Estructuras de datos

Lo que debes saber antes de comenzar esta unidad

- Ciclos `while` y contadores.

```
while num < 10:
    num += 1
```

- Ciclos for.

```
for i in range(10):
    print(i)
```

- Las funciones nos permiten:
  - Reutilizar código.
  - Ordenarlo.
  - Evitar repeticiones innecesarias (DRY).
  - Abstraernos del problema.

- Para crear una función ocuparemos la instrucción `def`.

```
def funcion():
```

- Las funciones pueden recibir parámetros (obligatorios y opcionales).

```
def funcion(x, y, z = 5):
    print(z)
```

- Las funciones deben tener un retorno explícito.

```
def funcion(x, y, z=5):
    return z
```

- Lo que se define dentro de una función queda dentro de la función.

```
def foo():
    return bar = 5

foo()
print(bar)
```

- El alcance de una variable significa desde donde podemos accederla. Hemos estudiado dos tipos de variables.
    - Locales (desde sólo se pueden acceder desde el método).
    - Globales (se pueden acceder desde cualquier parte del programa).

- Las variables globales son peligrosas, especialmente cuando trabajamos con más personas o código de otros puesto que es muy probable que sobre-escribamos variables.

- Mostrar un menú y permitir escoger opciones.

- Validar la entrada de un dato.

Capítulo 1: Introducción a listas (arrays)

Competencias:

- Conocer el uso de las listas.
- Crear listas.
- Mostrar una lista.
- Acceder a un elemento de una lista ocupando su índice.

Introducción a Listas

Las listas, también conocidas como arreglos en español, son contenedores que permiten agregar múltiples datos; En lugar de guardar un sólo número o un solo string, nos permitirán guardar varios simultáneamente.

```
# Estos elementos pueden ser de un mismo tipo de datos, por ejemplo solo strings:

animales = ['gato','perro','raton']

# O pueden ser de distintos tipos de dato.

lista_heterogenea = [1, "gato", 3.0, False]
```

Los elementos de la lista se pueden modificar, ya sea modificando los valores de éstos, agregando elementos, o eliminando elementos. Por eso es que se dice que las listas son mutables.

Las listas son muy utilizadas dentro de la programación. Son útiles para resolver diversos tipos de problemas. Por ejemplo, si se quiere hacer una aplicación web, se puede traer los datos de la base de datos a una lista y operar los datos desde ahí. También son útiles para leer archivos y crear gráficos entre múltiples otras funciones.

Definir y mostrar una lista

Para crear una lista utilizaremos la siguiente sintaxis.

```
a = [1, 2, 3, 4]
```

Se definen con los paréntesis de corchete `[]`; Todo lo que esté dentro de los `[]`, separados por coma, son los elementos de la lista.

Lo más sencillo que podemos hacer con una lista es mostrar sus elementos. Para esto podemos ocupar `print` o llamar a su objeto contenedor.

```
print([1, 2, 'hola', 4])
```

```
[1, 2, 'hola', 4]
```

```
a
```

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

Índices

Cada elemento en la lista tiene una posición específica, la que se conoce como índice.

El índice permite acceder al elemento que está dentro de la lista. Los índices van de cero hasta `n - 1`, donde `n` es la cantidad de elementos en la lista.

En una lista que contiene 4 elementos, el primer elemento está en la posición cero, y el último en la 3.

Para acceder al elemento de una posición específica de la lista, simplemente se debe escribir el nombre de la lista, y entre paréntesis de corchetes, el índice de la posición.

```
colores = ["verde", "rojo", "rosa", "azul"]
print(colores[0])
print(colores[1])
print(colores[3])
```

```
verde
rojo
azul
```

Índices mas allá de los límites

En caso de que el índice sea mayor o igual a la cantidad de elementos en la lista, Python arrojará un IndexError: list index out of range, que indica que el índice se posiciona fuera del rango de la lista.

```
colores[8]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-4-cf0cdf800f9c> in <module>()
----> 1 colores[8]
IndexError: list index out of range
```

Índices negativos

Los índices también se pueden utilizar con números negativos y de esta forma referirse a los elementos desde el último al primero.

```
a = [1, 2, 3, 4, 5]
a[-1]
```

```
5
```

Desafío

En base a la lista `a`, ¿qué se muestra en cada petición con los índices?

```
a = [1, 2, 3, 4, 'hola', 8]
```

- `a[0]`.
- `a[7]`.
- `a[a[0]]`.
- `a[4]`.
- `a[-1]`.

ARGV

Ya se ha trabajado anteriormente con listas. Cuando implementábamos `sys.argv` para extraer información del usuario desde la línea de comando, lo que devuelve es una lista.

```
import sys
type(sys.argv)
```

```
list
```

Es por eso que se accede a sus elementos como `sys.argv[0]` y `sys.argv[1]`.

Resumen

- Las listas nos permiten guardar varios datos en una sola variable.
- Podemos mostrar los datos dentro de una lista con `print` o llamándolas desde su objeto contenedor.
- Los elementos dentro de una lista tienen un índice que indica su posición.
- Podemos acceder al elemento ocupando el índice, ej: `a[0]` => primer elemento.
- Los índices negativos nos permiten acceder a la lista desde atrás, ej: `a[-1]` => Último elemento.
- Utilizar un índice más allá del largo de una lista arroja un error IndexError.

Capítulo 2: Operaciones y funciones básicas en una lista

Competencias

- Agregar y eliminar elementos a una lista.
- Contar elementos en una lista.
- Saber si un elemento se encuentra dentro de una lista.

Motivación

Las listas de Python tienen una serie de funciones que permiten realizar operaciones básicas, entre ellas, ordenar los elementos de una lista, saber cuántas veces se encuentra un elemento en la lista, borrar y agregar elementos. Por lo mismo, es muy importante saber leer la documentación asociada a éstas.

Leyendo la documentación de listas

La información oficial de Python sobre las listas (y de manera más general, estructuras de datos) se encuentra en docs.python.org. Se puede llegar a ella fácilmente por medio de google. Lo único que uno debe asegurarse es que la versión que se esté consultando sea la misma con la que se está trabajando.

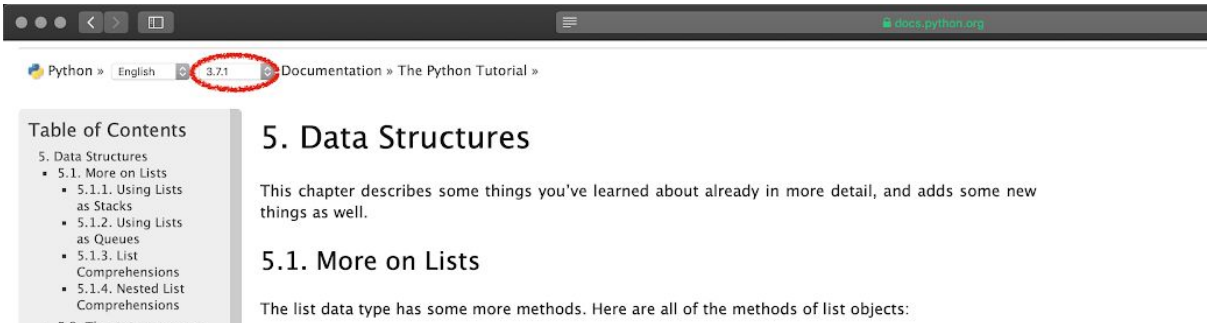

Imagen 1. Verificando la versión.

Funciones aplicables a listas

Cuando definimos una lista en Python, el intérprete infiere cuál es la mejor representación de la expresión. En base a este punto, también le delega a la expresión una serie de acciones que pueda realizar. Estas acciones se conocen como **funciones de lista**.

Cuando generamos una nueva lista y la asignamos a una variable, su generación viene con una serie de atributos y funciones asociadas. Esta es otra de las virtudes del paradigma objeto orientado: cada objeto o representación creada vendrá con una serie de funcionalidades agregadas.

La forma canónica para llamar a una función de un objeto se llama notación de punto: `objeto.función(argumentos)`.

Al generar un objeto llamado `lista_de_numeros` que se compone de los números del 1 al 10, Python le concederá una serie de acciones dado que infiere que su mejor representación es una lista. Se puede ver cuáles son todas las posibles acciones, utilizando el atributo `__dir__()`.

```
lista_de_numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9 ,10]
```

```
print(lista_de_numeros.__dir__())
```

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

Aquellos elementos que están envueltos por `__` se conocen como magic built-in o dunders, y buscan generar flexibilizaciones en el comportamiento de la clases. Serán retomados cuando hablemos de clases. Todas las que no son dunders son las acciones disponibles a acceder.

Función `append(x)`

Para agregar elementos a una lista, se debe utilizar la función `append`. Por ejemplo, si queremos agregar el celeste a nuestra lista de colores, se hace de la siguiente forma:

```
colores.append("celeste")
```

```
# pedimos una representación actualizada de la lista
colores
```

```
['verde', 'rojo', 'rosa', 'azul', 'celeste']
```

Los nuevos elementos siempre se agregaran al final de la lista.

Cabe destacar que no es necesario declarar de forma explícita la asignación al objeto, dado que append es una función inherente al objeto, opera y lo actualiza dentro de ella.

Función `insert(i, x)`

Esta función nos permite agregar elementos en una posición específica.

Por ejemplo, si agregamos el número 13 a nuestra `lista_de_numero`, tendremos el problema de que saltamos el número 12.

```
lista_de_numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
lista_de_numeros.append(13)
```

```
lista_de_numeros
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13]
```

Para resolver este problema, podemos especificar en qué posición de la lista vamos a incorporar un elemento. Para poder ingresar un elemento en una posición específica de la lista podemos hacer uso de la función `.insert(i, x)`. La función necesita la posición a modificar como primer argumento `(i)` y el elemento a ingresar como el segundo `(x)`. Insertemos el número 12 donde corresponde.

```
lista_de_numeros.insert(11, 12)
```

```
lista_de_numeros
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

Función `pop()`

Para sacar el último elemento dentro de una lista, y obtenerlo, se debe utilizar la función .pop(). Por defecto, la función eliminará el último elemento de la lista y lo imprimirá. Si la expresión es asignada a una variable, la variable almacenará el elemento que se sacó.

También es posible pasar como argumento de la función una posición específica. De esta forma, se buscará dentro de la lista el elemento en esa posición, y éste será eliminado y entregado.

```
colores.pop()
```

```
'celeste'
```

```
colores.pop(3)
```

```
'azul'
```

Función `remove(x)`

Para remover un elemento específico, se utiliza la función `remove(x)`, donde `x` es el elemento específico a eliminar. En caso que el elemento no esté presente en la lista, Python arrojará un error `ValueError`.

```
colores.remove("verde")
```

```
colores
```

```
['rojo', 'rosa']
```

```
colores.remove("azul")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call
last)

<ipython-input-13-86badd94da6c> in <module>()
----> 1 colores.remove("azul")

ValueError: list.remove(x): x not in list
```

Función `reverse()`

Se puede invertir el orden de los elementos de una lista utilizando `.reverse`.

```
numeros = [100, 20, 70, 500]
animales = ["perro", "gato", "hurón", "erizo"]
numeros.reverse()
animales.reverse()
```

```
numeros
```

```
[500, 70, 20, 100]
```

```
animales
```

```
['erizo', 'hurón', 'gato', 'perro']
```

Función `sort()`

Se puede ordenar los elementos de forma ascendente utilizando `.sort`. En caso de que se trate de strings, se ordenan de forma ascendente en el abecedario.

```
animales.sort()
numeros.sort()
```

```
animales
```

```
['erizo', 'gato', 'hurón', 'perro']
```

```
numeros
```

```
[20, 70, 100, 500]
```

Si se quiere ordenar de forma descendente, se puede aplicar `reverse()` luego de aplicar `sort()`.

```
numeros.reverse()
numeros
```

```
[500, 100, 70, 20]
```

Un aspecto relevante a recalcar y tomar en cuenta, es que al trabajar con listas en Python **todas estas funciones se realizaron sobre la misma lista**. Por tanto, si no se guarda la lista original en algún objeto separado, esta información se pierde.

Desafío: Integrando funciones de listas

En este ejercicio se aplican varias de las funciones vistas. Definiremos una lista para los ingredientes de una pizza, y luego solicitaremos al usuario si desea agregar, modificar, o eliminar ingredientes.

Aplicaremos un menú de opciones, y un ciclo `while` para integrar también contenido de unidades anteriores.

También utilizaremos el operador `in`, el cual devolverá `True` si el elemento consultado se encuentra en la lista, y `False` en caso contrario.

```
# Definir ingredientes base
ingredientes = ["masa tradicional", "salsa de tomate", "queso"]
opcion = 1

# Inicializar ciclo while con opciones
while opcion < 6:
    print("¡Gracias por ordenar con nosotros!, ¿Qué desea realizar?")
    print("1. Consultar ingredientes de la pizza")
    print("2. Cambiar tipo de masa")
    print("3. Cambiar tipo de salsa")
    print("4. Agregar ingredientes")
    print("5. Eliminar ingredientes")
    print("6. Ordenar")

    # Guardar opción ingresada como int
    opcion = int(input())

    # Opción 1, mostrar ingredientes
    if opcion == 1:
        print("Ingredientes seleccionados:", ingredientes)
        print()

    # Opción 2, cambiar tipo de masa
    if opcion == 2:
```

```
        # 2a: Consultar si entre los ingredientes está la masa
tradicional
        if "masa tradicional" in ingredientes:
            print("¿Cambiar masa tradicional por masa delgada?")
            print("Escriba 'S' para cambiar, o 'N' para conservar masa
tradicional")
            print()
            cambiar = input().upper()

            # Si la respuesta es sí, se elimina masa tradicional, y se
inserta la masa delgada
            if cambiar == "S":
                ingredientes.remove("masa tradicional")
                ingredientes.insert(0, "masa delgada")

        # 2b: Si tiene masa delgada
        elif "masa delgada" in ingredientes:
            print("¿Cambiar masa delgada por tradicional?")
            print("Escriba 'S' para cambiar, o 'N' para conservar masa
delgada")
            print()
            cambiar = input().upper()

            if cambiar == 'S':
                ingredientes.remove("masa delgada")
                ingredientes.insert(0, "masa tradicional")

        else:
            print("¡Su pizza no tiene masa!")
            print("Escoja '1' si desea masa tradicional, o '2' si desea
masa delgada")
            print()

            masa = input()

            if masa == "1":
                ingredientes.insert(0, "masa tradicional")
            elif masa == "2":
                ingredientes.insert(0, "masa delgada")
```

```
    # Opción 4, agregar ingredientes
    elif opcion == 4:
        # Lista de ingredientes
        print("¿Qué ingrediente desea agregar?")
        print("Tomate")
        print("Aceitunas")
        print("Queso")
        print("Peperoni")
        print("Pollo")
        print()
        nuevo = input.lower()

        # Validar que sea un ingrediente permitido
        while nuevo != "tomate" and nuevo != "aceitunas" and nuevo !=
"queso" and nuevo != "peperoni" and nuevo != "pollo":
            print("Ingrese un ingrediente válido")
            print("Tomate")
            print("Aceitunas")
            print("Queso")
            print("Peperoni")
            print("Pollo")
            print()
            nuevo = input().lower()

        # Agregar ingrediente al final de la lista
        ingredientes.append(nuevo)

    elif opcion == 5:
        # Consultamos el ingrediente para eliminar
        print("Ingredientes actuales:", ingredientes)
        print("Escriba el que desea eliminar")
        print()
        eliminar = input().lower()

        # Otra forma de validar
        while eliminar not in ingredientes:
            print("Ingredientes actuales:", ingredientes)
            print("Escriba el que desea eliminar")
            print()
            eliminar = input().lower()

        # Eliminar con remove
        ingredientes.remove(eliminar)
```

Sugerencias de mejoras

1. Validaciones para que dentro de las opciones principales solo se escojan de la 1 a la 6.

2. Validaciones para que dentro de las opciones que lo requieran, solo se ingresen los campos permitidos.

3. Agregar más mensajes de print para ayudar en el flujo al usuario.

4. Replicar la lógica de las masas con la salsa (salsa de tomate o salsa BBQ).

5. Crear otra lista con los "ingredientes base", y utilizar esta lista para mostrar los ingredientes disponibles, y validar la entrada del usuario.

6. Consultar por agregar o eliminar ingredientes continuamente (sin tener que volver al menú principal) hasta que el usuario indique que ya no desea hacer modificaciones.

7. Contar la frecuencia de cada ingrediente.

8. Ordenar los ingredientes alfabéticamente, omitiendo la masa y la salsa que deben salir al comienzo.

9. Omitir masa y salsa en la lista de ingredientes para eliminar.

10. Asignar un precio a cada ingrediente y calcular el valor final de la pizza, y mostrarlo al escoger la opción "ordenar".

Operaciones: Concatenación de listas

Otro comportamiento útil de las listas es la capacidad de responder a instrucciones que no son inherentes a clase. Éstas se conocen como operaciones. Ya hemos trabajado anteriormente con ellas. Por ejemplo, al concatenar dos strings entre sí, se está haciendo uso de una operación `+`.

Un aspecto a mencionar es que muchas de las operaciones **también son inferidas en función de las estructuras de datos a manipular**. Si sabemos que podemos concatenar dos strings con +, cuando concatenamos dos listas obtenemos lo siguiente:

```
# definamos dos listas de animales
animales = ['Gato', 'Perro', 'Tortuga']
animales_2 = ['Hurón', 'Hamster', 'Erizo de Tierra']

# Si las concatenamos, podremos obtener una lista de mascotas
mascotas = animales + animales_2

# Veamos algunas características
print(animales)
print(len(animales))
print(animales_2)
print(len(animales_2))
print(mascotas)
print(len(mascotas))
```

```
['Gato', 'Perro', 'Tortuga']
3
['Hurón', 'Hamster', 'Erizo de Tierra']
3
['Gato', 'Perro', 'Tortuga', 'Hurón', 'Hamster', 'Erizo de Tierra']
6
```

La función `len()` responde de similar manera a los strings cuando es aplicado en una lista. Genera un reporte de la cantidad de elementos contenidos en la lista.

Operaciones: Repitiendo listas

Al utilizar el operador `*`, los elementos de la lista se multiplican. Por ejemplo, se nos ha informado que hay 4 veces más perros, gatos y tortugas que hurones, hamsters y erizos de tierra. Podríamos generar una lista nueva e ingresar cada elemento nuevamente, pero sería tedioso para nosotros.

Lo que vamos a realizar es multiplicar los elementos contenidos en la lista mediante el operador `*`.

```
animales_actualizados = animales *4

mascotas = animales_actualizados + animales_2

# Veamos algunas características
print(animales_actualizados)
print(len(animales_actualizados))
print(animales_2)
print(len(animales_2))
print(mascotas)
print(len(mascotas))
```

```
['Gato', 'Perro', 'Tortuga', 'Gato', 'Perro', 'Tortuga', 'Gato',
'Perro', 'Tortuga', 'Gato', 'Perro', 'Tortuga']
12
['Hurón', 'Hamster', 'Erizo de Tierra']
3
['Gato', 'Perro', 'Tortuga', 'Gato', 'Perro', 'Tortuga', 'Gato',
'Perro', 'Tortuga', 'Gato', 'Perro', 'Tortuga', 'Hurón', 'Hamster',
'Erizo de Tierra']
15
```

Membresías en listas

Se puede saber si cierto elemento se encuentra en la lista, mediante el operador booleano `in`. Por ejemplo, queremos ver si encontramos un Tapir en la lista de mascotas.

```
'Tapir' in mascotas
```

```
False
```

Por ser un operador booleano, el retorno del operador `in` solo puede ser `True` o `False`. Si ahora preguntamos por la existencia de gatos, podríamos hacer lo siguiente:

```
'Gato' in mascotas
```

```
True
```

También es posible saber cuántos elementos existen de un tipo, mediante la función `.count()` que existe en la lista. No confundir con la función `count()` de los objetos string.

```
mascotas.count('Gato')
```

```
4
```

Comprensiones de listas

Un elemento en particular de Python es la existencia de comprensiones de listas. Mediante éstas, es posible modificar todos sus elementos y retornar una nueva lista con los nuevos elementos, de forma más concisa que utilizando el tradicional ciclo `for`.

Ejemplo

Se requiere escribir en mayúsculas todos los nombres de animales en nuestra lista `mascotas`. Sin comprensiones de lista, esto se puede lograr como:

```
mascotas_mayusculas = []

for i in mascotas:
    mascotas_mayusculas.append(i.upper())

mascotas_mayusculas
```

```
['GATO',
 'PERRO',
 'TORTUGA',
 'GATO',
 'PERRO',
 'TORTUGA',
 'GATO',
 'PERRO',
 'TORTUGA',
 'GATO',
 'PERRO',
 'TORTUGA',
 'HURÓN',
 'HAMSTER',
 'ERIZO DE TIERRA']
```

Si bien el ejercicio es válido, se pierde tiempo en definir la lista contenedora y realizando append en ésta en cada iteración. Una comprensión de lista presenta una forma más elegante de realizar lo mismo.

```
mascotas_mayusculas = [i.upper() for i in mascotas]
```

```
mascotas_mayusculas
```

```
['GATO',
 'PERRO',
 'TORTUGA',
 'GATO',
 'PERRO',
 'TORTUGA',
 'GATO',
 'PERRO',
 'TORTUGA',
 'GATO',
 'PERRO',
 'TORTUGA',
 'HURÓN',
 'HAMSTER',
 'ERIZO DE TIERRA']
```

La forma canónica de una compresión de lista es:

```
nuevo_objeto = [expresion for iterable in lista]
```

¿Qué pasa si se desea poner en mayúsculas solo a los gatos, y los demás en minúsculas?. La solución con `for` sería la siguiente:

```
mascotas_mayusculas = []

for i in mascotas:
    if i == 'Gato':
        mascotas_mayusculas.append(i.upper())
    else:
        mascotas_mayusculas.append(i.lower())

mascotas_mayusculas
```

```
['GATO',
 'perro',
 'tortuga',
 'GATO',
 'perro',
 'tortuga',
 'GATO',
 'perro',
 'tortuga',
 'GATO',
 'perro',
 'tortuga',
 'hurón',
 'hamster',
 'erizo de tierra']
```

Mientras que con una comprensión de lista:

```
[x.upper() if x =='Gato' else x.lower() for x in mascotas]
```

```
['GATO',
 'perro',
 'tortuga',
 'GATO',
 'perro',
 'tortuga',
 'GATO',
 'perro',
 'tortuga',
 'GATO',
 'perro',
 'tortuga',
 'hurón',
 'hamster',
 'erizo de tierra']
```

Capítulo 3: Iterando una lista a partir del índice

Competencias

- Recorrer los elementos de una lista.
- Entender la importancia de guardar la lista original en ciertos casos.
- Transformar los elementos dentro de la lista mientras es recorrida.

Motivación

En este capítulo estudiaremos cómo recorrer una lista y operar sobre sus datos para resolver diversos tipos de problema.

Formas de iterar

En Python existen diversas formas de iterar una lista o un arreglo. Cada una tiene sus puntos a favor y en contra. A grandes rasgos, podemos identificar las siguientes estrategias de iteración:

- Iterando por los elementos.
- Iterando por su posición.

¿Qué tipos de problema se puede resolver recorriendo una lista?

- Recorrer los elementos de una lista permite operar sobre sus valores.

- Hay tareas frecuentes que se pueden enfrentar como programadores, tales como filtrar o transformar valores.

- Por ejemplo, dada una lista de precios, es posible encontrar el valor promedio de ésta, o incrementar todos los valores.

Iterando una lista por posición

Hemos visto que es posible acceder a un elemento de una lista en una posición específica utilizando la sintaxis `a[posición]`. Esto se puede aplicar al iterar sobre ciclos `for` o `while`.

Con `for`

```
a = [1, 2, 3, 4, 5, 6]
n = len(a)

for i in range(n):
    print(a[i])
```

```
1
2
3
4
5
6
```

Con `while`

```
count = 0

while count < n:
    print(a[count])
    count += 1
```

```
1
2
3
4
5
6
```

Cuidado con las condiciones de borde

En el ejemplo anterior se definió la condición de tope como `n = len(a)`. Esto es una buena práctica dado que nunca se debe declarar explícitamente el número de iteraciones a realizar. Es mejor declararlo dinámicamente con operaciones. Veamos el siguiente ejemplo:

```
# Tenemos nuestra lista a, y un colaborador la modifica
for i in [10, 12, 42]:
    a.append(i)
```

```
a
```

```
[1, 2, 3, 4, 5, 6, 10, 12, 42]
```

```
# Definamos nuestro iterador de forma fija
n = 5
i = 0

while i < n:
    print(a[i])
    i += 1
```

```
1
2
3
4
5
```

El problema existente en este caso es que toda la nueva información agregada en la lista (números 10, 13 y 42), es ignorada. Para evitar este tipo de problema, es mejor implementar la condición de entrada como `n = len(a)`. Esto hace que el ciclo y el código sea dinámico a nuevas versiones de la lista `n`.

Iterando la lista por sus elementos

Al iterar directamente una lista en un ciclo `for`, el iterador `i` toma el valor del **elemento recorrido** en cada iteración, **no su índice**.

Ejemplo

Se tiene una lista heterogénea (que tiene elementos con distintos tipos de datos), y se desea contar los elementos strings **accediendo a su posición específica**.

Un primer enfoque sería implementar un loop clásico.

```
lista_heterogenea = ['Lechugas', 'Tomates', 5, 10,
                     True, False, True, 'Papas',
                     5.1, 45.2, 1, 2, 0]

count_str = 0
for i in lista_heterogenea:
    if type(lista_heterogenea[i]) is 'str':
        count_str += 1
    else:
        pass
```

```
---------------------------------------------------------------------------
---

TypeError                                 Traceback (most recent call
last)

<ipython-input-25-f67ebd4371c7> in <module>()
      5 count_str = 0
      6 for i in lista_heterogenea:
----> 7     if type(lista_heterogenea[i]) is 'str':
      8         count_str += 1
      9     else:
```

```
TypeError: list indices must be integers or slices, not str
```

En este ejemplo, Python arrojará un `TypeError: list indices must be integers or slices, not str`. Este error indica que el índice de la lista es inválido.

Esto se debe a que se está utilizando un string, el elemento que se está recorriendo en la iteración, como índice de la posición del elemento.

Para resolver este problema, se necesita encontrar una función que entregue tanto la posición como el elemento en sí de la lista. Para ello se debe utilizar `enumerate`.

Primero veremos qué entrega la función.

```
for index, element in enumerate(lista_heterogenea):
    print("El elemento {} tiene la posición -> {}".format(element,
index))
```

```
El elemento Lechugas tiene la posición -> 0
El elemento Tomates tiene la posición -> 1
El elemento 5 tiene la posición -> 2
El elemento 10 tiene la posición -> 3
El elemento True tiene la posición -> 4
El elemento False tiene la posición -> 5
El elemento True tiene la posición -> 6
El elemento Papas tiene la posición -> 7
El elemento 5.1 tiene la posición -> 8
El elemento 45.2 tiene la posición -> 9
El elemento 1 tiene la posición -> 10
El elemento 2 tiene la posición -> 11
El elemento 0 tiene la posición -> 12
```

Mediante `enumerate` se puede acceder tanto al elemento mismo como a su índice dentro de la lista. Un aspecto a considerar es que **se debe declarar los dos iterables** en el ciclo `for`. Si no se quiere extraer el elemento y solo se requiere el índice, se puede ignorar su iterable en las expresiones, o implementar un placeholder representado con `_`.

```
for index, _ in enumerate(lista_heterogenea):
    print(index)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
```

Ya entendiendo qué hace enumerate, es fácil desarrollar la solución:

```
count_str = 0
for i, _ in enumerate(lista_heterogenea):
    if type(lista_heterogenea[i]) is str :
        count_str += 1
    else:
        pass

print("La cantidad de strings en la lista es: ", count_str)
```

```
La cantidad de strings en la lista es:  3
```

¿Qué tipo de problemas se pueden resolver mediante la iteración?

Aplicando lo aprendido, es posible:

- Transformar el tipo de dato de todos los elementos, por ejemplo de entero a string.
- Filtrar (seleccionar o mostrar) solo los elementos que cumplen algún criterio.
- Reducir; Por ejemplo sumar todos los elementos, concatenarlos o multiplicarlos.

Transformación

Sabiendo que el usuario puede introducir múltiples datos vía sys.argv, ¿Qué pasaría si se quiere transformar todos estos datos a enteros?. Para lograrlo, se debe recorrer el arreglo de `sys.argv` elemento a elemento, transformar cada uno, y guardando cada nuevo elemento en una nueva lista.

```
import sys

user_arguments = sys.argv
n = len(user_arguments)

# una buena práctica es siempre guardar la información en un nuevo
objeto

new_arguments = []

for i in range(n):
    if i > 0:
        new_arguments.append(int(user_arguments[i]))
```

Filtrado

Filtrar una lista consiste en seleccionar elementos específicos omitiendo el resto mediante una condición. Por ejemplo, podríamos crear un programa que maneje datos astronómicos, y se solicite descartar (filtrar) las mediciones erróneas.

Ejemplo de filtrado

Se pide crear un programa que filtre todos los números de una lista que sean menores a 1000. Esto es lo mismo que decir "**seleccionar todos los elementos mayor o iguales a mil**".

```
a = [100, 200, 1000, 5000, 10000, 10, 5000]
n = len(a)
filtered_array = []

for i in range(n):
    if a[i] >= 1000:
        filtered_array.append(a[i])

filtered_array
```

```
[1000, 5000, 10000, 5000]
```

En los siguientes capítulos se resolverán muchos problemas de este tipo.

Resumen

En este capítulo se estudió la base de resolución de problemas con listas. A partir de ahora se seguirá trabajando con esta lógica para resolver diversos tipos de problemas.

Capítulo 4: Desafíos de listas

Competencia

- Aplicar operaciones de listas.

Desafío - Adictos a redes

- Se tiene una lista con la cantidad de minutos usados en redes sociales de distintos usuarios.
- Se pide crear la función `adictos_a_redes(lista)` que reciba una lista con los minutos de uso.
- La función debe retornar un nuevo arreglo con todas las medidas inferiores a 90 minutos como 'bien' y todas las mayores o iguales a 90 como 'mal'.

El output debería ser algo similar a lo siguiente:

```
scan_addicts([120, 50, 600, 30, 90, 10, 200, 0, 500])
# ["mal", "bien", "mal", "bien", "bien", "bien", "mal", "bien", "mal"]
```

```
# Solución
def adictos_a_redes(lista):
    results = []
    for i in lista:
        if i > 90:
            results.append('mal')
        else:
            results.append('bien')
    return results

lista_adictos = [120, 50, 600, 30, 90, 10, 200, 0, 500]
print(adictos_a_redes(lista_adictos))
```

```
['mal', 'bien', 'mal', 'bien', 'bien', 'bien', 'mal', 'bien', 'mal']
```

Desafío - Adictos v2

- Se pide crear el programa `adictos2.0.py` con una función llamada `scan_addicts2`.
- La función debe recibir una lista con los minutos de uso de los usuarios.
- Debe retornar una nueva lista cambiando todas las medidas inferiores a 90 minutos como 'bien', entre 90 y 180 como 'mejorable', y todas las mayores o iguales a 180 como 'mal'.

Tip: *Cuidado con las condiciones de borde; analizar los casos de 90 y 180.*

```
# Solución

def scan_addicts2(lista):
    results = []

    for i in lista:
        if i >= 180:
            results.append('mal')
        elif i >= 90:
            results.append('mejorable')
        else:
            results.append('bien')

    return results

lista_adictos = [120, 90, 600, 30, 90, 10, 200, 180, 500]
print(scan_addicts2(lista_adictos))
```

```
['mejorable', 'mejorable', 'mal', 'bien', 'mejorable', 'bien', 'mal',
'mal', 'mal']
```

Desafío: Transformando segundos a minutos

- Se tiene una lista con la cantidad de segundos que demoraron algunos procesos.
- Se necesita una función para transformar todos los datos a minutos, (las fracciones de minutos serán ignoradas).

Tip: *Por defecto Python 3 implementa una división que devuelve un flotante. Lo que necesitamos es la parte entera. Ocupe // para resolver este inconveniente.*

```
seconds = [100, 50, 1000, 5000, 1000, 500]
```

```
# Solución

def to_minutes(lista):
    results = []
    for i in lista:
        results.append(i // 60)
    return results

seconds = [100, 50, 1000, 5000, 1000, 500]
print(to_minutes(seconds))
```

```
[1, 0, 16, 83, 16, 8]
```

Desafío: `normalización.py`

La normalización vectorial busca reexpresar entre 0 y 1 una serie de valores contenidos en una lista. Suena mucho más complejo de lo que es.

Consiste en que, dado un arreglo con valores, se genere un nuevo arreglo donde todos los valores están entre cero y uno.

Para lograrlo, cada valor del arreglo se debe dividir por el módulo, y el módulo se calcula como la raíz de la suma de cada uno de los elementos al cuadrado.

$$Modulo([1,2,3]) = \sqrt{(1^2 + 2^2 + 3^2)}$$

Se necesita:

- Una función para calcular el módulo de un arreglo.
- Dividir cada uno de los elementos del arreglo por el módulo y guardarlos en un nuevo arreglo.

El resultado es posible de verificar dado que la suma de todos los valores al cuadrado debe ser uno.

Tip: *Para implementar la raíz cuadrada, importar el módulo* `math` *y ocupar la función* `sqrt`.

```
# Solución

import math

def modulo(lista):
    suma = 0
    for i in lista:
        suma = suma + i**2
    return math.sqrt(suma)

vector = [1, 2, 3, 4, 5, 6]
modulo(vector)
```

```
3.7416573867739413
```

```
def normalizar(lista):
    m = modulo(lista)
    normalized_array = []

    for i in lista:
        normalized_array.append(i / m)

    return normalized_array

output_array = normalizar([1, 2, 3])
print(output_array)
```

```
[0.2672612419124244, 0.5345224838248488, 0.8017837257372732]
```

```
## Verificamos el resultado

# Dijimos que la suma de cada uno de los números al cuadrado debe ser
uno,
# esto lo podemos probar iterando el arreglo.

suma = 0
for i in output_array:
    suma += i ** 2

print(suma)
```

Desafío: Artículos de tienda

- Se tiene una lista de categoría, nombres y precios de artículos que están intercalados, donde primero viene la categoría, luego el nombre del artículo y luego su precio.

- Los precios están ingresados como datos de texto, no como numéricos:

```
articulos = ["celular", "LG K10", "90000", "tablet",  "Galaxy TAB",
"80000", "smart tv", "LED 43 Samsung", "485000", "celular", "Galaxy J7",
"120000", "celular", "Huawei Y5", "59900", "notebook", "Lenovo ideapad",
"250000", "tablet", "Huawei media", "139000", "notebook", "Acer",
"145000"]
```

- El dueño de la tienda requiere detectar aquellos artículos que cuesten más de $80.000, que no sean notebooks, y aplicarles un descuento de 10%.

- Se requiere que estos nuevos precios se separen en listas por categoría de artículo, ordenados de mayor a menor.

Solución

```
articulos = ["celular", "LG K10", "90000", "tablet",  "Galaxy TAB",
"80000", "smart tv", "LED 43 Samsung", "485000",
             "celular", "Galaxy J7", "120000", "celular", "Huawei Y5",
"59900", "notebook", "Lenovo ideapad", "250000",
             "tablet", "Huawei media", "139000", "notebook", "Acer",
"145000"]

# Primero se definen listas vacías para cada categoría
celulares = []
tablets = []
smart_tv = []
notebooks = []

# Generamos variable auxiliar para almacenar la categoría recorrida
aux = ""


# Luego se itera la lista de artículos con enumerate
for index, elemento in enumerate(articulos):

    # Si es el primer elemento, o su índice es divisible por 3, entonces
el elemento es una categoría
    # Se almacena por tanto ese valor en la variable auxiliar
    if index == 0 or index % 3 == 0:
        aux = elemento

    # Si el index recorrido menos 2 es divisible por 3, el elemento
iterado es un precio
    elif (index - 2) % 3 == 0:
        # Almacenamos el precio como int
        int_precio = int(elemento)
        precio_final = int_precio

        # Se aplica el descuento según el precio actual y la categoría
almacenada en aux
        if precio_final > 80000 and aux is not "notebook":
            precio_final = int(precio_final * 0.9)
```

```
        # Se agrega el precio a la lista correspondiente
        if aux == "celular":
            celulares.append(precio_final)
        elif aux == "notebook":
            notebooks.append(precio_final)
        elif aux == "smart tv":
            smart_tv.append(precio_final)
        elif aux == "tablet":
            tablets.append(precio_final)

# Ordenar precios de mayor a menor
celulares.sort()
celulares.reverse()

notebooks.sort()
notebooks.reverse()

smart_tv.sort()
smart_tv.reverse()

tablets.sort()
tablets.reverse()

# Output
print("Celulares:", celulares)
print("Notebooks:", notebooks)
print("Smart tv:", smart_tv)
print("Tablets:", tablets)
```

```
Celulares: [108000, 81000, 59900]
Notebooks: [250000, 145000]
Smart tv: [436500]
Tablets: [125100, 80000]
```

Capítulo 5: Operaciones funcionales sobre listas

Competencias

- Conocer las operaciones funcionales y sus ventajas.
- Aprender a utilizar `map`.
- Aprender a utilizar `reduce`.
- Aprender a utilizar `filter`.

¿Qué son las operaciones funcionales?

En este capítulo se verá otro enfoque para transformar, filtrar y reducir valores de una lista. Este enfoque se conoce como operaciones funcionales.

Parte de las ventajas de las operaciones funcionales es su expresividad y elegancia.

Existen tres grandes operaciones funcionales en Python: `map`, `reduce` y `filter`.

Map

Map sirve para aplicar una operación a cada elemento de la lista. Al aplicarse, devuelve un objeto de tipo `map` que puede ser convertido a una lista.

La sintaxis de `map` toma dos parámetros de entrada: la expresión a ejecutar y en qué lista se va a aplicar la función.

La expresión a ejecutar se encapsula en un `lambda`: una forma de generar funciones temporales. Un `lambda` se define de la siguiente forma canónica:

```
lambda x: x * 2
```

```
lambda <parametro>: <expresiones_en_el_parametro>
```

Donde:

- `lambda`: Es la expresión que informa a Python que vamos a desarrollar una función anónima.
- `x`: Es el parámetro a modificar.
- `:`: Es el punto de cierre. Posterior a esto, todo se considerará como expresiones a ejecutar.

Evaluemos si ambas listas son iguales:

```
# con for

a = [1, 2, 3, 4, 5, 6, 7]
b_for = []

for i in a:
    b_for.append(i * 2)

b_for
```

```
[2, 4, 6, 8, 10, 12, 14]
```

```
# con map

b_map = list(map(lambda i: i * 2, a))
```

```
b_map
```

```
[2, 4, 6, 8, 10, 12, 14]
```

```
b_for == b_map
```

```
True
```

`map` vs. `for`

Utilizar `map` es mejor que un `for` cuando se desea transformar todos los datos de una lista y devolver una nueva lista. Otra alternativa para lograr algo similar son las comprensiones de listas, con la desventaja de que no son tan simples de entender a primera vista.

Filter

La segunda operación funcional muy útil sobre listas es `filter`. Ésta permite extraer todos los elementos que cumplan con algún criterio específico. De similar manera que con `map`, nos puede retornar una lista si lo envolvemos con el método `list`.

```
a = [1,2,3,4,5,6,7]

return_even = []
for i in a:
    if i % 2 == 0:
        return_even.append(i)

return_even_for
```

```
[2, 4, 6]
```

```
a = [1,2,3,4,5,6,7]

return_even_filter = list(filter(lambda x: x % 2 == 0, a))

return_even_filter
```

```
[2, 4, 6]
```

```
return_even_for == return_even_filter
```

```
True
```

Reduce

Reduce permite operar sobre todos los resultados de la lista. En lugar de devolver una lista, devuelve un único valor con el resultado de la operación aplicada. Por ello, el bloque requiere que se pasen dos variables; la que itera y la que guarda el resultado.

```
from functools import reduce
# Se opera sobre a, donde uno de los parámetros (x) actúa como
acumolador,
# y el otro (y) actúa como iterador
reduce(lambda x, y : x + y, a)
```

```
suma = 0
for i in a:
    suma += i

print(suma)
```

```
28
```

Capítulo 6: Pandas

Competencias

- Importar la librería `pandas` para la ingesta y el preprocesamiento de datos numéricos.
- Conocer las principales estructuras de datos de `pandas` y sus dimensiones.
- Manipular un archivo `.csv` desde `pandas`.
- Realizar operaciones de selección en un `DataFrame`.
- Conocer las principales operaciones en una Serie.
- Generar iteraciones con los distintos métodos de DataFrame.

Introducción a Pandas

Muchas veces trabajaremos con datos que vienen ordenados en archivos de texto plano como `.csv`, `.xls`, `.txt`. Si bien es posible realizar ingesta de éstos con Python nativo, haremos uso explícito de librerías específicas para esta tarea.

En este capítulo se utilizará `pandas`, una librería orientada a la manipulación y limpieza de estructuras de datos que provee estructuras de datos y funciones orientadas a hacer el trabajo con tablas más rápido, fácil y expresivo (McKinney, 2014, 4).

Ésta librería combina el alto desempeño de operaciones vectorizadas de `numpy` (a revisar en el Capítulo 7) con las manipulaciones flexibles de hojas de cálculo y bases de datos relacionales.

Existen dos grandes estructuras de datos en `pandas`: los `DataFrame` y las `Series`. Cada una tiene peculiaridades que revisaremos a detalle posteriormente.

Para poder trabajar con `pandas`, debe importarse con la siguiente nomenclatura:

```
import pandas as pd
```

De esta manera, cada vez que se quiera utilizar alguna función de `pandas`, se debe preceder con el alias `pd`.

Leyendo archivos `csv`

Para ingresar una base de datos en formato `csv`, se utiliza la función `pd.read_csv`. Este método devolverá la representación pandas del archivo en el espacio de trabajo. Una de las buenas prácticas indica guardar la base de datos en la memoria del espacio de trabajo de la siguiente manera:

```
df = pd.read_csv('nations.csv')
```

De esta forma, se obtiene un objeto al cual generar peticiones y aplicar operaciones.

DataFrame

Un `DataFrame` es la representación de **toda la información existente del archivo**. De esta forma, al importar la base de datos al ambiente de trabajo, parte del procesamiento interno de los datos realizados en `pandas` es asignarle una clase específica, llamada `DataFrame`.

Esta clase concederá una serie de atributos, funciones y opciones al objeto para poder manipularlo. Por ejemplo, inspeccionar los datos ingresados.

Para averiguar las dimensiones de los datos (cuántas filas y cuántas columnas tiene la base), se debe acceder a la propiedad `df.shape`. El retorno del comando sugiere que existen 194 filas y 14 columnas.

```
df.shape
```

```
(194, 14)
```

También es posible ver los datos de las primeras filas, utilizando la función `df.head()`. Por defecto, la función retornará las 5 primeras observaciones. Si se desea solicitar más o menos, se debe ingresar la cantidad de observaciones entre sus paréntesis, como argumento.

Vamos a extraer las primeras tres observaciones:

```
df.head(3)
```

```
  Unnamed: 0   country  region           gdp  ...   femlab   literacy
co2  gini
0           1   Algeria  Africa   7300.399902  ...   0.4522  72.599998
15.0   NaN
1           2     Benin  Africa   1338.800049  ...   0.8482  41.700001
1.2   NaN
2           3  Botswana  Africa  12307.400391  ...   0.8870  84.099998
9.2   NaN
```

Existe una columna llamada Unnamed: 0. Ésta representa el número de fila de los datos ingresados, pero pandas ya asigna un índice indicando la posición de cada fila. Por tanto, esta columna no aporta información útil, y debe eliminarse.

Eliminar una columna del dataframe

Esto es posible con la función `df.drop()`. Cabe destacar que este método retornará una nueva versión de los datos, por lo que se debe sobreescribir el objeto si se desea que los cambios realizados persistan.

```
df = df.drop(columns='Unnamed: 0')
```

```
# solicitemos las primeras 3 observaciones
df.head(3)
```

```
(3, 14)
```

Series

Hasta el momento se han aplicado operaciones que afectan a la tabla en su conjunto. Pero en la práctica, muchas de las operaciones de interés se realizan **a nivel de columna**.

En `pandas`, las columnas se conocen como `Series`. A grandes rasgos, las `Series` no distan de las listas en sus principales componentes: Ambas son vectores unidimensionales que almacenan una cantidad finita de datos.

La ventaja de las `Series` es que presentan una serie de acciones y atributos que permiten extraer información de manera rápida.

Por ejemplo, supongamos que queremos contar cuántas observaciones pertenecen a cada país. Para lograr esto, partamos por acceder a la columna específica:

```
df['region']
```

```
0       Africa
1       Africa
2       Africa
3       Africa
4       Africa
5       Africa
6       Africa
7       Africa
8       Africa
9       Africa
10      Africa
11      Africa
12      Africa
13      Africa
14      Africa
15      Africa
16      Africa
17      Africa
18      Africa
19      Africa
20      Africa
21      Africa
22      Africa
23      Africa
```

```
24       Africa
25       Africa
26       Africa
27       Africa
28       Africa
29       Africa
         ...
164      Europe
165      Europe
166      Europe
167      Europe
168      Europe
169      Europe
170      Europe
171      Europe
172      Europe
173      Europe
174      Europe
175      Europe
176      Europe
177      Europe
178      Europe
179    Oceania
180    Oceania
181    Oceania
182    Oceania
183    Oceania
184    Oceania
185    Oceania
186    Oceania
187    Oceania
188    Oceania
189    Oceania
190    Oceania
191    Oceania
192    Oceania
193    Oceania
Name: region, Length: 194, dtype: object
```

Al ocupar la sintaxis `df[<nombre_columna>]`, se está accediendo a la Serie `region`.

Dentro de la serie, es posible conocer la cantidad de cuántas veces se presenta cada atributo dentro de ella, utilizando la función `value_counts()`.

```
df['region'].value_counts()
```

```
Africa      52
Asia        49
Europe      43
Americas    35
Oceania     15
Name: region, dtype: int64
```

Acorde a los resultados, observamos que la mayor cantidad de países pertenecen a África. ¿Qué pasa si se quiere reportar las probabilidades? Una posibilidad es ocupar el operador `/` para dividir cada registro por la cantidad de países existentes. Esto se conoce como una **operación vectorizada**, y se estudiará con detención más adelante

```
df['region'].value_counts() / len(df)
```

```
Africa      0.268041
Asia        0.252577
Europe      0.221649
Americas    0.180412
Oceania     0.077320
Name: region, dtype: float64
```

Al trabajar con una Serie, el retorno de una operación será otra Serie. Por tanto, es posible **concatenar acciones a éstas**. En este caso, si se desea identificar el promedio de países por continente, es posible hacerlo concatenando la función `mean` al final de `value_counts`.

```
df['region'].value_counts().mean()
```

```
38.8
```

Crear un subset mediante `loc` y `slice`

Hasta el momento sabemos que `pandas` presenta dos tipos de estructuras de datos: los `DataFrames` y las `Series`. Las primeras son la representación de una matriz de datos con Filas y Columnas, mientras que las segundas responden a un vector unidimensional de datos que se puede alojar en la matriz. También sabemos que ambas estructuras vienen por definición con una serie de acciones y atributos a los cuales se puede acceder.

Para acceder a alguna observación específica (ya sea fila o columna), se debe hacer mediante la notación de `loc` e `iloc`:

- `loc`: Localiza una observación específica mediante el nombre de una columna y su posición. La forma para acceder a una observación responde a la siguiente nomenclatura.

$$Matriz[Filas,\ Columnas]$$

Por ejemplo, para acceder al campo `'life'` de la novena observación, la sintaxis es:

```
df.loc[8, 'life']
```

```
48.5666656494141
```

También es posible acceder a la fila completa, incluyendo todas las columnas. Esto se logra mediante el operador `:` (**slice**). Éste indica que se está considerando todos los elementos de una lista o serie (los nombres de las columnas también son una Serie).

En este caso, le instruimos a pandas extraer todos los registros de los campos en la novena observación.

```
df.loc[8,:]
```

```
country         Chad
region        Africa
gdp           1266.2
school           1.5
adfert         164.5
chldmort         209
life          48.5667
pop          10509983
urban            26.4
femlab         0.8006
literacy         33.6
co2               0.1
gini              NaN
Name: 8, dtype: object
```

También es posible hacer un subset de datos seleccionando todas las filas y un rango específico de columnas.

Queremos extraer sólo las columnas gdp school adfert chldmort de este subset.

```
df_col_subset = df.loc[:, ['gdp', 'school', 'adfert', 'chldmort']]
df_col_subset.head(1)
```

```
          gdp    school  adfert  chldmort
0  7300.399902  6.716667     7.3     34.75
```

En caso de que se requiera seleccionar "desde" una columna "hasta" otra columna (incluyendo todas las columnas intermedias), también es posible el uso de **slice** `:`. Como slice va a seleccionar todo, sólo sirve si el rango de columnas que se desea **es contínuo**.

```
df_col_subset_1 = df.loc[:, 'gdp':'chldmort']
# veamos si ambos objetos son similares
df_col_subset == df_col_subset_1
```

	gdp	school	adfert	chldmort
0	True	True	True	True
1	True	True	True	True
2	True	True	True	True
3	True	True	True	True
4	True	True	True	True
5	True	True	True	True
6	True	True	True	True
7	True	True	True	True
8	True	True	True	True
9	True	True	True	True
10	True	True	True	True
11	True	True	True	True
12	True	True	True	True
13	True	True	True	True
14	True	True	True	True
15	True	True	True	True
16	True	True	True	True
17	True	True	True	True
18	True	True	True	True
19	True	True	True	True
20	True	True	True	True
21	True	True	True	True
22	True	True	True	True
23	True	True	True	True
24	True	True	True	True
25	True	True	True	True
26	True	True	True	True
27	True	True	True	True
28	True	True	True	True
29	True	True	True	True
..	...	...	...	...
164	True	True	True	True
165	True	True	True	True
166	True	True	True	True
167	True	True	True	True
168	True	True	True	True
169	True	True	True	True
170	False	False	True	True
171	True	True	True	True
172	True	True	True	True
173	True	True	True	True
174	True	True	True	True
175	True	True	True	True

```
176   True    True    True      True
177   True    True    True      True
178   True    True    True      True
179   True    True    True      True
180   True    True    True      True
181   True    True    True      True
182   True    True    True      True
183  False    True    True      True
184   True    True    True      True
185  False   False    True      True
186   True    True    True      True
187  False    True    True      True
188   True    True    True      True
189   True    True    True      True
190   True    True    True      True
191   True    True    True      True
192  False   False    True      True
193   True    True    True      True
[194 rows x 4 columns]
```

Crear un subsert mediante filtrado

Otra de las fortalezas de `pandas`, es la facilidad con la cual se puede realizar filtrado de datos. Por ejemplo, si se desea hacer un subset solo con los países de origen americano, esto se logra con la siguiente sintaxis:

```
df_americas = df[df['region'] == 'Americas']
```

Lo que se hace mediante esta sintaxis, es consultar a `pandas` por todas aquellas observaciones que contienen `'Americas'` en su campo `'region'`.

Como es una operación lógica, ésta entregará una lista de valores `True` o `False`. A partir de este punto, el `DataFrame` interpreta que seleccionará todos aquellos registros donde la condición evaluada sea verdadera.

Este nuevo objeto obtenido con lo que se ha seleccionado, seguirá siendo un `DataFrame`, y por ende tiene todos sus atributos:

```
# tamaño
df_americas.shape
```

```
(35, 13)
```

```
# el valor más alto del producto interno bruto
df_americas.gdp.max()
```

```
42895.0
```

```
# a qué país corresponde éste?
df_americas[df_americas['gdp'] == df_americas['gdp'].max()]['country']
```

```
84    United States
Name: country, dtype: object
```

Otro ejemplo:

Se desea extraer solo la información de la columna `gdp`, que representa el Producto Interno Bruto:

```
df['gdp']
```

```
0       7300.399902
1       1338.800049
2      12307.400391
3       1063.400024
4        349.200012
5       1986.800049
6       3052.199951
7        677.000000
8       1266.199951
9       1099.000000
10      3628.000000
11       279.799988
12      1539.199951
13      1972.199951
14      4754.399902
15     27645.800781
16       579.599976
17       741.400024
18     13183.200195
19      1218.400024
20      1303.000000
21       957.599976
22       967.599976
23      1405.400024
24      1284.400024
25       345.000000
26     14497.799805
27       919.799988
28       660.400024
29      1032.800049
           ...
164    36646.398438
165    48169.398438
166    15446.400391
167    21628.400391
168    10560.400391
```

```
169    13424.799805
170             NaN
171     9473.200195
172    18552.199219
173    25321.199219
174    27862.199219
175    33621.398438
176    37105.800781
177     6124.000000
178    33295.800781
179     4662.000000
180    33707.199219
181     4259.600098
182     2294.199951
183             NaN
184     2906.800049
185             NaN
186    25199.599609
187             NaN
188     1953.800049
189     4012.600098
190     2249.199951
191     4072.199951
192             NaN
193     3809.800049
Name: gdp, Length: 194, dtype: float64
```

Datos perdidos en una serie

Se observa que en la columna `gdp` se presentan algunos registros como `NaN`. Estos indican que son **datos perdidos**, y pueden ser algo conflictivo para análisis posteriores.

Se puede consultar por el porcentaje de datos perdidos en la columna de la siguiente forma:

```
# Para esto concatenaremos funciones. Partimos por preguntar si los valores son
# nulos o no, esto devolverá un booleano. Posteriormente con sum podemos
# contar la cantidad de ocurrencias

df['gdp'].isnull().sum()
```

```
15
```

Observamos que existen 15 observaciones nulas, porque la función "sum" considerará cada evaluación `True` como `1` y cada evaluación `False` como 0.

¿Cómo podemos averiguar de qué países son?

Para lograr esto, es posible **ingresar una operación lógica entre los corchetes**. En este caso, la operación lógica es de la siguiente forma `df['gdp'].isnull() == True`.

```
df_gdp_missing = df[df['gdp'].isnull() == True]
```

Formas de iteración en un DataFrame

Hasta el momento se ha visto cómo generar "subsets" de datos seleccionando tanto filas como columnas específicas. Pero algo que puede ser necesario para el flujo de trabajo, es iterar fila a fila y evaluar cada una para algún caso específico.

¿Cómo iterar las filas o registros del DataFrame?

Nuestra primera solución puede ser algo como:

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

```
country
region
gdp
school
adfert
chldmort
life
pop
urban
femlab
literacy
co2
gini
```

Esto no es satisfactorio, dado que queremos extraer las observaciones pero en su lugar se obtiene los nombres de las columnas.

En el caso de un `DataFrame`, existe más de una forma de poder recorrer los registros. Para implementar estos flujos, están los métodos df.iterrows(), df.iteritems() y df.itertuples():

df.iteritems()

Con `iteritems`, es posible recorrer todos los elementos de un `DataFrame` **por columna**. Esto significa que se va a estar recorriendo todos los campos de una base de datos a la vez. El método funciona de la siguiente manera:

```
for colname, serie in df.iteritems():
    print(colname)
```

```
country
region
gdp
school
adfert
chldmort
life
pop
urban
femlab
literacy
co2
gini
```

Es importante recalcar que `iteritems`, de similar manera a cuando se utilizaba `enumerate`, **necesita tener dos iterables declarados**. El primer iterable responde al nombre de la columna, mientras que el segundo devuelve los valores asociados a cada columna en forma de `Serie`. Este punto es importante, dado que es posible concatenar acciones que operen a nivel de serie dentro del loop.

Por ejemplo, inspeccionemos si todos los datos corresponden al tipo numérico o no. Para ello, se hará uso de la siguiente función de `pandas`: `pd.api.types.is_numeric_dtype`.

```
for colname, serie in df.iteritems():
    tmp = pd.api.types.is_numeric_dtype(serie)
    print("{} es {}".format(colname, tmp))
```

```
country es False
region es False
gdp es True
school es True
adfert es True
chldmort es True
life es True
pop es True
urban es True
femlab es True
literacy es True
co2 es True
gini es True
```

Observamos que en este ejemplo sólo country y region no son numéricas.

df.iterrrows()

De similar manera que con `iteritems`, con `iterrows` es posible recorrer todos los elementos de un `DataFrame` **por fila**. Esto significa que se va a estar recorriendo todas las columnas, por registro, de una base de datos a la vez.

Por motivos prácticos, para ver el efecto de los `iterrows`, vamos a seleccionar sólo una observación, incluyendo un `if`.

```
for index, row_serie in df.iterrows():
    if index == 1:
        print(row_serie)
        print(type(row_serie))
```

```
country       Benin
region       Africa
gdp          1338.8
school          3.1
adfert         111.7
chldmort      122.75
life         54.7333
pop          8237634
urban             41
femlab        0.8482
literacy        41.7
co2              1.2
gini             NaN
Name: 1, dtype: object
```

Observamos que lo que devuelve es el registro específico de Benin. Esto es análogo a consultar lo siguiente:

```
df[df['country'] == 'Benin']
```

Capítulo 7: Numpy

Competencias

- Importar la librería `numpy` para el procesamiento y cálculo de datos numéricos.
- Manipular los principales tipos de datos de `numpy`.
- Conocer e implementar las principales funciones vectorizadas de `numpy`.
- Integrar funciones de `numpy` a objetos `pandas`.

¿Qué es Numpy?

Numpy es un módulo o librería, que constituye una de las piedras angulares en el desarrollo de la computación numérica en Python. La mayoría de los módulos implementados hacen uso de los objetos de numpy.

Algunas de las características de numpy incluyen:

1. Listas n-dimensionales que facilitan el trabajo aritmético.

- Funciones matemáticas orientadas a la velocidad de implementación, sin la necesidad de utilizar loops.

- Capacidad de trabajar con álgebra lineal, generación de números pseudoaleatorios, funciones distributivas, entre otros.

Para entender la velocidad de `numpy`, mediremos el tiempo que se demora en ejecutar un loop entre una lista y un `ndarray`, que es su equivalente en `numpy` de una lista.

Primero se debe importar numpy siguiendo la siguiente convención

```
import numpy as np
```

```
# generamos un array de 1000000 observaciones con `np.arange`
n_sims = 1000000
demo_array = np.arange(n_sims)
# generamos un objeto range y posteriormente lo pasamos a formato lista
demo_list = list(range(n_sims))
```

```
# ndarray
%time
demo_array_2 = demo_array * 2
```

```
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 7.87 µs
```

```
%time
demo_list_2 = [x * 2 for x in demo_list]
```

```
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 6.91 µs
```

Listas de numpy (ndarray)

Numpy presenta un arreglo n-dimensional llamado `ndarray`. Este es un contenedor muy rápido y flexible para grandes cantidades de datos. Mediante estos, se pueden generar operaciones entre vectores de manera similar a como se realizan entre escalares.

Ejemplo

Generemos una matriz de $4X4$ de números aleatorios. Para ello utilizamos `np.random.randn()`.

```
datos_aleatorios = np.random.randn(4,4)
datos_aleatorios
```

```
array([[-0.59320874, -0.28681656,  0.44013481, -0.87396026],
       [-0.4234027 , -0.14286703, -0.54442265, -0.19114353],
       [ 0.95927452, -0.58262029, -0.44897912,  1.73131123],
       [ 0.88405659,  1.05930008, -0.06930013, -0.99025956]])
```

`ndarray` permite ejecutar operaciones aritméticas en bloques completos de datos de manera similar a como lo realizamos con escalares. Ahora vamos a multiplicar todos los elementos dentro de la matriz recién creada:

```
datos_multiplicados = datos_aleatorios * 3
datos_multiplicados
```

```
array([[-1.77962623, -0.86044967,  1.32040443, -2.62188078],
       [-1.27020811, -0.42860108, -1.63326795, -0.5734306 ],
       [ 2.87782355, -1.74786086, -1.34693736,  5.1939337 ],
       [ 2.65216977,  3.17790023, -0.20790038, -2.97077868]])
```

Ahora podemos comparar si las operaciones hacen sentido al evaluar la división de nuestros `datos_multiplicados` en 3, comparados a los datos aleatorios.

```
datos_aleatorios == (datos_multiplicados / 3)
```

```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True, False],
       [ True,  True,  True, False]])
```

Los `ndarray` son contenedores multidimensionales genéricos para datos homogéneos. Esto es muy relevante, porque para aprovechar la naturaleza del tipeo dinámico de Python, debemos asegurarnos de la homogeneidad de los elementos a ingresar. Mediante esta definición de datos homogéneos, es posible realizar esta serie de operaciones de manera fácil y rápida.

Todo `ndarray` tiene un `shape` asociado que se expresa mediante una tupla. La tupla nos entrega información sobre las dimensiones y tamaño de cada dimensión. También tiene un `dtype` que nos entrega información sobre los tipos de datos contenidos en el `ndarray`.

```
print(datos_aleatorios.shape)
print(datos_aleatorios.dtype)
```

```
(4, 4)
float64
```

Ejemplos de crear `ndarrays`

La manera más fácil es implementar los objetos en formato secuencial como las listas, dentro de la declaración `np.array()`. La función generará de forma automática la conversión a un objeto `ndarray`.

```
numero_accidentes = [6, 7, 3, 4, 12, 3, 4, 2, 3, 8, 2, 23, 35, 23, 12,
23]
accidentes_ndarray = np.array(numero_accidentes)
accidentes_ndarray
print(accidentes_ndarray.shape)
print(accidentes_ndarray.dtype)
```

```
(16,)
int64
```

Para los casos n-dimensionales, np.array reconocerá de forma automática el shape de la lista.

```
accidentes_por_dia = [list(range(1, len(numero_accidentes)  +1)),
                      numero_accidentes]
print(accidentes_por_dia)
```

```
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16], [6, 7, 3, 4,
12, 3, 4, 2, 3, 8, 2, 23, 35, 23, 12, 23]]
```

Numpy ofrece una serie de funciones para crear distintos tipos de `ndarray`.

- **`np.asarray`:** Convierte inputs sólo si éstos no son ndarray.
- **`np.arange`:** Genera un rango de manera similar a la implementación nativa de range(), devolviendo una lista en formato `ndarray`.
- **`np.ones`, `np.ones_like`:** Generan arrays poblados de 1 con las dimensiones especificadas. `np.ones_like` toma como referencia las dimensiones y tipo de dato de otro objeto.
- **`np.zeros`, `np.zeros_like`:** Generan arrays poblados de 0 con las dimensiones especificadas. np.zeros_like toma como referencia las dimensiones y tipo de dato de otro objeto.
- **`np.empty`, `np.empty_like`:** Generan arrais poblados mediante la asignación de memoria, pero no completan los arrays con elementos.
- **`np.full`, `np.full_like`:** Produce un array con determinadas dimensiones y tipo de dato, rellenadas con un valor determinado.

Indexing y Slicing en arrays vectoriales

Parte de las tareas más comunes es la indexación y creación de grupos. Para los arrays unidimensionales, las operaciones son similares a la indexación de listas en Python nativo. Para indexar un dato, simplemente se solicita a la lista que entregue el elemento guardado en la posición solicitada.

```
una_secuencia = np.arange(50)
una_secuencia[5]
```

```
5
```

Slicing hace referencia a la creación de subgrupos mediante la especificación de un rango inferior y superior de las posiciones de los elementos. Algunas de las operaciones que se pueden realizar mediante el operador Slice (`:`) son:

- Seleccionar subgrupos mediante un rango inferior y superior con `array[inf:sup]`.

```
# Al igual que en `range`, se considera hasta el índice anterior
# indicado como índice de término
una_secuencia[10:17]
```

```
array([10, 11, 12, 13, 14, 15, 16])
```

- Seleccionar subgrupos hasta un valor superior con `array[:sup]`.

```
# Selecciona todas las posiciones desde el comienzo hasta la posición 24
una_secuencia[:25]
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

- Seleccionar subgrupos desde un valor inferior con `array[inf:]`.

```
# Selecciona todas las posiciones desde la posición 5 hasta el final
una_secuencia[5:]
```

```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21,
       22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38,
       39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49])
```

Si se asigna un valor a un subgrupo seleccionado mediante slicing, esta se propagarán a la lista original desde donde proviene la subselección.

```
otra_secuencia = una_secuencia[10:18]
otra_secuencia[4] = 123456
print(otra_secuencia[4])
# El elemento de la posición 4 de subgrupo
# corresponde al elemento en la posición 14 en el arreglo original.
print(una_secuencia[14])
```

```
123456
123456
```

Arrays N-dimensionales

Vimos que con `pandas` es posible extraer un archivo `csv` ordenado en filas y columnas, y de éste generar una representación de la tabla en un `DataFrame`. Lamentablemente `numpy` no presenta esta facilidad, por lo que la representación de una matriz con filas y columnas se presenta como una *lista de listas*.

```
lista_de_listas = np.array([
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15]
])

lista_de_listas
```

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
```

Dentro del objeto `lista_de_listas`, es posible ingresar a una de las sub listas mediante `array[posicion]`.

```
lista_de_listas[2]
```

```
array([11, 12, 13, 14, 15])
```

También es posible solicitar un elemento en una sublista específica mediante `array[posicion_lista][posicion_elemento]`.

```
lista_de_listas[1][2]
```

```
8
```

Lo anterior también se puede lograr de la siguiente manera:

```
lista_de_listas[1, 2]
```

```
8
```

Para aquellos casos donde las listas contengan 3 o más dimensiones, los índices se evalúan de forma jerárquica desde mayor a menor.

```
lista_3d = np.array([
    [[3, 5, 7, 9],
     [8, 10, 12, 14],
     [5, 15, 20, 25]],
    [[6, 10, 14, 18],
     [16, 20, 24, 28],
     [10, 30, 40, 50]]
])
lista_3d
```

```
array([[[ 3,  5,  7,  9],
        [ 8, 10, 12, 14],
        [ 5, 15, 20, 25]],

       [[ 6, 10, 14, 18],
        [16, 20, 24, 28],
        [10, 30, 40, 50]]])
```

```
lista_3d[0]
```

```
array([[ 3,  5,  7,  9],
       [ 8, 10, 12, 14],
       [ 5, 15, 20, 25]])
```

```
lista_3d[0][0]
```

```
array([3, 5, 7, 9])
```

```
lista_3d[0][0][0]
```

```
3
```

Slice en arrays n-dimensionales

Las operaciones de slice en arrays n-dimensionales se realizan de manera similar a como se realizan con arrays unidimensionales. Tomemos nuestra `lista_de_listas` como ejemplo.

Si solicitamos los elementos indexados hasta 2, Python devolverá lo siguiente:

```
lista_de_listas[:2]
```

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

En este caso, el operador slice asumió por defecto que se ha solicitado sólo los dos primeros elementos en la primera dimensión. Así, se opera a nivel del primer contenedor.

El operador slice es muy flexbile, por lo que también es posible agregar más especificaciones. Ahora se solicitarán las dos primeras listas, pero ignorando sus primeros elementos.

```
lista_de_listas[:2, 1:]
```

```
array([[ 2,  3,  4,  5],
       [ 7,  8,  9, 10]])
```

Los operadores de slice se pueden utilizar en conjunto con índices simples y su resultado será el retorno de los elementos solicitados con menor dimensión.

```
lista_de_listas[1,:2]
```

```
array([6, 7])
```

```
lista_de_listas[0,2]
```

```
3
```

Indexación booleana

Una de las estrategias más comunes para generar subgrupos de un conjunto de datos es mediante operaciones booleanas. La flexibilidad de las expresiones booleanas permite clarificar y reducir la complejidad de los criterios de búsqueda.

Por ejemplo, supongamos que tenemos una serie de elementos asociados a una lista de nombres.

```
# generamos una lista unidimensional con nombres
nombres = np.array(['Gonzalo', 'Gonzalo', 'Gonzalo',
                    'Magdalena', 'David', 'Juan Pablo',
                    'Cristian', 'María José', 'Ignacio'])
```

```
# generamos una matriz de numeros aleatorios con 9 filas
# y 6 notas
datos = np.random.randint(1, high=7, size=54).reshape(9, 6)
```

Veamos la representación de la matriz.

```
datos
```

```
array([[5, 3, 3, 6, 4, 2],
       [6, 2, 5, 3, 3, 5],
       [2, 3, 4, 4, 6, 2],
       [1, 4, 6, 4, 1, 5],
       [6, 2, 4, 2, 4, 1],
       [1, 5, 1, 5, 5, 1],
       [2, 2, 4, 4, 1, 1],
       [2, 4, 3, 4, 6, 6],
       [1, 2, 1, 4, 6, 5]])
```

Podemos preguntar por la membresía de algún elemento específico en el array con el operador `==`.

```
'Gonzalo' == nombres
```

```
array([ True,  True,  True, False, False, False, False, False, False])
```

A diferencia del comportamiento de una lista nativa de Python, si preguntamos por algún elemento específico, numpy devolverá todas las evaluaciones para cada elemento.

```
nombres == 'Fernanda'
```

```
array([False, False, False, False, False, False, False, False, False])
```

Hasta el momento tenemos dos objetos separados, uno de nombres y otro de notas. Si queremos filtrar todas las notas de Magdalena, podemos implementar una indexación booleana dentro de las dimensiones de nuestra matriz datos.

```
datos[nombres == 'Magdalena']
```

```
array([[1, 4, 6, 4, 1, 5]])
```

También podemos definir operaciones por vía negativa. En este caso, deseamos excluir las observaciones que estén asociadas a David.

```
datos[~(nombres == 'David')]
```

```
array([[5, 4, 3, 2, 3, 1],
       [4, 2, 6, 3, 1, 6],
       [6, 1, 2, 6, 3, 1],
       [6, 1, 4, 6, 6, 4],
       [1, 5, 2, 6, 6, 1],
       [4, 5, 2, 3, 1, 5],
       [6, 6, 5, 2, 4, 4],
       [5, 5, 6, 5, 5, 3]])
```

También es posible sobreescribir todos los elementos que cumplan o no con una condición. En este caso, queremos reemplazar por cero todas aquellas observaciones menores o igual a 4.

```
datos[datos <= 4] =0
datos
```

```
array([[5, 4, 3, 2, 3, 0],
       [4, 2, 6, 3, 0, 6],
       [6, 0, 2, 6, 3, 0],
       [6, 0, 4, 6, 6, 4],
       [0, 3, 3, 2, 6, 6],
       [0, 5, 2, 6, 6, 0],
       [4, 5, 2, 3, 0, 5],
       [6, 6, 5, 2, 4, 4],
       [5, 5, 6, 5, 5, 3]])
```

Finalmente, si queremos extraer dos condiciones, es bueno privilegiar operadores lógicos **bitwise** como `&` o `|`.

```
datos[(nombres == 'Gonzalo') | (nombres == 'David')]
```

```
array([[5, 4, 3, 2, 3, 0],
       [4, 2, 6, 3, 0, 6],
       [6, 0, 2, 6, 3, 0],
       [0, 3, 3, 2, 6, 6]])
```

Transposicionamiento de Arrays

Transposicionar permite generar subindexaciones de los datos sin la necesidad de copiarlos.

```
# Se genera un arreglo unidimensional con 30 elementos
array_3 = np.arange(30)
# Se cambia las dimensiones del arreglo, para distribuir
# los mismos 30 elementos en una distribución de 5 filas y 6 columnas
array_3 = array_3.reshape((5, 6))
array_3
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29]])
```

La matriz creada tiene 5 filas y 6 columnas. Esto se expresa en notación algebraica con $N \times M$. Para este caso, array_3 = 5×6.

La transposición de una matriz significa pasar de las filas a las columnas y viceversa. Al aplicar el operador `T`, nuestra matriz pasa a ser la siguiente:

```
array_3.T
```

```
array([[ 0,  6, 12, 18, 24],
       [ 1,  7, 13, 19, 25],
       [ 2,  8, 14, 20, 26],
       [ 3,  9, 15, 21, 27],
       [ 4, 10, 16, 22, 28],
       [ 5, 11, 17, 23, 29]])
```

Funciones Universales

`numpy` provee una serie de funciones universales, también conocidas en la jerga como ufuncs. Éstas funciones buscan aplicar operaciones a todos los elementos contenidos en una estructura de datos `ndarray`.

```
array_4 = np.arange(20)
```

```
np.sqrt(array_4)
```

```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ,
       3.16227766, 3.31662479, 3.46410162, 3.60555128, 3.74165739,
       3.87298335, 4.        , 4.12310563, 4.24264069, 4.35889894])
```

```
np.exp(array_4)
```

```
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03, 2.20264658e+04, 5.98741417e+04,
       1.62754791e+05, 4.42413392e+05, 1.20260428e+06, 3.26901737e+06,
       8.88611052e+06, 2.41549528e+07, 6.56599691e+07, 1.78482301e+08])
```

Las funciones del ejemplo de arriba se conocen como funciones unarias, que reciben por defecto **un vector**.

También existen las funciones binarias, que como lo indica su nombre, reciben dos vectores y **devuelven un vector con los resultados de la operación**.

El mantra es siempre preferir trabajar con funciones orientadas al procesamiento de vectores por sobre loops. Mediante esta forma, es más fácil abstraerse de los detalles y enfocarse en los substanciales.

Operador ternario

`numpy` ofrece el operador ternario `np.where` que facilita la reconversión de elementos en base a si se cumple una condición booleana o no.

La sintaxis de `np.where` consta de tres partes:

```
np.where(<condicion>, <valor_positivo>, <valor_negativo>)
```

Donde `<condicion>` hace referencia a un criterio formalizado en alguna expresión booleana que entregue verdaderos o falsos. `<valor_positivo>` es el valor a asignar a aquellos elementos que satisfagan con la condición. `<valor_negativo>` es el valor a asignar a aquellos elementos que no satisfagan la condición.

Ejemplo de uso de np.where

```
# Se definen 3 listas para trabajar
altura_gatas = [3.60, 2.35, 2.97,
                3.79, 2.37, 2.56,3.49, 3.38, 4.70, 1.82]

altura_gatos = [4.20, 4.14, 3.41, 4.24, 4.96, 3.80,
 2.23, 2.69, 1.02, 4.67]

tabby = [True, False, False, True, False,
         True , True, False, False, False]
```

```
# Aquellos gatos cuya altura sea mayor a la media de la lista
(condición),
# serán gatos grandes, y se almacenan como True (valor positivo),
# y los que son menores se almacenan como False (valor negativo)
gatos_grandes = np.where(altura_gatos > np.mean(altura_gatos),
                         True, False)
gatos_grandes
```

```
array([ True,  True, False,  True,  True,  True, False, False, False,
        True])
```

```
# Aquellas gatas cuya altura sea menor a la media de la lista, Y que
además
# se encuentren en una posición correspondiente a True en la lista
"tabby" (condición),
# serán gatas pequeñas y se almacenan como True (valor positivo),
# y las que no lo son se almacenan como False (valor negativo)

gatas_pequenas_tabby = np.where(
    (altura_gatas < np.mean(altura_gatas)) & (tabby == True),
    True,
    False
)
```

```
gatas_pequenas_tabby
```

```
array([False, False, False, False, False, False, False, False, False,
       False])
```

`np.where` no está limitado a asignar booleanos; Los valores positivos y negativos asignados pueden ser números enteros, flotantes, cadenas, etc.

```
# generemos una matriz de números aleatorios
array_5 = np.random.randn(6, 6).round(3)
array_5
```

```
array([[ 0.154,  0.87 , -0.114,  0.333,  1.006, -0.938],
       [-0.794, -0.801, -0.719,  0.22 ,  0.895,  0.513],
       [-0.97 ,  1.034, -0.696,  2.582,  0.221,  0.705],
       [ 1.841,  0.843,  0.23 , -0.439,  0.568,  0.604],
       [ 0.329, -1.551,  1.16 , -1.5  , -0.775,  0.062],
       [ 0.971,  0.186, -0.303, -0.156, -0.051, -0.717]])
```

```
array_5 < 0
```

```
array([[False, False,  True, False, False,  True],
       [ True,  True,  True, False, False, False],
       [ True, False,  True, False, False, False],
       [False, False, False,  True, False, False],
       [False,  True, False,  True,  True, False],
       [False, False,  True,  True,  True,  True]])
```

```
# np.where permite asignar valores de forma automática.
# ahora deseamos redondear de forma bruta aquellos valores menores a
cero como 0
# y aquellos valores mayores a cero como 1
np.where(array_5 < 0, 0, 1)
```

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

```
#también podemos recodificar sólo aquellos valores menores a cero
# y no alterar los demás.
# al pasar array.
np.where(array_5 < 0 , 0, array_5)
```

```
array([[0.154, 0.87 , 0.   , 0.333, 1.006, 0.   ],
       [0.   , 0.   , 0.   , 0.22 , 0.895, 0.513],
       [0.   , 1.034, 0.   , 2.582, 0.221, 0.705],
       [1.841, 0.843, 0.23 , 0.   , 0.568, 0.604],
       [0.329, 0.   , 1.16 , 0.   , 0.   , 0.062],
       [0.971, 0.186, 0.   , 0.   , 0.   , 0.   ]])
```