Python para estadística

Mario Alfonso Morales Rivera

2022-05-28

Contenido

Capítulo 1

Introducción

Estimado lector, este documento corresponde a mis notas de clase del curso **Estadística Básica con Python**, que oriento como docente en el Programa de Estadística, facultad de ciencias Básicas, en la Universidad de Córdoba. Como toda creación humana es imperfecta, y este documento no es la excepción, le pido el favor que me ayude a mejorarlo. Puede hacerlo reportando palabras mal escritas, errores de ortografía (que de seguro abundan), párrafos que usted considere que son confusos o poco claros, bloques de código que no funcionen como se espera, entre otros. Estaré altamente agradecido con su ayuda y le recompensaré invitándole a un buen café que yo mismo prepararé para usted. Este documento se compiló usando Bookdown [Xie, 2021] y la versión 3.9.7 de Python [**?**].

1.1 ¿Por qué Python?

Python se ha convertido en uno de los lenguajes de programación más populares en los últimos años, especialmente para crear sitios web utilizando sus numerosos marcos web, como Django. Python es lo que se denomina un lenguaje de secuencias de comandos, ya que se pueden usar para escribir pequeños programas o secuencias de comandos rápidos. Python se distingue por su gran y activa comunidad de computación científica. La adopción de Python para la computación científica tanto en aplicaciones industriales como en investigación académica ha aumentado significativamente desde principios de la década de 2000 [**?**]. Para el análisis de datos, la computación interactiva, exploratoria y la visualización de datos, Python inevitablemente será comparable con muchos otros lenguajes y herramientas de programación comerciales y de código abierto de uso específico en estadística, como R, MATLAB, SAS, Stata y otros. En los últimos años, el soporte de biblioteca mejorado de Python (principalmente pandas) lo ha convertido en una sólida alternativa para las tareas de manipulación de datos. Combinado con la fortaleza de Python en la programación

de propósito general, es una excelente opción como lenguaje único para crear aplicaciones centradas en datos.

1.2 Instalación de Python

Hay muchas versiones de Python disponibles en Internet para descargar e instalar de forma gratuita. En este libro usamos Anaconda, que agrupa un conjunto de herramientas para trabajar ciencia de datos y aprendizaje automático (Machine Learning) usando Python y R en un computador personal. Anaconda, en su versión individual, desarrollada para usuarios independientes, es la plataforma preferida para hacer ciencia de datos, con mas de 25 millones de usuarios en todo el mundo.

Anaconda proporciona un repositorio basado en la nube donde puede encontrar e instalar más de 7500 paquetes de ciencia de datos y aprendizaje automático. Con el comando `conda-install`, puede comenzar a usar miles de paquetes de código abierto de Python, R, Conda, y muchos otros lenguajes.

La edición individual de Anaconda es una solución flexible de código abierto que proporciona las utilidades para crear, distribuir, instalar, actualizar y administrar software de manera multiplataforma. Conda facilita la gestión de múltiples entornos de datos que se pueden mantener y ejecutar por separado sin interferencias entre sí.

Anaconda Navigator es una interfaz gráfica de usuario que viene con la edición individual de Anaconda. Desde **Anaconda Navigator** es fácil lanzar aplicaciones, manejar paquetes y entornos sin necesidad de usar comandos en la terminal del sistema. Algunas aplicaciones disponibles en esta interfaz gráfica son:

- Spyder: Un entorno científico gratuito y de código abierto escrito en Python, para Python, diseñado por y para científicos, ingenieros y analistas de datos.
- JupyterLab: Es un entorno de desarrollo interactivo basado en la web para **NoteBooks**, código y datos. Permite a los usuarios configurar y organizar trabajos en ciencia de datos, computación científica, reportes estadísticos y aprendizaje automático.
- NumPy: [**?**] Es el paquete fundamental y necesario para el análisis de datos y la computación científica de alto rendimiento en Python.
- Pandas: [**?**] Contiene estructuras de datos de alto nivel y herramientas de manipulación diseñadas para hacer que el análisis de datos sea rápido y fácil en Python. Pandas se basa en NumPy y facilita su uso en aplicaciones centradas en este paquete.

- SciPy: Es una colección de algoritmos matemáticos y funciones construidas sobre NumPy. Agrega un poder significativo a Python al proporcionar al usuario comandos y clases de alto nivel para manipular y visualizar datos.
- Matplotlib: Matplotlib es una librería para crear visualizaciones estáticas, animadas e interactivas en Python.
- Conda: Es un sistema de código abierto para la administración de paquetes y entornos que se ejecuta en Windows, macOS y Linux. Con Conda se pueden instalar, ejecutar y actualizar rápidamente los paquetes y sus dependencias. Fue creado para programas de Python, pero puede empaquetar y distribuir software para cualquier otro lenguaje.

Para instalar anaconda vaya a la página oficial, y siga las instrucciones para la descarga e instalación que allí se encuentran. En caso de necesitar asesoría adicional para la instalación le recomiendo entrar a Youtube y seguir algunos de los tutoriales que enseñan cómo hacer la instalación dependiendo de su sistema operativo. También puede buscar en páginas y blogs que ofrecen tutoriales con las instrucciones de instalación y configuración paso a paso.

1.2.1 Ejecución de código Python

Una vez se ha hecho la instalación de Anaconda, los usuarios de Windows y Mac, en el menú de aplicaciones, deben tener el ícono de *Anaconda*, en el cual se da clic para ejecutar `anaconda-navigator`

luego de cargar `anaconda-navigator` se despliega una ventana donde aparecen, entre otros, los íconos que se muestran en la figura 1.1 desde donde se pueden lanzar las aplicaciones.

Se recomienda usar `Notebooks`, cuando además de escribir y ejecutar código se requiere documentarlo, es decir acompañarlo de texto (en formato Markdown), fórmulas matemáticas y ecuaciones, tablas de datos, gráficos, imágenes y en general contenido multimedia. Los `Notebooks` se pueden crear usando las aplicaciones `JupyterLab` o `Jupyter Notebook`. Si lo que desea es escribir y ejecutar

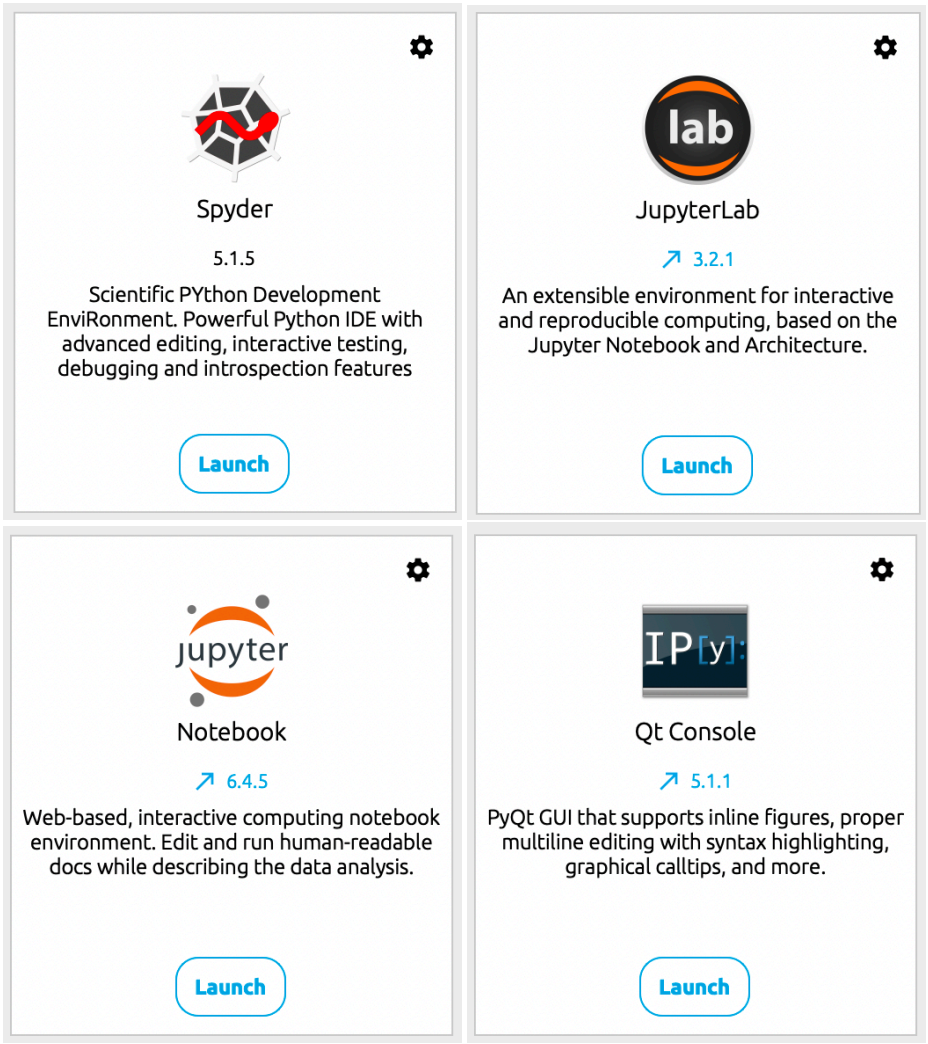

Figura 1.1: Anaconda Navigator: Algunas aplicaciones

código para resolver una tarea específica, es decir escribir `Scripts`, los cuales se guardan con extensión `.py` se recomienda usar `Spyder` y por último si el interés ejecutar código de manera interactiva, linea por lineas en la consola de Python se recomienda usar `Qt Console`.

Por último para verificar que Python está funcionando bien en su computadora, lance `QtConsole` y escriba

```
print("Hola Mundo")
```

```
## Hola Mundo
```

deberá aparece como en la figura 1.2

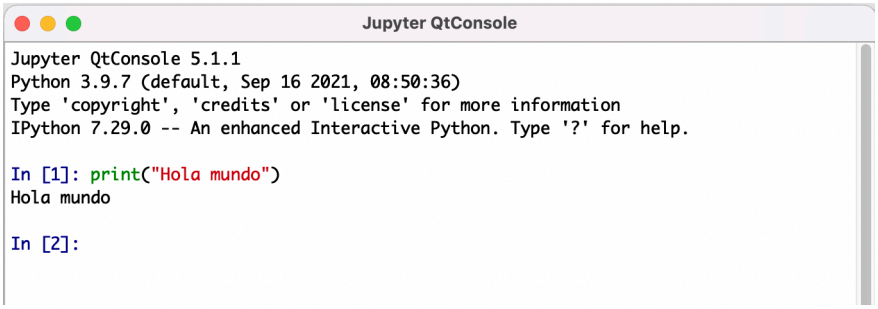

Figura 1.2: La primera línea de códido de todo programador

Capítulo 2

Contenedores de datos en Python

Python ofrece varias estructuras para el almacenamiento de datos, entre ellas tenemos

- Tuplas *(tuple)*
- Listas *(list)*
- Arreglos *(array)*
- Diccionarios *(dictionaries)*
- Series *(Series)*

- Marcos de datos *(DataFrames)*

Los arreglos que estudiaremos serán los definidos en el paquete **NumPy** [**?**], las *Series* y *data frames* no son del paquete estándar Python, en este curso usaremos las definidas en el paquete *Pandas* [**?**].

2.1 Tuplas

Las tuplas se usan para agrupar objetos que pueden ser del mismo o de diferente tipo, son objetos indexados, iterables e **inmutables**, esta última propiedad implica que, una vez creadas, a las tuplas no se les puede modificar (agregar, quitar o cambiar entradas) .

2.1.1 Creación.

En esta sección estudiaremos como crear tuplas, acceder e iterar sobre sus elementos. En el siguiente bloque se crean e imprimen las tuplas `t1` y `t2`

```
t1=('abc',4,True,5,False,2.5,4,True,"abc",4,True,"abc");t1
```

```
## ('abc', 4, True, 5, False, 2.5, 4, True, 'abc', 4, True, 'abc')
```

```
t2=(1,0.7,t1,"Hola",4)
```

Observe que la tupla `t1` contiene elementos de tipo cadena de caracteres (str), números enteros (int), valores lógicos (bool) y valores reales o de punto flotante (floats). Incluso dentro de una tupla se pueden tener otras tuplas como ocurre con la tupla `t2` que tiene *anidada* a la tupla `t1`.
Para acceder a los elementos de una tupla se usa `tupla[i]` donde `i` es un número entero entre cero (0) y la longitud de la tupla menos uno $(n-1)$. Por ejemplo, para recuperar elementos de la tupla `t1` usamos `t1[i]` con `i` entre cero y 11

```
t1[0] # primer elemento de la tupla t1
```

```
## 'abc'
```

```
t1[2] # tercer elemento de la tupla t1
```

```
## True
```

Esta característica común en todas las estructuras de datos indexadas con índice numérico en Python: el elemento k tiene índice $k-1$, olvidar este hecho podria conducir a errores fatales.

Los usuarios de lenguajes como `R` o `MATLAB` podrían verse tentados a realizar la siguiente asignación

```
t1[2]=0
```

buscando asignar el valor 0 al tercer elemento de la tupla, operacion que resultaría en el mensaje `TypeError: 'tuple' object does not support item assignment`, a esta característica es que nos referimos con que las tuplas son **inmutables**, una vez creadas no es posible cambiar, eliminar, agregar elementos. Esta característica tiene por lo menos tres ventajas: proteje contra cambios accidentales en el contenido de la tupla, se gana eficiencia en cuanto al uso de memoria de la máquina y mayor rapidez de acceso a sus elementos.

Otra forma de crear tuplas es mediante la siguiente sintaxis, con la segunda linea se imprime el tipo de objeto que es `t3`, con el fin de verificar que efectivamente se ha creado una tupla.

```
t3=10,20
type(t3)
```

```
## <class 'tuple'>
```

Esta forma de crear tuplas se conoce como *empaquetado*, también podemos recuperar los elementos de una tupla mediante el *desempaquetado*, de la siguiente manera

```
x,y=t3
print("x= ",x)
print("y= ",y)
```

el primer elemento de la tupla `t3` se asigna al objeto `x` y el segundo a `y` luego se imprimen sus valores. Se pueden crear tuplas vacías y tuplas que contiene un solo valor, llamadas tuplas unitarias como se muestra a continuación

```
tupvacia=() # crea una tupla vacía
tupunit=(4,) # crea una tupla con un único elemento
print(tupvacia,type(tupvacia),tupunit,type(tupunit))
```

```
## () <class 'tuple'> (4,) <class 'tuple'>
```

note el uso de la coma después del elemento de la tupla unitaria, eso debe ser así porque de lo contrario no se estaría creando una tupla, sino un objeto de tipo entero (`int`)

```
x=(4) # cuidado!!! x es un entero, no una tupla
print(tupunit,type(tupunit))
```

```
## (4,) <class 'tuple'>
```

2.1.2 Indexado de tuplas.

Se puede crear una tupla a partir de otra mediante órdenes de la forma `tupla[i:j]` esto se conoce como indexación, veamos algunos ejemplos:

```
a=(3,2,4,1,6,4) # se crea la tupla a
b=a[1:4] # tupla b, creada a partir de a
print(b,type(b))
```

```
## (2, 4, 1) <class 'tuple'>
```

Note en la salida anterior que la tupla `b` inicia en 2, y no en 3, que es el primer elemento de la tupla `a` ¿por qué?

```
a[:4]    # los cuatro primeros elementos de la tupla a
a[2:]    # desde el tercer elemento hasta el final
a[::-1]  # la tupla en reversa
```

2.1.3 Funciones y métodos aplicables a tuplas

La función `len()` regresa el número de elementos que contiene una tupla, el método `.index(x)` regresa el índice que le corresponde al elemento `x` en la tupla, si el elemento está más de una vez regresa el índice de la primera ocurrencia, en caso que `x` no esté ocurre un error. El método `.count()` regresa el número de veces que un elemento está en la tupla, en caso de no estar regresa creo, a continuación se muestra como usarlos sobre la tupla `t1` creada arriba

```
print(len(t1)) # Imprime  la longitud  de la tupla
```

```
## 12
```

```
t1[len(t1)-1]  # recupera el último elemento de la tupla t1
```

```
## 'abc'
```

```
t1.index(4)    # índice correspondiente al primer 4 en t1
```

```
## 1
```

```
t1.count(4)    # cuántas veces está 4 en t1
```

```
## 3
```

Habrá notado la diferencia en el llamado a `len()` frente a `.index()` y `.count()`: `len()` es una función genérica de Python , es independiente del objeto y puede aplicarse a diferentes tipos de objetos, mientras que los dos últimos son *métodos*, definidos sobre la clase *tuple*. Cuando se crea una tupla se le asocian los métodos propios de la clase, es decir, el método está asociado al objeto. Las funciones sólo se pueden llamar por su nombre, ya que se definen de forma independiente, pero los métodos no pueden ser llamados únicamente por su nombre, necesitamos invocar la clase por una referencia de esa clase en la que está definida. Esa es la razón por la que **no usamos** `index(t1,4)` o `count(t1,4)` sino `t1.index(4)` o `t1.count(4)` porque estos son métodos definidos sobre la clase `tuple`, mientras que `len()` es una función a la cual se le entrega una tupla y regresa su longitud, ella se puede usar con otros tipos de contenedores, incluso con cadenas de caracteres.

Para saber si un elemento hace parte de una tupla, usamos el operador `in` el cual regresa un valor lógico `True` o `False` dependiendo de si el elemento se encuentra o no en la tupla.

```
4 in t1     # True sí 4 está en t1, False si no
```

```
## True
```

```
"x" in t1   # True sí  "x" está en t1,  False si no
```

```
## False
```

```
"abc" in t1 # True sí "abc" está en t1,  False si no
```

```
## True
```

2.1.4 Operaciones con tuplas.

Se pueden concatenar y replicar tuplas usando los operadores `+` y `*` de la siguiente forma

```
Trats=("T1","T2","T3")
(2,3,4) + Trats # Concatena la tupla (2,3,4) con Trats
```

```
## (2, 3, 4, 'T1', 'T2', 'T3')
```

```
Trats*3         # replica la tupla Trats tres veces
```

```
## ('T1', 'T2', 'T3', 'T1', 'T2', 'T3', 'T1', 'T2', 'T3')
```

```
(2,3,4)*2       # replica la tupla (2,3,4) tres veces
```

```
## (2, 3, 4, 2, 3, 4)
```

2.1.5 Iterar sobre una tupla.

Una tupla es un objeto *iterable*, es decir, se pueden recorrer uno tras otro sus elementos usando bucles, a continuación un ejemplo usando `for`

```
for i in Trats:
    print(i)
```

```
## T1
## T2
## T3
```

2.1.6 Ejemplo de aplicación.

En el siguiente bloque de código se generan datos de una población normal con media cero y desviación estándar 10. Luego se usa la función `shapiro` para llevar a cabo la prueba de *Shapiro Wilks* sobre los datos. Lo que se quiere resaltar es que la función regresa una tupla que tiene como primer elemento el estadístico W y en la segunda posición el $p-$valor de la prueba, veamos

```
import scipy.stats as s
import numpy as np
np.random.seed(seed=123) ### para que el ejemplo sea reproducible
n = s.norm(0,10) #se instancia una v.a normal de media 0 y sd 10
datos=n.rvs(100) # muestra de tamaño 100 de la normal
res=s.shapiro(datos)
#print(res) # se imprime la respuesta
type(res) # el tipo de objeto
```

```
## <class 'scipy.stats.morestats.ShapiroResult'>
```

Note que `res`no es una tupla, pero se pueden recuperar los valores W y p mediante el desempaquetado o mediante el indexado como si lo fuera

```
W,p=res  # desempaquetado, equivalente a W=res[0] y p=res[1]
```

si estamos escribiendo un informe donde sea necesario reportar el resultado de la prueba podríamos usar

```
print("Estadístico de Shapiro - Wilks, %0.3f, p_valor = %0.4f" %res)
```

2.1.7 ¿Cómo saber si un objeto de Python es una tupla?

En algunas aplicaciones es necesario verificar si un objeto pertenece o no a una clase. Por ejemplo, para dirigir el flujo de ejecución del código de programación de una función: si el objeto entregado es una tupla, ejecute ciertas líneas de código, de lo contrario ejecute otras líneas. La función `isinstance(objeto,tipo)` regresa `True` si la clase del `objeto` entregado corresponde a `tipo`, en caso contrario regresa `False`.

```
isinstance(t1, tuple)
```

```
## True
```

```
type(t1) is tuple
```

```
## True
```

Otra forma es usando la función `type()`, como se muestra a continuación

```
type(t1) is tuple
```

```
## True
```

```
type(4) is tuple
```

```
## False
```

esta última línea regresa `False` porque el número `4` no es una tupla. En las siguientes líneas se ilustra como indagar por los tipos de datos más usados en estadística

```
type(x) is int    # True si x es un entero False en caso que no
type(x) is float  # True si x es un real False en caso que no
type(x) is str    # True si x es un string  False en caso que no
type(x) is bool   # True si x es un lógico  False en caso que no
```

A manera de ejemplo, programar una función que reciba un objeto y en caso que este sea una tupla regrese el mensaje `'Es una tupla'` en caso contrario regrese `'No es una tupla'`

```
# Se define la función
def verificartupla(x):
  if(type(x) is tuple):
    print("Es una tupla")
  else:
```

```
    print("No es una tupla")
# Fin de definición de función
```

```
# Uso de  la función
verificartupla(t1) # Retorna:  Es una tupla
verificartupla("hola") # Retorna:  No es una tupla
```

2.1.8 Ejercicios

1. Escribir una función que reciba una tupla (se debe verificar que sea una tupla) y cuente cuántos elementos contiene de tipo `float` (reales), `int` (enteros), `str` (cadenas) y `bool` (lógicos). En caso de contener objetos de otro tipo los cuente como otros.
2. Escriba lineas de código Python que permita sumar los elementos numéricos de una tupla, por ejemplo para la tupla `(3,False,'x',2.5,1.5,'n')` se debe obtener $3 + 2.5 + 1.5 = 7$.
3. Escriba lineas de código Python que permita promediar los elementos numéricos de una tupla.
4. Como se estudió en la sección 2.1.3, el método `.index(val)` aplicado a una tupla regresa el índice donde por primera vez se encuentra `val`. Programe una función que tome una tupla y un valor, en caso que el valor esté en la tupla debe regresar una tupla que contenga los indices que le correspondan al valor entregado. Por ejemplo, si la tupla es `(9,5,2,6,5,3,8,5,3)` y el valor entregado es `5`, la función debe regresar `(1,4,7)`, la tupla con los índices que corresponden a `5`; si el valor entregado es 6, la función debe regresar la tupla unitaria `(3,)`; si el valor entregado es 4, la función debe imprimir el mensaje `4 no está en la tupla`

2.2 Listas

Las listas sirven para agrupar objetos de diferente tipo, aunque típicamente se usan para alojar datos de la misma naturaleza (números, caracteres, lógicos, ...), son indexadas y, a diferencia de las tuplas, son **mutables**, es decir, se les puede agregar, quitar y cambiar elementos después de creadas. Las listas soportan todas las características de las tuplas, pero es posible adicionar, remover y modificar sus elementos.

2.2.1 Creación

Para crear una lista colocamos los elementos dentro de corchetes de la siguente forma:

```
l1=[-5,-4,-3,-2,-1,1,2,3,4,5] # lista de números enteros
l2=[2,"a",True]    #lista con elementos de distinto tipo
print(l2,type(l2))
```

```
## [2, 'a', True] <class 'list'>
```

2.2.2 Indexado de listas

Cada elemento de una lista está asociado con un *índice* que refleja su posición en el arreglo. Como en las tuplas, el primer elemento tiene índice 0, el segundo índice 1 y así sucesivamente. Asociados con la lista `l1` anterior tenemos 10 índices, iniciando en 0 y terminando en 9. Para acceder al elemento con índice 3, es decir, al **cuarto** elemento de la lista, escribimos `l1[3]` que corresponde a

```
l1[3] # Cuarto elemento de la lista l1
```

```
## -2
```

Python permite **índices negativos**, lo cual conduce a indexar por la derecha. `l[-1]` corresponde al último elemento de la lista, `l[-2]` corresponde al anterior a `l[-1]` y así sucesivamente. Esto también es aplicable a tuplas.

```
l1[-1] # Último de la lista l1
```

```
## 5
```

```
l1[-2] # Penúltimo  de la lista l1
```

```
## 4
```

2.2.3 Inserción, adición y borrado de elementos en una lista

A diferencia de las tuplas, las listas se pueden modificar insertando, adicionando o borrando elementos, veamos como se usa el método `insert()` para insertar el número -6 como primer elemento de la lista

```
l1.insert(0,-6)
print(l1)
```

```
## [-6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5]
```

En `l1.insert(0,-6)` el primer argumento, corresponde al índice donde se insertará el número dado en el segundo argumento (`-6`). La lista `l1` ahora tiene un elemento más, -6, en el índice 0 y el número -5 que estaba en este índice pasó al índice 1. Para agregar un elemento al final de la lista se usa el método `append()` como se muestra a continuación

```
l1.append(6)
print(l1)
```

```
## [-6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6]
```

note que se ha agregado `6` al final de la lista `l1`. Es posible adicionar más de un elemento al final de la lista concatenando otra lista con la operación `+`, análogo

a como se hace con tuplas.

```
l2+[4,5,6]
```

```
## [2, 'a', True, 4, 5, 6]
```

```
print(l2)
```

```
## [2, 'a', True]
```

en este caso la lista `l2` no se modifica, para conseguirlo usamos

```
l2=l2+[4,5,6]
print(l2)
```

```
## [2, 'a', True, 4, 5, 6]
```

Para borrar un elemento de una lista se usa `del lista[i]` donde `i` corresponde al índice del elemento a eliminar. Borremos el elemento `'a'` de la lista `l2`, (`i=1`)

```
del l2[1]
print(l2)
```

```
## [2, True, 4, 5, 6]
```

Mientras que `del` elimina elementos de una lista por posición, el método `remove()` los elimina por valor, seguidamente borramos el elemento `4` de la lista `l2`

```
l2.remove(4)
print(l2)
```

```
## [2, True, 5, 6]
```

si en la lista hay más de un elemento con el valor dado a `remove()` se elimina el de índice más pequeño, es decir, el primero que aparezca en la lista, veamos el siguiente ejemplo

```
l3=[2,5,4,-1,4,3,0,-2] # en esta lista 4 está dos veces
l3.remove(4)
l3
```

```
## [2, 5, -1, 4, 3, 0, -2]
```

observe que se ha eliminado el primer 4. Otro método para eliminar elementos de listas es `pop()` que recibe un índice, elimina el elemento que corresponde pero además regresa el elemento eliminado

```
l3.pop(2) # regresa -1 (índice 2)
```

```
## -1
```

```
print(l3) # l3 queda modificada, sin -1
```

```
## [2, 5, 4, 3, 0, -2]
```

si no se entrega índice el método `pop()` remueve (y regresa) el último elemento de la lista.

```
l3.pop()  # regresa -2 (el último elemento de l3)
```

```
## -2
```

```
print(l3) # l3 queda modificada, se remueve -2
```

```
## [2, 5, 4, 3, 0]
```

2.2.4 Longitud de la lista.

Similar a las tuplas, usamos la función `len()` para obtener la longitud de una lista

```
len(l2)
```

```
## 4
```

como una aplicación, podemos combinar las funciones `del` y `len()` para borrar el último elemento de una lista

```
del l2[len(l2)-1]
```

2.2.5 Indexación de listas

2.2.5.1 Índice de un elemento.

Con el método `.index()` se recupera el índice que corresponde a un valor en la lista, si el valor está en la lista más de una vez se regresa el primer índice.

```
print(l1)
```

```
## [-6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6]
```

```
l1.index(-1) # -1 tiene índice 5
```

```
## 5
```

```
l3=[2,5,4,-1,4,3,0,-2] # en esta lista 4 está dos veces
l3.index(4) # índice del primer 4
```

```
## 2
```

2.2.5.2 Sublistas.

Para extraer una sublista de una lista empleamos la sintaxis `l[i:j]`, `i` corresponde al índice del elemento inicial a partir del cual, incluido, comenzará la sublista, mientras que `j` indica el extremo superior, la sublista llegará hasta el elemento con índice `j-1`, veamos

```
l1
```

```
## [-6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6]
```

```
l1[2:7]
```

```
## [-4, -3, -2, -1, 1]
```

`l[i:]` es una sublista de `l` que inicia en el índice `i` hasta el final de la lista, análogamanete `l[:j]` es una sublista que inicia en el índice cero de la lista hasta el índice `j-1`, `l[:]` regresa la lista completa.

```
l1[2:]
```

```
## [-4, -3, -2, -1, 1, 2, 3, 4, 5, 6]
```

```
l1[:4]
```

```
## [-6, -5, -4, -3]
```

```
l1[:]
```

```
## [-6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6]
```

2.2.5.3 Verificar si un elemento está en la lista

De manera análoga a como se hizo con las tuplas se puede verificar si un elemento hace parte de una lista con el operador `in`, si el elemento dado está en la lista se obtiene `True`, en caso contrario, `False`

```
print(-2 in l3, 6 in l3)
```

```
## True False
```

2.2.6 Otras formas de crear listas

2.2.6.1 La función `list()`.

Es posible construir listas a partir de `tuplas`, `diccionarios` , `sets` (conjuntos) y otros objetos, mediante la función `list()`. Veamos un ejemplo

```
lt1=list(t1) # se crea lista a partir de la tupla t1
print(lt1)
```

```
## ['abc', 4, True, 5, False, 2.5, 4, True, 'abc', 4, True, 'abc']
```

```
print(type(lt1))
```

```
## <class 'list'>
```

2.2.6.2 La función `range()`.

En el siguiente ejemplo se usa la función `range()` que representa una secuencia *inmutable* (como las tuplas), estrictamente creciente o decreciente, de números. Esta función se utiliza en bucles `for` para realizar un número específico de iteraciones en función de un valor, que se reduce o aumenta en una constante (`step`) hasta que se alcanza un límite. Ilustremos su uso con ejemplos

```
range(9) # secuencia desde cero hasta 8
```

```
## range(0, 9)
```

La función `range()` regresa un objeto de tipo `range`, pero se puede convertir en una lista mediante la función `list()`

```
print(range(9),type(range(9)))
```

```
## range(0, 9) <class 'range'>
```

```
list(range(9))
```

```
## [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
# secuencia desde 3 hasta 22-1 incrementando en 2
list(range(3,22,2))
```

```
## [3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

En el ejemplo anterior 3 es el parámetro inicio (`start`), entero a partir del cual se devolverá la secuencia, 22 es el entero que indica el final (`stop`); la secuencia termina en el entero más grade que es menor o igual a `stop-1`, por último 2 es el paso, (`step`), que determina el incremento entre cada entero de la secuencia. Se puede iterar sobre el objeto (`range`) sin necesidad de convertirlo a una lista, uso que es bastante frecuente cuando se programa en Python

```
for i in range(3,15,3):
  print(i)
```

```
## 3
## 6
## 9
## 12
```

2.2.6.3 Comprensión de listas

La comprensión de listas es una forma elegante de definir y crear listas basadas en listas existentes. Es una forma concisa de generar una lista al aplicar una operación a los elementos de otra lista, esto se ilustra en el siguiente ejemplo:

```
[-2+0.5*x for x in range(9)]
```

```
## [-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0]
```

lo que se hace en el código anterior es evaluar la expresión $-2+0.5x$ para cada valor x en la secuencia generada por la función `range()`. Otro ejemplo, suponga que tiene una lista con valores de temperatura expresada en grados centígrado (C) y se le pide que haga la conversión a grados Fahrenheit (F) mediante la transformación $F = \frac{9}{5}C + 32$, podemos hacer la conversión de la siguiente forma

```
C=[25,32,36,19,20,24] # temperaturas en grados centígrados
F=[9.0/5*x+32 for x in C ] # temperaturas en grados Fahrenheit
F
```

```
## [77.0, 89.6, 96.8, 66.2, 68.0, 75.2]
```

En la comprensión de listas se pueden utilizar sentencias condicionales para generar nuevas listas, veamos un ejemplo: de la lista `l3` obtener una sub-lista que contenga sólo los elementos positivos.

```
print(l3)
```

```
## [2, 5, 4, -1, 4, 3, 0, -2]
```

```
[x for x in l3 if x>0]
```

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

2.2.7 Listas anidadas.

Como con las tuplas, es posible tener dentro de una lista otras listas o tuplas u objetos de distinto tipo, a continuación se crea una lista que *anida* las listas `C` y `F` creadas anteriormente

```
T=[C,F]
```

para recuperar el tercer elemento de la lista `F`, que está anidada en `T` se usa la indexación, recuperando primero a `F` y luego al elemento dentro de `F`

```
T[1][2]
```

```
## 96.8
```

2.2.8 Ordenar una lista.

El método `sort` ordena los elementos de una lista, por defecto lo hace de manera ascendente, es decir, de menor a mayor, si se quiere que el orden sea descendente usamos la opción `reverse=True`.

```
l3
```

```
## [2, 5, 4, -1, 4, 3, 0, -2]
```

```
l3.sort()
l3
```

```
## [-2, -1, 0, 2, 3, 4, 4, 5]
```

```
l3.sort(reverse=True) #orden descendente
l3
```

```
## [5, 4, 4, 3, 2, 0, -1, -2]
```

2.2.9 Ejercicios

- Use la compresión de lista para extraer de una lista de enteros la sublista de los que sean pares. Por ejemplo, de la lista que se genera por `range(20)` extraer todos los números pares y crear una lista con ellos.
- Función seno aplicada a cada elemento de una lista. En el módulo `math` esta definida la función `sin`, pero sólo acepta valores reales simples, es decir no es una función **vectorizada**. Use la compresión de lista para crear función `sinv` que tome una lista de valores reales y entregue una lista con los correspondientes senos de las componentes calculados usando la función `sin` del paquete `math`. Con la línea de código `from math import sin` se tiene disponible la función, luego se puede usar como `sin(0.2)`.
- Consulte los siguientes métodos asociados a listas y muestre ejemplos de su uso
  - `count()`
  - `split()`
  - `extend()`

2.3 Arreglos

Los arreglos en Python se pueden entender como una variante de las listas donde todos los elementos deben ser del mismo tipo. En el paquete básico (o estándar) de Python hay un tipo de objeto llamado `array`, pero este no es eficiente para cálculos estadísticos y matemáticos, por tanto no lo trataremos en este libro. Estudiaremos los arreglos definidos en el paquete *Numerical Python*, ampliamente conocido como **NumPy** [**?**]. Este módulo debe estar instalado previamente y se carga con el comando `import numpy`. Los arreglos de `numpy` tienen las siguientes características

- Se pueden llevar a cabo un amplio número de operaciones matemáticas sobre el arreglo completo, haciendo que no sea necesario el uso de bucles para recorrer los elementos del arreglo. Característica conocida como *vectorización*
- Los arreglos con un solo índice (o un índice unidimensional) se conocen como vectores.
- Arreglos con dos índices (o un índice bidimensional) se usan como una eficiente estructura de datos, matrices y tablas.

- Los arreglos pueden tener mas de dos índices, es decir, pueden ser $n-$dimensionales.

2.3.1 Creación de arreglos

Lo primero que haremos para crear un arreglo es cargar el paquete `numpy`, lo haremos con el nombre `np` como es la costumbre

```
import numpy as np
```

A continuación se crean arreglos de `numpy` a partir de tuplas y listas.

```
t=(2,3,4,1) # una tupla
at=np.array(t) # un arreglo de numpy creado a partir de t
print(at,type(at))
```

```
## [2 3 4 1] <class 'numpy.ndarray'>
```

en la salida anterior `ndarray` significa *arreglo n dimensional*. A continuación se ilustra la creación de arreglos a partir de listas

```
l2=[7.5,8.1,9] # una lista
al=np.array(l2) # arreglo de numpy a partir de l2
print(al,type(al))
```

```
## [7.5 8.1 9. ] <class 'numpy.ndarray'>
```

Con el siguiente código se crea un arreglo bidimensional, una matriz de 2 filas y 3 columnas, observe que se entrega una lista con dos listas anidadas de la misma longitud, las listas anidadas pasan a ser las filas de la matriz

```
a2=np.array([[1, -1  , 0], [0, 1, -1]])
print(a2,type(a2))
```

```
## [[ 1 -1  0]
##  [ 0  1 -1]] <class 'numpy.ndarray'>
```

Análogamente se pueden crear arreglos a partir de tuplas anidadas

```
t2=((2,3),(4,1))
ta2=np.array(t2)
print(ta2,type(ta2))
```

```
## [[2 3]
##  [4 1]] <class 'numpy.ndarray'>
```

Con los métodos, `.ndim` y `.shape` se obtiene el número de dimensiones y la longitud de cada dimensión de un arreglo

```
print("Dimensiones de a2 ", a2.ndim )
```

```
## Dimensiones de a2  2
```

```
print("Longitud de cada dimensiones de a2 ",a2.shape)
```

```
## Longitud de cada dimensiones de a2  (2, 3)
```

El método `.reshape` sirve para crear arreglos $n-$dimensionales a partir de un arreglo unidimensional veamos un ejemplo. Ingresar la matriz

$$A = \begin{bmatrix} 9 & 137 & 297 & 261 \\ 136 & 2114 & 4176 & 3583 \\ 269 & 4176 & 8257 & 7104 \\ 260 & 3583 & 7104 & 12276 \end{bmatrix} \tag{2.1}$$

como un arreglo de `numpy`.

```
# los datos en un arreglo unidimensional
Aa=np.array([9,137,297,261,
             136,2114,4176,3583,
             269,4176,8257,7104,
             260,3583,7104,12276])
#se convierte el arreglo en un arreglo bidimensional
A=Aa.reshape(4,4);A
```

```
## array([[    9,   137,   297,   261],
##        [  136,  2114,  4176,  3583],
##        [  269,  4176,  8257,  7104],
##        [  260,  3583,  7104, 12276]])
```

note que de forma predeterminada el llenado se hace por filas, es decir, los primeros cuatro elementos del arreglo conforman la primera fila, los cuatro siguientes la segunda y así sucesivamente. En caso que se requiera, podemos conseguir que el llenado de la matriz se haga por columna, en ese caso se obtendría la transpuesta de la matriz (2.1) dada en el ejemplo.

```
np.reshape(Aa,(4,4),order="F")
```

```
## array([[    9,   136,   269,   260],
##        [  137,  2114,  4176,  3583],
##        [  297,  4176,  8257,  7104],
##        [  261,  3583,  7104, 12276]])
```

no se confunda, la `F` **no** es de fila, es de `Fortran`[1].

2.3.2 Arreglos especiales

El paquete `numpy` tiene métodos para creación de arreglos especiales, por ejemplo, arreglos de ceros, de unos, en general arreglos con algún valor repetido, con cierta repetición de series de números, con números de un tipo específico de

[1] Fortran: Lenguaje de programación de alto nivel que se usa principalmente en aplicaciones científicas y matemáticas

datos (`dtype`). A continuación creamos un arreglo de n ceros y otro con tantos ceros y con el tipo de datos que tiene la lista `l2`

```
a0s=np.zeros(5) # arreglo de 5 ceros
print(a0s,type(a0s))
```

```
## [0. 0. 0. 0. 0.] <class 'numpy.ndarray'>
```

```
c=np.zeros_like(l2)
print(c,type(c))
```

```
## [0. 0. 0.] <class 'numpy.ndarray'>
```

Análogo al anterior pero con unos.

```
u=np.ones(4) # arreglo de 4 unos
u2=np.ones_like(t2) # arreglo de 1 con la misma longitud y tipo de t2
print(u2)
```

```
## [[1 1]
##  [1 1]]
```

La función `linespace()` crea un arreglo con n elementos con valores separados uniformemente en el intervalo $[p, q]$, estos arreglos son útiles al momento de graficar funciones

```
a=np.linspace(0,1,5)
print(a,type(a))
```

```
## [0.   0.25 0.5  0.75 1.  ] <class 'numpy.ndarray'>
```

Observe que el arreglo anterior, creado por `linespace()`, tiene $n = 5$ elementos, inicia en $p = 0$ y termina en $q = 1$. La función `arrange()` del paquete `numpy` se usa de manera similar a la función `range()` que estudiamos en la sección de listas, con la diferencia que la primera puede generar secuencias de números reales (no solo enteros) con incrementos que también pueden ser números reales.

```
np.arange(1.2,6,0.5)
```

```
## array([1.2, 1.7, 2.2, 2.7, 3.2, 3.7, 4.2, 4.7, 5.2, 5.7])
```

2.3.3 Matrices especiales

El paquete `numpy` cuenta con funciones para la creación de matrices especiales, como la identidad, nula, escalar entre otras. En cuanto a la matriz identidad, hay dos funciones para crearla: `eye()` e `identity()`

```
id=np.eye(2)
id
```

```
## array([[1., 0.],
##        [0., 1.]])
```

```
id2=np.identity(3)
id2
```

```
## array([[1., 0., 0.],
##        [0., 1., 0.],
##        [0., 0., 1.]])
```

La diferencia entre `eye()` e `identity()` es que con `eye()` se puede indicar, con el argumento `k`, cuál de las diagonales de la matriz se llenará de unos, de manera predeterminada `k=0`, si se indica `k=1` se llena con unos la primera diagonal por encima de la diagonal principal, si `k<0` se llenan las diagonales por debajo de la principal. Ya que las matrices escalares son matrices diagonales con todos los elementos iguales en la diagonal, es decir, son de la forma $c \times I$, donde I es la identidad, podemos crear matrices escalares multiplicando la identidad por la constante c, de la siguiente forma (con $c = 3$):

```
E2=3*id2
E2
```

```
## array([[3., 0., 0.],
##        [0., 3., 0.],
##        [0., 0., 3.]])
```

A continuación se crea un arreglo bidimensional de 4×4 con todos los elementos iguales, en este caso, a 2.

```
np.full((4,4),2)
```

```
## array([[2, 2, 2, 2],
##        [2, 2, 2, 2],
##        [2, 2, 2, 2],
##        [2, 2, 2, 2]])
```

2.3.4 Tipos de datos en los arreglos

Numpy tiene 5 tipos numéricos básicos que representan booleanos (`bool`), enteros (`int`) , enteros sin signo (`uint`), punto flotante (`float`) y complejo (`complex`). Los tipos con números en su nombre, por ejemplo `int64`, indican el tamaño en bits del tipo (es decir, cuántos bits se necesitan para representar un solo valor en la memoria)

Con el método `dtype` asociado a los arreglos de `numpy` podemos recuperar el tipo de datos que contiene el objeto y también crearlos para que contengan el tipo de dato deseado. A continuación se pide el tipo de dato que contiene el arreglo `a`

```
a.dtype
```

```
## dtype('float64')
```

para crear un arreglo que contenga datos de tipo real `float` de 64 bits usamos el siguiente código

```
ar1 = np.array([1, 2, 3], dtype=np.float64)
print(ar1,ar1.dtype)
```

```
## [1. 2. 3.] float64
```

a continuación arreglos con otros tipos de datos

```
# enteros de 64 bits
ar3 = np.array([1, 2, 3], dtype=np.int64)
# enteros de 32 bits
ar3 = np.array([1, 2, 3], dtype=np.int32)
# enteros de 16 bits
ar4 = np.array([1, 2, 3], dtype=np.int16)
# enteros de 8 bits
ar5 = np.array([1, 2, 3], dtype=np.int8)
# Booleanos (True, False)
ar6 = np.array([1, 0, 1], dtype=np.bool_)
print(ar6,ar6.dtype)
```

Podemos hacer la conversión de un tipo a otro, por ejemplo, para convertir el arreglo `ar1`, que es de tipo `float`, a entero de 32 bits, procedemos de la siguiente forma

```
ar1int=ar1.astype(np.int32)
print(ar1int,ar1int.dtype)
```

```
## [1 2 3] int32
```

2.3.5 Indexación en arreglos de NumPy

Los arreglos $n-$dimensionales de `numpy` se indexan usando la indexación estándar de Python (`x[obj]`), que hemos estudiado con tuplas (sección 2.1) y listas (sección 2.2). Hay diferentes tipos de indexación disponibles según el objeto: indexación básica, indexación avanzada y acceso de campo.

2.3.5.1 Indexación básica

2.3.5.1.1 Indexación de un solo elemento La *indexación de un solo elemento* funciona exactamente igual que para otras secuencias estándar de Python. Inicia en cero y acepta índices negativos para la indexación desde el final del arreglo.

```
a=np.arange(10) # arreglo de 0 a 9
a[2] # tercer elemento del arreglo
```

```
## 2
```

```
a[-2] # el segundo desde el final hacia adelante
```

```
## 8
```

Un ejemplo con arreglos bidimensionales: elemento en la segunda fila y tercera columna de la matriz A, es decir, el elemento a_{23}

```
A[1,2]
```

```
## 4176
```

segunda fila, penúltima columna de A

```
A[1,-2]
```

```
## 4176
```

Cuando se tiene un arreglo $n-$dimensional y se entrega un único valor, predeterminadamente el valor se toma como índice de la primera dimensión (en el caso de matrices, la fila), a continuación se selecciona la segunda fila de A.

```
A[1]
```

```
## array([ 136, 2114, 4176, 3583])
```

```
A[1][2] #el tercero de la segunda fila, equivalente a A[1,2]
```

```
## 4176
```

2.3.5.1.2 Segmentación La *segmentación* se produce cuando en `x[obj]` `obj` es de la forma `inicio:fin:paso`, un entero o una tupla de objetos de segmentación

```
a=np.arange(10) # arreglo de 0 a  9
a[1:7:2] # 1:7:2 de 1 a 7 de dos en dos
```

```
## array([1, 3, 5])
```

```
a[5:]    # del sexto elemento del arreglo a en adelante.
```

```
## array([5, 6, 7, 8, 9])
```

2.3.5.2 Herramientas de indexado dimensional

Las herramientas de indexado dimensional permiten seleccionar toda una dimensión, por ejemplo en el caso de matrices, toda una fila o conjunto de filas, o toda una columna o grupo de columnas, en general seleccionar submatrices, veamos algunos ejemplos con la matriz (2.1)

```
A[1:,]  # de la fila dos de A  en adelante
```

```
## array([[  136,  2114,  4176,  3583],
```

```
##        [  269,  4176,  8257,  7104],
##        [  260,  3583,  7104, 12276]])
```

```
A[:,1]  # segunda columna de A
```

```
## array([ 137, 2114, 4176, 3583])
```

```
A[(0,2),] # fila 1 y 3 de A
```

```
## array([[   9,  137,  297,  261],
##        [ 269, 4176, 8257, 7104]])
```

2.3.5.3 Indexación avanzada

2.3.5.3.1 Indexación entera La indexación entera de arreglos permite la selección de elementos arbitrarios en el arreglo en función de su índice $n-$dimensional. Cada arreglo de enteros representa los índices en la correspondiente dimensión. Se permiten valores negativos en los arreglos de índices y funcionan como se explicó en las secciones 2.2.2 y 2.3.5.1: en el siguiente ejemplo se seleccionan del arreglo `a` el elemento con índice 3 dos veces, luego el de índice cero y por ultimo el de índice 8:

```
a[np.array([3, 3, 0, 8])]
```

```
## array([3, 3, 0, 8])
```

```
a[np.array([3, 3, -3, 8])]
```

```
## array([3, 3, 7, 8])
```

El código siguiente seleccionan los elementos en las esquinas de la matriz A (2.1)

```
A[np.array([0, 0, 3, 3]),np.array([0, 3, 0, 3])]
```

```
## array([   9,   261,   260, 12276])
```

A continuación se selecciona la submatriz (menor) de 2×2 de la esquina superior izquierda de la matriz A.

```
A[0:2,0:2]
```

```
## array([[   9,  137],
##        [ 136, 2114]])
```

otra forma de conseguir el mismo resultado es mediante

```
A[np.ix_([0,1],[0,1])]
```

```
## array([[   9,  137],
##        [ 136, 2114]])
```

2.3.5.3.2 Indexación booleana Podemos usar como índice un arreglo de valores lógicos, este tipo de indexación es muy útil porque permite extraer valores que cumplan cierta condición, por ejemplo, extraer los pares, los que sean mayores que la media, los que estén por debajo de primer cuartil, entre otras condiciones, veamos como se hace

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

```
## array([0, 2, 4, 6, 8])
```

en el código anterior se está usando el operador `%` que regresa el residuo de la división, `a%2==0` regresa `True` para aquellos elementos del arreglo que sean pares y `False` en caso contrario. El resultado es que se extraen del arreglo `a` aquellos elementos que sean pares, es decir los que se correspondan con `True` en el vector lógico. A continuación otros ejemplos

```
a[a>5] # valores del arreglo a mayores que 5
```

```
## array([6, 7, 8, 9])
```

```
a[a<=3] # valores del arreglo a menores o iguales que 3
```

```
## array([0, 1, 2, 3])
```

2.4 Diccionarios

Un diccionario es un objeto muy flexible de Python para almacenar diferentes tipos de datos y objetos, tales como números, caracteres, listas y arreglos. Como estudiamos en la sección 2.2, una lista es una colección de objetos indexada desde cero (0) hasta el número de objetos menos uno $(n-1)$. En lugar de buscar un elemento a través de un índice de números enteros puede ser más fácil, práctico (o útil) usar texto, los diccionarios nos permiten identificar cada elemento por una clave `(key)` que por lo general es un *string*. Rigurosamente hablando, un diccionario en Python es una lista donde los índices pueden ser texto.

Suponga que necesitamos guardar las edades de tres personas: Brenda, Luis y Diego. Para eso podemos usar una lista

```
edad=[22,15,19]
```

pero de esta forma debemos recordar la secuencia de nombres, por ejemplo que el índice cero corresponde a Brenda, el índice 1 a Luis y el 2 a Diego. Esto es, la edad de Luis se obtiene como `edad[1]`. Un diccionario con los nombres de las personas como índice es más conveniente porque esto permite escribir `edad['Luis']` para recuperar la edad de Luis, sin tener que recordar el índice numérico.

2.4.1 Creación de diccionarios

2.4.1.1 Creación mediante `{clave:valor, ...}`

Los diccionarios se crean por una de las dos siguientes formas

```
edad={'Luis':15,'Brenda':22,'Diego':19}
print(edad)
```

```
## {'Luis': 15, 'Brenda': 22, 'Diego': 19}
```

2.4.1.2 Creación mediante la función `dict()`

```
edad=dict(Luis=15,Brenda=22,Diego=19)
print(edad)
```

```
## {'Luis': 15, 'Brenda': 22, 'Diego': 19}
```

2.4.1.3 Creación mediante la función `dict(zip())`

Se le entrega a la función `zip()` dos elementos iterables ya sea una cadena, una lista o una tupla, de la misma longitud y `dict()` devolverá un diccionario cuya clave i—ésima es el elemento i del primer iterable y el valor asignado a esa clave es i—ésimo del segundo iterable

```
nombres=("Luis","Brenda","Diego")
edades=(15,22,19)
edad=dict(zip(nombres,edades));edad
```

```
## {'Luis': 15, 'Brenda': 22, 'Diego': 19}
```

Los índices de texto en los diccionarios se conocen como claves *(Keys)*. En el diccionario `edad` las claves son `'Luis'`, `'Brenda'` y `'Diego'` y los valores son 15, 22 y 19.

En cualquier momento, después de su creación, se puede adicionar una pareja `texto:valor` al diccionario, por ejemplo

```
edad["Jaime"]=21
print(edad)
```

```
## {'Luis': 15, 'Brenda': 22, 'Diego': 19, 'Jaime': 21}
```

En el ejemplo anterior todos los valores del diccionario son números enteros, pero es posible crear diccionarios con diferentes tipos de valores, como en el siguiente ejemplo

```
d1= {
     "clave1":234,
     "clave2":True,
     "clave3":"Valor 1",
     "clave4":[1,2,3,4]}
```

como con los objetos estudiados en las secciones anteriores, la función `type()` nos permite confirmar si un objeto de Python es un diccionario

```
type(d1)
```

```
## <class 'dict'>
```

```
type(d1) is dict
```

```
## True
```

¿El valor en `clave2` del diccionario `d1` es un entero?

```
type(d1['clave2']) is int
```

```
## False
```

2.4.2 Acceso a claves y valores de un diccionario

Para verificar si una clave está en el diccionario `d` usamos `clave in d`, lo cual regresará un valor lógico, por ejemplo , ¿está la clave `Mary` en el diccionario `edad`?

```
'Mary' in edad
```

```
## False
```

Podemos usar la sentencia `if` para tomar decisiones, por ejemplo: en caso que se tenga el dato de `Mary` en el diccionario se imprime su edad de lo contrario se imprime un mensaje informando que ese nombre no se encuentra

```
nombre='Mary'
if nombre in edad:
  print("La edad de %s es %g" % (nombre,edad[nombre]) )
else:
  print("No hay datos para %s" % nombre )
```

```
## No hay datos para Mary
```

```
nombre='Brenda'
if nombre in edad:
  print("La edad de %s es %g años" % (nombre,edad[nombre]) )
else:
  print("No hay datos para %s" % nombre )
```

```
## La edad de Brenda es 22 años
```

Tanto las claves como los valores en un diccionario se pueden extraer y convertir a una lista, veamos

```
claves=edad.keys()
print(claves)
```

```
## dict_keys(['Luis', 'Brenda', 'Diego', 'Jaime'])
```

```
print(type(claves))
```

```
## <class 'dict_keys'>
```

note que `claves`, el objeto regresado por el método `.keys()`, no es una lista, pero podemos crearla con la función `list()`

```
list(claves)
```

```
## ['Luis', 'Brenda', 'Diego', 'Jaime']
```

a continuación se recuperan los valores del diccionario `edad` usando el método `.values()`, en este caso en la misma linea de código se hace la conversión a lista

```
valores=list(edad.values())
print(valores)
```

```
## [15, 22, 19, 21]
```

De manera análoga a tuplas, listas y arreglos es posible iterar sobre las claves de un diccionario usando bucles, por ejemplo

```
for nombre in edad:
  print("La Edad de %s es %g años" % (nombre,edad[nombre] ) )
```

```
## La Edad de Luis es 15 años
## La Edad de Brenda es 22 años
## La Edad de Diego es 19 años
## La Edad de Jaime es 21 años
```

Suponga que deseamos la lista anterior ordenada alfabéticamente por los nombres, eso podemos lograrlo con la función `sorted()` , la cual regresa una lista con las claves ordenadas

```
sorted(edad)
```

```
## ['Brenda', 'Diego', 'Jaime', 'Luis']
```

```
for nombre in sorted(edad):
  print("La Edad de %s es %g años" % (nombre,edad[nombre] ) )
```

```
## La Edad de Brenda es 22 años
## La Edad de Diego es 19 años
## La Edad de Jaime es 21 años
## La Edad de Luis es 15 años
```

la función `sorted()` no ordena ni modifica el diccionario, solo regresa una listas que contiene las claves ordenadas, en el siguiente bloque de código creamos un nuevo diccionario ordenado a partir de `edad`

```
edad_o={}
for clave in sorted(edad):
    edad_o[clave]=edad[clave]
print(edad_o)
```

```
## {'Brenda': 22, 'Diego': 19, 'Jaime': 21, 'Luis': 15}
```

Alternativamente, podemos ordenar los elementos del diccionario por las claves usando el método `.OrderedDict()` del módulo `collections`.

```
import collections
EdadOr=collections.OrderedDict(sorted(edad.items()))
print(dict(EdadOr))
```

```
## {'Brenda': 22, 'Diego': 19, 'Jaime': 21, 'Luis': 15}
```

observe que el ordenamiento se hace de manera ascendente, para hacerlo en orden descendente use `reverse=True` en la función `sorted()`

```
sorted(edad,reverse=True)
```

Se puede hacer una copia de un diccionario mediante el método `.copy()`

```
edadC=edad.copy()
```

como en el caso de las listas y arreglos, se puede borrar una entrada de un diccionario con `del`

```
del edadC["Luis"]
edadC
```

```
## {'Brenda': 22, 'Diego': 19, 'Jaime': 21}
```

2.4.3 Otros métodos aplicables a diccionarios

La mayoría de los métodos que estudiamos para listas son aplicables a diccionarios

2.4.3.1 Método pop

Remueve específicamente una clave de diccionario y devuelve valor correspondiente. Lanza una excepción `KeyError` si no encuentra la clave dada.

```
edadC=edad.copy() # copia de edad
edadC.pop("Luis")
```

```
## 15
```

```
edadC
```

```
## {'Brenda': 22, 'Diego': 19, 'Jaime': 21}
```

observe que regresa la edad de Luis y que en `edadC` ya no está la clave `Luis`. El método `.popitem()` elimina el ultimo ítem del diccionario y regresa una tupla con la clave y el valor correspondientes

```
edadC=edad.copy() # copia de edad
edadC.popitem()
```

```
## ('Jaime', 21)
```

2.4.3.2 Método `update()`

Recibe como parámetro otro diccionario. Si se tienen claves iguales, actualiza el valor de la clave repetida; si no hay claves iguales, este par clave-valor es agregado al diccionario.

```
edad2={"Brenda":25,"Mary":18}  # Brenda ya está en edad
edad.update(edad2);edad
```

```
## {'Luis': 15, 'Brenda': 25, 'Diego': 19, 'Jaime': 21, 'Mary': 18}
```

2.4.3.3 Método `.get()`

El método `get()` recibe una clave, devuelve el valor que le corresponde

```
edad.get("Brenda")
```

```
## 25
```

2.4.3.4 Método `.setdefault()`

Este método tiene dos usos: el primero es como el método `get()` y para agregar un nuevo elemento al diccionario

```
edad.setdefault("Brenda")
```

```
## 25
```

```
edad2.setdefault("Lila",16); edad2
```

```
## 16
## {'Brenda': 25, 'Mary': 18, 'Lila': 16}
```

observe como se agrega al diccionario `edad2` una nueva pareja, `Lila:16`.

Example 2.1 (Tuplas a diccionarios)**.** Considere la siguiente tupla, conformada por 6 tuplas anidadas:

```
tup = (('A', 5), ('A', 7),
       ('B', 2),('B', 9),('B', 1),
       ('C',4) )
```

note que `'A'` está dos veces, `'B'` 3 veces y `'C'` una vez en la tupla. El ejercicio consiste en trasladar esta información a un diccionario donde `'A'`, `'B'` y `'C'` conformen las claves y los valores que coresponden a cada clave se organicen en una lista.

```
dicn = {} # diccionario vacío para guardar los datos
for clave, valor in tup: # se recorre la tupla
    if clave in dicn: # en caso que la clave ya esté en dicn
        dicn[clave].append(valor) # agrega el valor a la lista
    else: # en caso que aún no esté
    #anexa a dicn la clave con el valor como
        dicn[clave] = [valor]
dicn
```

```
## {'A': [5, 7], 'B': [2, 9, 1], 'C': [4]}
```

observe que la tupla inicial tiene longitud 6, mientras que la longitud del diccionario, que contiene la misma información es 3. Esta forma de guardar los datos es mucho más eficiente en cuanto al uso de memoria y espacio en disco duro. Otra forma de realizar la misma tarea es mediante la combinación de los métodos `.setdefault()` y `.append()`

```
dicn = {} # diccionario vacío para guardar los datos
for clave, valor in tup:
    dicn.setdefault(clave, []).append(valor)
dicn
```

```
## {'A': [5, 7], 'B': [2, 9, 1], 'C': [4]}
```

Example 2.2 (Polinomios como un diccionario)**.** Las claves en los diccionarios no tienen que ser necesariamente cadenas de caracteres *(strings)*, en realidad cualquier objeto **inmutable** (como las tuplas) de Python puede usarse como clave. En este ejemplo se muestra como los diccionarios con enteros como claves se pueden usar para representar polinomios.

Los datos asociados con el polinomio

$$p(x) = -1 + x^2 + 3x^7 \tag{2.2}$$

se pueden expresar como un conjunto de pares coeficiente–exponente, en este caso el coeficiente -1 corresponde al exponente cero para la variable x ya que $-1 = -1x^0$, el coeficiente 1 pertenece al exponente 2, el coeficiente 3 corresponde al exponente 7. Un diccionario se puede usar para hacer corresponder el exponente con el coeficiente.

```
p={0:-1, 2:1, 7:3}
for m in p:
        print("El coeficiente es %g, el exponende es %d"%(p[m],m))
```

```
## El coeficiente es -1, el exponende es 0
```

```
## El coeficiente es 1, el exponende es 2
## El coeficiente es 3, el exponende es 7
```

A continuación se define una función que recibe el polinomio (como diccionario) y un valor de la variable x_0 y evalúa $p(x_0)$

```
def evalpoli(poli,x):
    return  sum([poli[m]*x**m for m in poli])
evalpoli(p,1)
```

```
## 3
```

así que el polinomio (2.2) en $x_0 = 1$, toma un valor de 3, es decir $p(1) = 3$. Observe que en la definición de la función hemos usado la compresión de listas (sección 2.2.6.3). La variable `m` varía sobre las claves del diccionario, luego se recupera el valor correspondiente a esa clave con `poli[m]` y se multiplica por `x` elevado a la `m`.

2.4.4 Ejercicios

- **Vectorizar** la función `evalpoli` definida en el ejemplo 2.2, es decir, programarla de tal forma que tome un arreglo de `numpy`, una lista o una tupla con valores numéricos y regrese un arreglo con el polinomio $p()$ evaluado en cada elemento del objeto entregado.

2.5 Series y marcos de datos (data frames)

Pandas [**?**] es el paquete de Python que proporciona estructuras de datos rápidas, flexibles y útiles para análisis estadístico, ofrece funciones que facilitan la entrada, organización y transformación de datos. Las dos estructuras de datos principales de Pandas, `Series` (unidimensional) y `DataFrame` (bidimensional), manejan la gran mayoría de los casos de uso típicos en finanzas, estadísticas, ciencias sociales y muchas áreas de la ingeniería.

2.5.1 Series de Pandas

Una *Serie* de Pandas es un `ndarray` (sección 2.3) unidimensional etiquetado. El objeto, que admite la indexación basada en enteros y etiquetas, proporciona una gran cantidad de métodos para realizar operaciones relacionadas con el índice. A continuación se muestra como crear series de Pandas. Lo primero es importar el paquete `Pandas`, lo hacemos con el nombre `pd` como es costumbre entre la comunidad de usuarios de este paquete.

```
import pandas as pd
datoS=pd.Series([22,15,19])
datoS
```

```
## 0    22
```

```
## 1    15
## 2    19
## dtype: int64
```

Cuando al crear la serie no se entregan las etiquetas o índices, de manera predeterminada la función `Series()` asigna etiquetas numéricas, como los indices de un arreglo de `numpy` o una lista. Note que la salida impresa de la serie además de los valores de las edades `22, 15 y 19`, imprimen las etiquetas (índices), asociadas a cada valor. Podemos asignar las etiquetas al momento de la creación de la serie

```
datoS=pd.Series([22,15,19],index=['a', 'b', 'c'])
datoS
```

```
## a    22
## b    15
## c    19
## dtype: int64
```

podemos convertir un diccionario a una serie de Pandas

```
edad
```

```
## {'Luis': 15, 'Brenda': 25, 'Diego': 19, 'Jaime': 21, 'Mary': 18}
```

```
edadS=pd.Series(edad)
edadS
```

```
## Luis      15
## Brenda    25
## Diego     19
## Jaime     21
## Mary      18
## dtype: int64
```

similar a los diccionarios, podemos recuperar los indices de una serie mediante el método `.index` y los valores con el método `.values`

```
edadS.index
```

```
## Index(['Luis', 'Brenda', 'Diego', 'Jaime', 'Mary'], dtype='object')
```

```
print(edadS.values,type(edadS.values))
```

```
## [15 25 19 21 18] <class 'numpy.ndarray'>
```

observe que los valores de la serie se recuperan como un arreglo de *NumPy* (véase la sección 2.3).

La recuperación de datos de una serie se hace de manera análoga a como se hace con los arreglos o los diccionarios, por indice, por etiqueta y por alguna condición, veamos algunos ejemplos.

```
edadS[1] # segundo elemento (por índice)
```

```
## 25
```

```
edadS["Brenda"] # equivalente al anterior (por etiqueta)
```

```
## 25
```

```
edadS[edadS>20] # por la condición mayor que 20
```

```
## Brenda    25
## Jaime     21
## dtype: int64
```

2.5.1.1 Datos faltantes en series de Pandas

Las series de Pandas tiene la propiedad de poder tratar con datos faltantes los cuales se representan como: `"NA" = Not available`, `"NaN" = Not a number`, `None = Null/nonetype object` a continuación se crean series que tienen datos faltantes

```
pd.Series([21, 15, None], index=['A', 'B', 'C'])
```

```
## A    21.0
## B    15.0
## C     NaN
## dtype: float64
```

```
pd.Series([21, 15, np.nan], index=['A', 'B', 'C']) # na de numpy
```

```
## A    21.0
## B    15.0
## C     NaN
## dtype: float64
```

a partir de un diccionario con valores en `None`

```
edadDic={'A':12, 'B':15, 'C':None}
pd.Series(edadDic)
```

```
## A    12.0
## B    15.0
## C     NaN
## dtype: float64
```

```
edadSN=pd.Series(edadDic,index=["A","B","D"])
edadSN
```

```
## A    12.0
## B    15.0
## D     NaN
```

```
## dtype: float64
```

La letra `D` no está como clave en el diccionario pero está como índice al crear la serie, por eso se le asigna `NaN`.

El método `.notnull()` nos permite conocer si en una serie de Pandas hay datos faltantes. En la siguiente salida, en la etiqueta `D`, donde hay un dato faltante resulta en `False`

```
edadSN.notnull()
```

```
## A     True
## B     True
## D    False
## dtype: bool
```

el método `.all()` aplicable a una serie de Pandas de valores booleanos (`dtype: bool`) regresa `True` en caso que todos los valores de la serie sean `True` y `False` en otro caso

```
edadSN.notnull().all()
```

```
## False
```

eso nos permite indagar si en una serie hay o no datos faltantes. Hay dos formas de proceder cuando se tienen datos faltantes: una es simplemente eliminarlos, lo cual se hace en series de Pandas con el método `.dropna()` y la otra se conoce como *imputación*, que consiste en estimar el (los) dato(s) faltantes mediante algún procedimiento.

```
edadSN.dropna()
```

```
## A    12.0
## B    15.0
## dtype: float64
```

se ha eliminado el dato faltante. Se cuenta con mucha literatura en cuanto a métodos de imputación, uno sencillo es el de medias, que consiste en reemplazar el dato faltante por el promedio de los datos disponibles

```
edadSN.fillna(edadSN.mean())
```

```
## A    12.0
## B    15.0
## D    13.5
## dtype: float64
```

se ha *imputado* el dato faltante (`edadSN.fillna()`) reemplazándolo por la media (`edadSN.mean()`) de los datos disponibles.

2.5.2 Marcos de datos de Pandas

El marco de datos de Pandas o *Pandas* `DataFrame` es la estructura principal de datos en el paquete Pandas. Es un arreglo en forma de tabla (bidimensional) con columnas de (potencialmente) tipos heterogéneos de datos. Los `DataFrame` son de tamaño mutable, sus filas y columnas (ejes) están etiquetados y se pueden efectuar operaciones aritméticas teniendo en cuenta dichas etiquetas. Los `DataFrames` de Pandas se pueden considerar como un contenedor similar a un diccionario para objetos que son series de Pandas.

2.5.2.1 Creación y operaciones con `data-frames`

Para crear un `DataFrames` de Pandas lo primero que hacemos es importar el paquete *Pandas*, lo hacemos con el nombre `pd` mediante la línea `import pandas as pd`, luego creamos un `DataFrame` de nombre `df` con dos columnas y cuatro filas.

```
import pandas as pd
df = pd.DataFrame({"col1":[1,3,2,4,7,9,5],
                   "col2":[9,0,2,1,7,3,4],
                   "col3":["F","M","F","F","F","M","M"]})
```

Note el uso de un diccionario con listas como valores, las claves del diccionario pasan a ser los nombres de las columnas del marco de datos y los valores las columnas. Con los métodos `.head(k)` y `.tail(k)` podemos examinar las k primeras y últimas filas, respectivamente.

```
df.head(4) # 4 primeras filas
```

```
##    col1  col2 col3
## 0     1     9    F
## 1     3     0    M
## 2     2     2    F
## 3     4     1    F
```

```
df.tail(3) # 3 últimas filas
```

```
##    col1  col2 col3
## 4     7     7    F
## 5     9     3    M
## 6     5     4    M
```

Para obtener información sobre la tabla de datos usamos el método `.info()` que proporciona detalles esenciales de los datos: la clase de de objeto, el número de filas y columnas, el tipo de datos que contiene cada columna, la cantidad de memoria ocupada

```
df.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
```

```
## RangeIndex: 7 entries, 0 to 6
## Data columns (total 3 columns):
##  #   Column  Non-Null Count  Dtype
## ---  ------  --------------  -----
##  0   col1    7 non-null      int64
##  1   col2    7 non-null      int64
##  2   col3    7 non-null      object
## dtypes: int64(2), object(1)
## memory usage: 296.0+ bytes
```

para recuperar concretamente el número de filas y columnas de la tabla de datos podemos usar el método `.shape`, que regresa una tupla con dos elementos, el primero corresponde al número de filas y el segundo al número de columnas

```
df.shape
```

```
## (7, 3)
```

El método `.index` regresa un objeto de clase `RangeIndex` que contiene la información de los índices asociados a las filas del marco de datos. Con la función `list()` podemos obtener una lista con los índices

```
df.index
```

```
## RangeIndex(start=0, stop=7, step=1)
```

```
list(df.index)
```

```
## [0, 1, 2, 3, 4, 5, 6]
```

se observa en la salida anterior que el marco de datos `df` tiene 7 filas y 3 columnas. Como se estudió en la sección 2.1 podemos guardar esa información en variables de Python para su uso en alguna rutina, mediante `nfil,ncol = df.shape`

2.5.2.2 Selección y adición de columnas.

Para seleccionar una columna, cuyo nombre es `x`, en un `Pandas DataFrame`, procedemos de la siguiente forma: `df['x']` o también mediante `df.x`. Seleccionamos `col1` del marco de datos `df`

```
df['col1']
```

```
## 0    1
## 1    3
## 2    2
## 3    4
## 4    7
## 5    9
## 6    5
## Name: col1, dtype: int64
```

observe que la clase del objeto recuperado es una serie de Pandas.

```
type(df['col1'])
```

```
## <class 'pandas.core.series.Series'>
```

Otra forma de conseguir lo mismo es mediante

```
df.col1
```

A continuación se crea la columna `rcol2` calculando la raíz cuadrada positiva de la columna `col2`, es decir, se aplica la transformación $y = \sqrt{x}$, luego se agrega al `DataFrame df`, de tal forma que este queda con tres columnas

```
df['rcol2']=np.sqrt(df["col2"])
df.head(3)
```

```
##    col1  col2 col3     rcol2
## 0     1     9    F  3.000000
## 1     3     0    M  0.000000
## 2     2     2    F  1.414214
```

note que ahora `df` tiene tres columnas. Equivalente a la línea de código anterior es

```
df['rcol1']=np.sqrt(df.col2)
```

Seguidamente se crea la columna `sum_col` como la suma de las columnas `col1` y `col2`

```
df['sum_col']=df.col1 + df.col2
# equivalente a df['sum_col']=df["col1"]+df["col2"]
df.head(3)
```

```
##    col1  col2 col3     rcol2  sum_col
## 0     1     9    F  3.000000       10
## 1     3     0    M  0.000000        3
## 2     2     2    F  1.414214        4
```

Si necesita extraer más de una columna al tiempo, por ejemplo extraer del `DataFrame df` las columnas de nombre `x1` y `x2` se usa la sintaxis `df[['x1','x2']]`, veamos un ejemplo

```
df[['sum_col','col1']].head(3)
```

```
##    sum_col  col1
## 0       10     1
## 1        3     3
## 2        4     2
```

2.5.3 Selección de filas y columnas con `iloc`

El método `iloc` selecciona tanto filas como columnas de un marco de datos de Pandas, basado en la posición del elemento (fila o columna). La sintaxis es de la forma `df.iloc[<fila>,<columna>]` donde `df` es un `DataFrame` de Pandas, `fila` un número entero o una lista de enteros indicando la posición (índice) de las filas a seleccionar. Análogamente para `columna`. Cuando se entrega un solo valor, por ejemplo `df.iloc[valor]` la selección se hace sobre las filas, veamos su funcionamiento con ejemplos concretos

```
# selecciona las primeras 3 filas
df.iloc[0:3]
# selecciona la primera fila del data
```

```
##    col1  col2 col3     rcol2  sum_col
## 0     1     9    F  3.000000       10
## 1     3     0    M  0.000000        3
## 2     2     2    F  1.414214        4
```

```
df.iloc[0] # equivalente a df.iloc[0,]
# La segunda fila del data
```

```
## col1         1
## col2         9
## col3         F
## rcol2      3.0
## sum_col     10
## Name: 0, dtype: object
```

```
df.iloc[1]
### filas con índices específicos
```

```
## col1         3
## col2         0
## col3         M
## rcol2      0.0
## sum_col      3
## Name: 1, dtype: object
```

```
df.iloc[[0,4,6]]
# selecciona la última fila del data
```

```
##    col1  col2 col3     rcol2  sum_col
## 0     1     9    F  3.000000       10
## 4     7     7    F  2.645751       14
## 6     5     4    M  2.000000        9
```

```
df.iloc[-1]
```

```
## col1         5
```

```
## col2         4
## col3         M
## rcol2      2.0
## sum_col      9
## Name: 6, dtype: object
```

note que en la última línea se ha usado un índice negativo, lo que indica selección desde la última fila. Para seleccionar columnas se procede de forma análoga

```
# Las dos primeras columnas con todas las filas
df.iloc[:,0:2]
# filas y columnas con  índices específicas
```

```
##    col1  col2
## 0     1     9
## 1     3     0
## 2     2     2
## 3     4     1
## 4     7     7
## 5     9     3
## 6     5     4
```

```
df.iloc[[0,3,6], [0,2]]
```

```
##    col1 col3
## 0     1    F
## 3     4    F
## 6     5    M
```

También se pueden seleccionar filas (o columnas) usando valores lógicos, serán seleccionadas las filas (o columnas) que les corresponda un `True`, como hemos visto con las series, esta forma de seleccionar es muy útil, pues permite filtrar los datos por el cumplimiento de ciertas condiciones que sean de interés.

```
# selecciona las columnas que les corresponda True
df.iloc[:,[False,True,False,True,False]]
```

```
##    col2     rcol2
## 0     9  3.000000
## 1     0  0.000000
## 2     2  1.414214
## 3     1  1.000000
## 4     7  2.645751
## 5     3  1.732051
## 6     4  2.000000
```

a continuación seleccionamos las filas de `df` que cumplan con la condición que `col2` sea mayor o igual que 7

```
df.iloc[list(df.col2>=7)]
```

```
##    col1  col2 col3     rcol2  sum_col
## 0     1     9    F  3.000000       10
## 4     7     7    F  2.645751       14
```

Seleccionemos del marco de datos `df` aquellas filas que cumplan con la condición que `col3` sea igual a "F"

```
df.iloc[list(df.col3=="F")]
```

```
##    col1  col2 col3     rcol2  sum_col
## 0     1     9    F  3.000000       10
## 2     2     2    F  1.414214        4
## 3     4     1    F  1.000000        5
## 4     7     7    F  2.645751       14
```

Un último ejemplo, seleccionemos del marco de datos `df` las filas que correspondan a indice par.

```
df.iloc[df.index % 2 == 0]
```

```
##    col1  col2 col3     rcol2  sum_col
## 0     1     9    F  3.000000       10
## 2     2     2    F  1.414214        4
## 4     7     7    F  2.645751       14
## 6     5     4    M  2.000000        9
```

2.5.4 Selección de filas y columnas con `loc`

Con el método `loc` se puede acceder a un grupo de filas o columnas de la tabla de datos mediante etiquetas o un arreglo cuyos elementos son valores lógicos. En el siguiente ejemplo se seleccionan aquellas filas para las cuales se verifica que los valores de `col1` son mayores que 4

```
df.loc[df.col1>4]
```

```
##    col1  col2 col3     rcol2  sum_col
## 4     7     7    F  2.645751       14
## 5     9     3    M  1.732051       12
## 6     5     4    M  2.000000        9
```

Selección de las filas de `df` que cumplan simultáneamente las condiciones `col1>4` y `col2<=3`

```
df.loc[(df.col1>=4) & (df.col2<=3)]
```

```
##    col1  col2 col3     rcol2  sum_col
## 3     4     1    F  1.000000        5
## 5     9     3    M  1.732051       12
```

2.5.5 Unir marcos de datos de Pandas

En caso que se tengan dos tablas de datos con exactamente las mismas columnas, lo cual podría darse, por ejemplo, cuando tienes información de hombres y la misma información para mujeres pero separadas, en distintas tablas, podríamos unir las dos tablas con `append`. Para ilustrar el uso de `append` crearemos dos marcos de datos, `dfM` y `dfF` que contienen datos de edad, estatura, y género, note que la información está separada por esta última variable

```
dfM=pd.DataFrame({"Edad":[24,21,18,26],
     "Estat":[1.65,1.72,1.69,1.8],
     "Genero":["M","M","M","M"]})
dfF=pd.DataFrame({"Edad":[23,22,19,25,21],
     "Estat":[1.60,1.58,1.62,1.71,1.64],
     "Genero":["F","F","F","F","F"]})
```

Usemos el método `.append()` para unir los dos marcos de datos en uno

```
df1=dfM.append(dfF)
df1
```

```
##    Edad  Estat Genero
## 0    24   1.65      M
## 1    21   1.72      M
## 2    18   1.69      M
## 3    26   1.80      M
## 0    23   1.60      F
## 1    22   1.58      F
## 2    19   1.62      F
## 3    25   1.71      F
## 4    21   1.64      F
```

note en la salida anterior que los índices de filas son los mismos de los marcos de datos originales, esto puede causar confusión al momento de seleccionar filas usando indices, para evitar ese comportamiento usamos la opción `ignore_index=True`

```
df1=dfM.append(dfF,ignore_index=True)
df1
```

```
##    Edad  Estat Genero
## 0    24   1.65      M
## 1    21   1.72      M
## 2    18   1.69      M
## 3    26   1.80      M
## 4    23   1.60      F
## 5    22   1.58      F
## 6    19   1.62      F
## 7    25   1.71      F
```

```
## 8    21   1.64      F
```

Otra forma de unir dos marcos de datos de Pandas en mediante la función `concat()`, veamos como se hace

```
tablas=[dfM,dfF]
df1=pd.concat(tablas,ignore_index=False)
df1
```

```
##    Edad  Estat Genero
## 0    24   1.65      M
## 1    21   1.72      M
## 2    18   1.69      M
## 3    26   1.80      M
## 0    23   1.60      F
## 1    22   1.58      F
## 2    19   1.62      F
## 3    25   1.71      F
## 4    21   1.64      F
```

es posible asignar claves a cada bloque representado por los niveles de género, para que, en caso que sea necesario, se facilite la posterior separación

```
dfC=pd.concat(tablas,keys=['Masculino', 'Femenino'])
dfC
```

```
##               Edad  Estat Genero
## Masculino 0    24   1.65      M
##           1    21   1.72      M
##           2    18   1.69      M
##           3    26   1.80      M
## Femenino  0    23   1.60      F
##           1    22   1.58      F
##           2    19   1.62      F
##           3    25   1.71      F
##           4    21   1.64      F
```

De ese modo, si queremos recuperar el grupo femenino podríamos hacerlo de la siguiente forma.

```
dfC.loc['Femenino']
```

```
##    Edad  Estat Genero
## 0    23   1.60      F
## 1    22   1.58      F
## 2    19   1.62      F
## 3    25   1.71      F
## 4    21   1.64      F
```

cuando los marcos de datos que queremos concatenar tienen algunas columnas

que no son comunes, `concat()` hace coincidir las columnas comunes y las no comunes las adiciona a la tabla y llena las filas que quedan sin datos con `NaN`, veamos un ejemplo

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                      index=[0, 1, 2, 3])
df1
```

```
##     A   B   C   D
## 0  A0  B0  C0  D0
## 1  A1  B1  C1  D1
## 2  A2  B2  C2  D2
## 3  A3  B3  C3  D3
```

```
df2 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                    'D': ['D2', 'D3', 'D6', 'D7'],
                    'F': ['F2', 'F3', 'F6', 'F7']},
                         index=[2, 3, 6, 7])
df2
```

```
##     B   D   F
## 2  B2  D2  F2
## 3  B3  D3  F3
## 6  B6  D6  F6
## 7  B7  D7  F7
```

note que `df1` y `df2` tienen dos columnas y dos filas en común: `B` , `D` y `2`, `3` respectivamente, ahora concatenamos

```
dfC12=pd.concat([df1,df2])
dfC12
```

```
##      A   B    C   D    F
## 0   A0  B0   C0  D0  NaN
## 1   A1  B1   C1  D1  NaN
## 2   A2  B2   C2  D2  NaN
## 3   A3  B3   C3  D3  NaN
## 2  NaN  B2  NaN  D2   F2
## 3  NaN  B3  NaN  D3   F3
## 6  NaN  B6  NaN  D6   F6
## 7  NaN  B7  NaN  D7   F7
```

note que se hacen coincidir las columnas cuidando que la información de las filas con el mismo indice se mantenga. Con la opción `join="inner"` solo se concatenan las columnas que sean comunes a los dos marcos de datos

```
dfC12=pd.concat([df1,df2],join="inner")
dfC12
```

```
##     B   D
## 0  B0  D0
## 1  B1  D1
## 2  B2  D2
## 3  B3  D3
## 2  B2  D2
## 3  B3  D3
## 6  B6  D6
## 7  B7  D7
```

La opción `axis=1` es para concatenar por filas, es decir coloca un marco de datos al lado del otro, haciendo coincidir las filas con el mismo índice

```
dfC12=pd.concat([df1,df2],axis=1)
dfC12
```

```
##      A    B    C    D    B    D    F
## 0   A0   B0   C0   D0  NaN  NaN  NaN
## 1   A1   B1   C1   D1  NaN  NaN  NaN
## 2   A2   B2   C2   D2   B2   D2   F2
## 3   A3   B3   C3   D3   B3   D3   F3
## 6  NaN  NaN  NaN  NaN   B6   D6   F6
## 7  NaN  NaN  NaN  NaN   B7   D7   F7
```

De manera análoga que con las columnas, con la opción `join="inner"` solo se conservan las filas que sean comunes a los dos marcos de datos,

```
dfC12=pd.concat([df1,df2],axis=1,join="inner")
dfC12
```

```
##     A   B   C   D   B   D   F
## 2  A2  B2  C2  D2  B2  D2  F2
## 3  A3  B3  C3  D3  B3  D3  F3
```

2.5.6 Ordenar un marco de datos

Para ordenar un `DataFrame` de Pandas se cuenta con el método `sort_values`, en los siguientes ejemplos se usa el marco de datos `df`, al cual le adicionamos otra columna de nombre `col3`

```
df["col3"]=np.array([1,2,3,4,1,2,3])
dfor=df.sort_values(by=['col1']) # ordenamos df por col1
dfor # datos ordenados
```

```
##    col1  col2  col3     rcol2  sum_col
## 0     1     9     1  3.000000       10
```

```
## 2     2     2     3  1.414214        4
## 1     3     0     2  0.000000        3
## 3     4     1     4  1.000000        5
## 6     5     4     3  2.000000        9
## 4     7     7     1  2.645751       14
## 5     9     3     2  1.732051       12
```

ahora ordenamos `df` primero por `col3` y luego por `col1`

```
df["col3"]=np.array([1,2,3,4,1,2,3])
dfor=df.sort_values(by=['col3','col1'])
dfor # datos ordenados
```

```
##    col1  col2  col3     rcol2  sum_col
## 0     1     9     1  3.000000       10
## 4     7     7     1  2.645751       14
## 1     3     0     2  0.000000        3
## 5     9     3     2  1.732051       12
## 2     2     2     3  1.414214        4
## 6     5     4     3  2.000000        9
## 3     4     1     4  1.000000        5
```

2.5.7 Convertir una columna numérica a un factor

En el análisis de datos es frecuente el caso donde se requiera convertir una variable que inicialmente se registra como numérica a una variable categórica o `factor`. La variable `col3` del marco `df` es numérica y queremos convertirla en un factor, veamos como se hace

```
# primero se hace una copia de col3 para conservarla como numérica
df["col3f"]=df["col3"]
rep=rep={1:"c1",2:"c2",3:"c3",4:"c4"} # un diccionario
df["col3f"]=df["col3"].replace(rep)
#df
df.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 7 entries, 0 to 6
## Data columns (total 6 columns):
##  #   Column   Non-Null Count  Dtype
## ---  ------   --------------  -----
##  0   col1     7 non-null      int64
##  1   col2     7 non-null      int64
##  2   col3     7 non-null      int64
##  3   rcol2    7 non-null      float64
##  4   sum_col  7 non-null      int64
##  5   col3f    7 non-null      object
## dtypes: float64(1), int64(4), object(1)
```

Tablas 2.1: Datos de edades de un grupo de personas

20	74	36	49	39	47	60	63	34	50
46	44	28	28	39	65	52	65	55	73
77	25	71	47	15	76	23	19	58	62
57	25	59	46	39	42	25	21	19	55
47	33	55	79	63	30	58	31	17	47

```
## memory usage: 464.0+ bytes
```

observe que el tipo de dato que contiene `col3f` es `object`, es decir, cadena de texto o `string`. Para hacer la conversión a una variable categórica se usa el método `astype`

```
df["col3f"]=df.col3f.astype('category')
df.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 7 entries, 0 to 6
## Data columns (total 6 columns):
##  #   Column   Non-Null Count  Dtype
## ---  ------   --------------  -----
##  0   col1     7 non-null      int64
##  1   col2     7 non-null      int64
##  2   col3     7 non-null      int64
##  3   rcol2    7 non-null      float64
##  4   sum_col  7 non-null      int64
##  5   col3f    7 non-null      category
## dtypes: category(1), float64(1), int64(4)
## memory usage: 619.0 bytes
```

Una situación que se presenta con frecuencia en el análisis de datos es categorizar una variable cuantitativa, por lo general de naturaleza continua. Por ejemplo, tenemos los siguientes datos que se muestran en la tabla 2.1 corresponden a edades de personas

Usaremos los siguientes grupos etarios según la OMS: juventud [14 - 26 años], adultez joven (26 - 40 años] adultez intermedia (40 - 50 años] adultez tardía (50 - 60 años] y adulto mayor (60 años o más)

```
edad=pd.Series([20, 74, 36, 49, 39, 47, 60, 63, 34, 50,
46, 44, 28, 28, 39, 65, 52, 65, 55, 73,
77, 25, 71, 47, 15, 76, 23, 19, 58, 62,
57, 25, 59, 46, 39, 42, 25, 21, 19, 55,
47, 33, 55, 79, 63, 30, 58, 31, 17, 47] )
edadf=pd.cut(edad, bins=[14,26,40,50,60,float('Inf')],
labels= ['Juventud','Adultez joven',
```

```
'Adultez intermedia','adultez tardía','Adulto Mayor'])
dfedad=pd.DataFrame({"Edad":edad,"Categ":edadf})
dfedad.head()
```

```
##    Edad               Categ
## 0    20            Juventud
## 1    74        Adulto Mayor
## 2    36       Adultez joven
## 3    49  Adultez intermedia
## 4    39       Adultez joven
```

ahora podemos contar cuantas personas hay en cada grupo etáreo usando el método `count()`

```
edadf.value_counts()
```

```
## Adulto Mayor          11
## Juventud              10
## Adultez joven         10
## Adultez intermedia    10
## adultez tardía         9
## dtype: int64
```

estos métodos para contar, sumar, promediar, en general, para calcular estadísticas de resumen los estudiaremos en el capítulo ...

2.5.8 Ejercicios

Capítulo 3

Hello bookdown

All chapters start with a first-level heading followed by your chapter title, like the line above. There should be only one first-level heading (`#`) per .Rmd file.

3.1 A section

All chapter sections start with a second-level (`##`) or higher heading followed by your section title, like the sections above and below here. You can have as many as you want within a chapter.

An unnumbered section

Chapters and sections are numbered by default. To un-number a heading, add a `{.unnumbered}` or the shorter `{-}` at the end of the heading, like in this section.

Capítulo 4

Cross-references

Cross-references make it easier for your readers to find and link to elements in your book.

4.1 Chapters and sub-chapters

There are two steps to cross-reference any heading:

1. Label the heading: `# Hello world {#nice-label}`.
    - Leave the label off if you like the automated heading generated based on your heading title: for example, `# Hello world` = `# Hello world {#hello-world}`.
    - To label an un-numbered heading, use: `# Hello world {-#nice-label}` or `{# Hello world .unnumbered}`.
2. Next, reference the labeled heading anywhere in the text using `\@ref(nice-label)`; for example, please see Chapter 4.
    - If you prefer text as the link instead of a numbered reference use: any text you want can go here.

4.2 Captioned figures and tables

Figures and tables *with captions* can also be cross-referenced from elsewhere in your book using `\@ref(fig:chunk-label)` and `\@ref(tab:chunk-label)`, respectively.

See Figure 4.1.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Don't miss Table 4.1.

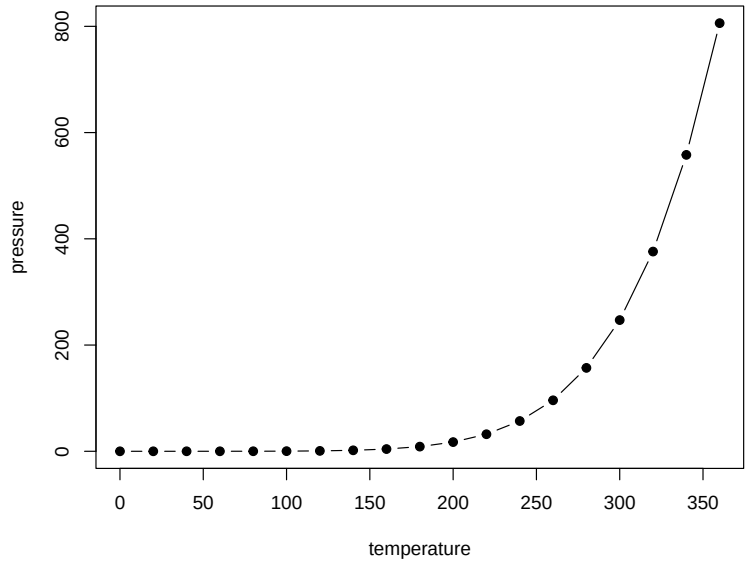

Figura 4.1: Here is a nice figure!

```
knitr::kable(
  head(pressure, 10), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

Tablas 4.1: Here is a nice table!

temperature	pressure
0	0.0002
20	0.0012
40	0.0060
60	0.0300
80	0.0900
100	0.2700
120	0.7500
140	1.8500
160	4.2000
180	8.8000

Capítulo 5

Parts

You can add parts to organize one or more book chapters together. Parts can be inserted at the top of an .Rmd file, before the first-level chapter heading in that same file.

Add a numbered part: `# (PART) Act one {-}` (followed by `# A chapter`)

Add an unnumbered part: `# (PART\*) Act one {-}` (followed by `# A chapter`)

Add an appendix as a special kind of un-numbered part: `# (APPENDIX) Other stuff {-}` (followed by `# A chapter`). Chapters in an appendix are prepended with letters instead of numbers.

Capítulo 6

Footnotes and citations

6.1 Footnotes

Footnotes are put inside the square brackets after a caret `^[]`. Like this one [1].

6.2 Citations

Reference items in your bibliography file(s) using `@key`.

For example, we are using the **bookdown** package [Xie, 2021] (check out the last code chunk in index.Rmd to see how this citation key was added) in this sample book, which was built on top of R Markdown and **knitr** [Xie, 2015] (this citation was added manually in an external file book.bib). Note that the `.bib` files need to be listed in the index.Rmd with the YAML `bibliography` key.

The RStudio Visual Markdown Editor can also make it easier to insert citations: https://rstudio.github.io/visual-markdown-editing/#/citations

[1] This is a footnote.

Capítulo 7

Blocks

7.1 Equations

Here is an equation.

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{7.1}$$

You may refer to using `\@ref(eq:binom)`, like see Equation (7.1).

7.2 Theorems and proofs

Labeled theorems can be referenced in text using `\@ref(thm:tri)`, for example, check out this smart theorem 7.1.

Theorem 7.1. *For a right triangle, if c denotes the* length *of the hypotenuse and a and b denote the lengths of the* ***other*** *two sides, we have*

$$a^2 + b^2 = c^2$$

Read more here https://bookdown.org/yihui/bookdown/markdown-extensions-by-bookdown.html.

7.3 Callout blocks

The R Markdown Cookbook provides more help on how to use custom blocks to design your own callouts: https://bookdown.org/yihui/rmarkdown-cookbook/custom-blocks.html

Capítulo 8

Sharing your book

8.1 Publishing

HTML books can be published online, see: https://bookdown.org/yihui/bookdown/publishing.html

8.2 404 pages

By default, users will be directed to a 404 page if they try to access a webpage that cannot be found. If you'd like to customize your 404 page instead of using the default, you may add either a `_404.Rmd` or `_404.md` file to your project root and use code and/or Markdown syntax.

8.3 Metadata for sharing

Bookdown HTML books will provide HTML metadata for social sharing on platforms like Twitter, Facebook, and LinkedIn, using information you provide in the `index.Rmd` YAML. To setup, set the `url` for your book and the path to your `cover-image` file. Your book's `title` and `description` are also used.

This `gitbook` uses the same social sharing data across all chapters in your book- all links shared will look the same.

Specify your book's source repository on GitHub using the `edit` key under the configuration options in the `_output.yml` file, which allows users to suggest an edit by linking to a chapter's source file.

Read more about the features of this output format here:

https://pkgs.rstudio.com/bookdown/reference/gitbook.html

Or use:

```
?bookdown::gitbook
```

Bibliografía

Yihui Xie. *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition, 2015. URL http://yihui.org/knitr/. ISBN 978-1498716963.

Yihui Xie. *bookdown: Authoring Books and Technical Documents with R Markdown*, 2021. URL https://CRAN.R-project.org/package=bookdown. R package version 0.24.