot* podría hacer un gráfico diferente según el tipo de datos que se le pase como argumento.

Las funciones también son objetos. Una función puede ser argumento de otra función.

[12]:
```
import numpy as np

def DePi2(f):       # toma la función f como argumento
    z = f(np.pi/2) # halla f de pi/2
    return z       # y lo devuelve

def Doble(x):
    y = 2 * x
    return y

def Seno(x):
    y = np.sin(x)
    return y
```

[13]:
```
Seno(np.pi/2)
```

[13]:
```
1.0
```

[14]:
```
Seno(0)
```

[14]:
```
0.0
```

[15]:
```
Doble(np.pi/2) # esto no es sorprendente
```

[15]:
```
3.141592653589793
```

[16]:
```
DePi2(Doble)   # pero esto sí que es raro
```

[16]:
```
3.141592653589793
```

[17]:
```
DePi2(Seno)    # y esto
```

[17]:
```
1.0
```

Las funciones se pueden llamar a sí mismas (propiedad conocida como *recursividad*). Una función que se llama a sí misma es el factorial de x, que se puede definir como el producto de x por el factorial de $x-1$, hasta llegar al 1, cuyo factorial es 1.

[18]:
```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

num = 3
print("El factorial de", num, "es", factorial(num))
```

```
El factorial de 3 es 6
```

11. Entrada y salida: ficheros

La interacción con el usuario a través del teclado (para introducir texto) se puede hacer a través de la función `input`: `texto = input('Escribe algo: ')`. La variable que devuelve es un *string* de caracteres.

Para leer o escribir ficheros, hay muchas opciones. Se indican a continuación algunas de entre las más sencillas y habituales.

Función open

La función open devuelve un objeto tipo fichero; ese objeto tiene aparejados varios métodos, uno de los cuales es `readline`:

[1]:
```
fich = open('tablalogs.txt')
leer = True
while leer:
    línea = fich.readline()
    if len(línea)==0:
        leer = False
        print('\n')
    else:
        print(línea, end='')

fich.close()
```

```
1       0.000
2       0.693
3       1.099
4       1.386
```

Obsevaciones.– El objeto `fich` posee muchos métodos, como `write`, `read` o `seek`. Uno de ellos es `readline`, que es el que se usa aquí para leer el contenido línea por línea. **Todo fichero abierto debe cerrarse**. Dejarlo abierto puede comprometer su integridad. Si se produce un error y el programa no llega a la línea `fich.close()`, hay que cerrarlo ejecutando ese comando en la consola.

Se pueden leer todos los contenidos y almacenarlos en una variable de texto. Atención, esta variable contiene *caracteres de escape* (saltos de línea, etc.), que evidentemente se interpretan correctamente en la función `print`.

```
>>> fich = open('tablalogs.txt')
>>> texto_leído = fich.read();
>>> fich.close()
>>> texto_leído
'1\t0.000\n2\t0.693\n3\t1.099\n4\t1.386'
>>> print(texto_leído)
1    0.000
2    0.693
3    1.099
4    1.386
```

Cabe mencionar que la función print se puede redirigir a un fichero:

[2]:
```
import numpy as np
fichero6 = open('ArchivoConPrint.txt','wt')

for número in range(1,10):
    print(número,np.sqrt(número),sep=' ; ',file=fichero6)

fichero6.close()

fichero7 = open('ArchivoConPrint.txt','rt')
leer = True
while leer:
    línea = fichero7.readline()
    if len(línea)==0:
        leer = False
    else:
        print(línea, end='')

fichero7.close()
```

```
1 ; 1.0
2 ; 1.4142135623730951
3 ; 1.7320508075688772
4 ; 2.0
5 ; 2.23606797749979
6 ; 2.449489742783178
7 ; 2.6457513110645907
8 ; 2.8284271247461903
9 ; 3.0
```

Funciones del módulo NumPy

El módulo NumPy contiene dos métodos cuya operación es similar a load y save de Matlab: son `np.loadtxt()` y `np.savetxt()`. Deben operar sobre una lista cuyos elementos sean todos del mismo tipo, pues deben poder almacenarse en un *numpy array*. Esa lista debe estar escrita en formato ASCII.

[1]:
```
import numpy as np
datos = np.loadtxt('tablalogs.txt')

print(datos)
```

```
[[1.    0.   ]
 [2.    0.693]
 [3.    1.099]
 [4.    1.386]]
```

[2]:
```
print(type(datos))
```

```
<class 'numpy.ndarray'>
```

[3]:
```
for fila in range(4):
    for columna in range(2):
        print(datos[fila][columna])
```

```
1.0
0.0
2.0
0.693
3.0
1.099
4.0
1.386
```

[4]:
```
# para guardar datos:
Contenido = np.ones((5,2)) * (-1)

for contador in range(len(Contenido)):
    Contenido[contador][0] = contador
    Contenido[contador][1] = np.sqrt(contador)

np.savetxt('raíces.txt',Contenido)
```

Módulo pickle

En conjunción con la función `open`, el módulo `pickle` contiene dos métodos, a saber, `load` y `dump`, que permiten leer y escribir en un fichero de manera muy versátil.

```
>>> import pickle
>>> lista = ['manzana',12,np.pi]

>>> fichero1 = open('archivo.bi','wb')
>>> # ESCRIBIR
>>> pickle.dump(lista,fichero1)
```

```
>>> fichero1.close()

>>> fichero2 = open('archivo.bi','rb')
>>> # LEER
>>> leído = pickle.load(fichero2)
>>> fichero2.close()

>>> print(leído)
['manzana', 12, 3.141592653589793]
>>> type(leído)
<class 'list'>
```

Observaciones .– La función `open` admite muchas opciones. En particular, `r` y `w` significan respectivamente abrir 'para leer' y 'para escribir'; `b` y `t` significan 'binario' y 'texto'. La ventaja de pickle es que devuelve una lista y no un texto.

Módulo io

El módulo `io` (abreviatura de *input / output*) contiene muchos procedimientos, de los cuales los básicos son `open`, `write` y `read`. Una de las ventajas es que maneja codificación UTF-8.

```
>>> import io

>>> # texto unicode, con todos los caracteres del español
>>> mensaje = 'Liberté \nEgalité \nBeyoncé'

>>> fichero3 = io.open('Archivo.txt','wt',encoding='utf-8')
>>> fichero3.write(mensaje)
25
>>> fichero3.close()

>>> fichero4 = io.open('Archivo.txt','rt',encoding='utf-8')
>>> texto = fichero4.read()
>>> fichero4.close()
>>> print(texto)
Liberté
Egalité
Beyoncé
```

Un problema al abrir un fichero con la opción `w` es que si el fichero existe destruye lo que hay. Para evitar eso, se puede emplear la opción `x`, que abre un fichero para escribir a condición de que el fichero no exista; en caso de que ya exista un fichero con ese nombre, da error:

```
>>> fichero_noAbierto = io.open('MiNombre.txt','xt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'MiNombre.txt'
```

12. Más allá

Entre todos los documentos que pueden servir para profundizar en el aprendizaje de Python, los siguientes constituyen una sugerencia personal:

1. Bibliografía
   - C. H. SWAROOP, "**A byte of Python**", disponible en Internet: python.swaoopch.com
   - R. GONZÁLEZ DUQUE, "**Python para todos**", libro gratuito (disponible en Internet).
   - S. R. DOTY, "**Python basics**" (disponible en Internet en diferentes lugares).
   - C. HILL, "**Learning scientific programming with Python**", Cambridge University Press https://scipython.com/about/the-book/
2. Recursos en Internet
   - S. THURLOW, "**A Beginner's Python Tutorial**", disponible tanto en GitHub github.com/stoive/pythontutorial como en en Wikibooks
   - **Google's Ptython class**: https://developers.google.com/edu/python.
3. Documentación (en estas páginas Web se pueden encontrar manuales y ayuda para el aprendizaje)
   - Página web oficial de Python https://www.python.org/
   - Documentación de la versión: https://docs.python.org/3/
   - Documentación de módulos NumPy, matplotlib, etc. (p. ej. en https://numpy.org/ hay abundante documentación y *tutorials*).

NumPy

1. Objetos y atributos

El módulo NumPy (numpy.og) ofrece una clase, llamada *numpy array*, con la cual las variables se tratan como matrices. Esta clase es similar a una lista pero todos sus elementos deben ser homogéneos (de modo que no se puede mezclar texto con números, por poner un caso). Los métodos definidos para esta clase hacen que se pueda programar en Python de manera muy similar a Matlab (las funciones admiten vectores, la referenciación es similar, etcétera). El cálculo con *numpy arrays* es más rápido que con listas de Python. Además, se incluye una extensa gama de funciones matemáticas (p. ej. generación de números aleatorios, álgebra lineal, etc.), y una excelente documentación –lo cual, lamentablemente, no es lo habitual en Python: numpy.org/doc/stable/. En particular, se recomienda la lecturalos capítulos Quickstart y The absolute basics for beginners

El módulo NumPy se carga así (la abreviatura `np` es casi un convenio)

```
>>> import numpy as np
```

El objeto básico de NumPy es el *array* multidimensional. El eje 0 es el vertical (columnas), el 1 el horizontal (filas), y el 2 el correspondiente a la profundidad. Estos objetos tienen entre otros los siguientes atributos: `ndim` (número de dimensiones), `shape` (foma), `size` (número de elementos), y `dtype` (tipo de los objetos en el *array*).

```
>>> vector = np.array([1, 2, 3, 4, 5])
>>> print(vector)
[1 2 3 4 5]
>>> número = np.array(np.pi)
>>> print(número)
3.141592653589793
>>> matriz = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(matriz)
[[1 2 3]
 [4 5 6]]
>>> tensor = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
>>> print(tensor)
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
>>> # dimensiones de los arrays creados
>>> print(vector.ndim)  # dimensión 1
1
>>> print(número.ndim)  # nótese: dimensión cero
0
>>> print(matriz.ndim)  # dos dimensiones
2
```

```
>>> print(tensor.ndim)  # tres dimensiones
3
```

Aparte de la referencia a los elementos del numpy array con el mismo método que las listas, también admiten referencias entre corchetes con varios números: los primeros son el número del eje (la dimensión), y el último, el número del elemento:

```
>>> print(matriz[0][1]) # esto se puede hacer con una lista
2
>>> print(matriz[0,1])  # pero esto solo con numpy arrays:
2
>>> print(tensor[0, 1, 2]) # 0 y 1 son las dimensiones, 2 el número de elemento
6
>>> # referencias reversas (índices negativos)
>>> mat = np.array([[1,2,3,4,5], [6,7,8,9,10]])
>>> print('penúltimo elemento, segunda dimensión ', mat[1, -2])
penúltimo elemento, segunda dimensión  9
>>> # extracción de una parte del array ("slicing"): como las listas
>>> v = np.array([1, 2, 3, 4, 5, 6, 7])
>>> print(v[1:5])
[2 3 4 5]
>>> print(v[4:])
[5 6 7]
>>> print(v[:4])
[1 2 3 4]
>>> print(v[-3:-1])
[5 6]
>>> print(v[1:5:2])
[2 4]
>>> print(v[::2])
[1 3 5 7]
>>> # pero con matrices también se puede hacer así:
>>> M = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
>>> print(M[1, 1:4])
[7 8 9]
>>> print(M[0:2, 2])
[3 8]
>>> print(M[0:2, 1:4])
[[2 3 4]
 [7 8 9]]
```

2. Métodos: creación y unión de *arrays*

Para crear *numpy arrays* se puede usar `np.array`, `np.zeros`, `np.ones`, y la función *random* del módulo *random*: `np.random.random`. Se pueden crear vectores equiespaciados con np.arange y np.linspace. El primero toma como argumento el principio, el final y el paso, y el segundo el principio, el final y el número de puntos. Se concatenan *arrays* con el método `concatenate` (indicando el eje, o dimensión), o bien con los métodos *stack* `np.vstack`, `np.hstack`, `np.dstack` (respectivamente *v*ertical, *h*orizontal, *d*epth). El método `reshape` permite modificar la forma de un *array*.

```
>>> A = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
>>> B = A.reshape(4, 3) # filas, columnas
>>> print(B)
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
>>> arr1 = np.array([[1, 2], [3, 4]])
>>> arr2 = np.array([[5, 6], [7, 8]])
>>> arr = np.concatenate((arr1, arr2), axis=1) # ¿qué daría axis=0?
>>> print(arr)
[[1 2 5 6]
 [3 4 7 8]]
>>> arr1 = np.array([1, 2, 3])
>>> arr2 = np.array([4, 5, 6])
>>> arr = np.hstack((arr1, arr2))
>>> print(arr)
[1 2 3 4 5 6]
>>> arr = np.vstack((arr1, arr2))
>>> print(arr)
[[1 2 3]
 [4 5 6]]
>>> ristra1 = np.arange(0,0.7,0.2)
>>> print(ristra1)
[0.  0.2 0.4 0.6]
>>> ristra2 = np.linspace(0,0.7,5)
>>> print(ristra2)
[0.    0.175 0.35  0.525 0.7  ]
```

3. Funciones universales

Son métodos que devuelven valores lógicos o índices: `all`, si todos los elementos cumplen la condición; `any`, si alguno la cumple; `nonzero`, cuáles son los elementos distintos de cero; `where`, en qué sitios se cumple la condición. El método `np.where` es un poco enrevesado: la sintaxis es `np.where(condición, array si cierto, array si falso)`, donde los *arrays* pueden ser un número. En cambio, el método `np.argwhere` devuelve los índices de los lugares donde se cumple la condición.

```
>>> ArrayLógico = np.array([True, False, True])
>>> np.all(ArrayLógico)
False
>>> np.any(ArrayLógico)
True
>>> MiArray = np.arange(-2,4,1)
>>> print(MiArray)
[-2 -1  0  1  2  3]
>>> Ind_Lógico = np.where(MiArray>0,True,False)
>>> # Devuelve True donde cierto, False donde falso
>>> print(MiArray[Ind_Lógico])
[1 2 3]
```

```
>>> NuevoArray = np.where(MiArray>0,MiArray, -1)
>>> # Devuelve MiArray donde cierto, -1 donde falso
>>> print(NuevoArray)
[-1 -1 -1  1  2  3]
>>> sitios = np.argwhere(MiArray>0)
>>> print(sitios)
[[3]
 [4]
 [5]]
```

4. Rutinas matemáticas

A partir de aquí, se pueden explorar las funciones que ofrece NumPy, llamadas en rigor Rutinas. Incluyen las trigonométricas (`np.sin`, `np.arctan`), redondeo (`np.floor`, `np.ceil`), álgebra (`np.sum`, `np.cumsum`, `np.diff`), exponenciales y logaritmos (`np.exp`, `np.log`, `np.log10`), estadísticas (`np.mean`, `np.quantile`), etcétera. En estas tablas, las funciones que toman un *array* como argumento se escriben `np.función`, mientras que los métodos (que se ejecutan añadiendo .método al nombre del *array*) se escriben `método()`.

Creación
`np.array([ , , ])`
`np.zeros`
`np.ones`
`np.random`
`np.arange`
`np.linspace`

Atributos
`ndim`
`shape`
`size`
`dtype`
`itemsize`

Métodos
`reshape()`
`resize()`
`sort()`
`round()`
`np.ravel`

Matemáticas
`np.sin`
`np.cos`
`np.tan`
`np.pi`
`np.exp`
`np.log`
`np.sqrt`

Estadística
`max()`
`min()`
`mean()`
`std()`
`np.median`
`np.quantile`
`np.histogram`

Unión
`np.concatenate`
`np.hstack`
`np.vstack`
`np.dstack`

E / S
`np.loadtxt`
`np.savetxt`

Lógica
`np.all`
`np.any`
`np.where`
`np.argwhere`
`np.nonzero`

Además, dentro de NumPy hay módulos específicos para el muestreo aleatorio (`random`), álgebra lineal, polinomios, la transformada de Fourier, y más. Se recomienda el capítulo dedicado a NumPy del libro de C. HILL, "**Learning Scientific Programming with Python**" (Cambridge University Press), y la lista de rutinas de NumPy.

matplotlib

1. El módulo matplotlib. Gráficos sencillos.

Este capítulo se ha elaborado con Jupyter Notebook, una interfaz de Python (la estética de los comandos es un poco diferente). Para instalarlo, ejecutar `conda install -n environment -c conda-forge notebook`.

De nuevo es necesaria la misma advertencia que para el capítulo anterior: **matplotlib** es un módulo con enormes capacidades para la representación gráfica; en este documento sólo se indican algunos de los procedimientos esenciales más sencillos. Contiene varios 'submódulos', por llamarles así. Aquí nos centraremos en `pyplot`, uno de los más usados.

En matplotlib se distinguen dos estilos de programación. Uno de ellos es el '*Object Oriented*', en el cual se comienza definiendo la figura, luego los ejes (a partir de la figura), y luego los gráficos (a partir de los ejes), etcétera. Es un estilo formal y metódico, muy útil para organizar gráficos complejos. Esta metodología es la que se emplea, por ejemplo, en la *Grammar of Graphics* que da lugar a **ggplot**. Aquí emplearemos otro estilo, más parecido al de Matlab, en el cual los comandos de `pyplot` van creando automáticamente ejes, figuras, etcétera.

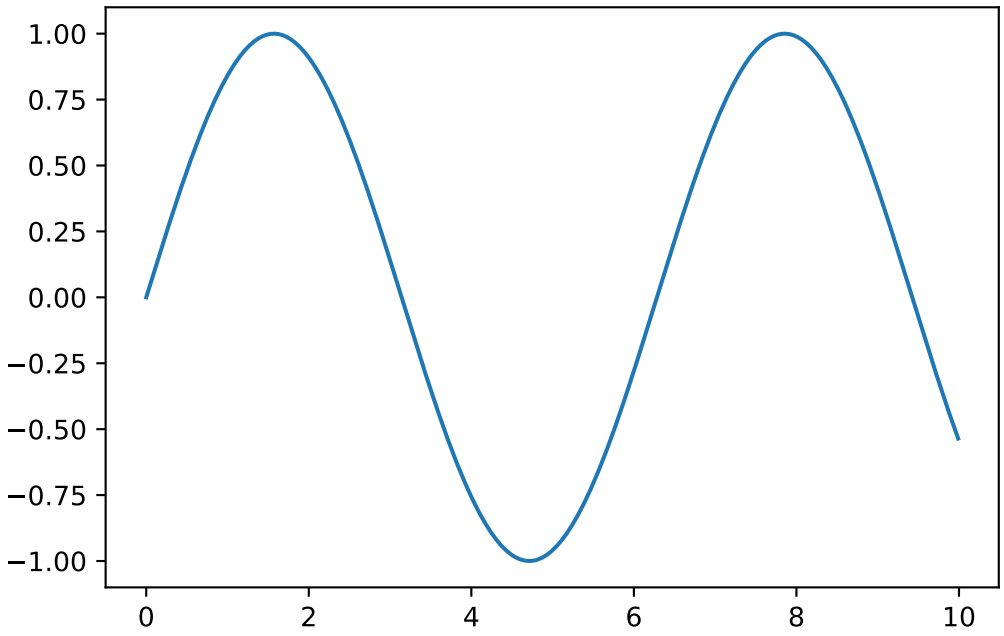
Se importa NumPy y matplotlib.pyplot (los *alias* son casi un convenio):

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Un gráfico sencillo: x frente a y

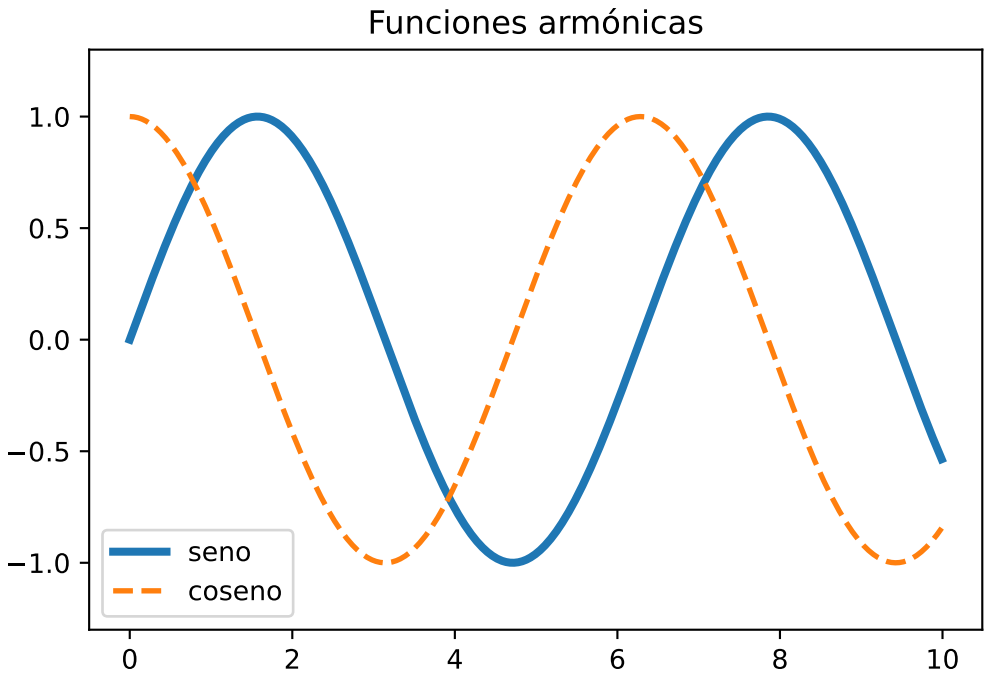
[2]:
```
x = np.arange(0,10,0.01)
y = np.sin(x)
plt.plot(x,y)
```

[2]:
```
[<matplotlib.lines.Line2D at 0x1d66be29250>]
```



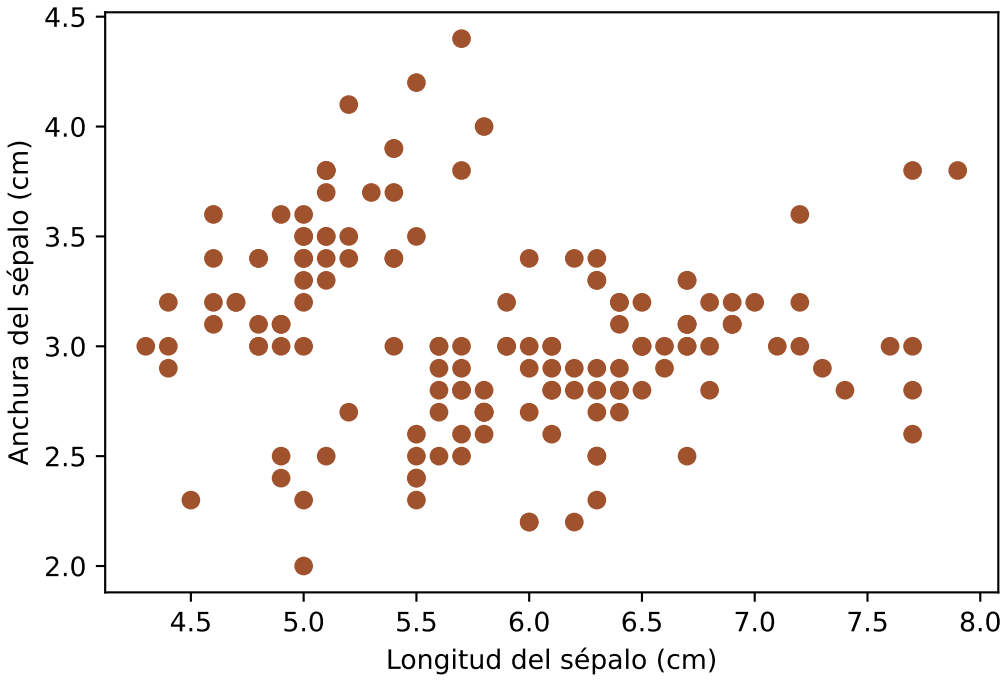
A partir de aquí, se puede modificar la apariencia y los elementos del gráfico. El ; al final de la línea evita la descripción del objeto de salida (`matplotlib.lines.` etc.)

[3]:
```
x = np.arange(0,10,0.01)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x,y1, linewidth = 3, label = 'seno');
plt.plot(x, y2, linewidth = 2, linestyle = 'dashed', label='coseno');
plt.yticks(np.linspace(-1,1,5));
plt.ylim(-1.3,1.3);
plt.title('Funciones armónicas');
plt.legend();
```



Los datos se representan con símbolos, a los cuales se les puede cambiar las propiedades fácilmente.

[4]:
```
iris = np.genfromtxt('iris.csv',␣
 ↪delimiter=',',usecols=(0,1,2,3),skip_header=1)
plt.plot(iris[:,0],iris[:,1],'o', color = 'sienna');
plt.xlabel('Longitud del sépalo (cm)');
plt.ylabel('Anchura del sépalo (cm)');
```



2. Resumen de los comandos más usuales de matplotlib.pyplot

Todos los comandos van precedidos de `plt`.

- `plot(x, y)` – gráfico de una línea con segmentos que unen los valores x e y.
- `show()` – muestra el gráfico
- `title(‘‘string’’)` – título del gráfico
- `xlabel(‘‘string’’)` – título del eje x
- `ylabel(‘‘string’’)` – título del eje y
- `figure()` – crea una figura y especifica sus atributos
- `subplot(nrows, ncols, index)` – divide una figua en varios gráficos
- `subplots(nrows, ncols, figsize)` – alternativa para crear subplots: devuelve una tupla de una figura y cierto número de ejes.
- `bar(categorical variables, values, color)` – gráfico de barras
- `barh(categorical variables, values, color)` – gráfico de barras horizontal
- `legend(loc)` – crear la leyenda de un gráfico
- `xticks(index, categorical variables)` – lugares de las marcas del eje x (id. para el eje y).
- `pie(value, categorical variables)` – crea una gráfico de sectores
- `hist(values, number of bins)` – crea un histograma
- `boxplot(data)` – crea un diagrama de cajas
- `ylim(start value, end value)` – especifica los límites del eje y (id. para el eje x).
- `scatter(x-axis values, y-axis values)` – crea un gráfico de dispersión (‘scatter plot’)
- `axes()` – añade unos ejes a la figura
- `scatter3D(x-axis values, y-axis values, z-axis values)` – gráfico de dispersión tridimensional
- `plot3D(x-axis values, y-axis values, z-axis values)` – gráfico tridimensional
- `text(x, y, ‘‘string’’)` – escribe un texto en el lugar especificado.
- `annotate(‘‘string’’, ...)` – escribe un texto y dibuja una flecha en el lugar especificado.

A continuación se proporciona la lista de los colores predefinidos en matplotlib. Se puede definir cualquier color que se desee, en diferentes formatos (RGB, CMYK, HSV, etc.). Por otro lado, ya hay definidas varias paletas de colores.

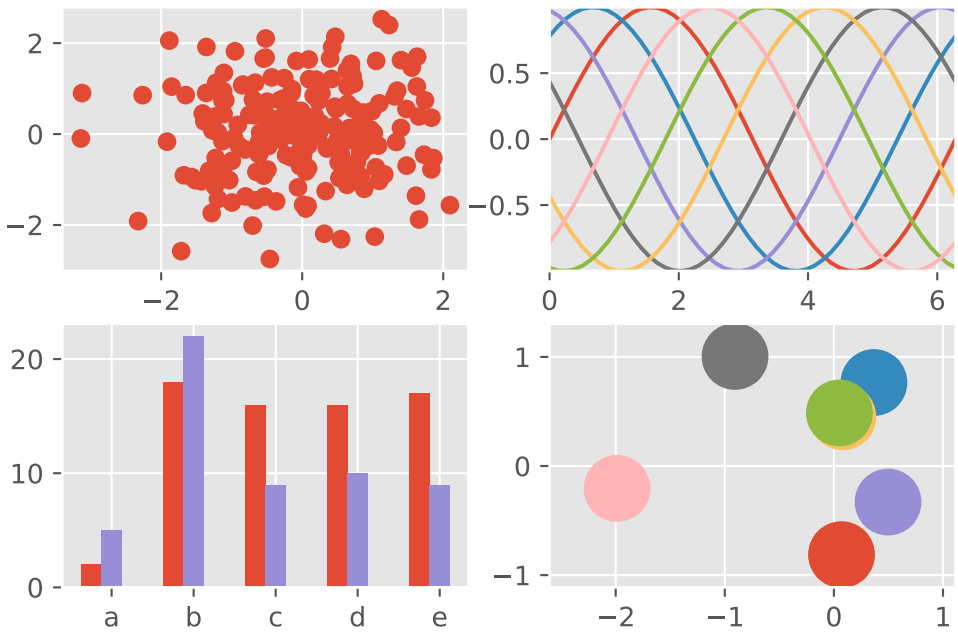
3. Documentación y ejemplos

La documentación de matplotlib es excelente y contiene, además de una completa referencia, diversas guías introductorias que son muy recomendables. En la página Web de matplotlib matplotlib.org, en el menú *Tutorials* se ofrecen varios a distintos niveles. Se sugiere además leer el apartado *Plot types* de esa misma página. Las "*Cheat Sheets*" (ver anexo) constituyen un resumen muy práctico. Otra buena lectura es el capítulo correspondiente a matplotlib en la obra C. HILL, "**Learning Scientific Programming with Python**" (Cambridge University Press).

Por otro lado, en el apartado *Examples* se proporciona código para crear gráficos de muy distintos tipos. Basta copiar y pegar. El siguiente código se obtuvo de dicha Galería. Se incluye aquí también porque está escrito en el estilo *Object Oriented*. Se inicia creando los ejes con el procedimiento `plt.subplot`, y luego se van añadiendo los gráficos como *hijos* ("*children*") de los ejes (por ejemplo, `ax1.plot`, `ax3.bar`, etc).

[5]:
```
import numpy as np
import matplotlib.pyplot as plt
```

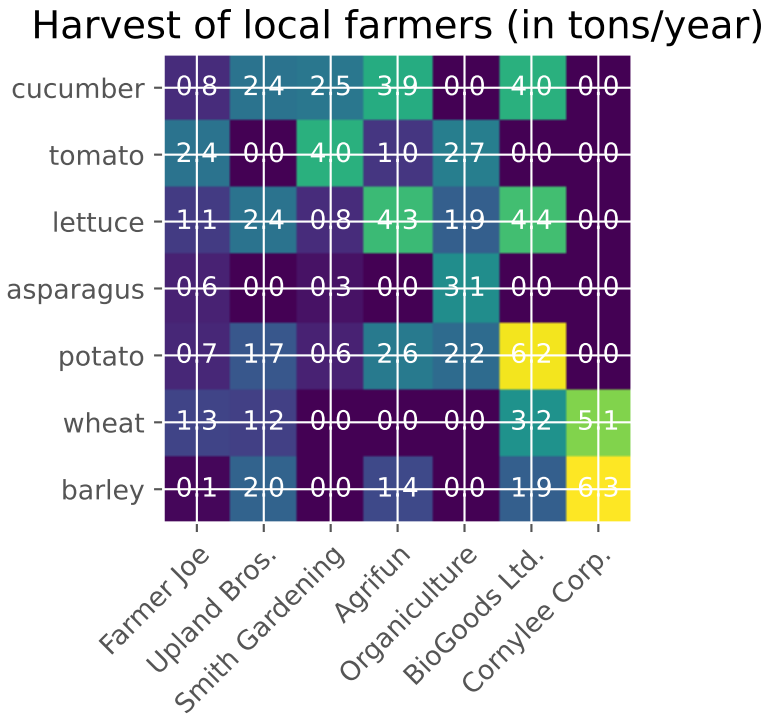
```
plt.style.use('ggplot')

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, axs = plt.subplots(ncols=2, nrows=2)
ax1, ax2, ax3, ax4 = axs.flat

# scatter plot (Note: `plt.scatter` doesn't use default colors)
x, y = np.random.normal(size=(2, 200))
ax1.plot(x, y, 'o')

# sinusoidal lines with colors from default color cycle
L = 2*np.pi
x = np.linspace(0, L)
ncolors = len(plt.rcParams['axes.prop_cycle'])
shift = np.linspace(0, L, ncolors, endpoint=False)
for s in shift:
    ax2.plot(x, np.sin(x + s), '-')
ax2.margins(0)

# bar graphs
x = np.arange(5)
y1, y2 = np.random.randint(1, 25, size=(2, 5))
width = 0.25
ax3.bar(x, y1, width)
ax3.bar(x + width, y2, width,
        color=list(plt.rcParams['axes.prop_cycle'])[2]['color'])
ax3.set_xticks(x + width, labels=['a', 'b', 'c', 'd', 'e'])

# circles with colors from default color cycle
for i, color in enumerate(plt.rcParams['axes.prop_cycle']):
    xy = np.random.normal(size=2)
    ax4.add_patch(plt.Circle(xy, radius=0.3, color=color['color']))
ax4.axis('equal')
ax4.margins(0)

plt.show()
```



Otro ejemplo; bajo el epígrafe *heatmap* se encuentra el siguiente código:

[6]:
```
vegetables = ["cucumber", "tomato", "lettuce", "asparagus",
              "potato", "wheat", "barley"]
farmers = ["Farmer Joe", "Upland Bros.", "Smith Gardening",
           "Agrifun", "Organiculture", "BioGoods Ltd.", "Cornylee Corp."]

harvest = np.array([[0.8, 2.4, 2.5, 3.9, 0.0, 4.0, 0.0],
                    [2.4, 0.0, 4.0, 1.0, 2.7, 0.0, 0.0],
                    [1.1, 2.4, 0.8, 4.3, 1.9, 4.4, 0.0],
                    [0.6, 0.0, 0.3, 0.0, 3.1, 0.0, 0.0],
                    [0.7, 1.7, 0.6, 2.6, 2.2, 6.2, 0.0],
                    [1.3, 1.2, 0.0, 0.0, 0.0, 3.2, 5.1],
                    [0.1, 2.0, 0.0, 1.4, 0.0, 1.9, 6.3]])


fig, ax = plt.subplots()
im = ax.imshow(harvest)

# Show all ticks and label them with the respective list entries
ax.set_xticks(np.arange(len(farmers)), labels=farmers)
ax.set_yticks(np.arange(len(vegetables)), labels=vegetables)

# Rotate the tick labels and set their alignment.
plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
         rotation_mode="anchor")

# Loop over data dimensions and create text annotations.
for i in range(len(vegetables)):
    for j in range(len(farmers)):
```

```
        text = ax.text(j, i, harvest[i, j],
                       ha="center", va="center", color="w")

ax.set_title("Harvest of local farmers (in tons/year)")
fig.tight_layout()
plt.show()
```



Los códigos de esa página pueden ser un buen punto de partida para crear un gráfico de calidad.

Tópicos adicionales

1. Tuplas: *packing & unpacking*

Supongamos que una función devuelva varios valores. Lo habitual es hacerlo así:

[1]:
```
def minmax_list(lista):
    min_por_ahora = lista[0]
    max_por_ahora = lista[0]
    for a in lista:
        if a > max_por_ahora:
            max_por_ahora = a
        if a < min_por_ahora:
            min_por_ahora = a

    return min_por_ahora, max_por_ahora
```

Nótese que lo que devuelve es una **tupla**, porque hay una coma entre los dos argumentos de salida: `a,b` es una tupla (aunque falten los paréntesis).

[2]:
```
li = [1,2,3,4,5]
recibido = minmax_list(li)
print(recibido)
type(recibido)
```

```
(1, 5)
```

[2]:
```
tuple
```

Si se desean recibir *separadamente* el máximo y el mínimo, hay que **desempaquetar** la tupla, lo cual se consigue recibiendo los argumentos de salida en otra tupla:

[3]:
```
mínimo, máximo = minmax_list(li)
print(mínimo)
print(máximo)
type (mínimo)
```

```
1
5
```

[3]:
```
int
```

Otra manera de desempaquetar las tuplas es con el operador $*$. Por ejemplo: la función `np.hypot(a,b)` calcula la hipotenusa de un triángulo de lados a y b, es decir, $\sqrt{a^2 + b^2}$:

[4]:
```
import numpy as np
print(np.hypot(3,4)) # raíz cuadrada de 9 + 16
```

```
5.0
```

Ahora bien, como `np.hypot` necesita dos argumentos ...

[5]:
```
lados = (3,4) # esto es una tupla
h = np.hypot(lados)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_10892/3801916387.py in <module>
      1 lados = (3,4) # esto es una tupla
----> 2 h = np.hypot(lados)

TypeError: hypot() takes from 2 to 3 positional arguments but 1 were given
```

Hay que desempaquetar la tupla:

[6]:
```
h = np.hypot(*lados) # el asterisco desempaqueta la tupla
print(h)
print(*lados) # si no hay coma, no hay tupla ...
```

```
5.0
3 4
```

Consideremos otro caso. Esta función halla la solución de un polinomio de segundo grado de coeficientes a, b, c

[7]:
```
def raíces(a,b,c):
    """calcula las raíces de una ecuación de segundo grado"""
    discriminante = np.sqrt(b**2 - 4*a*c)
    raíz1 = (-b + discriminante)/(2*a)
    raíz2 = (-b - discriminante)/(2*a)
    return raíz1, raíz2 # atención: esto es una tupla
```

La primera línea es la ayuda de la función:

[8]:
```
print(raíces.__doc__)
```

```
calcula las raíces de una ecuación de segundo grado
```

Ejecutemos la función para hallar las raíces de $y = x^2 - 5x + 6$

[9]:
```
soluciones = raíces(1.0,-5.0,6.0)
print(soluciones)
print(*soluciones)
```

```
(3.0, 2.0)
3.0 2.0
```

Otra manera de desempaquetar la tupla:

[10]:
```
sol1,sol2 = soluciones
print(sol1)
print(sol2)
```

```
3.0
2.0
```

También se puede desempaquetar directamente:

[11]:
```
r1,r2 = raíces(a=1.0,b=-5.0,c=6.0)
print(r1)
print(r2)
```

```
3.0
2.0
```

Si una de las variables no se va a usar, se suele recoger en un objeto al que por convenio se le da el nombre _:

[12]:
```
dimensiones = (1080,1920)
_ , ancho = dimensiones
print(ancho)
```

```
1920
```

Empaquetar y desempaquetar tuplas puede usarse con diferentes fines, como definir múltiples valores de manera compacta

[13]:
```
(x1,x2,x3) = (10,20,30)
print(x2)
```

```
20
```

o intercambiar valores:

[14]:
```
print("x1 vale {}, y x2 vale {}".format(x1,x2))
(x1,x2) = (x2,x1) # primero se evalúa lo que hay a la derecha del igual
print("ahora, x1 vale {}, y x2 vale {}".format(x1,x2))
```

```
x1 vale 10, y x2 vale 20
ahora, x1 vale 20, y x2 vale 10
```

2. *Formateo del texto*

La salida de datos o mensajes en forma texto, sea por pantalla o redirigida hacia un fichero, se puede modificar adecuadamente para aumentar su legibilidad. El concepto básico es que ese formateo se lleva a cabo con un método de las cadenas de caracteres llamado `format`. Ese método permite la inclusión de variables en la cadena de caracteres, indicada mediante llaves. Así,

[1]:
```
'{} {}'.format('uno', 'dos')
```

[1]:
```
'uno dos'
```

A partir de aquí, se pueden introducir muchas opciones. Se ilustran las más usuales.

Orden

Los números dentro de las llaves indican el orden de la secuencia a sustituir:

[4]:
```
'{1}, {0}, {2}'.format('uno','dos','tres')
```

[4]:
```
'dos, uno, tres'
```

Formateo de números

La abreviatura para un entero es `:d` y para un *float* es `:f`

[6]:
```
import numpy as np
print('{:d}'.format(25))
print('{:f}'.format(np.pi))
```

```
25
3.141593
```

Detrás de : se pueden especificar diversas opciones.

[7]:
```
print('{:4d}'.format(25)) # mostrar en un campo de cuatro espacios
```

```
  25
```

[8]:
```
print('{:04d}'.format(25)) # id., rellenando con ceros
```

```
0025
```

[10]:
```
print('{:+d}'.format(25)) # mostrar signo
```

```
+25
```

[11]:
```
print('{:06.2f}'.format(np.pi)) # en un campo de seis espacios, dos␣
↪decimales, rellenar con ceros
```

```
003.14
```

Alternativamente, se puede emplear esta sintaxis:

[16]:
```
print('{:{ancho}.{prec}f}'.format(np.pi,ancho=6,prec=2))
```

```
  3.14
```

Datos de una lista

El método .format puede hacer referencia a variables definidas:

[10]:
```
x=22
print('{}'.format(x))
```

```
22
```

O también a elementos de una lista:

[12]:
```
lista = [ 7, 14, 21, 28 ]
print('{d[1]} {d[3]}'.format(d=lista))
```

```
14 28
```

Hay muchas otras posibilidades. Consultar la documentación de `format` o la página pyformat.info/

3. *List comprehension*

Python ofrece una sintaxis muy compacta y elegante para crear listas. En muchos otros lenguajes de programación, crear una colección implica muchas veces ejecutar un bucle. Por poner un ejemplo, piénsese cómo se crearía una lista con los cuadrados de los enteros del 1 al 5.

[1]:
```
cuad = []
for num in range(1,6,1):
    cuad.append(num**2)
print(cuad)
```

```
[1, 4, 9, 16, 25]
```

La alternativa es crear una lista con la sintaxis siguiente:

lista = [expresión for ítem in iterable if condición == True]

Este comando actúa de manera que se recorre el iterable, y para cada elemento *item* que cumpla la condición se calcula la expresión y se añade ese resultado a la lista. La condición es opcional y se puede omitir.

Un *iterable* es un conjunto cuyos elementos se pueden recorrer de uno en uno, como una lista, una tupla, o una cadena de caracteres. Se puede usar `range` para crear un iterable.

[2]:
```
cuadrados = [x**2 for x in range(1,6,1)]
print(cuadrados)
```

```
[1, 4, 9, 16, 25]
```

Ya que x es una variable muda, bien se podría escribir

[3]:
```
c = [_**2 for _ in range(1,6,1)]
print(c)
```

```
[1, 4, 9, 16, 25]
```

para indicar que _ no es sino una variable auxiliar.

En el caso siguiente, se ejecuta una *list comprehension* con una condición:

[4]:
```
frutas = ['plátano', 'manzana', 'limón', 'melocotón']
con_a = [x for x in frutas if "a" in x]
print(con_a)
```

```
['plátano', 'manzana']
```

Las expresiones también pueden tener condiciones:

[5]:
```
Limón_no = [x if x != 'limón' else 'naranja' for x in frutas]
print(Limón_no)
```

```
['plátano', 'manzana', 'naranja', 'melocotón']
```

lo cual se leería así: para cada ítem de `frutas`, añade el ítem a la lista `Limón_no` si el ítem no es `limón`, en otro caso (es decir, si sí que es `limón`), pon `naranja`.

El siguiente comando haría lo siguiente: del 1 al 10, guarda el número junto con "Par" si es divisible por 2, y el número con "Impar" en caso contrario.

[6]:
```
ParImpar = [[n,'Par'] if n%2 == 0 else [n,'Impar'] for n in range(1,10)]
print(ParImpar)
```

```
[[1, 'Impar'], [2, 'Par'], [3, 'Impar'], [4, 'Par'], [5, 'Impar'], [6, 'Par'],
[7, 'Impar'], [8, 'Par'], [9, 'Impar']]
```

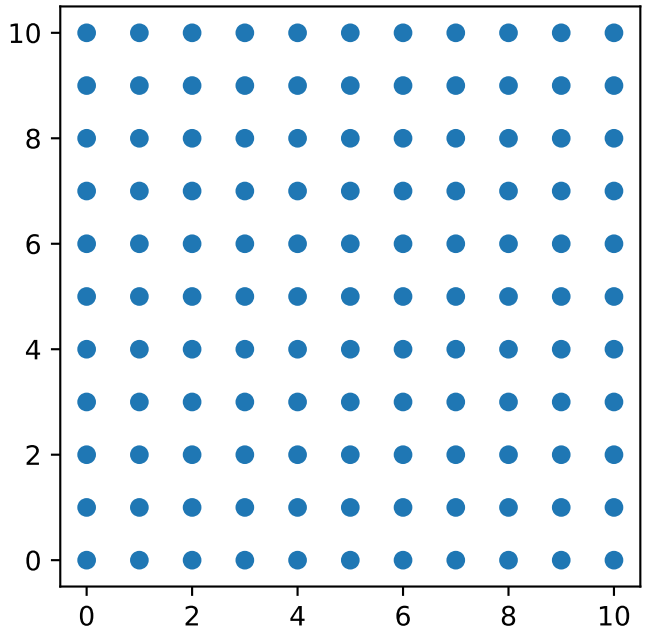
La *list comprehension* funciona también en colecciones que requieren listas, como un diccionario:

[7]:
```
dic_cuads = {x: x**2 for x in range(5) }
print(dic_cuads)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

He aquí cómo se genera un ''*mesh*'', o mallado (una cuadrícula o rejilla, es decir, todos los (x_i, y_i) a intervalos regulares; en este caso, en una zona del plano)

[9]:
```
import numpy as np
import matplotlib.pyplot as plt
pts = [[x,y] for x in range(11) for y in range(11)]
puntos = np.array(pts)
plt.plot(puntos[:,0],puntos[:,1],'o')
plt.gca().set_aspect('equal', adjustable='box')
plt.draw()
```



[10]:
```
print(puntos)
```

```
[[ 0  0]
 [ 0  1]
 [ 0  2]
 [ 0  3]
 [ 0  4]
 [ 0  5]
```

```
[ 0  6]
[ 0  7]
[ 0  8]
[ 0  9]
[ 0 10]
[ 1  0]
[ 1  1]
 ...
[10  8]
[10  9]
[10 10]]
```

Ahora, en cada uno de esos puntos se podría hallar una función $z = f(x, y)$ para después dibujarla en tres dimensiones, por ejemplo.

Finalmente, adviértase que se puede utilizar una *list comprehension* anidada dentro de otra:

[11]:
```
M = [[j for j in range(3)] for i in range(2)]
print(M)
```

```
[[0, 1, 2], [0, 1, 2]]
```

4. El módulo `random`

El paquete `random` está incluido en la instalación básica de Python y proporciona una manera rápida y sencilla de obtener números aleatorios. Se ilustran aquí algunos de los usos más habituales.

[1]:
```
import random # se importa sin asignar ningún alias
num_azar = random.random() # genera un número aleatorio
print(num_azar)
num_azar = random.random() # entre cero y uno
print(num_azar)
```

```
0.9330093204886947
0.6935571558823811
```

Obsérvese cómo cada vez que se usa el comando cambia el número al azar elegido. En realidad, son números pseudoaleatorios (dicho rápidamente, pasan los *tests* de aleatoriedad, pero se obtienen mediante un algoritmo). En esta sección se denominan aleatorios o pseudoaleatorios de manera equivalente.

Para obtener varios números aleatorios, se puede usar una sintaxis como esta:

[2]:
```
aleat4 = [random.random() for _ in range(4)] # cuatro números al azar
print(aleat4)
```

```
[0.4771603359402251, 0.36445157309341636, 0.9859389717887498, 0.465241588165076]
```

Si se desea un número entero al azar, en un rango determinado, se emplea el método `randrange`:

[3]:
```
entero_azar = random.randrange(1,10)
print(entero_azar)
```

```
1
```

Barajar equivale a obtener una permutación al azar. Téngase en cuenta que este método cambia la lista original.

[4]:
```
nums = [1,2,3,4,5]
random.shuffle(nums) # ojo, que esto cambia nums
print(nums)
```

```
[4, 2, 1, 3, 5]
```

Se puede obtener un elemento al azar de una lista:

[5]:
```
amigos = ["Javier", "Marina", "Iker"]
MAPS = random.choice(amigos)
print(MAPS)
```

```
Iker
```

Para escoger una muestra al azar de una colección: método `sample`

[6]:
```
nums_lotería = range(63)
nums_ganadores = random.sample(nums_lotería, 6)
print(nums_ganadores)
```

```
[41, 28, 56, 29, 4, 6]
```

Los números pseudoaleatorios se calculan a partir de una "semilla" (*seed*). El generador de números aleatorios va cambiando esta semilla automáticamente de una vez a otra. Pero puede ser que en alguna ocasión interese obtener exactamente los mismos números 'aleatorios', es decir, que se genere un número al azar pero que siempre sea el mismo. Eso es útil si, por ejemplo, varias personas diferentes deben comprobar un cálculo, o si se desea obtener siempre la misma figura, etc. Para eso, se puede especificar la semilla. Ahora bien, luego hay que tener la precaución de restablecer estado del generador de números aleatorios para que siga como estaba antes, de manera que siga sacando números diferentes cada vez. Eso se hace así:

[7]:
```
estado = random.getstate() # se guarda el estado para restablecerlo después
random.seed(6) # semilla
numaleat = random.random()
print(numaleat)
random.seed(6) # semilla
numaleat = random.random()
print(numaleat)
```

```
0.793340083761663
0.793340083761663
```

Es un número pseudoaleatorio obtenido a partir de la semilla indicada. Como la semilla es la misma, el resultado es el mismo.

Una vez completado el procedimiento, hay que dejar el generador de números aleatorios como estaba antes:

[8]:
```
random.setstate(estado)
numaleat = random.random()
print(numaleat)
numaleat = random.random()
print(numaleat)
```

```
0.503344689392641
0.7553120887991164
```

5. Métodos con funciones *lambda*

Estos métodos se aplican a listas y permiten generar nuevas listas, con la ayuda de una función (normalmente se emplea una función lambda).

map

La sintaxis es: `list(map(función,lista de entrada))`. Se llama a la función con todos los ítems de la lista de entrada, y se devuelve una nueva lista que contiene los elementos que produce la función con cada uno de los ítems de la lista original.

[3]:
```
MiLista = [1, 6, 5, 3.14, 12, 17, 4]
Dobles = list(map(lambda x: x*2, MiLista))
print(Dobles)
```

```
[2, 12, 10, 6.28, 24, 34, 8]
```

filter

La sintaxis es `list(filter( función, lista de entrada))`. Se llama a la función con todos los ítems de la lista de entrada, y devuelve una nueva lista que contiene los ítems para los cuales la función se evalúa como `True`. En este contexto, "filtrar" se toma como sinónimo de retirar lo inválido.

[5]:
```
Pares = list(filter(lambda x: x%2==0, MiLista))
print(Pares)
```

```
[6, 12, 4]
```

6. El módulo SymPy

El **cálculo simbólico** es una posibilidad nada despreciable en los entornos de programación científicos, como Matlab o Python. Consiste en resolver problemas algebraicamente, no numéricamente; es decir, con símbolos, como x, y, etcétera. Para ello, el ordenador debe contar con algunas funciones que permitan definir tales símbolos y las reglas algebraicas con las que debe operar; por ejemplo, $x^2 + x^2 = 2x^2$.

En Python, el módulo `sympy` permite el cálculo simbólico. Se ilustra aquí con algunos ejemplos.

Para comenzar, se importa el módulo y se definen algunos símbolos; x, y, z, t son símbolos (variables), y las variables i, j, k representan números enteros.

[1]:
```
from sympy import *
x,y,z,t = symbols('x y z t')
i,j,k = symbols('i j k', integer = True)
```

En el módulo `sympy` están contenidas las reglas del álgebra:

[2]:
```
expr = (x+y)**5
expand(expr)
```

[2]: $x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5$

[3]:
```
expand(expr - x*y**4)
```

[3]: $x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 4xy^4 + y^5$

así como las derivadas

[4]:
```
diff(sin(x)*exp(x),x)
```

[4]: $e^x \sin(x) + e^x \cos(x)$

y las integrales: esta integral $\int \text{sen}(x)\ e^x\ dx$ se calcula así

[5]:
```
integrate(sin(x)*exp(x),x)
```

[5]: $\frac{e^x \sin(x)}{2} - \frac{e^x \cos(x)}{2}$

Se pueden calcular integrales definidas, e incluso integrales impropias: veamos cuánto vale $\int_{-\infty}^{\infty} \text{sen}(x)\ e^x\ dx$ (el infinito se representa con dos ceros seguidos, así: `00`)

[6]:
```
integrate(sin(x**2),(x,-oo, oo))
```

[6]: $\frac{\sqrt{2}\sqrt{\pi}}{2}$

El módulo también "sabe" hacer límites ...

[7]:
```
limit(sin(x)/x,x,0)
```

[7]: 1

... y resolver ecuaciones de segundo grado

[8]:
```
solve(x**2 - 5 *x +6,x)
```

[8]:
```
[2, 3]
```

Para resolver ecuaciones diferenciales, hay que definir algunos símbolos como funciones; a continuación resolvamos $\frac{df(t)}{dt} = -a\, f(t)$

[9]:
```
f = symbols('f',cls=Function)
a = symbols('a')
dsolve(Eq(diff(f(t),t),-a*f(t)),f(t))
```

[9]: $f(t) = C_1 e^{-at}$

La derivada de f es un método definido para el objeto f (de clase `'Function'`). La derivada segunda se escribe `f(t).diff(t,t)`. Resolvamos ahora $\frac{d^2 f(t)}{dt^2} = a$ (aceleración a constante):

[10]:
```
dsolve(Eq(f(t).diff(t,t),a),f(t))
```

[10]: $f(t) = C_1 + C_2 t + \frac{at^2}{2}$

(movimiento uniformemente acelerado). Otros ejemplos:

[11]:
```
import numpy as np
integrate(cos(x)**2,(x,0,np.pi))
```

[11]: 1,5707963267949

[12]:
```
diff(x**3)
```

[12]: $3x^2$

[13]:
```
integrate(x**2,x)
```

[13]: $\frac{x^3}{3}$

[14]:
```
integrate(log(x),x)
```

[14]: $x \log(x) - x$

[15]:
```
integrate(sin(x)*cos(x),x)
```

[15]: $\frac{\sin^2(x)}{2}$

[16]:
```
expresión = sin(x)
expresión.series(x,0,8)
```

[16]: $x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + O\left(x^8\right)$

Como se ve, esto puede ahorrar bastante tiempo – si uno sabe lo que está haciendo.

El módulo es demasiado extenso como para explorarlo con detalle. Se recomienda a la persona interesada consultar la documentación.

Anexo

Bibliografía adicional

Aparte de los libros que se han ido citando a lo largo de este documento, se recomiendan también los siguientes:

- Scipy Lecture Notes, un repositorio de documentación donde se incluyen valiosas explicaciones acerca de NumPy, matplotlib, scipy y otros módulos de interés científico.
- C. ROSSANT, **"IPython Interactive Computing and Visualization Cookbook"**, Packt Publishing. El PDF se ofrece gratuitamente en Internet.

Cheat sheets

Por comodidad, a continuación se incluyen algunas hojas de resumen ("cheat sheets") de dominio público (suele indicarse en las mismas su origen; muchas han sido obtenidas de DataCamp. Son buenos resúmenes para tener a mano.

Python For Data Science *Cheat Sheet*

Python Basics

Learn More Python for Data Science Interactively at www.datacamp.com

Variables and Data Types

Variable Assignment

```
>>> x=5
>>> x
 5
```

Calculations With Variables

`>>> x+2` → `7`	Sum of two variables
`>>> x-2` → `3`	Subtraction of two variables
`>>> x*2` → `10`	Multiplication of two variables
`>>> x**2` → `25`	Exponentiation of a variable
`>>> x%2` → `1`	Remainder of a variable
`>>> x/float(2)` → `2.5`	Division of a variable

Types and Type Conversion

`str()`	`'5', '3.45', 'True'`	Variables to strings
`int()`	`5, 3, 1`	Variables to integers
`float()`	`5.0, 1.0`	Variables to floats
`bool()`	`True, True, True`	Variables to booleans

Asking For Help

```
>>> help(str)
```

Lists

Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements

Index starts at 0

Subset	
`>>> my_list[1]`	Select item at index 1
`>>> my_list[-3]`	Select 3rd last item
Slice	
`>>> my_list[1:3]`	Select items at index 1 and 2
`>>> my_list[1:]`	Select items after index 0
`>>> my_list[:3]`	Select items before index 3
`>>> my_list[:]`	Copy my_list
Subset Lists of Lists	
`>>> my_list2[1][0]`	my_list[list][itemOfList]
`>>> my_list2[1][:2]`	

List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

List Methods

`>>> my_list.index(a)`	Get the index of an item
`>>> my_list.count(a)`	Count an item
`>>> my_list.append('!')`	Append an item at a time
`>>> my_list.remove('!')`	Remove an item
`>>> del(my_list[0:1])`	Remove an item
`>>> my_list.reverse()`	Reverse the list
`>>> my_list.extend('!')`	Append an item
`>>> my_list.pop(-1)`	Remove an item
`>>> my_list.insert(0,'!')`	Insert an item
`>>> my_list.sort()`	Sort the list

Libraries

Import libraries

```
>>> import numpy
>>> import numpy as np
```

Selective import

```
>>> from math import pi
```

pandas — Data analysis

learn — Machine learning

NumPy — Scientific computing

matplotlib — 2D plotting

Install Python

ANACONDA — Leading open data science platform powered by Python

spyder — Free IDE that is included with Anaconda

jupyter — Create and share documents with live code, visualizations, text, ...

Numpy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3],[4,5,6]])
```

Selecting Numpy Array Elements

Index starts at 0

Subset	
`>>> my_array[1]` → `2`	Select item at index 1
Slice	
`>>> my_array[0:2]` → `array([1, 2])`	Select items at index 0 and 1
Subset 2D Numpy arrays	
`>>> my_2darray[:,0]` → `array([1, 4])`	my_2darray[rows, columns]

Numpy Array Operations

```
>>> my_array > 3
array([False, False, False,  True], dtype=bool)
>>> my_array * 2
array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
array([6, 8, 10, 12])
```

Numpy Array Functions

`>>> my_array.shape`	Get the dimensions of the array
`>>> np.append(other_array)`	Append items to an array
`>>> np.insert(my_array, 1, 5)`	Insert items in an array
`>>> np.delete(my_array,[1])`	Delete items in an array
`>>> np.mean(my_array)`	Mean of the array
`>>> np.median(my_array)`	Median of the array
`>>> my_array.corrcoef()`	Correlation coefficient
`>>> np.std(my_array)`	Standard deviation

Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2
'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
'thisStringIsAwesomeInnit'
>>> 'm' in my_string
True
```

String Operations

Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```

String Methods

`>>> my_string.upper()`	String to uppercase
`>>> my_string.lower()`	String to lowercase
`>>> my_string.count('w')`	Count String elements
`>>> my_string.replace('e', 'i')`	Replace String elements
`>>> my_string.strip()`	Strip whitespaces

Python For Data Science *Cheat Sheet*

NumPy Basics

Learn Python for Data Science **Interactively** at www.DataCamp.com

NumPy

The **NumPy** library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

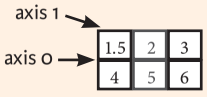
Use the following import convention:

```
>>> import numpy as np
```

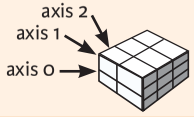
NumPy Arrays

1D array

2D array



3D array



Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

```
>>> np.zeros((3,4))                    Create an array of zeros
>>> np.ones((2,3,4),dtype=np.int16)    Create an array of ones
>>> d = np.arange(10,25,5)             Create an array of evenly
                                       spaced values (step value)
>>> np.linspace(0,2,9)                 Create an array of evenly
                                       spaced values (number of samples)
>>> e = np.full((2,2),7)               Create a constant array
>>> f = np.eye(2)                      Create a 2X2 identity matrix
>>> np.random.random((2,2))            Create an array with random values
>>> np.empty((3,2))                    Create an empty array
```

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

```
>>> np.int64        Signed 64-bit integer types
>>> np.float32      Standard double-precision floating point
>>> np.complex      Complex numbers represented by 128 floats
>>> np.bool         Boolean type storing TRUE and FALSE values
>>> np.object       Python object type
>>> np.string_      Fixed-length string type
>>> np.unicode_     Fixed-length unicode type
```

Inspecting Your Array

```
>>> a.shape          Array dimensions
>>> len(a)           Length of array
>>> b.ndim           Number of array dimensions
>>> e.size           Number of array elements
>>> b.dtype          Data type of array elements
>>> b.dtype.name     Name of data type
>>> b.astype(int)    Convert an array to a different type
```

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

```
>>> g = a - b                          Subtraction
  array([[-0.5,  0. ,  0. ],
         [-3. , -3. , -3. ]])
>>> np.subtract(a,b)                   Subtraction
>>> b + a                              Addition
  array([[ 2.5,  4. ,  6. ],
         [ 5. ,  7. ,  9. ]])
>>> np.add(b,a)                        Addition
>>> a / b                              Division
  array([[ 0.66666667,  1.        ,  1.        ],
         [ 0.25      ,  0.4       ,  0.5       ]])
>>> np.divide(a,b)                     Division
>>> a * b                              Multiplication
  array([[  1.5,   4. ,   9. ],
         [  4. ,  10. ,  18. ]])
>>> np.multiply(a,b)                   Multiplication
>>> np.exp(b)                          Exponentiation
>>> np.sqrt(b)                         Square root
>>> np.sin(a)                          Print sines of an array
>>> np.cos(b)                          Element-wise cosine
>>> np.log(a)                          Element-wise natural logarithm
>>> e.dot(f)                           Dot product
  array([[ 7.,  7.],
         [ 7.,  7.]])
```

Comparison

```
>>> a == b                             Element-wise comparison
  array([[False,  True,  True],
         [False, False, False]], dtype=bool)
>>> a < 2                              Element-wise comparison
  array([True, False, False], dtype=bool)
>>> np.array_equal(a, b)               Array-wise comparison
```

Aggregate Functions

```
>>> a.sum()                            Array-wise sum
>>> a.min()                            Array-wise minimum value
>>> b.max(axis=0)                      Maximum value of an array row
>>> b.cumsum(axis=1)                   Cumulative sum of the elements
>>> a.mean()                           Mean
>>> b.median()                         Median
>>> a.corrcoef()                       Correlation coefficient
>>> np.std(b)                          Standard deviation
```

Copying Arrays

```
>>> h = a.view()          Create a view of the array with the same data
>>> np.copy(a)            Create a copy of the array
>>> h = a.copy()          Create a deep copy of the array
```

Sorting Arrays

```
>>> a.sort()              Sort an array
>>> c.sort(axis=0)        Sort the elements of an array's axis
```

Subsetting, Slicing, Indexing

Also see Lists

Subsetting

```
>>> a[2]                               Select the element at the 2nd index
  3
>>> b[1,2]                             Select the element at row 0 column 2
  6.0                                  (equivalent to b[1][2])
```

Slicing

```
>>> a[0:2]                             Select items at index 0 and 1
  array([1, 2])
>>> b[0:2,1]                           Select items at rows 0 and 1 in column 1
  array([ 2.,  5.])
>>> b[:1]                              Select all items at row 0
  array([[1.5, 2., 3.]])               (equivalent to b[0:1, :])
>>> c[1,...]                           Same as [1,:,:]
  array([[[ 3.,  2.,  1.],
          [ 4.,  5.,  6.]]])
>>> a[ : :-1]                          Reversed array a
  array([3, 2, 1])
```

Boolean Indexing

```
>>> a[a<2]                             Select elements from a less than 2
  array([1])
```

Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]       Select elements (1,0),(0,1),(1,2) and (0,0)
  array([ 4. , 2. , 6. , 1.5])
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]       Select a subset of the matrix's rows
  array([[ 4. ,  5. ,  6. ,  4. ],     and columns
         [ 1.5,  2. ,  3. ,  1.5],
         [ 4. ,  5. ,  6. ,  4. ],
         [ 1.5,  2. ,  3. ,  1.5]])
```

Array Manipulation

Transposing Array

```
>>> i = np.transpose(b)                Permute array dimensions
>>> i.T                                Permute array dimensions
```

Changing Array Shape

```
>>> b.ravel()                          Flatten the array
>>> g.reshape(3,-2)                    Reshape, but don't change data
```

Adding/Removing Elements

```
>>> h.resize((2,6))                    Return a new array with shape (2,6)
>>> np.append(h,g)                     Append items to an array
>>> np.insert(a, 1, 5)                 Insert items in an array
>>> np.delete(a,[1])                   Delete items from an array
```

Combining Arrays

```
>>> np.concatenate((a,d),axis=0)       Concatenate arrays
  array([ 1,  2,  3, 10, 15, 20])
>>> np.vstack((a,b))                   Stack arrays vertically (row-wise)
  array([[ 1. ,  2. ,  3. ],
         [ 1.5,  2. ,  3. ],
         [ 4. ,  5. ,  6. ]])
>>> np.r_[e,f]                         Stack arrays vertically (row-wise)
>>> np.hstack((e,f))                   Stack arrays horizontally (column-wise)
  array([[ 7.,  7.,  1.,  0.],
         [ 7.,  7.,  0.,  1.]])
>>> np.column_stack((a,d))             Create stacked column-wise arrays
  array([[ 1, 10],
         [ 2, 15],
         [ 3, 20]])
>>> np.c_[a,d]                         Create stacked column-wise arrays
```

Splitting Arrays

```
>>> np.hsplit(a,3)                     Split the array horizontally at the 3rd
  [array([1]),array([2]),array([3])]   index
>>> np.vsplit(c,2)                     Split the array vertically at the 2nd index
[array([[[ 1.5,  2. ,  1. ],
         [ 4. ,  5. ,  6. ]]]),
 array([[[ 3.,  2.,  3.],
         [ 4.,  5.,  6.]]])]
```

Python For Data Science *Cheat Sheet*

Pandas Basics

Learn Python for Data Science **Interactively** at www.DataCamp.com

Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis** tools for the Python programming language.

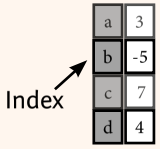
Use the following import convention:

```
>>> import pandas as pd
```

Pandas Data Structures

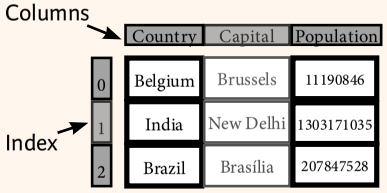
Series

A **one-dimensional** labeled array capable of holding any data type




```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame



A **two-dimensional** labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasília'],
            'Population': [11190846, 1303171035, 207847528]}

>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

```
>>> s['b']
 -5

>>> df[1:]
   Country    Capital  Population
1   India  New Delhi  1303171035
2  Brazil   Brasília   207847528
```

Get one element

Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[[0],[0]]
 'Belgium'
>>> df.iat[[0],[0]]
 'Belgium'
```

Select single value by row & column

By Label

```
>>> df.loc[[0], ['Country']]
 'Belgium'
>>> df.at[[0], ['Country']]
 'Belgium'
```

Select single value by row & column labels

By Label/Position

```
>>> df.ix[2]
Country      Brazil
Capital    Brasília
Population  207847528
```

Select single row of subset of rows

```
>>> df.ix[:,'Capital']
0     Brussels
1    New Delhi
2     Brasília
```

Select a single column of subset of columns

```
>>> df.ix[1,'Capital']
 'New Delhi'
```

Select rows and columns

Boolean Indexing

```
>>> s[~(s > 1)]
>>> s[(s < -1) | (s > 2)]
>>> df[df['Population']>1200000000]
```

Series `s` where value is not >1
`s` where value is <-1 or >2
Use filter to adjust DataFrame

Setting

```
>>> s['a'] = 6
```

Set index `a` of Series `s` to 6

Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
```

Drop values from rows (axis=0)
Drop values from columns(axis=1)

Sort & Rank

```
>>> df.sort_index()
>>> df.sort_values(by='Country')
>>> df.rank()
```

Sort by labels along an axis
Sort by the values along an axis
Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```

(rows,columns)
Describe index
Describe DataFrame columns
Info on DataFrame
Number of non-NA values

Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min()/df.max()
>>> df.idxmin()/df.idxmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```

Sum of values
Cummulative sum of values
Minimum/maximum values
Minimum/Maximum index value
Summary statistics
Mean of values
Median of values

Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
>>> df.applymap(f)
```

Apply function
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
 a     10.0
 b      NaN
 c      5.0
 d      7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
 a    10.0
 b    -5.0
 c    5.0
 d    7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
```

Read multiple sheets from the same file

```
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

`read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()`

```
>>> pd.to_sql('myDf', engine)
```

Python For Data Science *Cheat Sheet*

Matplotlib

Learn Python Interactively at www.DataCamp.com

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.

matplotlib

1 Prepare The Data

Also see Lists & NumPy

1D Data

```
>>> import numpy as np
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))
>>> data2 = 3 * np.random.random((10, 10))
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample_data
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

2 Create Plot

```
>>> import matplotlib.pyplot as plt
```

Figure

```
>>> fig = plt.figure()
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

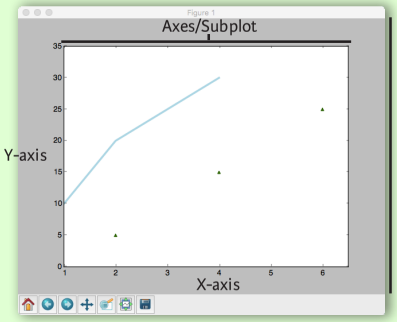
Axes

All plotting is done with respect to an `Axes`. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()
>>> ax1 = fig.add_subplot(221) # row-col-num
>>> ax3 = fig.add_subplot(212)
>>> fig3, axes = plt.subplots(nrows=2,ncols=2)
>>> fig4, axes2 = plt.subplots(ncols=3)
```

Plot Anatomy & Workflow

Plot Anatomy



Workflow

The basic steps to creating plots with matplotlib are:

1 Prepare data 2 Create plot 3 Plot 4 Customize plot 5 Save plot 6 Show plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4]                          Step 1
>>> y = [10,20,25,30]
>>> fig = plt.figure()                     Step 2
>>> ax = fig.add_subplot(111)              Step 3
>>> ax.plot(x, y, color='lightblue', linewidth=3)   Step 3, 4
>>> ax.scatter([2,4,6],
               [5,15,25],
               color='darkgreen',
               marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png')
>>> plt.show()                             Step 6
```

4 Customize Plot

Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
                   cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,y,'--',x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

Text & Annotations

```
>>> ax.text(1,
            -2.1,
            'Example Graph',
            style='italic')
>>> ax.annotate("Sine",
                xy=(8, 0),
                xycoords='data',
                xytext=(10.5, 0),
                textcoords='data',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="arc3"),)
```

Mathtext

```
>>> plt.title(r'$sigma_i=15$', fontsize=20)
```

Limits, Legends & Layouts

Limits & Autoscaling

```
>>> ax.margins(x=0.0,y=0.1)                  Add padding to a plot
>>> ax.axis('equal')                         Set the aspect ratio of the plot to 1
>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5])    Set limits for x-and y-axis
>>> ax.set_xlim(0,10.5)                      Set limits for x-axis
```

Legends

```
>>> ax.set(title='An Example Axes',          Set a title and x-and y-axis labels
           ylabel='Y-Axis',
           xlabel='X-Axis')
>>> ax.legend(loc='best')                    No overlapping plot elements
```

Ticks

```
>>> ax.xaxis.set(ticks=range(1,5),           Manually set x-ticks
                 ticklabels=[3,100,-12,"foo"])
>>> ax.tick_params(axis='y',                 Make y-ticks longer and go in and out
                   direction='inout',
                   length=10)
```

Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5,         Adjust the spacing between subplots
                         hspace=0.3,
                         left=0.125,
                         right=0.9,
                         top=0.9,
                         bottom=0.1)
>>> fig.tight_layout()                       Fit subplot(s) in to the figure area
```

Axis Spines

```
>>> ax1.spines['top'].set_visible(False)     Make the top axis line for a plot invisible
>>> ax1.spines['bottom'].set_position(('outward',10))   Move the bottom axis line outward
```

3 Plotting Routines

1D Data

```
>>> fig, ax = plt.subplots()
>>> lines = ax.plot(x,y)                     Draw points with lines or markers connecting them
>>> ax.scatter(x,y)                          Draw unconnected points, scaled or colored
>>> axes[0,0].bar([1,2,3],[3,4,5])           Plot vertical rectangles (constant width)
>>> axes[1,0].barh([0.5,1,2.5],[0,1,2])      Plot horiontal rectangles (constant height)
>>> axes[1,1].axhline(0.45)                  Draw a horizontal line across axes
>>> axes[0,1].axvline(0.65)                  Draw a vertical line across axes
>>> ax.fill(x,y,color='blue')                Draw filled polygons
>>> ax.fill_between(x,y,color='yellow')      Fill between y-values and o
```

Vector Fields

```
>>> axes[0,1].arrow(0,0,0.5,0.5)             Add an arrow to the axes
>>> axes[1,1].quiver(y,z)                    Plot a 2D field of arrows
>>> axes[0,1].streamplot(X,Y,U,V)            Plot a 2D field of arrows
```

Data Distributions

```
>>> ax1.hist(y)                              Plot a histogram
>>> ax3.boxplot(y)                           Make a box and whisker plot
>>> ax3.violinplot(z)                        Make a violin plot
```

2D Data or Images

```
>>> fig, ax = plt.subplots()
>>> im = ax.imshow(img,                      Colormapped or RGB arrays
                   cmap='gist_earth',
                   interpolation='nearest',
                   vmin=-2,
                   vmax=2)
```

```
>>> axes2[0].pcolor(data2)                   Pseudocolor plot of 2D array
>>> axes2[0].pcolormesh(data)                Pseudocolor plot of 2D array
>>> CS = plt.contour(Y,X,U)                  Plot contours
>>> axes2[2].contourf(data1)                 Plot filled contours
>>> axes2[2]= ax.clabel(CS)                  Label a contour plot
```

5 Save Plot

Save figures

```
>>> plt.savefig('foo.png')
```

Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

6 Show Plot

```
>>> plt.show()
```

Close & Clear

```
>>> plt.cla()                                Clear an axis
>>> plt.clf()                                Clear the entire figure
>>> plt.close()                              Close a window
```

Numpy Cheat Sheet

Python Package

Created By: Arianne Colton and Sean Chen

Numpy (Numerical Python)

What is NumPy?

Foundation package for scientific computing in Python

Why NumPy?

- Numpy **'ndarray'** is a much more efficient way of storing and manipulating **"numerical data"** than the built-in Python data structures.
- Libraries written in lower-level languages, such as C, can operate on data stored in Numpy **'ndarray'** without copying any data.

N-DIMENSIONAL ARRAY (NDARRAY)

What is NdArray?

Fast and space-efficient multidimensional array (container for homogeneous data) providing vectorized arithmetic operations

Create NdArray	`np.array(seq1)` # seq1 - is any sequence like object, i.e. [1, 2, 3]

Create Special NdArray	`1, np.zeros(10)`
	# one dimensional ndarray with 10 elements of value 0
	`2, np.ones(2, 3)`
	# two dimensional ndarray with 6 elements of value 1
	`3, np.empty(3, 4, 5)` *
	# three dimensional ndarray of uninitialized values
	`4, np.eye(N) or np.identity(N)`
	# creates N by N identity matrix

NdArray version of Python's `range`	`np.arange(1, 10)`
Get # of Dimension	`ndarray1.ndim`
Get Dimension Size	`dim1size, dim2size, .. = ndarray1.shape`
Get Data Type **	`ndarray1.dtype`
Explicit Casting	`ndarray2 = ndarray1.astype(np.int32)` ***

* Cannot assume empty() will return all zeros. It could be garbage values.

** Default data type is **'np.float64'**. This is equivalent to Python's float type which is 8 bytes (64 bits); thus the name 'float64'.

*** If casting were to fail for some reason, **'TypeError'** will be raised.

SLICING (INDEXING/SUBSETTING)

- Slicing (i.e. `ndarray1[2:6]`) is a **'view'** on the original array. **Data is NOT copied**. Any modifications (i.e. `ndarray1[2:6] = 8`) to the 'view' will be reflected in the original array.
- Instead of a 'view', explicit copy of slicing via :

```
ndarray1[2:6].copy()
```

- Multidimensional array indexing notation :

```
ndarray1[0][2] or ndarray1[0, 2]
```

* Boolean indexing :

```
ndarray1[(names == 'Bob') | (names == 'Will'), 2:]
# '2:' means select from 3rd column on
```

* Selecting data by boolean indexing **ALWAYS** creates a copy of the data.

* The 'and' and 'or' keywords do NOT work with boolean arrays. Use & and |.

* Fancy indexing (aka 'indexing using integer arrays')

Select a subset of rows in a particular order :

```
ndarray1[ [3, 8, 4] ]
ndarray1[ [-1, 6] ]
# negative indices select rows from the end
```

* Fancy indexing **ALWAYS** creates a copy of the data.

Numpy (Numerical Python)

Setting data with assignment :

```
ndarray1[ndarray1 < 0] = 0 *
```

* If ndarray1 is two-dimensions, ndarray1 < 0 creates a two-dimensional boolean array.

COMMON OPERATIONS

1. Transposing

- A special form of reshaping which returns a **'view'** on the underlying data without copying anything.

```
ndarray1.transpose()    or
ndarray1.T              or
ndarray1.swapaxes(0, 1)
```

2. Vectorized wrappers (for functions that take scalar values)

- `math.sqrt()` works on only a `scalar`

```
np.sqrt(seq1) # any sequence (list, ndarray, etc) to return a ndarray
```

3. Vectorized expressions

- `np.where(cond, x, y)` is a vectorized version of the expression 'x if condition else y'

```
np.where([True, False], [1, 2], [2, 3]) => ndarray (1, 3)
```

- Common Usages :

```
np.where(matrixArray > 0, 1, -1)
=> a new array (same shape) of 1 or -1 values
```

```
np.where(cond, 1, 0).argmax() *
=> Find the first True element
```

* `argmax()` can be used to find the index of the maximum element. Example usage is find the first element that has a "price > number" in an array of price data.

4. Aggregations/Reductions Methods (i.e. mean, sum, std)

Compute mean	`ndarray1.mean()` or `np.mean(ndarray1)`
Compute statistics over axis *	`ndarray1.mean(axis = 1)` `ndarray1.sum(axis = 0)`

* axis = 0 means column axis, 1 is row axis.

5. Boolean arrays methods

Count # of 'Trues' in boolean array	`(ndarray1 > 0).sum()`
If at least one value is 'True'	`ndarray1.any()`
If all values are 'True'	`ndarray1.all()`

Note: These methods also work with non-boolean arrays, where non-zero elements evaluate to True.

6. Sorting

Inplace sorting	`ndarray1.sort()`
Return a sorted copy instead of inplace	`sorted1 = np.sort(ndarray1)`

7. Set methods

Return sorted unique values	`np.unique(ndarray1)`
Test membership of ndarray1 values in [2, 3, 6]	`resultBooleanArray = np.in1d(ndarray1, [2, 3, 6])`

- Other set methods : `intersect1d()`,`union1d()`, `setdiff1d()`,`setxor1d()`

8. Random number generation (np.random)

- Supplements the built-in Python random * with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions.

```
samples = np.random.normal(size =(3, 3))
```

* Python built-in random ONLY samples one value at a time.

Created by Arianne Colton and Sean Chen

www.datasciencefree.com

Based on content from 'Python for Data Analysis' by Wes McKinney

Updated: August 18, 2016

Matplotlib for beginners

Matplotlib is a library for making 2D plots in Python. It is designed with the philosophy that you should be able to create simple plots with just a few commands:

1 Initialize

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

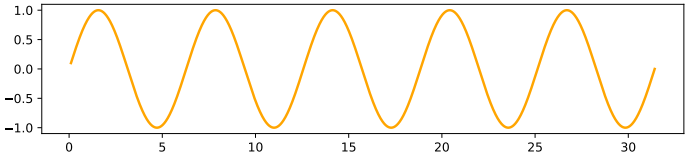
2 Prepare

```
X = np.linspace(0, 4*np.pi, 1000)
Y = np.sin(X)
```

3 Render

```
fig, ax = plt.subplots()
ax.plot(X, Y)
fig.show()
```

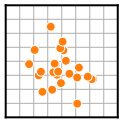
4 Observe



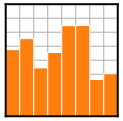
Choose

Matplotlib offers several kind of plots (see Gallery):

```
X = np.random.uniform(0, 1, 100)
Y = np.random.uniform(0, 1, 100)
ax.scatter(X, Y)
```



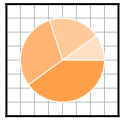
```
X = np.arange(10)
Y = np.random.uniform(1, 10, 10)
ax.bar(X, Y)
```



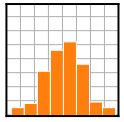
```
Z = np.random.uniform(0, 1, (8,8))

ax.imshow(Z)
```

```
Z = np.random.uniform(0, 1, (8,8))

ax.contourf(Z)
```

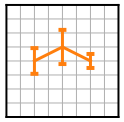
```
Z = np.random.uniform(0, 1, 4)

ax.pie(Z)
```



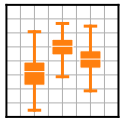
```
Z = np.random.normal(0, 1, 100)

ax.hist(Z)
```



```
X = np.arange(5)
Y = np.random.uniform(0, 1, 5)
ax.errorbar(X, Y, Y/4)
```



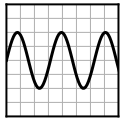
```
Z = np.random.normal(0, 1, (100,3))

ax.boxplot(Z)
```



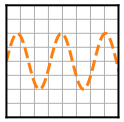
Tweak

You can modify pretty much anything in a plot, including limits, colors, markers, line width and styles, ticks and ticks labels, titles, etc.

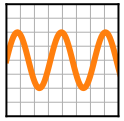
```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, color="black")
```



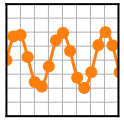
```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, linestyle="--")
```



```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, linewidth=5)
```



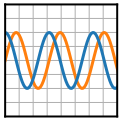
```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, marker="o")
```



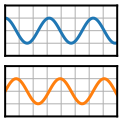
Organize

You can plot several data on the the same figure, but you can also split a figure in several subplots (named *Axes*):

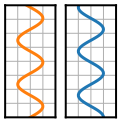
```
X = np.linspace(0, 10, 100)
Y1, Y2 = np.sin(X), np.cos(X)
ax.plot(X, Y1, X, Y2)
```



```
fig, (ax1, ax2) = plt.subplots((2,1))
ax1.plot(X, Y1, color="C1")
ax2.plot(X, Y2, color="C0")
```

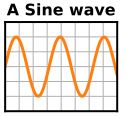


```
fig, (ax1, ax2) = plt.subplots((1,2))
ax1.plot(Y1, X, color="C1")
ax2.plot(Y2, X, color="C0")
```

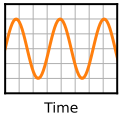


Label (everything)

```
ax.plot(X, Y)
fig.suptitle(None)
ax.set_title("A Sine wave")
```




```
ax.plot(X, Y)
ax.set_ylabel(None)
ax.set_xlabel("Time")
```



Explore

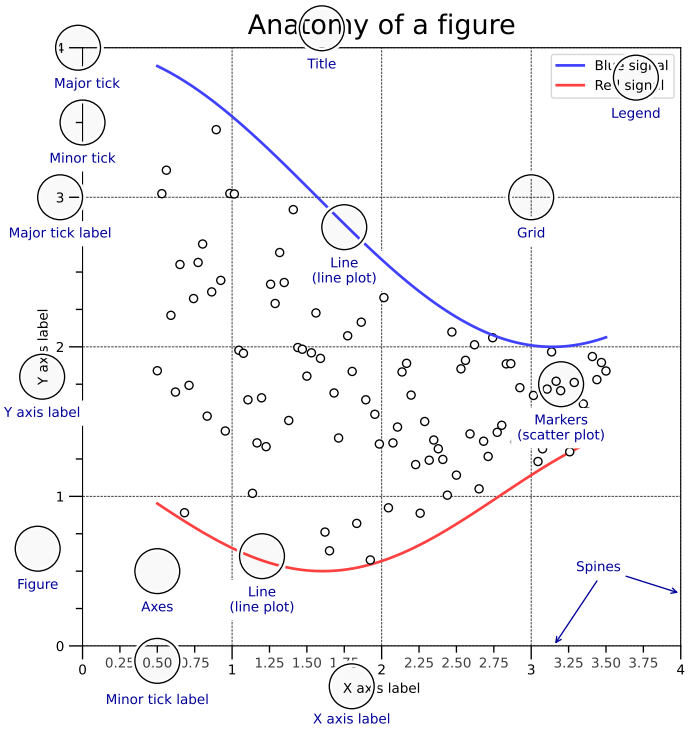
Figures are shown with a graphical user interface that allows to zoom and pan the figure, to navigate between the different views and to show the value under the mouse.

Save (bitmap or vector format)

```
fig.savefig("my-first-figure.png", dpi=300)
fig.savefig("my-first-figure.pdf")
```

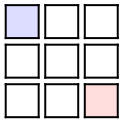
Matplotlib for intermediate users

A matplotlib figure is composed of a hierarchy of elements that forms the actual figure. Each element can be modified.

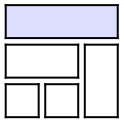


Figure, axes & spines

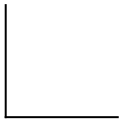
```
fig, axs = plt.subplots(3,3)
axs[0,0].set_facecolor("#ddddff")
axs[2,2].set_facecolor("#ffffdd")
```



```
gs = fig.add_gridspec(3, 3)
ax = fig.add_subplot(gs[0, :])
ax.set_facecolor("#ddddff")
```

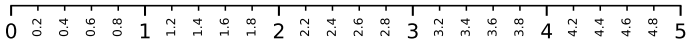


```
fig, ax = plt.subplots()
ax.spines["top"].set_color("None")
ax.spines["right"].set_color("None")
```



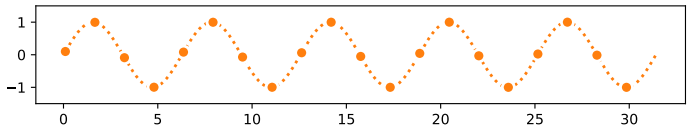
Ticks & labels

```
from mpl.ticker import MultipleLocator as ML
from mpl.ticker import ScalarFormatter as SF
ax.xaxis.set_minor_locator(ML(0.2))
ax.xaxis.set_minor_formatter(SF())
ax.tick_params(axis='x',which='minor',rotation=90)
```



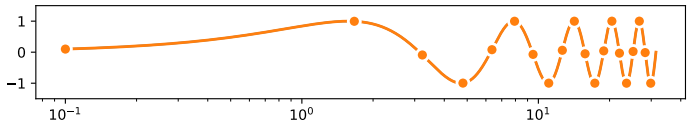
Lines & markers

```
X = np.linspace(0.1, 10*np.pi, 1000)
Y = np.sin(X)
ax.plot(X, Y, "C1o:", markevery=25, mec="1.0")
```



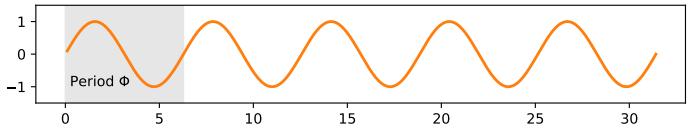
Scales & projections

```
fig, ax = plt.subplots()
ax.set_xscale("log")
ax.plot(X, Y, "C1o-", markevery=25, mec="1.0")
```



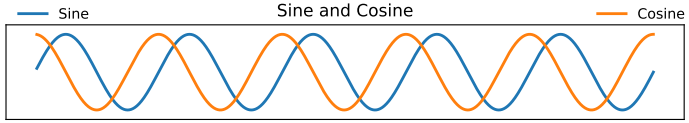
Text & ornaments

```
ax.fill_betweenx([-1,1],[0],[2*np.pi])
ax.text(0, -1, r" Period $\Phi$")
```



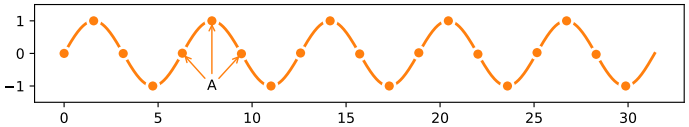
Legend

```
ax.plot(X, np.sin(X), "C0", label="Sine")
ax.plot(X, np.cos(X), "C1", label="Cosine")
ax.legend(bbox_to_anchor=(0,1,1,.1),ncol=2,
          mode="expand",  loc="lower left")
```



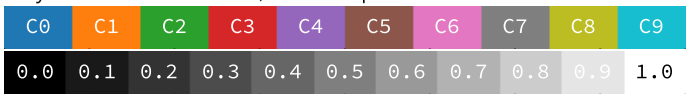
Annotation

```
ax.annotate("A", (X[250],Y[250]),(X[250],-1),
  ha="center", va="center",arrowprops =
  {"arrowstyle" : "->", "color": "C1"})
```



Colors

Any color can be used, but Matplotlib offers sets of colors:



Size & DPI

Consider a square figure to be included in a two-columns A4 paper with 2cm margins on each side and a column separation of 1cm. The width of a figure is (21 - 2*2 - 1)/2 = 8cm. One inch being 2.54cm, figure size should be 3.15×3.15 in.

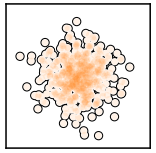
```
fig = plt.figure(figsize=(3.15,3.15), dpi=50)
plt.savefig("figure.pdf", dpi=600)
```

Matplotlib tips & tricks

Transparency

Scatter plots can be enhanced by using transparency (alpha) in order to show area with higher density. Multiple scatter plots can be used to delineate a frontier.

```
X = np.random.normal(-1, 1, 500)
Y = np.random.normal(-1, 1, 500)
ax.scatter(X, Y, 50, "0.0", lw=2) # optional
ax.scatter(X, Y, 50, "1.0", lw=0) # optional
ax.scatter(X, Y, 40, "C1",  lw=0, alpha=0.1)
```



Rasterization

If your figure has many graphical elements, such as a huge scatter, you can rasterize them to save memory and keep other elements in vector format.

```
X = np.random.normal(-1, 1, 10_000)
Y = np.random.normal(-1, 1, 10_000)
ax.scatter(X, Y, rasterized=True)
fig.savefig("rasterized-figure.pdf", dpi=600)
```

Offline rendering

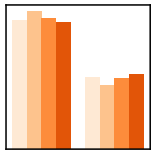
Use the Agg backend to render a figure directly in an array.

```
from matplotlib.backends.backend_agg import FigureCanvas
canvas = FigureCanvas(Figure())
... # draw some stuff
canvas.draw()
Z = np.array(canvas.renderer.buffer_rgba())
```

Range of continuous colors

You can use colormap to pick from a range of continuous colors.

```
X = np.random.randn(1000, 4)
cmap = plt.get_cmap("Oranges")
colors = cmap([0.2, 0.4, 0.6, 0.8])

ax.hist(X, 2, histtype='bar', color=colors)
```



Text outline

Use text outline to make text more visible.

```
import matplotlib.patheffects as fx
text = ax.text(0.5, 0.1, "Label")
text.set_path_effects([
  fx.Stroke(linewidth=3, foreground='1.0'),
  fx.Normal()])
```

Multiline plot

You can plot several lines at once using None as separator.

```
X,Y = [], []
for x in np.linspace(0, 10*np.pi, 100):
  X.extend([x, x, None]), Y.extend([0, sin(x), None])
ax.plot(X, Y, "black")
```

Dotted lines

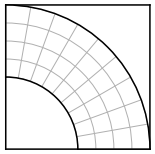
To have rounded dotted lines, use a custom `linestyle` and modify `dash_capstyle`.

```
ax.plot([0,1], [0,0], "C1",
        linestyle = (0, (0.01, 1)), dash_capstyle="round")
ax.plot([0,1], [1,1], "C1",
        linestyle = (0, (0.01, 2)), dash_capstyle="round")
```

Combining axes

You can use overlaid axes with different projections.

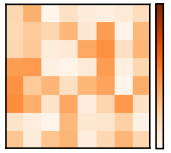
```
ax1 = fig.add_axes([0,0,1,1],
                   label="cartesian")
ax2 = fig.add_axes([0,0,1,1],
                   label="polar",
                   projection="polar")
```



Colorbar adjustment

You can adjust a colorbar's size when adding it.

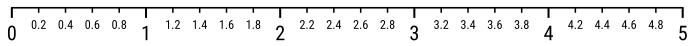
```
im = ax.imshow(Z)

cb = plt.colorbar(im,
        fraction=0.046, pad=0.04)
cb.set_ticks([])
```



Taking advantage of typography

You can use a condensed font such as Roboto Condensed to save space on tick labels.

```
for tick in ax.get_xticklabels(which='both'):
    tick.set_fontname("Roboto Condensed")
```



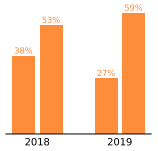
Getting rid of margins

Once your figure is finished, you can call `tight_layout()` to remove white margins. If there are remaining margins, you can use the `pdfcrop` utility (comes with TeX live).

Hatching

You can achieve a nice visual effect with thick hatch patterns.

```
cmap = plt.get_cmap("Oranges")
plt.rcParams['hatch.color'] = cmap(0.2)
plt.rcParams['hatch.linewidth'] = 8
ax.bar(X, Y, color=cmap(0.6), hatch="/" )
```



Read the documentation

Matplotlib comes with an extensive documentation explaining the details of each command and is generally accompanied by examples. Together with the huge online gallery, this documentation is a gold-mine.