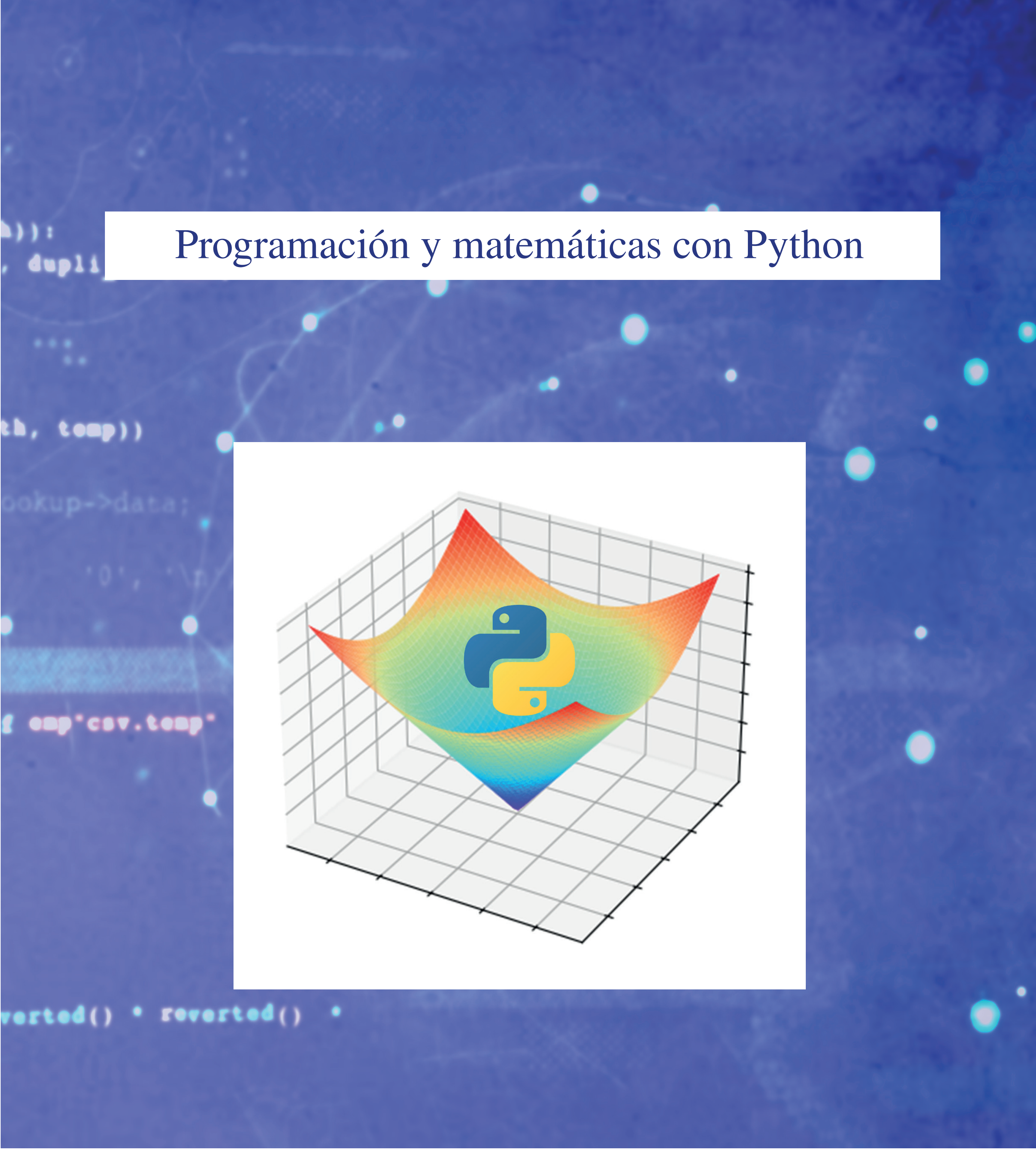
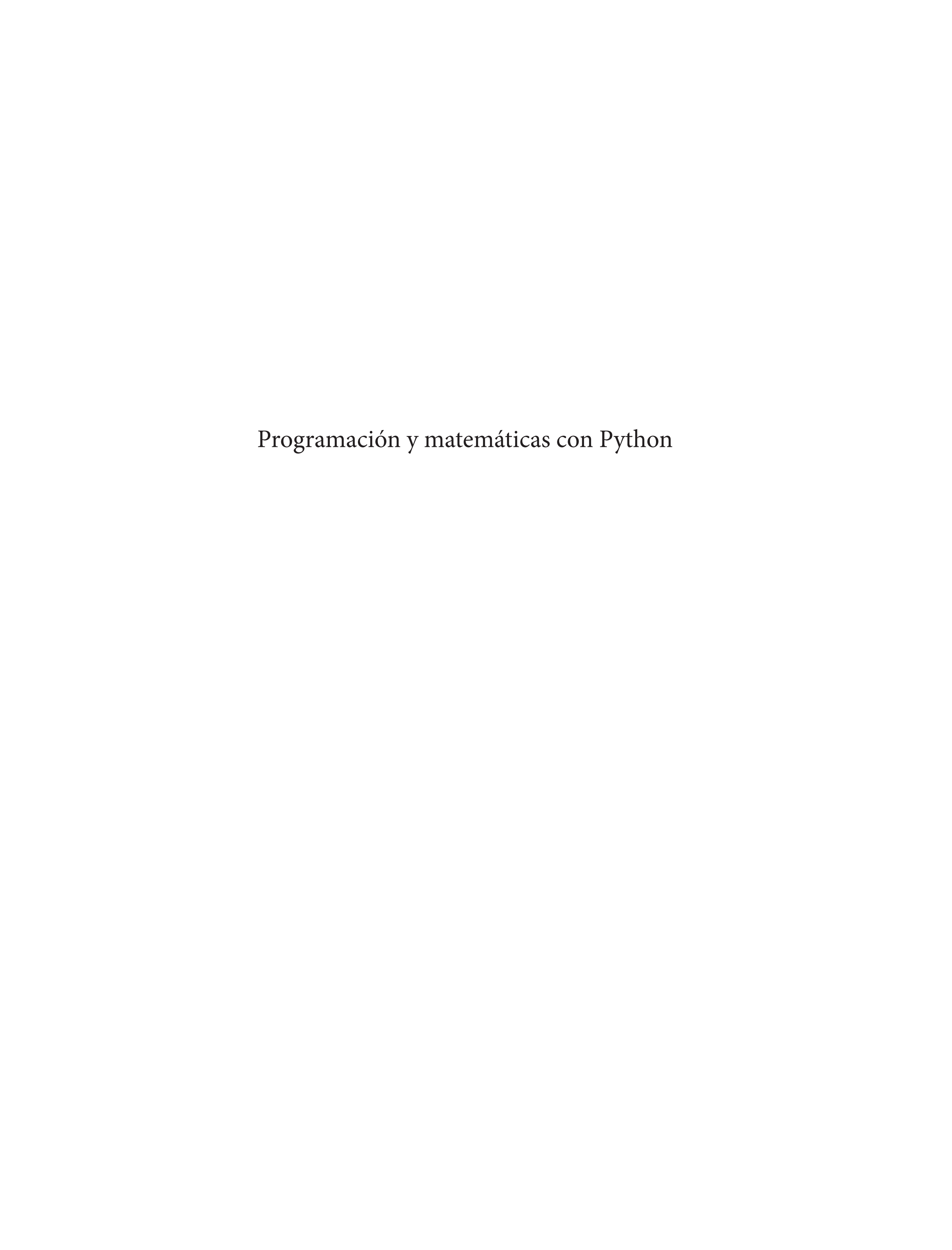
Programación y matemáticas con Python



Erika Lorena Álvarez Ramírez



Programación y matemáticas con Python

Universidad Autónoma de la Ciudad de México

M en C. Juan Carlos Aguilar Franco
Rector

Dra. María Elizbeth Álvarez Sánchez
Coordinadora Académica

Lic. Jorge Luis Rubio Hernández
Coordinador de Difusión Cultural y Extensión Universitaria

Equipo de la Biblioteca del Estudiante

Ángeles Godínez Guevara
Responsable

Ana Beatriz Alonso Osorio
Ana Lina Graciano Franco
Daniel Valentín Cruz
Florina Piña Cancino
María del Pilar Aparicio Romero
Sergio Javier Cortés Becerril

Programación y matemáticas con Python

Erika Lorena Álvarez Ramírez

FICHA CATALOGRÁFICA E-S/N

Álvarez Ramírez, Erika Lorena

Programación y matemáticas con Python / Erika Lorena Álvarez Ramírez. -- Primera edición. -- Ciudad de México : Universidad Autónoma de la Ciudad de México, Biblioteca del Estudiante, 2025.

180 páginas : diagramas, gráficas, tablas ; 21 cm

Bibliografía: páginas 179-180.
ISBN: En trámite

1. Python (Lenguaje de programación para computadora). 2. Estructura de datos (Computación). 3. Matrices (Matemáticas). 4. Aprendizaje automático (Inteligencia artificial). I. Título.

LC QA76.73.P98 Dewey 005.133

Programación y matemáticas con Python

primera edición, 2025

© Erika Lorena Álvarez Ramírez
D.R. © Universidad Autónoma de la Ciudad de México
García Diego 168, col. Doctores,
alc. Cuauhtémoc, c. p. 06720, México, D F

ISBN: 978-607-2615-62-5

https://www.uacm.edu.mx/Organizacion/CoordinacionAcademica/Biblioteca_Estudiante

Material educativo universitario de distribución gratuita para estudiantes de la UACM. Prohibida su venta

Hecho e impreso en México

Presentación

Para los estudiantes de Ciencia y Tecnología de la UACM, una de las últimas materias a certificar, a pesar de ser del primer semestre es Introducción a la Programación. Una de las sugerencias que se han hecho en la discusión de la revisión de los planes de estudio del Colegio, es que Introducción a la programación se dé en un lenguaje más sencillo e intuitivo, como `Python`, para todas las licenciaturas excepto probablemente Ingeniería de Software.

`Python` es un lenguaje de programación intuitivo que, entre otras bondades, tiene asociadas bibliotecas sobre ciertas áreas de las ciencias, para poder usar sus funciones de manera sencilla. Es gratuito y tiene una amplia distribución, al figurar en la top list de los lenguajes de programación más utilizados.

Hay materias de matemáticas y de física del Ciclo Básico del Colegio de Ciencia y Tecnología que usualmente sólo se imparten de manera teórica en el aula. Lo ideal es acompañar estas materias utilizando el aula de cómputo, como parte de las estrategias de enseñanza-aprendizaje, para explorar soluciones a problemas cambiando parámetros y graficando. `Python` es un lenguaje ideal para este propósito, por su sencillez, y este libro fomenta su uso.

En el caso del Ciclo Superior, la licenciatura en Modelación Matemática cuenta con materias que requieren forzosamente del aula de cómputo, como es el caso de las materias de Programación Avanzada, Modelación Matemática y Computacional I y II, Métodos Numéricos en Ecuaciones Diferenciales y Álgebra Lineal Numérica, pero también materias como la de Optimización, Álgebra Lineal Avanzada, Estadística I, entre otras, se ven beneficiadas por el uso de software dirigido a los temas que se ven en esas materias. Este libro fomenta el uso de un lenguaje abierto con bibliotecas especializadas para tratar algunos temas, contra el uso de softwares de paga. En particular, además de la graficación, en este libro se toca el tema de resolución numérica de Ecuaciones Diferenciales, que se encuentran en los temarios actuales de Métodos Numéricos en Ecuaciones Diferenciales y Modelación Matemática y Computacional I, el tema de Estadística, que se encuentra en los temarios de Estadística y probabilidad (Ciclo Básico), Estadística I y más materias optativas de esa línea, el tema de Regresión Lineal, que se ve en los temarios de diversas materias (Álgebra Lineal Numérica, Probabilidad II, Estadística II, laboratorios de Física), y, por supuesto, el tema de Álgebra Lineal, para todas las materias de esa área. Los temas abarcados en este libro, sin embargo, no son de uso exclusivo de la licenciatura en Modelación Matemática, pues son lo bastante generales como para ser de interés también para las ingenierías y algunos de los posgrados.

El lector encontrará, adicionalmente, módulos de introducción a tratamientos de datos y de machine learning, tan necesarios en la actualidad, que serán de utilidad no sólo para los estudiantes, sino también para los profesores del Colegio.

No hay, hasta el momento, libros publicados en la UACM en relación con este lenguaje tan extendido. El abordaje de los temas es con aplicación directa con ejemplos, dejando ejercicios de práctica para el lector. El lenguaje del libro es sencillo, de fácil acceso para el usuario.

Índice general

1. Introducción **13**
1.1. Instalación de la distribución Anaconda 13

2. Lo básico de Python **17**
2.1. Comentarios en el programa 17
2.2. Operaciones con números 17
2.3. Módulos 19
2.3.1. Importación de módulos 19
2.3.2. El módulo math 20
2.3.3. El módulo random 20
2.4. Cadenas (strings) 21
2.4.1. Concatenación, slicing y len() 23
2.4.2. Imprimir el resultado de una operación como cadena 24
2.5. Listas 25
2.5.1. Copia de una lista 27
2.6. Booleanos 28
2.7. Condicionales 29
2.8. Bucles for (for loops) 30
2.8.1. for i in range(a,b) 30
2.8.2. for i in A 32
2.8.3. for i,j in enumerate(A) 33
2.8.4. for i,j in zip(A,B) 33
2.8.5. Llenar una lista, lista por comprensión 34
2.9. Bucles while (while loops) 37
2.10. Funciones 38
2.10.1. Recursividad 40
2.11. Entradas externas 41
2.11.1. Inputs 41
2.11.2. Leer archivos 42

3. Estructuras de datos básicos **47**
3.1. Conjuntos 47
3.2. Diccionarios 49
3.3. Tuplas 52

4. Numpy **57**

4.1. Generación de arreglos . . . 57
4.1.1. Vectores . . . 58
4.1.2. Matrices . . . 60
4.2. Operaciones con vectores y matrices . . . 61
4.2.1. Operaciones básicas . . . 61
4.2.2. Broadcasting . . . 63
4.2.3. Diferencias entre un vector y una matriz . . . 64
4.2.4. Álgebra Lineal . . . 65
4.3. Comparaciones . . . 66
4.4. Selección de elementos de un arreglo . . . 67
4.4.1. Slicing . . . 67
4.4.2. Selección con fancy indexing . . . 68
4.4.3. Selección por condición . . . 69
4.5. Operaciones de agregación . . . 70
4.5.1. Operaciones por renglón o por columnas . . . 71

5. Imágenes y gráficas 73
5.1. Matplotlib.pyplot . . . 74
5.1.1. Gráficas en dos dimensiones: plot y scatter . . . 74
5.1.2. Subfiguras y gráficas con estilo orientado a objetos . . . 77
5.1.3. Histogramas, hist . . . 80
5.1.4. Gráficas de barra: bar y barh . . . 81
5.2. Imágenes y proyecciones en el plano . . . 83
5.2.1. Leer y mostrar imágenes: imread e imshow . . . 83
5.2.2. Heat maps y curvas de nivel: imshow, contour y pcolormesh . . . 86
5.3. Gráficas en tres dimensiones . . . 90
5.3.1. Puntos en 3D: scatter3D . . . 90
5.3.2. Curvas en 3D: plot3D . . . 91
5.3.3. Superficies en 3D: plot_surface . . . 91
5.4. Animaciones . . . 93

6. Scipy y aplicaciones en matemáticas 95
6.1. Soluciones numéricas a EDO . . . 95
6.1.1. Método Runge-Kutta para una ecuación diferencial . . . 96
6.1.2. Método de Runge-Kutta para un sistema de dos EDO . . . 98
6.1.3. Campos de direcciones . . . 101
6.1.4. Función odeint de scipy.integrate . . . 101
6.2. Distribuciones de probabilidad . . . 105
6.2.1. Distribución binomial . . . 105
6.2.2. Distribución normal . . . 107
6.3. Estadística: intervalos de confianza . . . 108
6.3.1. Promedios . . . 111
6.3.2. Proporciones . . . 112
6.4. Estadística: prueba de hipótesis . . . 113
6.4.1. Diferencia de promedios . . . 113
6.4.2. Diferencia de proporciones . . . 114
6.5. Regresión lineal con mínimos cuadrados . . . 115

6.5.1. Ajuste de una recta con mínimos cuadrados 116
6.5.2. stats.linregress 120
6.5.3. statsmodel.api.OLS 121

7. Pandas **125**
7.1. Lectura y exploración de datos 125
7.2. Agregación, filtrado, agrupamiento 131
7.2.1. Ejercicio de práctica: coronavirus 135

8. Machine Learning **141**
8.1. Métodos supervisados: Clasificación 141
8.1.1. Preparación y exploración de datos: Flores iris 142
8.1.2. Datos de entrenamiento y prueba y preparación para clasificar . 147
8.1.3. Vecinos más cercanos 149
8.1.4. Árbol de decisión 151
8.1.5. Regresión Logística 156
8.1.6. Support Vector Machines 159
8.1.7. Gaussian Naive Bayes 163
8.1.8. Reescalamiento de datos 166
8.2. Aprendizaje no supervisado: PCA 167
8.3. Aprendizaje no supervisado: Clustering 171
8.3.1. k-Means Clustering 171
8.3.2. Clustering jerárquico 174

Bibliografía **177**

1 Introducción

Hay muchos lenguajes de programación disponibles, y, dependiendo del objetivo, unos tendrán ventaja sobre otros.[1] En lo que se refiere a ciencias, `Python` es uno de los mejores lenguajes, en parte por su sencillez, y en parte porque tiene varios módulos específicos a determinadas áreas, por ejemplo, en ciencia de datos, aprendizaje automático, astronomía, robótica, etc. En todo caso, `Python` es el lenguaje de programación de mayor crecimiento actualmente, y la mayoría de las principales Universidades de Estados Unidos usan este lenguaje en sus cursos de Introducción a la Programación.[2] En este libro, entonces, trataremos de mostrar las bases de `Python`, para que cuando un estudiante de matemáticas, física o ingeniería se tope con problemas de sus diversas materias los pueda desarrollar fácilmente con este lenguaje.

El contenido del libro se divide en, primero, dos capítulos que dan las bases para programar, siguiendo con dos capítulos para `numpy` y `matplotlib`, que son módulos o bibliotecas para trabajar, entre otros, en operaciones de vectores y matrices, y para graficar, respectivamente. No se necesita ser un experto en programación para avanzar con estos segundos capítulos, es algo similar, por ejemplo, a Matlab, donde lo que se quiere es hacer operaciones matemáticas sencillas, y graficar. Enseguida, viene un capítulo con aplicaciones para matemáticas de semestres más avanzados, como Ecuaciones Diferenciales Ordinarias y su solución numérica, Probabilidad, Estadística, y Regresión Lineal con mínimos cuadrados. Finalmente, para mantenerse al corriente con los temas actuales de las ciencias y la programación, se termina el libro con un capítulo de Introducción a la Ciencia de datos, con el módulo `pandas`, y un capítulo sobre Machine Learning, con el módulo `sklearn`.

En este libro, por su sencillez, se trabajará con Programación estructurada, no se incluirá todavía la Programación orientada a objetos.

1.1. Instalación de la distribución Anaconda

Hay varias distribuciones disponibles de `Python`, que, además de la distribución básica, contienen muchos más módulos que nos son de utilidad, por ejemplo, para graficar, y así ya no sería necesario en general instalar módulos cada vez que se requirieran. La distribución que se utilizó para este libro, y que recomendamos ampliamente, es la

[1] https://keepcoding.io/blog/mejor-lenguaje-de-programacion/

[2] https://cacm.acm.org/blogcacm/python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/

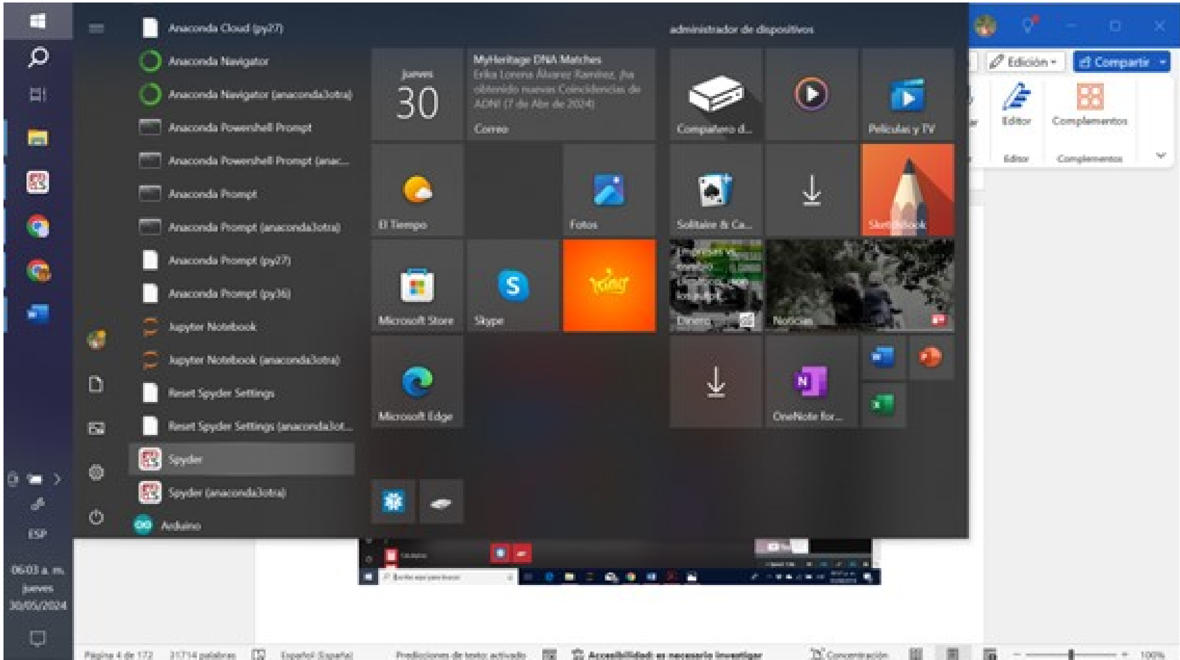


Figura 1.1: Abrir spyder desde Windows

distribución `Anaconda`, disponible en https://www.anaconda.com/products/individual.

`Anaconda` contiene el editor `spyder`, que consiste en tres ventanas: el editor, la terminal `IPython`, y una ventanita para desplegar figuras o cuando hay dudas sobre alguna función. Es posible realizar operaciones directamente en la terminal, pero cuando se escriben programas que deban salvarse en un archivo, por estar largos o para usarse posteriormente, se usa necesariamente el editor, y, la salida o resultado del programa, se despliega en la terminal.

Para abrir `Spyder` en Windows, en la ventana de Inicio localiza `Anaconda` y da click en `spyder` (Figura 1.1). En otros sistemas operativos, y también en Windows, puedes abrir la terminal, `"Anaconda prompt"`, y teclear `spyder`. O puedes abrir `Anaconda Navigator` y lanzar `spyder` desde ahí.

En el editor, que se encuentra en la ventana de la izquierda, tenemos los programas largos, y, para poder mandar imprimir algo a la terminal, localizada en la ventana inferior derecha, se usa la función `print()`. Como se ve en la Figura 1.2, podemos tener varios archivos de `Python` en diferentes pestañas. Para crear un nuevo archivo vas a Archivo → Nuevo archivo, Archivo → Guardar, seleccionar la carpeta donde va a estar el archivo, dar un nombre y en "tipos de archivo soportados" escoger Archivos `Python`. Para correr el programa se usa el botón de play.

En el editor, varios de los errores se pueden encontrar fácilmente, hay un símbolo de alerta al inicio de las líneas que contienen errores, y al pasar el cursor por encima se lee el tipo de error cometido. También, cuando se escribe un objeto que tiene funciones o métodos disponibles, éstos se despliegan. Además, al escribir una función, al momento de poner los paréntesis se pueden ver cuáles son los argumentos que requiere la función.

En lo que respecta a la terminal, cuando ingresamos algo, aparece la etiqueta `In [n]:`,

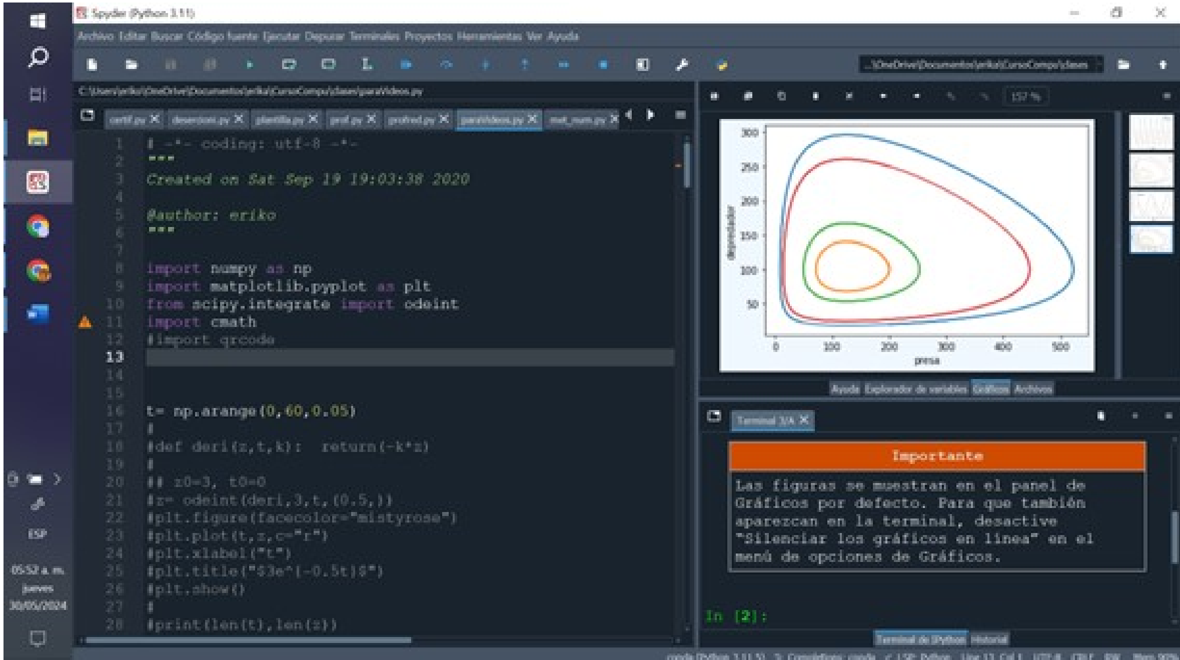


Figura 1.2: Editor (izquierda), figuras (arriba, derecha) y terminal (abajo, derecha) de spyder)

de input, donde n es la n-ésima entrada, y el resultado o output viene señalado con `Out[n]:`

Además del `spyder`, `Anaconda` viene con una "notebook", la `Jupyter notebook`, que es una aplicación web que permite escribir código junto con notas. Además de `Python`, la `jupyter` soporta otros lenguajes de programación como R, Julia, SQL, y muchos más. De hecho, `Jupyter` es el acrónimo de Julia-Python-R, aunque el lenguaje de inicio fue `Python`.

Aunque es más recomendable usar el `spyder` de inicio, podemos dar una breve descripción de la `jupyter`. Para abrir la aplicación, seguimos el mismo procedimiento que con el `spyder` pero lo reemplazamos por `jupyter notebook`. Se va a abrir una página de internet, así que para crear un nuevo archivo primero escogemos el folder deseado. Una vez en el folder, en la ventana de `New` seleccionamos `Python 3` (Figura 1.3).

Se abre el archivo `(ipynb)` (Figura 1.4) y se le puede dar un nombre dando click en la parte de arriba en `Untitled`. Para salvar está el botón de siempre para salvar, y para salir vamos a File y seleccionamos Close and halt. Para escribir código, se escribe directamente en las celdas, y para escribir texto se da click en Cell → Cell Type → Markdown, donde puedes escribir sin mayor preámbulo o se le puede dar formato al texto con Markdown o html. Para correr las celdas se usan las teclas Shift-Enter, o se da click en el botón de Run. Para eliminar un archivo, nos regresamos a la pestaña que dice Home, seleccionamos el archivo a eliminar y aparece un bote de basura en rojo en la parte de arriba, damos click. Puedes ver más de la `jupyter notebook` en la sección de `Help`, aunque, como ya mencionamos, es preferible que te entrenes en el `spyder` para checar más fácilmente los errores.

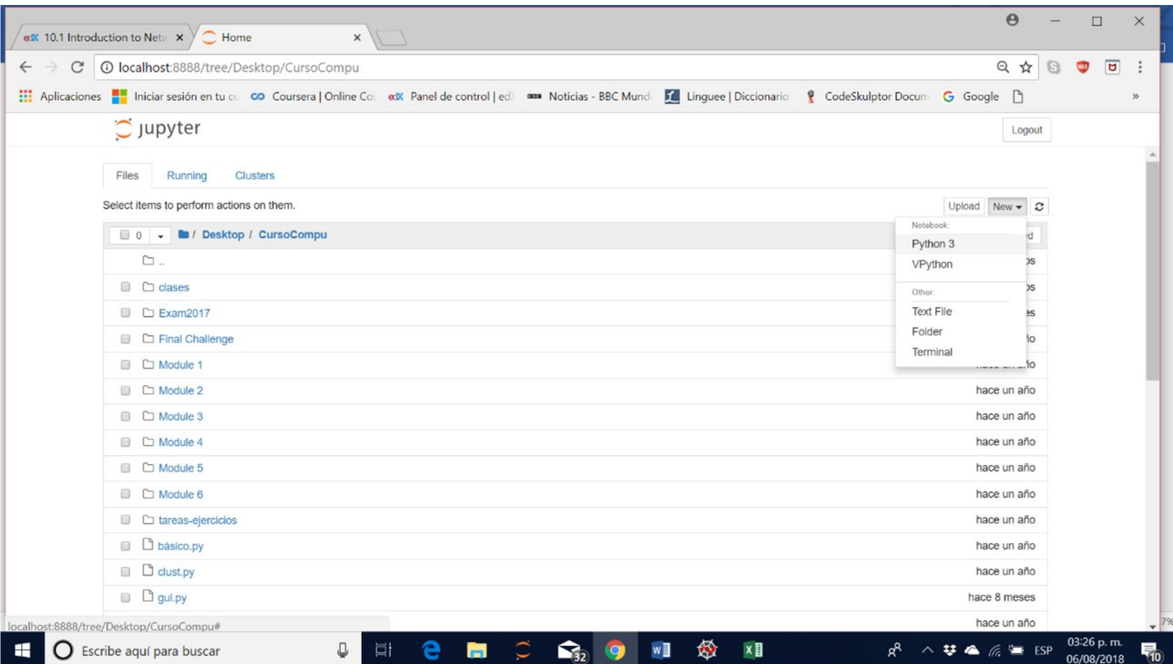


Figura 1.3: Abriendo un nuevo archivo en la jupyter notebook

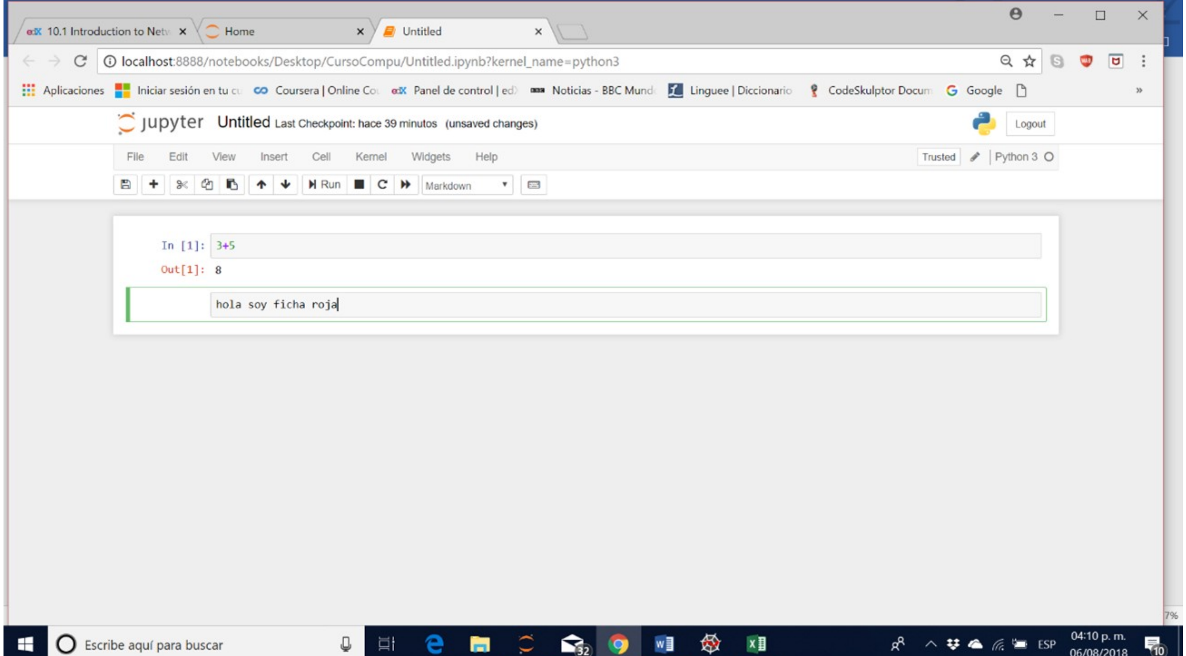


Figura 1.4: En la jupyter notebook

2 Lo básico de Python

En este capítulo damos los elementos básicos para programar en Python. Los objetivos de aprendizaje son que el lector aprenderá a operar con números, cadenas y booleanos en este lenguaje. Aplicará estructuras condicionales y ciclos for y while para resolver problemas. Conocerá y aplicará en problemásticas la estructura de listas. Aprenderá a definir funciones que son una de las bases fundamentales en programación, al poder ser llamadas cada vez que se requieran, sin tener que repetir líneas de código. Aprenderá a importar módulos, que son bibliotecas que contienen funciones para ciertas aplicaciones, y operará con algunas de ellas. Aprenderá a interaccionar con entradas de usuarios externos y a leer archivos de texto para procesar la información.

Como ya se mencionó en la Introducción, al trabajar en el editor se deben mandar imprimir los resultados a la terminal con `print()`. En este libro, la notación `In [n]` se refiere a lo que ingresamos en la computadora, el input, y `Out[n]` es la respuesta, el output. Cuando una expresión de código es muy larga como para necesitar más de un renglón, se usa el salto de línea "\".

2.1. Comentarios en el programa

Cuando escribes un programa, a veces requerimos escribir notas o comentarios que no afecten su funcionamiento, por lo que deben aparecer de manera diferente a los comandos para que cuando se corra el programa sean ignorados. En Python, los comentarios de una sola línea son precedidos por un símbolo de gato, #, y si es un comentario largo de varias líneas, se encierra todo el comentario entre tres comillas, ya sea simples o dobles. También, en spyder, cuando estás escribiendo un programa largo, y una sección de cálculos o de gráficas no es relevante en ese momento, puedes comentar toda esa sección con la herramienta Comentar/Descomentar. Para ello seleccionas la sección a comentar, y con el botón derecho del mouse seleccionas Comentar/Descomentar, o alternativamente, en Editar seleccionas Comentar/Descomentar

2.2. Operaciones con números

Los números reales se clasifican en enteros, `int`, y en floats, `float`, con punto decimal. Podemos ver el tipo de variable con la función `type()`

```
1 In [1]:
2 print(type(5))
3 Out[1]:
```

```
int
```

```
In [2]:
print(type(3.14))
Out[2]:
float
```

Las operaciones básicas son la suma (+), resta (−), multiplicación (∗), división (/), división entera (//), potenciación (∗∗) y el módulo o residuo (%):

```
In [3]:
print(4+5, 4-3, 2*4, 16/3, 16//3, 4**3, 7%3)
Out[3]:
9 1 8 5.333333333333333 5 64 1
```

Como podemos ver de este ejercicio, se pueden realizar varias operaciones en un solo renglón, separando con comas las operaciones.

Para escribir en notación científica, se usa la forma neN, donde n es el número, e corresponde a 10 elevado a la, y N es la potencia:

```
In [4]:
# las constantes de Planck y de la luz:
h, c= 6.63e-34, 3e8
# de E = hν = hc/λ calculamos la energía de un fotón con λ = 1Å:
print(h*c/1e-10)
Out[4]:
1.9889999999999998e-15
```

En este último ejercicio definimos dos variables en un renglón con un sólo signo de igual. Podemos hacer lo mismo, en un renglón, dividiendo con un punto y coma cada definición. Si se definen variables en renglones separados, no es necesario el punto y coma:

```
In [5]:
a = 2; b = 3
c = 4
print(a*b*c)
Out[5]:
24
```

Ejercicios:

1. Calcula aproximadamente cuántos días han transcurrido desde el Big Bang, si la edad del Universo, según los científicos, es de 13.8×10^9 años.

2. Calcula aproximadamente cuántos nucleones (protón o neutrón) hay en una persona con una masa de 50 kg, si cada nucleón tiene una masa de 1.67×10^{-27} kg. Desprecia la masa de los electrones, unas $1,800$ veces menor.

3. Si $x = 27$, $y = 35$, y $z = 48$, ¿cuánto es $\frac{8x^2y^5}{3z^7}$?

2.3. Módulos

2.3.1. Importación de módulos

Hay funciones y operaciones especiales que están guardadas en módulos. El `Python` básico tiene instalados unos módulos por default, y hay otros muchos más que vienen con la versión de `Anaconda`. Si se requirieran de otros, el usuario tiene la libertad de instalar más. En `Anaconda`, la forma usual es ir a la terminal, `Anaconda prompt`, y teclear `conda install nombre_módulo`, aunque, hay siempre excepciones y en esos casos es mejor buscar en Google la manera correcta de instalar el módulo.

La forma simple de llamar a un módulo es con `import nombre_módulo`. Para usar una función, una constante, o un submódulo dentro del módulo, hay que anteponer el nombre de éste. Sin embargo, hay veces que las funciones dentro del módulo son empleadas frecuentemente, y es preferible abreviar el nombre de éste. Para los módulos más usados hay ya convenciones establecidas, por ejemplo, el módulo `numpy`, que sirve para operaciones matriciales y vectores en general, se abrevia `np`. Otro módulo muy utilizado es el `matplotlib.pyplot` para graficar, que se abrevia como `plt`. La forma más general de llamar a un módulo es entonces, `import nombre_módulo as alias`.

Por ejemplo, el número `e` está dentro del módulo `math`:

```
1 In [1]:
2 import math
3 print(math.e)
4 Out[1]:
5 2.718281828459045
```

O podemos hacer una suma de vectores con `numpy`:

```
1 In [2]:
2 import numpy as np
3 print(np.array([3,4,5]) + np.array([8,7,2]))
4 Out[2]:
5 [11 11  7]
```

Cuando se requiere de una función algo o muy especializada, de un módulo que en general tiene varios submódulos con muchas funciones, es mejor bajar sólo esa función usando `from`. Por ejemplo, del módulo `sklearn` para Machine Learning, que está lleno de submódulos y funciones, podemos hacer

```
1 In [3]:
2 from sklearn.model_selection import train_test_split
```

y así, en lugar de escribir `sklearn.model_selection.train_test_split()`, sólo usamos la función `train_test_split()`, o un alias adecuado. Otro ejemplo, podemos importar el valor de π del módulo math, y ya no sería necesario escribir `math.pi`.

```
1 In [4]:
2 from math import pi
3 print(pi/3)
4 Out[4]:
5 1.0471975511965976
```

2.3.2. El módulo math

Para acceder a las funciones más usuales y básicas, es suficiente importar el módulo `math` con `import math`. Para ver el contenido de `math`, hacemos `dir(math)`

```
In [1]:
import math
print(dir(math))
Out[1]:
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

En cada módulo, las funciones que empiezan con guión bajo son métodos internos. Las funciones que vamos a utilizar son el resto, y en lenguaje de programación a estas funciones se les llama atributos o métodos. En general, los atributos son constantes o descripciones del objeto con el que estamos trabajando y no llevan paréntesis, mientras que los métodos son considerados como acciones, y llevan paréntesis, tal cual como trabajamos con funciones en matemáticas. Por ejemplo:

```
In [2]:
print(math.sin(math.pi/2))
Out[2]:
1.0
```

El `pi` en `math.pi` no lleva paréntesis, mientras que la función seno sí, `math.sin()`.

Ejercicios:

1. Encuentra el ángulo, en radianes y grados, cuya tangente es igual a 1.2

2.3.3. El módulo random

En varios ejemplos de los capítulos siguientes, vamos a generar números aleatorios con el módulo `random`. Veamos que hay en él:

```
In [1]:
import random
print(dir(random))
Out[1]:
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_ONE', '_Sequence', '_Set', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '_accumulate', '_acos', '_bisect', '_ceil', '_cos', '_e', '_exp', '_floor', '_index', '_inst', '_isfinite', '_log', '_os', '_pi', '_random', '_repeat', '_sha512', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn', 'betavariate', 'choice', 'choices', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randbytes', 'randint', 'random
```

```
', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', '
triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
```

Para ver para qué sirven algunas de estas funciones, podemos usar la función `help()`, ya sea para todo el módulo, o nada más para la función en la que estamos interesados, sin los paréntesis de la función al final. Por ejemplo:

```
1  In [2]:
2  print(help(random.sample))
3  Out[2]:
4  Help on method sample in module random:
5
6  sample(population, k) method of random.Random instance
7      Chooses k unique random elements from a population sequence or set
   .
8      Returns a new list containing elements from the population while
   leaving the original population unchanged.  The resulting list is
   in selection order so that all sub-slices will also be valid random
    samples.  This allows raffle winners (the sample) to be
   partitioned into grand prize and second place winners (the
   subslices).
9      Members of the population need not be hashable or unique.  If the
   population contains repeats, then each occurrence is a possible
   selection in the sample.
10     To choose a sample in a range of integers, use range as an
   argument.
11     This is especially fast and space efficient for sampling from a
   large population:   sample(range(10000000), 60)
```

Los métodos que más suelen usarse son `seed`, que es para poder reproducir nuestros cálculos en cada corrida del programa, `random`, `randint`, `randrange`, `uniform`, `gauss`, `sample`, `shuffle`, entre otros.

Ejercicios:

1. Checa las diferencias entre `random.random`, `random.randint` y `random.randrange`, y selecciona la que más te guste o la que te sirva para seleccionar un número aleatorio entre el 1 y el 100.

2. Escoge el número 7 como semilla, `random.seed(7)`, y después selecciona un número de entre la serie $(7, 11, 15)$, `random.choice((7,11,15))`. Corre el programa varias veces, ¿qué sucede?

2.4. Cadenas (strings)

Después de los números, el siguiente tipo de variable es la string, `str`, o cadena. Se define usando comilla doble (`"algo"`) o simple (`'algo'`), y, si es la cadena vacía, se puede inicializar con `s=""` o `s=str()`. En Python, la cadena es inmutable, lo que significa que los métodos que utilicemos no van a modificarla, hay que redefinirla para lograr un cambio permanente. Para ver sus métodos, al igual que con el contenido de los módulos, usamos `dir()`:

```
In [1]:
s="Hola"
print(dir(s))
Out[1]:
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Por ejemplo, si queremos que `s` esté en mayúsculas, usamos el método `upper()`

```
In [2]:
print(s.upper())
Out[2]:
'HOLA'
```

Sin embargo, cuando llamamos a `s` de nuevo, vemos que no se ha modificado

```
In [3]:
print(s)
Out[3]:
'Hola'
```

hay que redefinir a `s` para hacer un cambio permanente

```
In [4]:
s=s.upper()
print(s)
Out[4]:
'HOLA'
```

De nuevo, con `type()` vemos el tipo de variable, `str`

```
In [5]:
print(type(s))
Out[5]:
str
```

e igual que con los módulos, usamos `help(str.método)`, para ver la información sobre el método:

```
In [6]:
print(help(s.split))
Out[6]:
Help on built-in function split:
```

```
split(...) method of builtins.str instance
    S.split(sep=None, maxsplit=-1) -> list of strings

    Return a list of the words in S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
```

Ejercicios:

Explora los métodos de las cadenas y responde las siguientes preguntas.

Dada la cadena

`s = "tres tristes tigres tragaban trigo, en tres tristes trastos \`
` sentados en un trigal"`

- cuenta el número de letras `"t"` en la cadena
- encuentra el índice de la sub-cadena "traga"
- reemplaza los espacios sencillos en blanco por tres guiones, `"---"`

2.4.1. Concatenación, slicing y len()

Para unir dos o más cadenas, usamos el `+`:

```
In [1]:
s="Hola soy ficha roja"
t=", y yo soy la ficha azul"
print(s+t)
Out[1]:
'Hola soy ficha roja, y yo soy la ficha azul'
```

Para obtener un fragmento de la cadena, se selecciona con corchetes dando el intervalo de índices que queremos. En `Python`, los índices se empiezan a contar desde cero, y en un intervalo el límite inferior es cerrado, y el superior es abierto, `[a,b)`. Por ejemplo, `s[3:11]` se refiere a la sub-cadena que empieza desde el índice 3, que es el cuarto elemento (0-1-2-3), hasta el índice 10, que es el 11-avo elemento:

```
In [2]:
print(s[3:11])
Out[2]:
'a soy fi'
```

Para saber la longitud de la cadena, existe una función que se usa en general para cualquier iterable en `Python`, `len()`. Pero no es necesario saber la longitud de la cadena si queremos imprimir un pedazo de la cadena que abarque hasta el final, se puede omitir el límite superior en este caso:

```
In [3]:
print(s[7: len(s)])
print(s[7:])
Out[3]:
```

```
y ficha roja
y ficha roja
```

Podemos también contar para atrás, nada más que en este caso la numeración empieza desde -1, que es equivalente a `len(cadena)-1`. En general, el índice `-i` equivale a `len(cadena)-i`,

```
In [4]:
print(s[-7:-2])
Out[4]:
'ha ro'
```

Finalmente, podemos multiplicar a una cadena:

```
In [5]:
print("ra" + 12*"ta" + ", mira, ra" + 12*"ta"+ ", baila, ra" +\
26*"ta" + ", baila rock and roll")
Out[5]:
'ratatatatatatatatatatatata, mira, ratatatatatatatatatatatata, baila,
   ratatatatatatatatatatatatatatatatatatatatatatatatatata, baila rock
   and roll'
```

Ejercicios:

1. Dada la cadena `s=''ferrocarrilero''`, a) selecciona las primeras tres letras, b) selecciona las últimas cuatro letras, c) encuentra cuántas letras tiene la cadena.

2.4.2. Imprimir el resultado de una operación como cadena

¿Cómo hacemos cuando queremos imprimir el resultado de un cálculo, junto con algún contexto en forma de cadena? Hay varias formas de hacerlo. Primero, hay que decir que no hay ningún problema en escribir números como cadenas, por ejemplo `''5''` es la representación en cadena del número 5. Lo que queremos es combinar los resultados de operaciones que sean numéricas con cadenas.

Podemos, como ya hemos visto, imprimir separando con comas las cadenas, como siempre con comillas, y los números sin ningún formato en especial:

```
In [1]:
print("hola, yo soy el resultado de 2 + 5:", 2+5)
Out[1]:
hola, yo soy el resultado de 2 + 5: 7
```

```
In [2]:
# redondear un resultado a dos cifras:
x, y= 35.87, 24.32
print("z = xy:", round(x*y,2))
Out[2]:
z = xy: 872.36
```

Podemos también convertir explícitamente a los números en cadenas con la función `str()` y concatenar. En este caso, hay que tener cuidado de dejar un espacio en blanco:

```
In [3]:
print("a los " + str(2*3) + " años se enamoró, luego a los "+str(6+1)\
+ ", pues se casó, lo que tenía que pasar pasó, a los " +str(4*2) + \
" años papá resultó")
Out[3]:
a los 6 años se enamoró, luego a los 7, pues se casó, lo que tenía que
    pasar pasó, a los 8 años papá resultó
```

Podemos hacer uso también del método `format` de las cadenas. En esencia, lo que queramos que aparezca en la cadena se encierra entre llaves, y hay opciones para obtener ciertos formatos, como alinear a la izquierda, derecha, centrado, ocupar un cierto número de caracteres como cuando se llena una tabla, entre otras.

```
In [4]:
import random
print("tengo {} tomates, {} manzanas y {} \
calabazas".format(random.randint(1,10), random.randrange(1,4),\
 random.choice((2,7,8))))
Out[4]:
'tengo 2 tomates, 2 manzanas y 7 calabazas'
```

Para saber más de `format`, puedes ir a la página https://pyformat.info/, pues hay mucho más sobre ello.

2.5. Listas

Las listas son el tipo de bases de datos más básico y utilizado en `Python`. Pueden tener cualquier tamaño y son mutables, lo que significa que, si aplicamos un método, la lista se modifica permanentemente. Los elementos de las listas no tienen que ser necesariamente del mismo tipo, se pueden mezclar números, cadenas, listas dentro de la lista, etc. Se pueden definir usando corchetes o con la función `list()`

```
In [1]:
A = []
B = list()
```

Los métodos disponibles:

```
In [2]:
print(dir(A))
Out[2]:
['__add__', '__class__', '__class_getitem__', '__contains__', '
    __delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '
    __format__', '__ge__', '__getattribute__', '__getitem__', '
    __getstate__', '__gt__', '__hash__', '__iadd__', '__imul__', '
    __init__', '__init_subclass__', '__iter__', '__le__', '__len__', '
    __lt__', '__mul__', '__ne__', '__new__', '__reduce__', '
    __reduce_ex__', '__repr__', '__reversed__', '__rmul__', '
    __setattr__', '__setitem__', '__sizeof__', '__str__', '
    __subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', '
    index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Por ejemplo, podemos agregar un elemento a la vez con `append()`

```
In [3]:
A.append(5)
print(A)
Out[3]:
[5]
```

o extender la lista con elementos de otra,

```
In [4]:
A.extend([7,11,3,4,9])
print(A)
Out[4]:
[5, 7, 11, 3, 4, 9]
```

que también se puede hacer con el símbolo +,

```
In [5]:
B = [2,8,6]
A = A + B
print(A)
Out[5]:
[5, 7, 11, 3, 4, 9, 2, 8, 6]
```

Podemos ordenarla,

```
In [6]:
A.sort()
print(A)
Out[6]:
[2, 3, 4, 5, 6, 7, 8, 9, 11]
```

eliminar un elemento dando su índice,

```
In [7]:
A.pop(2)    # debe eliminar el tercer elemento (0-1-2)
print(A)
Out[7]:
[2, 3, 5, 6, 7, 8, 9, 11]
```

eliminar un elemento por su valor,

```
In [8]:
A.remove(6)
print(A)
Out[8]:
[2, 3, 5, 7, 8, 9, 11]
```

o agregar un elemento en un índice dado, entre otras posibilidades.

```
In [9]:
A.insert(0,13)    # A.insert(index,element)
print(A)
Out[9]:
[13, 2, 3, 5, 7, 8, 9, 11]
```

Finalmente, de la misma forma que en las cadenas, además de índices positivos podemos acceder también a algún elemento empezando a contar desde -1 desde el final de la lista:

```
In [10]:
print(A[7], A[5], A[-1], A[-3])
Out[10]:
11 8 11 8
```

Puedes investigar más de los métodos de las listas con `help`.

Ejercicios:

Dado la lista

`A = [11, 7, 5, 3, 9, 17, 8, 4, 2, 12, 13]`

agrega el número 27 a la lista

agrega los elementos `[16,20,31]` a la lista

borra el número 8 de la lista

borra el elemento con índice 5 de la lista

inserta de nuevo el número 8 en el índice 2

encuentra el índice que tiene el número 12 en la lista

ordena la lista

encuentra la longitud total de la lista

2.5.1. Copia de una lista

Para hacer una copia de una lista, no basta con definir otra lista e igualarlas, pues harán referencia al mismo objeto:

```
In [1]:
A = [3,5,4]
B = A
A.append(7)
print("A:",A)
print("B:",B)
Out[1]:
A: [3, 5, 4, 7]
B: [3, 5, 4, 7]
```

Lo que le hicimos a la lista `A` afectó a la lista `B` al referirse al mismo objeto. Hay al menos tres formas de hacer una buena copia de una lista simple:

```
In [2]:
A = [3,5,4]
B = A[:]          # primera forma
C = A.copy()      # segunda forma
D = list(A)       # tercera forma
A.append(7)       # modificamos A
A[1]=8
print("A:",A)
print("B:",B)
print("C:",C)
```

```
print("D:",D)
Out[2]:
A: [3, 8, 4, 7]
B: [3, 5, 4]
C: [3, 5, 4]
D: [3, 5, 4]
```

Modificamos `A` y no repercutió en las copias de nuestro ejemplo. Sin embargo, esto no funciona del todo para listas de listas (listas anidadas):

```
In [3]:
A = [[3,5],4,[8,6]]
B = A[:]
C = A.copy()
D = list(A)
C[0] = [48,49]    # cambiamos una entrada de la matriz C
B[2].append(7)  # agregamos un elemento a una entrada de la matriz B
print("A:",A)
print("B:",B)
print("C:",C)
print("D:",D)
Out[3]:
A: [[3, 5], 4, [8, 6, 7]]
B: [[3, 5], 4, [8, 6, 7]]
C: [[48, 49], 4, [8, 6, 7]]
D: [[3, 5], 4, [8, 6, 7]]
```

Vemos que asignar un valor diferente a alguna entrada no modifica el resto de las listas, pero el modificar una sublista sí se replica en el resto, así que hay que tener otra forma de hacer copias. Para ello está la función `deepcopy` del módulo `copy`:

```
In [4]:
from copy import deepcopy
A = [[3,5],4,[8,6]]
B = deepcopy(A)    # copiamos
A[0] = [48,49]     # cambiamos una entrada de una matriz
B[2].append(7)     # agregamos un elemento a una entrada de una matriz
print("A:",A)
print("B:",B)
Out[4]:
A: [[48, 49], 4, [8, 6]]
B: [[3, 5], 4, [8, 6, 7]]
```

Ahora sí, lo que hicimos en una lista no afectó a la otra.

2.6. Booleanos

El siguiente tipo de variable es el booleano, `bool`, que admite dos valores, `True` y `False`. Las operaciones con booleanos son `and`, `or` y `not`. Las tablas de verdad son muy sencillas. La operación `and` con dos afirmaciones sólo es verdadera cuando las dos afirmaciones son ciertas, mientras que el resultado con el operador `or` sólo es falso cuando las dos afirmaciones son falsas. El operador `not`, por su parte, cambia el sentido de una afirmación. Las operaciones se muestran en la tabla 2.1:

P1	P2	P1 and P2	P1 or P2
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

P	not P
True	False
False	True

Tabla 2.1: Tablas de verdad

Algunos ejemplos en `Python` son:

```
In [1]:
print(True and False, True and not False, False or not True)
Out[1]:
False True False
```

Obtenemos variables booleanas, por ejemplo, cuando comparamos objetos. Las comparaciones posibles son `>`, `<`, `>=`, `<=`, `==`, `!=` (diferente a):

```
In [2]:
print(5>4, 5>=4, 5<4, 5<=4, 5==4, 5!=4)
print(type(5==2))
Out[2]:
True True False False False True
Bool
```

Otra forma de obtener una variable booleana es con el operador `in`, si queremos saber por ejemplo si un objeto se encuentra dentro de una iterable:

```
In [3]:
print("orro" in "cotorro")
Out[3]:
True
```

Ejercicios:

¿Cuáles son los resultados de las siguientes comparaciones? Inténtalo primero con lápiz y papel, y después en `Python`:

a)`("María">"Victoria") or ((3**3<2**5) and (17%5=="almeja".count("a")))`

b) la longitud de la palabra "Aguascalientes" es igual que la división de 234 entre 17.

c) `(not "Hola!".isalnum()) and "pedro".replace("d","r")=="perro"`

2.7. Condicionales

Los condicionales son enunciados que nos permiten arrojar un resultado dependiendo de la condición dada. En español sería, SI cond1, haz esto, O SI cond2, haz aquello, etc., hasta llegar a DE OTRA FORMA, haz esto otro.

En `Python`, se comienza con `if` para la primera condición, el número necesario de `elif` (puede no haber) para otras posibles condiciones, y, finalmente (no obligatorio), el `else` para el resto de posibilidades no comprendidas en el `if` y `elif`. La notación

consiste en escribir la condición seguida de dos puntos (:), y en el siguiente renglón, con indentación, lo que se va a ejecutar dada la condición. Hay que subrayar que, en `Python`, la indentación es muy importante. Después de los dos puntos hay una indentación por default de cuatro espacios, o un tab, en el siguiente renglón, que se aplica solito en el editor. No necesariamente debe ser ese número de espacios, aunque una vez seleccionado el espaciamiento, el resto del cuerpo debe quedar alineado de manera acorde, los `if`, `elif` y `else` a la misma altura, y las diferentes acciones a la misma altura entre ellas:

```
In [1]:
import random
# seleccionamos un entero entre el 1 y el 100
a = random.randint(1,100)
if a%3 == 0:
    print(a,"es divisible entre 3")
elif a%3==1:
    print("al dividir",a,"entre 3, el residuo es 1")
else:
    print("el residuo de la división de",a,"entre 3, es 2")
Out[1]:
57 es divisible entre 3
```

Ejercicios:

En una cadena escribe tu nombre (uno solo). Si la longitud de tu nombre es mayor a 7, imprime "tu nombre es algo largo", más tu nombre; si es menor que 7, imprime "tu nombre es algo corto", más tu nombre, y si es igual a 7, imprime "tu nombre tiene la justa medida", más tu nombre.

2.8. Bucles for (for loops)

Los bucles `for` sirven para iterar una acción un determinado número de veces. En `Python`, tenemos diferentes tipos de `for`, la diferencia entre ellos es si la iteración se da en un rango o intervalo de números, o si se itera por elemento de una secuencia. La notación, como en el caso de los condicionales, consiste en escribir el enunciado del `for` seguida de dos puntos, y la acción o acciones a llevarse a cabo en el siguiente o siguientes renglones de manera indentada.

2.8.1. for i in range(a,b)

El `for` más básico es un bucle que itera por índice en un intervalo. Se puede usar en dos sentidos, cuando el índice es importante y se usa explícitamente en el cuerpo de instrucciones, o cuando simplemente se pide repetir una acción un determinado número de veces. En `for i in range(a,b)`, el índice `i` puede tener cualquier nombre, es una variable interna que se usa en las instrucciones, y en el `range`, se puede dar el inicio de intervalo, se da el final, y se puede dar como tercer argumento el paso. Los números deben ser enteros. También puede darse nada más el final del intervalo, si se comienza a contar desde cero. Cuando sólo se usa el `for` para repetir una acción un determinado número de veces, se suele usar un guión bajo, `_` , sólo por convención.

```
In [1]:
# range(inicio, final, paso), i es el índice:
for i in range(2,11,3):
    print("soy el número",i,"y mi cuadrado es", i**2)
Out[1]:
soy el número 2 y mi cuadrado es 4
soy el número 5 y mi cuadrado es 25
soy el número 8 y mi cuadrado es 64
```

```
In [2]:
print(6*"ma"+"gui quero")
# range(sólo final), no hay necesidad de un i explícito, se puede usar
# el guión bajo, por convención, pero no es obligatorio!:
for _ in range(2):
    print("mamagui quero")
print("comer")
Out[2]:
mamamamamamamagui quero
mamagui quero
mamagui quero
comer
```

Hagamos un ejemplo de una iteración, donde encontramos un valor después de un número determinado de pasos. Por ejemplo, usemos la fórmula de Leibnitz para encontrar el valor de π después de 10 pasos. La fórmula es:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

Entonces:

```
In [3]:
k=0    # k es la aproximación a π (multiplicada por 4)
for n in range(0,10):
    k= k+ (-1)**n/(2*n+1)
    # para imprimir en cada iteración la aproximación a π:
    print(n,4*k)
Out[3]:
0 4.0
1 2.666666666666667
2 3.466666666666667
3 2.8952380952380956
4 3.3396825396825403
5 2.9760461760461765
6 3.2837384837384844
7 3.017071817071818
8 3.2523659347188767
9 3.0418396189294032
```

Ejercicios

1. En el ejemplo de la aproximación a π, ¿qué pasaría si la asignación $k = 0$ se hiciera dentro del ciclo `for`?, ¿qué pasa si no se asigna un valor a k de inicio?, ¿dónde hay que colocar el `print` si sólo se quiere imprimir el resultado final (último término de la

iteración)? ¿Se puede imprimir la variable n definida para el `range` fuera del bucle `for`? Si la fórmula no fuera una suma de términos sino un producto, iterativo, ¿cuál debe ser la asignación inicial de k?

2.8.2. for i in A

El otro tipo básico de `for` es el que itera por elemento en una secuencia. De igual forma, en `for i in A`, el elemento `i` puede tener cualquier nombre.

```
In [1]:
A = ["elefante", "león", "tarántula", "escorpión", "rata", "gato", \
  "perro", "burro", "tigre"]
t="tiene un nombre largo!"
for j in A:                # j se refiere a cada palabra de A
    if len(j)>6:
        if j[-1]=="a":     # si la palabra termina en a
            s="la"
        else:              # de otra forma
            s="el"
        print(s,j,t)
Out[1]:
el elefante tiene un nombre largo!
la tarántula tiene un nombre largo!
el escorpión tiene un nombre largo!
```

En este ejemplo el programa sólo imprime algo cuando el tamaño de la palabra es mayor a seis letras, y después, usando otro conjunto de condicionales, se fija si se debe anteponer el artículo femenino o masculino a la palabra dependiendo de si la palabra termina en “a” o no.

Hagamos un ejemplo con dos ciclos `for`, para ver lo de la indentación:

```
In [2]:
for i in ["tomate", "manzana", "pimiento", "chile"]:
    for j in ["rojo", "verde"]:
        s=i+" "+j
        if len(s)%5==0:  # si la longitud de s es múltiplo de 5
            print(s)
Out[2]:
chile rojo
```

Ejercicios:

Dada la siguiente lista de animales:

```
A = ["canguro", "serpiente", "león", "jaguar", "caballo", "puma",\
 "tigre", "venado", "alce", "reno", "gato", "perro", "ardilla",\
 "mapache", "zorro", "iguana", "rana", "lobo", "lémur", "leopardo",\
 "zopilote", "mariposa", "ballena", "tiburón", "piraña", "coyote",\
 "armadillo", "tortuga"]
```

* ordena la lista en orden alfabético
* encuentra la palabra más larga, rompe empates arbitrariamente
* para cada palabra, encuentra su número de vocales y consonantes

2.8.3. for i,j in enumerate(A)

Un tercer tipo de for que se suele utilizar, es el que usa ambos, índice y elemento de una secuencia, llamando a `enumerate`, y que en realidad se podría sustituir por un `for` con `range`:

```
In [1]:
for ind, i in enumerate("tortilla"):
    if i in "aeiou":
        print("en el índice",ind,"hay una vocal, la",i)
Out[1]:
en el índice 1 hay una vocal, la o
en el índice 4 hay una vocal, la i
en el índice 7 hay una vocal, la a
```

Ejercicios:

1. Imprime el resultado del ejemplo resuelto usando un `for` con `range`.

2. Dada la siguiente lista:

```
A= ["Beethoven", "Mozart", "Tchaikovsky", "Saint-Saenz", "Wagner",\
"Verdi", "Bizet", "Vivaldi", "Rossini", "Schubert", "Grieg", "Bach",\
"Ravel", "Mahler", "Berliotz", "Mussorgsky", "Dvorak", "Borodin",\
"Gershwin", "Williams", "Moncayo", "Offenbach", "Strauss"]
```

ordena la lista alfabéticamente e imprime sus elementos enumerándolos, comenzando la cuenta desde el 1. (Hint: la función `enumerate()` puede tener como segundo argumento, además del iterable a enumerar, el número de inicio de la numeración. Como alternativa también podrías mandar a imprimir no el índice, que por default empieza desde cero, sino una variación a éste).

2.8.4. for i,j in zip(A,B)

Finalmente, otro tipo de `for` es el que usa dos o más iterables, y que combina elementos de esas secuencias con el mismo índice. Ahora se llama a `zip` e, igualmente, se podría sustituir este for por un `for` con `range()`:

```
In [1]:
A= ["camello", "león", "delfín", "ratón", "elefante", "caballo", \"
    lobo"]
B = [40, 14, 30, 2, 65, 30, 8]
for i, j in zip(A,B):
    print("el", i, "tiene un promedio de vida de", j, "años")
Out[1]:
el camello tiene un promedio de vida de 40 años
el león tiene un promedio de vida de 14 años
el delfín tiene un promedio de vida de 30 años
el ratón tiene un promedio de vida de 2 años
el elefante tiene un promedio de vida de 65 años
el caballo tiene un promedio de vida de 30 años
el lobo tiene un promedio de vida de 8 años
```

Ejercicios:

1. Dadas las siguientes listas:

```
comp= ["Beethoven", "Mozart", "Tchaikovsky", "Saint-Saenz", "Wagner",
    "Verdi"]
año= [1770, 1756, 1840, 1835, 1813, 1813]
país=["Alemania","Austria","Rusia","Francia","Alemania","Italia"]
pieza= ["Sinfonía no. 5", "Concierto para piano no. 21",\
"Obertura 1812", "Sinfonía no. 3 (con órgano)","Tannhauser",\
"El trovador"]
```

manda a imprimir, por cada compositor, un enunciado del tipo, *fulanito* nació en *año* en *país* y compuso *pieza*.

2.8.5. Llenar una lista, lista por comprensión

Ahora que ya sabemos usar el `for`, vamos a ver cómo llenar una lista. La forma tradicional es usar el `for` y el `append`:

```
In [1]:
A=[]
for i in range(10):
    A.append(4*i+3)
print(A)
Out[1]:
[3, 7, 11, 15, 19, 23, 27, 31, 35, 39]
```

pero hay una forma más compacta y rápida que se conoce como list comprehension, lista por comprensión, donde los corchetes resguardan al elemento de la lista seguido del `for`:

```
In [2]:
B = [4*i+3 for i in range(10)]
print(B)
Out[2]:
[3, 7, 11, 15, 19, 23, 27, 31, 35, 39]
```

Más ejercicios resueltos

1. Encuentra tu nombre en una cadena generada al azar

Con las letras que conforman tu nombre, forma una cadena lo suficientemente larga donde los caracteres aparezcan aleatoriamente. El problema consiste en encontrar el o los índices en la cadena donde aparece tu nombre. Para saber de qué tamaño debe ser la cadena, considera lo siguiente. Si las letras de tu nombre no se repiten y hay n de ellas, entonces la probabilidad de reproducir tu nombre en un segmento de tamaño n es $1/n^n$. Por ejemplo, para el nombre `"erika"` que tiene cinco letras, la probabilidad de que salga `"e"` en el primer lugar es $1/5$, la probabilidad de que salga `"r"` en el segundo lugar es $1/5$, etc., así que con $5^5 = 3,125$, la probabilidad es 1 en $3,125$ de que se forme `"erika"` aleatoriamente con cinco letras seguidas, por lo que una cadena de $20,000$ caracteres es suficiente para ver `"erika"` al menos una vez. Si tienes un nombre muy largo, es mejor que uses un alias para evitar formar una cadena demasiado larga. Vamos

entonces a utilizar el método `choice()` del módulo `random` para agregar uno a uno los símbolos para conformar la cadena larga:

```
In [1]:
import random
random.seed(5)
# escogemos un número como semilla, seed, en caso de que queramos
# reproducir nuestro resultado
s=""; N= 20000
# concatenamos con + para formar la cadena
for i in range(N):
    s=s+random.choice("erika")
```

Respuesta.

Recorremos la cadena `s` desde el índice 0, para buscar el nombre, pero paramos algo antes del final, porque el nombre tiene un cierto tamaño, en nuestro ejemplo, 5. Haciendo slicing en la cadena, comparamos esa sub-cadena con el nombre. En caso de éxito, imprimimos el índice:

```
In [2]:
for i in range(N-5+1):       # paramos antes del final
    if s[i:i+5]=="erika":    # si la subcadena es igual al nombre
        print(i)             # imprime el índice
Out[2]:
12558
19004
```

2. Primeros términos de la serie de Fibonacci

La serie de Fibonacci es muy conocida y, en resumen, cuenta conejos. El término n-ésimo es la suma de los 2 términos previos, $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$ El problema a resolver consiste en encontrar los primeros 20 términos.

Respuesta.

Como nada más hay que imprimir los números y no necesitamos guardarlos para un uso posterior, vamos a generar los números al vuelo con un `for`. Vamos a necesitar de 3 números en cada iteración, los dos últimos números de la serie que son los sumandos, y la suma que es el nuevo número en la serie. Llamemos a al primer sumando, b al segundo, y c la suma. En la siguiente iteración, el segundo número, b, se convierte en el primer número a, y lo que obtuvimos como suma, c, se convierte en nuestro segundo sumando:

```
In [1]:
a = 0; b = 1      # primeros números en la serie
for i in range(20):
    print(i,a)    # en qué iteración estamos y el n-ésimo término
    c= a+b        # suma
    a= b          # actualiza a
    b= c          # actualiza b
Out[1]:
0 0
1 1
2 1
3 2
```

```
4 3
5 5
6 8
7 13
8 21
9 34
10 55
11 89
12 144
13 233
14 377
15 610
16 987
17 1597
18 2584
19 4181
```

3. Encuentra el mínimo y el máximo de una lista de números, y sus posiciones

Como dice el enunciado, vamos a encontrar el máximo y el mínimo de una lista aleatoria de números, y el índice que ocupan en la lista. La lista es de tamaño $n = 100$ que tomamos aleatoriamente del conjunto de números enteros del 1 al $1,000$.

```
In [1]:
import random
random.seed(5)
# una de las formas para generar 100 números entre el 1 y el 1,000,
# con posibilidad de repetirse:
A = [random.randint(1,1000) for _ in range(100)]
```

Respuesta

Para encontrar el máximo y el mínimo de una iterable, `Python` tiene las funciones `max()` y `min()`. Sin embargo, el problema nos pide encontrar dónde se localizan estos números en la lista `A`. Chequemos como quiera de una vez cuáles son el máximo y el mínimo:

```
In [1]:
print(max(A), min(A))
Out[1]:
(996, 2)
```

Ahora sí, para resolver este tipo de problema, la forma usual es recorrer la lista comparando cada nuevo elemento con el máximo y mínimo que encontramos en los pasos previos. Para ello, definamos un mínimo y un máximo de inicio, y, cada que encontremos un número más pequeño que nuestro mínimo, o un número más grande que nuestro máximo, los actualizamos, así como los índices respectivos. Empezamos dando un número grande al mínimo, fácil de vencer a la primera, y un número pequeño al máximo:

```
In [2]:
maximus, minimus = 0, 200000
indmax, indmin = 0, 0
for ind, i in enumerate(A):
    if i > maximus:    # si i es mayor al valor de maximus actual
        maximus = i    # actualizamos maximus y actualizamos
```

```
7          indmax = ind    # el índice donde se encuentra maximus
8      if i < minimus:     # hacemos lo mismo con minimus
9          minimus = i
10         indmin = ind
11 print("el número más grande es",maximus,\
12       "y está localizado en la posición"+ str(indmax) + ",")
13 print("y el número más pequeño es",minimus,\
14       "y está localizado en la posición", indmin)
15 Out[2]:
16 el número más grande es 996 y está localizado en la posición 53,
17 y el número más pequeño es 2 y está localizado en la posición 52
```

2.9. Bucles while (while loops)

El bucle `while` nos sirve para repetir una acción mientras una condición se siga. Su acción es parecida al bucle `for` cuando hay un contador, pero cuando hay una condición que no sabemos cuándo termina requerimos del `while` necesariamente. En español, escribiríamos MIENTRAS pasa condición, haz lo siguiente. El `while` es algo peligroso si la condición siempre se cumple, podríamos quedar en un loop infinito. Como en los casos anteriores, la notación es la de 2 puntos e indentación:

while condición:

haz algo.

Por ejemplo:

```
1 In [1]:
2 # while con contador:
3 i=1
4 while i!=5:
5     print(i,i**3)
6     i+= 1          # equivalente a i=i+1
7 Out[1]:
8 1 1
9 2 8
10 3 27
11 4 64
```

En este ejemplo tenemos un contador `i`, que empieza en 1 y se incrementa por uno en cada iteración. Cuando i es igual a 5 la condición del `while` ya no se cumple, no se ejecuta la acción y salimos del bucle. Si hubiéramos olvidado incrementar `i` en cada iteración con `i+=1`, equivalente a `i=i+1`, `i` valdría siempre 1 y la consola imprimiría el primer renglón por siempre. Afortunadamente, la terminal tiene un botón cuadrado en la parte superior derecha para detener la corrida del programa en caso de ser necesario, y también hay una cruz roja para matar esa terminal y activar una nueva.

Hay otra forma de salir de un bucle, que es con el `break`, que aplica también para el `for loop`. Se escribe entonces un `while True`, que se cumple por siempre, la condición se escribe dentro del bucle, y se rompe con el `break`:

```
1 In [2]:
2 import random
```

```
3 while True:
4     a = random.randint(1,10)
5     print("yo tenía", a, "perritos")
6     if a==1:
7         break
8 Out[2]:
9 yo tenía 8 perritos
10 yo tenía 3 perritos
11 yo tenía 7 perritos
12 yo tenía 6 perritos
13 yo tenía 5 perritos
14 yo tenía 3 perritos
15 yo tenía 6 perritos
16 yo tenía 10 perritos
17 yo tenía 7 perritos
18 yo tenía 7 perritos
19 yo tenía 3 perritos
20 yo tenía 1 perritos
```

Ejercicios:

Dada la lista

```
In [1]:
import random
random.seed(3)
# una forma de tomar 150 números aleatorios del 0 al 499, sin repetir
A = random.sample(range(500),150)
```

guarda en otra lista los primeros 25 números impares y manda a imprimirla.

2.10. Funciones

Las funciones se escriben de manera muy sencilla en `Python`. La notación es

def nombrefunción(argumentos):

cuerpo de la función

return el resultado deseado

Como en los casos anteriores, la notación es la de 2 puntos y la indentación, preferentemente de 4 espacios (un tab). Dependiendo del objetivo de la función, puede existir el `return` o no, si es que se quiere devolver un valor.

Definamos por ejemplo una función que encuentre la suma de 2 números:

```
1 In [1]:
2 def suma(a,b):
3     return(a+b)      #los paréntesis son opcionales
4 print(suma(5,3))
5 Out[1]:
6 8
```

Si no queremos guardar el resultado de una función, tenemos la opción de sólo mandar a imprimir:

```
In [2]:
def suma2(a,b):
    print(a+b)
suma2(5,3)
Out[2]:
8
```

Como ya se mencionó, podemos no retornar nada, ni siquiera imprimir. Por ejemplo, si tenemos una lista y una función la va a actualizar, como la lista es mutable, sólo llamamos a la función para que actúe sobre la lista:

```
In [3]:
A = [3,4,5]
def escala(A,k):
    for i in range(len(A)):
        A[i]=A[i]*k
escala(A,3)
print(A)
Out[3]:
[9, 12, 15]
```

Pero si pones el return solito en la función, aunque no lo requiera, no hay error, como quiera termina el algoritmo.

Podemos también devolver más de un resultado, usando lo que se llama tupla y que mencionaremos más adelante, pero que básicamente consiste en guardar más de un objeto entre paréntesis. Por ejemplo, podemos definir una función que encuentre la suma y la resta de 2 números:

```
In [4]:
def sumaresta(a,b):
    return(a+b, a-b)
print(sumaresta(5,3))
Out[4]:
(8, 2)
```

Si queremos operar con estos resultados, es mejor guardarlos por separado. A eso se le llama "desempacar":

```
In [5]:
suma, resta= sumaresta(7,8)
print("ejercicio: la suma de dos números es",suma, "y la resta es",\
 resta, ", cuáles son los números?")
Out[5]:
ejercicio: la suma de dos números es 15 y la resta es -1 , cuáles son
   los números?
```

Podemos también definir la función con valores de argumentos que vengan por default. En ese caso y si no se quiere cambiar esos valores, no se requiere poner esos argumentos cuando se llama a la función. Pero si uno quiere cambiar el valor de alguna variable, se puede hacer, o poniendo el nombre de la variable y su valor en el argumento, o dando sólo su valor en el orden que le corresponde, si esto es posible.

Por ejemplo, digamos que tenemos la ecuación del gas ideal, $PV = nRT$, y queremos calcular el valor del volumen dado el número de moles, suponiendo que tenemos la presión y temperatura estándar de 1atm $= 1.013 \times 10^5$ Pa y $20°C = 293K$. Más aún, podemos usar las dos unidades usuales, las del SI, usando m^3 para el volumen y Pa para la presión, con la constante del gas ideal $R = 8.314$ J/(mol K), o la de litro-atmósfera, con $R = 0.08206$ l atm/(mol K). Podemos escoger por default las unidades de litro-atmósfera, y entonces nuestra ecuación queda:

```
In [6]:
def volumen(n,P=1,T=293,R=0.08206):
    return(n*R*T/P)
print("V en litros, n=10, a T y P estándar:")
print(round(volumen(10),3))
print("V en litros, n=10, T=27 C:")
print(round(volumen(10,T=300),3))
print("V en m^3, n=10, T=14 C, P=77,650 Pa (CdMex día nublado):")
print(round(volumen(10, T=287, P=77650, R=8.314),3))
# lo mismo de arriba, pero en litros, las variables se dan en orden
print("V en litros, n=10, P=0.767 atm, T=14 C (igual que arriba):")
print(round(volumen(10,0.767,287),3))
Out[6]:
V en litros, n=10, a T y P estándar:
240.436
V en litros, n=10, T=27 C:
246.18
V en m^3, n=10, T=14 C, P=77,650 Pa (CdMex día nublado):
0.307
V en litros, n=10, P=0.767 atm, T=14 C (igual que arriba):
307.056
```

Ejercicios:

1. Ecuación de segundo grado

Escribe una función que devuelva las soluciones de una ecuación de segundo grado, dando como argumentos los coeficientes reales de la variable al cuadrado, de la variable lineal, y la constante, en ese orden. En lugar de usar el módulo `math` para la raíz cuadrada, llama al módulo `cmath`, para manejar también soluciones imaginarias.

2. Productos entre vectores

a) Dados 2 vectores A y B del mismo tamaño, escribe una función que calcule su producto punto

b) Dados 2 vectores A y B en 3 dimensiones, escribe una función que calcule su producto cruz

2.10.1. Recursividad

En algunos casos, podemos tener una función que se llame a sí misma. A esto se le llama recursión. Como el número de llamadas no se puede hacer infinito, tenemos que tener uno o más casos base para los cuales salgamos de la función y retornemos un resultado. Como ejemplo, veamos el caso típico del factorial.

Sabemos que $n! = n(n-1)(n-2)\dots(1) = n(n-1)!$, y que $0! = 1$. Podemos definir una función recursiva de la siguiente forma:

```
In [1]:
def factorial(n):
    if n==0:
        return(1)
    return(n*factorial(n-1))
print(factorial(5))
Out[1]:
120
```

En este ejemplo, la función se llama así misma pero con el valor del argumento reducido en una unidad. Cuando finalmente $n = 0$, la función ya no se vuelve a llamar a sí misma, devuelve 1 y se termina el cálculo.

Ejercicios:

Suma de los primeros n números naturales. Define una función que encuentre la suma de los primeros n números naturales, con la fórmula que encontró Gauss cuando estaba chiquito. Escribe otra función que encuentre la suma recursivamente. Encuentra la suma de los primeros 250 números naturales usando las dos funciones.

2.11. Entradas externas

2.11.1. Inputs

Vamos a ver ahora cómo podemos leer información, ya sea desde la consola, o leyendo un archivo externo. Si queremos interactuar con un usuario, seguramente nosotros mismos, para que entre información sencilla desde la consola, se usa la función `input()`.

Puedes usar `input()` de dos formas: o llenas el argumento de la función con la pregunta que quieres que sea respondida, o, la pregunta la puedes imprimir desde antes y llamas a `input()` sin argumento. Por ejemplo:

```
a= input("¿Cómo te llamas? ")
print("gusto en conocerte", a)
```

vs

```
print("¿Cómo te llamas?")
a= input()
print("gusto en conocerte",a)
```

Lo que ingresa desde la terminal es una cadena, así que, si lo que pedimos es un número que va a ser parte del argumento de una función, tenemos que convertir la cadena a número con `int()` o con `float()`.

```
In [1]:
a= int(input("dame un número "))
dame un número 5
```

```
In [2]:
b= int(input("dame otro número "))
dame otro número 3
```

```
In [3]:
print("la suma de tus números es", a+b)
Out[3]:
la suma de tus números es 8
```

Ejercicios:

Escribe un archivo donde la computadora juegue con un usuario piedra, papel o tijeras. La computadora puede hacer su selección arbitrariamente usando `random.choice(secuencia)` del módulo `random`. Trata de convertir el input del usuario a mayúsculas o minúsculas, con el método `lower()` o el `upper()`, para que no importe cómo ingrese su respuesta el usuario. También, mientras éste dé una mala respuesta, dile que está tecleando mal y pregúntale de nuevo si ¿piedra, papel o tijeras?

2.11.2. Leer archivos

En lo siguiente vamos a ver cómo leer documentos de texto `.txt`. Si quisiéramos leer datos de archivos de excel, en `Python` hay módulos especiales para leer éstos y archivos `csv`, como el módulo `pandas`, que trata de manera eficiente las tablas de datos y que veremos más adelante. La forma general de abrir, trabajar y cerrar un archivo es:

`f = open("archivo.txt")`, se abre el archivo

..., se procesa

`f.close()`, se cierra el archivo.

En realidad, el comando completo para abrir el archivo es `f = open("arch", "r")`, `"r"` de `"read"`, que es el default y se puede omitir. También hay opción `"w"` para escribir borrando el contenido del archivo si ya existía, si no, crea un nuevo archivo; `"a"` para agregar algo a un archivo sin borrar su contenido, y `"r+"` para leer y escribir. Si los archivos de texto y de `Python` están en el mismo directorio, sólo pones el nombre del archivo de texto como en el ejemplo, de otra forma tendrías que dar todo el path al archivo.

Vamos a trabajar con la fábula de *El gato, el lagarto y el grillo*, de Tomás de Iriarte (Tenerife, 18 de septiembre de 1750- Madrid, 17 de septiembre de 1791), dividida en tres archivos. Tú puedes practicar con un archivo de texto que tenga el contenido de tu agrado.

Empecemos entonces primero viendo qué se puede hacer con el archivo con `dir()`, aunque, como ya sabemos llenar una lista con un `for`, vamos a imprimir sólo los métodos externos, que no empiezan con guión bajo:

```
In [1]:
f= open("el gato.txt")
B= [i for i in dir(f) if i[0]!="_"]  # si el método no empieza con _
f.close()
```

```
5 print(B)
6 Out[1]:
7 ['buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno'
    , 'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlines',
    'read', 'readable', 'readline', 'readlines', 'reconfigure', 'seek',
     'seekable', 'tell', 'truncate', 'writable', 'write', '
    write_through', 'writelines']
```

Vamos a quedarnos con unas cuantas funciones para leer el archivo. Aunque no venga en `dir()`, una de las formas más usuales para procesar el texto es línea por línea con un `for`. Adicionalmente, para desplegar de manera correcta los acentos vamos a usar el parámetro `encoding = ''utf8''`:

```
1 In [2]:
2 f= open("el gato.txt", encoding="utf8")
3 A= [line for line in f]
4 f.close()
5 print(A)
6 Out[2]:
7 ['Ello es que hay animales muy científicos \n', 'en curarse con varios
     específicos \n', 'y en conservar su construcción orgánica, \n', '
    como hábiles que son en la Botánica; \n', 'pues conocen las hierbas
     diuréticas,\n', 'catárticas, narcóticas, eméticas, \n', 'febrí
    fugas, estípticas, prolíficas, \n', 'cefálicas también y sudorí
    ficas. \n', 'En esto era gran práctico y teórico \n', 'un gato,
    pedantísimo retórico,\n', 'que hablaba en un estilo tan enfático \n
    ', 'como el más estirado catedrático. \n']
```

Vemos que se formó una lista de líneas, con cada línea terminando en un `"\n"` que indica salto de línea. Si revisamos el archivo de texto directamente, no hay nada como un ''\n'', éste nada más nos dice que la siguiente parte del texto se encuentra en la línea de abajo. Para quitar el ''\n'', usamos el método `strip()` que elimina los espacios en blanco y saltos de línea del principio y final de la cadena:

```
1 In [3]:
2 f=open("el gato.txt", encoding="utf8")
3 A=[line.strip() for line in f]
4 f.close()
5 print(A)
6 Out[3]:
7 ['Ello es que hay animales muy científicos', 'en curarse con varios
    específicos', 'y en conservar su construcción orgánica,', 'como há
    biles que son en la Botánica;', 'pues conocen las hierbas diuré
    ticas,', 'catárticas, narcóticas, eméticas,', 'febrífugas, estí
    pticas, prolíficas,', 'cefálicas también y sudoríficas.', 'En esto
    era gran práctico y teórico', 'un gato, pedantísimo retórico,', '
    que hablaba en un estilo tan enfático', 'como el más estirado
    catedrático.']
```

Siguiendo, vamos a leer ahora la siguiente parte de la fábula, toda de una vez, con `read()`

```
1 In [4]:
2 f= open("el lagarto.txt",encoding="utf8")
3 a= f.read()
4 f.close()
```

```
print(a)
Out[4]:
Yendo a caza de plantas salutíferas,
dijo a un lagarto: ¡Qué ansias tan mortíferas!
Quiero, por mis turgencias semi-hidrópicas,
chupar el zumo de hojas heliotrópicas.
Atónito el lagarto con lo exótico
de todo aquel preámbulo estrambótico,
no entendió más la frase macarrónica
que si le hablasen lengua babilónica;
pero notó que el charlatán ridículo
de hojas de girasol llenó el ventrículo,
y le dijo: Ya, en fin, señor hidrópico,
he entendido lo que es zumo heliotrópico.
```

La variable a es una gran cadena conteniendo todo el archivo. Aunque no lo veamos, a contiene saltos de línea, así que podemos dividir por renglones usando el método `split()` con argumento `"\n"`, lo que nos deja una lista:

```
In [5]:
print(a.split("\n"))
Out[5]:
['Yendo a caza de plantas salutíferas, ', 'dijo a un lagarto: ¡Qué
    ansias tan mortíferas! ', 'Quiero, por mis turgencias semi-hidró
    picas,', 'chupar el zumo de hojas heliotrópicas. ', 'Atónito el
    lagarto con lo exótico ', 'de todo aquel preámbulo estrambótico, ',
     'no entendió más la frase macarrónica ', 'que si le hablasen
    lengua babilónica;', 'pero notó que el charlatán ridículo ', 'de
    hojas de girasol llenó el ventrículo, ', 'y le dijo: Ya, en fin, se
    ñor hidrópico, ', 'he entendido lo que es zumo heliotrópico.']
```

Vamos a intentar ahora con `readline()`, usando la última parte de la fábula:

```
In [6]:
f= open("el grillo.txt",encoding="utf8")
b= f.readline()
f.close()
print(b)
Out[6]:
¡Y no es bueno que un grillo, oyendo el diálogo,
```

Sólo nos leyó una línea, la cual contiene el `"\n"`. Para leer todo el archivo tendríamos que ir línea por línea con este método. Intentemos ahora con `readlines()`

```
In [7]:
f=open("el grillo.txt",encoding="utf8")
B=f.readlines()
f.close()
print(B)
Out[7]:
['¡Y no es bueno que un grillo, oyendo el diálogo,\n', 'aunque se fue
    en ayunas del catálogo \n', 'de términos tan raros y magníficos, \n
    ', 'hizo del gato elogios honoríficos! \n', 'Sí; que hay quien
    tiene la hinchazón por mérito, \n', 'y el hablar liso y llano por
    demérito.\n', '\n', 'Mas ya que esos amantes de hiperbólicas \n', '
    cláusulas y metáforas diabólicas, \n', 'de retumbantes voces el dep
```

```
ósito \n', 'apuran, aunque salga un despropósito, \n', 'caiga sobre
su estilo problemático\n', 'este apólogo esdrújulo-enigmático.']
```

Este resultado es más parecido al obtenido con el `for`. Nada más que, con el `for` se pueden procesar las líneas de una vez, aprovechando que estamos en el ciclo.

Por completez, dejamos la fábula completa para el lector.

El gato, el lagarto y el grillo, de Tomás de Iriarte

Por más ridículo que sea el estilo retumbante, siempre habrá necios que le aplaudan, sólo por la razón de que se quedan sin entenderle

Ello es que hay animales muy científicos
en curarse con varios específicos
y en conservar su construcción orgánica,
como hábiles que son en la Botánica;
pues conocen las hierbas diuréticas,
catárticas, narcóticas, eméticas,
febrífugas, estípticas, prolíficas,
cefálicas también y sudoríficas.
En esto era gran práctico y teórico
un gato, pedantísimo retórico,
que hablaba en un estilo tan enfático
como el más estirado catedrático.
Yendo a caza de plantas salutíferas,
dijo a un lagarto: «¡Qué ansias tan mortíferas!
Quiero, por mis turgencias semi-hidrópicas,
chupar el zumo de hojas heliotrópicas».
Atónito el lagarto con lo exótico
de todo aquel preámbulo estrambótico,
no entendió más la frase macarrónica
que si le hablasen lengua babilónica;
pero notó que el charlatán ridículo
de hojas de girasol llenó el ventrículo,
y le dijo: «Ya, en fin, señor hidrópico,
he entendido lo que es zumo heliotrópico».
¡Y no es bueno que un grillo, oyendo el diálogo,
aunque se fue en ayunas del catálogo
de términos tan raros y magníficos,
hizo del gato elogios honoríficos!
Sí; que hay quien tiene la hinchazón por mérito,
y el hablar liso y llano por demérito.

Mas ya que esos amantes de hiperbólicas
cláusulas y metáforas diabólicas,
de retumbantes voces el depósito
apuran, aunque salga un despropósito,
caiga sobre su estilo problemático
este apólogo esdrújulo-enigmático.

Ejercicios:

En un documento de texto `(.txt)` copia y pega una canción que te guste. Imprime en la consola las cuatro primeras líneas de la canción, sin el título.

3 Estructuras de datos básicos

Hay estructuras de datos abstractas que se pueden implementar en cualquier lenguaje de programación. `Python` ya tiene incorporadas algunas de estas esctructuras. En especial, una estructura llamada hash table es el amigable diccionario. Los objetivos de aprendizaje de este capítulo es que el lector conozca y opere con los conjuntos, diccionarios y tuplas, que conozca sus diferencias para aplicarlos en diferentes contextos.

3.1. Conjuntos

Los conjuntos en `Python` operan tal cual como los conjuntos en matemáticas. A diferencia de las listas, los conjuntos no están ordenados, sus elementos están como en una bolsa. Y, por ser conjuntos, no hay elementos repetidos. Podemos construir un conjunto a partir de un iterable como una lista o una cadena con `set()`:

```
In [1]:
A= set("Parangaricutirimícuaro")
print(A)
Out[1]:
{'P', 'a', 'c', 'g', 'i', 'm', 'n', 'o', 'r', 't', 'u', 'í'}
```

Aunque haya unas llaves resguardando a los elementos, los conjuntos vacíos no se inicializan con llaves, como las listas vacías con los corchetes, pues las llaves se usan también en otra estructura llamada diccionario. Para tener un conjunto vacío entonces, lo construimos a partir de una lista o una tupla vacía:

```
In [2]:
B = set([])
C = set(())
print(type(B), type(C))
Out[2]:
<class 'set'> <class 'set'>
```

Los métodos externos con que cuentan los conjuntos son:

```
In [3]:
print([i for i in dir(A) if i[0]!="_"])
Out[3]:
['add', 'clear', 'copy', 'difference', 'difference_update', 'discard',
    'intersection', 'intersection_update', 'isdisjoint', 'issubset', '
   issuperset', 'pop', 'remove', 'symmetric_difference', '
   symmetric_difference_update', 'union', 'update']
```

Vemos que, en lugar del método `append()` para agregar un elemento, como en las listas, aquí se usa `add()`. Vamos ahora por ejemplo a quitar la `"í"` de nuestro conjunto A, pues ya tenemos como representante a la `"i"`:

```
In [4]:
A.remove("í")
print(A)
Out[4]:
{'r', 'o', 'm', 'P', 'i', 't', 'c', 'u', 'a', 'g', 'n'}
```

Vamos ahora a formar otros dos conjuntos para poder operar con ellos con las reglas usuales. Por ejemplo, veamos las diferencias entre las funciones de matemáticas básicas del módulo `math` y del módulo `cmath`, el segundo para trabajar con números complejos:

```
In [5]:
import math
import cmath
A = set([i for i in dir(math) if i[0]!="_"])
B = set([i for i in dir(cmath) if i[0]!="_"])
print("A:", A)
print("B:", B)
Out[5]:
A: {'tau', 'atan2', 'erfc', 'exp2', 'atan', 'asinh', 'inf', 'prod', 'ceil', 'sqrt', 'acosh', 'fmod', 'pow', 'fsum', 'cos', 'log10', 'dist', 'hypot', 'expm1', 'gamma', 'erf', 'cbrt', 'radians', 'sin', 'isinf', 'pi', 'factorial', 'sinh', 'isnan', 'frexp', 'tan', 'asin', 'fabs', 'degrees', 'log1p', 'isqrt', 'log2', 'ulp', 'gcd', 'nan', 'isclose', 'lcm', 'tanh', 'perm', 'e', 'log', 'ldexp', 'cosh', 'copysign', 'comb', 'floor', 'exp', 'lgamma', 'isfinite', 'nextafter', 'modf', 'remainder', 'atanh', 'trunc', 'acos'}
B: {'tau', 'atan', 'asinh', 'inf', 'sqrt', 'acosh', 'cos', 'log10', 'polar', 'sin', 'isinf', 'pi', 'sinh', 'isnan', 'tan', 'asin', 'phase', 'nan', 'isclose', 'rect', 'tanh', 'e', 'log', 'cosh', 'exp', 'nanj', 'isfinite', 'infj', 'atanh', 'acos'}
```

Aunque muchas de las funciones de los dos conjuntos compartan nombre, ¡las funciones son diferentes! Si usas las funciones de `cmath` y tienes puros números reales, vas a estar cargando la parte compleja por todos lados, `+0j`, es mejor en esos casos quedarse con `math`. Teniendo esto claro, sigamos con el ejemplo y, vamos a ver si, como parece a primera vista, B es un subconjunto de A:

```
In [6]:
print(B.issubset(A))
Out[6]:
False
```

B no es un subconjunto de A. Entonces veamos qué funciones están en B que no están en A. Para ello, tomamos la diferencia, que se puede escribir tal cual con el signo de menos, $-$, o se puede usar el método `difference`:

```
In [7]:
print(B-A)
print(B.difference(A))
Out[7]:
{'polar', 'nanj', 'phase', 'rect', 'infj'}
{'polar', 'nanj', 'phase', 'rect', 'infj'}
```

Algunas de estas funciones deben ser para la forma polar de los números complejos. Además de la diferencia, se pueden sustituir otros métodos por símbolos para las operaciones más usadas en conjuntos, como el `& (and)` para la intersección, el | `(or)` para la unión, y el `^` para los elementos que están en A y en B, quitando su intersección. Probemos esta última:

```
In [8]:
print(A^B)
Out[8]:
{'atan2', 'erfc', 'exp2', 'prod', 'ceil', 'fmod', 'pow', 'fsum', 'dist
    ', 'hypot', 'expm1', 'gamma', 'erf', 'polar', 'cbrt', 'radians', '
    factorial', 'frexp', 'fabs', 'degrees', 'log1p', 'phase', 'isqrt',
    'log2', 'ulp', 'gcd', 'lcm', 'rect', 'perm', 'ldexp', 'copysign', '
    comb', 'floor', 'lgamma', 'nanj', 'nextafter', 'modf', 'infj', '
    remainder', 'trunc'}
```

Ejercicios:

Construye una lista de los números que son múltiplos del 4, del 0 al 200, inclusive, y una lista con los números múltiplos del 3, en el mismo rango. Después, usando métodos de conjuntos, encuentra los números comunes a las dos listas, sin importar el orden de aparición.

3.2. Diccionarios

La siguiente estructura de datos, el diccionario, también conocido en general como hash table, es tan útil como las listas, es otro caballito de batalla. El diccionario consiste en guardar valores con ciertas claves dadas. Accedemos fácilmente a un valor sólo dando la clave.

Inicializamos un diccionario con:

```
In [1]:
d = dict()
```

y también se puede inicializar un diccionario con `d={}`. Los métodos del diccionario los encontramos, como siempre, con dir()

```
In [2]:
print([i for i in dir(d) if i[0]!="_"])
Out[2]:
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem'
    , 'setdefault', 'update', 'values']
```

Para guardar un valor `value` en una clave `key`, se usa `d[key]=value`, donde la clave tiene que ser única, y además tiene que ser un objeto inmutable, como una cadena, no podemos tener listas u objetos que puedan modificarse permanentemente, esto es, la clave es segura. El valor puede ser cualquier tipo de objeto.

Para acceder al valor de la clave `key` sólo se llama a `d[key]` o a `d.get(key)`, que también nos puede servir para dar una opción de valor en caso de que `key` no se encuentre en el diccionario, `d.get(key, default_value)`. Para remover un elemento

de un diccionario, se usa `d.pop(key)`. Para acceder tanto a las claves como a los valores, se usa `d.items()`. Por ejemplo:

```
In [3]:
d["a"]=1           # asignamos valores
d["b"]=2
for key, value in d.items():
    print(key, value)
Out[3]:
b 2
a 1
```

```
In [4]:
d["b"]=3            # cambiamos el valor de "b"
d.pop("a")          # eliminamos a "a"
print("d = ",d)     # imprimimos el diccionario
# si no estamos seguros de que una clave pertenezca al diccionario, le
    # damos un valor por default
print("d[c]:",d.get("c",0))
# pero si estamos seguros que un elemento está en el diccionario, lo
# llamamos fácilmente
print("d[b]:",d["b"])
Out[4]:
d =  {'b': 3}
d[c]: 0
d[b]: 3
```

El diccionario también puede tener elementos desde el principio, sin que todo lo tengamos que llenar con d[key]=value. La notación, en ese caso, es que los elementos están separados por coma, y la clave está separada del valor por dos puntos:

```
In [5]:
pintar_animal = {"león":"amarillo", "tigre":"naranja", "elefante": \
"gris", "pollito":"amarillo", "cotorro":"verde"}
print(pintar_animal["elefante"])
Out[5]:
'gris'
```

Ejercicio resuelto: chismógrafo

Pregunta a unos de tus amigos cuáles son algunas de sus cosas favoritas y guárdalas en un archivo excel, con la primera columna para el nombre de tus amigos. El chismógrafo en cuestión consiste en el a) color, b) animal, c) canción, d) película y e) libro favoritos de tus amigos en ese orden.

- Convierte el archivo en `txt` separado por tabs. Para ello ve a Archivo → Guardar como y seleccionas el tipo de archivo de Texto (delimitado por tabulaciones) (*.txt)

-Lee el archivo en Python (trata de poner el archivo `.txt` y el archivo `.py` en el mismo fólder para no escribir todo el path), creando un diccionario con clave el nombre de los estudiantes, y valor una lista de sus cosas favoritas.

- Para cada estudiante, escribe: "El animal favorito de fulanito es el/la animalito, le gusta tararear cancioncita, y le encantó leer librito". (Por supuesto que tienes que reemplazar por las cosas favoritas).

Respuesta:

Leemos el archivo y tenemos cuidado de no agregar en el diccionario la primera fila con los títulos. Recordamos que usamos `strip()` para quitar espacios en blanco al principio y final de la línea, y los saltos de línea, y split(símbolo) para dividir una cadena en una lista, con el símbolo de división siendo en este caso una tabulación o tab, `"\t"`. El primer elemento de cada lista es el nombre del estudiante, y el resto de la lista tiene sus cosas favoritas:

```
In [1]:
g=open("favoritos.txt")
g.readline()            # nos saltamos la primera línea de títulos
d= dict()
for line in g:          # a partir de la segunda línea
    A=line.strip().split("\t")     # una lista A por estudiante
    d[A[0]] = A[1:]     # d[nombre]= [lista de favoritos]
g.close()
for a,b in d.items():   # checamos que leímos bien el archivo
    print(a,b)
Out[1]:
Luis ['verde militar', 'buitre abante', 'Vals 2 de Shostakovich', '
   Jurasic Park', 'Las mil y una noches']
David ['morado', 'perro', 'A mi manera', 'El padrino', 'La apología de
    Sócrates']
Omar ['verde', 'tiburón', 'oh bambino mio caro', 'midiendo el mundo',
   'análisis matemático']
Erika ['azul', 'león', 'Obertura 1812', 'Coco', 'Harry Potter']
Nataly ['turquesa', 'perro', 'amnesia', 'Harry Potter', 'Mal amor']
Roberto ['rojo', 'serpiente', 'Ella', 'La gran pelea', 'Cuentos de
   Andersen']
Carlos IV ['azul', 'araña', 'No more work', 'Terminator', 'El arte de
   amar']
Ana ['azul', 'gato', 'humanity', 'saw', 'crónicas de una muerte
   anunciada']
Beto ['morado', 'gato', 'my inmortal', 'El Padrino', 'El Padrino']
Jahir ['azul', 'perro', 'viva la vida', 'Los indestructibles', 'El
   zarco']
Adán ['azul', 'puma', 'vaquero rockanrolero', 'Transformers', 'El
   Quijote']
Frida ['azul', 'cobaya', 'cuando tú no estás', 'MegaMente', 'La
   soledad de los números primos']
```

```
In [2]:
for a,b in d.items():
    print("El animal favorito de "+a+" es el "+b[1]+", le gusta\
   tararear "+b[2]+" y le encantó leer "+b[-1])
Out[2]:
El animal favorito de Luis es el buitre abante, le gusta tararear Vals
    2 de Shostakovich y le encantó leer Las mil y una noches
El animal favorito de David es el perro, le gusta tararear A mi manera
    y le encantó leer La apología de Sócrates
El animal favorito de Omar es el tiburón, le gusta tararear oh bambino
    mio caro y le encantó leer análisis matemático
El animal favorito de Erika es el león, le gusta tararear Obertura
   1812 y le encantó leer Harry Potter
El animal favorito de Nataly es el perro, le gusta tararear amnesia y
```

A, Á, a, á → 0	G, g → 6	M, m → @	R, r → (	X, x →>
B, b → 1	H, h → 7	N, n → $	S, s →)	Y, y →<
C, c → 2	I, Í, i, í → 8	Ñ, ñ → +	T, t → [	Z, z →=
D, d → 3	J, j → 9	O, Ó, o, ó → −	U, Ú, u, ú →]	
E, É, e, é → 4	K, k → #	P, p → ∗	V, v → {	
F, f → 5	L, l → %	Q, q → /	W, w →}	

Tabla 3.1: Un código de encriptación

```
    le encantó leer Mal amor
El animal favorito de Roberto es el serpiente, le gusta tararear Ella
    y le encantó leer Cuentos de Andersen
El animal favorito de Carlos IV es el araña, le gusta tararear No more
     work y le encantó leer El arte de amar
El animal favorito de Ana es el gato, le gusta tararear humanity y le
    encantó leer crónicas de una muerte anunciada
El animal favorito de Beto es el gato, le gusta tararear my inmortal y
     le encantó leer El Padrino
El animal favorito de Jahir es el perro, le gusta tararear viva la
    vida y le encantó leer El zarco
El animal favorito de Adán es el puma, le gusta tararear vaquero
    rockanrolero y le encantó leer El Quijote
El animal favorito de Frida es el cobaya, le gusta tararear cuando tú
    no estás y le encantó leer La soledad de los números primos
```

Ejercicios:

a) Dada la Tabla 3.1 de encriptación (Recuerda que estos signos en Python van entrecomillados):

y mapeando los espacios en blanco a guiones bajos (_), encripta el siguiente mensaje:

"qué voy a hacer no sé no encuentro nada nada nada la solución no sé cómo encontrarla dime trato de olvidarte y yo quiero olvidarte y yo no sé cómo te olvido siempre en mi mente".

b) desencripta el siguiente mensaje, usando el mismo lenguaje de encriptación, y donde todo quede en minúsculas, sin acento:

``2]0$3-_4%_0@-(_%%460_0)8_34_4)[0_@0$4(0_]$-_$8_)4_30_$8_2]4$[0_4%
_20(][0%_(4{4(3424_<_4%_6]0@0278[-_5%-(424_<_%0_)-60_)4_(4{84$[0''

3.3. Tuplas

En esta ocasión vamos a aprovechar la estructura de tuplas para resolver un problema. El problema consiste en que, en una universidad, las materias del ciclo básico se programan para que no haya traslape en un dado semestre, pero, hay varios estudiantes que toman materias de diferente semestre para los cuales puede haber traslapes. Hagamos un programa que cuente, vía entrevista previa, cuántos estudiantes requieren llevar un

par dado de materias, para evitar traslapes entre ellas, dando prioridad a las que tengan una mayor frecuencia, y dejando fuera los pares de materias que pertenecen al mismo semestre y por los cuáles no nos tendríamos que preocupar (esto es opcional).

Suponemos que los datos están en excel, así que convirtamos primero a `.txt` separado por tabs, o a `.csv` (separado por comas). Una vez que tenemos nuestro archivo `.txt` o `.csv`, hacemos

```
In [1]:
f= open("C:/Users/eriko/Desktop/porquería/horarios/materias.txt")
# si el archivo está en el mismo directorio, solo pones "materias.txt"
A = [line.strip().split("\t") for line in f]
f.close()
print(A)
Out[1]:
[['Beto', 'MN', 'PA', 'II', 'MD2', 'TF'],
 ['Frida', 'TF', 'PE', 'MNED', 'PA'],
 ['Jahir', 'MN', 'PA', 'EM', 'CV'],
 ['Omar', 'AM', 'MD2', 'PA', 'AS'],
 ['Ana', 'TF', 'PE', 'MNED', 'PA', 'MD2'],
 ['Nataly', 'MN', 'TF', 'PA', 'MD1', 'CV'],
 ['Luis', 'MN', 'MEC2', 'PA', 'MD1', 'CV'],
 ['Adam', 'TF', 'MD2', 'PA', 'AM', 'PE'],
 ['Carlos', 'TF', 'AM', 'PA', 'AR1']]
```

Queremos trabajar con las materias nada más, así que construyamos otra lista que contenga sólo a estas:

```
In [2]:
materias=[i[1:] for i in A]    # misma lista pero desde segundo índice
print(materias)
Out[2]:
[['MN', 'PA', 'II', 'MD2', 'TF'],
 ['TF', 'PE', 'MNED', 'PA'],
 ['MN', 'PA', 'EM', 'CV'],
 ['AM', 'MD2', 'PA', 'AS'],
 ['TF', 'PE', 'MNED', 'PA', 'MD2'],
 ['MN', 'TF', 'PA', 'MD1', 'CV'],
 ['MN', 'MEC2', 'PA', 'MD1', 'CV'],
 ['TF', 'MD2', 'PA', 'AM', 'PE'],
 ['TF', 'AM', 'PA', 'AR1']]
```

La idea es contar, para cada par de materias, cuántos estudiantes las van a tomar a ambas. Necesitamos algo del tipo cuenta de `(mat1, mat2)` igual a un número. Lo que se nos puede venir a la mente rápidamente es usar un diccionario para llevar la cuenta. Nada más que el detalle es que necesitamos guardar un par de cadenas, las materias, que podamos separar fácilmente, y no podemos usar una lista como clave, por ser mutable. Podríamos intentar juntar las dos materias en una sola cadena separada por ejemplo por un guion, pero, precisamente para eso están las tuplas. Las tuplas guardan uno o más elementos, de forma similar a las listas, con la diferencia de que la tupla es inmutable. Su notación es $a = (o_1, o_2, \ldots, o_n)$, y así, nuestro diccionario sería de la forma `cuenta[(mat1, mat2)]`.

Las tuplas pueden usarse como ya vimos en el tema de las funciones, cuando definimos

una función y queremos retornar más de un objeto, o como en el caso actual, que requerimos una clave de más de un objeto en un diccionario, entre otros usos.

Siguiendo con el ejemplo, contemos cuántos estudiantes están pidiendo el mismo par de cursos. Por cada estudiante, vamos a ordenar primero su lista de materias, pues no queremos que `(mat1, mat2)` se considere distinta a `(mat2, mat1)`. Después, con 2 for loops vamos a formar todos los posibles pares de materias.

```
In [3]:
cuenta=dict()          # aquí llevamos la cuenta de cada par de materias
for lista in materias:
    lista.sort()      # ordenamos la lista de cada estudiante
    for i in range(len(lista)):
        for j in range(i+1,len(lista)):
        cuenta[(lista[i],lista[j])]=\
        cuenta.get((lista[i],lista[j]),0)+1
```

donde si el par `(lista[i],lista[j])` todavía no existe en el diccionario, le damos un valor por default de cero.

Este diccionario no es muy fácil de visualizar,

```
In [4]:
print(cuenta)
Out[4]:
{('MN', 'TF'): 2, ('MD2', 'MN'): 1, ('AM', 'MD2'): 2, ('AS', 'MD2'):
   1, ('II', 'MD2'): 1, ('MNED', 'PA'): 2, ('MD2', 'PA'): 4, ('MD1', '
   MEC2'): 1, ('MD2', 'PE'): 2, ('PA', 'TF'): 6, ('AS', 'PA'): 1, ('AM
   ', 'AS'): 1, ('AM', 'TF'): 2, ('CV', 'PA'): 3, ('MD2', 'TF'): 3, ('
   EM', 'MN'): 1, ('CV', 'MEC2'): 1, ('EM', 'PA'): 1, ('CV', 'MD1'):
   2, ('MD1', 'MN'): 2, ('PA', 'PE'): 3, ('MNED', 'PE'): 2, ('II', 'PA
   '): 1, ('AR1', 'TF'): 1, ('PE', 'TF'): 3, ('AM', 'PE'): 1, ('CV', '
   EM'): 1, ('II', 'MN'): 1, ('MD1', 'TF'): 1, ('MN', 'PA'): 4, ('MEC2
   ', 'PA'): 1, ('AM', 'AR1'): 1, ('MD1', 'PA'): 2, ('MEC2', 'MN'): 1,
    ('II', 'TF'): 1, ('AM', 'PA'): 3, ('AR1', 'PA'): 1, ('MD2', 'MNED'
   ): 1, ('CV', 'MN'): 3, ('CV', 'TF'): 1, ('MNED', 'TF'): 2}
```

Sería más conveniente que tuviéramos el diccionario invertido, donde juntáramos los pares de materias con igual frecuencia. Definamos otro diccionario donde invirtamos los papeles de `key` y `value`:

```
In [5]:
frecuencias=dict()
# si la frecuencia b todavía no existe en el diccionario,
# le asignamos una lista vacía por default
# el par de materias, a, se agrega poco a poco
for a,b in cuenta.items():
    frecuencias[b]=frecuencias.get(b,[])+[a]
```

Las claves del diccionario frecuencias son, por tanto, las frecuencias, y sus valores son una lista de los pares de materias con esa frecuencia

```
In [6]:
print(frecuencias)
Out[6]:
```

```
{1: [('CV', 'TF'), ('MEC2', 'MN'), ('II', 'MD2'), ('MD1', 'TF'), ('EM', 'PA'), ('CV', 'MEC2'), ('CV', 'EM'), ('EM', 'MN'), ('MEC2', 'PA'), ('II', 'TF'), ('MD2', 'MNED'), ('AM', 'AS'), ('AS', 'PA'), ('MD1', 'MEC2'), ('AM', 'AR1'), ('AR1', 'TF'), ('II', 'PA'), ('AR1', 'PA'), ('MD2', 'MN'), ('AM', 'PE'), ('AS', 'MD2'), ('II', 'MN')],
 2: [('MD1', 'MN'), ('AM', 'TF'), ('MN', 'TF'), ('AM', 'MD2'),
  ('MD1', 'PA'), ('MNED', 'PA'), ('MNED', 'PE'), ('MNED', 'TF'),
  ('MD2', 'PE'), ('CV', 'MD1')],
 3: [('PE', 'TF'), ('CV', 'MN'), ('MD2', 'TF'), ('AM', 'PA'), ('PA', 'PE'), ('CV', 'PA')],
 4: [('MD2', 'PA'), ('MN', 'PA')],
 6: [('PA', 'TF')]}
```

El diccionario no ordena en general sus claves, no es un conjunto ordenado como las listas, por lo que, un paso final sería ordenar las frecuencias con la función `sorted()`, con argumento para ordenar en orden descendente, y ahora sí, imprimir a partir de esa lista ordenada las materias:

```
In [7]:
ordenados = sorted(frecuencias, reverse=True)
# ordenados es sólo una lista de las frecuencias, no un nuevo
# diccionario
print("frecuencias ordenadas en orden decreciente", ordenados)
for i in ordenados:
    print(i, frecuencias[i])
Out[7]:
frecuencias ordenadas en orden decreciente [6, 4, 3, 2, 1]
6 [('PA', 'TF')]
4 [('MD2', 'PA'), ('MN', 'PA')]
3 [('PE', 'TF'), ('CV', 'MN'), ('MD2', 'TF'), ('AM', 'PA'), ('PA', 'PE'), ('CV', 'PA')]
2 [('MD1', 'MN'), ('AM', 'TF'), ('MN', 'TF'), ('AM', 'MD2'), ('MD1', 'PA'), ('MNED', 'PA'), ('MNED', 'PE'), ('MNED', 'TF'), ('MD2', 'PE'), ('CV', 'MD1')]
1 [('CV', 'TF'), ('MEC2', 'MN'), ('II', 'MD2'), ('MD1', 'TF'), ('EM', 'PA'), ('CV', 'MEC2'), ('CV', 'EM'), ('EM', 'MN'), ('MEC2', 'PA'), ('II', 'TF'), ('MD2', 'MNED'), ('AM', 'AS'), ('AS', 'PA'), ('MD1', 'MEC2'), ('AM', 'AR1'), ('AR1', 'TF'), ('II', 'PA'), ('AR1', 'PA'), ('MD2', 'MN'), ('AM', 'PE'), ('AS', 'MD2'), ('II', 'MN')]
```

Concluimos exhortando que, por favor, si esos pares de materias pertenecen a diferentes semestres, no se programen en el mismo horario, dando prioridad a los pares de materias con mayor frecuencia.

4 Numpy

En este capítulo veremos uno de los módulos más utilizados en las ciencias y la ciencia de datos, que es el `numpy`, que nos sirve para hacer operaciones vectoriales, así como facilitar la graficación, entre otras. `Numpy` en sí es todo un mundo y seguramente quedaremos cortos en todo lo que se puede hacer con este módulo, pero hay muchos tutoriales y recursos en internet para complementar lo visto aquí. Los objetivos de aprendizaje para el lector en este capítulo son, generar arreglos, en particular de tipo vector y matriz para operar con ellos en el ámbito del Álgebra Lineal. Mapear funciones de manera sencilla, con la ayuda de los vectores. Aprender diversas funciones de `numpy` para sus aplicaciones en diferentes ámbitos, en otros módulos que requieren de la estructura de arreglos. Seleccionar elementos de un arreglo dados diversos criterios, para aplicaciones varias. Realizar operaciones de tipo estadístico con los arreglos. Para ahondar más en el módulo de `numpy`, es la bibliografía se recomienda el manual en Ciencias de Datos en Python de Jake VanderPlas (2016), pero hay mucho material en línea que puedes revisar.

Recuerda que, aunque no se muestra en todos los ejercicios, cuando empiezas una sesión y trabajas con `numpy` debes importar el módulo con `import numpy as np`. También, como recordatorio, aunque parezca que haya que memorizar cómo usar una función, en `spyder` cuando se escribe ésta, además de que se despliegan todas las posibles funciones que empiezan con determinada letra, al escribir los paréntesis aparece qué argumentos se deben dar a la función, y cuáles son los valores por default de algunos de ellos, que podríamos cambiar en caso de ser necesario.

4.1. Generación de arreglos

Como primera aplicación, digamos que, por ejemplo, queremos encontrar los valores de la función $3x - 2\sin(4x) + 8$ en un intervalo de, por ejemplo, $(-5, 4)$. En `Python`, podríamos escribir la función y después aplicarla en ese intervalo con valores enteros con un `for`, o con cualquier tipo de valor, no necesariamente entero, con un `while`. También podríamos usar una función que no hemos visto llamada `map()` que mapea la función a un listado de números. Vamos a hacerlo de las dos maneras:

```
1 In [1]:
2 import math
3 def función(x):
4     return(3*x-2*math.sin(4*x)+8)
5 A1 = [función(x) for x in range(-5,5)]
6 print(A1)
```

```
Out[1]:
[-5.174109498544745, -4.5758066333301315, -2.0731458360008705,
   3.978716493246764, 3.486395009384143, 8.0, 12.513604990615857,
   12.021283506753235, 18.07314583600087, 20.57580663333013]
```

Si aplicamos la función con `map` en el mismo intervalo, hay que comentar un detalle. Hay objetos que se llaman "generadores" que calculan resultados sin que se guarden en la memoria, se ejecutan al vuelo. Para guardar los resultados debemos anteponer `list` al mapeo, al ser `map` un generador:

```
In [2]:
A2 = list(map(función, range(-5,5)))
print(A2)
Out[2]:
[-5.174109498544745, -4.5758066333301315, -2.0731458360008705,
   3.978716493246764, 3.486395009384143, 8.0, 12.513604990615857,
   12.021283506753235, 18.07314583600087, 20.57580663333013]
```

Si ahora lo hacemos con `numpy`, cuyo alias es `np`, no hay necesidad de hacer un `for` loop o de mapear en un intervalo:

```
In [3]:
import numpy as np
x = np.arange(-5,5)       # crea un arreglo en el intervalo [-5,5)
y = 3*x-2*np.sin(4*x)+8
print(y)
Out[3]:
[-5.1741095  -4.57580663 -2.07314584  3.97871649  3.48639501  8.
 12.51360499 12.02128351 18.07314584 20.57580663]
```

Como vemos en el ejemplo, en la multiplicación $3x$, cada una de las entradas del arreglo x se multiplica por el escalar 3, y cuando se suma el 8, a cada una de las entradas del arreglo $3x$ se le suma el escalar 8. Y, al igual que el módulo `math`, `numpy` tiene las funciones usuales como las funciones trigonométricas, o constantes como π, sólo que `numpy` puede aplicar funciones a todo un arreglo, como en el ejemplo con la función seno, y no sólo a un número, es más general. `Numpy` tiene incluso el módulo `random` como submódulo, que se usa de manera similar a como lo aprendimos en el segundo capítulo, sólo que ahora hay que especificar la cantidad de números aleatorios que queramos, cuando aplique. Para checar el contenido de `numpy` usamos `dir()`, como siempre, pero en esta ocasión no lo vamos a imprimir en estas páginas por lo enorme del módulo que, además, contiene submódulos.

4.1.1. Vectores

Los objetos de `numpy`, como ya adelantamos, son arreglos, `ndarray`, pero con ello no nos referimos necesariamente nada más a objetos tipo vectores, también se pueden representar matrices y objetos de mayor dimensión. Empezando con los arreglos de tipo vector, unas de las formas de generarlos son:

```
In [4]:
# se da un rango y el paso, que no es necesariamente entero
a = np.arange(-2,3,0.5)
# se da un rango y la cantidad de números que queremos en ese rango
```

```
b = np.linspace(-2,3,21)
# convierte una iterable, como un rango o una lista, en un arreglo
c = np.array(range(-2,4))
print("a",a)
print("b",b)
print("c",c)
Out[4]:
a [-2.  -1.5 -1.  -0.5  0.   0.5  1.   1.5  2.   2.5]
b [-2.   -1.75 -1.5  -1.25 -1.   -0.75 -0.5  -0.25  0.    0.25  0.5
  0.75  1.    1.25  1.5   1.75  2.    2.25  2.5   2.75  3.  ]
c [-2 -1  0  1  2  3]
```

Vemos el tipo de objeto, como siempre, con `type()`, pero ahora también podemos ver el tipo de elementos que hay en el arreglo con el atributo `dtype`:

```
In [5]:
for i in (a,b,c):
    print(type(i), i.dtype)
Out[5]:
<class 'numpy.ndarray'> float64
<class 'numpy.ndarray'> float64
<class 'numpy.ndarray'> int32
```

En este ejemplo vemos que los elementos del vector `c` son de tipo entero, `int`. Si quisiéramos cambiarlos a `float`, por ejemplo, lo podemos hacer desde el inicio especificando el `dtype`, o con una conversión posterior:

```
In [6]:
d=np.array(range(-2,3),dtype=float)
c=c.astype(float)
print(d, d.dtype)
print(c, c.dtype)
Out[6]:
[-2. -1.  0.  1.  2.] float64
[-2. -1.  0.  1.  2.] float64
```

Podemos generar también arreglos aleatorios con el submódulo `random`:

```
In [7]:
np.random.seed(7)
# 10 enteros aleatorios entre el 1 y el 50:
print(np.random.randint(1,50,10))
# 3 números aleatorios de una gaussiana con mu=50 y sigma=5:
print(np.random.normal(50,5,3))
# 5 números aleatorios entre el 3 y el 20:
print(np.random.uniform(3,20,5))
Out[7]:
[48  5 26  4 20 24 40 29 15 24]
[46.05538486 50.01032786 49.99554807]
[14.54690993 16.66356361  9.47599926  4.1209179   7.89847519]
```

Ejercicios:

1. Crea un arreglo tipo vector en el intervalo $(0, \pi/6)$ que consista de 10 puntos. Encuentra la tangente de todos estos valores.

2. Crea un arreglo tipo vector en el intervalo $(0, 3)$ con un paso de tamaño 0.2. Encuentra el cubo de todas sus componentes.

3. Crea un arreglo de 9 entradas con números tomados con la misma probabilidad (distribución uniforme) entre el 4 y el 8. Encuentra el logaritmo de estos números.

4.1.2. Matrices

Para arreglos tipo matriz, una forma de generarlas es a partir de una lista de listas, o lista anidada, con la forma general `np.array()`, al igual que con vectores:

```
In [1]:
# matriz de 2x3 a partir de números dados, y sus dimensiones:
A=np.array([[1,4,6],[7,11,15]])
print(A,", dims= n,m=" ,A.shape)
Out[1]:
[[ 1  4  6]
 [ 7 11 15]] , dims= n,m= (2, 3)
```

Otra forma es para cuando se quieren matrices diagonales, sólo se usa la función `eye` (I, y `eye` suenan igual en inglés, "ai"):

```
In [2]:
# matriz diagonal de 3x3 multiplicada por una constante:
print(5*np.eye(3))
Out[2]:
[[5. 0. 0.]
 [0. 5. 0.]
 [0. 0. 5.]]
```

Otra forma es con el submódulo `random`, donde en lugar de dar un solo número como el número de elementos, damos el tamaño o `shape` de la matriz, (n, m). Dependiendo de la función, también será posible o no tener diferente intervalo de valores para las diferentes coordenadas:

```
In [3]:
# matriz de 3x2 de enteros aleatorios del 1 al 100:
# no se puede dar diferentes intevalos:
print(np.random.randint(1,100,(3,2)))
# matriz de 2x3 de números aleatorios del 1 al 20:
print(np.random.uniform(1,20,(2,3)))
# matriz de 2x3 de números con la primera columna entre el 1 y el 2,
# la segunda entre el 10 y el 20, y la tercera entre el 100 y 200:
print(np.random.uniform((1,10,100),(2,20,200),(2,3)))
Out[3]:
[[76 59]
 [48 17]
 [13 94]]
[[ 9.35052495 15.85297476 14.00794956]
 [14.99681483 10.29903439 19.12583672]]
[[  1.15373916  16.49118853 155.74135276]
 [  1.86853158  11.43628016 144.79043472]]
```

Otra forma de generar matrices es empezar con matrices con entradas igual a ceros o unos, que posteriormente serán cambiadas. Se pueden generar dando el tamaño de las

dimensiones con las funciones `zeros` y `ones`, o tomándolas de alguna otra matriz, con las funciones `zeros_like` y `ones_like`:

```
In [4]:
# matriz de 3x2 de ceros:
A=np.zeros((3,2))
print(A)
# matriz de unos del mismo tamaño que A:
print(np.ones_like(A))
Out[4]:
[[0. 0.]
 [0. 0.]
 [0. 0.]]
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

Ejercicios:

1. Genera una matriz de 2×3 con números aleatorios entre el 0 y el 1. Además de la distribución uniforme, existe la función `random` del submódulo `random` que genera estos números positivos menores a 1, y que tiene como único argumento el tamaño de la matriz.

2. Genera una matriz de ceros del mismo tamaño que la del problema 1.

4.2. Operaciones con vectores y matrices

4.2.1. Operaciones básicas

Entre una de las aplicaciones de `numpy` más directas y útiles en matemáticas, está la operación con matrices y el álgebra lineal en general. Estas operaciones se hacen de manera más rápida que con listas y con `for` loops. Vamos a mostrar las operaciones más usuales:

```
In [1]:
a = np.array([1,2,3])                    # un vector de 3 entradas
b = np.array([4,5,6])                    # un vector de 3 entradas
A = np.array([[1,2,3],[4,5,6]])          # una matriz de 2x3
B = np.array([[1,2],[3,4],[5,6]])        # una matriz de 3x2
# algunas operaciones:
print("5a:", 5*a)                        # escalar por vector
print("a+b:", a+b)                       # suma de vectores
print("a-b:", a-b)                       # resta de vectores
print("ab punto:", np.dot(a,b))          # producto punto de vectores
print("ab outer:", np.outer(a,b)) # vector columna por vector renglón
print("axb:", np.cross(a,b))             # producto cruz de vectores
print("Aa:", np.dot(A,a))                # matriz por vector
print("A^T:", A.T)                       # transpuesta de A
print("AB:", np.dot(A,B))                # producto de matrices
Out[1]:
5a: [ 5 10 15]
a+b: [5 7 9]
```

```
a-b: [-3 -3 -3]
ab punto: 32
ab outer: [[ 4  5  6]
 [ 8 10 12]
 [12 15 18]]
Axb: [-3  6 -3]
Aa: [14 32]
A^T: [[1 4]
 [2 5]
 [3 6]]
AB: [[22 28]
 [49 64]]
```

¡Hay que tener cuidado con el producto de matrices y vectores! La forma en que operan los arreglos se llama element-wise, o elemento a elemento, como lo vemos en la suma y resta de vectores, el primer elemento de uno con el primer elemento del otro, y así. Si usáramos el signo `*` como producto, en lugar de `dot`, se multiplicarían los términos en forma element-wise:

```
In [2]:
print(a*b)
Out[2]:
array([ 4, 10, 18])
```

Ejercicios:

1. Suma de vectores. Un turista, en una ciudad con cuadras muy bien alineadas y del mismo tamaño, se desplaza desde su hotel 7 cuadras hacia el Este y 2 cuadras hacia el Norte para llegar al primer lugar turístico. Al terminar, sale y recorre 2 cuadras hacia el Oeste y una cuadra hacia el Norte para su segunda parada. Horas más tarde se va finalmente a comer a un muy buen restaurante localizado 3 cuadras hacia el Sur y 2 cuadras hacia el Oeste desde su último punto. Con respecto al hotel, ¿cuál es su vector de desplazamiento medido en número de cuadras?

2. Producto punto. En física, cuando se aplica una fuerza constante el trabajo se define como el producto punto del vector fuerza y el vector desplazamiento. Una persona jala una caja con una fuerza cuyas componentes son $(15, 7)N$. La caja se desplaza 6 m por el suelo. ¿Qué trabajo realiza la persona sobre la caja?

3. Producto cruz. La componente magnética de la fuerza de Lorentz sobre una partícula está dada por $F = (q/c)\, v \times B$, donde q es la carga de la partícula, v su vector de velocidad, c la velocidad de la luz, y B el vector de campo magnético. Un electrón, cuya carga es igual a $q = -1.6 \times 10^{-19}$ C, tiene un vector de velocidad igual a $v = (0.3c, 0.1c)$ en un plano, y cruza una región donde hay un campo magnético cuyas componentes son $B = (0.03, 0.02)T$ en el mismo plano del vector velocidad. Encuentra la magnitud de la fuerza que siente el electrón debido a la presencia del campo magnético, en Newtons (Nota: en `numpy`, cuando los vectores son en 3D, el producto cruz se expresa también como un vector en 3D. Cuando los vectores viven en 2D, el producto cruz se expresa como una magnitud, que, como sabemos, es la magintud de un vector que vive en la dirección perpendicular al plano).

4. Producto de matriz y vector. El vector $v = (\sqrt{3}, 1)$ se rota 60° en la dirección antihoraria en el plano x-y. ¿Cuáles son sus componentes después de la rotación? Recuerda que la operación de rotación está dada por $v' = Rv$, con la matriz de rotación dada por $R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$.

4.2.2. Broadcasting

En algunos casos, se pueden hacer operaciones con objetos de diferentes dimensiones. Ya vimos un ejemplo con la función $3x - 2\sin(4x) + 8$, donde el 8 puede pasar de ser un escalar a ser un vector del mismo tamaño que x. A esta ampliación del tamaño de las dimensiones se le llama broadcasting, que, traducido al español, significa radiodifusión, pero aquí el broad significa ampliar. Hay reglas para ello, por ejemplo, no podemos sumar un vector de dos entradas con uno de tres entradas, pues los vectores son unidimensionales y el tamaño de su única dimensión difiere. Pero, por ejemplo, podemos sumar un número a un vector porque, aunque el número no tiene dimensiones, se le puede dotar de una dimensión que iguale en tamaño a la del vector. En general, cuando se pueda dotar de una dimensión extra a un objeto, esta dimensión se amplía para igualar el tamaño de una de las dimensiones del otro objeto. Sin embargo, la expansión de dimensiones no es tan arbitraria, las nuevas dimensiones que aparecen son las primeras. Por ejemplo, si un vector se tiene que expandir para sumarse con una matriz, el vector, de tener una dimensión, pasa a tener dos dimensiones, la de los renglones y la de las columnas, y el tamaño de la dimensión de las columnas va a ser igual al tamaño original del vector, mientras que la nueva dimensión, de los renglones, que van primero, se amplía para tener el mismo tamaño de la matriz. Por tanto, para que la suma no marque error el tamaño de las columnas de la matriz a sumar debe ser igual al tamaño del vector original.

Veamos unos ejemplos:

- Un número se puede expandir para tener las dimensiones de un vector o una matriz:

```
In [3]:
a=np.array([1,2,3])                # vector de 3 entradas
A=np.array([[1,2,3],[4,5,6]])      # matriz de 2x3
num=3
print("a+num:",a+num)      # vector más escalar(->convertido a vector)
print("A+num:",A+num)      # matriz más escalar(->convertido a matriz)
Out[3]:
a+num: [4 5 6]
A+num: [[4 5 6]
 [7 8 9]]
```

En el ejemplo de arriba, como el número de columnas de A, 3, es el mismo que el tamaño de a, entonces sí se pueden sumar vía broadcast; a se expande a una matriz de 2×3, duplicándose, para igualar las dimensiones de A:

```
In [4]:
print("A+a:", A+a)      # matriz más vector(->convertido a matriz)
Out[4]:
A+a: [[2 4 6]
 [5 7 9]]
```

En este ejemplo, si la matriz hubiera sido de 3×2, la suma con el vector marcaría error.

Ejercicios:

Checa si las siguientes operaciones se pueden resolver con broadcasting, hazlo a mano de ser posible, y después checa con `Python` el resultado:

$$a = (3,2,1), \quad A = \begin{pmatrix} 2 & 4 & 6 \\ 7 & 9 & 15 \end{pmatrix}$$

a) $a + A$, b) $a + A^T$, c) $a * A$ (¡element wise!)

4.2.3. Diferencias entre un vector y una matriz

Veamos el tamaño de un arreglo tipo vector:

```
In [1]:
a=np.array([8,4,2])
print(a.shape)
Out[1]:
(3,)
```

Vemos que el vector tiene una sola dimensión, con tamaño 3 (las tuplas de un solo elemento tienen una coma, para diferenciarlas de un objeto entre paréntesis normales).

En algunas operaciones, el vector es tratado como si fuera un ente diferente a las matrices. Por ejemplo, en la multiplicación entre matrices, con `dot`, se debe de tener cuidado con las dimensiones de las matrices, hay que transponer en caso de que sea necesario. Con los vectores, no importa, `numpy` entiende que debe hacer un producto punto o un producto exterior entre ellos, con `dot` y `outer`, sin necesidad de transponer ningún vector (en el primero, se multiplica un vector renglón por un vector columna y en el segundo se invierten los papeles), sólo importa que los vectores tengan el mismo tamaño. En otros casos, hay operaciones matriciales que marcan error cuando se realizan entre una matriz y un vector, porque se espera que los arreglos tengan la misma dimensión. Hay una función, `reshape()`, que permite cambiar las dimensiones de cualquier arreglo, siempre que su número de entradas no cambie. Con esta función podemos cambiar nuestro vector para que se entienda como una matriz, y podamos hablar sin confusión de vectores de tipo renglón o de tipo columna.

Veamos por ejemplo la función `concatenate()`, que junta dos o más arreglos, o dos o más matrices para formar una más grande, actuando sobre una matriz y el vector a de arriba:

```
In [2]:
A=np.array([[3,6,2],[1,3,7]])
print(np.concatenate((A,a)))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-31-2de60d974339> in <module>()
      1 A=np.array([[3,6,2],[1,3,7]])
----> 2 np.concatenate((A,a))

```

```
ValueError: all the input arrays must have same number of dimensions
```

Se marca un error, por el número diferente de dimensiones. En estos casos, es mejor prevenir y usar `reshape()`:

```
In [3]:
a=a.reshape(1,3)  # matriz de 1 renglón y 3 columnas, o vector renglón
print(a)
print(a.shape)
Out[3]:
[[8 4 2]]
(1, 3)
```

Hay que notar que el vector ahora tiene doble paréntesis, ahora es una matriz de un solo renglón, tiene dos dimensiones que ahora sí son claramente de tamaño 1×3. Tratamos de nuevo de concatenar:

```
In [4]:
print(np.concatenate((A,a)))
Out[4]:
array([[3, 6, 2],
       [1, 3, 7],
       [8, 4, 2]])
```

Ejercicios:

1. Convierte el arreglo `np.arange(1,30,3)` en una matriz de una sola columna. Para saber la dimensión del vector original, usa el atributo `shape`.

2. Convierte el arreglo `np.random.randint(1,21,12)` en una matriz de 4×3.

3. Convierte el arreglo `np.linspace(3,9,20)` en una matriz de un solo renglón.

4.2.4. Álgebra Lineal

Hay más operaciones matriciales que se pueden acceder con un submódulo de `numpy`, el `linalg`, por álgebra lineal. Como siempre, podemos checar este submódulo con `dir`:

```
In [1]:
print([i for i in dir(np.linalg) if i[0]!="_"])
Out[1]:
['LinAlgError', 'absolute_import', 'bench', 'cholesky', 'cond', 'det',
  'division', 'eig', 'eigh', 'eigvals', 'eigvalsh', 'info', 'inv', '
  lapack_lite', 'linalg', 'lstsq', 'matrix_power', 'matrix_rank', '
  multi_dot', 'norm', 'pinv', 'print_function', 'qr', 'slogdet', '
  solve', 'svd', 'tensorinv', 'tensorsolve', 'test']
```

Puedes checar con `help` para saber más sobre las condiciones para aplicar estas funciones, ¡y para qué sirven! Vamos a hacer las operaciones más conocidas con una matriz cuadrada:

```
In [2]:
C = np.array([[-1,2],[2,1]])          # matriz cuadrada de 2x2
c = np.array([4,5])                   # vector
print("det(C):", np.linalg.det(C))    # determinante de C
```

```
print("inversa de C:", np.linalg.inv(C))
# extrayendo eigenvalores y eigenvectores de C
val, vec = np.linalg.eig(C)
print("eigenvalores: ",val)
print("eigenvectores:",vec)
# resuelve la ecuación matricial Cx=c, encuentra el valor de x
print("x=", np.linalg.solve(C,c))
Out[2]:
det(C): -5.0
inversa de C: [[-0.2  0.4]
 [ 0.4  0.2]]
eigenvalores:  [-2.23606798  2.23606798]
eigenvectores: [[-0.85065081 -0.52573111]
 [ 0.52573111 -0.85065081]]
x= [ 1.2  2.6]
```

Ejercicios:

1. Dada la matriz $A = \begin{pmatrix} 0.4 & 0.2 \\ 0.5 & 0.8 \end{pmatrix}$, encuentra a) la inversa de la matriz, b) su determinante, y c) sus eigenvalores y eigenvectores.

2. Resuelve el siguiente sistema de ecuaciones usando lo visto en esta sección:

$$3x + y - 2z = 2, \quad 2x - y + 4z = 7, \quad -x + 2y - z = 3$$

4.3. Comparaciones

Cuando comparamos listas completas, tenemos una sola variable booleana como respuesta:

```
In [1]:
A= [[1,2,3],[4,5,6]]
B= [[1,2,3],[4,5,6]]
print(A == B)
Out[1]:
True
```

Sin embargo, cuando se comparan arreglos, tenemos como resultado un arreglo de booleanos, pues la operación, de nuevo, es pair-wise:

```
In [2]:
A= np.array(A)
B= np.array(B)
print(A == B)
Out[2]:
array([[ True,  True,  True],
       [ True,  True,  True]])
```

Esto puede generar errores por ejemplo si tenemos un `if` donde `Python` espera una sola variable como respuesta. En ese caso, se usa `np.all()` para saber si todos los elementos del arreglo cumplen con la condición:

```
In [3]:
print(np.all(A == B))
Out[3]:
True
```

También se cuenta con `np.any()`, para saber si al menos un elemento cumple con alguna condición:

```
In [4]:
A= np.array([1,2,3])
B= np.array([3,2,1])
print(A>B)
print(np.any(A>B))
Out[4]:
[False False  True]
True
```

Por broadcasting, también se pueden comparar un arreglo con un número:

```
In [5]:
print(A>5)
print(np.any(A>5))
Out[5]:
[False False False]
False
```

Vemos que cada elemento de `A` se comparó con el 5.

Ejercicios:

Crea un arreglo de tamaño 20 entre 0 y 1 con la función `random` del módulo `random`. Si todos los números del arreglo son mayores a 0.05, imprime "¡gané!". De otra forma, imprime "lástima Margarita".

4.4. Selección de elementos de un arreglo

4.4.1. Slicing

En lo que se refiere a seleccionar elementos adyacentes de un arreglo, cuando éste es de tipo vector se procede igual que en las listas, con el slicing, dando el inicio y final del índice del arreglo, recordando que el extremo final es abierto:

```
In [1]:
a = np.array([5,3,7,11,39])
print(a[1:3])
Out[1]:
[3 7]
```

Cuando los arreglos son de tipo matriz, hay algunas diferencias con las listas anidadas. En resumen, tenemos que, si la matriz en cuestión es `A`:

- `A[i]` da el renglón `i`, al igual que con las listas,

- pero a diferencia de las listas, la columna `j` se llama con `A[:,j]`. Con listas habría que llamar a un for loop para seleccionar a esos elementos,

- el elemento en el renglón `i` y columna `j` se llama con `A[i,j]`. En las listas, el análogo sería `A[i][j]`, que también funciona en los arreglos de `numpy`,

- para tener elementos del renglón `i`, inclusivo, al `j`, exclusivo, y columnas de la `k` a la `l`, como podemos imaginar, se usa `A[i:j,k:l]`:

```
In [2]:
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
print("A[2]:", A[2])      # tercer renglón (índice 2)
print("A[:,1]", A[:,1])   # segunda columna (índice 1)
# elemento del segundo renglón (índice 1) y tercera columna (índice 2)
print("A[1,2]:", A[1,2])
# renglones con índices 1 y 2, columnas con índices 0 y 1
print("A[1:3,:2]:",A[1:3,:2])
Out[2]:
A[2]: [7 8 9]
A[:,1] [2 5 8]
A[1,2]: 6
A[1:3,:2]: [[4 5]
 [7 8]]
```

Con esto en mente, vamos a hacer un ejercicio donde generemos una matriz llena de ceros de un cierto tamaño, y después modificamos sus entradas de acuerdo a una cierta condición:

```
In [3]:
A=np.zeros((3,3))      # matriz de ceros de 3x3
for i in range(3):
    for j in range(3):
        if j-i>=0:
            A[i,j]=(i+1)*(j+2)
print(A)
Out[3]:
array([[ 2.,  3.,  4.],
       [ 0.,  6.,  8.],
       [ 0.,  0., 12.]])
```

Ejercicios:

1. Dada la matriz $A = \begin{pmatrix} 4 & 8 & 11 \\ 25 & 3 & 8 \\ 2 & 6 & 7 \end{pmatrix}$, encuentra la submatriz que consiste en los 2 primeros renglones y las 3 columnas de la matriz A. Multiplica esta submatriz por los 3 últimos elementos del vector $a = (4, 8, 9, 11)$

2. Dada la matriz `np.random.random((20,3))`, selecciona la primera columna.

4.4.2. Selección con fancy indexing

Cuando los elementos de la selección no son contiguos, se puede dar una lista de los índices deseados, en cualquier orden. Esto es conocido como fancy indexing, que podríamos

traducir como indexación elegante:

```
In [1]:
a=np.array([5,7,6,1,7,0,4,6])          # un arreglo tipo vector
print("elementos del tercer, cuarto y último índice de a:")
print(a[[3,4,-1]])
A= np.array([[1,3,8],[12,11,3],[4,6,12]])     # una matriz
# para seleccionar 2 renglones, se necesita una lista, pero para
# seleccionar una sola columna un número es suficiente:
print("segunda columna y primer y tercer renglones de A:")
print(A[[0,2],1])
# para seleccionar todos los renglones se usa :,
print("los 3 renglones y la primera y tercera columnas de A:")
print(A[:,[0,2]])
Out[1]:
elementos del tercer, cuarto y último índice de a:
[1 7 6]
segunda columna y primer y tercer renglones de A:
[3 6]
los 3 renglones y la primera y tercera columnas de A:
[[ 1  8]
 [12  3]
 [ 4 12]]
```

Ejercicios:

Dada la matriz $A = \begin{pmatrix} -5 & 6 & 2 & -9 \\ 4 & 3 & 5 & 8 \\ 2 & -1 & 7 & -4 \\ -8 & 4 & 11 & 3 \end{pmatrix}$, selecciona a) la primera y última columnas, y b) el primer y último renglones.

4.4.3. Selección por condición

Cuando tenemos una lista, para seleccionar elementos dada una condición se requiere de un for loop que itere en toda la lista y un condicional para imprimir esos elementos. Pero con los arreglos podemos simplificar las cosas en algunos casos. Digamos que tenemos un arreglo de números y queremos quedarnos con los números impares o con los números mayores a un cierto umbral. Esto se logra fácilmente de la siguiente forma:

```
In [1]:
np.random.seed(3)
# 10 enteros aleatorios entre el 1 y el 29
A=np.random.randint(1,30,10)
print(A)
# selección de números impares:
print(A[A%2==1])
# selección de números menores que 20:
print(A[A<20])
Out[1]:
[11 25 26  4 25  9  1 22 20 11]
[11 25 25  9  1 11]
[11  4  9  1 11]
```

Cuando decimos, en el ejemplo, que $A < 20$, tenemos un arreglo lleno de `Trues` y `Falses`, como ya vimos en la sección de comparación de arreglos. Al hacer `A[A<20]`, seleccionamos los elementos donde $A < 20$ es `True`.

Si queremos más de una condición, se usan los operadores and, or y not, pero la notación es ahora con los símbolos `&`, |, y `~`, respectivamente. Las diferentes condiciones deben estar encerrada entre paréntesis:

```
In [2]:
print(A[(A<20) & (A%2==1)])  # números impares menores a 20
Out[2]:
array([11,  9,  1, 11])
```

Se pueden también seleccionar elementos de un arreglo poniendo una condición sobre otro arreglo:

```
In [3]:
np.random.seed(3)
# 25 enteros aleatorios entre el 0 y el 100
x=np.random.randint(0,101,25)
y=np.sin(x)
# seleccionamos las y's con las x's múltiplos de 5
print(y[x%5==0])
Out[3]:
[ 0.          -0.54402111  0.91294525  0.89399666]
```

Ejercicios:

1. Dada la lista de estudiantes

`A = np.array([‘‘Benito’’, ’’Pedro’’, ’’Alicia’’, ’’Carmen’’, ’’Jesús’’])`

y sus calificaciones respectivas `B = np.array([8,7,9,6,5])`, selecciona a los estudiantes que obtuvieron una calificación igual o mayor a 8 (es muy fácil, pero hazlo con `numpy`)

2. Crea un arreglo de 500 números aleatorios entre el 1 y el $1,000$ y selecciona los múltiplos de 3 cuyas raíces cúbicas sean mayor o igual a 8.

4.5. Operaciones de agregación

Ya hemos visto que `numpy` tiene muchas funciones que, al aplicarlas a un arreglo, retorna los resultados de todos los elementos del arreglo:

```
In [1]:
A=np.pi*np.array([[1/2,1/4],[1,1/6]])
print("sen(A) sin redondear:")
print(np.sin(A))
print("redondeando a 4 decimales:")
print(np.round(np.sin(A),4))
Out[1]:
sen(A) sin redondear:
[[1.00000000e+00 7.07106781e-01]
 [1.22464680e-16 5.00000000e-01]]
```

```
redondeando a 4 decimales:
[[1.      0.7071]
 [0.      0.5   ]]
```

Además de estas funciones, Python cuenta con funciones de agregación, que actúan sobre todos o un conjunto de los datos para dar como resultado un número. Por ejemplo, querríamos sumar los datos o sacar su promedio. Estas operaciones son más comunes en tablas de datos, como en Excel, nada más que con Python uno tiene la ventaja de programar cosas más específicas y hacer gráficas de una manera más fácil y rápida. Hay un módulo más ad-hoc para trabajar con tablas de datos, que es el pandas, el cual maneja bien archivos con datos faltantes y el cual mostraremos más adelante. Por mientras veamos cómo manejar estas operaciones en numpy.

4.5.1. Operaciones por renglón o por columnas

Digamos que tenemos unos datos con la calificación de tres exámenes de cinco estudiantes, donde cada renglón tiene las calificaciones de un estudiante en particular, y queremos hacer estadísticas por persona y por examen:

```
In [1]:
# calificación de 5 estudiantes en 3 exámenes:
A= np.array([[7.2,8.5,6.9], [10,10,9.5], [6,7.2,4.9], [6.2,7,6.8],\
    [9.5,10,9.3]])
```

Si sacamos el promedio, con la función mean (hay 2 formas de hacerlo):

```
In [2]:
print(np.mean(A))
print(A.mean())
Out[2]:
7.933333333333334
7.933333333333334
```

Esto está muy bien, pero lo que queremos es encontrar el promedio de cada estudiante, o el promedio de los resultados de cada examen por separado, no el promedio de todos los números. Resulta que, cuando estamos escribiendo en spyder los paréntesis de la función, vemos que uno de los argumentos es axis=None. Lo que nos dice axis es qué eje debe colapsarse para hacer la operación, y si por default no está ninguno de los dos ejes, es None, se toma el promedio de todos los números.

La notación es como sigue, axis=0 significa que vamos en la dirección de los renglones, de abajo hacia arriba, sumando los números y promediando. El resultado es que se “colapsan” los renglones y obtenemos el promedio de las columnas. Por otro lado, axis=1 significa que vamos en la dirección de las columnas, de izquierda a derecha, resultando en el “colapso” de las columnas y terminando con el promedio de los renglones. Así, si queremos saber el promedio de los estudiantes escogemos axis=1, y para el promedio de cada examen, escogemos axis=0:

```
In [3]:
print(A.mean(axis=0))
print(A.mean(axis=1))
Out[3]:
```

```
[7.78 8.54 7.48]
[7.53333333 9.83333333 6.03333333 6.66666667 9.6        ]
```

El mejor promedio es el del segundo estudiante, y el examen donde les fue mejor a los estudiantes, en promedio, fue el segundo.

Ejercicios:

1. Dada la matriz $\begin{pmatrix} 450 & 221 & 114 \\ 325 & 324 & 628 \\ 836 & 716 & 902 \end{pmatrix}$, encuentra la suma, el promedio, la mediana y la desviación estándar de los datos de cada columna. Además de la función `mean()`, usa las funciones `sum()`, `median()` y `std()`.

2. Dada la matriz

```
np.random.seed(8)
A=np.random.random((3,10))
```

Encuentra los valores máximos y mínimos por renglón. Encuentra también los índices donde se encuentran esos números. Para ello, además de las funciones `max()` y `min()`, usa las funciones `argmax()` y `argmin()`.

3. Dadas las matrices $A = \begin{pmatrix} 1 & 3 \\ 7 & 9 \end{pmatrix}$, y $B = \begin{pmatrix} -6 & 1 \\ 3 & 4 \end{pmatrix}$, concatena las matrices primero para tener una nueva matriz de tamaño 2×4, y después para tener una matriz de tamaño 4×2. Usa la función `concatenate()` para ello.

5 Imágenes y gráficas

Uno de los requerimientos más importantes en el área de Ciencia y Tecnología es la graficación de funciones. Hay diversos softwares gratuitos, como GeoGebra, que son de mucha utilidad y son muy buenos para esta tarea. Para que, además, podamos graficar los resultados de nuestra programación, así como para realizar operaciones más complejas, hay softwares especializados que son de paga y que la comunidad académica usa ampliamente. `Python` cuenta con las funcionalidades para desempeñar este trabajo de manera gratuita. Los objetivos de aprendizaje de este capítulo son que el lector conozca las funciones que existen para graficar, en 2D y 3D, puntos, líneas, proyecciones y superficies, y las aplique en diversos ejercicios. Que conozca también las funciones que se necesitan para graficar datos estadísticos y resuelva algunas situaciones. Que el lector sepa leer y manipular imágenes, de vital importancia por ejemplo para las imágenes médicas, entre otras aplicaciones. Que programe autómatas celulares. Que aprenda a hacer pequeñas animaciones. Para ahondar en la parte teórica de la graficación de funciones en 3D y las proyecciones en el plano, en la bibliografía se recomienda el libro de Cálculo de James Stewart (2006), y para ahondar más en el módulo de graficación de `matplotlib.pyplot` se recomienda el manual en Ciencias de Datos en Python de Jake VanderPlas (2016)

Como nota importante para visualizar las imágenes, en `spyder`, la forma por default de desplegarlas es de manera estática, "en línea", para que se desplieguen todas las imágenes en la ventana superior derecha. La otra forma de desplegar imágenes es la interactiva, "automático", donde aparece una ventanita aparte y uno puede interactuar con la figura, hacer zoom y otras. De las dos formas las figuras se pueden salvar fácilmente. En la forma estática se da click en el botón derecho del ratón y se escoge la opción para salvar, y en la interactiva está el símbolo para salvar la imagen. Para pasar de una opción a otra, en `spyder` uno va a Herramientas → Preferencias → Terminal de IPython → Gráficas, y en el cuadro de Salida, se escoge "Automático" o "En línea". Para que el cambio se lleve a cabo al momento, hay que matar la terminal que se está usando, con el botón de la cruz roja, o, se cierra y se vuelve a abrir `spyder`. El modo automático es más útil para gráficas en 3D para poder rotar, o para pequeñas animaciones.

Aunque no se diga explícitamente en algunos ejercicios, para todas las gráficas y cuando se requiera de `numpy`, hay que importar los 2 módulos con `import numpy as np`, e `import matplotlib.pyplot as plt`.

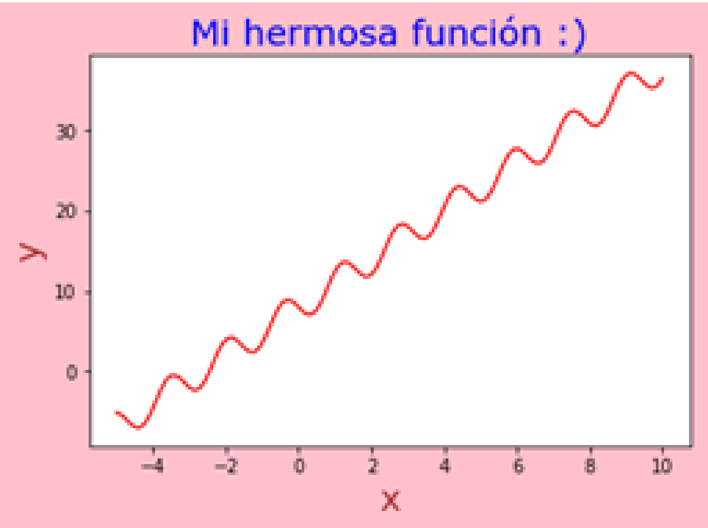


Figura 5.1: Gráfica con plot

5.1. Matplotlib.pyplot

5.1.1. Gráficas en dos dimensiones: plot y scatter

El módulo más tradicional para graficar en Python es el `matplotlib.pyplot`, el cual tiene como alias `plt`. Aquí, `pyplot` es un submódulo de `matplotlib`. La primera forma de graficar que veremos es la llamada "estilo Matlab". La receta para graficar en 2D es escribir las x's en una lista o arreglo de `numpy`, seguido de las y's en otra lista o arreglo, y otros argumentos optativos. Aparte se colocan el título y el nombre de los ejes, como primeras acciones básicas. Por ejemplo (Figura 5.1):

```
In [1]:
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(-5,10,100)
y=3*x-2*np.sin(4*x)+8
plt.figure(facecolor="pink")
plt.plot(x,y,c="r")
plt.xlabel("x",color="brown",size=20)
plt.ylabel("y",color="brown", size=20)
plt.title("Mi hermosa función :)", color="b", size=20,\
 family="Verdana")
plt.show()
```

Como siempre, para ver qué hace cada función usamos `help()`, por ejemplo, `help(plt.figure)`.

Vamos a ver cada función:

`plt.figure()` se coloca al inicio de cada figura, pare evitar que, al graficar, se empalme con la anterior. Como argumento, se puede poner el tamaño de la figura, `figsize=(10,8)`, por decir, el color de fondo, `facecolor=’’pink’’`, entre otras.

`plt.plot()` es la función principal para graficar. Tiene como argumentos la secuencia

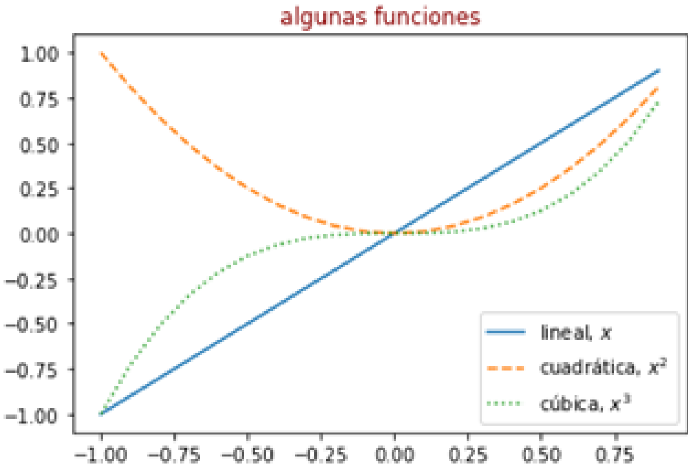


Figura 5.2: Varias funciones en una misma figura

de números x's, la secuencia de números y's, y como opciones el `color` o `c`, que puede recibir cualquier notación de color, como nombres, "red", sistema RGB, $(0, 1, 0)$, etc. Para tener una lista de nombres de colores, se puede checar https://es.wikipedia.org/wiki/Colores_web. También aquí va el estilo de línea `linestyle` o `ls=’’dashed’’`, etc.

`plt.xlabel()` y `plt.ylabel()` son para darle nombre a los ejes, con argumentos optativos el tamaño de la letra, `fontsize` o `size`, el tipo de letra, `family`, entre otras.

`plt.title()`, para el título, con argumentos similares a `plt.xlabel()`.

`plt.show()` es para mostrar la figura. No es obligatoria en `spyder`, la imagen se muestra de cualquier forma, pero, si se grafican varias figuras en un mismo programa, si se no se pone `show()`, las figuras se van almacenando en memoria y se muestran hasta al final, es mejor agregar esta función.

Si se quieren más de dos curvas en la misma figura, es conveniente poner una etiqueta para cada gráfica, con `plt.legend()` y `label`. Además, en caso de tener que imprimir en blanco y negro, sería necesario distinguir las curvas, por lo que agregaríamos el tipo de línea con `ls` o `linestyle`, que soporta los tipos `‘‘dotted’’ (‘‘-‘‘)`, `‘‘dashed’’ (‘‘--’’)`, `‘‘dotted’’ (‘‘:’’)` y `‘‘dashdot’’ (‘‘-.’’)`:

```
In [2]:
x=np.arange(-1,1,0.1)
ys=[x, x**2, x**3]
etiquetas=["lineal, $x$", "cuadrática, $x^2$", "cúbica, $x^3$"]
líneas = ["-","--",":"]
plt.figure()
for y, eti, lin in zip(ys, etiquetas, líneas):
    plt.plot(x, y, label=eti, ls=lin)
plt.legend()
plt.title("algunas funciones",color="darkred")
plt.show()
```

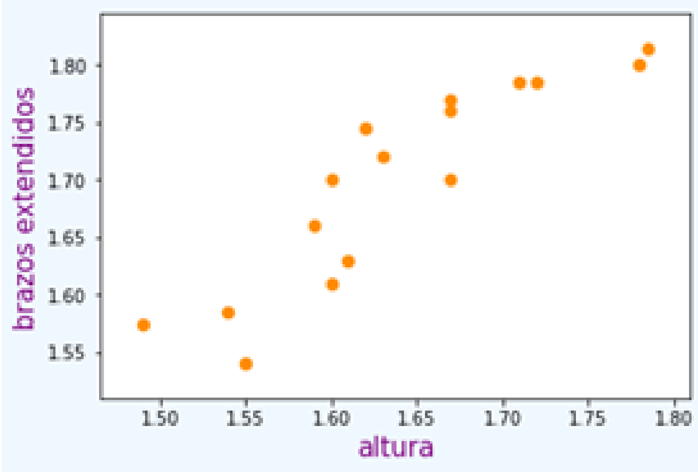


Figura 5.3: Graficar puntos con scatter

Vemos en este ejemplo que los textos soportan LaTeX. También, no hay que especificar necesariamente el color, por cada gráfica `matplotlib` selecciona un color diferente (Figura 5.2).

Para que en lugar de líneas grafiquemos puntos, como datos, en lugar de `plot()` usamos `scatter()`, con los mismos argumentos aunque ahora podemos agregar el tamaño de los puntos, con `s`, de `size`, y, podemos agregar la forma de los puntos, con `marker` (también disponible en `plot`, pero tiene más sentido en `scatter`).

Por ejemplo, veamos algunas medidas del largo de los brazos extendidos de unas personas contra sus alturas. Según Vitrubio, el cociente de estos dos números es idealmente igual a uno y por tanto esperamos una línea recta con pendiente de alrededor de 1. Según los datos, tenemos:

```
1  In [3]:
2  datos= np.array([(1.67,1.77),(1.67,1.76),(1.60,1.70),(1.63,1.72),\
3  (1.55,1.54),(1.60,1.61),(1.59,1.66),(1.785,1.815),(1.62,1.745),\
4  (1.61,1.63),(1.54,1.585),(1.72,1.785),(1.71,1.785),(1.49,1.575),\
       (1.67,1.70),(1.78,1.80)])
5  plt.figure(facecolor="aliceblue")
6  # la primera columna de datos son las x's, la segunda las y's
7  plt.scatter(datos[:,0], datos[:,1], s=40, marker="o",c= "darkorange")
8  plt.xlabel("altura",color="purple",size=15)
9  plt.ylabel("brazos extendidos",color="purple",size=15)
10 plt.show()
```

Con los pocos datos se puede ver (Figura 5.3) que sí hay un crecimiento lineal, aunque en general el largo de los brazos extendidos en un poco mayor que la altura.

Algunos de las formas de los puntos disponibles son ``D'' -diamante, ``s'' -cuadrado, ``*'' -estrella, ``^'' -triángulo, ``o'' -círculo, y varias más que se pueden encontrar en https://matplotlib.org/api/markers_api.html.

Hay que remarcar en que hay que tener cuidado con las abreviaciones; `c` o `color` está permitido dentro de `plot`, para la gráfica, pero para las cadenas se usa `color`, solamente. Lo mismo con `size`, éste se usa para las cadenas, mientras que `s` se usa para el tamaño

de los puntos. Para los colores, las abreviaciones permitidas son “r”, “g”, “b”, “c”, “m”, “y”, “k” y “w” para el rojo, verde, azul, cian (turquesa), magenta, amarillo, negro y blanco.

Ejercicios:

Grafica en la misma figura los puntos de las funciones

$$\sin(0.5x), \quad 3\cos(2x) \quad \text{y} \quad 4\sin\left(x - \frac{\pi}{4}\right)$$

en el intervalo $(0, 5\pi)$ (da unos 300 puntos para cada gráfica, para ello usa `linspace`). Aplica diferentes tipos de puntos y dale a cada gráfica una etiqueta. Para ello usa un `for` con `zip` que tome 3 listas, una para las funciones, otra para el `label`, y otra pare el `marker`. Agrega título y nombres a los ejes.

5.1.2. Subfiguras y gráficas con estilo orientado a objetos

Para graficar más de una figura en el mismo cuadro se usa la función `subplot`, con 3 argumentos, el número de renglones y el número de columnas, que quedan fijos, y el número consecutivo de la figura, que se lee de izquierda a derecha y de arriba a bajo, y cuya numeración empieza con 1, no con 0. Por ejemplo, se pueden hacer 6 figuras en 2 renglones y 3 columnas como sigue (Figura 5.4):

```
1  In [1]:
2  x= np.arange(0,6.5,0.2)
3  plt.figure()
4  for i in range(1,7):
5      plt.subplot(2,3,i)
6      plt.plot(x,np.sin(i*x),color="dodgerblue")
7      plt.xlabel("x",color="b",fontsize=14)
8      plt.ylabel("sin("+str(i)+"x)",color="b",fontsize=14)
9  plt.tight_layout()
10 plt.show()
```

El `plt.tight_layout()` es necesario para que los nombres de los ejes no queden superpuestos con las subfiguras de al lado. Pudieron haber sido 5 subfiguras en este ejemplo, y en ese caso el último cuadro quedaría vacío. Hay que hallar la configuración óptima de renglones y columnas para dar el mejor resultado visual.

Hay otra función muy popular para hacer subfiguras que es `subplots()`, en plural, sólo que esta otra función en realidad es la que se utiliza en la segunda forma que tiene `matplotlib.pyplot` para graficar, y que se refiere como el “estilo orientado a objetos”. En esta otra forma de graficar, en lugar de llamar a las funciones directamente con `plt`, se crea una figura y ejes y, con éstos últimos, que hacen de “objeto”, se llaman a las funciones. Veamos un ejemplo primero donde se grafica una sola figura, usando `subplots()` (Figura 5.5):

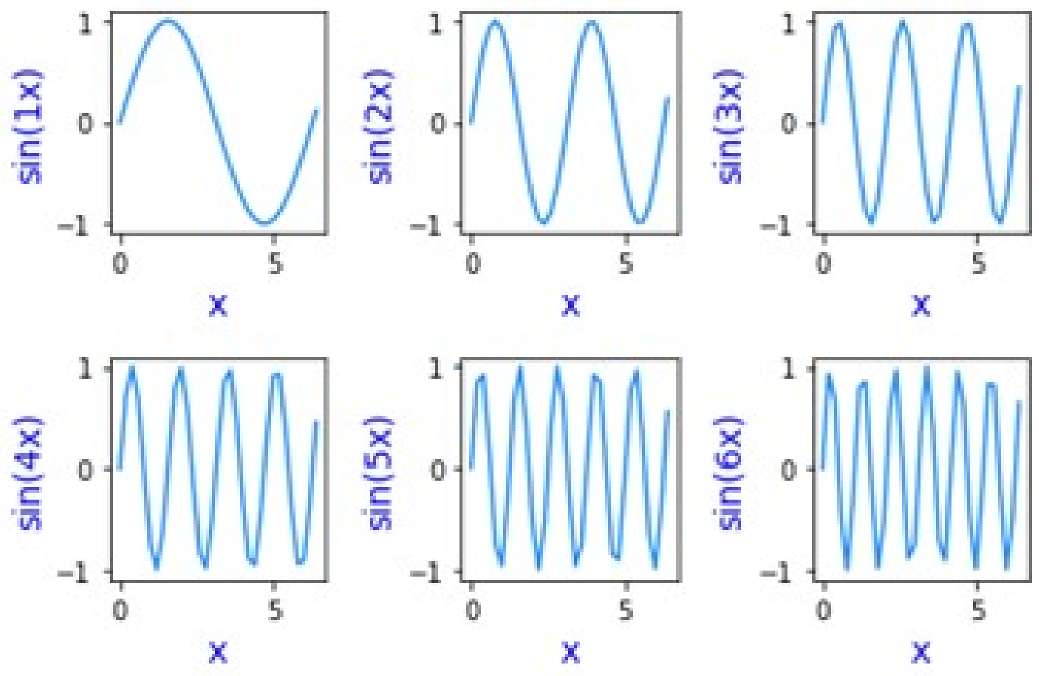


Figura 5.4: Subfiguras con subplot

```
In [2]:
x=np.arange(-np.pi,2*np.pi,np.pi/24)
y=3*np.sin(2*x)-np.cos(x)
fig, ax= plt.subplots(figsize=(8,6),facecolor="aliceblue")
ax.plot(x,y,color="dodgerblue")
ax.set_xlabel("x",fontsize=20,color="r")
ax.set_ylabel("y",fontsize=20,color="r")
ax.set_title("Con 'ax'",fontsize=20,color="r")
plt.show()
```

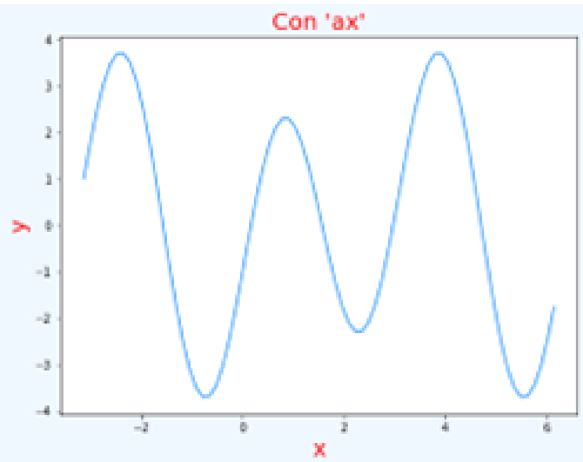


Figura 5.5: Gráfica con estilo orientado a objetos (subplots)

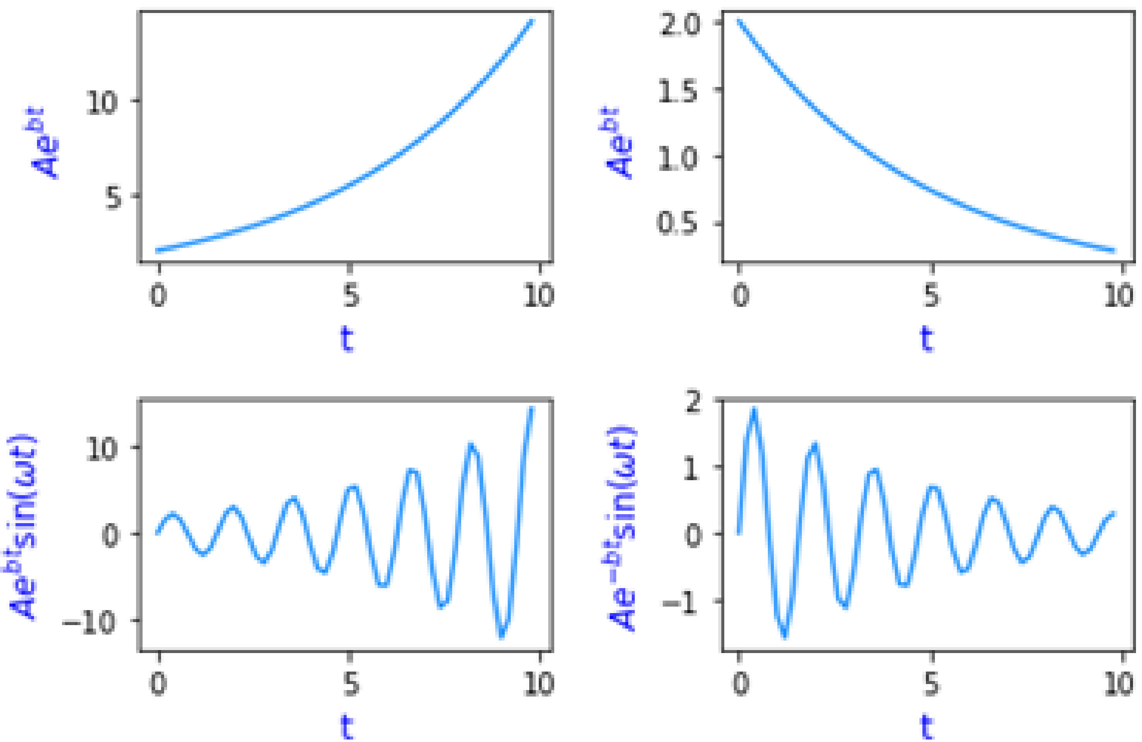


Figura 5.6: Subfiguras con plt.subplots

En este ejemplo vemos que, aunque existen las mismas funciones llamando a `ax` o a `plt`, por ejemplo `plot`, hay otras funciones, como las de etiquetar los ejes o dar el título, donde se coloca el prefijo `set` en el modo orientado a objetos. Veamos otro ejemplo donde hay subfiguras. Desde el inicio se da el número de renglones y columnas en `subplots`, y, para acceder a cada subfigura se llama a `ax` y se dan las coordenadas [renglón,columna] de la subfigura, con la numeración empezando desde 0, no desde 1 (Figura 5.6).

```
1  In [3]:
2  x=np.arange(0,10,0.2)
3  fig, ax= plt.subplots(2,2)
4  ax[0,0].plot(x,2*np.exp(0.2*x),color="dodgerblue")
5  ax[0,0].set_xlabel("t",color="b",fontsize=12)
6  ax[0,0].set_ylabel("$Ae^{bt}$",color="b",fontsize=12)
7  ax[0,1].plot(x,2*np.exp(-0.2*x),color="dodgerblue")
8  ax[0,1].set_xlabel("t",color="b",fontsize=12)
9  ax[0,1].set_ylabel("$Ae^{bt}$",color="b",fontsize=12)
10 ax[1,0].plot(x,2*np.exp(0.2*x)*np.sin(4*x),color="dodgerblue")
11 ax[1,0].set_xlabel("t",color="b",fontsize=12)
12 ax[1,0].set_ylabel("$Ae^{bt}\sin(\omega t)$",color="b",fontsize=12)
13 ax[1,1].plot(x,2*np.exp(-0.2*x)*np.sin(4*x),color="dodgerblue")
14 ax[1,1].set_xlabel("t",color="b",fontsize=12)
15 ax[1,1].set_ylabel("$Ae^{-bt}\sin(\omega t)$",color="b",fontsize=12)
16 plt.tight_layout()
17 plt.show()
```

plt	ax
plt.figure()	fig, ax= plt.subplots()
plt.subplot(r,c,i) plt.plot()	fig, ax= plt.subplots(r,c) ax[i,j].plot()
plt.plot(), plt.scatter() plt.legend(), plt.hist(), etc.	ax.plot(), ax.scatter() ax.legend(), ax.hist(), etc.
plt.xlabel(), plt.ylabel() plt.title(), etc.	ax.set_xlabel(), ax.ylabel() ax.set_title(), etc.

Tabla 5.1: Dos formas de graficar, tipo Matlab y tipo orientado a objetos

En este texto vamos a seguir usando cuando se pueda la forma más directa de graficar, con `plt`, pero como referencia en la Tabla 5.1 se presentan las principales diferencias entre las dos formas.

5.1.3. Histogramas, hist

El histograma se usa para visualizar distribuciones de variables numéricas, es decir, cuando tenemos una variable y queremos ver con que frecuencia aparecen los datos en diferentes subintervalos. La función a utilizar es `hist`, con argumento una lista o arreglo de números, y algunas opciones, como el número de barras que se selecciona, con bins, color, etc. Por ejemplo, veamos la distribución de masas de un grupo de estudiantes universitarios, todos hombres (Figura 5.7):

```
In [1]:
A= [49.5, 90, 89.5, 77, 78, 54.5, 110.5, 65, 65.5, 62, 60, 66.5,\
67.5, 98, 67, 73, 86, 67, 83, 75.5, 64.5, 66, 71.5, 82, 85, 61.5,\
102, 103, 77.5, 112,82]
plt.figure()
plt.hist(A,color="blue",bins=7)
plt.title("distribución masa estudiantes varones")
plt.show()
```

Figura 5.7: Un histograma con la función hist

Figura 5.8: Gráfica de barra con plt.bar

Ejercicios:

1. En tu casa, selecciona al menos 30 libros (de primaria, la Biblia, diccionarios, lo que encuentres), mide su largo y haz un histograma con estos datos.

5.1.4. Gráficas de barra: bar y barh

Las gráficas de barra sirven para visualizar la distribución de datos de tipo categórico, como la frecuencia de los colores favoritos de la gente. La función a utilizar es bar, y, a diferencia de los histogramas, como argumento damos el conjunto de datos, sin repetir, y las alturas o frecuencias de esos datos. Por ejemplo, tomemos los datos de un grupo de estudiantes a los que se les preguntó qué tipo de perro preferían entre los perros de a) trabajo (Labrador, Golden Retriever, Collie, Pastor Alemán, etc., perros inteligentes que se entrenan fácilmente para hacer un trabajo), b) intimidantes (Rottweiler, Pitbull, Boxer, etc.), c) compañía-pequeños (Chihuahua, Schnauzer, Pug, Pomeranian, etc.) y d) ninguno. Los datos de las estudiantes mujeres fueron:

```
In [1]:
A=["trabajo", "intimidante", "pequeño", "ninguno"]
muj=[19,9,17,7]
plt.figure()
plt.bar(A,muj,color="royalblue")
plt.title("tipo de perros preferidos por mujeres")
plt.show()
```

Los perros de trabajo y los pequeños fueron preferidos sobre los intimidantes, como era de imaginarse (Figura 5.8). ¿Y los datos de los estudiantes hombres? Para hacer la comparación, podemos graficar en la misma gráfica apilando dos grupos de barras, por cada género, pero primero normalizamos los números para tener proporciones pues el número de mujeres y hombres es diferente. Para poder tener las barras encimadas, hacemos la opción bottom, que por default es cero, igual a la altura de las barras previas:

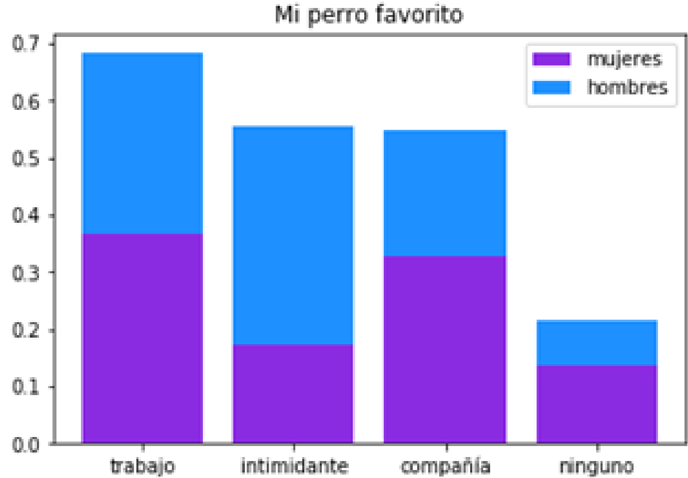


Figura 5.9: Gráfica de barras apiladas

```
1 In [2]:
2 # convertimos la lista en arreglo para dividir fá cilmente entre el
# nú mero de personas
muj= np.array(muj)/52              # 52 mujeres
hom= np.array([20,24,14,5])/63    # 63 hombres
plt.figure()
plt.bar(A, muj, label="mujeres", color="blueviolet")
plt.bar(A, hom, label="hombres", bottom=muj, color="dodgerblue")
plt.legend()
plt.title("Mi perro favorito")
plt.show()
```

En comparación, la proporción de hombres que prefieren los perros intimidantes es mayor que la de mujeres, y en términos globales, los perros de trabajo fueron los más votados (Figura 5.9).

¿Qué pasa si se tienen muchas variables categóricas? Los nombres se amontonan en el eje x y no se ve bien qué se está graficando. Hay opciones para rotar los nombres, pero hay una mejor opción que es la de las barras horizontales, que se llama con `barh` y que se usa de manera similar a `bar`. Por ejemplo, grafiquemos las horas de sueño de varios animales (Figura 5.10):

```
In [3]:
A= [("ratón",20.1), ("perezozo",20), ("tlacuache",19.4),\
 ("musaraña",16), ("ardilla",14), ("gato",13.2), ("león",13),\
  ("paloma",11.9),("delfín",10), ("chimpancé",10.8), ("perro",10.7),\
  ("hombre",8), ("cerdo",8.4), ("foca",6), ("elefante asiático",5.3),\
   ("vaca",5), ("oveja",4), ("caballo",2.9)]
x= [i[0] for i in A]      # los animales
y= [i[1] for i in A]      # las horas
plt.figure()
plt.barh(x,y,color="mediumslateblue")
plt.title("horas de sueño de animales")
plt.show()
```

Si se quieren gráficas apiladas, en lugar de la opción `bottom` se usa la opción `left`.

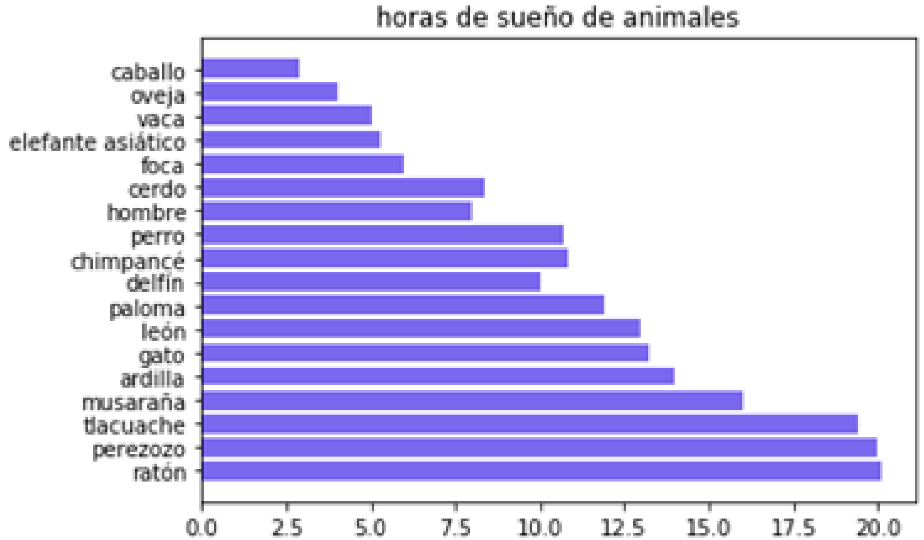


Figura 5.10: Gráfica de barras horizontal

Ejercicios:

Se le preguntó a un grupo de estudiantes universitarios si creían en un tipo de Dios. Las respuestas posibles fueron a) sí, soy creyente, b) no, soy ateo, c) no sé, tengo dudas existenciales. El número de respuestas por género fue:

	sí creo	no creo	no sé	total
mujeres	37	6	13	56
hombres	25	16	25	66

Haz una gráfica de barras apilada por género de las proporciones de los resultados.

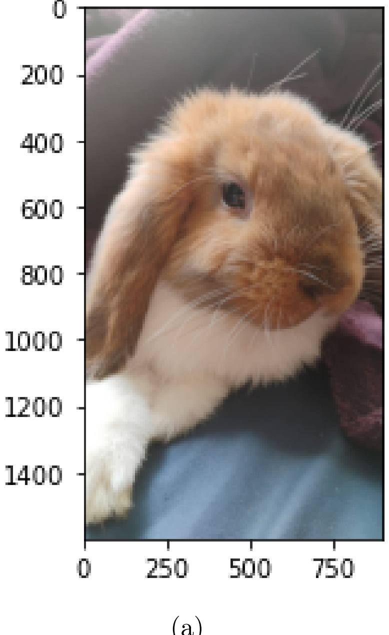
5.2. Imágenes y proyecciones en el plano

A continuación, veremos algunas funciones de `matplotlib` que sirven para describir el valor de una función que dependa de dos variables en el mismo plano, como es el caso de las curvas de nivel o de los heat maps, que son mapas donde a cada pixel le corresponde un color de acuerdo con una variable que se esté describiendo, como temperatura, tamaño de la población, etc. También veremos cómo leer imágenes, manipularlas y mostrarlas. Empezaremos con este último tema primero.

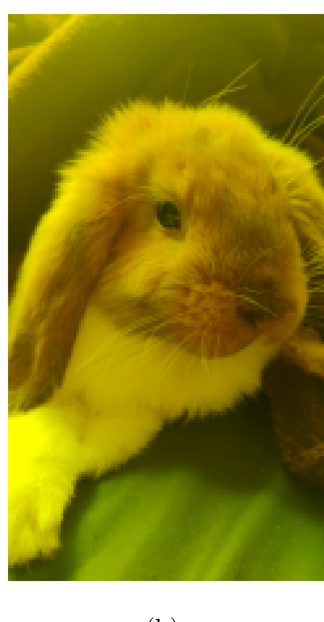
5.2.1. Leer y mostrar imágenes: imread e imshow

Para leer una imagen de tipo png, jpg, jpeg, bmp, etc., se usa `imread` (lee-imagen), que lee la información del archivo y la convierte en un arreglo de `numpy` Para mostrar la información del arreglo de `numpy`, se usa `imshow` (muestra-imagen). Esto quiere decir que, para mostrar algo como imagen, sólo basta tener un arreglo de `numpy` tipo matriz, con el valor de pixeles que nosotros inventemos.

Digamos que tenemos una imagen guardada en el mismo archivo de `Python` (para no



(a)



(b)

Figura 5.11: (a) Mostrar imágenes (Penélope) con plt.imshow, (b) Imagen de Penélope manipulada

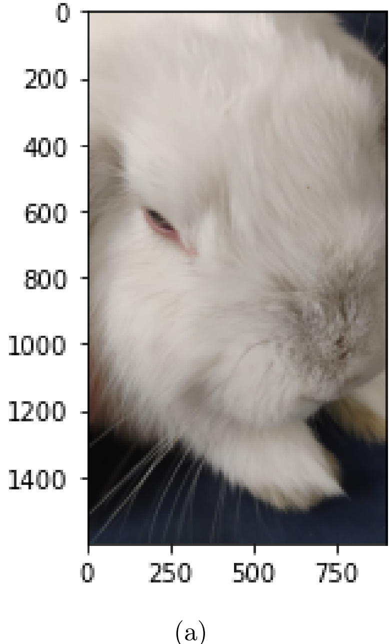
tener que escribir todo el path al llamar al archivo), entonces, para mostrarla en pantalla hacemos (Figura 5.11):

```
In [1]:
import matplotlib.pyplot as plt
penelopita= plt.imread("Penélope.jpeg")
plt.figure()
plt.imshow(penelopita)
plt.show()
```

En este ejemplo en particular, el arreglo de `numpy`, imagen, tiene tres dimensiones, las dos primeras para los renglones y columnas que conforman los pixeles, y una más porque en cada pixel no hay un simple número, sino una triada del sistema RGB de color, hay esa tercera dimensión para escoger el contenido de rojo, de verde o de azul, [R,G,B]. Si la figura fuera en grises, entonces sí nada más habría un número en cada pixel. También, en la tercera dimensión puede haber 4 números, el último para la transparencia `alpha`, si se usa el sistema RGBA.

Veamos qué pasa si quitamos el contenido de azul de la imagen (Figura 5.11):

```
In [2]:
# hacemos una copia de la imagen, y en todos los pixeles de la copia
# hacemos 0 el contenido de azul (índice = 2)
penelopita2= np.copy(penelopita)
penelopita2[:,:,2] = 0
plt.figure()
plt.imshow(penelopita2)
plt.axis("off")
plt.show()
```



(a)

(b)

Figura 5.12: (a) Imagen de Blanca Nieves, (b) Imagen de Blanca Nieves manipulada

Quitamos de una vez los ejes con `plt.axis(“off”)`.

Carguemos otra imagen para hacer más experimentos (Figura 5.12):

```
In [3]:
blancanieves=plt.imread("BlancaNieves.jpeg")
plt.figure()
plt.imshow(blancanieves)
plt.show()
```

Podemos cambiar de color a BlancaNieves para ponerla de color azul. Para ello, como los colores blancuzcos se representan con colores con contenido parecido de rojo, verde y azul, y de intensidad alta (en una escala del 0 al 255, o del 0 al 1), y los colores negruzcos igualmente tienen contenidos similares de los tres colores, pero en intensidad baja, podemos seleccionar pixeles donde los tres canales de color tengan una intensidad mayor a un cierto valor, para cambiar su color. Vamos a usar `np.all()` para comparar los 3 canales de color en todos los pixeles, y obtener una sola variable booleana (Figura 5.12):

```
In [4]:
n, m, k= blancanieves.shape     # tamaño de la imagen
azulnieves= np.copy(blancanieves)
for i in range(n):
    for j in range(m):
        if np.all(blancanieves[i,j]>= [100,100,100]):
            azulnieves[i,j]= [110,50,255]
plt.figure()
plt.imshow(azulnieves)
plt.axis("off")
plt.show()
```

(1, 0, 0) rojo	(0, 1, 0) verde	(0, 0, 1) azul	(1, 1, 0) amarillo
(1, 0, 1) magenta	(0, 1, 1) cian	(1, 1, 1) blanco	(0.5, 0.5, 0.5) gris
(0, 0, 0) negro	(1, 0.5, 0) naranja	(0.5, 0, 1) morado	(1, 0, 0.5) salmón
(0, 0.5, 1) azul pitufo	(0, 0, 0.5) azul oscuro	(0, 0.5, 0) verde oscuro	(0.5, 0, 0) rojo oscuro

Tabla 5.2: Tabla de colores con combinaciones RGB

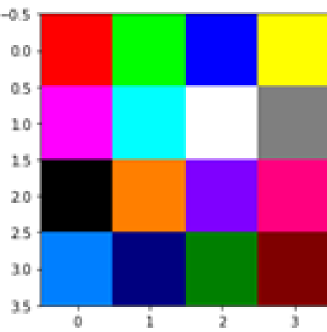


Figura 5.13: Mostrar nuestras propias imágenes usando numpy e imshow

Creemos ahora nuestro propio arreglo de `numpy` para dejar más en claro el hecho de que las imágenes son matrices con triadas RGB como entradas. Vamos a poner diferentes combinaciones de 1, 0 y 0.5 (puede ser cualquier fracción) a cada color y ver el resultado (Figura 5.13). Como recordatorio de óptica, en la Tabla 5.2 se muestran los resultados de sumar luces de diferentes colores con mayor o menor intensidad (los verdaderos nombres y combinaciones se encuentran en la web):

```
In [5]:
import numpy as np
import matplotlib.pyplot as plt
im1= np.array([[[1,0,0], [0,1,0], [0,0,1], [1,1,0]],\
               [[1,0,1], [0,1,1], [1,1,1], [0.5,0.5,0.5]],\
               [[0,0,0], [1,0.5,0], [0.5,0,1], [1,0,0.5]],\
               [[0,0.5,1], [0,0,0.5], [0,0.5,0], [0.5,0,0]]])
plt.imshow(im1)
plt.show()
```

Ejercicios:

Lee una figura que descargues en el mismo directorio de un archivo `Python`, encuentra su tamaño con `shape`, divide mentalmente la figura en cuatro cuadrantes y muestra en pantalla la subfigura que corresponda a tomar el cuadrante superior izquierdo.

5.2.2. Heat maps y curvas de nivel: imshow, contour y pcolormesh

Siguiendo con `imshow`, en esta ocasión la vamos a usar para mostrar curvas de nivel de una función de dos variables. Como vamos a proyectar la función en el plano, no va a ser suficiente dar puntos a lo largo del eje x y del eje y, pues la función proyectada estaría nada más en la cruz formada por los ejes. Para poder mapear a toda la región

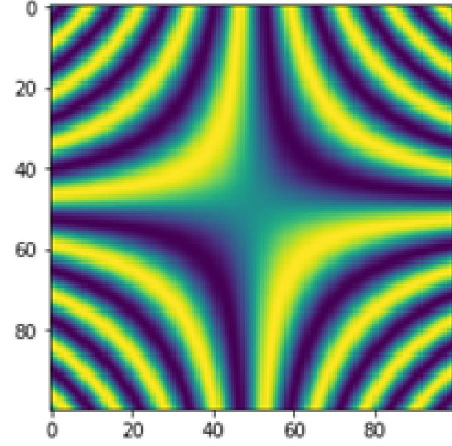


Figura 5.14: Proyección en el plano de $z = \sin xy$ con imshow

vamos a usar la función `meshgrid`, que hace una malla de puntos de las intersecciones de las x's y y's que demos. Entonces, digamos que queremos mapear la función $\sin(xy)$:

```
In [1]:
x= np.linspace(-5,5,100)  # 100 puntos en el intervalo (-5,5)
y= np.linspace(-5,5,100)
X,Y= np.meshgrid(x,y)
z= np.sin(X*Y)
plt.figure()
plt.imshow(z)
plt.show()
```

Vamos a comentar algunas cosas. Aunque los intervalos que escogimos fueron de $(-5, 5)$ en los dos ejes, la figura muestra otros intervalos. También, aunque no sea claro en la Figura 5.14, ésta está invertida, de arriba a abajo. Finalmente, no obtuvimos una figura en grises, a pesar de no haber dado una tercera dimensión para el contendido RGB, y, por default, nos salió esa combinación de colores, que podemos modificar.

Como ya vimos, `imshow` muestra imágenes de formatos png, etc. Pero, la notación usual de las imágenes es comenzar con el cero en la esquina superior izquierda, e incrementar la cuenta de los pixeles hacia abajo y hacia la derecha. Por ello, hay que invertir el orden de los renglones para nuestros propósitos, seleccionando `origin=''lower''`. Luego, escogimos 100 puntos para cada eje, y eso es lo que vemos en la figura, donde se cuenta el número de pixeles. Hay que meter con la mano el intervalo deseado con `extent[xmin,xmax,ymin,ymax]`. Sobre la versión de los colores, podemos escoger una gama variada de `colormaps` con la función `cmap`. La versión por default es `''viridis''`, y la gama completa está en
https://matplotlib.org/stable/users/explain/colors/colormaps.html. Vamos a darle un toque de frozen a la figura escogiendo la versión `''cool''`. Finalmente, se puede agregar una barra para indicar el valor de la función dependiendo del color, con la función `colorbar` (Figura 5.15):

```
In [2]:
plt.figure()
plt.imshow(z, extent=[-5,5,-5,5], origin="lower", cmap="cool")
```

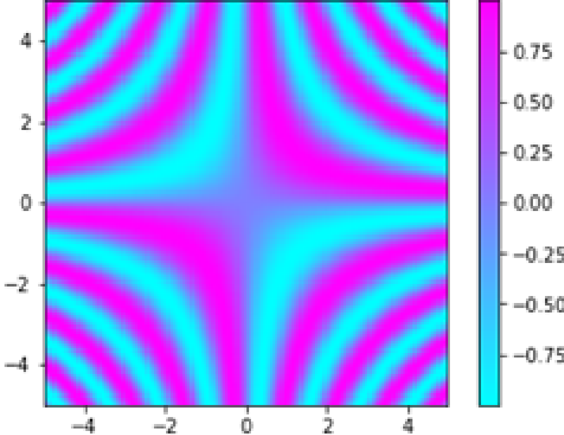


Figura 5.15: Proyección en el plano de $z = \sin xy$, con barra de colores y colormap cool

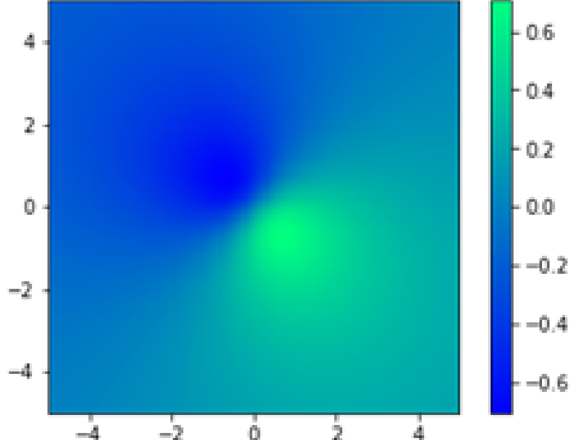


Figura 5.16: Proyección en el plano de $z = \frac{x-y}{1+x^2+y^2}$ con colormap winter

```
4 plt.colorbar()
5 plt.show()
```

En este ejemplo, quedan claras las curvas de nivel, pero, ¿siempre es así?, intentemos otra función:

```
1 In [3]:
2 x= np.linspace(-5,5,100)   # 100 puntos en el intervalo (-5,5)
3 y= np.linspace(-5,5,100)
4 X,Y= np.meshgrid(x,y)
5 z= (X-Y)/(1+X**2+Y**2)
6 plt.figure()
7 plt.imshow(z, extent=[-5,5,-5,5], origin="lower", cmap="winter")
8 plt.colorbar()
9 plt.show()
```

La Figura 5.16 queda muy bonita, parece que 2 cometas se van a encontrar. De la barra de colores vemos que la zona con el verde más claro y brillante es donde hay un máximo e, igualmente, el azul más brillante es para donde hay un mínimo. Pero las fronteras

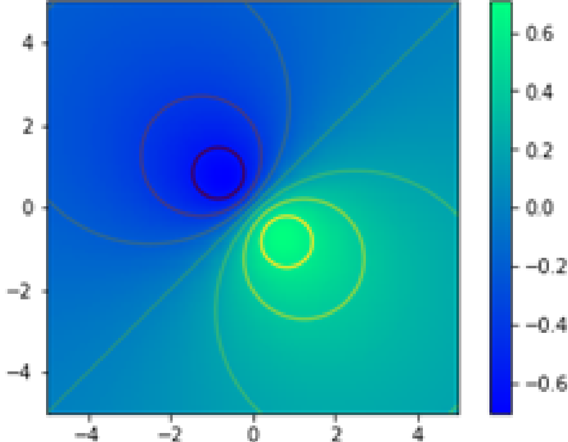


Figura 5.17: Proyección en el plano y curvas de nivel de $z = \frac{x-y}{1+x^2+y^2}$

están tan difuminadas que no parece que haya curvas de nivel. En realidad, la gráfica que obtuvimos es un "mapa de calor", heat map, que sí nos da idea sobre las curvas de nivel en algunos de los casos, pero para ese propósito en específico está contruida la función `contour`, que se puede usar sola o junto con `imshow` (o alguna otra función), para el delineado claro de las curvas. Además de z, se debe agregar la malla de puntos X y Y como argumentos de `contour` (Figura 5.17):

```
In [4]:
plt.figure()
plt.imshow(z, extent=[-5,5,-5,5], origin="lower", cmap="winter")
plt.colorbar()
plt.contour(X,Y,z)
plt.show()
```

Otra función que tiene el mismo rol que `imshow` en las gráficas heat map es `pcolormesh`, que es más fácil de usar pues no hay que invertir la imagen, y no hay que especificar el intervalo de valores (Figura 5.18):

```
In [5]:
x= np.linspace(-5,5,100)  # 100 puntos en el intervalo (-5,5)
y= np.linspace(-5,5,100)
X,Y= np.meshgrid(x,y)
z= np.sin(X)-np.sin(Y)
plt.figure()
plt.pcolormesh(X,Y,z, cmap="rainbow")
plt.colorbar()
plt.contour(X,Y,z)
plt.show()
```

Ejercicios:

Haz las gráficas de las curvas de nivel de las funciones

$$a)\ z = y^2 - x^2, \quad b)\ z = (x^2 - y^2)^2, \quad y \quad c)\ z = |x| + |y|$$

usa `contour` junto con `pcolormesh` o `imshow`.

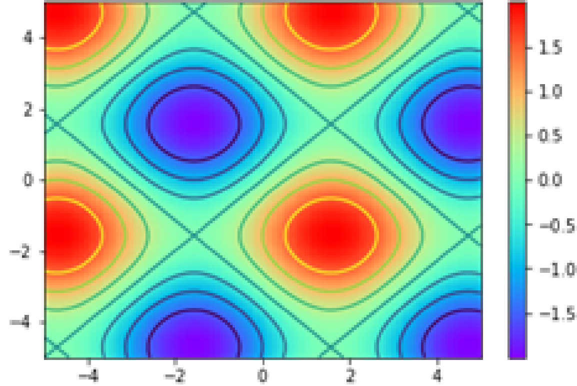


Figura 5.18: Proyección y curvas de nivel de la función $z = \sin x - \sin y$, con pcolormesh

5.3. Gráficas en tres dimensiones

Para graficar en 3 dimensiones, o para hacer otro tipo de gráficas, como polares, vamos a usar la forma de graficar "orientada a objetos", pues no hay el equivalente con la forma `plt` estándar. Sin embargo, en lugar de usar la función `subplots` como en 2D, vamos a llamar a `axes` (¡no `axis`!) directo, o a `gca`. También, como estamos en 3D, es preferible la versión interactiva para mostrar las gráficas, porque así se pueden rotar. Checa el incio del capítulo para ello. Además del `matplotlib.pyplot` hay que llamar a la función `mplot3d` del módulo `mpl_toolkits`.

5.3.1. Puntos en 3D: scatter3D

Empezemos graficando algunos puntos. En lugar de `scatter`, la función a usar es `scatter3D`. Para los puntos, generamos por ejemplo algunos números aleatorios de una distribución normal en 3D (Figura 5.19):

```
In [1]:
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import numpy as np
# damos una seed para reproducir los resultados:
np.random.seed(7)
# 200 puntos en 3D, con coordenadas de una distribución normal
# con mu=(1,2,3) y sigma=(4,1,5)
datos= np.random.normal([1,2,3], [4,1,5], (200,3))
plt.figure()
ax= plt.gca(projection="3d")    # plt.gca() o plt.axes()
ax.scatter3D(datos[:,0], datos[:,1], datos[:,2], c="magenta", s=30)
ax.set(xlabel="x", ylabel="y", zlabel="z")
ax.set_title("puntos aleatorios 3D", color="blue", size=15)
plt.show()
```

Vamos a explicar las nuevas funciones. El `plt.gca()` se refiere a "get current axes" (o usar `plt.axes()`) y significa que vamos a tomar los ejes, darles un nombre, `ax`, y tratarlos como objeto. El argumento de `gca()` es la proyección que se va a usar, aquí es "3d", pero podría ser "polar" para coordenadas polares. En la siguiente línea, usamos

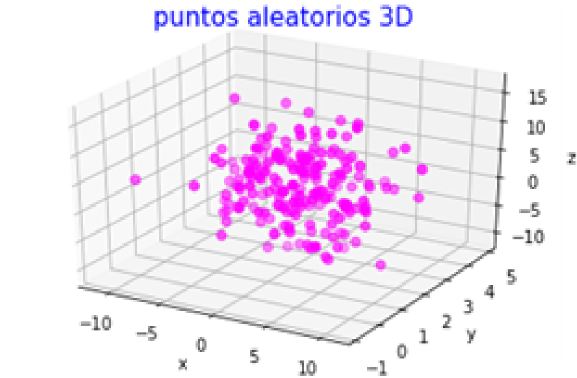


Figura 5.19: Puntos en 3D con scatter3D

el método del objeto `ax` para graficar, que es `scatter3D` porque son puntos. Y en la siguiente línea, usamos el método para dar título, nombre de ejes y otros, al mismo tiempo, `ax.set()`. Equivalentemente, para poder usar opciones en los títulos, nombres de ejes, y otras, podemos aplicar independientemente las funciones `ax.set_xlabel()`, `ax.set_title()`, etc, como ya se había comentado.

5.3.2. Curvas en 3D: plot3D

Para graficar una curva en 3D, de los cursos de cálculo sabemos que esto se logra dando un parámetro, esto es, son curvas paramétricas. Vamos a graficar como ejemplo una de las curvas más famosas, una espiral, donde el parámetro es la misma z, y las otras 2 variables dependen de ésta. Para ello usamos la función `plot3D` (Figura 5.20):

```
In [2]:
# intervalo de 0 a 11*pi, en pasos de pi/8:
z = np.arange(0, 11*np.pi, np.pi/8)    # z es el parámetro
x = z*np.cos(z)
y = z*np.sin(z)
plt.figure()
ax = plt.gca(projection="3d")
ax.plot3D(x, y, z, c="orange", lw=10)
plt.axis("off")
plt.show()
```

La única diferencia con el procedimiento de la sección previa es que ahora usamos la función `plot3D`, para tener líneas y no puntos, y removimos los ejes.

Ejercicios:

Grafica la siguiente curva paramétrica:

$$r(t) = \left(t^2, \sin t - t\cos t, \cos t + t\sin t\right), \quad t \in (0, 6\pi)$$

5.3.3. Superficies en 3D: plot_surface

Para graficar superficies en 3D, al igual que con las proyecciones en el plano, se requiere de `meshgrid` para la malla de puntos, pues hay que cubrir toda un área. La función a

Figura 5.20: Una curva en 3D, una espiral, con plot3D

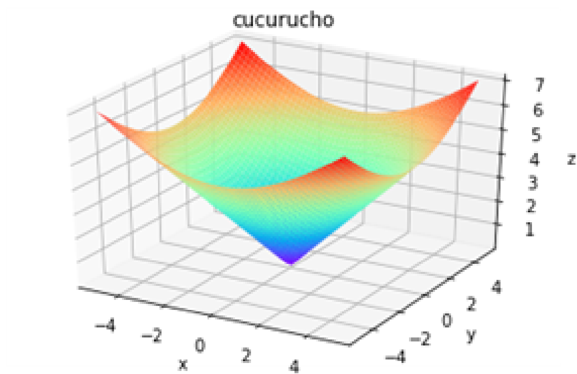


Figura 5.21: Una superficie en 3D, un cucurucho, con plot_surface

usar ahora es `plot_surface` (Figura 5.21):

```
In [3]:
x = np.linspace(-5,5,100)     # 100 números entre -5 y 5
y = np.linspace(-5,5,100)
X,Y = np.meshgrid(x,y)
z = np.sqrt(X**2+Y**2)
plt.figure()
ax= plt.gca(projection="3d")
ax.plot_surface(X,Y,z, cmap="rainbow")
ax.set(xlabel="x", ylabel="y", zlabel="z", title="cucurucho")
plt.show()
```

La figura admite tanto color, para colores sólidos, como `cmap` para degradados.

Ejercicios:

Dibuja en 3D las superficies de las funciones

$$a)\ z = y^2 - x^2, \quad b)\ z = (x-y)^2, \quad y \quad c)\ z = |xy|$$

5.4. Animaciones

Hay módulos especiales en Python para animaciones, pero podemos hacer animaciones simples con pyplot en el modo interactivo, así que en esta sección cambiamos a ese modo como se explicó al principio del capítulo. La función para lograrlo es pause, que va a esperar el tiempo que le demos en el argumento, en segundos, para dibujar el siguiente cuadro.

Vamos a ejemplificar con los llamados autómatas celulares, donde hay unas celdas, en general en 2D, y esas celdas siguen instrucciones de acuerdo con lo que está alrededor. Para simular las celdas vamos a usar una matriz y representarla como imagen con imshow.

Empezamos con la **regla de paridad**, donde las instrucciones son:

- los vecinos de una celda son las celdas de arriba, abajo, izquierda y derecha
- la celda está prendida (1) o apagada (0)
- la celda se prende cuando la suma de sus vecinos es impar, se apaga cuando es par.

Para una celda dada con índices (i,j), los vecinos de arriba y abajo son, respectivamente, las celdas (i-1,j) y (i+1,j), mientras que los vecinos izquierdo y derecho son las celdas con índices (i,j-1) y (i,j+1).

Vamos a empezar con una sola celda prendida y con condiciones periódicas a la frontera, esto es, los vecinos a la derecha de las celdas de la última columna son las celdas de la primera columna, y lo mismo con el primer y último renglones. Cuando hay este tipo de condiciones se usa el módulo (%), con módulo el tamaño del intervalo. Por ejemplo, si hay 5 columnas, la vecina derecha o "sexta" columna pasa a ser la primera. El índice de la "sexta" columna es 5 y el intervalo es también de tamaño 5, por lo que $5\%5 = 0$ nos retorna a la primera columna con índice 0. Entonces:

```
1  In [1]:
2  import numpy as np
3  import matplotlib.pyplot as plt
4  n=100; m=100
5  # matriz de 100x100 llena de ceros:
6  imagen= np.zeros((n,m))
7  # un solo punto prendido, en el centro
8  imagen[50,50]= 1
9  plt.figure()
10 # damos unos 30 cuadros de animación:
11 for _ in range(30):
12     # muestra la imagen:
13     plt.imshow(imagen,cmap="autumn")
14     # espera medio segundo:
15     plt.pause(0.5)
16     # copia lo que tenías en la imagen porque se va a ir modificando:
17     old=imagen.copy()
18     for i in range(n):
19         for j in range(m):
20             # suma de tus vecinos, arriba, abajo, etc:
21             t=old[(i-1)%n,j] + old[(i+1)%n,j] + \
22             old[i,(j-1)%m] + old[i,(j+1)%m]
23             if t%2==0:
```

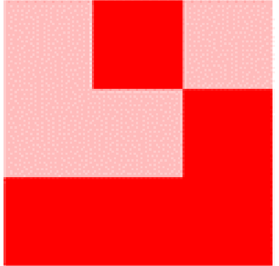


Figura 5.22: Un patrón inicial, un planeador, para el juego de la vida de Conway

```
24                    imagen[i,j]=0
25               else:
26                    imagen[i,j]=1
27 plt.show()
```

Se queda para el lector ver la animación resultante.

Ejercicios:

Annealing rule. Simula lo que pasa a unas celdas con la annealing rule, donde las reglas son:
- los vecinos de una celda son los 8 alrededor
- la celda está prendida (1) o apagada (0)
- si la suma de los vecinos a una celda es menor o igual a 3, o igual a 5, se prende; si es igual a 4 o mayor o igual a 6, se apaga.

Comienza con una matriz llena de 0's y 1's al azar con `np.random.randint(0,2,(n,m))`, con `n,m` el tamaño de la matriz, y con condiciones periódicas a la frontera.

Juego de la vida de Conway. Conway desafortunadamente murió por coronavirus el 11 de abril de 2019 y entre sus legados dejó este juego de la vida de autómatas celulares. Las reglas son:
- los vecinos de una celda son los 8 alrededor
- la celda está prendida (1) o apagada (0)
- una celda apagada con 3 vecinas prendidas, se prende; una celda prendida con 2 o 3 vecinas prendidas permanece prendida, de otra forma, se apaga.

Comienza con una matriz llena de 0's excepto por el patrón en el centro. Aunque puede haber varias configuraciones de inicio , usa como patrón para las celdas encendidas el de la Figura 5.22 (el rojo es la celda encendida, como el tercer renglón):

6 Scipy y aplicaciones en matemáticas

En este capítulo veremos el módulo `scipy` que es un módulo para cálculo científico. Tiene aplicaciones para Ecuaciones Diferenciales, Estadística, Integrales, Optimización, Interpolación, más de Álgebra Lineal, Series de Fourier, Funciones especiales, Procesamiento de señales, entre otras. Este libro no es tan ambicioso y sólo veremos algunas aplicaciones de Ecuaciones Diferenciales, Estadística y Regresión Lineal. Estos temas están presentes de manera explícita en los programas de estudio de algunas materias de la licenciatura de Modelación Matemática, o carreras similares, pero también están presentes y son de utilidad para las ingenierías. Los objetivos de aprendizaje de este capítulo son aplicar las funciones de `scipy` en diversos problemas de los ámbitos de las Ecuaciones Diferenciales Ordinarias y de la Probabilidad y Estadística. También, hacer un repaso de los fundamentos teóricos de la Inferencia Estadística y las Pruebas de Hipótesis. Hacer un repaso de la derivación de las ecuaciones necesarias para hacer una Regresión Lineal. Construir nuestras propias funciones para la resolución numérica de Ecuaciones Diferenciales Ordinarias y para hacer Regresiones Lineales con el método de mínimos cuadrados. Para ahondar más en la parte teórica de estos temas de matemáticas, en la bibliografía se recomiendan los libros del área de Ecuaciones Diferenciales y Modelación Matemática de Dennis Zill (2018) y de Barnes y Fulford (2015), los libros de Álgebra Lineal de David Lay (2016) y de Bernard Kolman y David Hill (2006), y el curso en línea de Estadística en Coursera de Mine Cetinkaya, de la Universidad de Duke.

6.1. Soluciones numéricas a EDO

Hay varios métodos numéricos para resolver ecuaciones diferenciales. En este apartado nos enfocaremos en Ecuaciones Diferenciales Ordinarias con Problemas de Valor Inicial.

En el submódulo `integrate` del módulo `scipy` de `Python`, hay varias funciones para resolver ecuaciones diferenciales de manera numérica, aunque sólo comentaremos la función `odeint`. Pero antes de eso, implementemos nuestras propias funciones para resolver Ecuaciones Diferenciales, primero para una ecuación con una variable dependiente, y después para un sistema de dos ecuaciones, utilizando el método de Runge-Kutta de cuarto orden.

6.1.1. Método Runge-Kutta para una ecuación diferencial

En el método de Runge-Kutta de cuarto orden, la aproximación a la solución de la ecuación

$$\frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0$$

está dada por la serie recursiva:

$$x_{n+1} = x_n + h, \quad y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

con h el tamaño del paso y donde

$$k_1 = f(x_n, y_n), \quad k_2 = f(x_n + \frac{h}{2}, \quad y_n + \frac{h}{2}k_1),$$

$$k_3 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2), \quad k_4 = f(x_n + h, y_n + hk_3)$$

Implementar Runge-Kutta es muy simple usando `numpy`.

```
In [1]:
import numpy as np
import matplotlib.pyplot as plt
def Runge(x0,xf,y0,h,fun):
    A = np.arange(x0,xf+h,h)
    # se pone límite xf+h al ser intervalo abierto al final
    B=np.zeros_like(A)
    # A, vector de las xs, B, vector de las ys
    B[0]=y0
    for i in range(len(A)-1):
        k1=fun(A[i],B[i])
        k2=fun(A[i]+h/2,B[i]+h*k1/2)
        k3=fun(A[i]+h/2,B[i]+h*k2/2)
        k4=fun(A[i]+h,B[i]+h*k3)
        B[i+1]=B[i]+h*(k1+2*k2+2*k3+k4)/6
    return(A,B)
```

Hagamos el ejemplo de una función analítica que se puede resolver fácilmente, como un decaimiento exponencial (cambiamos nombres a las variables, en lugar de x, y, es t, x, para enfatizar el tiempo):

$$\frac{dx}{dt} = -kx, \quad x(0) = 3, \quad k = 0.5/\text{día}$$

```
In [2]:
def decaimiento(t,x):
    return(-0.5*x)
```

Llamamos a la función de Runge-Kutta con $t_0 = 0$, $x_0 = 3$. El tamaño del paso lo escogemos a $h = 0.05$ y damos un valor cualquiera a t_f, por decir, 5.

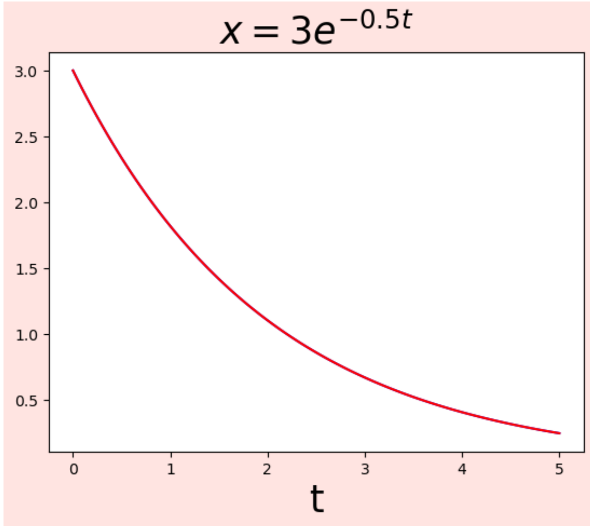


Figura 6.1: Solución numérica a la ecuación diferencial de decrecimiento exponencial.

```
In [3]:
# t0=0, tf=5, x0=3, h=0.05, fun=decaimiento
t, x= Runge(0,5,3,0.05,decaimiento)
plt.figure(facecolor="mistyrose")
plt.plot(t,x,c="r")
plt.xlabel("t",fontsize=25)
plt.title("$x=3e^{-0.5t}$",fontsize=25)
plt.show()
```

En el título de la Figura 6.1 ya dimos la solución $x = 3e^{-0.5t}$.

Hagamos otro ejercicio, el del crecimiento de una población con restricciones (crecimiento dependiente de la densidad de población), cuya ecuación es la ecuación logística:

$$\frac{dx}{dt} = rx\left(1 - \frac{x}{K}\right)$$

con r la razón de reproducción y K la capacidad de carga. Suponiendo una población inicial $x(0) = 100$, una razón de reproducción $r = 0.5/\text{año}$, una capacidad de carga $K = 300$, y dejando pasar unos $t_f = 20$ años, definimos a la función logística, llamamos a la función de Runge-Kutta con tamaño de paso $h = 0.05$ y graficamos (Figura 6.2):

```
In [4]:
r= 0.5; K= 300
def logistica(t,x):
    return(r*x*(1-x/K))
# t0=0, tf=20, x0=100, h=0.05, fun=logistica
t, x= Runge(0,20,100,0.05,logistica)
plt.figure(facecolor="lavender")
plt.plot(t,x,c="blueviolet")
plt.xlabel("t",fontsize=25)
plt.title("Ecuación Logística",fontsize=25)
plt.show()
```

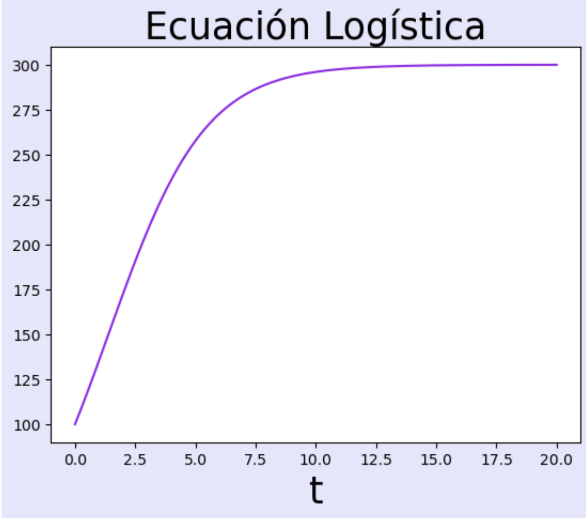


Figura 6.2: Solución a la ecuación diferencial logística, sobre el crecimiento de una población con restricciones.

6.1.2. Método de Runge-Kutta para un sistema de dos EDO

En el caso de un sistema de 2 ecuaciones diferenciales de la forma:

$$\frac{dx}{dt} = f(t,x,y), \quad \frac{dy}{dt} = g(t,x,y), \quad x(t_0) = x_0, \quad y(t_0) = y_0$$

la aproximación numérica por Runge-Kutta de cuarto orden está dada por:

$$t_{n+1} = t_n + h, \quad x_{n+1} = x_n + \frac{h}{6}(m_1 + 2m_2 + 2m_3 + m_4)$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

con h el tamaño del paso y donde

$$m_1 = f(t_n, x_n, y_n), \quad k_1 = g(t_n, x_n, y_n)$$

$$m_2 = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}m_1, y_n + \frac{h}{2}k_1\right), \quad k_2 = g\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}m_1, y_n + \frac{h}{2}k_1\right),$$

$$m_3 = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}m_2, y_n + \frac{h}{2}k_2\right), \quad k_3 = g\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}m_2, y_n + \frac{h}{2}k_2\right),$$

$$m_4 = f(t_n + h, x_n + hm_3, y_n + hk_3), \quad k_4 = g(t_n + h, x_n + hm_3, y_n + hk_3)$$

De nuevo, es sencillo implementar la función en `Python`, aunque consideraremos ecuaciones autónomas, donde las funciones f y g no dependen explícitamente del tiempo, al ver ejemplos sólo de este tipo:

```
In [5]:
def Runge_2(t0,tf,x0,y0,h,fun):
    t= np.arange(t0,tf+h,h)
    x= np.zeros_like(t); y=np.zeros_like(t)
```

```
    x[0]= x0; y[0]= y0
    for i in range(len(t)-1):
        m1, k1= fun(x[i],y[i])
        m2, k2= fun(x[i]+h*m1/2,y[i]+h*k1/2)
        m3, k3= fun(x[i]+h*m2/2,y[i]+h*k2/2)
        m4, k4= fun(x[i]+h*m3,y[i]+h*k3)
        x[i+1]= x[i]+ h*(m1+2*m2+2*m3+m4)/6
        y[i+1]= y[i]+ h*(k1+2*k2+2*k3+k4)/6
    return(t,x,y)
```

Para la función de Runge- Kutta propuesta, la función `fun` devuelve a f y g en una tupla.

Procedamos con el ejemplo del modelo presa-depredador, dado por

$$\frac{dx}{dt} = \beta x - c_1 xy, \quad \frac{dy}{dt} = c_2 xy - \alpha y, \quad x(t_0) = x_0, \quad y(t_0) = y_0$$

Donde x denota a la presa, y al depredador, y con c_1 y c_2 los términos de interacción entre la presa y el depredador que hace que disminuya la presa y aumenten los depredadores, respectivamente, β el término de crecimiento de la presa y α el término de disminución de los depredadores. Seleccionemos para nuestro ejemplo los valores $\beta = 1$/año, $\alpha = 0.5$/año, $c_1 = 0.01$/año, $c_2 = 0.004$/año.

Definimos entonces la función para la presa-depredador en `Python` y llamamos a la función de Runge-Kutta con un tamaño de paso $h = 0.05$ y con unos valores iniciales de $x(0) = 200$ presas, $y(0) = 100$ depredadores, y un tiempo final de 20 años, por decir. Graficamos x y y vs t (Figura 6.3):

```
In [6]:
beta=1; alpha=0.5; c1=0.01; c2=0.004
def presa_dep(x,y):
    return(beta*x-c1*x*y, c2*x*y-alpha*y)
# t0=0, tf=20, x0=200, y0=100, h=0.05, fun=presa_dep
t, x, y= Runge_2(0, 20, 200, 100, 0.05, presa_dep)
plt.figure()
plt.plot(t, x, label="presa", ls="-")
plt.plot(t, y, label="depredador", ls="-.")
plt.legend(fontsize=18)
plt.xlabel("t (años)",fontsize=20)
plt.show()
```

Aprovechemos este ejemplo para graficar en el espacio fase y vs x. Variemos la condición inicial del número de depredadores para ver cómo se comportan las curvas solución (Figura 6.4).

```
In [7]:
plt.figure(facecolor="aliceblue")
for k, lin in zip((20,100,150,250),("--","-",":","-.")):
    t, x, y= Runge_2(0, 20, 200, k, 0.05, presa_dep)
    plt.plot(x, y, label=k, ls=lin)
plt.xlabel("presa",fontsize=20)
plt.ylabel("depredador",fontsize=20)
plt.legend(fontsize=20)
plt.show()
```

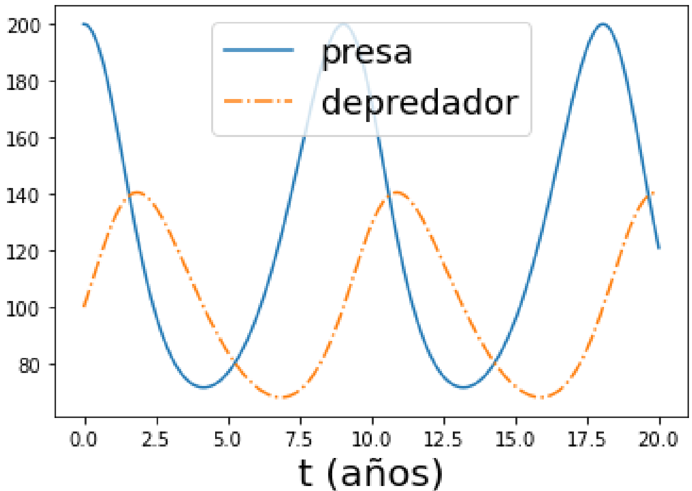


Figura 6.3: Soluciones a las ecuaciones de Lotka-Volterra de presa-depredador, en función del tiempo.

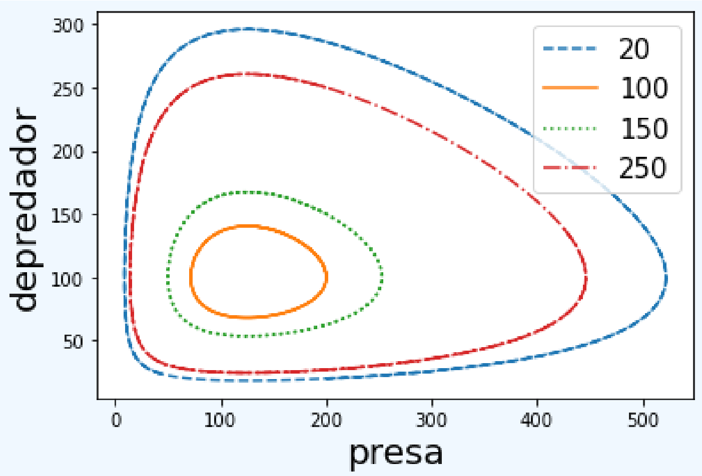


Figura 6.4: Soluciones en el espacio fase del sistema presa-depredador, para diferentes condiciones inciales del depredador. Todas las soluciones orbitan alrededor de un punto de equilibrio, expresado en términos de los parámetros del sistema.

Como se aprecia, las curvas solución en el espacio fase, dadas las diferentes condiciones iniciales, todas circularían alrededor de un punto de equilibrio estable, dado por los parámetros del sistema. El tema de la estabilidad de los puntos críticos amerita otro libro, pero para no dejar al lector con la duda podemos al menos hablar sobre cómo determinar los puntos críticos.

Para ello, las ecuaciones diferenciales tienen que igualarse a cero, como es la costumbre. Para el sistema presa depredador tendríamos:

$$\frac{dx}{dt} = \beta x - c_1 xy = x(\beta - c_1 y) = 0, \quad \frac{dy}{dt} = c_2 xy - \alpha y = y(c_2 x - \alpha) = 0$$

Si $x = 0$, entonces dx/dt es igual a cero, pero para que dy/dt también sea cero al mismo tiempo, necesariamente y tendría que ser cero. Un punto crítico es entonces $(0, 0)$. La otra posibilidad es que, para que dx/dt sea cero, $y = \beta/c_1$, y entonces para que dy/dt sea cero también al mismo tiempo, necesariamente $x = \alpha/c_2$. Insertando los valores de los parámetros, el punto crítico del ejemplo que vimos es:

```
In [8]:
x_crit= alpha/c2; y_crit= beta/c1
print("punto crítico:")
print(x_crit, y_crit)
Out[8]:
punto crítico:
125.0 100.0
```

6.1.3. Campos de direcciones

Para finalizar, grafiquemos el campo de direcciones del sistema presa-depredador con los parámetros de la sección anterior. Para ello vamos a usar la función `meshgrid`, para hacer una malla con los puntos donde vamos a colocar los vectores de campo, y la función `quiver`, para graficar esos vectores (Figura 6.5).

```
In [9]:
# malla de 10 puntos en la dirección x, entre 0 y 200 y de 8 puntos
# en la dirección y, entre 0 y 140; en lugar de cero, ponemos un
# número cercano para evitar mensajes de división entre cero
X,Y = np.meshgrid(np.linspace(0.001,200,10),np.linspace(0.001,140,8))
# se preparan los vectores direccionales u y v en las direcciones de
# f(x,y) y g(x,y), evaluados en los puntos de la malla
u = beta*X-c1*X*Y; v = c2*X*Y-alpha*Y
# se normalizan los vectores
u, v= u/np.sqrt(u**2+v**2), v/np.sqrt(u**2+v**2)
# graficamos
plt.figure(facecolor="lavender")
plt.quiver(X,Y,u,v,color="blueviolet")
plt.xlabel("presa")
plt.ylabel("depredador")
plt.show()
```

6.1.4. Función odeint de scipy.integrate

El submódulo `integrate` del módulo `scipy` contiene varias funciones para resolver sistemas de ecuaciones diferenciales de cualquier tamaño de manera numérica. En par-

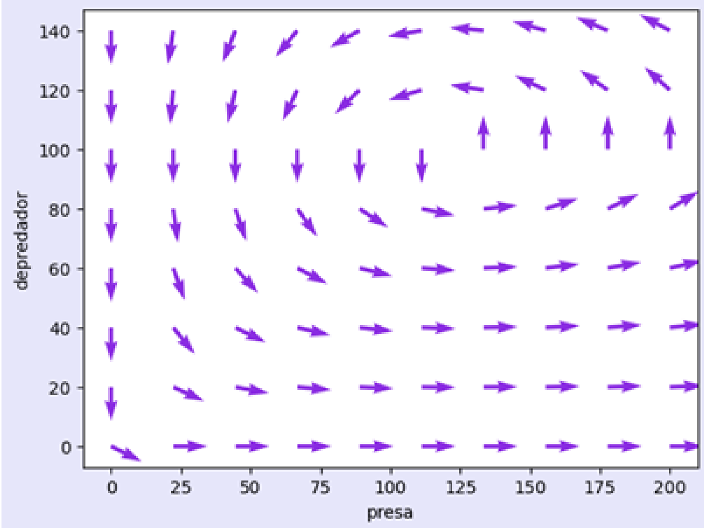


Figura 6.5: Campo de direcciones en el espacio fase de un sistema presa-depredador

ticular, cuenta con la función `odeint`, que funciona de la siguiente manera.

Supongamos que tenemos n variables dependientes z_i, con la variable independiente t, entonces el sistema de n ecuaciones diferenciales se escribe como:

$$\frac{dz_1}{dt} = f_1(t, z_1, \ldots, z_n), \quad \ldots \quad , \frac{dz_n}{dt} = f_n(t, z_1, \ldots, z_n)$$

Los pasos para aplicar `odeint` para resolver para cada z_i son:

1) dar un rango de valores para t

2) definir una función que engloba a todas las derivadas $\frac{dZ_i}{dt}$ con argumentos: una variable z que englobe a todas las variables z_i, la variable t y los parámetros que requieran las derivadas; la función la escribimos de forma que devuelva una tupla con las expresiones de las derivadas, una por entrada.

3) llamar a la función `odeint` con argumentos: la función definida en el paso 2, las condiciones iniciales de las z_i (si es más de una variable, en una tupla), la variable t, y una tupla con los parámetros. Se devuelve una matriz donde cada columna es una solución a cada variable z_i, para cada valor de t. Si hay una sola variable, entonces sólo se devuelve una lista de valores de z para cada t.

Hagamos un ejemplo con un sistema de dos ecuaciones, donde en lugar de z_1 y z_2 volvemos a emplear las variables x y y, por claridad. Digamos que tenemos de nuevo el modelo presa-depredador, pero donde ahora la presa x tiene un crecimiento logístico, esto es, crece pero está limitada por los recursos disponibles, y también está controlada porque son alimento del depredador. Como antes, el depredador y crece gracias a la presa, y muere por razones naturales, o porque se agota la presa:

$$\frac{dx}{dt} = \beta x \left(1 - \frac{x}{K}\right) - c_1 xy, \quad \frac{dy}{dt} = c_2 xy - \alpha y$$

Veamos el caso cuando las poblaciones iniciales son $x(0) = 200$ presas y $y(0) = 30$ depredadores, los parámetros son los mismos que en el ejercicio de la presa-depredador,

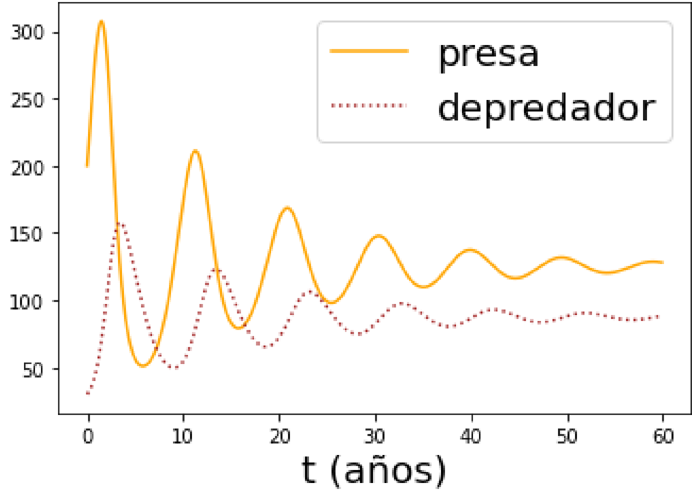


Figura 6.6: Solución a un sistema presa-depredador con crecimiento limitado de la presa por los recursos disponibles.

esto es, $\beta = 1/\text{año}$, $\alpha = 0.5/\text{año}$, $c_1 = 0.01/\text{año}$, $c_2 = 0.004/\text{año}$, y la capacidad de carga es $K = 1000$. Demos un tiempo de 60 años para ver el desarrollo de las poblaciones:

```
In [1]:
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
t= np.arange(0,60,0.1)     # paso 1
beta=1; alpha=0.5; c1=0.01; c2 = 0.004; K=1000;
# paso 2, derivadas, (z) engloba a las variables x y y
def deri(z,t,beta,alpha,K,c1,c2):
# argumentos: z, t, parámetros
    x, y = z
    return(beta*x*(1-x/K)-c1*x*y, c2*x*y-alpha*y)
# paso 3, las condiciones iniciales van en orden, en una tupla
z= odeint(deri,(200,30),t,(beta,alpha,K,c1,c2))
```

Vemos que en el argumento de deri, ponemos una sola variable z que después desglosamos en las variables requeridas en el primer renglón de la función. Lo que retorna `odeint` son 2 vectores columna donde cada vector es una secuencia de números con la solución a cada variable, del mismo tamaño que el arreglo t, por lo que hay que seleccionar cada columna para graficar (Figura 6.6):

```
In [2]:
plt.figure()
# x=z[:,0], primer columna,y=z[:,1], segunda columna
plt.plot(t,z[:,0],label="presa",ls="-",c="orange")
plt.plot(t,z[:,1],label="depredador",ls=":",c="brown")
plt.xlabel("t (años)",fontsize=20)
plt.legend(fontsize=20)
plt.show()
```

Vemos que el tamaño de las dos poblaciones oscila con el tiempo y eventualmente llegan a un valor de equilibrio. Podemos graficar también las dos soluciones en el espacio fase (Figura 6.7):

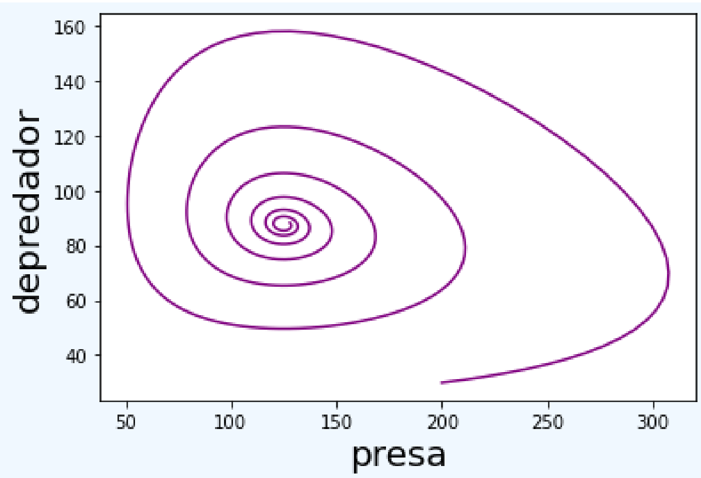


Figura 6.7: Espacio fase del sistema presa-depredador con recursos limitados para la presa.

```
In [3]:
plt.figure(facecolor="aliceblue")
plt.plot(z[:,0],z[:,1],c="purple")
plt.xlabel("presa",fontsize=20)
plt.ylabel("depredador",fontsize=20)
plt.show()
```

Vemos aquí también la oscilación de las soluciones para acercarse a un punto estable o atractor. Se deja al lector que encuentre el valor del punto crítico.

Ejercicios:

1. Dado el modelo SIR (Susceptibles-Infectados-Recuperados) para el contagio de una enfermedad curable, donde las ecuaciones están dadas por:

$$\frac{dS}{dt} = -\beta SI, \quad \frac{dI}{dt} = \beta SI - \gamma I, \quad \frac{dR}{dt} = \gamma I$$

encuentra las soluciones numéricas en un periodo de 20 días dadas las condiciones iniciales de que hay 950 personas susceptibles en un pueblo, 1 persona infectada y 0 personas recuperadas. Los valores de los parámetros son $\beta = 0.0015$/día y $\gamma = 0.4$/día. Dibuja las soluciones con respecto al tiempo, para ese intervalo de tiempo.

2. Dibuja las soluciones del problema de dos especies que compiten por recursos, a) con respecto al tiempo y b) en el espacio fase, en un periodo de tiempo de 5 años. Las ecuaciones que describen a las dos especies en competencia son:

$$\frac{dx}{dt} = a_1 x - b_1 x^2 - c_1 xy, \quad \frac{dy}{dt} = a_2 y - b_2 y^2 - c_2 xy$$

donde, para cada ecuación, el primer término es el término de crecimiento, el segundo es el de la competencia entre la misma especie, y el tercero es la competencia entre las 2 especies. Los valores de los parámetros son:

$$a_1 = 0.2, \quad a_2 = 0.09, \quad b_1 = 0.007, \quad b_2 = 0.01, \quad c_1 = 0.005, \quad c_2 = 0.004$$

todos por unidad de año. Supón que las poblaciones inicales son $x(0) = 1,000$ y $y(0) = 900$.

6.2. Distribuciones de probabilidad

`Python` cuenta con el submódulo `stats` del módulo `scipy` para albergar diversas distribuciones de probabilidad. Cada una de ellas cuenta con funciones para obtener ciertos resultados, y a continuación describiremos las más empleadas y que son comunes a ellas:

- `pmf`, probability mass function, función de probabilidad para distribuciones de variables discretas. Da la probabilidad evaluada en k. Puede tener una sola k como argumento o un arreglo de k´s, más los parámetros propios de la distribución.
- `pdf`, probability density function, función de probabilidad para distribuciones de variables continuas. Da la probabilidad evaluada en x. Puede tener una sola x como argumento o un arreglo de x´s, más los parámetros propios de la distribución.
- `cdf`, cummulative distribution function, función de probabilidad acumulada hasta x, que acepta un solo valor de la variable o un arreglo de valores, más los parámetros propios de la distribución.
- `ppf`, percent point function, función inversa a `cdf`, da el valor de x dada una probabilidad acumulada.
- `mean, std, var`, el promedio, la desviación estándar y la varianza de la variable

6.2.1. Distribución binomial

La más famosa de las distribuciones de probabilidad discretas es la binomial, que es la suma de ensayos de Bernoulli. Como recordatorio, esta distribución se aplica cuando tenemos un problema donde se pregunta por la probabilidad de que para n ensayos resulten k éxitos, donde el éxito tiene probabilidad p, y la única otra alternativa es el fracaso, con probabilidad $1 - p$. La distribución de probabilidad, el promedio y la varianza están dados por:

$$P(k, n, p) = \binom{n}{k} p^k (1-p)^{n-k}, \quad \mu = np \quad \text{y} \quad \sigma^2 = np(1-p).$$

Por ejemplo, supongamos que en un fin de semana un canal propone dos películas para el horario estelar, y la que tenga más votos se proyecta. Los resultados de la votación son un 42 % para Harry Potter y 58 % para Terminator. Si se seleccionan seis personas al azar, ¿cuál es la probabilidad de que exactamente tres de ellas voten por Harry Potter?

Con `Python` es fácil responder a esta pregunta. La distribución binomial se accede mediante la función `binom`, y la pregunta se contesta usando la función `pmf`, con argumento número de éxitos $k = 3$, y dando también como parámetros el número de ensayos y la probabilidad de éxito n y p:

```
1 In [1]:
2 from scipy.stats import binom
3 # número de intentos, probabilidad de éxito y número de éxitos
```

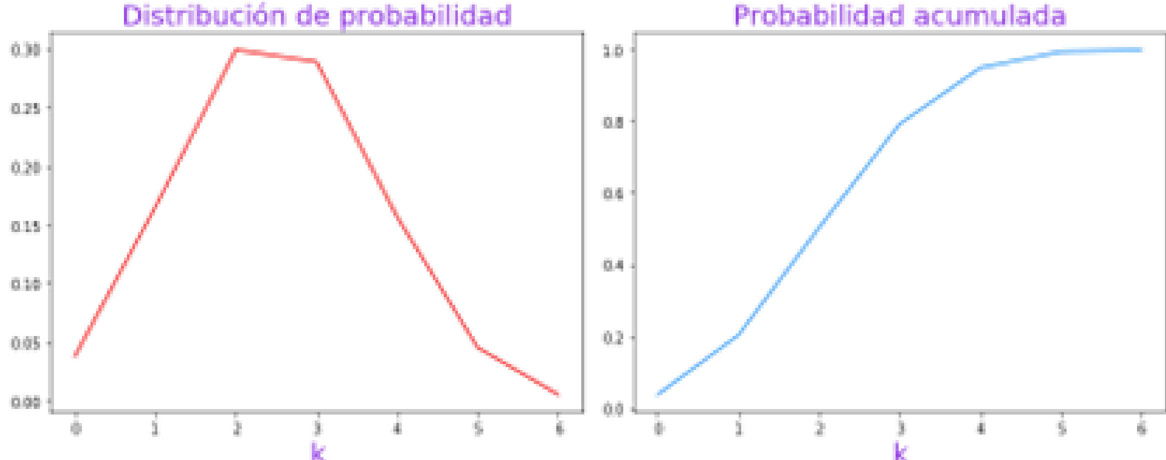


Figura 6.8: Izquierda: Distribución de probabilidad binomial, con número de ensayos $n = 6$ y probabilidad de éxito $p = 0.42$. Derecha, la probabilidad acumulada de la distribución de la izquierda.

```
n=6; p=0.42; k=3
print(binom.pmf(k,n,p))
Out[1]:
0.28910915712000007
```

Si ahora se pregunta por la probabilidad de que a lo más tres personas hayan votado por la película de Harry Potter, la función más apropiada para responder es la `cdf`, pues suma las probabilidades desde $k = 0$ hasta $k = 3$ éxitos:

```
In [2]:
print("a lo más k:",binom.cdf(k,n,p))
Out[2]:
a lo más k: 0.79201423936
```

Las gráficas de la distribución de probabilidad y de la probabilidad acumulada se muestran en la Figura 6.8:

```
In [3]:
x= np.arange(n+1)
# valores posibles de k, desde 0 hasta el total de intentos n
y= binom.pmf(x,n,p) # probabilidades exactas de los valores de x
z= binom.cdf(x,n,p) # probabilidades acumuladas de los valores de x
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)   # primera subFigura
plt.plot(x,y,color="r")
plt.xlabel("k", fontsize=20, color="blueviolet")
plt.title("Distribución de probabilidad", fontsize=20, color=\
"blueviolet")
plt.subplot(1,2,2)   # segunda subFigura
plt.plot(x,z,color="dodgerblue")
plt.title("Probabilidad acumulada", fontsize=20, color="blueviolet")
plt.xlabel("k", fontsize=20, color="blueviolet")
plt.tight_layout()
plt.show()
```

Ejercicios:

Según datos de la Secretaría de Salud del 2020 en México, 73 % de los adultos tienen obesidad. Si se escogen ocho adultos al azar,
a) ¿Cuál es el número esperado de adultos con obesidad? Usa la función `mean` con argumento k, n y p.
b) ¿Cuál es la probabilidad de que todos ellos tengan obesidad?
c) ¿Cuál es la probabilidad de que al menos seis tengan obesidad?

6.2.2. Distribución normal

La distribución normal o de Gauss es la más famosa de todas las distribuciones al haber muchos ejemplos de medidas que siguen esta distribución. Es una distribución continua. La función de distribución con promedio μ y desviación estándar σ está dada por:

$$\frac{1}{\sqrt{2\pi}\sigma}\int_{-\infty}^{x} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

En `stats`, se accede a esta distribución con la función `norm`. Veamos un ejemplo:

Dentro del grupo de adultos mayores deportistas de una ciudad se toma una muestra representativa y se obtiene que el promedio de la estatura de las mujeres es de 1.56 m y desviación estándar 9 cm. ¿Cuál es la probabilidad de que al tomar al azar a una mujer deportista de la tercera edad, ésta tenga una estatura menor a 1.50 m?

Como se pregunta por la probabilidad hasta un cierto valor, se usa la función de probabilidad acumulada, `cdf`, y entonces:

```
In [1]:
from scipy import stats
import numpy as np
import matplotlib.pyplot as plt
mu=1.56; sigma=0.09
a=1.50       # valor extremo buscado
print(stats.norm.cdf(a,mu,sigma))
Out[1]:
0.2524925375469227
```

Para tener una representación gráfica de la situación de este problema, se grafica la función de probabilidad con *pdf*, pero además, para representar el área deseada, vamos a usar la función `fill_between(x,y1,y2)`, que llena el área entre `y1` y `y2` para un intervalo x (Figura 6.9):

```
In [2]:
plt.figure()
paso = 0.02
# intervalo para área sombreada:
x1= np.arange(1.30, a+paso, paso)
# un intervalo amplio para mostrar buena parte de la Figura:
x2= np.arange(1.30, 1.80, paso)
# área sombreada:
plt.fill_between(x1,0,stats.norm.pdf(x1,mu,sigma),color="dodgerblue")
# gaussiana:
plt.plot(x2,stats.norm.pdf(x2,mu,sigma),color="b")
```

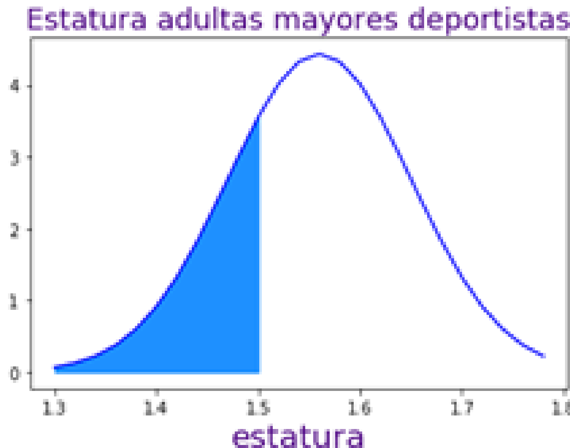


Figura 6.9: Probabilidad acumulada en una gaussiana

```
12 plt.title("Estatura adultas mayores deportistas", color="indigo",\
     fontsize=18)
13 plt.xlabel("estatura", color="indigo",fontsize=20)
14 plt.show()
```

Si se pregunta por la probabilidad de tener una estatura mayor a un cierto valor a, se restaría a la probabilidad total, 1, la probabilidad acumulada hasta ese valor, $1 - \mathrm{cdf}(a, \mu, \sigma)$.

Ejercicios:

El promedio del coeficiente intelectual de la gente es de 100, con desviación estándar igual a 15, ¿cuál es la probabilidad de que una persona puntúe más de 140 puntos?, ¿y la probabilidad de que su coeficiente esté entre 110 y 130?

6.3. Estadística: intervalos de confianza

Seguimos en esta sección con el submódulo `stats`. Primero vamos a recordar un poco sobre la idea de intervalos de confianza. Cuando se estima una propiedad de una población, digamos que un promedio, se toma una muestra y se espera que el promedio de esa muestra sea una buena estimación del promedio real de toda la población. Si tomamos otra muestra, en general se encuentra otra estimación diferente para el promedio. Así seguimos tomando promedios de diferentes muestras. Si hacemos un histograma de éstos, y el número de datos en las muestras es grande, el Teorema del Límite Central nos dice que la distribución de los promedios es gaussiana, con promedio el promedio real de la población, y desviación estándar dada por la desviación estándar de los datos originales entre la raíz del número de datos de la muestra. A esta desviación estándar de los promedios se le llama Error Estándar, SE (Standard Error), y entonces $1\ \mathrm{SE} = \sigma/\sqrt{n}$. Como estamos hablando de una gaussiana, sabemos que entre -1 SE y $+1$ SE se encuentra alrededor del 68 % de los datos, entre -2 SE y $+2$ SE el 95 % de los datos, y entre -3 SE y $+3$ SE se abarca hasta el 99.7 %, como se muestra en la Figura 6.10.

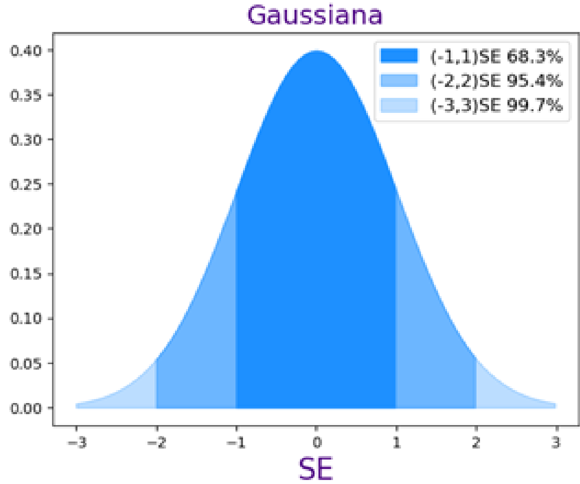


Figura 6.10: Probabilidades para diferentes intervalos, en términos del SE

Esto significa que, si tomamos una sola muestra, como es el caso en la vida real, hay un 68 % de probabilidad de que el promedio que midamos esté entre −1 SE y +1 SE del promedio real. La forma de proceder, entonces, es, tomar el promedio de nuestra muestra, y agregar un margen de error donde se trate de capturar el promedio real de la población con una cierta probabilidad, que se llama nivel de confianza, CL (confidence level). Así, si queremos 68.3 % de nivel de confianza, reportamos la medición como $x \pm 1$ SE, al 68.3 % de nivel de confianza, y si queremos el 95 % reportamos la medida como $x \pm 1.96$ SE, al 95 % de nivel de confianza (no es exactamente 2 SE). Esto nos construye los intervalos de confianza.

Para proceder con los cálculos, hay que pasar primero de la variable de nuestro problema x a una variable estandarizada z que justo mide a cuántos errores estándar (o desviaciones estándar usualmente, depende del problema) estamos del promedio, y el cambio de x a z está dado por

$$z = \frac{x - \bar{x}}{SE}.$$

Así, el centro de la distribución es $z = 0$, y $z = 1$ corresponde a 1 SE.

Entonces, dado un nivel de confianza, para encontrar el valor crítico z^* correspondeinte al intervalo $x \pm z^*$ SE, nos ayudamos de la Figura 6.11. La función $\varphi(z)$ que se muestra no es más que la probabilidad acumulada de $-\infty$ hasta z. Si $z > 0$, entonces $\varphi(-z)$ corresponde a la cola izquierda de la gaussiana. De las figuras, se ve que el área central que nos interesa, CL, es igual a

$$CL = 1 - 2\varphi(-z) = 1 - 2(1 - \varphi(z)) = 2\varphi(z) - 1$$

Por lo que, como lo que queremos encontrar es la z^* crítica, sacamos la inversa de $\varphi(z)$ conociendo el CL: $z^* = \varphi^{-1}\left(\frac{(1+CL)}{2}\right)$.

En stats ya hemos utilizado la función `cdf` para hallar la probabilidad acumulada. Su inversa es `ppf`, y si queremos encontrar la z^* crítica que se requiere al 95 % de CL, que es lo más usual en algunos reportes, entonces:

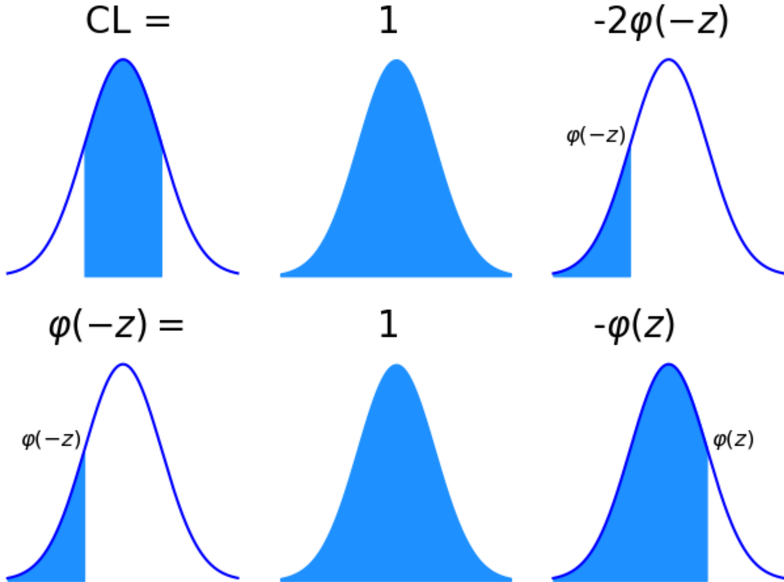


Figura 6.11: Forma pictórica de ver que, por un lado, la parte central del CL es igual a $1 - 2\varphi(-z)$, y que $\varphi(-z) = 1 - \varphi(z)$.

```
In [1]:
from scipy import stats
print(stats.norm.ppf((1+0.95)/2))
Out[1]:
1.959963984540054
```

Y de ahora en adelante podemos aprendernos este número, 1.96.

En la práctica, cuando se encuentra el margen de error se requeriría la verdadera desviación estándar de la población para encontrar el SE, pero, al igual que el verdadero promedio, seguramente no lo conocemos y la reemplazamos por la desviación estándar de la muestra, s, y entonces $SE = s/\sqrt{n}$. Cuando el número de datos es grande, no debería haber problema y estaríamos seguros de nuestro intervalo. Sin embargo, cuando el número de datos es menor a 30 cuando la distribución de la muestra es centrada, o menor a un cierto número mayor que 30 cuando la distribución de la muestra es sesgada (cargada a la izquierda o la derecha), entonces se usa la distribución t de student, que en principio es la que debemos usar siempre que no se sepa la desviación estándar real de la población. La distribución t de student luce muy similar a la gaussiana, nada más que es más chaparra en el medio y más ancha en las colas. Entre mayor sea el número de datos, mayor es la similitud entre la normal y la t de student. El parámetro, o los grados de libertad de esta distribución es el número de datos n menos 1. Vamos a comparar los 2 tipos de distribuciones para diferentes valores de n (Figura 6.12):

```
In [2]:
import matplotlib.pyplot as plt
import numpy as np
x= np.arange(-3,3,0.1)
plt.plot(x, stats.norm.pdf(x), label="normal")
for n in (5,10,30):
    plt.plot(x, stats.t.pdf(x,n-1), label="t-student, n="+str(n))
```

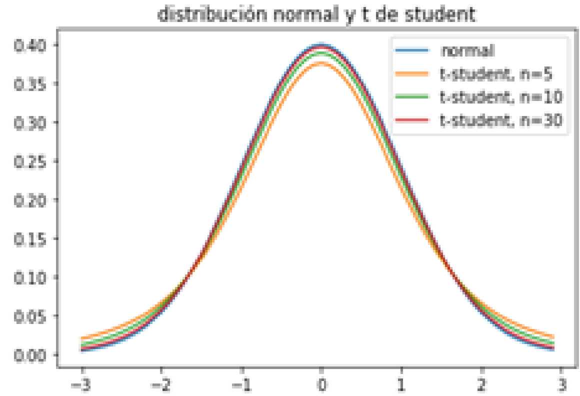


Figura 6.12: Distribución normal o gaussiana y distribución t de student. Conforme aumenta el número de datos, la distribución t de student tiende a la distribución normal.

```
plt.title("distribución normal y t de student")
plt.legend()
plt.show()
```

Vemos que cuando $n = 30$ las 2 distribuciones casi se sobreponen. Ahora, si queremos encontrar la t^* crítica para un nivel de confianza del 95 %, entonces, para un cierto número de datos tenemos que:

```
In [3]:
for n in (10,20,30,40):
    print(stats.t.ppf((1+0.95)/2,n-1))
Out[3]:
2.2621571627409915
2.093024054408263
2.045229642132703
2.022690911734728
```

Conforme aumentamos n nos acercamos más al valor menos conservativo de 1.96 de la distribución normal.

6.3.1. Promedios

Vamos a hacer el ejemplo de obtener un intevalo al 95 % de nivel de confianza del promedio de las masas de los estudiantes universitarios varones en el plantel de Iztapalapa, con 32 datos y usando la distribución t de student para ser un poco más conservativos. Los datos:

```
In [4]:
A= np.array([49.5, 90, 89.5, 77, 78, 54.5, 110.5, 65, 65.5, 62, 60,\
    66.5, 67.5, 98, 67, 73, 86, 67, 83, 75.5, 64.5, 66, 71.5, 82, 85,\
    61.5, 102, 103, 77.5, 112, 87, 86.5])
```

Obtenemos el promedio y la desviación estándar usando `numpy`:

```
In [5]:
prom = A.mean(); sigma = A.std(); n = len(A)
print(prom, sigma, n)
```

```
Out[5]:
(77.609375, 15.631865679098416, 32)
```

Ahora su error estándar y, usando `stats`, el valor crítico t^*:

```
In [6]:
SE= sigma/np.sqrt(n)
t= stats.t.ppf((1+0.95)/2, n-1)
print(SE,t)
Out[6]:
(2.7633495560719363, 2.0395134463964077)
```

Finalmente, el margen de error:

```
In [7]:
ME= t*SE
print(ME)
Out[7]:
5.635888576702258
```

Por lo que tenemos un 95 % de confianza de que el promedio de las masas de todos los estudiantes varones del plantel Iztapalapa está en el intervalo entre 77.6 ± 5.6 kg, es decir, que está entre 72.0 kg y 83.2 kg.

6.3.2. Proporciones

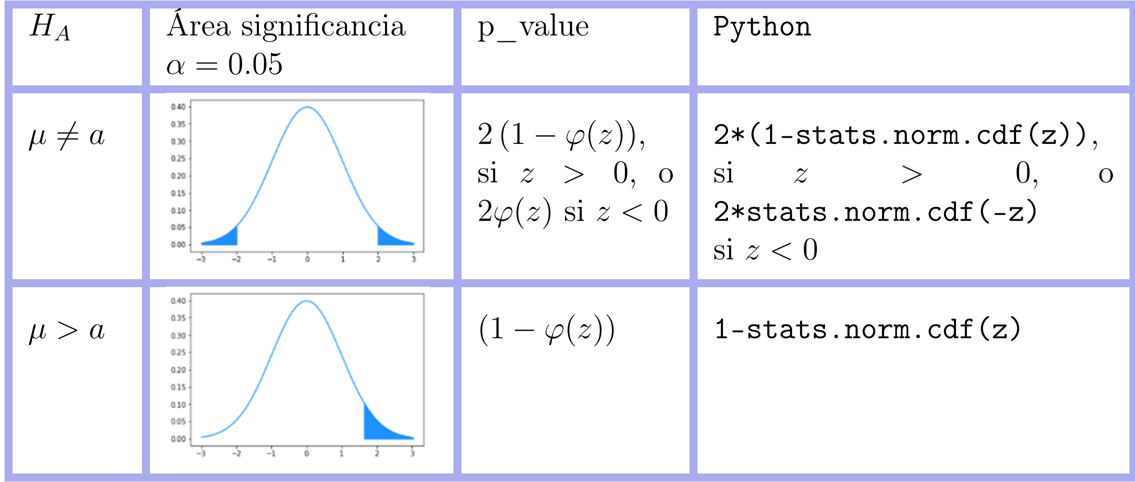
Si lo que ahora medimos es una proporción p, como cuando tenemos variables categóricas, se trabaja de la misma forma que en la sección anterior pero ahora el error estándar está dado por $SE = \sqrt{\frac{p(1-p)}{n}}$. Digamos que queremos saber a qué proporción de personas les gusta la cebolla en el plantel Iztapalapa. Se les preguntó a 116 personas, entre estudiantes, profesores y administrativos, de las que 86 contestaron que sí les gusta y 30 no. Entonces, la proporción de personas a las que sí les gusta la cebolla:

```
In [8]:
p= 86/116
print(p)
Out[8]:
0.7413793103448276
```

Si queremos dar un intervalo al 95 % de confianza:

```
In [9]:
n= 116
SE= np.sqrt(p*(1-p)/n)
t= stats.t.ppf((1+0.95)/2, n-1)
print("SE y t*: ", SE, t)
ME= t*SE
print("ME: ", ME)
Out[9]:
SE y t*:  0.040655833501930375 1.9808075410672
ME:  0.0805313815889962
```

Por lo que el intervalo al 95 % de confianza de la proporción real de adultos (de la ciudad de México y el Estado de México) a las que les gusta la cebolla es de 0.74 ± 0.08, es decir, el porcentaje está entre 66 % y 82 %.



H_A	Área significancia $\alpha = 0.05$	p_value	Python
$\mu \neq a$		$2\,(1-\varphi(z))$, si $z > 0$, o $2\varphi(z)$ si $z < 0$	`2*(1-stats.norm.cdf(z))`, si $z > 0$, o `2*stats.norm.cdf(-z)` si $z < 0$
$\mu > a$		$(1-\varphi(z))$	`1-stats.norm.cdf(z)`

Tabla 6.1: Cálculos del p_value

Ejercicios:

Encuentra el intervalo al 95 % de confianza de la proporción de adultos a los que les gusta el ajo, si se les preguntó a 116 adultos y 72 contestaron que sí, los demás que no.

6.4. Estadística: prueba de hipótesis

Continuamos con el submódulo `stats`. Digamos que tenemos un parámetro de una población, como un promedio, y queremos compararlo con el promedio de otra población o con un valor que esperamos a priori. La hipótesis nula, H_0, es que el promedio de nuestra problación sea igual al valor de la población de referencia o del valor a priori, $\mu = a$, y la hipótesis alternativa, H_A es que los valores son diferentes, $\mu \neq a$, o podría ser que un valor fuera mayor que el otro, $\mu > a$ o $\mu < a$. Tomamos entonces el promedio de una muestra, $\bar{x}$, y si $x \approx a$, esperamos que $\bar{x}$ esté cerca del centro de la distribución, a, mientras que entre más diferente sea el valor, $\bar{x}$ se va al área de las colas. Como siempre, para lo siguiente vamos a estandarizar x, de forma que $z = \frac{x-\mu}{SE}$, y suponemos que el parámetro a estimar sigue una distribución normal o t de student . Se define entonces el nivel de significancia α, al área de la cola o de las colas, predefinida, para la que, si entra $\bar{z}$ al área, se rechaza H_0, mientras que el p_value es el área sombreada de la cola o colas a partir del valor de $\bar{z}$.

En la Tabla 6.1 vemos los casos en que H_A es $\mu \neq a$ y en que $\mu > a$. El nivel de significancia (el área sombreada) es $\alpha = 5\,\%$ en las figuras. El p_value debe ser menor o igual a esta área o probabilidad para rechazar la H_0, y en la tabla se muestra su cálculo usando la función de probabilidad acumulada $\varphi(z)$, y la forma de implementarlo en `Python` con `cdf`:

6.4.1. Diferencia de promedios

Vamos a hacer una prueba de hipótesis concerniente a diferencias de promedios de dos poblaciones independientes. Cuando los datos son dependientes, como cuando se compara, para las mismas personas, cómo es su estado de salud después de un cier-

to tratamiento, o sus calificaciones después de un curso de recuperación, se toma la diferencia de la variable medida y a partir de ahí se calcula el error estándar de esas diferencias. Pero, cuando tenemos poblaciones independientes, el error estándar está dado por $SE = \sqrt{SE_1^2 + SE_2^2}$.

Como ejemplo tomemos los resultados de la prueba de PISA para estudiantes de 15 años de los dos sexos de todos los países involucrados. Según los datos del 2015, en el área de lectura, las niñas promedian un valor de 506 puntos con un SE de 0.5, mientras que los niños promedian un puntaje de 470 puntos con un SE de 0.6. ¿Es esta diferencia de resultados estadísticamente significativa? Para averiguarlo, hacemos que nuestra H_0 sea $\mu_m - \mu_h = 0$, y la H_A es que $\mu_m - \mu_h \neq 0$. Hay un número suficiente de datos, es la prueba de PISA, así que no necesitamos buscar más información para hallar el número de estudiantes, y usamos la distribución gaussiana para hallar el p_value. Ahora normalizamos de la forma: $z = \frac{(\bar{x}_m - \bar{x}_h) - 0}{SE}$. Poniendo un nivel de significancia $\alpha = 0.05$:

```
In [1]:
z= (506-470)/np.sqrt(0.5**2+0.6**2)
p_value= 2*(1-stats.norm.cdf(z))
print("z y p_value: ", z, p_value)
Out[1]:
z y p_value:  46.093276775842554 0.0
```

Desde que vemos una z tan grande, ya sabemos que el p_value debe ser muy pequeño, sólo hay que recordar que con $z = 1.96$ ya tenemos 95 % de probabilidad en el medio y queda $\alpha = 5\,\%$ en las colas. Por ser p_value menor que α, rechazamos la H_0, y hay suficiente evidencia para decir que los 2 tipos de poblaciones muestran diferente habilidad en la lectura.

Ejercicios:

Según los datos de la prueba de PISA de 2015 y considerando a todos los países involucrados, los resultados en matemáticas muestran que las niñas promedian un valor de 486 puntos con un SE de 0.5, y los niños promedian 494 puntos, con un SE de 0.6. ¿Se puede decir que hay una diferencia por género estadísticamente significativa en los resultados de PISA en matemáticas?

6.4.2. Diferencia de proporciones

Seguimos con diferencias, pero ahora de proporciones. Sólo que, como estamos haciendo una hipótesis sobre una proporción, la desviación estándar, que es proporcional a p, y por tanto, el error estándar, no lo vamos a obtener de los datos directos, sino que vamos a hacer ciertas suposiciones.

Vamos a ver un ejemplo. Se le preguntó a un grupo de universitarios del plantel Casa Libertad si les gusta el heavy metal o no. Las respuestas, según el género, fueron que a 33 mujeres de un total de 55 sí les gusta el heavy metal, mientras que a 32 hombres de 61 también les gusta. ¿Hay alguna diferencia significativa entre el gusto por el heavy metal en los 2 grupos? La H_0 es que el porcentaje en todos los estudiantes del plantel es el mismo sin importar género, y la H_A es que las proporciones son diferentes. Las proporciones y su diferencia son:

```
In [1]:
nh=61; nm=55
ph=32/nh; pm=33/nm
print("ph, pm y (ph-pm): ", ph, pm, ph-pm)
Out[1]:
ph, pm y (ph-pm):  0.5245901639344263 0.6 -0.07540983606557372
```

La diferencia es negativa, podríamos tomar el valor absoluto pero dejémoslo así para saber tratar todos los casos. Ahora, como vamos a proponer una proporción p para hallar el SE, lo que se suele usar, cuando hay una diferencia de proporciones independientes, es no favorecer a ningún grupo y usar la proporción del total de los que dicen que sí, respecto al total de entrevistados, que se llama p_pool, $p_p = (32+33)/(61+55)$, y así el SE se vuelve:

$$SE = \sqrt{\frac{p_p(1-p_p)}{n_h} + \frac{p_p(1-p_p)}{n_m}}$$

Usamos la distribución t de student, y para ser lo más conservativos posibles usamos, para los grados de libertad df, el menor de los números de datos, pues entre más datos hay más parecido con la distribución normal. Entonces $df = \text{mín}(n_h-1, n_m-1)$. Además, como t va a ser negativo, usamos para el p_value $2\varphi(t)$ en lugar de $2(1-\varphi(t))$ (o en todo caso se usa el valor absoluto de t junto con la segunda expresión):

```
In [2]:
p_pool= (32+33)/(nh+nm)
SE= np.sqrt(p_pool*(1-p_pool)*(1/nh+1/nm))
t= (ph-pm)/SE
p_value= 2*stats.t.cdf(t,nm-1)
print("t y p_value",t,p_value)
Out[2]:
t y p_value -0.8170741807500731 0.41747638617518845
```

Como el p_value no es menor que $\alpha = 0.05$, no hay evidencia suficiente para rechazar la H_0, no podemos decir que hay diferencia por género en el gusto por el heavy metal para el grueso de estudiantes del plantel Iztapalapa.

Ejercicios:

¿Hay alguna diferencia por género en el gusto de estudiantes universitarios en Iztapalapa por el reggaetón? Los datos obtenidos es que a 21 de 55 mujeres sí les gusta, y a 26 de 61 hombres también.

6.5. Regresión lineal con mínimos cuadrados

`Python` tiene varias funciones para hacer regresión lineal con mínimos cuadrados. Hay al menos cinco formas de hacerlo, que son con `stats` de `scipy`, `statsmodel`, `numpy`, `sklearn` y con `pandas`. En esta sección exploraremos las dos primeras, por dar una mayor información estadística, pero, antes de eso, comencemos con nuestras propias funciones para también hacer un repaso de a qué se refiere este ajuste.

6.5.1. Ajuste de una recta con mínimos cuadrados

En 2 dimensiones, dados n puntos (x_i, y_i) producto de alguna medición, el ajuste de una recta por el método de mínimos cuadrados consiste en minimizar la suma de los cuadrados de las diferencias de los n valores de la variable dependiente y, y la aproximación que se hace con la recta, dada la variable independiente x. En más dimensiones se generaliza a aproximaciones ya no con rectas sino con hiperplanos. Cuando la relación entre las variables no es lineal, se trata de hacer transformaciones a las variables para lograr, si es el caso, el comportamiento lineal.

La función a minimizar en 2D es entonces la suma de los errores

$$\sum_{i=1}^{n} \xi_i^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

con $\hat{y} = mx + b$, donde m y b son la pendiente y la ordenada al origen de la recta a ajustar a los n puntos (x_i, y_i).

Hay más de una forma de minimizar la función. La forma de cálculo diferencial es derivando la función respecto a las variables a minimizar o maximizar, que en este caso son los parámetros m y b, e igualándola a cero para encontrar los valores óptimos de esos parámetros:

$$f(m,b) = \sum_i (y_i - mx_i - b)^2 =$$

$$= \sum_i y_i^2 + m^2 \sum_i x_i^2 + nb^2 - 2m \sum_i x_i y_i - 2b \sum_i y_i + 2mb \sum_i x_i$$

$$\rightarrow \quad \frac{\partial f}{\partial b} = 2nb - 2\sum_i y_i + 2m \sum_i x_i = 0 \quad \Rightarrow \quad b = \frac{\sum_i y_i}{n} - m\frac{\sum_i x_i}{n} = \langle y \rangle - m\langle x \rangle$$

En la última ecuación usamos la definición del promedio o valor esperado de una variable, $\langle x \rangle = \frac{\sum_i x_i}{n}$. Es de esperarse el resultado, nos está diciendo que, en promedio, $\langle y \rangle = m\langle x \rangle + b$. Derivando respecto al otro parámetro:

$$\frac{\partial f}{\partial m} = 2m \sum_i x_i^2 - 2\sum_i x_i y_i + 2b \sum_i x_i = 0$$

Sustituyendo b en la ecuación de arriba:

$$m \sum_i x_i^2 - \sum_i x_i y_i + (\langle y \rangle - m\langle x \rangle) \sum_i x_i = 0$$

Factorizando m y dividiendo entre n:

$$m\left(\frac{\sum_i x_i^2}{n} - \langle x \rangle \frac{\sum_i x_i}{n}\right) = \frac{\sum_i x_i y_i}{n} - \langle y \rangle \frac{\sum_i x_i}{n} \quad \Rightarrow \quad m = \frac{\langle xy \rangle - \langle x \rangle \langle y \rangle}{\langle x^2 \rangle - \langle x \rangle^2} = \frac{\sigma_{xy}}{\sigma_x^2}$$

Al final escribimos a la pendiente en términos de la covarianza de x y y, σ_{xy}, y la varianza de x, σ_x^2. Sólo hay que tener cuidado, a la hora de hacer los cálculos, en que

la varianza y la covarianza algunas veces en los programas como `Python` se calculan dividiendo entre n o entre $n-1$, según la función. Como es una división, se cancelan las n o las $n-1$, sólo hay que ser consistentes en la elección del calculador tanto en el numerador como en el denominador.

Recopilando, tenemos que:

$$m = \frac{\sigma_{xy}}{\sigma_x^2}, \quad b = \langle y \rangle - m\langle x \rangle \tag{6.1}$$

Hay otro método, usando Álgebra Lineal, que es eficiente y que se puede escalar fácilmente a cuando hay más variables involucradas. El método consiste en escribir las n ecuaciones $y_i = mx_i + b + \xi_i$ en forma matricial:

$$\begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} \begin{pmatrix} b \\ m \end{pmatrix} + \begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

$$Ap + \xi = y \tag{6.2}$$

Con p el vector con los parámetros (b, m), y y ξ los vectores de las y's y de los errores, y A la matriz de n renglones con primera columna una lista de unos, y la segunda columna el vector x. La estrategia del Álgebra Lineal es ver a ξ como un vector ortogonal al espacio generado por las columnas de A, de modo que al hacer el producto de la traspuesta de A con la ecuación (6.2), el término del error se elimine:

$$A^T \xi = 0 \tag{6.3}$$

Desarrollando la ecuación (6.3) (con $\xi_i = y_i - mx_i - b$) se obtienen los mismos resultados que en las ecuaciones (6.1). Para imaginarnos la situación, es como si nos aproximáramos a un punto fuera de una recta. Nos movemos por la recta y lo que resta para llegar al punto es la distancia del punto a la recta, que sabemos que es mínima en la dirección perpendicular del punto a la recta. O si nos imaginamos a un punto fuera de un plano, y queremos aproximarnos al punto. Nos movemos por el plano, y lo que resta para llegar al punto, si queremos que sea mínimo, es la distancia en la dirección perpendicular del punto al plano. De esta forma, ξ se minimiza cuando es perpendicular a Ap.

Multiplicando a la izquierda la ecuación (6.2) por A^T, y usando la ecuación (6.3) tenemos:

$$A^T Ap = A^T y \tag{6.4}$$

La ecuación (6.4) es muy sencilla para resolver, pues si sólo tenemos una variable independiente y A es una matriz de $n \times 2$, entonces $A^T A$ tiene un tamaño de 2×2 y $A^T Ap$ es un vector de tamaño 2. En general, si hay r variables independientes, $A^T A$ es una matriz de tamaño $(r+1) \times (r+1)$.

Vamos a hacer un ejemplo con las medidas sugeridas por el hombre de Vitrubio. Medimos la extensión de los brazos y la altura de 33 personas. Según Vitrubio, las medidas deben ser las mismas.

```
In [1]:
import numpy as np
import matplotlib.pyplot as plt
puntos= np.array([(1.67,1.77),(1.67,1.76),(1.60,1.70),(1.63,1.72),\
(1.55,1.54),(1.60,1.61),(1.59,1.66),(1.785,1.815),(1.62,1.745),\
(1.61,1.63),(1.54,1.585),(1.72,1.785),(1.71,1.785),(1.49,1.575),\
(1.67,1.70),(1.78,1.80),(1.52,1.575),(1.73,1.885),(1.62,1.68),\
(1.68,1.765),(1.66,1.725),(1.45,1.485),(1.71,1.79),(1.73,1.84),\
(1.65,1.68),(1.58,1.685),(1.63,1.635),(1.73,1.73),(1.58,1.56),\
(1.55,1.595),(1.625,1.67),(1.715,1.83),(1.49,1.55)])
x = puntos[:,0]     # la primera columna de puntos son las xs
y = puntos[:,1]     # la segunda columna son las ys
```

Usemos primero el método de Álgebra Lineal con la ecuación (6.4). Para ello, preparamos la matriz A de $n \times 2$ con puros unos, con `np.ones`, y después la segunda columna la reemplazamos con el vector x:

```
In [2]:
A= np.ones((len(x),2))
A[:,1]= x           # reemplazamos la segunda columna por el vector x
C= np.dot(A.T,A)    # hacemos el producto de A traspuesta y A
c= np.dot(A.T,y)    # hacemos el producto de A traspuesta y y
# resolvemos el sistema de ecuaciones Cp=c, con p=(b,m)
b, m = np.linalg.solve(C,c)
print(m,b)
Out [2]:
1.0872209258008234 -0.08257271475082933
```

La pendiente de cerca de 1.1 nos confirma la desviación respecto al ideal de 1 de Vitrubio. Graficamos ahora comparando el ajuste contra el de Vitrubio (Figura 6.13):

```
In [3]:
plt.figure()
plt.scatter(x,y,color="dodgerblue")
plt.plot(x, m*x+b, c="r", label="ajuste")            # recta ajustada
plt.plot(x,x,c="blueviolet",label="Vitrubio ideal") # recta y = x
plt.legend()
plt.xlabel("altura",color="purple",fontsize=20)
plt.ylabel("brazos extendidos",color="purple",fontsize=20)
plt.show()
```

La recta quedó bien ajustada, y en la Figura 6.13 se observa que, al menos entre los participantes, hubo una tendencia a tener brazos más largos respecto a su estatura, según el ideal de Vitrubio.

Comparemos con el cálculo directo en términos de varianzas, covarianzas y promedios, Ecs. (6.1):

```
In [4]:
# calculamos sigma_xy y sigma^2_x con la matriz de covarianza,
# en la diagonal principal están las varianzas de x y y
# y en la diagonal cruzada la covarianza entre x y y
m2= np.cov(x,y)[0,1]/np.cov(x,y)[0,0]
b2= np.mean(y)-m2*np.mean(x)
print(m2,b2)
Out [4]:
1.0872209258008656 -0.08257271475089834
```

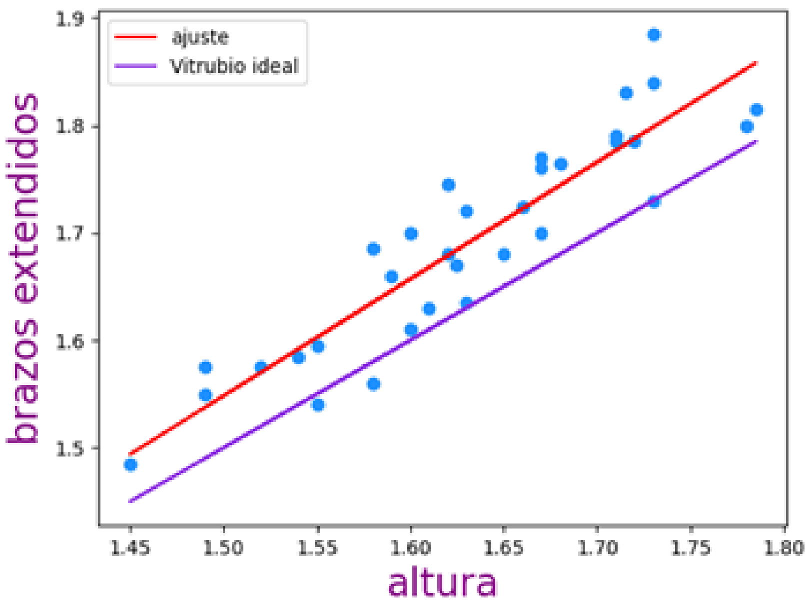


Figura 6.13: Datos medidos del largo de los brazos extendidos respecto a la altura de algunas personas. Se muestra la recta ajustada mediante mínimos cuadrados vs la recta ideal $y = x$.

Aunque el cálculo con las varianzas, covarianzas y promedios fue más directo y simple, usando un programa de cálculo como en `Python`, se prefiere el método de Álgebra Lineal para la generalización sencilla a más variables.

Para terminar esta sección, hablemos del caso en que la variable dependiente no dependa linealmente de la variable independiente o variables independientes. En este caso se podrían hacer cambios de variables para que las potencias de todas las nuevas variables sean uno.

Pongamos el ejemplo del péndulo simple. Haciendo un diagrama de cuerpo libre, se llega a la siguiente relación entre el periodo del péndulo T (tiempo que tarda en hacer una oscilación) y su largo L:

$$T = 2\pi\sqrt{\frac{L}{g}}$$

Donde g es la gravedad del planeta. Se puede entonces cambiar a la variable $z = \sqrt{L}$ y la ecuación queda lineal:

$$T = \frac{2\pi}{\sqrt{g}}z$$

Hay formas de linearizar ecuaciones, como usar logaritmos, pero ese es un tema que amerita sus propias notas.

Ejercicios:

1. Con un grupo de compañeros, checa la aseveración de que el tamaño de la planta de los pies es el mismo que el de los antebrazos (del codo a la muñeca). Haz unas mediciones

y encuentra la pendiente de la recta que mejor se ajusta. Haz más experimentos como relacionar la estatura de las personas con su masa.

2. Juanita tiene que presentar un proyecto de ciencias y se le ocurre a última hora hacer un experimento con un péndulo para medir la gravedad de la Tierra. Hace mediciones de diferentes longitudes de una cuerda, amarrada a una masita, que funciona como péndulo, y sus respectivos periodos de oscilación. Cambia a una nueva variable, que es la raíz de la longitud, para ajustar la mejor recta de T vs $\sqrt{L}$ y encontrar así la gravedad. ¿Qué valor de la gravedad encontró Juanita si obtuvo las siguientes medidas? (Spoiler: no le salió 9.8 m/s^2).

L	0.8 m	1.0 m	1.2 m	1.4 m	1.6 m
T	1.7 s	2.0 s	2.2 s	2.3 s	2.4 s

6.5.2. stats.linregress

En esta sección vamos a ver la función `linregress` del submódulo `stats` del módulo `scipy`, para hacer una regresión lineal. Esta función retorna 5 valores: la pendiente m, la ordenada al origen b, el coeficiente de correlación R, el p_value y el error estándar SE.

Como recordatorio, el coeficiente de correlación mide qué tan relacionados están x y y. Si los puntos medidos están todos sobre la recta propuesta, y la pendiente es positiva, entonces $R = 1$. Si están sobre la recta propuesta pero la pendiente es negativa, $R = -1$. R es un número entre -1 y 1, entre mayor sea el valor absoluto, mayor la relación lineal entre x y y. Por definición, $R = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$, donde σ_{xy} es la covarianza de x y y, y σ_x y σ_y son sus desviaciones estándar. Sobre el SE, éste es el error estándar de la pendiente, por si queremos dar un margen de error, $m \pm ME = m \pm t^*$ SE. Para la t^*, ésta se calcula como en las secciones previas, dado un intervalo de confianza, sólo que ahora los grados de libertad no son igual al número de datos menos 1, sino a $n - 2$, pues dando la pendiente y la ordenada al origen perdemos 2 grados de libertad. Por último, sobre el p_value, éste procede de tener como hipótesis nula que la pendiente es cero, por lo que un valor pequeño de p_value nos asegura tener una pendiente diferente de cero.

Sigamos con el ejemplo de Vitrubio para ver cómo funciona `linregress`:

```
In [1]:
import numpy as np
import matplotlib.pyplot as plt
puntos= np.array([(1.67,1.77),(1.67,1.76),(1.60,1.70),(1.63,1.72),\
(1.55,1.54),(1.60,1.61),(1.59,1.66),(1.785,1.815),(1.62,1.745),\
(1.61,1.63),(1.54,1.585),(1.72,1.785),(1.71,1.785),(1.49,1.575),\
(1.67,1.70),(1.78,1.80),(1.52,1.575),(1.73,1.885),(1.62,1.68),\
(1.68,1.765),(1.66,1.725),(1.45,1.485),(1.71,1.79),(1.73,1.84),\
(1.65,1.68),(1.58,1.685),(1.63,1.635),(1.73,1.73),(1.58,1.56),\
(1.55,1.595),(1.625,1.67),(1.715,1.83),(1.49,1.55)])
x = puntos[:,0]     # la primera columna de puntos son las xs
y = puntos[:,1]     # la segunda columna son las ys
```

Llamamos a la función linregress:

```
In [2]:
from scipy import stats
m, b, R, p_value, SE = stats.linregress(x,y)
print("m:", m, "b:", b, "R:", R, "p_value:", p_value, "SE:", SE)
Out[2]:
m: 1.0872209258008656 b: -0.08257271475089834 R: 0.9126160834010653
   p_value: 1.403731868280971e-13 SE: 0.08747390050041842
```

Para terminar, vamos a encontrar el intervalo al 95 % de confianza de m:

```
In [3]:
t= stats.t.ppf((1+0.95)/2, len(x)-2)
ME= t*SE
print("ME:", ME)
print(round(m-ME,2), round(m+ME,2))
Out[3]:
ME: 0.17840419627934484
0.91 1.27
```

O sea que $m = 1.09 \pm 0.18$, o m está entre 0.91 y 1.27 al 95 % de nivel de confianza, por lo que no se descarta el ideal de Vitrubio, a ese nivel de confianza.

6.5.3. statsmodel.api.OLS

Veamos ahora el desempeño de la función OLS del módulo `statsmodel`. Esta función se ocupa cuando hay más de una variable independiente (como en el caso de las funciones de `sklearn`). Si ingresamos tal cual los datos en la función, el resultado va a carecer de la ordenada al origen:

```
In [4]:
import statsmodels.api as sm
model= sm.OLS(y,x).fit()  # la y va antes que la x!
print(model.summary())
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:
      0.999
Model:                            OLS   Adj. R-squared:
      0.999
Method:                 Least Squares   F-statistic:
   5.585e+04
Date:                Tue, 11 Sep 2018   Prob (F-statistic):
   1.87e-53
Time:                        18:06:11   Log-Likelihood:
     58.927
No. Observations:                  33   AIC:
     -115.9
Df Residuals:                      32   BIC:
     -114.4
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025
     0.975]
------------------------------------------------------------------------------
```

```
x1                 1.0368       0.004     236.330       0.000        1.028
       1.046
==============================================================================
Omnibus:                          0.427    Durbin-Watson:
       1.812
Prob(Omnibus):                    0.808    Jarque-Bera (JB):
       0.558
Skew:                             0.025    Prob(JB):
       0.756
Kurtosis:                         2.365    Cond. No.
        1.00
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
    correctly specified.
```

Antes de corregir por la falta de ordenada al origen, podemos ver en la parte central del reporte todo lo referente a la pendiente, identificada como x1: su valor m, SE, t^* al 95 % de confianza, p_value y extremos del intervalo al 95 % de confianza.

En la parte de arriba vemos a R^2 y R^2_{adj}. Ya dijimos lo que es el coeficiente de correlación R. También se usa R^2, que tiene el significado de ser el cociente entre la variabilidad de los datos explicada por el modelo, y la variabilidad de los datos total. Cuando hay más de una variable independiente, R^2 aumenta, el modelo está mejor explicado con más variables. Pero puede ser que en realidad unas variables no aporten mucho, podría haber redundancia. Así, la R^2_{adj} penaliza la introducción de más variables explicatorias. Si aumenta, es que estuvo bien meter una variable más, si no, es mejor quedarse hasta la variable anterior. En el reporte, al haber una sola variable, $R^2 = R^2_{\text{adj}}$, pero al ajustar para permitir la ordenada al origen R^2_{adj} va a cambiar, en este caso, va a disminuir.

Llamemos ahora a la misma función pero permitiendo la ordenada al origen. La diferencia es el agregado de la constante en la primera línea del código:

```
In [5]:
x= sm.add_constant(x)      # para poder tener ordenada al origen
model= sm.OLS(y,x).fit()   # la y va antes que la x!
print(model.summary())
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y    R-squared:
       0.833
Model:                            OLS    Adj. R-squared:
       0.827
Method:                 Least Squares    F-statistic:
       154.5
Date:                Tue, 11 Sep 2018    Prob (F-statistic):
    1.40e-13
Time:                        18:06:26    Log-Likelihood:
      59.104
No. Observations:                  33    AIC:
      -114.2
Df Residuals:                      31    BIC:
      -111.2
Df Model:                           1
```

```
15 Covariance Type:            nonrobust
16 ==============================================================================
17                  coef    std err          t      P>|t|      [0.025
     0.975]
18 ------------------------------------------------------------------------------
19 const         -0.0826      0.143     -0.577      0.568      -0.374
      0.209
20 x1             1.0872      0.087     12.429      0.000       0.909
      1.266
21 ==============================================================================
22 Omnibus:                       0.859   Durbin-Watson:
      1.837
23 Prob(Omnibus):                 0.651   Jarque-Bera (JB):
      0.775
24 Skew:                         -0.052   Prob(JB):
      0.679
25 Kurtosis:                      2.256   Cond. No.
       44.3
26 ==============================================================================
27
28 Warnings:
29 [1] Standard Errors assume that the covariance matrix of the errors is
    correctly specified.
```

Como vemos, ya aparece la información para los dos parámetros, la pendiente (`x1`) y la ordenada al origen (`const`). La información es consistente con nuestros cálculos previos.

7 Pandas

Uno de los temas que más auge ha tenido en los últimos años en el área de ciencias es la Ciencia de Datos. En este capítulo vamos ver una introducción al campo usando el módulo de `pandas` para leer archivos csv y darle sentido a los datos. Los archivos csv, separados por comas, son un tipo de archivo plano utilizado ampliamente para guardar información en tablas. Es similar al Excel, con la diferencia de que los archivos cvs no constan de varias pestañas, y de ahí el término plano. `Pandas` puede leer directamente archivos Excel, pero vamos a circunscribirnos a archivos csv. Los objetivos de aprendizaje de esta unidad, son que el lector aprenda a leer archivos csv, explore sus datos, los grafique, extraiga información de ellos, como información estadística, y los opere y manipule para los fines pertinentes. Para ahondar en la información de este tema, en la bibliografía se recomienda el manual en Ciencias de Datos en Python de Jake VanderPlas (2016).

7.1. Lectura y exploración de datos

El tipo de datos que se crea al leer tablas en `Pandas` es un `DataFrame`, o estructura de datos tabular, mientras que cada columna es una `Series`.

Vamos a utilizar algunos datos de la pandemia de coronavirus para ejemplificar y aprender a manipular los `DataFrames`. La pandemia inició en China en el 2019 y se propagó al resto del mundo a finales de ese año y principalmente en el 2020.

Veamos los datos de exceso de fallecimientos acumulados de algunos países. Éstos se obtienen comparando los fallecimientos que se esperan en un momento del año, respecto al comportamiento de los años anteriores. Los datos son el acumulado de esos números comenzando desde el 1 de enero de 2020. Este número da una aproximación más precisa del impacto real del coronavirus, al no catalogarse algunos de los fallecimientos como atribuible al coronavirus, debido en parte a la falta de pruebas.

En un archivo Excel, copia la Tabla 7.1 sobre el exceso de fallecimientos de algunos países (datos obtenidos de Our world in Data, https://github.com/owid/covid-19-data/tree/master/public/data).

Ahora, convierte el archivo en un archivo csv con "Guardar como" y selecciona el tipo de archivo CSV UTF-8 (delimitado por comas, el UTF-8 es para aceptar caracteres como letras con acentos). Llama al archivo exceso_mortalidad.

Para leer el archivo se importa `pandas` y se le da el alias `pd`, por costumbre. Como vamos

día	México	Francia	Italia	España
05/07/2020	96304	14838	44400	42178
04/10/2020	211167	18033	50096	52683
03/01/2021	324302	47363	102037	70087
04/04/2021	485482	55893	121843	76615
04/07/2021	501729	63230	140258	77972
03/10/2021	620989	67264	152767	88153
02/01/2022	647104	80073	168226	97313
03/04/2022	697921	84833	181854	98107

Tabla 7.1: Exceso de fallecimientos por el coronavirus, entre julio de 2020 y abril de 2022

a graficar, de una vez importamos también `pyplot`. Leamos el archivo, guardemos la información en la variable `exceso`, que es un `Dataframe`, y veamos su contenido imprimiendo los primeros 5 renglones para no tener que imprimir todo el archivo:

```
In [1]
import pandas as pd
import matplotlib.pyplot as plt
exceso= pd.read_csv("exceso_mortalidad.csv")
print(exceso.head())
Out[1]:
          día  México  Francia  Italia  España
0  05/07/2020   96304    14838   44400   42178
1  04/10/2020  211167    18033   50096   52683
2  03/01/2021  324302    47363  102037   70087
3  04/04/2021  485482    55893  121843   76615
4  04/07/2021  501729    63230  140258   77972
```

El método `head()` muestra los primeros cinco renglones, pero puede ponerse el número deseado como argumento. Vemos que la primera columna es un rango de números que comienza desde el 0, y que esa columna no existe en el archivo csv original. Esa columna, el `index`, no forma parte de los datos y nos va a servir para ubicarlos, así como los nombres de las columnas, `columns`, que en nuestro ejemplo son "día" y los nombres de los países. Podríamos dejar que los renglones sean ubicados por un número, pero sería más útil si los datos fueran ubicados por la fecha. Para que la columna de "día" pase a ser el `index`, volvemos a leer el archivo y usamos la opción `index_col` para asignarla (también se puede usar la función `pd.set_index()`):

```
In [2]:
exceso=pd.read_csv("exceso_mortalidad.csv",index_col="día")
print(exceso.head())
Out[2]:
            México  Francia  Italia  España
día
05/07/2020   96304    14838   44400   42178
04/10/2020  211167    18033   50096   52683
03/01/2021  324302    47363  102037   70087
04/04/2021  485482    55893  121843   76615
04/07/2021  501729    63230  140258   77972
```

Si hubiera otro tipo de información en los primeros renglones, se pueden saltar con la opción `skiprows` en el argumento. Por ejemplo, si hubiera 2 renglones antes del nombre de las columnas, se pondría `skiprows=2`. Si en cambio, no hubiera información de las columnas, lo ideal sería agregar nombres en el archivo Excel o csv directamente, pero por si alguna razón no se puede o debe modificar el archivo, entonces se usa la opción `header=None`, y los nombres de las columnas será su numeración empezando desde el cero.

Pidamos ahora información del tipo de datos y su número:

```
In [3]:
print(exceso.info())
Out[3]:
<class 'pandas.core.frame.DataFrame'>
Index: 8 entries, 05/07/2020 to 03/04/2022
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   México   8 non-null      int64
 1   Francia  8 non-null      int64
 2   Italia   8 non-null      int64
 3   España   8 non-null      int64
dtypes: int64(4)
memory usage: 320.0+ bytes
```

Si quisiéramos saber sólo el tamaño de la tabla, el nombre de las columnas o el nombre de los renglones, se puede llamar, respectivamente:

```
In [4]:
print(exceso.shape)
print(exceso.columns)
print(exceso.index)
Out[4]:
(8, 4)
Index(['México', 'Francia', 'Italia', 'España'], dtype='object')
Index(['05/07/2020', '04/10/2020', '03/01/2021', '04/04/2021', '
    04/07/2021', '03/10/2021', '02/01/2022', '03/04/2022'],
      dtype='object', name='día')
```

Ahora procedamos a graficar. Si se grafican todas las columnas a la vez, se hace un solo llamado a `plot`, donde el eje x es el `index`. Se pueden dar argumentos para graficar (Figura 7.1).

```
In [5]:
exceso.plot(figsize=(9,5),lw=10)  # lw, ancho de la línea
Out[5]:
<Axes: xlabel='día'>
```

Se ve que en México hubo un gran exceso de fallecimientos, de hecho fue el doble de lo que se atribuyó al coronavirus, aunque una porción puede explicarse por el fallecimiento de personas vulnerables que no asistieron a servicio médico por temor a contagiarse.

Aprovechando este mismo archivo, vamos a seleccionar ahora elementos de la tabla. En las tablas de datos, por lo general un nuevo dato se agrega con un renglón más, mientras las características de los datos se definen por columnas. Por ejemplo, al llenar datos

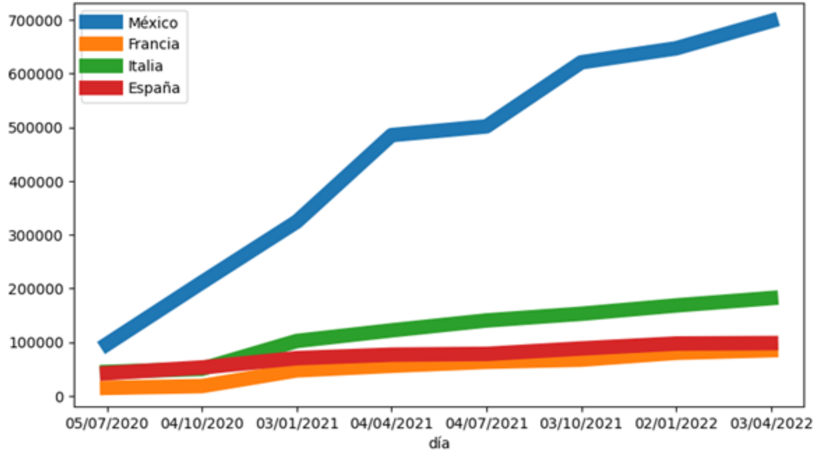


Figura 7.1: Exceso de mortalidad acumulada en el periodo del coronavirus, respecto al número de fallecimientos previstos según las tendencias de los años previos.

de estudiantes, las características como la matrícula, el nombre, el semestre, se colocan como título de las columnas, y si entra un nuevo estudiante se llena un nuevo renglón completo. Por lo mismo, si se quiere sólo un conjunto de características, se seleccionan esas columnas en particular. En `pandas`, por tanto, es más importante seleccionar la columna. Por ejemplo, para los datos `exceso`, podemos seleccionar a México como único argumento del DataFrame:

```
In [6]:
print(exceso["México"])
Out[6]:
día
05/07/2020     96304
04/10/2020    211167
03/01/2021    324302
04/04/2021    485482
04/07/2021    501729
03/10/2021    620989
02/01/2022    647104
03/04/2022    697921
Name: México, dtype: int64
```

Esto difiere de como funciona `numpy`, donde los renglones tienen preferencia sobre las columnas. Vemos que, además de los valores, aparece el `index` (día).

Si queremos sólo los valores, por alguna razón, los seleccionamos con el método `values`:

```
In [7]:
print(exceso["México"].values)
Out[7]:
[ 96304 211167 324302 485482 501729 620989 647104 697921]
```

Una sola columna como la que seleccionamos, deja de ser un `DataFrame` y es sólo una `Series`, mientras que cuando seleccionamos los valores de la columna, convertimos la serie en un arreglo de `numpy`:

```
In [8]:
print(type(exceso))
print(type(exceso["México"]))
print(type(exceso["México"].values))
Out[8]:
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.series.Series'>
<class 'numpy.ndarray'>
```

Si queremos una selección de columnas, en este caso, de países, se llaman con una lista como argumento del dataframe:

```
In [9]:
print(exceso[["México","España"]])
Out[9]:
            México  España
día
05/07/2020   96304   42178
04/10/2020  211167   52683
03/01/2021  324302   70087
04/04/2021  485482   76615
04/07/2021  501729   77972
03/10/2021  620989   88153
02/01/2022  647104   97313
03/04/2022  697921   98107
```

¿Qué tal si lo que queremos es un renglón o un conjunto de renglones?, en ese caso, no basta con llamar a los elementos con corchetes, se tiene que usar el método `loc` o `iloc`, dependiendo de si se llama al renglón por nombre o por índice. Por ejemplo, si se quiere saber el exceso de fallecimientos del 4 de julio del 2021 en todos los países, hacemos:

```
In [10]:
print(exceso.loc["04/07/2021"])
Out[10]:
México     501729
Francia     63230
Italia     140258
España      77972
Name: 04/07/2021, dtype: int64
```

O hacemos:

```
In [11]:
print(exceso.iloc[4])
Out[11]:
México     501729
Francia     63230
Italia     140258
España      77972
Name: 04/07/2021, dtype: int64
```

Si queremos varios renglones, lo hacemos con slice. Para el `loc` el slice es inclusivo, incluye el último elemento de la selección, y para `iloc` es exclusivo, no lo incluye.

```
In [12]:
print(exceso.loc["04/10/2020":"03/10/2021"])
Out[12]:
            México  Francia  Italia  España
día
04/10/2020  211167    18033   50096   52683
03/01/2021  324302    47363  102037   70087
04/04/2021  485482    55893  121843   76615
04/07/2021  501729    63230  140258   77972
03/10/2021  620989    67264  152767   88153
```

Si seleccionamos un renglón o varios, y algunas columnas en específico, también se usa `loc`:

```
In [13]:
print(exceso.loc["04/07/2021":"02/01/2022",["México","Italia"]])
Out[13]:
            México  Italia

día
04/07/2021  501729  140258
03/10/2021  620989  152767
02/01/2022  647104  168226
```

Finalmente, si se seleccionan columnas pero con slice, hay que usar también el método `loc`:

```
In [14]:
print(exceso.loc[:,"Francia":"España"])
Out[14]:
            Francia  Italia  España
día
05/07/2020    14838   44400   42178
04/10/2020    18033   50096   52683
03/01/2021    47363  102037   70087
04/04/2021    55893  121843   76615
04/07/2021    63230  140258   77972
03/10/2021    67264  152767   88153
02/01/2022    80073  168226   97313
03/04/2022    84833  181854   98107
```

Ejercicios:

a) Para la Tabla 7.2 sobre el total de pruebas de covid por cada mil habitantes en algunos países, grafica los datos en función del día:

b) Selecciona los datos de los días del 4 de abril de 2020 al 2 de enero de 2022 desde Alemania a España.

c) Selecciona los datos de México, Alemania y España al mismo tiempo

día	México	Alemania	Italia	España
05/07/2020	5.285	75.976	95.177	71.956
04/10/2020	14.132	215.782	198.92	218.899
03/01/2021	27.553	431.834	454.526	485.605
04/04/2021	46.417	618.57	864.277	793.681
04/07/2021	57.378	783.695	1221.139	1007.825
03/10/2021	82.808	904.044	1576.645	1260.465
02/01/2022	95.231	1120.184	2389.372	1554.438
03/04/2022	118.664	1466.496	3414.298	1871.436

Tabla 7.2: Total acumulado de pruebas por covid por cada mil habitantes

7.2. Agregación, filtrado, agrupamiento

Veamos algunas de las operaciones que se utilizan comúnmente al trabajar con tablas de datos. Para ello, vamos a tomar como ejemplo unos datos que muestran de manera sencilla cómo se realizan estas operaciones. En una encuesta, se les pide a algunos estudiantes de ciertas carreras y planteles de una universidad que evalúen dos programas de televisión dando un puntaje entre el 0 y el 5. Los resultados se muestran en la Tabla 7.3.

Copia los datos en un Excel y convierte el archivo en un archivo csv, con el nombre de `estud`. Empezamos leyendo y explorando el archivo:

```
In [1]:
import pandas as pd
estud=pd.read_csv("estud.csv")
print(estud.head())
Out[1]:
  matrícula plantel   carrera  prog 1  prog 2
0     15-095      DV      FeHI       3       5
1     13-005     SLT       NyS       2       4
2     16-206     SLT  Software       4       3
3     16-115      CL     CPyAU       5       4
4     14-204     Cua      ISEI       0       2
```

Como hay columnas numéricas, podemos darnos una idea de la estadística de éstas con el método `describe`:

```
In [2]:
print(estud.describe())
Out[2]:
          prog 1     prog 2
count  20.000000  20.000000
mean    2.750000   2.800000
std     1.517442   1.472556
min     0.000000   0.000000
25%     2.000000   2.000000
50%     3.000000   3.000000
75%     4.000000   4.000000
max     5.000000   5.000000
```

matrícula	**plantel**	**carrera**	**prog 1**	**prog 2**
15-095	DV	FeHI	3	5
13-005	SLT	NyS	2	4
16-206	SLT	Software	4	3
16-115	CL	CPyAU	5	4
14-204	Cua	ISEI	0	2
16-104	DV	CL	3	0
13-202	DV	FeHI	3	3
17-008	SLT	PS	2	1
15-221	SLT	PS	5	4
14-098	CL	CPyAU	1	4
13-119	Cua	NyS	3	2
14-283	CL	ISET	4	3
13-235	SLT	CPyAU	2	5
16-214	DV	CL	3	4
15-126	CL	ISEI	2	3
17-111	CL	ISET	5	3
16-201	Cua	NyS	1	2
14-180	DV	CPyAU	4	1
16-142	CL	CPyAU	0	0
16-108	SLT	NyS	3	3

Tabla 7.3: Calificación de 2 programas de televisión

Podríamos pedir por separado alguna variable de agregación como el promedio. Como ya se mencionó, las columnas tienen prioridad en pandas, así que por default el eje sobre el que se hace el promedio es `axis=0`, o columnas:

```
In [3]:
print(estud.mean())
Out[3]:
prog 1     2.75
prog 2     2.80
dtype: float64
```

Si quisiéramos el promedio de lo que opinó cada estudiante, es decir, de los renglones, hay que especificar el eje, `axis=1`:

```
In [4]:
print(estud.mean(axis=1))
Out[4]:
0      4.0
1      3.0
2      3.5
3      4.5
4      1.0
5      1.5
6      3.0
7      1.5
8      4.5
9      2.5
10     2.5
11     3.5
12     3.5
13     3.5
14     2.5
15     4.0
16     1.5
17     2.5
18     0.0
19     3.0
dtype: float64
```

Se puede obtener funciones de agregación de columnas por separado:

```
In [5]:
print(estud["prog 1"].mean())
Out[5]:
2.75
```

Podemos agregar columnas al DataFrame simplemente llamándolo con argumento el nombre de la nueva columna, y operando con las columnas ya existentes, si fuera el caso. Por ejemplo, podemos agregar una columna con la suma de los 2 puntajes:

```
In [6]:
estud["suma"]=estud["prog 1"]+estud["prog 2"]
print(estud.head())
Out[6]:
  matrícula plantel   carrera  prog 1  prog 2  suma
0    15-095      DV      FeHI       3       5     8
1    13-005     SLT       NyS       2       4     6
```

```
2     16-206      SLT   Software        4        3      7
3     16-115       CL      CPyAU        5        4      9
4     14-204      Cua        ISEI       0        2      2
```

Es sencillo operar numéricamente con las columnas, pero para operar con aquellas cuyos elementos sean cadenas primero hay que llamar a la función `str`, y utilizar sus métodos (por decir, `str.lower()`). En nuestro ejemplo, además de la información del plantel y carrera a la que pertenecen los estudiantes se proporciona la matrícula. Resulta ser que los 2 primeros números de la matrícula representan la generación del estudiante. Así que podemos extraer la generación con slicing, tomando los primeros 2 números:

```
In [7]:
estud["gen"]=estud["matrícula"].str[:2]
print(estud.head())
Out[7]:
  matrícula plantel   carrera  prog 1  prog 2  suma gen
0     15-095       DV      FeHI       3        5      8  15
1     13-005      SLT        NyS       2        4      6  13
2     16-206      SLT  Software       4        3      7  16
3     16-115       CL     CPyAU       5        4      9  16
4     14-204      Cua       ISEI       0        2      2  14
```

¿Cómo saber los valores distintos que aparecen en cada columna?, con el método `unique`, que da el conjunto de los valores únicos de la columna dada:

```
In [8]:
print(estud["carrera"].unique())
Out[8]:
['FeHI' 'NyS' 'Software' 'CPyAU' 'ISEI' 'CL' 'PS' 'ISET']
```

Si queremos filtrar renglones para un valor específico de una variable, se hace como en `numpy`. Por ejemplo, si filtramos a los estudiantes que pertenecen al plantel "CL":

```
In [9]:
print(estud[estud["plantel"]=="CL"])
Out[9]:
   matrícula plantel carrera  prog 1  prog 2  promedio  gen
3      16-115      CL   CPyAU       5       4       4.5  16
9      14-098      CL   CPyAU       1       4       2.5  14
11     14-283      CL    ISET       4       3       3.5  14
14     15-126      CL    ISEI       2       3       2.5  15
15     17-111      CL    ISET       5       3       4.0  17
18     16-142      CL   CPyAU       0       0       0.0  16
```

Pero si queremos filtrar un grupo mayor de valores, se usa el método `isin`:

```
In [10]:
print(estud[estud["carrera"].isin(["ISEI","ISET","Software"])])
Out[10]:
   matrícula plantel   carrera  prog 1  prog 2  promedio  gen
2      16-206     SLT  Software       4       3       3.5  16
4      14-204     Cua      ISEI       0       2       1.0  14
11     14-283      CL      ISET       4       3       3.5  14
14     15-126      CL      ISEI       2       3       2.5  15
15     17-111      CL      ISET       5       3       4.0  17
```

Por último, si queremos contar, o alguna otra operación, de acuerdo a características de un grupo, se usa la función `groupby`. Por ejemplo, podemos contar cuántas personas hay por plantel y por carrera de los estudiantes que contestaron la encuesta. Se agrupa entonces por atributo "plantel" y "carrera", en ese orden, y para contar se puede especificar como columna la "matrícula", pues si no se especifica nada se contaría sobre todas las columnas restantes:

```
In [11]:
plan_carr= estud.groupby(["plantel","carrera"])["matrícula"].count()
print(plan_carr)
Out[11]:
plantel  carrera
CL       CPyAU       3
         ISEI        1
         ISET        2
Cua      ISEI        1
         NyS         2
DV       CL          2
         CPyAU       1
         FeHI        2
SLT      CPyAU       1
         NyS         2
         PS          2
         Software    1
Name: matrícula, dtype: int64
```

Además de `count` se puede usar otra función de agregación, como `sum`, por ejemplo, si se quisiera sumar los puntajes de cada serie, agrupados por carrera y plantel.

Ejercicios:

Considera la Tabla 7.4 sobre películas de animación por computadora, rankeada por su calificación en rottentomatoes, y donde en la columna de "protagonista", se describe al personaje o personajes principales:

a) Selecciona las películas de Disney, y obtén el porcentaje promedio de calificaciones. Haz lo mismo con las películas de Pixar.

b) Agrupa los datos por el tipo de protagonista, y cuenta el número de películas que pertenece a cada categoría de protagonista.

7.2.1. Ejercicio de práctica: coronavirus

En este ejercicio vamos a retomar el tema del coronavirus. Vamos a descargar un archivo de una página donde la Universidad de Oxford (Our World in Data) hospeda información sobre este tema. Ve a la página https://github.com/owid/covid-19-data/tree/master/public/data y en la parte de "Download our complete COVID-10 dataset" descarga el archivo csv, owid-covid-data.csv, en el mismo folder donde lo analices con un archivo de `python`. Echa un vistazo al archivo para ver la información contenida. En la misma pagina puedes encontrar información sobre el significado de las columnas. Viendo el archivo, vemos que los países se encuentran en la columna `''location''`, pero

película	**protagonista**	**animación**	**%**
Toy Story 2	juguete	Pixar	100
Toy Story	juguete	Pixar	100
Buscando a Nemo	animal	Pixar	99
Cómo entrenar a tu dragón	persona	DreamWorks	99
Intensamente	persona	Pixar	98
Toy Story 3	juguete	Pixar	98
Zootopia	animal	Disney	98
Up	persona	Pixar	98
Toy Story 4	juguete	Pixar	97
Spider man: un nuevo universo	súper héroe	Sony pictures	97
Coco	persona	Pixar	97
Los Increíbles	súper héroe	Pixar	97
La familia Mitchell vs las máquinas	persona	Sony pictures	97
La gran aventura LEGO	juguete	Warner	96
Ratatouille	animal	Pixar	96
Monsters, Inc	fantástico	Pixar	96
Soul	persona	Pixar	95
Moana	persona	Disney	95
Wall-E	robot	Pixar	95
El gato con botas: el último deseo	animal	DreamWorks	95
Buscando a Dory	animal	Pixar	94
Raya y el último dragón	persona	Disney	93
Monstruo del mar	fantástico	Sony pictures	93
Los Increíbles 2	súper héroe	Pixar	93
El Principito	persona	Method animation	92
Operación regalo	persona	Sony pictures	92
Antz: Hormiguitaz	animal	DreamWorks	92
Bichos: una aventura en miniatura	animal	Pixar	92
Cómo entrenar a tu dragón 2	persona	DreamWorks	92
LEGO Batman: la película	juguete	Warner	90
Cómo entrenar a tu dragón 3	persona	DreamWorks	90
Frozen	persona	Disney	90
Bolt	animal	Disney	90
Grandes héroes	súper héroe	Disney	90
Shrek 2	fantástico	DreamWorks	89
Enredados	persona	Disney	89

Tabla 7.4: Películas de animación por computadora

también pueden ser identificados por su código en la columna ‘‘iso_code’’. La fecha de los datos se encuentra en la columna ‘‘date’’.

Vamos entonces a importar al archivo con pandas y seleccionar para hacer gráficas y para análisis en general a México, Japón, Chile, Corea del Sur, España, Alemania e Italia. Checa que todo esté bien explorando con el método head() (no mostramos el resultado por ser largo):

```
In [1]:
import pandas as pd
import matplotlib.pyplot as plt
todos=pd.read_csv("owid-covid-data.csv")
todos=todos[todos["iso_code"].isin(["MEX", "CHL", "ESP", "ITA", \
"DEU", "KOR", "JPN"])]
print(todos.head())
```

Como siempre, queremos poner como índice la columna de la fecha. Podemos hacer esto con index_col, como en la sección pasada, pero vamos a usar la función set_index como alternativa.

En la sección pasada tuvimos pocos datos y no fue muy necesario manejar de forma correcta la columna de fechas, pero lo mejor es usar el método to_datetime de pandas sobre todo para mostrar figuras sin los números encimados. Procedemos entonces a cambiar el formato de la columna de las fechas a datetime, y la convertimos en nuestro index. Restringimos también los datos seleccionando las fechas desde el 1 de abril de 2020 hasta el 1 de abril de 2022. Como el formato datetime cambia el orden de los días, meses y años (día/mes/año), al seleccionar las fechas hay que ser cuidadosos de seguir el nuevo formato (año-mes-día).

```
In [2]:
todos["date"]= pd.to_datetime(todos["date"])
todos= todos.set_index("date")
todos= todos.loc["2020-04-01":"2022-04-01"]
print(todos.head())
```

Enseguida, vamos a seleccionar sólo algunas de las columnas a graficar, la de pruebas por cada mil habitantes (‘‘total_tests_per_thousand’’), la de exceso de mortalidad acumulada por millón de habitantes (‘‘excess_mortality_cumulative_per_million’’) y la de total de dosis de vacunas por cada cien habitantes (‘‘total_vaccinations_per_hundred’’). Escribimos en una lista los nombres de las columnas y en otra los títulos correspondientes para las gráficas. Para graficar, usamos un bucle for para seleccionar la columna y el título, y un for interno para seleccionar cada país (en lugar de usar una lista explícita con los países, abreviamos con la lista generada con el método unique de la columna de países). A diferencia de lo que vimos en la sección 6.1, vamos a agregar uno a uno a cada país en la gráfica con pyplot. También, en lugar de plot graficamos puntos con scatter para enfatizar los saltos de información (Figura 7.2, Figura 7.3 y Figura 7.4).

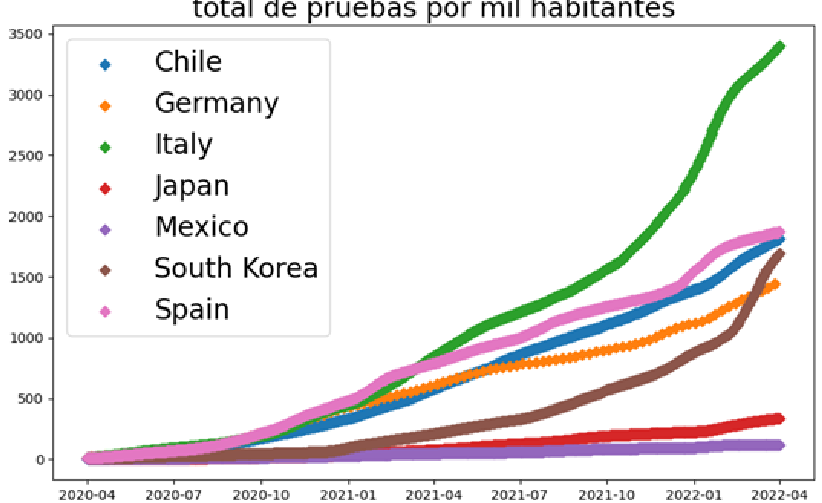


Figura 7.2: Total de pruebas por mil habitantes.

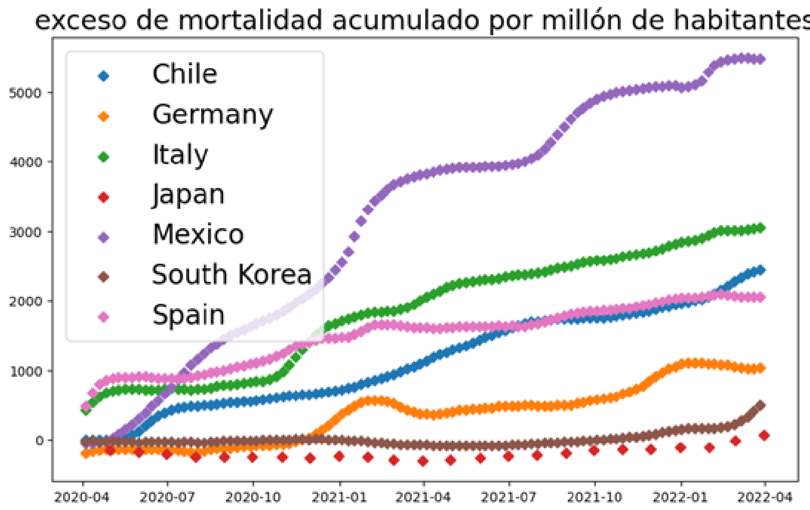


Figura 7.3: Exceso de mortalidad acumulada por millón de habitantes.

```
In [3]:
columnas=["total_tests_per_thousand",\
 "excess_mortality_cumulative_per_million",\
"total_vaccinations_per_hundred"]
títulos=["total de pruebas por mil habitantes","exceso de mortalidad\
    acumulado por millón de habitantes","total de dosis de vacunas por\
     cada cien habitantes"]
for j,k in zip(columnas,títulos):
    plt.figure(figsize=(10,6))
    for i in todos["location"].unique():
        a = todos[todos["location"]==i]  # seleccionamos el país
        plt.scatter(a.index,a[j],label=i,marker="D",s=30)
        # eje x los días, eje y la columna seleccionada
    plt.legend(fontsize=20)
    plt.title(k,fontsize=20)
    plt.show()
```

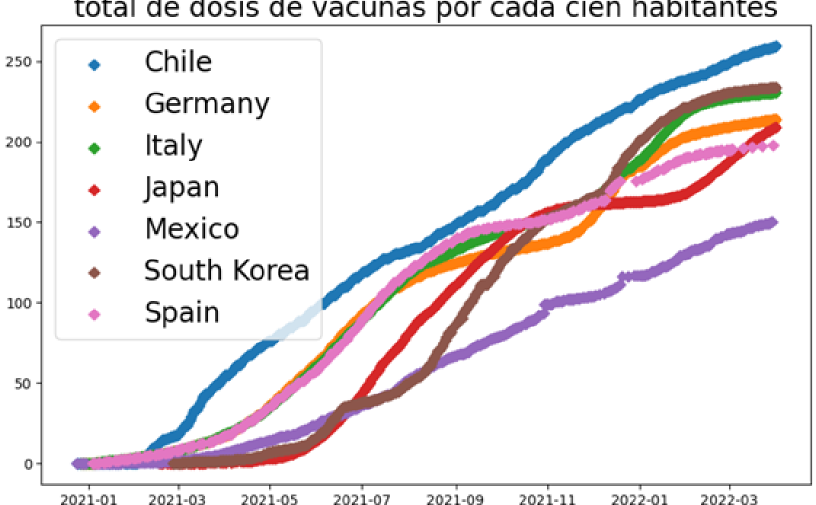


Figura 7.4: Total de vacunas por cada 100 habitantes (no es el total de vacunados, pues se puede repetir la dosis).

Ejercicios:

Usa el archivo del coronavirus para hacer tus propias indagaciones con los países y columnas deseadas.

8 Machine Learning

En este capítulo revisaremos temas de Machine Learning, traducido al español como Aprendizaje Automático o Aprendizaje de Máquina. Dejamos el título en inglés por ser el término más popular, pero a lo largo del capítulo usaremos su nombre en español. Hay mucho material sobre Aprendizaje Automático, en la bibliografía recomendamos el manual de Ciencias de datos de Jake VanderPlas (2016) y de Aprendizaje Automático de Aurélien Géron (2017) y de Andreas Müller (2017). En términos computacionales, vamos a usar el módulo `sklearn` (abreviación de `scikit-learn`). En este capítulo explicaremos el uso de ese módulo, pero también analizaremos la teoría y las matemáticas que hay detrás de los algoritmos que lo componen. Sobre este último punto, en la bibliografía se recomienda el curso en línea de Aprendizaje Automático de edX de Dasgupta Sanjoy, de la Universidad de California en San Diego.

El Aprendizaje Automático esencialmente se refiere al entrenamiento de un programa computacional con varios datos con fines de entenderlos y clasificarlos, y predecir el comportamiento de nuevas mediciones. Se divide en aprendizaje supervisado y aprendizaje no supervisado. El aprendizaje supervisado se refiere a entrenar modelos donde sabemos las características de los datos, de forma que nos permita predecir el comportamiento de nuevos datos. Los modelos que se ven en este tipo de aprendizaje pueden ser de tipo clasificación o de tipo regresión. Por otro lado, en el aprendizaje no supervisado no sabemos el comportamiento a priori de los datos, y dejamos que ellos mismos se expliquen. Ejemplos de este tipo de aprendizaje es el clustering y la reducción de dimensiones.

Los objetivos de aprendizaje de este capítulo son el entender las matemáticas y aplicar los diferentes métodos de Aprendizaje Automático en problemáticas varias. Los métodos a revisar son, supervisados de clasificación: vecinos más cercanos, árbol de decisión, regresión logística, support vector machines y gaussian naive bayes, y métodos no supervisados: análisis de componentes principales y clustering. El tema de la regresión se vio en el capítulo 6. El tema de Deep learning con las redes neuronales no lo vamos a ver aquí, ése es un tema para revisar con más amplitud en otro lado.

8.1. Métodos supervisados: Clasificación

Las tareas de clasificación se refieren a separar grupos de datos diferenciados por una categoría, como color, sí o no, etc. La etiqueta del grupo es una variable categórica. Antes de ver los diferentes métodos de clasificación, seleccionemos los datos con los

que vamos a trabajar, escogiendo datos que están guardados en la misma biblioteca de sklearn.

8.1.1. Preparación y exploración de datos: Flores iris

Para ejemplificar algunos de los métodos de clasificación, vamos a usar los famosos datos de la flor iris, que es un conjunto de datos bastante utilizado didácticamente justamente por ya estar cargado en la biblioteca. Este conjunto de datos lo vamos a usar a lo largo de todo el capítulo, así que, aunque no se escriba explícitamente en las secciones siguientes, vamos a hacer uso de los módulos que importemos y las variables que definamos de inicio, por lo que es buena idea guardar todas las instrucciones en un solo archivo de Python. La numeración de los In[n] y Out[n] consecuentemente la vamos a seguir de corrido.

Los datos son sobre tres tipos de flores iris, Setosa, Versicolor y Virginica, y se miden cuatro características de las flores: largo y ancho del sépalo, y largo y ancho del pétalo (puedes ver las imágenes de las flores y otra forma de preparar los datos en https://www.kaggle.com/code/sunaysawant/iris-eda). El tipo de iris es la etiqueta con la cual vamos a clasificar los datos.

Vamos a importar el submódulo datasets de sklearn para cargar los datos, y empezamos a explorarlos:

```
In [1]:
from sklearn import datasets
iris = datasets.load_iris()
print(dir(iris))
Out[1]:
['DESCR', 'data', 'feature_names', 'target', 'target_names']
```

Las medidas de los sépalos y pétalos están guardadas en data, que es un arreglo de numpy donde por cada renglón hay mediciones de una flor, y las columnas denotan las variables a medir: el largo y ancho del sépalo (columnas 0 y 1) y el largo y ancho del pétalo (columnas 2 y 3). Esa es la forma como deben guardarse los datos, ya sea que se tengan arreglos de numpy, DataFrames de pandas o hasta listas de listas, las columnas deben denotar a las variables medidas y los renglones a cada nuevo experimento. En target, un arreglo de numpy tipo vector, está guardada la información, por renglón, de la etiqueta o tipo de flor usando los valores 0, 1 y 2. Esto es, por cada renglón de data, le corresponde el tipo de flor en target, 0 por Setosa, 1 por Versicolor y 2 por Virginica. En target_names están tal cual los nombres del tipo de flor, y por eso sabemos el orden, y en feature_names las variables a medir (largo y ancho del sépalo y el pétalo).

Extraemos la información y exploramos su contenido:

```
In [2]:
X= iris.data; y= iris.target
nombres= iris.target_names; var =iris.feature_names
print(nombres)
print(var)
print(y)
print(X.shape, y.shape)  # tamaño de los datos
```

```
Out[2]:
['setosa' 'versicolor' 'virginica']
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal
    width (cm)']
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2
  2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
(150, 4) (150,)
```

Son 150 datos, y vemos que las flores no están puestas al azar, están ordenadas primero las setosa, luego las versicolor y al final las virginica.

La notación común es seleccionar X para los datos de las variables independientes, y y para la variable categórica. Las variables independientes son así `x_0=X[:,0]`, la primera columna, `x_1=X[:,1]`, la segunda columna, etc., (hay que recordar que la numeración en `Python` empieza con cero). Si trabajáramos desde `pandas`, `x_0=X["nombre_0"]`, etc.

Antes de hacer cualquier cosa con Aprendizaje Automático hay que terminar de explorar bien los datos. Grafiquemos entonces, de una sola vez, todas las combinaciones de pares de variables, para ver sus relaciones. Existe la función `scatter_matrix` en `pandas` para hacerlo, donde los elementos en la diagonal son los histogramas de las variables independientes. Se aplicaría: `scatter_matrix`(nombre del Dataframe [nombres de las columnas en una lista, y como opcional el tamaño de la figura con `figsize]`). Como vamos a seguir en `numpy`, hagámoslo a mano (Figura 8.1):

```
In [3]:
import matplotlib.pyplot as plt
k=0     # k es un contador para llevar la cuenta de la subfigura
plt.figure()
for j in range(4):          # por ser 4 variables
    for i in range(4):
        k+=1
        plt.subplot(4,4,k)  # 4 renglones, 4 columnas, figura k
        # la columna X[:i] para una variable, X[:,j] para la otra
        plt.scatter(X[:,i],X[:,j], c=y, cmap="rainbow")
        # c=y, color de puntos de acuerdo a la etiqueta 0, 1 o 2
        if j==3:  # cuando en el eje y está la última variable
            plt.xlabel(var[i])  # escribe el nombre de la variable
        if i==0:  # cuando en el eje x está la primera variable
            plt.ylabel(var[j],fontsize=7)
plt.tight_layout()
plt.show()
```

Para que aparecieran nada más los nombres de las variables en los ejes cartesianos, a la izquierda y en la parte de abajo de la figura, usamos unos `if` internos para que aparecieran sólo en algunas subfiguras.

Si aumenta el número de variables, graficar en una forma tipo matriz va a hacer difícil ver con detalle las figuras. Además, hay redundancia al graficar x_i vs x_j y luego x_j vs x_i, además de graficar las variables contra sí mismas, x_i vs x_i. Grafiquemos entonces eligiendo el número de columnas deseadas, y entonces dejando sólo el número de renglones necesarios. Si n es el número de variables y cada subfigura requiere de dos variables,

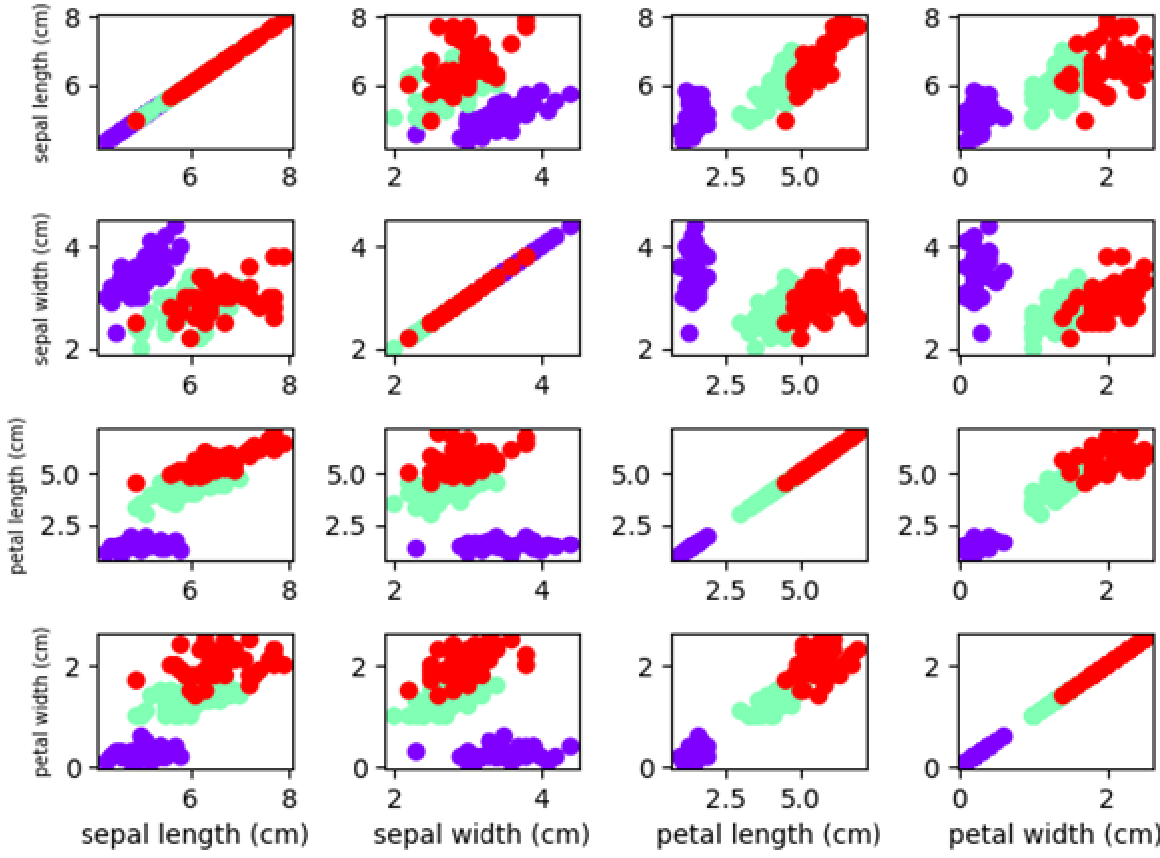


Figura 8.1: Gráficas en 2D de las combinaciones de las cuatro variables de los datos de las flores iris. Hay redundancia de información al invertir el orden de las variables, x_i vs x_j y x_j vs x_i, y al graficar cada variable contra sí misma, x_i vs x_i.

sin redundancia, entonces, las combinaciones posibles son:

$$C_n^2 = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$$

En el ejemplo de las iris, con 4 variables, sólo requerimos de 6 subfiguras. Supongamos que queremos graficar 3 subfiguras por renglón, es decir, tener 3 columnas, entonces, el total de renglones que se necesitan son el número de combinaciones entre 3. Como este número en general podría no ser entero, podemos usar la función `ceil` (de ceiling, techo), para quedarnos con el entero redondeado más alto (Figura 8.2).

```
In [4]:
n=len(var)
k=0   # k lleva la cuenta de la figura
combinaciones= n*(n-1)/2
columnas= 3
renglones= int(np.ceil(combinaciones/columnas))
plt.figure()
# i hasta n-1 y j desde i+1 para evitar redundancias
for i in range(n-1):
    for j in range(i+1,n):
        k+=1
        plt.subplot(renglones,columnas,k)
        plt.scatter(X[:,i],X[:,j],c=y,cmap="rainbow")
        plt.xlabel(var[i])
        plt.ylabel(var[j])
plt.tight_layout()
plt.show()
```

En cada una de las subfiguras se ve que los datos se separan decentemente, según el tipo de flor, pero de primera vista la mejor elección para representar los datos en 2 dimensiones sería tomar el largo y ancho de los pétalos (las 2 últimas columnas de `datos` o `X`). Esto lo vamos a hacer para que, por un lado, visualmente sean más claros los ejercicios de clasificación que veremos en las siguientes secciones, y por el otro, para trabajar con menos dimensiones.

Grafiquemos entonces respecto a las variables `x_2=X[:,2]` y `x_3=X[:,3]`, agregando la etiqueta de a qué tipo de flor pertenecen los datos. Para ello usamos un `for` para seleccionar el tipo de flor, 0, 1 o 2, y poner el color y etiqueta respectivos (Figura 8.3):

```
In [5]:
colores=["lime","r","blue"]
x2=X[:,2]; x3=X[:,3]
plt.figure()
for k in range(3):
    plt.scatter(x2[y==k],x3[y==k],c=colores[k],label=nombres[k])
    # con y == k seleccionamos los datos por condición
plt.xlabel(var[2],fontsize=20)
plt.ylabel(var[3],fontsize=20)
plt.legend(fontsize=15)
plt.show()
```

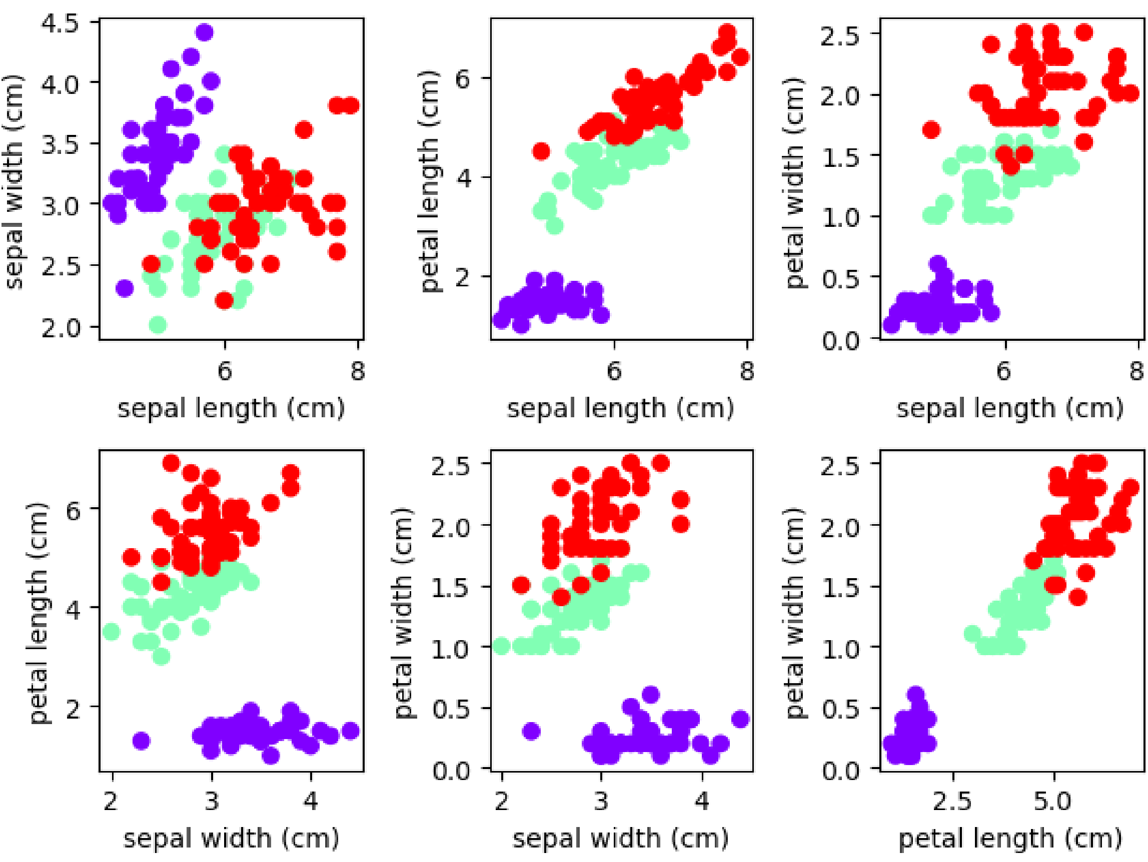


Figura 8.2: Dos renglones y tres columnas para graficar en 2D las seis combinaciones únicas de las cuatro variables.

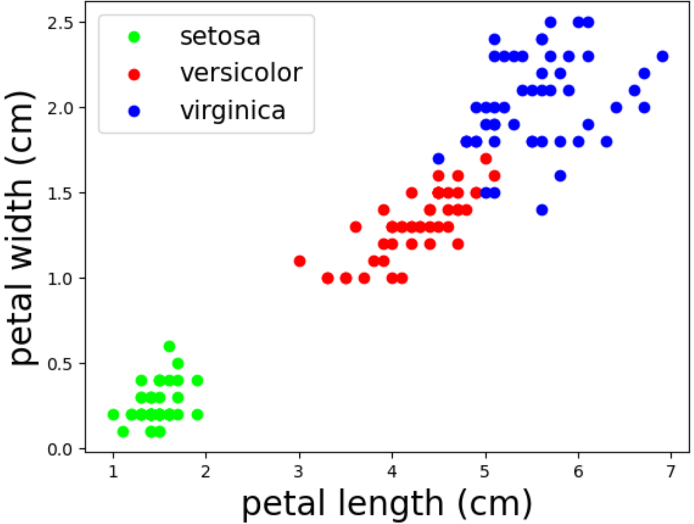


Figura 8.3: Selección de las dos variables de las últimas columnas para graficar los datos de las flores iris, el largo y ancho del pétalo.

Ejercicios:

1. En un archivo de `Python` independiente, para no mezclar con las flores iris, con el que seguiremos trabajando en las secciones subsecuentes, carga los datos sobre vinos, que se encuentra igualmente en el módulo `datasets` y se carga con la función `load_wine` (`datasets.load_wine`)
2. Explora el contenido de vinos con la función `dir`
3. ¿Cuántas son las variables que se midieron? Imprime el nombre de las variables
4. ¿Cuántas categorías de vinos hay? Imprime el nombre de las categorías
5. Guarda la información de los variables independientes en la matriz X y las categorías en y.
6. ¿Cuántos datos o mediciones hay?, encuentra el número.
7. Explora graficando pares de variables
8. Si clasificáramos usando sólo 2 variables, ¿cuál del siguiente par de variables arrojarían mejores resultados? (tiene menos superposición de puntos):
a) `proanthocyanis` vs `hue`
b) `non_flavanoid_phenols` vs `hue`
c) `total_phenols` vs `color_intensity`

8.1.2. Datos de entrenamiento y prueba y preparación para clasificar

Para aplicar un método de clasificación, primero tenemos que dividir los datos en un conjunto de entrenamiento (`train`) y uno de prueba (`test`), para que el modelo aprenda de los datos de entrenamiento y se pueda evaluar con los datos de prueba. La división la podríamos hacer por nuestra cuenta usando el módulo `random`, pero en `sklearn` ya existe la función `train_test_split`, que divide los datos de forma aleatoria, y, que al igual que en el módulo `random`, se puede dar una semilla o seed, para poder reproducir la división aleatoria, mediante la función `random_state`. Por default, el número de datos para entrenar es el 75 % de los datos. Los argumentos de `train_test_split` son los datos X y el target y, y agregamos la opción del `random_state`. Como ya comentamos anteriormente, en todas las secciones seguiremos trabajando con las flores iris, así que no hay que olvidar correr las instrucciones previas donde se importan módulos y se definen variables.

```
In [6]:
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =\
train_test_split(X, y,random_state=5)
```

Antes de ver cada método de clasificación en particular, vamos a ver cómo se aplican los métodos en general en `sklearn`.

El procedimiento comienza llamando al método que se va a utilizar con sus argumentos respectivos. En seguida se aplica la función `fit` con argumentos la X y y de entrenamiento, tal cual para entrenar los datos. Posteriormente se prueba con la función `predict` con argumento la X de prueba, cuyos resultados se comparan con la y de prueba. Los pasos se verían así:

- modelo = método(argumentos).`fit`(X entrenamiento, y de entrenamiento)
- predicción = modelo.`predict`(X de prueba)
- comparar la predicción con las y de prueba

Podríamos entrenar los datos con todas las variables, pero vamos en nuestro caso a seleccionar sólo las 2 que ya mencionamos, la del largo y ancho del pétalo. Definamos la función en general con los índices `i,j` de las columnas deseadas (dependiendo de los datos se pueden seleccionar todas las variables, un conjunto de ellas, o un par de ellas como en nuestro ejemplo):

```
1  In [7]:
2  def predice(método,i,j,X_train,y_train,X_test,y_test):
3      modelo= método.fit(X_train[:,[i,j]],y_train)
4      pred= modelo.predict(X_test[:,[i,j]])
5      error= np.sum(pred!= y_test)/len(y_test)
6      return(modelo, error)
```

Esta función la vamos a aplicar para cada uno de los métodos de las secciones siguientes.

Para no sólo tener el error cometido, sino visualizar los datos, vamos a definir una función para graficar, no sólo los datos, sino para colorear la zona de graficación según lo que predice el método utilizado. Uno de los argumentos de la función para graficar es el modelo retornado por la función `predice` de arriba, para predecir el color de las zonas. Vamos a construir una malla y colorear cada punto de ésta:

```
1  In [8]:
2  colores=["lime","r","blue"]
3  colores1=["olive","pink","dodgerblue"]
4  def zona(modelo,i,j,X_train,y_train,X_test,y_test,título,estilo):
5      plt.figure()
6      # se prepara el área para colorear, con los valores mínimo
7      # y máximo de las variables
8      xmin= min(np.min(X_train[:,i]),np.min(X_test[:,i]))
9      ymin= min(np.min(X_train[:,j]),np.min(X_test[:,j]))
10     xmax= max(np.max(X_train[:,i]),np.max(X_test[:,i]))
11     ymax= max(np.max(X_train[:,j]),np.max(X_test[:,j]))
12     # se encuentra el rango para redefinir los mínimos y máximos
13     # y definir una buena zona de graficación
14     xrange= xmax-xmin; yrange= ymax-ymin
15     xmin= xmin-xrange/10; xmax= xmax+xrange/10
16     ymin= ymin-yrange/10; ymax= ymax+yrange/10
17     # la malla o grid donde se va a colorear cada punto:
18     xx, yy= np.meshgrid(np.linspace(xmin,xmax,300),\
19             np.linspace(ymin,ymax,300))
20     grid=np.c_[xx.ravel(),yy.ravel()]
21     # se predice en los puntos de la malla:
22     Z= np.array([modelo.predict(a.reshape(1,2)) \
23           for a in grid]).reshape(xx.shape)
24     # se colorean los puntos de la malla, según la predicción Z:
25     plt.pcolormesh(xx,yy,Z,cmap=estilo)
26     # se colorean los puntos de entrenamiento y de prueba con
27     # diferente color:
28     for k in range(3):
29         plt.scatter(X_train[y_train==k,i],\
30          X_train[y_train==k,j], color=colores[k],\
31          label=nombres[k]+", train",s=50)
```

```
32          plt.scatter(X_test[y_test==k,i],\
33           X_test[y_test==k,j], color=colores1[k], s=50,\
34           label=nombres[k]+", test")
35      plt.xlabel(var[i])
36      plt.ylabel(var[j])
37      plt.legend()
38      plt.title(título)
39      if título=="clasificación con SVM":   # si el modelo es SVM
40          plt.scatter(modelo.support_vectors_[:,0],\
41            modelo.support_vectors_[:,1],\
42            facecolors="none",edgecolor="k",s=100)
43      plt.show()
```

Dentro de la función `zona`, para preparar la zona de graficación tomamos el mínimo de entre el mínimo de los datos de entrenamiento y de prueba, y lo mismo con el máximo. Podríamos tomar los valores mínimos y máximos directamente de los datos totales `X`, pero más adelante necesitaremos adecuar esta función en caso de un pre-procesamiento, así que conviene dejarlo así. También, como argumento de la función dimos el `estilo`, para darnos vuelo con los `colormap`. Unas pocas de las funciones que aparecen en la función `zona` no lo vimos en el capítulo de `numpy`. No te preocupes, hay funciones que aparecen de vez en cuando y cuando se requieren, se busca la información en internet. Como dijimos, `numpy` es todo un mundo.

8.1.3. Vecinos más cercanos

Este método es de los más sencillos. Dado un punto que queremos clasificar, buscamos cuál es la categoría de los k vecinos más cercanos para asignarle la categoría más frecuente al punto. Se escoge un valor de k impar para que haya menos posibilidades de empate. Cuando se escogen muchos vecinos la aproximación es gruesa, porque se ve de manera efectiva el dominio de los grupos, mientras que, si el número de vecinos es pequeño, es como hacer un zoom y ver las interacciones locales. La distancia por default entre los puntos es la distancia euclidiana. Se podría dejar como ejercicio implementar esta función, al ser cálculos sencillos.

Comparemos qué sucede cuando aplicamos el método de k vecinos más cercanos con $k = 1$ y $k = 9$. Importamos primero la función `KNeighborsClassifier`:

```
1 In [9]:
2 from sklearn.neighbors import KNeighborsClassifier
```

En la función `predice`, llamamos a esa función con $k = 1$. Hay más parámetros que se pueden cambiar, pero no vamos a utilizar. Por ejemplo, si quisiéramos que los puntos más cercanos tuvieran mayor peso, se podría cambiar el parámetro `weights`, de “`uniform`”, que es el valor por default, a “`distance`”, donde la influencia disminuye con el inverso de la distancia. Escogiendo el largo y el ancho del pétalo como variables, con índices 2 y 3, respectivamente:

```
1 In [10]:
2 modelo, error =\
3 predice(KNeighborsClassifier(1),2,3,X_train,y_train,X_test,y_test)
4 print(error)
5 zona(modelo,2,3,X_train,y_train,X_test,y_test,\
```

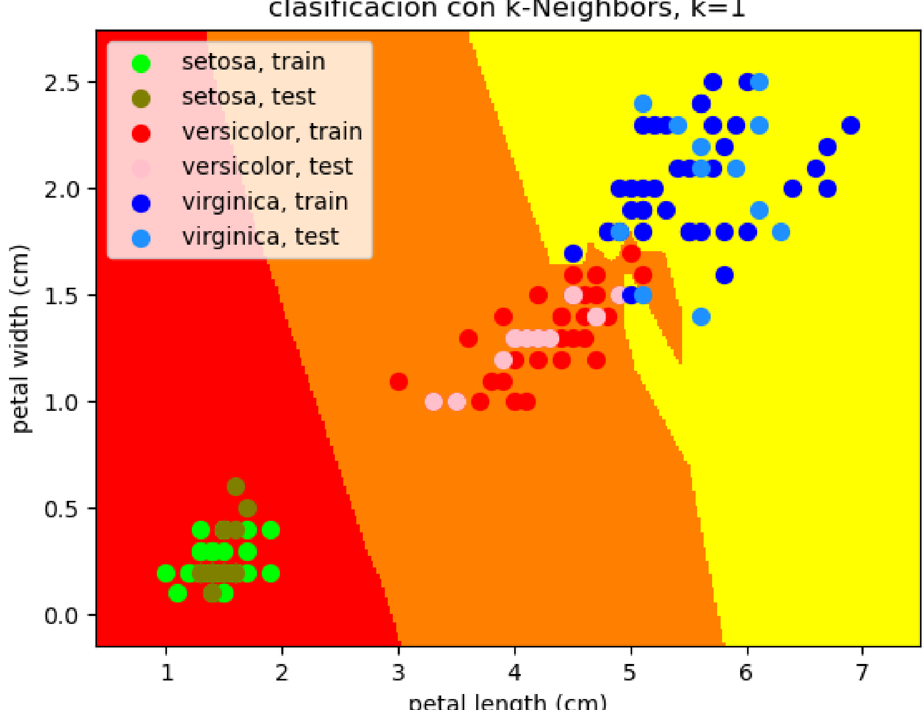


Figura 8.4: Clasificación de los datos de las flores iris con vecinos más cercanos. Con sólo un vecino, el modelo queda sobreajustado.

```
6  "clasificación con k-Neighbors, k=1","autumn")
7 Out[10]:
8 0.05263157894736842
```

A comparar con:

```
1 In [11]:
2 modelo, error =\
3 predice(KNeighborsClassifier(9),2,3,X_train,y_train,X_test,y_test)
4 print(error)
5 zona(modelo, 2, 3, X_train, y_train, X_test, y_test,\
6 "clasificación con k-Neighbors, k=9","autumn")
7 Out[11]:
8 0.05263157894736842
```

Comparando las figuras (8.4) y (8.5), vemos que, con más vecinos, los bordes son más suaves. Aunque para cada conjunto de datos se trataría de probar diferentes valores de k, una buena elección sería entre $k = 3$ y $k = 5$, pues $k = 1$ sobre ajusta los datos y valores grandes de k dan resultados más burdos.

Ejercicios:

1. Considera los puntos en el plano cartesiano, "azules", $(0,3)$, $(2,2)$ y $(2,4)$, y los puntos "rojos", $(5,3)$, $(5,1)$ y $(6,4)$. Si hay que clasificar al punto $(3,2)$, con $k = 1$ vecinos más cercanos, ¿en qué categoría de color quedaría el punto?, ¿y si se utilizara vecinos más cercanos con $k = 3$?

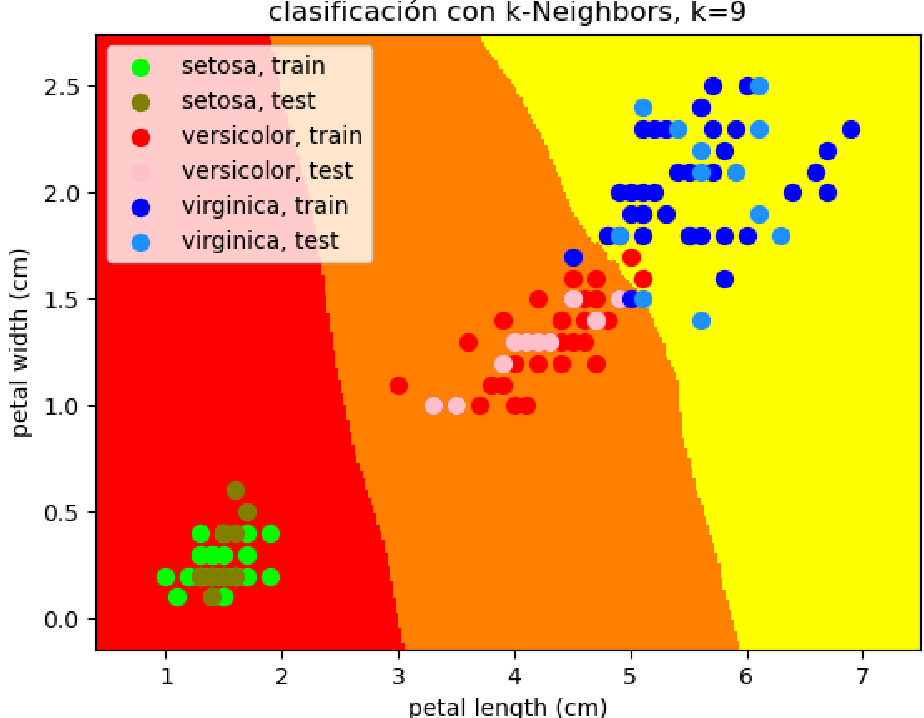


Figura 8.5: Clasificación con vecinos más cercanos. Con un número mayor de vecinos, el modelo se relaja.

2. Volvemos a usar los datos de los vinos, que se quedó como ejercicio de la primera sección. Separa los datos en datos de entrenamiento y de prueba usando la función `train_test_split` con los valores por default, más el `random_state` de tu gusto. Selecciona las variables “`alcohol`” y “`flavanoids`”, con índices 0 y 6, respectivamente, para entrenar y probar los datos, y para graficar usando Vecinos más cercanos con $k = 1$ y $k = 11$. ¿Cuál de las 2 elecciones reduce el error y hace ver mejor las fronteras?

8.1.4. Árbol de decisión

Este método también es muy intuitivo. Se basa en responder preguntas como en el juego de adivina un animal: ¿es cuadrúpedo?, sí o no, y se continúa con las preguntas. Un ejemplo de un árbol de decisión se muestra en la Figura 8.6, con el juego de adivina un personaje de Harry Potter:

Las variables independientes X pueden ser de tipo categórico, como un sí o un no, o numérico, como preguntar si algo mide más de cierta cantidad. Igualmente, la variable dependiente y puede ser categórica, por lo que sería un problema de clasificación, o numérica, un problema de regresión.

Veamos con el ejemplo de las flores iris cómo quedan clasificadas las zonas usando este método y después explicamos cómo funciona el algoritmo.

La función que requerimos es la función `DecisionTreeClassifier`:

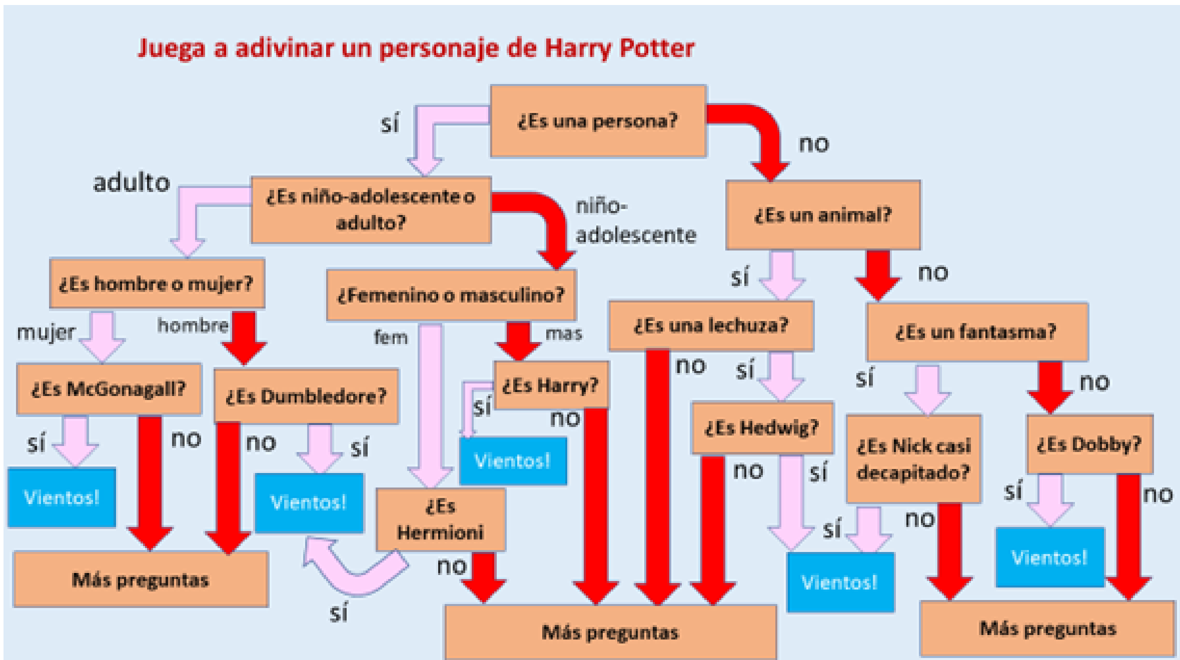


Figura 8.6: Ejemplo de árbol de decisión sobre personajes de Harry Potter

```
1 In [12]:
2 from sklearn.tree import DecisionTreeClassifier
3 modelo, error = predice(DecisionTreeClassifier(),2,3,X_train,y_train,
     X_test,y_test)
4 print(error)
5 zona(modelo,2,3,X_train,y_train,X_test,y_test,\
6      "clasificación con árbol de decisión","prism")
7 Out[12]:
8 0.05263157894736842
```

Lo que hace el programa es hacer pasar líneas entre los puntos, horizontales para la variable del ancho del pétalo, y verticales para el largo del pétalo. En cada paso, selecciona la división que mejor separe a los puntos por categoría, usando criterios que explicaremos más adelante. Rompe empates de manera aleatoria, por lo que no siempre se obtienen los mismos resultados. Por ejemplo, en el resultado de la Figura 8.7, la línea horizontal que divide los puntos de la setosa del resto pudo haber sido el primer paso. La siguiente división pudo ser la línea horizontal que divide a los otros 2 tipos de iris.

Vemos que la Figura 8.7 se ve un poco rara, pues por ejemplo hay una barra vertical verde en la zona de las versicolor. Esto se debe a que los puntos de la versicolor y de la virginica tienen un poco de traslape, y el algoritmo sigue haciendo divisiones para dejar a los puntos en la categoría correcta. A esto se le llama sobreajuste (overfitting). Antes de tratar de remediar la situación, corramos otras veces el programa para obtener otra posible configuración. El resultado se muestra en la Figura 8.8:

Para evitar el sobreajuste, se pueden buscar las opciones de la función `DecisionTreeClassifier` con `help(DecisionTreeClassifier)`. La opción `max_depth` nos puede ayudar para reducir la profundidad del árbol, hacerlo más corto. En la siguiente corrida, se selecciona `max_depth=3` (Figura 8.9):

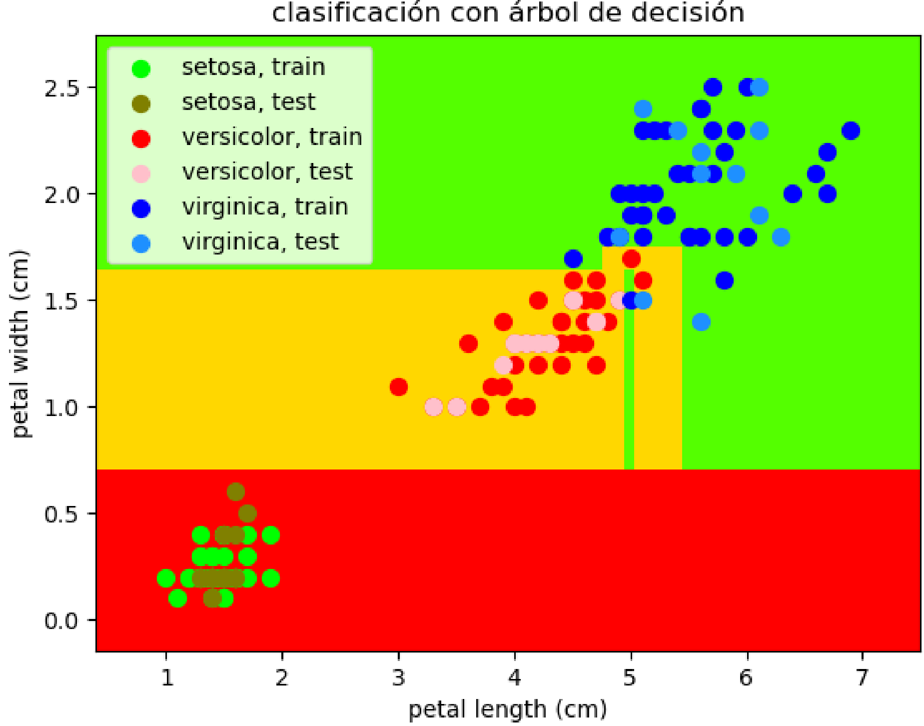


Figura 8.7: Clasificación de los datos de las flores irirs con el método de árbol de decisión.

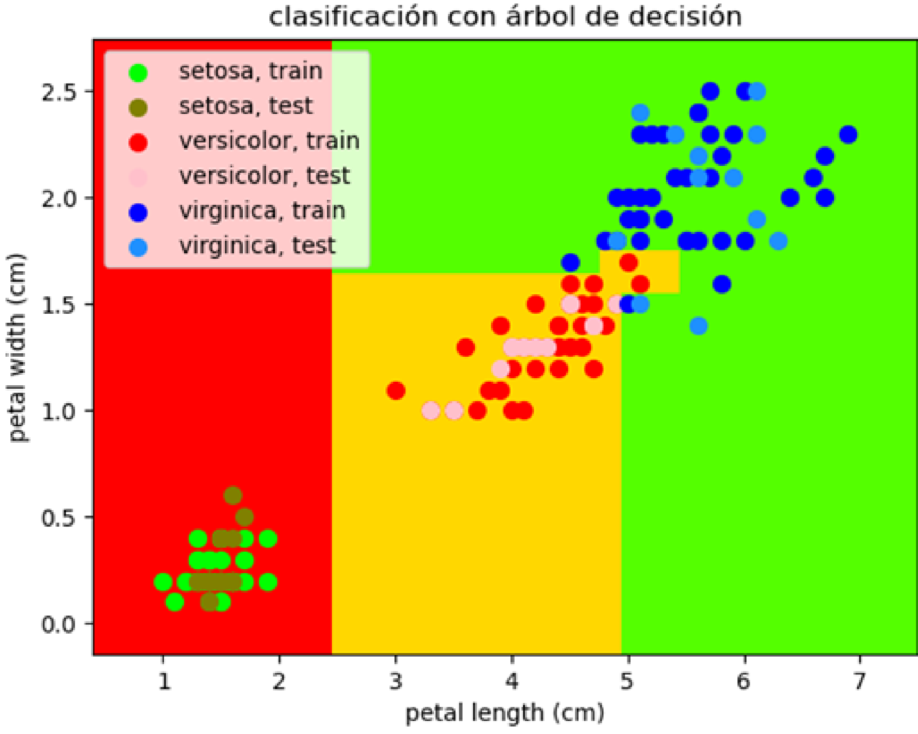


Figura 8.8: Otra configuración para la clasificación de los datos con árbol de decisión, con el mismo valor del error en los datos de prueba.

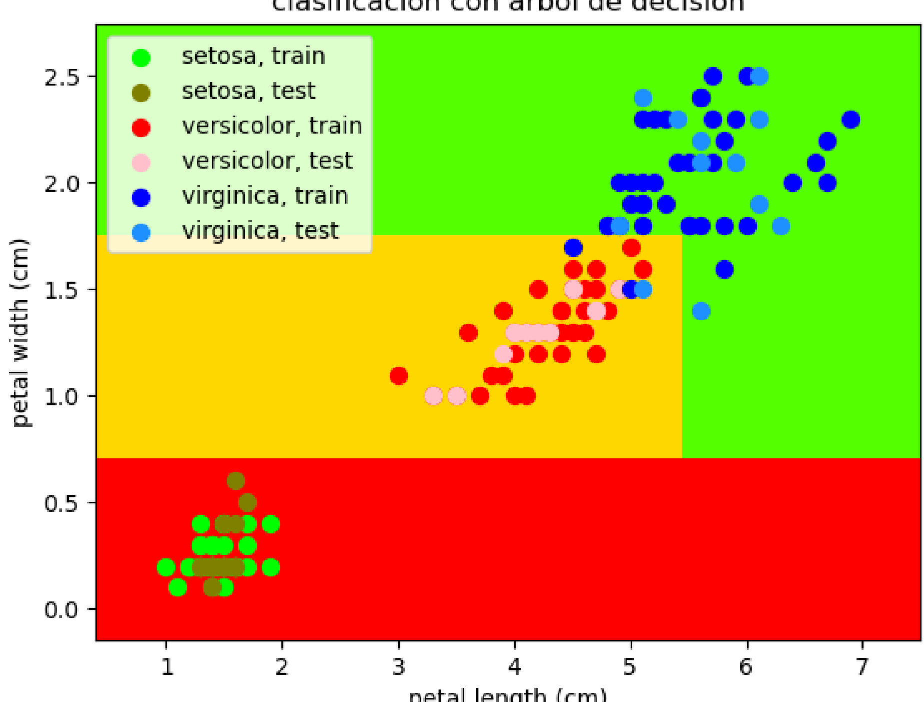


Figura 8.9: Clasificación de los datos con árbol de decisión, con una altura del árbol igual a 3 (o 3 bifurcaciones).

```
In [13]:
modelo, error= predice(DecisionTreeClassifier(max_depth=3),2,3,\
    X_train,y_train,X_test,y_test)
print(error)
zona(modelo,2,3,X_train,y_train,X_test,y_test,\
    "clasificación con árbol de decisión","prism")
Out[13]:
0.05263157894736842
```

¿Cuál es el criterio para la mejor partición en cada paso? En `sklearn` hay tres medidas, la impureza de Gini, la ganancia de información (entropía) y la pérdida logística (log_loss). Sólo mencionaremos la Impureza de Gini, que es el criterio por default que usa `DecisionTreeClassifier`.

En la impureza de Gini, se mide la proporción de datos de cierta categoría que quedan en una rama del árbol cuando se divide. Está dado por la expresión:

$$I_G = 1 - \sum_{i=1}^{m} f_i^2$$

donde m es el número de categorías y f la frecuencia de los datos de cada una de ellas. Vamos a graficar el árbol de decisión creado, y analicemos en cada paso la impureza de Gini para entenderla. Usamos para ello la función `plot_tree` (Figura 8.10):

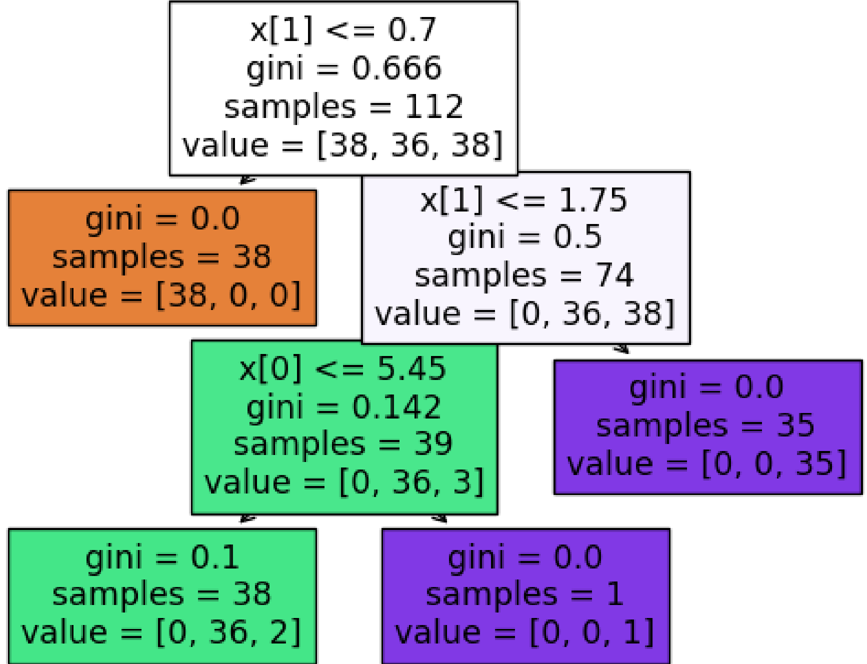


Figura 8.10: Árbol de decisión asociado a la figura (8.9), mostrando el índice de Gini en cada grupo. La bifurcación a la izquierda responde positivamente a la pregunta, a la derecha la respuesta es negativa.

```
In [14]:
from sklearn.tree import plot_tree
plot_tree(modelo, filled=True)
Out[14]:
[Text(0.4, 0.875, 'x[1] <= 0.7\ngini = 0.666\nsamples = 112\nvalue =
    [38, 36, 38]'),
 Text(0.2, 0.625, 'gini = 0.0\nsamples = 38\nvalue = [38, 0, 0]'),
 Text(0.6, 0.625, 'x[1] <= 1.75\ngini = 0.5\nsamples = 74\nvalue = [0,
    36, 38]'),
 Text(0.4, 0.375, 'x[0] <= 5.45\ngini = 0.142\nsamples = 39\nvalue =
    [0, 36, 3]'),
 Text(0.2, 0.125, 'gini = 0.1\nsamples = 38\nvalue = [0, 36, 2]'),
 Text(0.6, 0.125, 'gini = 0.0\nsamples = 1\nvalue = [0, 0, 1]'),
 Text(0.8, 0.375, 'gini = 0.0\nsamples = 35\nvalue = [0, 0, 35]')]
```

Al principio, hay un solo grupo de 112 elementos (recordemos que estamos trabajando con los datos de prueba, el 75 % de los 150 datos originales). Hay 38 datos de setosa y el mismo número de virginica, y hay 36 datos de versicolor. Se ingresan estos datos en la fórmula de la Impureza de Gini, quedando en este paso como:

$$I_G = 1 - \left(\frac{38}{112}\right)^2 - \left(\frac{36}{112}\right)^2 - \left(\frac{38}{112}\right)^2 = 0.666$$

Se hace la primera división, horizontal, según si el ancho del pétalo es menor o mayor a 0.7 cm. Las setosas quedan así separadas en el grupo del ancho del pétalo ≤ 0.7 cm,

quedando su Impureza de Gini:

$$I_G = 1 - \left(\frac{38}{38}\right)^2 - \left(\frac{0}{38}\right)^2 - \left(\frac{0}{38}\right)^2 = 0$$

Mientras que en el segundo grupo su Impureza está dada por:

$$I_G = 1 - \left(\frac{0}{74}\right)^2 - \left(\frac{36}{74}\right)^2 - \left(\frac{38}{74}\right)^2 = 0.5$$

Se vuelve a hacer una división horizontal, para dividir a las flores según si el ancho del pétalo es menor a 1.75 cm (pero mayor a 0.7 cm), creando 3 franjas horizontales. De esta forma quedan separadas las virginicas en la parte superior, el Índice de Gini ahí es cero, pero en la franja de en medio todavía hay 3 virginicas mezcladas con las 36 versicolor. Se hace una tercera y última división, donde se parte verticalmente la franja de en medio, según si el largo del pétalo es mayor o menor a 5.45 cm, dejando todavía 2 virginicas mezcladas con las versicolor. Como seleccionamos que la profundidad del árbol fuera de 3, ahí para el algoritmo.

Ejercicios:

1. Continuamos con el ejercicio de los datos de los vinos, seleccionando las columas 0 y 6 de ``alcohol'' y ``flavonoids''. Corre el árbol de decisión con las opciones por default, y después selecciona `max_depth=3` y `max_depth=2`. Corre algunas veces el programa para cada selección. ¿Cuál es la mejor elección del parámetro `max_depth`? Explica.
2. Dibuja el árbol de decisión creado con `max_depth=2` con `plot_tree`.

8.1.5. Regresión Logística

Este tipo de clasificación (a pesar de la palabra "regresión"), consiste en una frontera lineal del tipo $w \cdot x + b = 0$, que divide a 2 regiones, donde $w \cdot x + b > 0$ de un lado de la frontera y $w \cdot x + b < 0$ del otro. Aquí x denota a cualquiera de los n puntos $x^{(i)}$ que vive en d dimensiones (d variables), por lo que w también tiene d dimensiones. En 2 dimensiones, $w \cdot x + b = 0$ es una recta, en 3 dimensiones un plano, y en más dimensiones un hiperplano.

La idea es dar una probabilidad de que un punto que esté de un lado de la frontera tenga más del 50 % de pertenecer a una categoría dada. Supongamos que las 2 categorías se distinguen por tener $y = 1$ o $y = -1$ (equivalentes a sí o no, o algo del estilo). Entonces, la función logística cumple con lo que solicitamos, pues:

$$P(y = 1|x) = \frac{1}{1 + e^{-(w \cdot x + b)}}$$

cuando $w \cdot x + b = 0$, $P(y = 1|x) = 0.5$, cuando $w \cdot x + b > 0$, $P(y = 1|x) > 0.5$ y cuando $w \cdot x + b < 0$, $P(y = 1|x) < 0.5$ (Figura 8.11).

La probabilidad para el caso $y = -1$ es igual a $1 - P(y = 1|x)$, esto es:

$$P(y = -1|x) = 1 - \frac{1}{1 + e^{-(w \cdot x + b)}} = \frac{e^{-(w \cdot x + b)}}{1 + e^{-(w \cdot x + b)}} = \frac{1}{1 + e^{w \cdot x + b}}$$

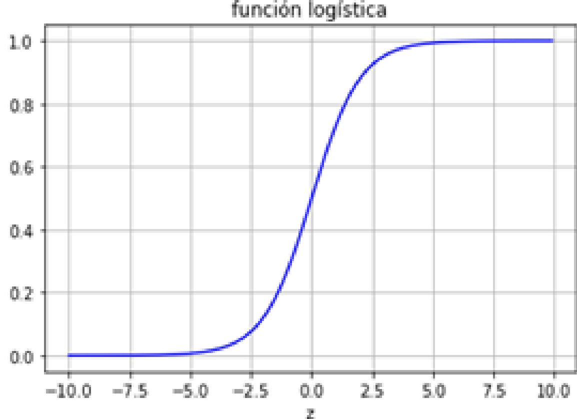


Figura 8.11: Función logística.

Por lo que de forma general, se puede tener la siguiente expresión, independientemente del valor de y:

$$P(y|x) = \frac{1}{1 + e^{-y(w \cdot x + b)}}$$

Lo que hace el programa es encontrar los mejores parámetros w y b mediante la optimización de una función de costo $L(w, b)$, la cual debe minimizarse. A partir de la probabilidad de cada punto $x^{(i)}$, se puede encontrar la probabilidad de todos los puntos, lo cual buscamos maximizar. La función de costo entonces se puede tomar como el negativo del producto de las probabilidades, para convertirlo en un problema de minimización. Más aún, para facilitar la operación, se toma el logaritmo de la función, para pasar de un producto de términos a una suma, y considerando que el logaritmo es una función creciente (cuando la base es mayor que 1, en este caso, e):

$$\prod_{i=1}^{n} P\left(y|x^{(i)}\right) \quad \rightarrow \quad L(w, b) = -\sum_{i=1}^{n} \ln P\left(y|x^{(i)}\right) = \sum_{i=1}^{n} \left(1 + e^{-y(w \cdot x^{(i)} + b)}\right)$$

Para encontrar los valores óptimos en aprendizaje automático se utiliza el método numérico de gradient descent, o alguna variación de éste, que de forma sencilla significa lo siguiente. Dada una función convexa a minimizar $L(w)$, si escogemos aleatoriamente un valor de w y si la derivada de la función evaluada en ese valor es negativa, esto es, la recta tangente al punto está inclinada hacia la izquierda, entonces sabemos que para reducir el valor de la función debemos movernos a la derecha para mejorar la predicción de w. Si por el contrario, la derivada es positiva, sabemos que nos debemos mover a la izquierda para mejorar la predicción de la w que minimiza la función (Figura 8.12)

Iterativamente entonces se realiza el procedimiento que mueve a la izquierda o a la derecha la w:

$$w_{i+1} = w_i - \eta \frac{dL}{dw}$$

donde η es el tamaño del paso. El procedimiento termina cuando en un siguiente paso $L(w)$ aumenta, en lugar de disminuir. Como en general son varios los parámetros a optimizar, la derivada se reemplaza por un gradiente. Algunos de los problemas de

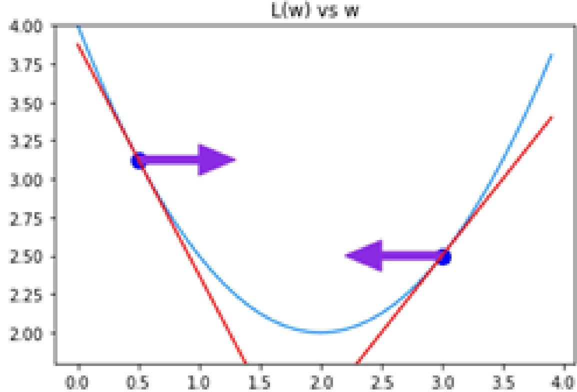


Figura 8.12: La minimización por gradient descent consiste en iterativamente moverse a la izquierda o a la derecha, según el signo de la derivada de la función convexa a minimizar, reduciendo en cada paso el valor de la función.

aprendizaje automático presentan funciones de costo convexas, donde hay un mínimo global, pero hay casos en que la función parece más como una montaña rusa, por lo que no se garantiza obtener la mejor solución global, al poder caer en un mínimo local. Como la minimización la hace `sklearn`, no nos preocupamos de ello, al menos para este libro.

Se puede generalizar la Regresión Logística para tener más de 2 categorías. Para dos categorías, podemos reescribir la ecuación logística como

$$P(y=1|x) = \frac{e^{w\cdot x+b}}{1+e^{w\cdot x+b}}$$

Si ahora hay m categorías en total, y si asociados a cada categoría k tenemos los parámetros w_k y b_k, entonces, la probabilidad de que un punto pertenezca a la categoría k se puede escribir como

$$P\left(y=k|x\right) = \frac{e^{w_k\cdot x+b_k}}{\sum_{i=1}^{m} e^{w_i\cdot x+b_i}}$$

Por lo que un punto x podrá ser catalogado en la categoría k si el numerador en la expresión se maximiza con esa k.

Implementemos la clasificación con Regresión Logística en el problema de las iris. Hacemos (Figura 8.13):

```
In [15]:
from sklearn.linear_model import LogisticRegression
modelo, error = predice(LogisticRegression(),2,3,X_train,y_train,
    X_test,y_test)
print(error)
zona(modelo,2,3,X_train,y_train,X_test,y_test,\
    "clasificación con Regresión Logística","Purples")
Out[15]:
0.05263157894736842
```

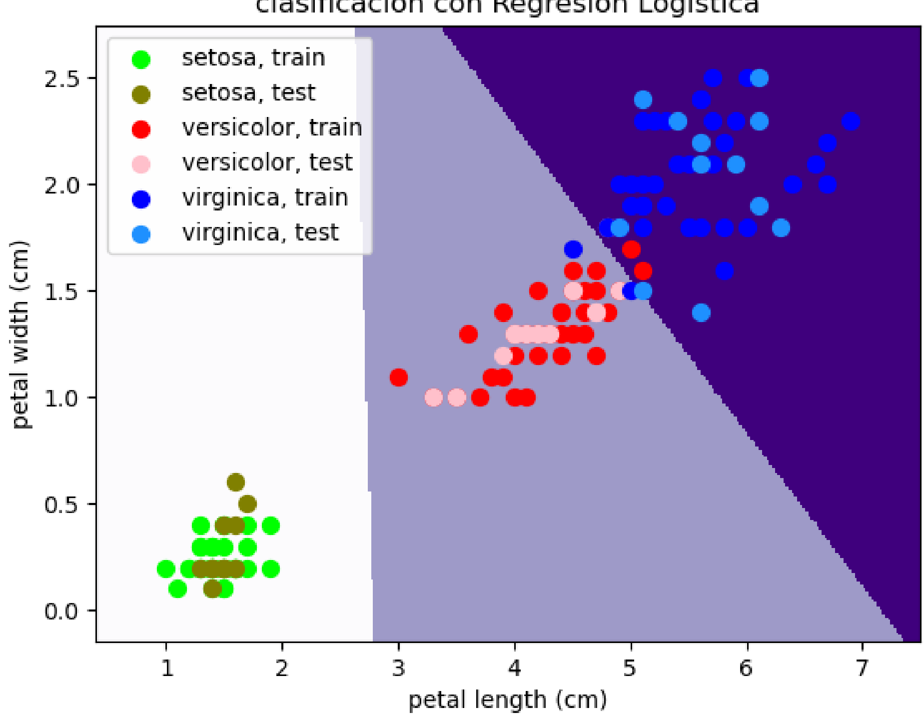


Figura 8.13: Clasificación de las flores iris con regresión logística

Ejercicios:

1. Continuando con el ejercicio de los datos de los vinos, con las columnas 0 y 6 de "`alcohol`" y "`flavonoids`", corre el programa con el método de regresión logística.

8.1.6. Support Vector Machines

Partimos nuevamente de la expresión lineal $w \cdot x + b = 0$. De nueva cuenta, en el caso de 2 categorías separamos 2 zonas con esa línea, pero no de manera arbitraria, pues, en principio, pueden ser varias las líneas que dividan a las 2 zonas. Lo que queremos es tener una franja bien delimitada entre los puntos, como una carretera, de manera que se maximice el margen, o grosor de ésta, y la línea fronteriza estaría a la mitad. Podemos escoger w y b (reescalar la ecuación) de tal forma que las líneas que conforman el margen sean las ecuaciones $w \cdot x + b = 1$ y $w \cdot x + b = -1$,

Recordando un poco de Geometría Analítica, en 2 dimensiones la distancia de un punto (x_0, y_0) a la recta $Ax + By + C = 0$ es igual a $d = |Ax_0 + By_0 + C|/\sqrt{A^2 + B^2}$, por lo que, si calculamos la distancia del punto a 2 rectas paralelas, la diferencia de esas distancias nos da la distancia entre las rectas paralelas. En nuestro caso, la distancia γ entre la recta $w \cdot x + b = 0$ y cualquiera de las rectas que conforman el margen es:

$$\gamma = \frac{w \cdot x + b + 1 - (w \cdot x + b)}{\sqrt{w_1^2 + w_2^2}} = \frac{1}{|w|}$$

Por lo que maximizar el margen 2γ, equivale a minimizar $|w|$, o mejor, $|w|^2$ (siempre es mejor maximinar o minimizar cuadrados que valores absolutos).

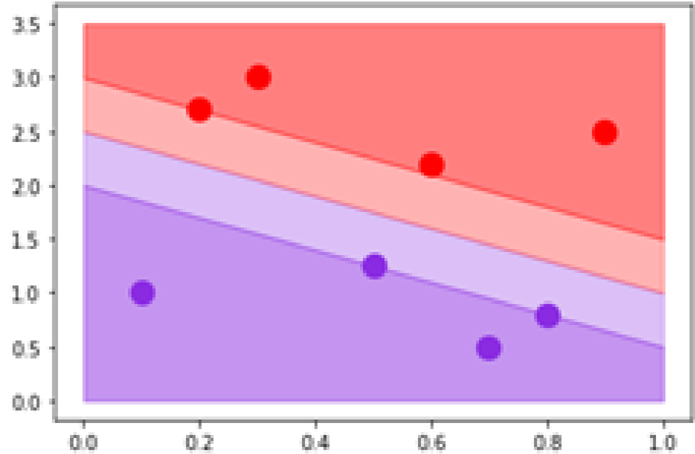


Figura 8.14: Cuando los puntos están divididos por áreas bien delimitadas, se marca la línea $w \cdot x + b = 0$, que es la frontera, y las líneas $w \cdot x + b = 1$ y $w \cdot x + b = -1$, que son las que marcan el margen, se busca maximizar éste. Los puntos que quedan sobre las líneas del margen se llaman vectores soporte. Los puntos con $w \cdot x + b \geq 1$ pertenecen a una categoría, y con $w \cdot x + b \leq -1$ pertenecen a la otra.

Como un ejemplo de reescalamiento de ecuaciones, supongamos que tenemos los puntos de la Figura 8.15:

Las rectas que delimitan la frontera y los márgenes son:

$$x_2 = -x_1 + 7, \quad x_2 = -x_1 + 5, \quad x_2 = -x_1 + 9$$

La recta central no difiere en 1 unidad de las rectas de los márgenes, difiere en dos unidades, así que se pueden reescalar las ecuaciones dividiendo por la mitad, para obtener las siguientes ecuaciones, escritas en la forma $w \cdot x + b = 0$ y $w \cdot x + b = \pm 1$:

$$\frac{1}{2}x_1 + \frac{1}{2}x_2 - \frac{7}{2} = 0,$$

$$\frac{1}{2}x_1 + \frac{1}{2}x_2 - \frac{5}{2} = 0 \quad \rightarrow \quad \frac{1}{2}x_1 + \frac{1}{2}x_2 - \frac{7}{2} = -1$$

$$\frac{1}{2}x_1 + \frac{1}{2}x_2 - \frac{9}{2} = 0 \quad \rightarrow \quad \frac{1}{2}x_1 + \frac{1}{2}x_2 - \frac{7}{2} = 1$$

Por lo que $w_1 = w_2 = 1/2$ y $b = -7/2$. Por supuesto, esto es parte de la explicación de los pasos, pero `sklearn` hace todo por nosotros.

¿Qué pasa si los puntos están mezclados cerca de la frontera y no hay posibilidad de dividir las zonas? En ese caso, hay que permitir un poco de laxitud, que haya puntos que puedan estar dentro del margen. Incluso, puntos que estén del lado contrario a su categoría. Lo que se hace entonces, es tratar de minimizar la función de costo:

$$L(w) = |w|^2 + C \sum_{i=1}^{n} \xi_i$$

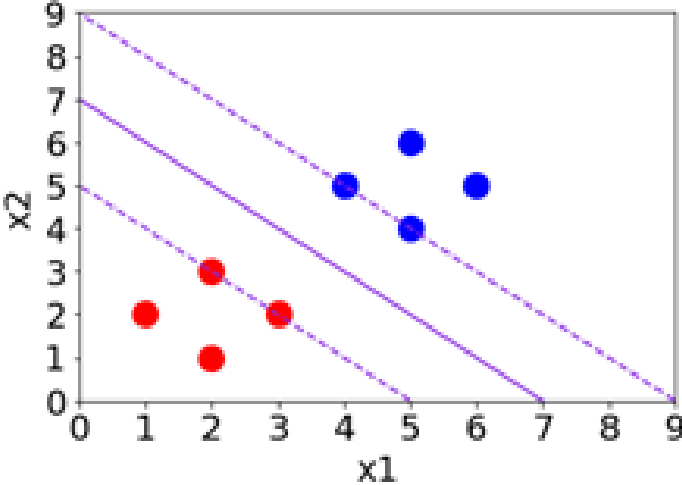


Figura 8.15: Puntos en el plano, para ejemplificar el reescalamiento de las ecuaciones.

donde cada ξ_i, con $\xi_i \geq 0$, es la posible violación por ingresar al margen o pasarse del otro lado de la categoría del punto $x^{(i)}$ (son n puntos). Los puntos $x^{(i)}$ estarían sujetos entonces a la ecuación:

$$y^{(i)}\left(w \cdot x^{(i)} + b\right) \geq 1 - \xi_i$$

Hay que notar que, cuando se tiene $w \cdot x + b = 0$, estamos sobre la frontera, y cuando $w \cdot x + b = \pm 1$, estamos sobre los márgenes. Seleccionando el margen $w \cdot x + b = 1$, se tendrían las siguientes situaciones de violación de los márgenes:

* $\xi = 0$, no hay violación del margen, los puntos quedan bien separados del lado del grupo que les toca ($w \cdot x + b \geq 1$).
* $0 < \xi \leq 1$, hay violación de los márgenes, pero aún $0 \leq w \cdot x + b < 1$, el punto está entre la frontera y el margen correcto del grupo del punto.
* $1 < \xi \leq 2$, hay violación de los márgenes y $-1 \leq w \cdot x + b < 0$, el punto está entre la frontera y el margen del otro lado del grupo que le corresponde al punto.
* $\xi > 2$, el punto ya dejó los márgenes y está del lado incorrecto del grupo que le toca.

Lo anterior se puede visualizar en la Figura 8.16, donde sólo se representan puntos "rojos" que se van adentrando a la zona "azul":

Para la mayoría de los puntos, $\xi = 0$, pues no están dentro del margen o del lado que no les corresponde. Cuando $C = 0$, la suma $\sum_{i=1}^{n} \xi_i$ puede tomar cualquier valor y se permite mucha laxitud en los márgenes, las restricciones son suaves, y cuando $C \to \infty$, la suma $\sum_{i=1}^{n} \xi_i$ se tiene que hacer lo más pequeña posible, las restricciones son duras.

Para generalizar a más de dos categorías, al igual que con la Regresión Logística, se pueden considerar m parámetros w (para tener varias rectas o hiperplanos), y entonces la función de costo se vuelve:

$$L(w) = \sum_{i=1}^{m} |w_i|^2 + C \sum_{i=1}^{n} \xi_i$$

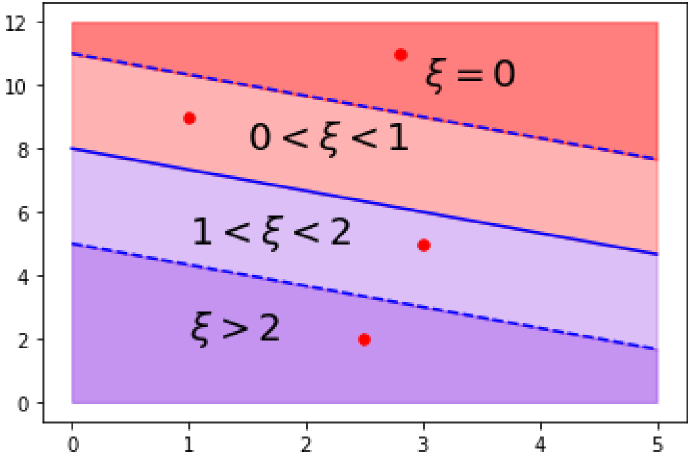


Figura 8.16: Permisividad para que los puntos "rojos" ingresen a otro tipo de clasificación, medido por las violaciones a los márgenes ξ_i

Para que un punto $x^{(i)}$ se considere en la categoría k entonces debe de pasar que, para cualquier j, con $j \neq k$:

$$w_k \cdot x^{(i)} + b_k - w_j \cdot x^{(i)} - b_j \geq 1 - \xi_i$$

esto es, que $w \cdot x + b$ se maximice para la categoría k.

Por último, no sólo se pueden tener fronteras lineales en SVM. Se puede expandir la base para, además de las variables lineales x, estén los cuadrados de las variables y los productos entre ellas. Por ejemplo, en 2 dimensiones, la base sería generada por $(x_1, x_2, x_1^2, x_2^2, x_1 x_2)$, y aunque se tengan hiperplanos en esas dimensiones con esas variables, en el espacio original (x_1, x_2) se podrían tener fronteras curvas. La elección de que la frontera sea lineal o curva se ajusta en `sklearn` eligiendo el kernel adecuado en las opciones.

Hagamos el ejercicio de las flores iris, primero pidiendo una frontera muy constreñida, con una C muy alta. Llamamos a la función `SVC` (`Support Vector Classifier`):

```
In [16]:
from sklearn.svm import SVC
modelo, error= predice(SVC(C=1e6),2,3,X_train,y_train,X_test,y_test)
print(error)
zona(modelo, 2, 3, X_train, y_train, X_test, y_test,\
"clasificación con SVM", "rainbow")
Out[16]:
0.05263157894736842
```

Es claro de la Figura 8.17 que hay un sobreajuste, la curva no se ve muy natural. Los puntos que están encerrados son los que sirvieron como vectores soporte. Las fronteras son curvas porque el kernel por default es `''rbf''` (`radial basis function`), aunque se podría elegir la opción `kernel=''linear''` para tener líneas rectas (para ver las opciones de `SVC`, hay que mandar a imprimir `help(SVC)`). Vamos a intentar ahora dejando el valor por default $C = 1$ para no constreñir demasiado los puntos:

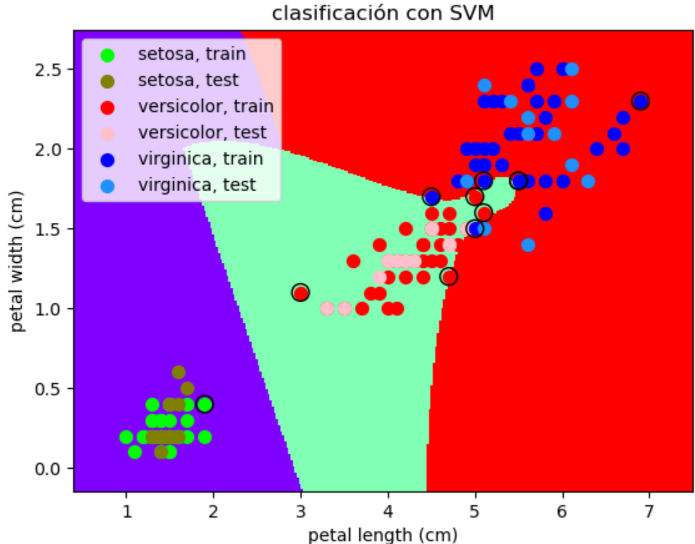


Figura 8.17: Clasificación con SVM, con parámetro C muy grande. El modelo queda sobreajustado

```
In [17]:
modelo, error= predice(SVC(),2,3,X_train,y_train,X_test,y_test)
print(error)
zona(modelo, 2, 3, X_train, y_train, X_test, y_test,\
"clasificación con SVM","rainbow")
Out[17]:
0.05263157894736842
```

Reduciendo el valor de C, aumenta el número de vectores soporte. Aunque el error es el mismo, las fronteras de la Figura 8.18 ahora lucen más naturales.

Ejercicios:

1. Supón que en un problema de clasificación de datos en 2 dimensiones, x_1 y x_2, las rectas que definen la frontera y los márgenes según Support Vector Machines, son:

$$x_2 = x_1/2 + 6, \quad x_2 = x_1/2 + 9, \quad x_2 = x_1/2 + 3$$

Reescalando para tener las ecuaciones $w \cdot x + b = 0$, $w \cdot x + b = \pm 1$, ¿cuáles son los valores de w_1, w_2 y b?
2. Continuando con los datos de los vinos, y con los índices de las columnas 0 y 6, representando a las variables ‘‘`alcohol`’’ y ‘‘`flavonoids`’’, corre las funciones respectivas para el método de Support Vector Machines, con los parámetros por default, y cambiando la C de grande (constricción fuerte) a pequeña (la que está por default, constricción suave). ¿Hay diferencia?, cuál valor de C es preferible?

8.1.7. Gaussian Naive Bayes

El método de Clasificación de Naive Bayes se basa en el Teorema de Bayes. Si tenemos un conjunto de datos de entrenamiento con m clases, para cada clase podemos ajustar

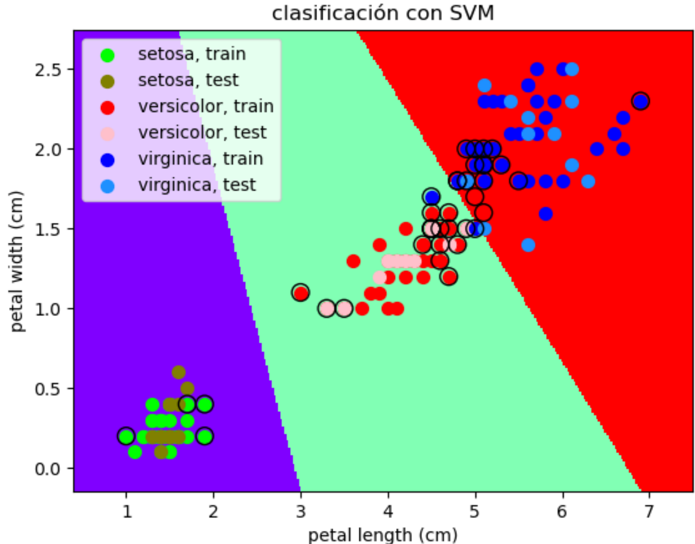


Figura 8.18: Clasificación con SVM, con parámetro C pequeño. El modelo no es tan restrictivo.

una distribución de probabilidad gaussiana. No sólo se consideran las distribuciones gaussianas individuales, también, en este método, importa la frecuencia de cada clase. Hablamos entonces de dos probabilidades, la probabilidad de una clase k, $P(k)$, y la probabilidad de un punto o suceso x dada esa clase k, $P(x|k)$, con lo cual se obtiene la probabilidad conjunta $P(x|k)P(k)$. Ahora, lo que queremos, dado un nuevo dato, es encontrar la probabilidad de que pertenezca a una categoría dada, esto es, la probabilidad de obtener la categoría k dado el punto x, $P(k|x)$. Es aquí donde entra el teorema de Bayes, pues sólo tenemos que hacer:

$$P(k|x) = \frac{P(x|k)P(k)}{P(x)}$$

Para categorizar un punto x, sólo se toma el valor más alto de $P(x|k)P(k)$ para todas las clases k, pues el denominador es el mismo para ese punto x.

Antes de proceder a un ejemplo, recordemos cuál es la función de densidad de la distribución gaussiana en una dimensión:

$$\rho(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Aunque en varias dimensiones se requiere la matriz de covarianza, que considera las correlaciones entre las variables, Gaussian Naive Bayes supone que no hay correlaciones y las gaussianas correspondientes a cada categoría son paralelas a los ejes de las variables. En dos dimensiones, la función de densidad de la distribución gaussiana, sin correlación entre las variables, quedaría:

$$\rho(x_1, x_2) = \frac{1}{2\pi\sigma_{x_1}\sigma_{x_2}} \exp\left(-\frac{(x_1-\mu_{x_1})^2}{2\sigma_{x_1}^2} - \frac{(x_2-\mu_{x_2})^2}{2\sigma_{x_2}^2}\right)$$

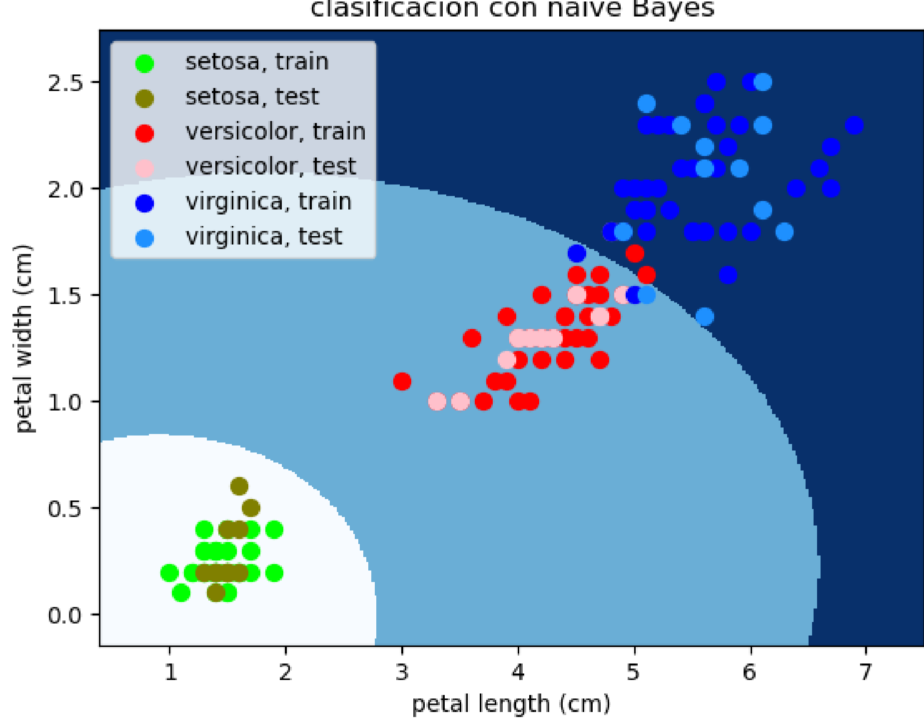


Figura 8.19: Clasificación de las flores iris con Gaussian Naive Bayes

Hagamos el ejemplo del conjunto de datos de las iris. Se llama a la función `GaussianNB` (Figura 8.19):

```
In [18]:
from sklearn.naive_bayes import GaussianNB
modelo, error= predice(GaussianNB(),2,3,X_train,y_train,X_test,y_test)
print(error)
zona(modelo, 2, 3, X_train, y_train, X_test, y_test,\
"clasificación con naive Bayes","Blues")
Out[18]:
0.05263157894736842
```

Ejercicios:

1. Para unos ciertos datos en una dimensión hay mediciones correspondientes a tres categorías, “rojo”, “verde” y “azul”. Si hay 60 datos “rojos”, 40 “verdes” y 50 “azules”, ¿cuál es la frecuencia de cada categoría?
2. El promedio de los datos “rojos” del problema 1 es de $\mu = 5$ y su desviación estándar es $\sigma = 1.6$. Los datos para la categoría “verde” son $\mu = 3$ y $\sigma = 1.4$, mientras que para la categoría “azul” son $\mu = 6$ y $\sigma = 2.5$, ¿Cuál de los valores de las funciones de densidad, si ajustamos distribuciones normales por cada categoría, es el más alto para el punto $x = 4$?
3. Si consideramos la probabilidad conjunta de los problemas 1 y 2 de arriba, esto es, $P(x = 4|\text{color})P(\text{color})$, ¿cuál es la más alta y, por tanto, cuál categoría sería la más probable para el punto $x = 4$?

4. Continuando con los datos de los vinos de los ejercicios anteriores, corre el programa seleccionando el método de Naive Bayes y las columnas 0 y 6 correspondientes a las variables ''`alcohol`'' y ''`flavonoids`''.

8.1.8. Reescalamiento de datos

Hasta ahora, el haber trabajado con los dato de las flores iris, donde las mediciones son del mismo orden, ha hecho que, para algunos de los métodos vistos, no nos hayamos topado con el problema de la diferencia de orden en las dimensiones de las diferentes mediciones, como por ejemplo, la diferencia en el ancho de una puerta respecto a su largo y alto. Cuando esto sucede, hay que reescalar los datos, para que las variables estén en igualdad de circunstancias para competir, sobre todo cuando hay distancias euclideanas de por medio. Hay varios tipos de reescalamientos, el que mostraremos aquí es el escalamiento `StandardScaler`, que deja los datos, en cada dimensión, con promedio cero y varianza uno. Antes de aplicar algún método de aprendizaje automático, preprocesamos los datos con ese u otro reescalamiento.

Empezando con las variables independientes `X` y la variable dependiente `y`, separamos como siempre los datos de prueba y de entrenamiento. Posteriormente, encontramos los parámetros para el reescalamiento (en este caso el promedio y la varianza) de las variables independientes considerando sólo los datos de entrenamiento y, dados esos parámetros, ajustamos tanto a los datos de entrenamiento como los de prueba. El procedimiento sería:

```
X_train, X_test, y_train, y_test=

train_test_split(X,y,random_state= algún número)

from sklearn.preprocessing import StandardScaler

escalador= StandardScaler()

escalador.fit(X_train)

X_train_esc= escalador.transform(X_train)

X_test_esc= escalador.transform(X_test)
```

y en lo subsecuente trabajamos con los datos escalados. Si quieres ver qué otros tipos de escalamiento hay, importa `sklearn.preprocessing` y usa `dir` para explorar su contenido.

Ejercicios

Considera los datos de los vinos de los ejercicios anteriores. Como es de imaginarse, el método de vecinos más cercanos, que involucra la distancia euclideana entre puntos, y, por tanto, la combinación de dos o más dimensiones, es sensible a la escala de los datos. Selecciona dos variables donde no haya mucha intersección entre los datos y en que las escalas sean muy diferentes. Usa el método de k vecinos más cercanos con los datos como están, graaficando la zona, y compara con el mismo método pero donde pre-procesas los datos con el escalador `StandardScaler`.

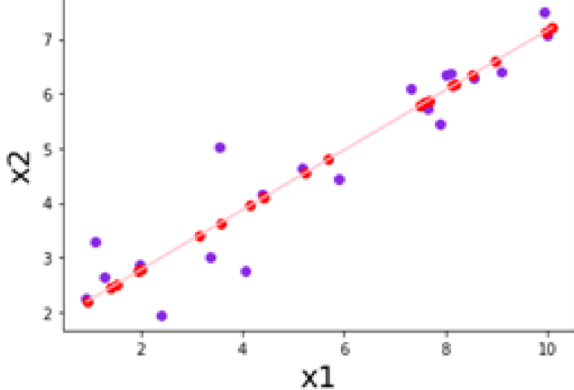


Figura 8.20: Proyección de los puntos en 2D sobre una recta en la dirección de máxima varianza

8.2. Aprendizaje no supervisado: PCA

El método del análisis de componentes principales (PCA, Principal Component Analysis) consiste en, dado un número n de puntos en d dimensiones, reducir el número de dimensiones de forma de que las que quedan tengan la máxima varianza. Por ejemplo, dados los puntos violetas mostrados en la Figura 8.20, se ve que hay una correlación entre las variables x_1 y x_2. Si reducimos el número de dimensiones de 2 a 1, el eje de la nueva variable que conviene seleccionar es a lo largo de la línea mostrada en la Figura 8.20, que es donde los datos presentan una mayor varianza o esparcimiento. Los puntos rosas son la proyección de los puntos violeta sobre el nuevo eje.

En dos dimensiones no se aprecia la utilidad de reducir el número de dimensiones, pero si estamos hablando de varias, ésta es una de las formas de reducir redundancia y quedarnos con variables que contengan la mayor información posible.

El procedimiento inicia como sigue. Dados n puntos x_i en d dimensiones, se encuentra el vector de los promedios en cada dimensión μ y la matriz de covariancia Σ de tamaño $d \times d$. Se trasladan los ejes a μ y se encuentran los eigenvectores de Σ, que dan las direcciones de los nuevos ejes.

Recordemos primero que la matriz de covarianza Σ está definida por

$$\Sigma = \frac{1}{n-1} B B^T$$

donde

$$B = \begin{pmatrix} x_1 - \mu & x_2 - \mu & \dots & x_n - \mu \end{pmatrix}$$

B es una matriz de $d \times n$, por las d dimensiones de cada punto y al haber n puntos. Por tanto, Σ es una matriz de $d \times d$ simétrica.

Hacemos un paréntesis para hablar de unas de las propiedades de las matrices simétricas.

Primero, los eigenvectores de una matriz simétrica (que se define por ser igual a su transpuesta, $A = A^T$) son ortogonales.

Demostración: Sea A la matriz simétrica, u_i sus eigenvectores ya normalizados, y λ_i sus eigenvalores, esto es, $Au_i = \lambda_i u_i$. Entonces, haciendo el producto punto de 2 de los eigenvectores u_i y u_j, y multiplicando por el eigenvalor λ_i, se tiene:

$$\lambda_i u_i^T u_j = (\lambda_i u_i)^T u_j = (Au_i)^T u_j = u_i^T A^T u_j = u_i^T A u_j = u_i^T \lambda_j u_j = \lambda_j u_i^T u_j$$
$$\Rightarrow \quad (\lambda_i - \lambda_j) u_i^T u_j = 0$$

Por lo que si $\lambda_i \neq \lambda_j$, entonces u_i y u_j son ortogonales.

Segundo, una matriz del tipo $A = B^T B$ (o $A = BB^T$) es simétrica, y sus eigenvalores son reales y mayores a cero.

Demostración: $A^T = (B^T B)^T = B^T B = A$, y

$$u_i^T A u_i = u_i^T B^T B u_i = (Bu_i)^T B u_i = |Bu_i|^2,$$
$$u_i^T A u_i = u_i^T \lambda_i u_i = \lambda_i u_i^T u_i = \lambda_i$$

Por lo que igualando las dos últimas expresiones se tiene que $\lambda_i \geq 0$.

Continuamos con el procedimiento de reducción de dimensiones. Una vez encontrados los eigenvectores de Σ, se acomodan en columnas en una matriz U de mayor a menor eigenvalor. ¿Por qué acomodamos de mayor a menor?, porque, en general, si ensandwichamos la matriz de covarianza entre un vector aleatorio unitario w, el resultado es la covarianza en la dirección de ese vector,

$$\text{covarianza en dirección } w = w^T \Sigma w$$

En particular, si ensandwichamos la matriz de covarianza entre uno de sus eigenvectores, el resultado es el eigenvalor asociado a ese eigenvector:

$$u_k^T \Sigma u_k = u_k^T \lambda_k u_k = \lambda_k$$

Así, como queremos seleccionar las direcciones perpendiculares de máxima varianza, ordenamos de mayor a menor.

En principio, U debería de ser una matriz de dimensión $d \times d$, pero el propósito de PCA es reducir el número de éstas, por lo que se seleccionan sólo k eigenvectores, los principales, quedando una matriz de $d \times k$:

$$U = (u_1 \quad u_2 \quad \cdots \quad u_k)$$

Lo que queremos es proyectar cada uno de los vectores x en las nuevas direcciones u_i, y esto se logra con el producto punto de los vectores, $u_i \cdot x$, o en notación matricial, $u_i^T x$. La matriz U ya conjunta los k eigenvectores principales, por lo que las proyecciones en las nuevas direcciones quedan como:

$$x_{PCA} = U^T x = \begin{pmatrix} u_1^T \\ \vdots \\ u_k^T \end{pmatrix} x = \begin{pmatrix} u_i^T x \\ \vdots \\ u_k^T x \end{pmatrix}$$

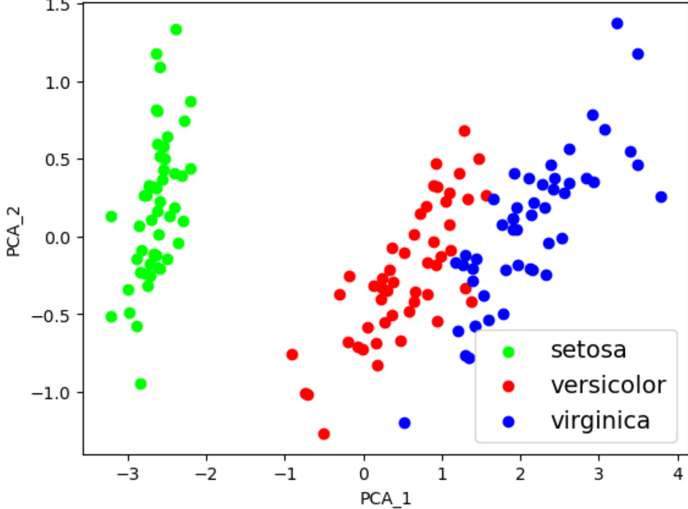


Figura 8.21: Gráfica de los datos iris en 2D con las 2 componentes principales

U^T es de $k \times d$ dimensiones, x tiene d dimesiones, y x_{PCA} tiene las k dimensiones reducidas.

Una vez realizada la reducción de dimensiones, se puede aplicar algún método por ejemplo de clasificación para seguir procesando los datos.

En `sklearn`, el procedimiento para aplicar PCA es el siguiente. Se llama a la función `PCA`, se especifica el número de dimensiones deseadas, y, al igual que con los métodos supervisados, se ajusta el modelo con la función `fit`, con la diferencia de que sólo se van a ajustar las variables medidas X, sin considerar el blanco o variable dependiente y, y se van a considerar todos los datos, no dividimos en datos de entrenamiento y de prueba. El siguiente paso, a diferencia de los métodos supervisados, es usar la función `transform` para rotar ejes. De hecho, se puede usar la función `fit_transform` para hacer el procedimiento en un solo paso.

Hagamos el ejemplo de las flores iris reduciendo el número de dimensiones a 2 y graficando (Figura 8.21):

```
In [19]:
from sklearn.decomposition import PCA
pca= PCA(n_components=2).fit(X)
X_pca= pca.transform(X)     # proyectar en 2 dims
plt.figure()
for k in range(3):
    plt.scatter(X_pca[y==k,0],X_pca[y==k,1],\
                color=colores[k],label=nombres[k])
plt.xlabel("PCA_1")
plt.ylabel("PCA_2")
plt.legend(fontsize=14)
plt.show()
```

Podemos saber en qué medida están contribuyendo las variables originales (4 variables) a las componentes principales (2 variables) con el método `components_`. Graficamos los resultados de las contribuciones en la Figura 8.22 (primera componente principal)

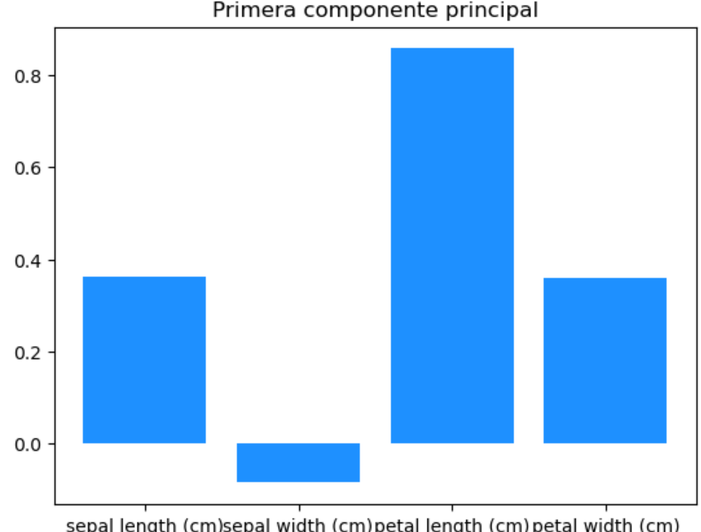


Figura 8.22: A la primera componente principal contribuye principalmente el largo del pétalo.

y Figura 8.23 (segunda componente principal):

```
In [20]:
comp=pca.components_
print(comp)
for c, text in zip(comp,["Primera","Segunda"]):
    plt.figure()
    plt.bar(var,c,color="dodgerblue")
    plt.title(text + " componente principal")
    plt.show()
Out [20]:
[[ 0.36138659 -0.08452251  0.85667061  0.3582892 ]
 [ 0.65658877  0.73016143 -0.17337266 -0.07548102]]
```

La variable que más contribuye a la primera componente principal es el largo del pétalo, mientras que a la segunda componente contribuyen casi en la misma medida el largo y ancho del sépalo.

Ejercicios:

1. Dados los siguientes 5 puntos en 3 dimensiones:

$$x_1 = \begin{pmatrix} 3 \\ 5 \\ -2 \end{pmatrix}, \quad x_2 = \begin{pmatrix} 6 \\ 4 \\ 5 \end{pmatrix}, \quad x_3 = \begin{pmatrix} -4 \\ 2 \\ 1 \end{pmatrix}, \quad x_4 = \begin{pmatrix} 0 \\ -2 \\ 0 \end{pmatrix}, \quad x_5 = \begin{pmatrix} 5 \\ 6 \\ 1 \end{pmatrix}$$

Encuentra su vector promedio μ y su matriz de covarianza Σ. ¿Cuáles son las dimensiones de Σ?

2. Dada la matriz de covarianza:

$$\Sigma = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}$$

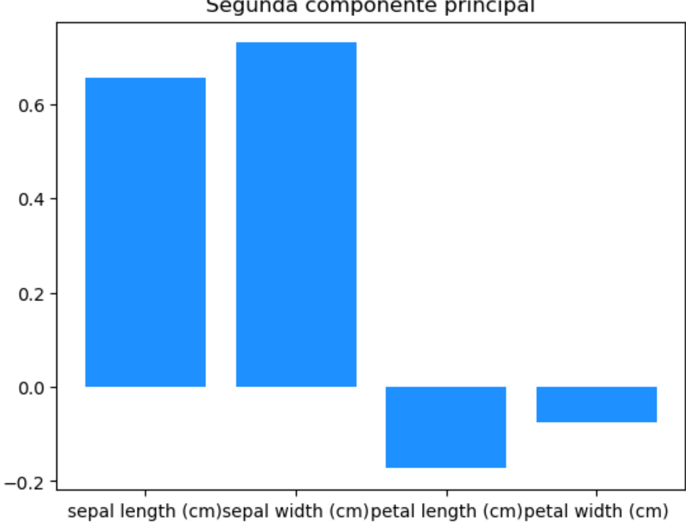


Figura 8.23: A la segunda componente principal contribuyen principalmente las medidas del sépalo.

Encuentra sus eigenvalores y eigenvectores. Construye la matriz U sólo con el eigenvector asociado al mayor eigenvalor o varianza. ¿Cuáles son las dimensiones de U?
3. Dado el punto $x = (4, 3)$ en el espacio de 2 dimensiones del problema 2, ¿cuál es el valor de `x_PCA` en el espacio con dimensión reducida?
4. Continuando con los datos de los vinos, entrena todo el conjunto de datos con el método de Análisis de Componentes Principales, con reducción a 2 dimensiones, y prueba sobre los mismos datos. Para ello escala previamente los datos con `StandardScaler`. Grafica los resultados en las 2 dimensiones reducidas.
5. Haz un análisis del problema 4 para ver qué variables contribuyen más a las 2 componentes principales. Como son muchas las variables, selecciona una gráfica ad-hoc como una gráfica de barras horizontal (`barh`).

8.3. Aprendizaje no supervisado: Clustering

El clustering se refiere a, dado un conjunto de datos, dividirlo en grupos sin información previa de su clasificación, los datos solos se ordenan en grupos.

8.3.1. k-Means Clustering

El método de `k-Means` Clustering es muy sencillo y se utiliza ampliamente. Consiste en, dados los datos, elegir primero el número k de clústeres en que se van a dividir. Enseguida, se eligen aleatoriamente k puntos que servirán como los primeros centros. A partir de ahí, se calcula la distancia de los puntos a todos los centros y de esa manera se determina a qué clúster corresponden. Una vez formados los clústeres, se calcula el centro de masa de cada uno que funcionarán como los nuevos centros. El procedimiento se repite, esto es, se calculan las distancias de los puntos a los centros para determinar a qué clúster corresponden, y después se recalculan los centros de masa. El procedimiento termina cuando los centros ya no cambian de una iteración a otra.

El cálculo del “centro de masa” es más sencillo que en física, pues todas las masas son iguales, y se restringe a tomar el promedio de las distancias en cada una de las d dimensiones. Por ejemplo, en 2 dimensiones, usando los nombres de las coordenadas usuales (x, y), el centro de masa está dado por:

$$x_{CM} = \left(\frac{1}{n} \sum_{i=1}^{n} x_i, \quad \frac{1}{n} \sum_{i=1}^{n} y_i \right)$$

Como los primeros centros se eligen aleatoriamente, en `sklearn` la función que se utiliza corre varias veces el programa para determinar la mejor elección, minimizando la suma de las distancias de los puntos a los centros.

Aunque ya conocemos la clasificación de las flores iris, vamos a seguir utilizando ese conjunto de datos para ver qué clústeres se forman.

Como es un método no supervisado, como lo vimos con el análisis de componentes principales, se usan las funciones sobre los datos originales, sin necesidad de dividir en datos de entrenamiento y de prueba, y sin tener conocimiento del tipo de clasificación guardada en la variable dependiente o blanco y. En este caso usamos la función `fit`, y también la función `predict`.

Comparemos los grupos reales con los clústeres formados con `KMeans`, en 2 figuras por separado (Figura 8.24). En la figura con los clústeres formados, podemos graficar también los centros para que quede clara su ubicación:

```
1  In [21]:
2  from sklearn.cluster import KMeans
3  kmeans=KMeans(3)        # 3 clústers
4  kmeans.fit(X[:,[2,3]])
5  pred=kmeans.predict(X[:,[2,3]])
6  centros=kmeans.cluster_centers_
7  plt.figure()
8  plt.subplot(2,1,1)      # primera subfigura
9  for k in range(3):
10     plt.scatter(X[y==k,2],X[y==k,3],\
11                 color=colores[k], label=nombres[k])
12     plt.xlabel(var[2])
13     plt.ylabel(var[3])
14     plt.legend()
15     plt.title("Verdaderos grupos")
16 plt.subplot(2,1,2)      # segunda subfigura
17 for k in range(3):
18     plt.scatter(X[pred==k,2],X[pred==k,3], label=nombres[k])
19     plt.scatter(centros[k][0],centros[k][1],color="yellow",\
20                 edgecolor="brown", s=100)
21     plt.xlabel(var[2])
22     plt.ylabel(var[3])
23     plt.title("Clusters formados")
24 plt.tight_layout()
25 plt.show()
```

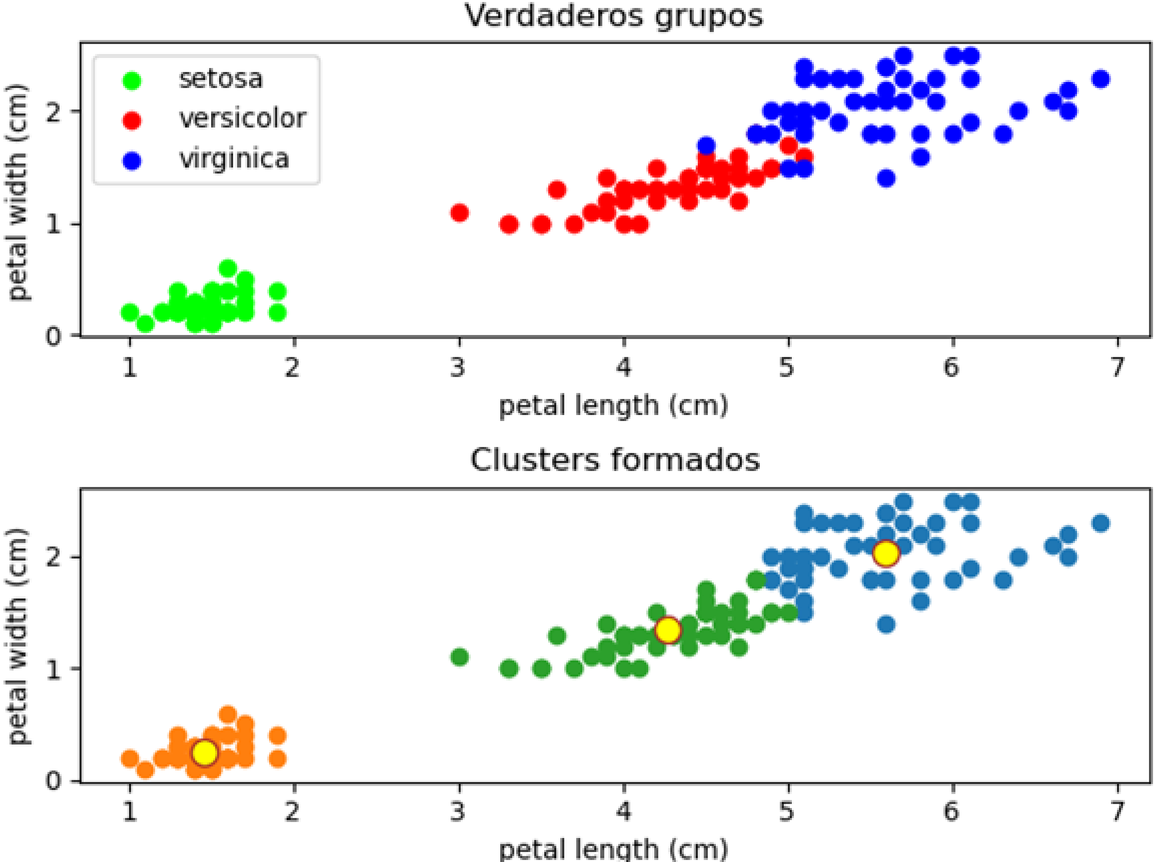


Figura 8.24: Arriba: la verdadera clasificación de los datos. Abajo: los datos agrupados con clustering de k-means. En amarillo se muestran los centros de los clústers

Ejercicios:

1. Dados los puntos:

$$(1,4),\quad (2,1),\quad (4,2),\quad (5,3)$$

Encuentra su centro de masa.

2. Volviendo a los datos de los vinos, y seleccionando las columnas 0 y 6, correspondientes a las variables "alcohol" y "flavonoids", entrena todos los datos y prueba con ellos mismos con el método de clustering de `k-means`.

8.3.2. Clustering jerárquico

Otro tipo de clustering es el clustering jerárquico, o hierarchical clustering en inglés, que consiste en ir agrupando elementos de uno en uno en clústeres, según la distancia de los puntos a éstos. Hay diferentes criterios para ir uniendo los clústeres, pero el que vamos a comentar es el del enlace promedio entre los puntos de 2 clústers, conocido como UPGMA (Unweighted Pair Group Method with Arithmetic Mean), que consiste en ir uniendo los puntos minimizando la distancia promedio entre los clústers, que está dada por, dados 2 clústeres C_1 y C_2:

$$D_{C_1,C_2} = \frac{\sum_{i\in C_1}\sum_{j\in C_2} D_{i,j}}{|C_1||C_2|}$$

Donde $|C_k|$ es el número de elementos en el clúster k. Esto es, se suman todas las distancias de los puntos de un clúster al otro, y se divide entre el número de combinaciones.

Así, al principio los 2 puntos más cercanos se unen en un clúster de 2 elementos. Después, ya sea que otro par de puntos se unan para formar otro clúster de 2 elementos, o un tercer punto se una al primer clúster. Así hasta que todos los puntos pertenecen a un solo clúster. Es en un paso intermedio cuando se decide con cuántos clústeres quedarse.

La forma de agrupar clústeres con Clustering jerárquico en `sklearn` es casi el mismo que con `KMeans`, con la función `AgglomerativeCluster`, la diferencia siendo que esta función no tiene el método `predict`, hay que aplicar el método `fit_predict` directamente para ajustar con los datos y probar el método en ellos mismos:

```
In [22]:
from sklearn.cluster import AgglomerativeClustering
def cluster(método, X):
    pred= método.fit_predict(X)
    return(pred)
```

Como antes, escogemos los datos de las flores iris y probamos la función `AgglomerativeClustering` con argumento 3 clústeres. Por default, el criterio de unión de los clústers es el criterio de `‘‘Ward’’`, pero seleccionemos en su lugar la distancia promedio entre los clústeres. Esto se selecciona con el parámetro `linkage` (Figura 8.25):

```
In [23]:
pred= cluster(AgglomerativeClustering(3,linkage="average"),\
              X[:,[2,3]] )
plt.figure()
plt.scatter(X[:,2],X[:,3],c=pred, cmap="prism")
```

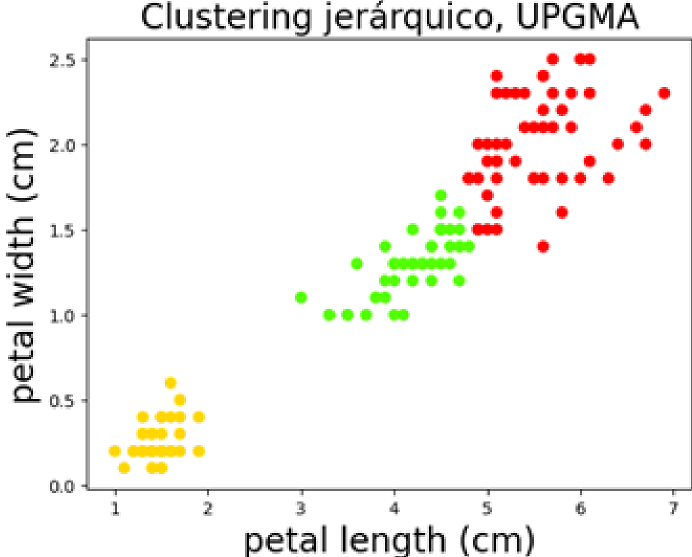


Figura 8.25: Clustering jerárquico de los datos, con criterio de unión UPGMA

```
6 plt.xlabel(var[2],fontsize=20)
7 plt.ylabel(var[3],fontsize=20)
8 plt.title("Clustering jerárquico, UPGMA",fontsize=20)
9 plt.show()
```

Seleccionamos UPGMA al ser uno de los métodos más utilizadoss para formar árboles filogenéticos, aunque no necesariamente es el mejor para formar clústers en general.

Para ver en detalle cómo se van conformando los clústers, grafiquemos un dendrograma. Primero grafiquemos cómo queda el dendrograma completo, con todos los puntos:

```
1 In [24]:
2 from scipy.cluster.hierarchy import dendrogram, average
3 plt.figure(figsize=(10,4))
4 dendrogram(average(X[:,[2,3]]))
5 plt.title("Dendrograma con clustering jerárquico, UPGMA")
6 plt.show()
```

En la Figura 8.26 se ve que, de arriba hacia abajo los datos se dividen en 2 grupos o clústeres, uno dando lugar al grupo naranja en la parte izquierda, y el otro al verde en la parte derecha. Los dos grupos se siguen subdividiendo, pero primero lo hace el grupo verde. Se puede seleccionar el número de clústeres deseados y trazar una línea horizontal para ver qué puntos quedan en los clústeres a ese nivel.

Para que no queden tan concentrados los puntos, hagamos otros dendrograma fijando el número de niveles que deseamos mostrar con el parámetro p (Figura 8.27):

```
1 In [25]:
2 plt.figure()
3 dendrogram(average(X[:,[2,3]]), p=2,truncate_mode="level")
4 plt.title("Dendrograma con clustering jerárquico, UPGMA")
5 plt.show()
```

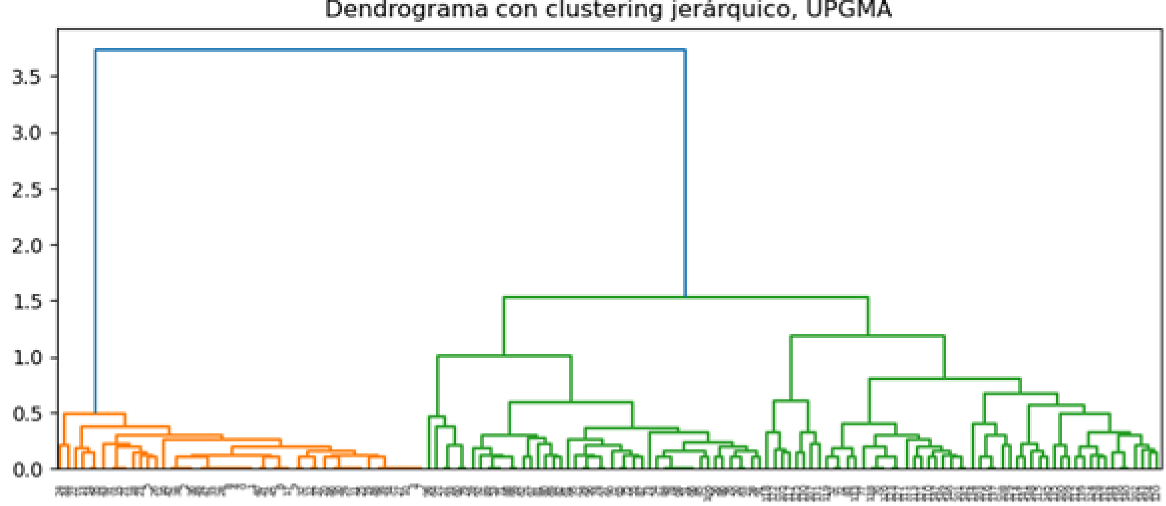


Figura 8.26: Dendrograma mostrando el clustering jerárquico de los 150 datos.

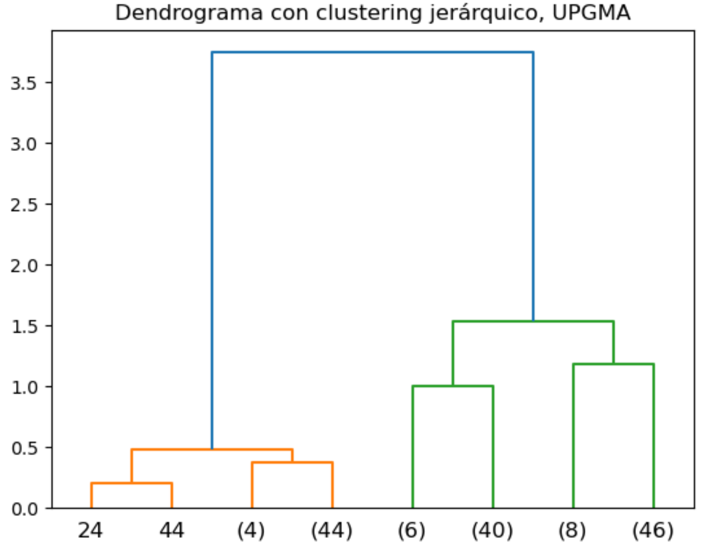


Figura 8.27: Dendrograma con número de niveles $p = 2$

En este dendrograma recortado a 2 niveles, los números sin paréntesis, esto es, el 24 y el 44, son los índices de los puntos que permanecen como hojas del árbol, o clústeres de un elemento. Los números entre paréntesis son el número de puntos que están concentrados en clústers de más de un elemento a ese nivel. En total, en este ejemplo, la suma de los números entre paréntesis da 148 puntos, más las 2 hojas, completando los 150 datos originales.

Ejercicios:

1. Dado el clúster C_1, con puntos con coordenadas:

$$(2,4), \quad (-1,3), \quad (0,2), \quad (1,3), \quad (1,2)$$

Y el clúster C_2, con puntos:

$$(5,7), \quad (8,9), \quad (6,8), \quad (7,7)$$

Encuentra la distancia promedio entre los clústeres, y su distancia mínima (la mínima distancia de los puntos de un clúster al otro).
2. Volviendo a los datos de los vinos, y seleccionando las columnas 0 y 6, correspondientes a las variables ``alcohol'' y ``flavonoids'', entrena todos los datos y prueba en ellos mismos con el método de clustering de jerarquía.
3. Dibuja el dendrograma asociado al ejercicio 2, recortado a 3 niveles.

Bibliografía

Libros de Programación

Compeau Phillip y Pevzner Pavel, (2018), *Bioinformatics Algorithms, An Active Learning Approach* (3^a). Active Learning Publishers.

Géron, Aurélien, (2017), *Hands-On, Machine Learning with Scikit-Learn & TensorFlow*, O'Reilly.

Müller, Andreas C. y Guido Sarah, (2017), *Introduction to Machine Learning with Python*, O'Reilly.

VanderPlas Jake, (2016), *Python Data Science Handbook*. O´Reilly, o su versión electrónica https://jakevdp.github.io/PythonDataScienceHandbook/

Libros de matemáticas

Barnes, B y Fulford, G. R., (2015), *Mathematical Modelling with case studies*, (3^a). CRC Press.

Kolman Bernard y Hill, David R. (2006), *Álgebra Lineal* (8^a), Pearson.

Lay, David C., Lay, Steven R, y McDonald Judi J., (2016), *Álgebra Lineal y sus aplicaciones* (5^a), Pearson.

Stewart James, (2006), *Cálculo, conceptos y contextos* (3^a). Thomson

Zill, Dennis G., (2018), *Ecuaciones Diferenciales con problemas de valores en la frontera* (9^a), Cengage.

Cursos en línea

Campbell Jennifer y Gries Paul, Learn to program: the Fundamentals [Coursera], https://www.coursera.org/learn/learn-to-program

Cetinkaya-Rundel Mine, Inferential Statistics [Coursera], https://www.coursera.org/learn/inferential-statistics-intro?

Sanjoy Dasgupta, Machine Learning Fundamentals [edX], https://www.edx.org/es

Referencias en línea

Colores web, en Wikipedia: https://es.wikipedia.org/wiki/Colores_web

Guo, Phillip, (2014, 7 de julio), Python is now the most popular introductory teaching language at top U.S. Universities, Communications of the ACM, https://cacm.acm.org/blogcacm/python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/

Juego de la vida, en Wikipedia, https://es.wikipedia.org/wiki/Juego_de_la_vida

Keep Coding, (2024, 10 de abril), ¿Cuál es el mejor lenguaje de programación? [Top 5], https://keepcoding.io/blog/mejor-lenguaje-de-programacion/

Matplotlib, Choosing Colormaps in Matplotlib: https://matplotlib.org/stable/users/explain/colors/colormaps.html

Matplotlib, matplotlib.markers: https://matplotlib.org/api/markers_api.html

Rotten Tomatoes, (s.f.), 100 best computer-animated movies ranked by tomatometer, https://editorial.rottentomatoes.com/guide/best-computer-animated-movies-of-all-time/

Programación y matemáticas con Python
se terminó de imprimir en diciembre de 2025,
en el taller de impresión de la
Universidad Autónoma de la Ciudad de México,
San Lorenzo, 290, Col. Del Valle,
Alcaldía Benito Juárez, C. P. 03100,
Ciudad de México con un tiraje de 1000 ejemplares.
Cuidado de la edición: Ángeles Godínez Guevara
Diseño editorial: Sergio Cortés Becerril

En este libro, el lector encontrará, además de la introducción a la programación con Python, módulos o bibliotecas para graficar, y de algunas áreas de las matemáticas como álgebra lineal, estadística, o métodos numéricos para ecuaciones diferenciales ordinarias, por lo que, además de ser un libro de apoyo para Introducción a la Programación, también lo es para asignaturas de matemáticas. Adicionalmente, el lector encontrará módulos de introducción a tratamientos de datos y de machine learning, tan necesarios en la actualidad que serán de utilidad no sólo para los estudiantes, sino también para la comunidad académica.

Erika Lorena Álvarez Ramírez es profesora-investigadora del Colegio de Ciencia y Tecnología. Forma parte de la Academia de Física en el Plantel Casa Libertad. Es Maestra en Ciencias (Física) por la UNAM, donde también cursó la Licenciatura en Física. Sus participaciones académicas y de los órganos de gobierno han sido como Coordinadora del Colegio de Ciencia y Tecnología (2023-2025), y como Secretaria Técnica de la Comisión de Hacienda del primero y sexto Consejos Universitarios.

UACM
Universidad Autónoma de la Ciudad de México
NADA HUMANO ME ES AJENO

Biblioteca BE del Estudiante