Introducción

Python es un lenguaje de programación versátil, que se potencia enormemente para el cómputo científico con bibliotecas como **numpy**, **scipy** y **matplotlib**.

`Créditos`: This tutorial was originally written by Justin Johnson for cs231n. It was adapted as a Jupyter notebook for cs228 by Volodymyr Kuleshov and Isaac Caswell. This version has been adapted for Colab by Kevin Zakka for the Spring 2020 edition of cs231n.
Luego fue traducido y adaptado a Python 3.12.2 por la cátedra.

Este tutorial cubrirá:

- Fundamentos de Python:
  - Tipos de datos básicos
  - Contenedores:
    - Listas (*lists*)
    - Diccionarios (*dicts*)
    - Conjuntos (*sets*)
    - Tuplas (*tuples*)
  - Funciones (*functions*)
  - Clases (*classes*)
- Numpy:
  - Arreglos (*arrays*)
  - Indexación de arreglos
  - Tipos de datos
- Matplotlib:
  - Gráficos (*plots*)
  - Subgráficos (*subplots*)
  - Imágenes

Versión de Python

Tanto en esta unidad, como en las venideras en este curso, utilizaremos la versión de *Python* `3.12.2`, que fue lanzada el 6 de Febrero de 2024.

cambiar a 3.10.12

```
!python --version
```

```
Python 3.12.2
```

Fundamentos de Python

Python es un lenguaje de programación de alto nivel, dinámicamente tipado y multiparadigma. Su código es tan legible que a menudo se compara con el pseudocódigo, permitiendo expresar ideas complejas con pocas líneas.

Tipos de datos básicos

Números

Enteros (*int*) y flotantes (*float*) funcionan como los otros lenguajes:

```
x = 3
print(x, type(x))
```

```
3 <class 'int'>
```

```
print(x + 1)   # Addition
print(x - 1)   # Subtraction
print(x * 2)   # Multiplication
print(x ** 2)  # Exponentiation
```

```
4
2
6
9
```

```
x += 1
print(x)
x *= 2
print(x)
```

```
4
8
```

```
y = 2.5
print(type(y))
print(y, y + 1, y * 2, y ** 2)
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

```
a = 4 // 2
type(a)
print(a)
```

```
2
```

Python también incluye tipos incorporados para números complejos; todos los detalles están en la documentación.

Booleans

Python implementa todos los operadores habituales de lógica booleana usando palabras en inglés en lugar de símbolos (&&, ||, etc.).

```
t, f = True, False
print(type(t))
```

```
<class 'bool'>
```

Ahora miremos el resultado de las operaciones:

```
print(t and f) # Logical AND;
print(t or f)  # Logical OR;
print(not t)   # Logical NOT;
print(t != f)  # Logical XOR;
```

```
False
True
False
True
```

Strings

```
hello = 'hello'   # Strings pueden usar comillas simples
world = "world"   # o comillas dobles, da exactamente igual
print(hello, len(hello))
```

```
hello 5
```

```
hw = hello + ' ' + world  # String concatenation
print(hw)
```

```
hello world
```

```
string_1 = 'hola'
string_2 = 'mundo'
hw12 = f'{string_1} {string_2}'  # string formatting
print(hw12)
```

```
hola mundo
```

El tipo de dato `string` tiene varios métodos muy útiles, veamos algunos:

```
s = "hello"
print(s.capitalize())  # Capitalize a string (mayúscula en la primera letra)
print(s.upper())       # Convert a string to uppercase (todo el string en MAYUS)
print(s.rjust(7))      # Right-justify a string (completa con espacios)
print(s.center(7))     # Center a string (completa con espacios)
print(s.replace('l', '(ell)'))  # Reemplaza todas las 'l' con '(ell)'
print('  world '.strip())  # Remueve los espacios al inicio y al final del string
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

Se pueden encontrar todos los métodos para strings en la documentación.

Containers

Python incluye varios tipos de contenedores *built-in*: lists, dictionaries, sets, y tuples.

built-in significa que viene ya nativo en Python.

Lists

Una lista en Python es equivalente a un array, pero se puede redimensionar y contener elementos de diferentes tipos:

```
xs = [3, 1, 2]   # Crea la lista
print(xs, xs[2])
print(xs[-1])     # El índice negativo arranca a contar de atrás para adelante.
                  # Es decir, el -1 siempre es el último de la lista, sin importar el largo de la lista.
```

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

```
xs[2] = 'foo'    # Las listas pueden contener distintos tipos de datos en cada posición
print(xs)
```

```
[3, 1, 'foo']
```

```
xs.append('bar') # Se le pueden agregar elementos
print(xs)
```

```
[3, 1, 'foo', 'bar']

x = xs.pop()     # Se pueden "poppear" elementos: te saca el último elemento de la lista, y te lo devuelve
print(x, xs)

bar [3, 1, 'foo']
```

Como siempre, se pueden encontrar más detalles de las listas en la documentación.

Slicing

Además de acceder elementos de listas uno a uno, Python provee sintaxis para acceder a "sub-listas"; esto se conoce como **slicing**:

```
nums = list(range(5))    # range es una función built-in, crea listas de enteros
print(nums)         # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])    # slice desde el index 2 al 4 (exclusive)
print(nums[2:])     # slice desde el index 2 hasta el final
print(nums[:2])     # slice desde el the inicio hasta el index 2 (exclusive)
print(nums[:])      # slice de la lista entera
print(nums[:-1])    # Los indexes pueden ser negativos
nums[2:4] = [8, 9]  # Asigna valores al slice
print(nums)

[0, 1, 2, 3, 4]
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

Loops

Se puede iterar sobre los elementos de una lista de esta manera:

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)

cat
dog
monkey
```

Si necesitas acceder al índice de cada elemento dentro de un bucle, existe la función built-in `enumerate`.

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print(f'#{idx}: {animal}')

#0: cat
#1: dog
#2: monkey
```

Comprensiones de listas:

Al programar, frecuentemente queremos transformar un tipo de datos en otro. Como ejemplo simple, considera el siguiente código que calcula el cuadrado de los números:

```
nums = [0, 4, 2, 3, 5]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)

[0, 16, 4, 9, 25]
```

Podés simplificar este código usando una comprensión de lista (*list comprehension*):

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares, type(squares[0]))

[0, 1, 4, 9, 16] <class 'int'>
```

Y también le podés agregar condiciones (`if`):

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)

[0, 4, 16]
```

Diccionarios

Un diccionario almacena pares de (clave, valor).
Se puede usar de esta manera:

```
d = {'gato': 'lindo', 'perro': 'simpático', 0: 'hola'}  # Crear
diccionario con datos
print(d['gato'])      # Le pregunto al diccionario qué "value" hay en
el "key" gato
print('gato' in d)     # También puedo preguntar si el key "gato"
existe EN el diccionario "d"
print(d[0])
```

```
lindo
True
hola

d['pez'] = 'mojado'   # Setear un nuevo par (key, value) en el
diccionario.
                      # El key es "pez" y el value es "mojado"
print(d['pez'])

mojado

print(d['mono'])  # KeyError: 'mono' not a key of d

---------------------------------------------------------------------
-----
KeyError                                  Traceback (most recent call
last)
Cell In[42], line 1
----> 1 print(d['mono'])  # KeyError: 'mono' not a key of d

KeyError: 'mono'

print(d.get('mono', 'N/A'))  # Pedir un "value" con un valor default
por si no existe en el diccionario
print(d.get('pez', 'N/A'))

N/A
mojado

del d['pez']        # Remueve el elemento "pez"
print(d.get('pez', 'N/A'))

N/A
```

Nuevamente, toda la información sobre diccionarios, en la documentación.

Es fácil iterar sobre los "key" en un diccionario:

```
d = {'gato': 4, 'gallo': 2, 'pulpo': 8}
for animal in d:
    print(f'El {animal} tiene {d[animal]} patas')

El gato tiene 4 patas
El gallo tiene 2 patas
El pulpo tiene 8 patas

d = {'gato': 4, 'gallo': 2, 'pulpo': 8}
for animal, legs in d.items():
    print(f'El {animal} tiene {legs} patas')
```

```
El gato tiene 4 patas
El gallo tiene 2 patas
El pulpo tiene 8 patas
```

Dictionary comprehensions: Igual que las *list comprehensions*, pero para diccionarios

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)
```

```
{0: 0, 2: 4, 4: 16}
```

Sets

Una veriable del tipo `set` es una colección desordenada de distintos elementos.

```
animals = {'gato', 'perro', 'perro'}
print('gato' in animals)   # Checkear si el elemento 'gato' está EN el set animals
print('pez' in animals)
print(animals)
```

```
True
False
{'perro', 'gato'}
```

```
animals.add('pez')      # Agregar el elemento 'pez' al set
print('pez' in animals)
print(len(animals))     # Cantidad de elementos en el set
```

```
True
3
```

```
animals.add('gato')    # Agregar elementos que ya existen, no le hace nada
print(len(animals))
animals.remove('gato')  # También se pueden remover elementos
print(len(animals))
```

```
3
2
```

Bucles: Iterar sobre un conjunto tiene la misma sintaxis que iterar sobre una lista; sin embargo, dado que los conjuntos son desordenados, no se pueden hacer suposiciones sobre el orden en el que se recorren los elementos del conjunto.

```
animals = {'gato', 'perro', 'pez'}
for idx, animal in enumerate(animals):
    print(f'#{idx+1}: {animal}')
```

```
#1: perro
#2: pez
#3: gato
```

Set comprehensions: Al igual que las listas y diccionarios, podemos construir conjuntos fácilmente:

```
!python --version
```

```
Python 3.12.2
```

```
from math import sqrt
print({int(sqrt(x)) for x in range(30)})
```

```
{0, 1, 2, 3, 4, 5}
```

Tuples

Una tupla es una lista ordenada de valores **inmutable**. En muchos aspectos, es similar a una lista, pero recordemos que la lista sí se puede modificar.
Otra de las diferencias más importantes es que las tuplas pueden usarse como "keys" en diccionarios y como elementos de conjuntos, mientras que las listas no pueden.
Veamos el ejemplo:

```
d = {(x, x + 1): x for x in range(10)}  # Crear un diccionario
print(f'{d=}')
t = (5, 6)       # Crear una tupla
print(type(t))
print(d[t])
print(d[(1, 2)])
print(f'{d=}')
```

```
d={(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4, (5, 6): 5,
(6, 7): 6, (7, 8): 7, (8, 9): 8, (9, 10): 9}
<class 'tuple'>
5
1
d={(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4, (5, 6): 5,
(6, 7): 6, (7, 8): 7, (8, 9): 8, (9, 10): 9}
```

```
# Como vimos, una tupla es INMUTABLE
t[0] = 1
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
Cell In[60], line 2
      1 # Como vimos, una tupla es INMUTABLE
----> 2 t[0] = 1
```

```
TypeError: 'tuple' object does not support item assignment
```

Funciones

Las funciones de Python se definen usando: `def`

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'
for x in [-1, 0, 1]:
    print(sign(x))

negative
zero
positive
```

Para darle argumentos a la función, se hace de la siguiente manera:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, {}'.format(name.upper()))  # Esta es otra forma
de imprimir una variable dentro de un string
    else:
        print('Hello, {}!'.format(name))

hello('Bob')
hello('Fred', loud=True)

Hello, Bob!
HELLO, FRED
```

Classes

Ahora vamos a ver la sintaxis de una clase en Python, pero no vamos a ahondar mucho en esto:

```
class Greeter:

    # Constructor
    def __init__(self, name):
        self.name = name  # Creamos un atributo (variable) de la clase
"Greeter"

    # Instance method
    def greet(self, loud=False):
```

```
        if loud:
          print('HELLO, {}'.format(self.name.upper()))
        else:
          print('Hello, {}!'.format(self.name))

g = Greeter('Fred')  # Instanciamos un objeto del tipo "Greeter" y lo
almacenamos en la variable g
g.greet()            # Llamamos al método (función) "greet()" de la
clase "Greeter"
g.greet(loud=True)
```

```
Hello, Fred!
HELLO, FRED
```

Numpy

Numpy es la biblioteca central para el cómputo científico en Python.
Proporciona un objeto de arrays multidimensionales de alto rendimiento y herramientas para trabajar con estos arreglos.

Para usar el paquete Numpy, primero hay que importar el paquete `numpy`:

```
import numpy as np  # Ponemos el alias np
```

Arrays

Un *Numpy array* es una cuadrícula de valores, todos del mismo tipo, y se `indexa` por una tupla de enteros no negativos.
Estos *arrays* pueden ser de

$$n$$

dimensiones.
Por ejemplo: si es de dimensión

$$1 \times n$$

significa que es un vector de una sola fila y

$$n$$

columnas.
En cambio, si tiene dimensiones

$$n \times m$$

significa que tiene

$$n$$

filas y

m

columnas.
Pregunta: ¿Qué creen que significa "indexar" en este contexto?

Para saber las dimensiones de un *numpy array*, se puede utilizar el método `shape`.

```
a = np.array([1, 2, 3])  # Crear un array de una dimensión (1 x 3)
print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5                 # Le asignamos un 5 al primer elemento del
array
print(a)
print(a.shape)
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
(3,)
```

```
b = np.array([[1,2,3],[4,5,6]])   # Ahora creamos un array de 2
dimensiones (2 x 3)
print(b)
```

```
[[1 2 3]
 [4 5 6]]
```

```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
(2, 3)
1 2 4
```

Numpy también tiene otro tipo de funciones para crear arrays:

```
a = np.zeros((2,2))  # Crea un array lleno de ceros
print(a)
```

```
[[0. 0.]
 [0. 0.]]
```

```
b = np.ones((1,2))   # Crea un array lleno de unos
print(b)
```

```
[[1. 1.]]
```

```
c = np.full((2,2), 7) # Crea un array con el mismo número en todas las
posiciones
print(c)
```

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

```
d = np.eye(2)        # Crea una matriz identidad
print(d)
```

```
[[1. 0.]
 [0. 1.]]
```

```
e = np.random.random((2,2)) # Crea una matriz con valores random en
cada posición
print(e)
```

```
[[0.83809452 0.15758383]
 [0.1431143  0.16375155]]
```

Array indexing

Numpy nos ofrece varios tipos de indexar los arrays:

Slicing: Similar a las listas de Python, los arreglos de Numpy pueden ser "rebanados" (traducción muy directa de sliced). Dado que los arreglos pueden ser multidimensionales, hay que especificar un slicing para cada dimensión del arreglo.

```
import numpy as np
d
# Crear un numpy array con shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Ahora usamos "slicing" para sacar el sub-array que queremos:
# - Quiero las primeras dos filas (filas 0 y 1)
# - Quiero las columnas del medio (columnas 1 y 2)
# - De qué shape me tiene que quedar el resultado entonces?

b = a[:2,3 1:]
print(b)
```

```
[[2 3]
 [6 7]]
```

```
c = a[:2, :]  # Qué significaban los dos puntos sin ningú número?
print(c)
```

```
[[1 2 3 4]
 [5 6 7 8]]
```

```
print(a[0, 1])
b[0, 0] = 77    # b[0, 0] es el mismo dato que a[0, 1]
print(a[0, 1])
```

```
77
77
```

También se puede combinar la indexación por enteros con la indexación por slicing:

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

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

```
# Recordemos qué pasaba en una lista con el último índice del slice:
my_list = [1, 2, 3, 4]
my_list[:3]
```

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

```
row_r1 = a[1, :]
row_r2 = a[1:2, :]
row_r3 = a[[1], :]
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
print(row_r3, row_r3.shape)
row_r3 = a[[1], [1]]
print(row_r3, row_r3.shape)
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[[5 6 7 8]] (1, 4)
[6] (1,)
```

```
# Lo mismo se puede hacer para las columnas:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
print()
print(col_r2, col_r2.shape)
```

```
[ 2  6 10] (3,)

[[ 2]
 [ 6]
 [10]] (3, 1)
```

Indexación de arrays por enteros: Cuando indexamos en arrays de Numpy usando slicing, el arreglo resultante siempre será un subarreglo del arreglo original. Y también nos permite armar arrays con valores arbitrarios del array más grande:

```
a = np.array([[1,2], [3, 4], [5, 6]])

print(a[[0, 1, 2], [0, 1, 0]])  # Qué está haciendo acá?
```

```
[1 4 5]
```

```
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

```
[1 4 5]
```

```
# Como vimos, reutiliza de nuevo el mismo índice, ya sea la misma fila o la misma columna
print(a[[0, 0], [1, 1]])

print(np.array([a[0, 1], a[0, 1]]))
```

```
[2 2]
[2 2]
```

Un truco útil con la indexación de arrays por enteros es seleccionar un elemento de cada fila de una matriz:

```
# Creamos la matriz de la que vamos a elegir los elementos
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
```

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

```
# Creamos el array con los índices que vamos a seleccionar
b = np.array([0, 2, 0, 1])

# Seleccionamos un elemnto de cada fila utilizando los índices que tenemos en "b"
print(a[np.arange(4), b])
```

```
[ 1  6  7 11]
```

```
# Así como lo seleccionamos, también podemos usarlo para modificarlo
a[np.arange(4), b] += 10
print(a)
```

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

La indexación de arreglos booleanos te permite seleccionar elementos arbitrarios de un arreglo. Este tipo de indexación se utiliza para seleccionar los elementos de un arreglo que satisfacen alguna condición.

```
a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)  # Encuentra los elementos de "a" que son más
grandes que 2
                    # Esto devuelve un numpy array de booleanos del
mismo shape que "a"
                    # donde cada posición de "bool_idx" nos dice si el
elemento de "a"
                    # de la misma posición es mayor que 2 o no.

print(bool_idx)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

```
# Ahora podemos hacer esto:
print(a[bool_idx])
```

```
[3 4 5 6]
```

```
# O también esto para hacer todo de una vez:
print(a[a > 2])
```

```
[3 4 5 6]
```

Para resumir, hemos omitido muchos detalles sobre la indexación de arreglos de Numpy; si deseas saber más, deberías leer la documentación.

Datatypes

Cada array de Numpy es una grilla de elementos del mismo tipo.
Numpy te ofrece un montón de tipos de datos numéricos para que puedas armar tus arrays. Numpy trata de adivinar el tipo de dato cuando creás un array, pero las funciones para crear arrays suelen incluir también un argumento opcional para que especifiques bien clarito el tipo de dato.

Acá te dejo un ejemplo:

```
x = np.array([1, 2])
y = np.array([1.0, 2.0])
z = np.array([1, 2], dtype=np.int64)  # Acá forzamos un datatype
particular

print(x.dtype, y.dtype, z.dtype)
```

```
int64 float64 int64
```

Podés leer todo sobre numpy datatypes en la documentación.

Array math

Las funciones matemáticas básicas operan elemento por elemento en los arrays, y están disponibles tanto como sobrecargas de operadores como en forma de funciones en el módulo de Numpy.

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

```
# Elementwise difference
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

```
# Elementwise product
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
# Elementwise division
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

```
# Elementwise square root (raíz cuadrada)
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

```
v = np.array([9, 10])
w = np.array([11, 12])

# Y esto es si no queremos hacer el "elementwise", si no el producto vectorial
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

Se puede usar `@` que es lo mismo que el `dot` (producto punto).

```
print(v @ w)
```

```
219
```

```
# Matrix / vector product
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

```
[29. 67.]
[29. 67.]
[29. 67.]
```

```
# Matrix / matrix product
print(x.dot(y))
print(np.dot(x, y))
print(x @ y)
```

```
[[19. 22.]
 [43. 50.]]
[[19. 22.]
 [43. 50.]]
[[19. 22.]
 [43. 50.]]
```

Numpy ofrece muchas funciones útiles para realizar cálculos en arrays; una de las más útiles es `sum`:

```
x = np.array([[1,2],[3,4]])

print(np.sum(x))
print(np.sum(x, axis=0))
print(np.sum(x, axis=1))
```

```
10
[4 6]
[3 7]
```

La lista de todas las funciones que tiene numpy las podés ver en la documentación.

```
print(x)
print("transpose\n", x.T)
```

```
[[1 2]
 [3 4]]
transpose
 [[1 3]
 [2 4]]
```

```
v = np.array([[1,2,3]])
print(v )
print("transpose\n", v.T)
```

```
[[1 2 3]]
transpose
 [[1]
 [2]
 [3]]
```

Broadcasting

El broadcasting en NumPy permite operar arreglos de diferentes formas. Bajo ciertas condiciones, el arreglo más pequeño se "difunde" sobre el más grande para que tengan formas compatibles. Esto permite vectorizar operaciones evitando bucles en Python y sin copiar datos innecesariamente, logrando así algoritmos eficientes. Sin embargo, en algunos casos puede llevar a un uso ineficiente de memoria y ralentizar el cálculo.

Las operaciones en NumPy generalmente se realizan elemento por elemento entre pares de arreglos que deben tener la misma forma.

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a * b
```

La regla de broadcasting de NumPy relaja esta restricción cuando las formas de los arreglos cumplen ciertas condiciones. El ejemplo más simple de broadcasting ocurre cuando un arreglo y un valor escalar se combinan en una operación:

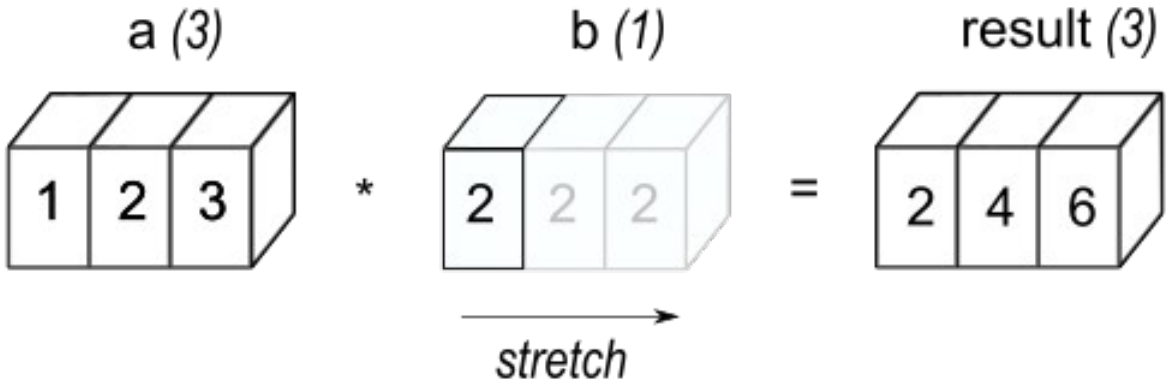
```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

```
array([2., 4., 6.])
```

El resultado es equivalente al ejemplo anterior donde **b** era un arreglo. Podemos pensar en el escalar **b** siendo extendido durante la operación aritmética a un arreglo con la misma forma que **a**.

Los nuevos elementos en **b** son simplemente copias del escalar original.

La analogía de la extensión es solo conceptual. NumPy es lo suficientemente inteligente como para usar el valor escalar original sin hacer copias reales, haciendo que las operaciones de broadcasting sean tan eficientes en memoria y cálculo como sea posible.



El segundo ejemplo, cuando se hizo el broadcasting del 2 en lugar de repetirlo 3 veces, es el más eficiente porque en memoria existe un solo escalar en lugar de un array.

Reglas para el broadcasting

Al operar con dos arreglos, NumPy compara sus formas elemento por elemento. Comienza con la dimensión final (es decir, la más a la derecha) y avanza hacia la izquierda.

Dos dimensiones son compatibles cuando:

- son iguales, o
- una de ellas es 1.

Si no se cumplen estas condiciones, se lanza una excepción ValueError: operands could not be broadcast together, indicando que los arreglos tienen formas incompatibles.

Los arreglos de entrada no necesitan tener el mismo número de dimensiones.

El arreglo resultante tendrá el mismo número de dimensiones que el arreglo de entrada con el mayor número de dimensiones, donde el tamaño de cada dimensión es el tamaño más grande de la dimensión correspondiente entre los arreglos de entrada.

Las dimensiones faltantes se asumen de tamaño uno.

Por ejemplo, si hay un array de 256x256x3 de valores RGB, y se quiere escalar cada color en la imagen por un valor diferente, se puede multiplicar la imagen por un arreglo unidimensional con 3 valores.

Alinear los tamaños de los ejes finales de estos arreglos según las reglas de broadcasting muestra que son compatibles:

```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
Result (3d array): 256 x 256 x 3
```

Cuando alguna de las dimensiones comparadas es uno, se utiliza la otra. En otras palabras, las dimensiones de tamaño 1 se estiran o "copian" para coincidir con la otra.

En el siguiente ejemplo, tanto los arreglos **A** como **B** tienen ejes con longitud uno que se expanden a un tamaño mayor durante la operación de broadcast:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

Un conjunto de arreglos se considera "broadcastable" a la misma forma si las reglas anteriores producen un resultado válido.

Por ejemplo, si:

- *a.shape* es (5,1)

- *b.shape* es (1,6)
- *c.shape* es (6,)
- *d.shape* es ()

De modo que *d* es un escalar, entonces *a*, *b*, *c* y *d* son todos transmisibles a la dimensión *(5,6)*; y

- a actúa como un arreglo (5,6) donde a[:,0] se transmite a las otras columnas,
- b actúa como un arreglo (5,6) donde b[0,:] se transmite a las otras filas,
- c actúa como un arreglo (1,6) y, por lo tanto, como un arreglo (5,6) donde c[:] se transmite a cada fila, y finalmente,
- d actúa como un arreglo (5,6) donde el valor único se repite.

Matplotlib

```

```

Este resumen cubrió varios puntos importantes que necesitás saber sobre Numpy, pero está lejos de ser completo. Chequeá la referencia de Numpy para descubrir mucho más sobre Numpy.

Matplotlib es una biblioteca para hacer gráficos. En esta sección, vemos una introducción muy breve al módulo `matplotlib.pyplot`.

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

Esto es algo para poder ver los gráficos acá mismo:

```
%matplotlib inline
```

Plotting

La función más importante en matplotlib es plot, que te permite graficar datos en 2D. Aquí hay un ejemplo simple:

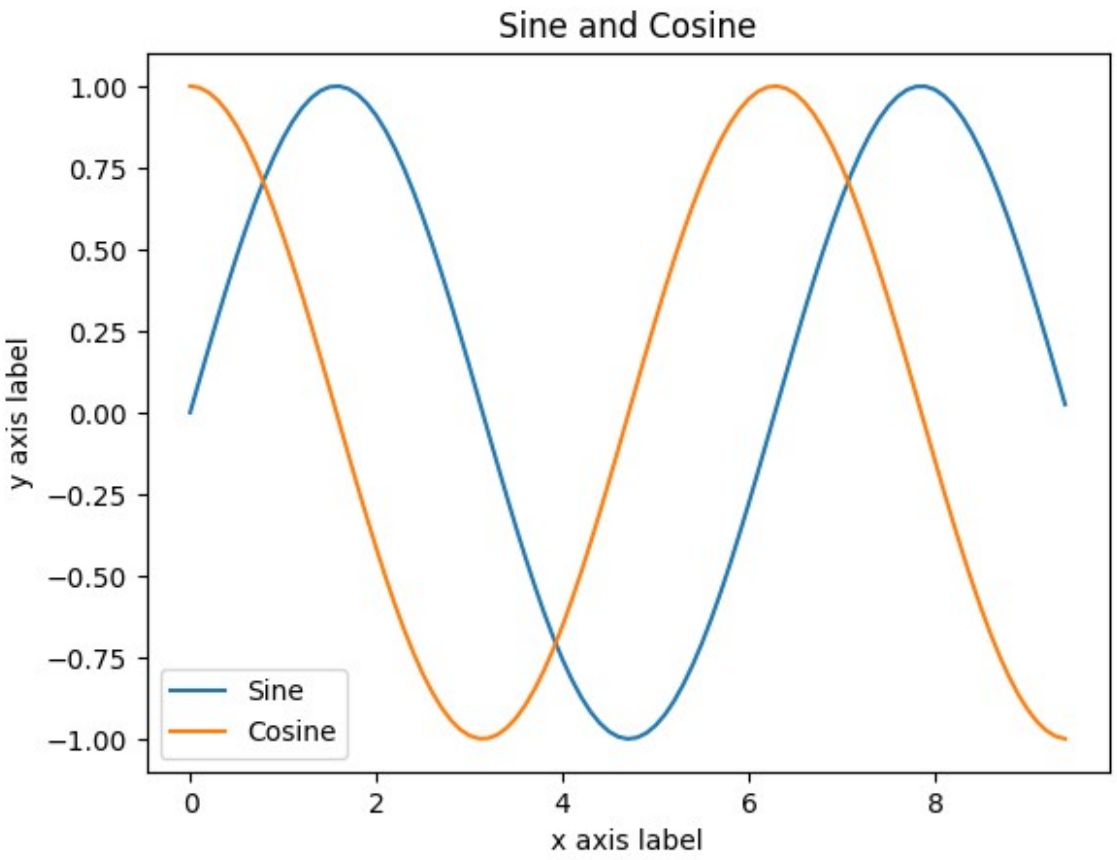
```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 2
      1 # Compute the x and y coordinates for points on a sine curve
----> 2 x = np.arange(0, 3 * np.pi, 0.1)
      3 y = np.sin(x)
      5 # Plot the points using matplotlib

NameError: name 'np' is not defined
```

Con solo un poco más de esfuerzo, podemos graficar múltiples líneas a la vez, y agregar un título, leyenda y etiquetas de ejes:

```
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
```

```
<matplotlib.legend.Legend at 0x7f413c37b350>
```

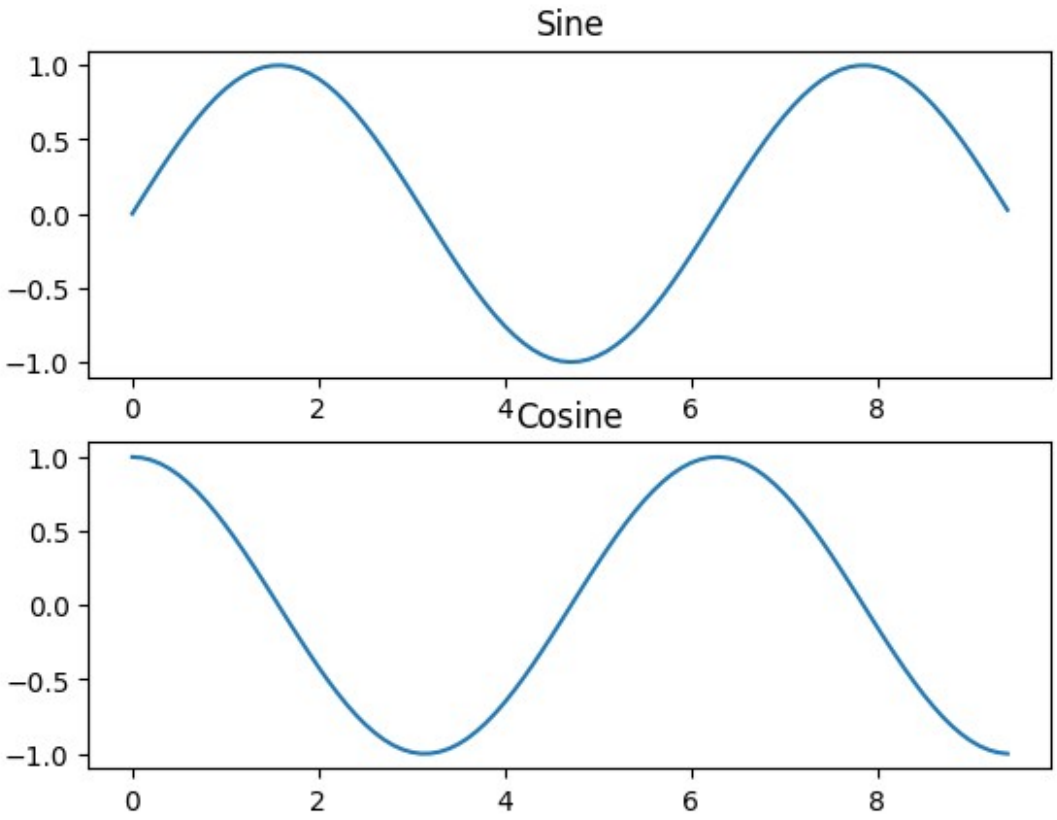


Subplots

Podés graficar distintas cosas en la misma figura usando la función `subplot`. Aquí hay un ejemplo:

```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')
```

```
# Show the figure.
plt.show()
```



Se puede encontrar más información de `subplot` en la documentación.

Ejercicios

Ahora vamos con algunos ejercicios para afianzar estos conceptos de Python que estuvimos viendo

Ejercicio 1

- Pedirle la edad al usuario y responder si es divisible por 3

Ejercicio 2

- Preguntar fecha de nacimiento y decir en qué estación nació (primavera, verano, otoño, invierno)

Ejercicio 3

- Contar cuántas veces aparece la letra "a" en el *string* almacenado en la variable "letras"

```
letras =
"aofnmgfoiajgmipoafjgmvaporgjmaeporgvmeñfvjkaeñvknmarvnampvma{rvkameñr
vkmaevlkaekvamlvnalvkaerjvakpvna{eñrvnaerv"
```

Ejercicio 4

- Almacenar la palabra "pato" en una variable llamada *palabra*
- Invertir la palabra e imprimirla

Ejercicio 5

Escribir un programa que pida al usuario dos números enteros (n y m) y muestre en pantalla: "La división entera entre **n** y **m** da un cociente **c** y un resto **r**" Donde **n** y **m** son los números introducidos por el usuario, y **c** y **r** son el cociente y el resto de la división entera respectivamente.

Ejercicio 6

Escribir un programa que almacene la cadena de caracteres contraseña en una variable, pregunte al usuario por la contraseña hasta que introduzca la contraseña correcta.

Ejercicio 7

Escribir un programa que permita gestionar la base de datos de clientes de una empresa. Los clientes se guardarán en un diccionario en el que el *key* de cada cliente será su CUIT, y el valor será otro diccionario con los datos del cliente (nombre, dirección, teléfono, correo, preferente), donde preferente tendrá el valor True si se trata de un cliente preferente.
El programa debe preguntar al usuario por una opción del siguiente menú: (1) Añadir cliente, (2) Eliminar cliente, (3) Mostrar cliente, (4) Listar todos los clientes, (5) Listar clientes preferentes, (6) Terminar. En función de la opción elegida el programa tendrá que hacer lo siguiente:

- Preguntar los datos del cliente, crear un diccionario con los datos y añadirlo a la base de datos.
- Preguntar por el CUIT del cliente y eliminar sus datos de la base de datos.
- Preguntar por el CUIT del cliente y mostrar sus datos.
- Mostrar lista de todos los clientes de la base datos con su CUIT y nombre.
- Mostrar la lista de clientes preferentes de la base de datos con su CUIT y nombre.
- Terminar el programa.

Ejercicio 8

Crear un numpy array con numeros enteros del 10 al 50

Ejercicio 9

Crear una matriz 3x3 con valores del 0 al 8

Ejercicio 10

Crear una matriz de identidad de 3x3

Ejercicio 11

- Generar una matriz "mat" de 4x4 con datos random
- Obtener la suma de los valores de mat
- Obtener la desviación estándar de los valores de mat
- Obtener las sumas de las columnas de mat