Introducción al lenguaje Python 2° clase

Técnicas de aprendizaje automático - Machine Learning

Repaso de la primera clase

Tipos de datos:

- Float
- Enteros
- Complejos
- Booleanos

- Strings

Estructuras de datos:

- Listas
- Tuplas
- Diccionarios

```
a = 4
type(a)
```
[136] ✓ 0.0s

```
int
```

```
b = 5.4
type(b)
```
[137] ✓ 0.0s

```
float
```

```
c = 4+3j
type(c)
```
[140] ✓ 0.0s

```
complex
```

```
d= "Python Course"
type(d)
```
[141] ✓ 0.0s

```
str
```

```
e = [4, -4, 1, 'prueba', 9.4, '80']
type(e)
```
[142] ✓ 0.0s

```
list
```

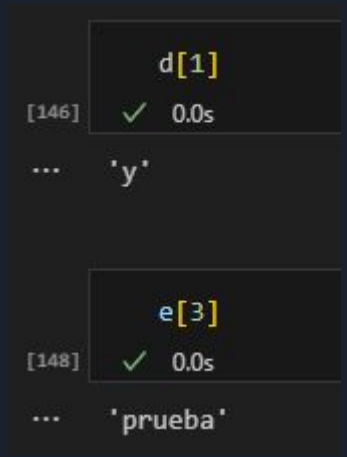
```
f = (3, 9, 'machine learning', 13.2, '8')
type(f)
```
[143] ✓ 0.0s

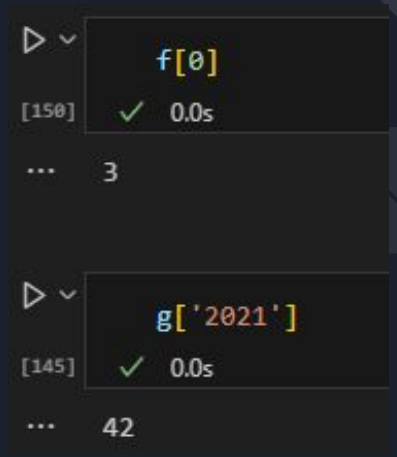
```
tuple
```

```
g = {'2020': 38, '2021': 42, '2022': 94, '2023':84}
type(g)
```
[144] ✓ 0.0s

```
dict
```

Los string, las listas, tuplas y diccionarios son estructuras en las cuales los datos ocupan lugares específicos (son iterables).





Funciones

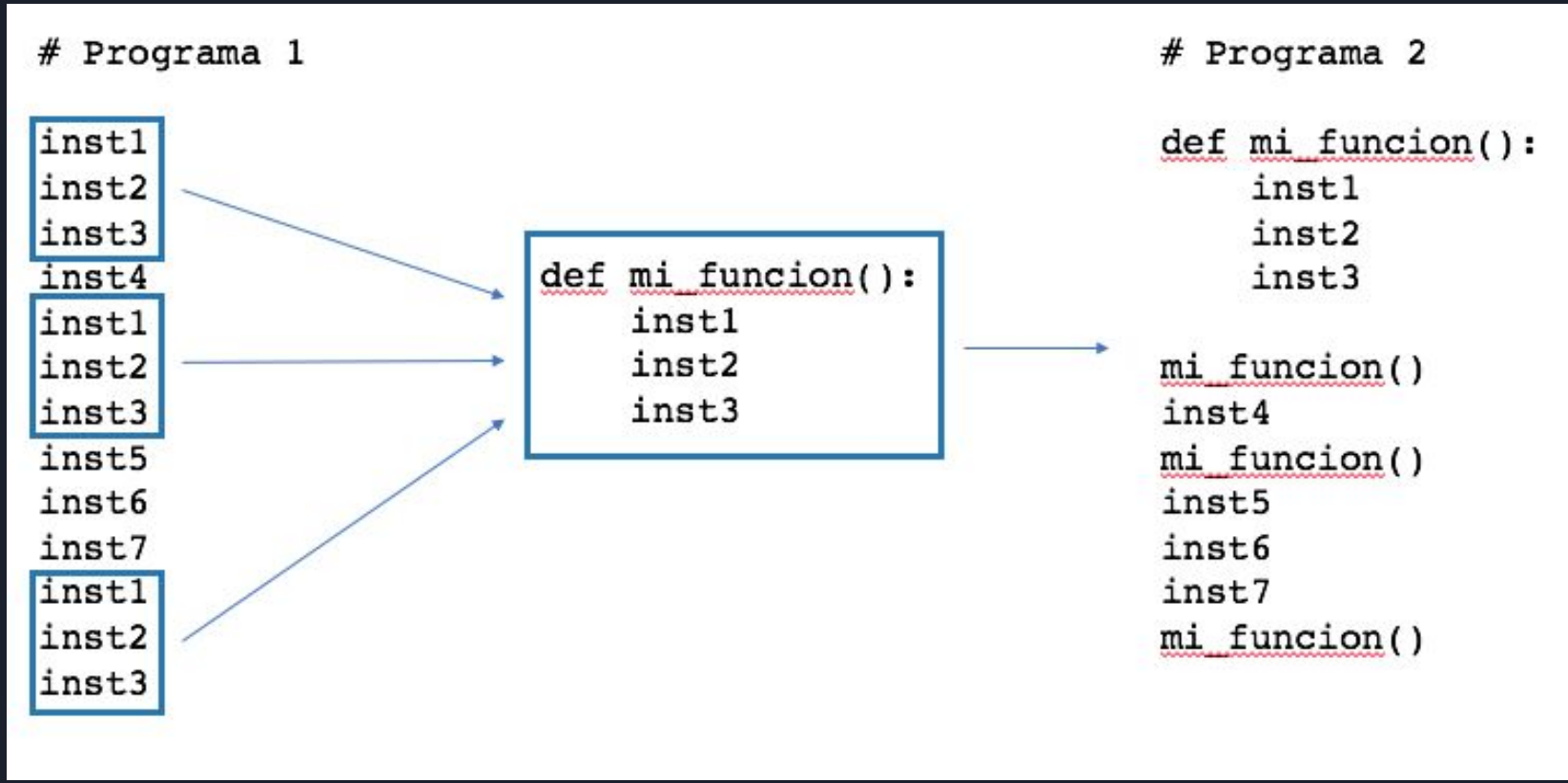
En Python, se llama función a un conjunto de líneas de código o instrucciones, que puede reutilizarse varias veces dentro de un script mayor. Su utilidad radica en que permite que los script sean más cortos y más eficientes.

Un usuario puede definir varias funciones específicas para un determinado proyecto, o usar funciones definidas en otros códigos.

La función puede tomar un número finito de parámetros de entrada, y se puede especificar el tipo de salida que la misma devolverá.

Primero es necesario definir la función, para luego poder llamarla.

Funciones

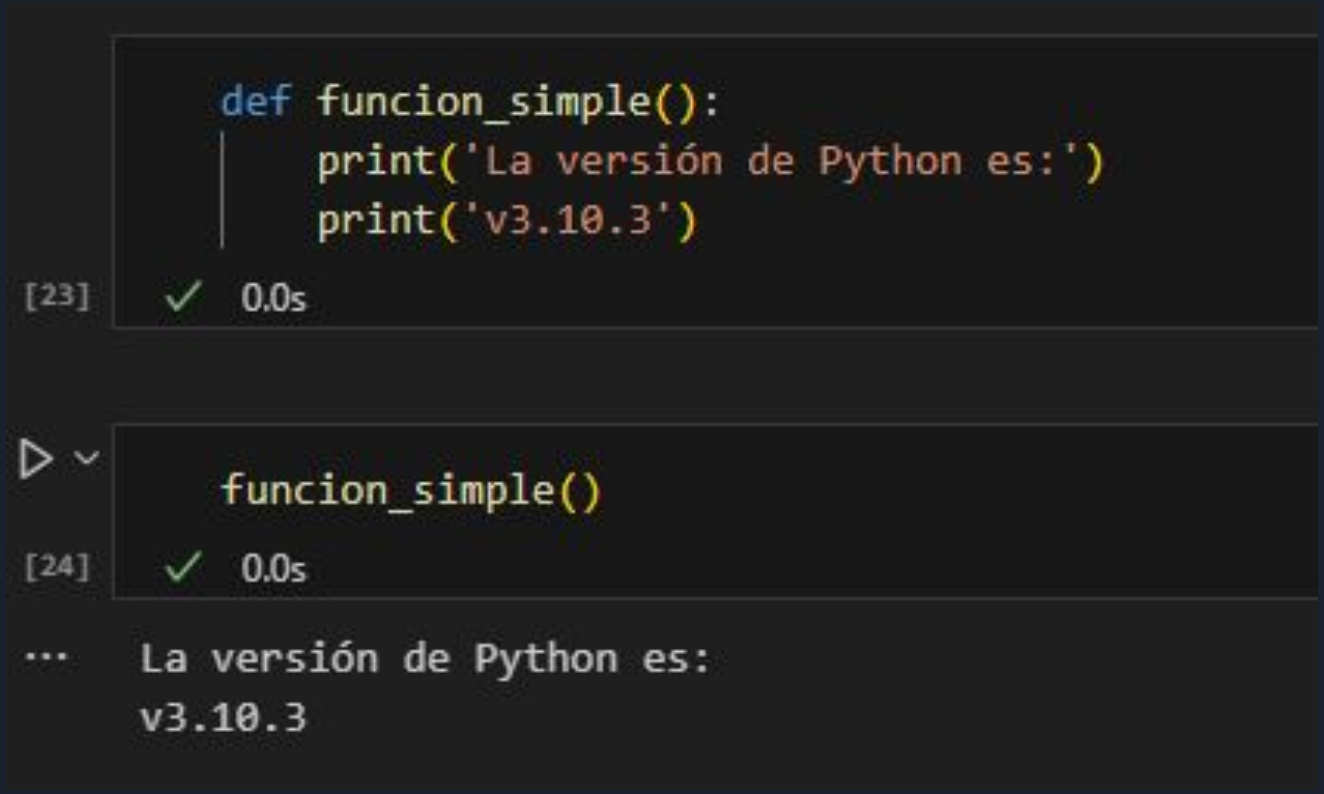

Programa 1
inst1
inst2
inst3
inst4
inst1
inst2
inst3
inst5
inst6
inst7
inst1
inst2
inst3
def mi_funcion():
inst1
inst2
inst3
# Programa 2
def mi_funcion():
inst1
inst2
inst3
mi_funcion()
inst4
mi_funcion()
inst5
inst6
inst7
mi_funcion()

Funciones: sintaxis

```
def <nombre_de_la_funcion>():
    #Linea de código a ejecutar 1
    #Linea de código a ejecutar 2
    ....
```

Esta es una función que no contiene ningún parámetro de entrada.

Funciones: sintaxis



Funciones: sintaxis

```
def <nombre_de_la_funcion>(param1, param2):
    #Línea de código a ejecutar 1
    #Línea de código a ejecutar 2
    ...
```

Esta es una función que contiene dos parámetros de entrada. En la definición de mi función yo debo especificar qué quiero hacer con mis parámetros.

Funciones: sintaxis

Una función tiene una forma determinada, y se debe respetar el orden en que se ingresa cada parámetro.

El nombre de los parámetros en la definición de una función es meramente simbólico, sólo determina qué es lo que la función debe hacer con cada cosa.

```
def presentacion(nombre, edad, facultad):
    print(f'Mi nombre es {nombre}.')
    print(f'Tengo {edad} años.')
    print(f'Estudié en {facultad}')

presentacion('Sebastian', '28', 'Facena')
```

```
[2]  ✓ 0.0s

...  Mi nombre es Sebastian.
     Tengo 28 años.
     Estudié en Facena
```

Funciones: sintaxis

Otra opción es especificar de forma explícita el valor de cada parámetro.

```
presentacion(facultad='Facena', nombre='Sebastian', edad='28')
```

[5] ✓ 0.0s

```
Mi nombre es Sebastian.
Tengo 28 años.
Estudié en Facena
```

Funciones: sintaxis

```
def sumar_y_multiplicar(a,b):
    suma = a + b
    producto = a*b
    print(f'La suma de los valores es {suma} y la multiplicación es {producto}.')
```

[30] ✓ 0.0s

```
sumar_y_multiplicar(4,10)
```

[31] ✓ 0.0s

```
La suma de los valores es 14 y la multiplicación es 40.
```

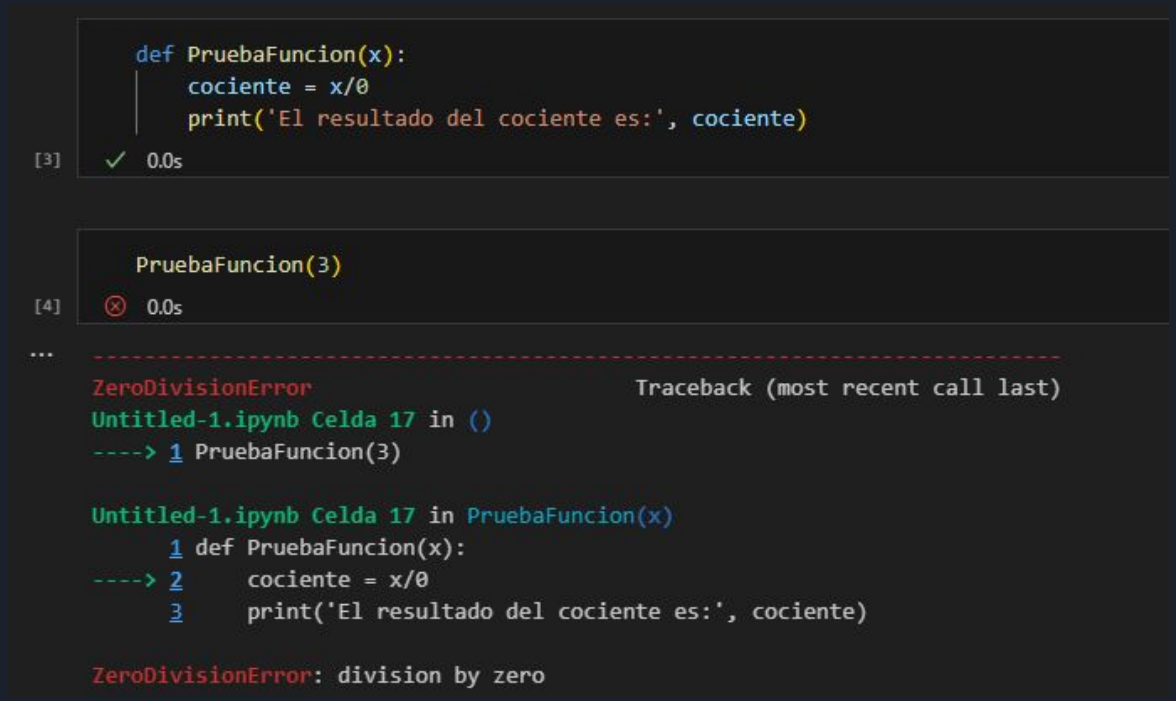
```
sumar_y_multiplicar(8,7)
```

[32] ✓ 0.0s

```
La suma de los valores es 15 y la multiplicación es 56.
```

Funciones: sintaxis

Al definir una función, solamente establecemos los pasos a seguir por el algoritmo. Si hay algún error, se detectará cuando se llame a la función.



```
def PruebaFuncion(x):
    cociente = x/0
    print('El resultado del cociente es:', cociente)
```

[3] ✓ 0.0s

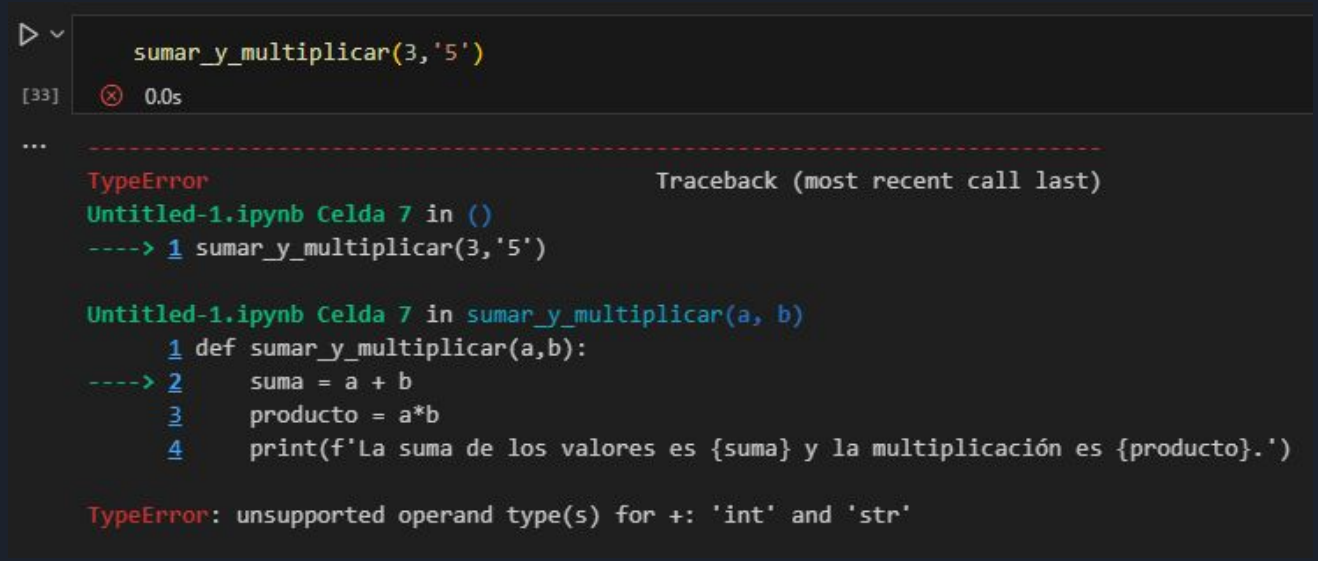
```
PruebaFuncion(3)
```

[4] ⊗ 0.0s

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Untitled-1.ipynb Celda 17 in ()
----> 1 PruebaFuncion(3)

Untitled-1.ipynb Celda 17 in PruebaFuncion(x)
      1 def PruebaFuncion(x):
----> 2     cociente = x/0
      3     print('El resultado del cociente es:', cociente)

ZeroDivisionError: division by zero
```

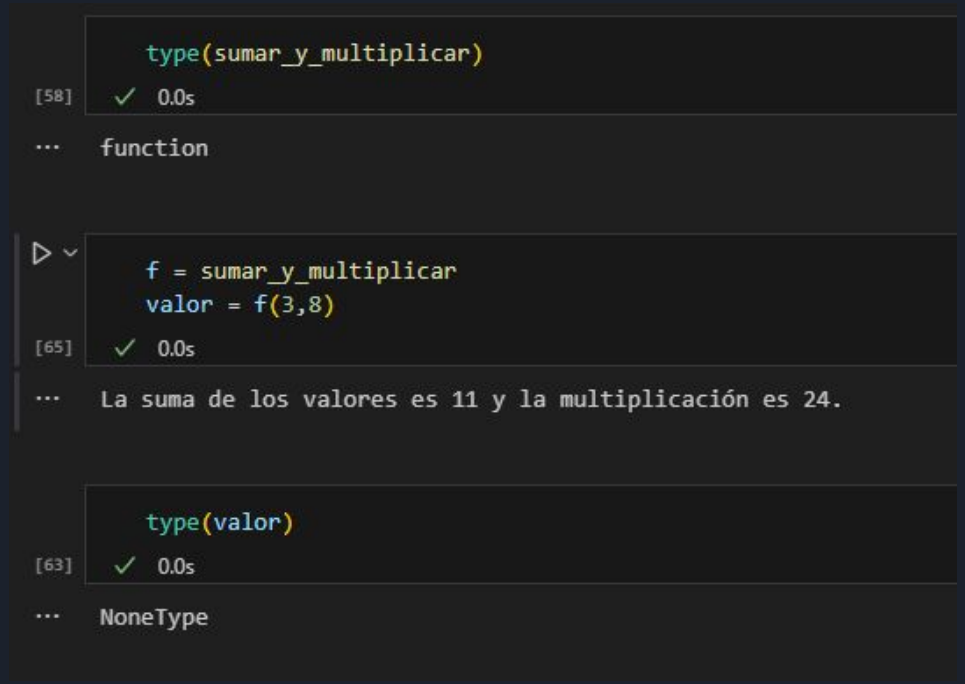
Funciones: errores

¿Qué pasa si la función no puede ejecutar el código definido con los parámetros que ingresé?



Funciones

Una función es un tipo de variable (un objeto) en sí.



Funciones

Más allá de su utilización como “memoria” de líneas de código, una forma muy útil de utilizar funciones en Python es la de permitirme generar comandos o métodos propios. Para ello se hace uso de la sentencia return.

La misma especifica qué es lo que devuelve la función al llamarla,es decir, la salida.

```
def <nombre_de_la_funcion>(param1, param2):

    #Líneas de código a ejecutar

    return algo_en_particular
```

Funciones: ejemplo con return

```
def RaizEcuacionLineal(m,b):
    #Sea una ecuación de la forma y=m*x+b
    #Defino una función que me calcule el valor de x que hace y=0
    #Tengo cuidado con el orden que ingreso los parámetros
    # "m" es la pendiente, "b" la ordenada al orígen
    raiz = -b/m
    return raiz
```

[46] ✓ 0.0s

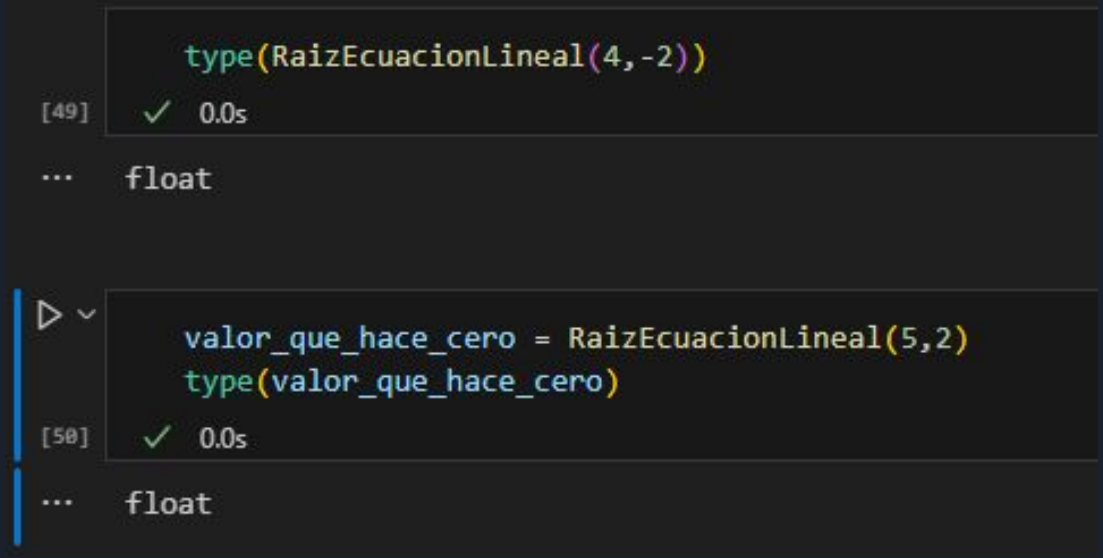
```
RaizEcuacionLineal(4,-2)
```

[48] ✓ 0.0s

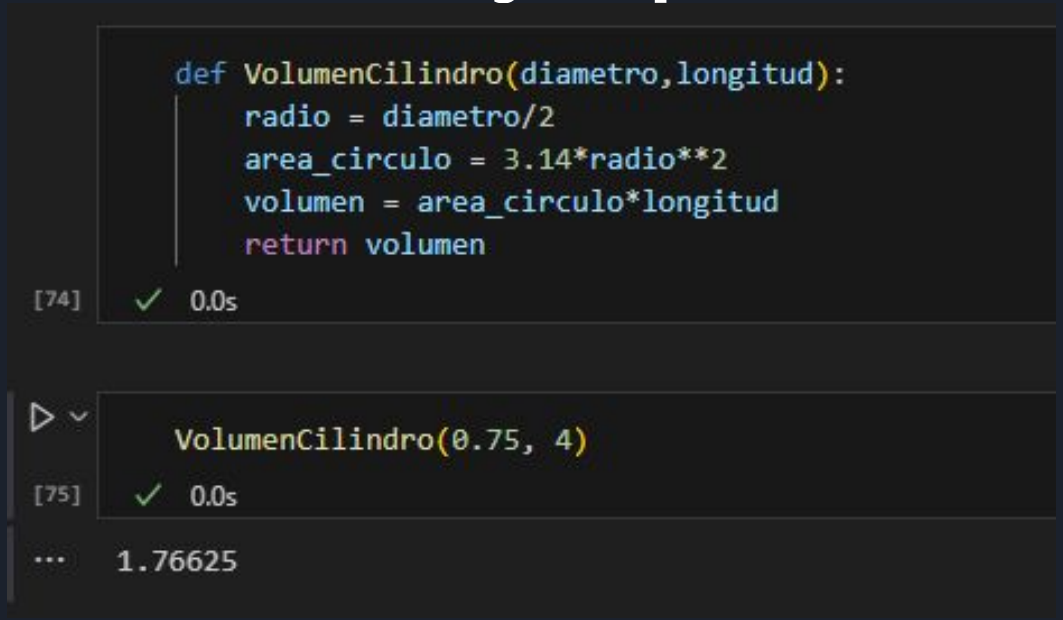
```
0.5
```

Funciones

En este caso, la función devuelve un resultado que es del mismo tipo de la variable especificada. Esto es útil cuando quiero implementar un comando que resulte en un tipo de salida específico.



Funciones: otro ejemplo con return



Incluso si dentro de mi función realizo varios cálculos, la misma sólo devolverá lo que especifica la sentencia return.

Funciones: valores por defecto

Puedo darle a los parámetros de la función valores predeterminados.

```
def FuerzaGravedad(masa, g=9.8):
    #Masa en kg
    #G en m/s**2
    return masa*g
```

[7] ✓ 0.0s

```
FuerzaGravedad(80)
```

[8] ✓ 0.0s

```
784.0
```

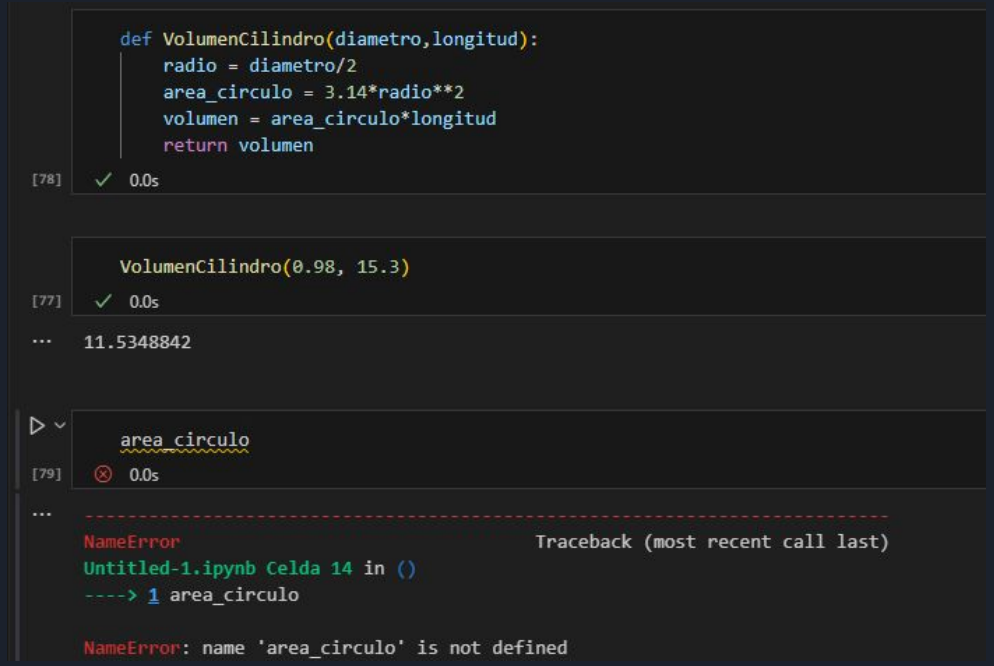
```
FuerzaGravedad(80, 1.625)
```

[9] ✓ 0.0s

```
130.0
```

Variables globales y locales

Una variable definida dentro de una función se considera una variable local, ya que no existe por fuera de la ejecución de la función (no puede ser vista por el programa principal).



Variables globales y locales

Una variable global es la que se define por fuera de una función. En principio, una función no cambia el valor de una variable global.

```
area_triangulo = 5

def ejemplo():
    area_triangulo = 4
    print(f'El area del triangulo es {area_triangulo} metros cuadrados.')

ejemplo()
print(area_triangulo)
```

[84] ✓ 0.0s

```
El area del triangulo es 4 metros cuadrados
5
```

Variables globales y locales

Sin embargo, puedo especificar en el cuerpo de la función que quiero trabajar con una determinada variable global.

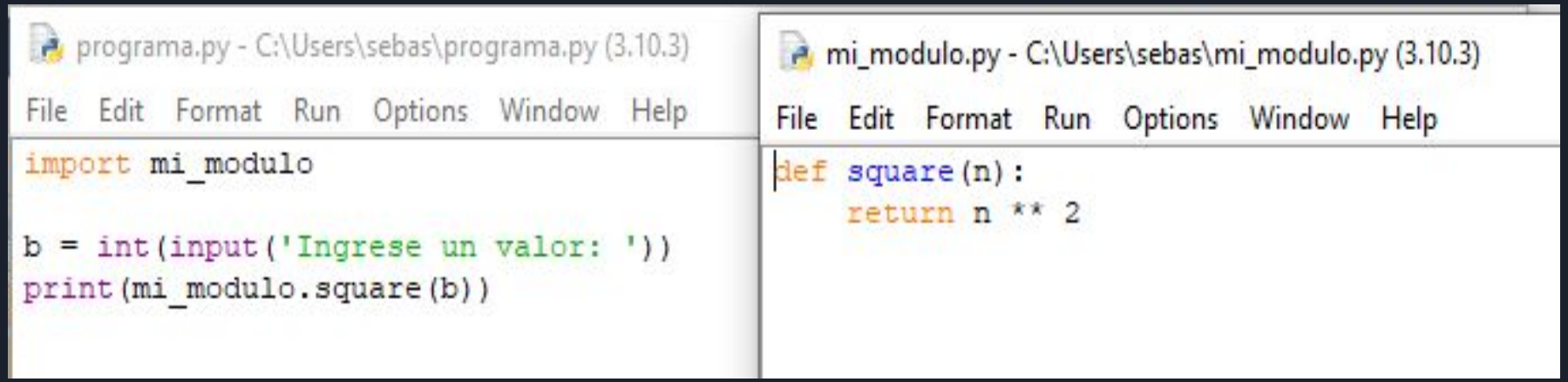
```
area_triangulo = 5

def ejemplo():
    global area_triangulo
    area_triangulo = 4
    print(f'El area del triangulo es {area_triangulo} metros cuadrados.')

ejemplo()
print(area_triangulo)
```

[85] ✓ 0.0s

```
El area del triangulo es 4 metros cuadrados.
4
```

Importar funciones

Puedo elegir trabajar con un archivo donde tengo un script y otro archivo donde tengo definidas funciones.



Importar funciones

Importar funciones

Otra forma:

programa.py - C:\Users\sebas\programa.py (3.10.3)

File Edit Format Run Options Window Help

```
from mi_modulo import square

b = int(input('Ingrese un valor: '))
print(square(b))
```

mi_modulo.py - C:\Users\sebas\mi_modulo.py (3.10.3)

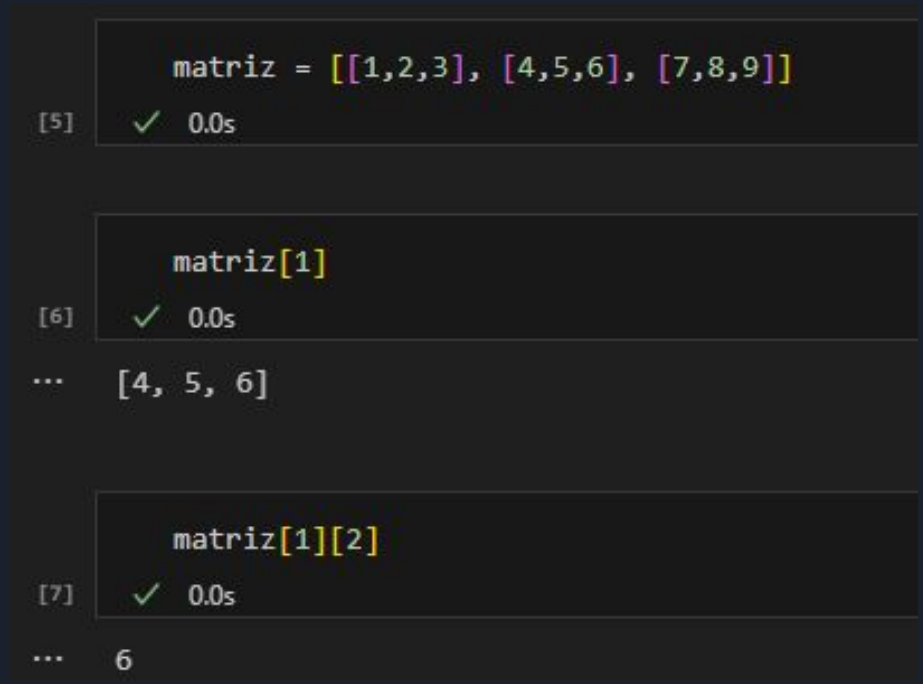
File Edit Format Run Options Window Help

```
def square(n):
    return n ** 2
```

Vectores y matrices

En Python, el elemento más semejante a un vector de datos es una lista.

Una lista puede contener como elementos otras listas, por lo que una matriz de datos puede armarse como una lista de listas.



Vectores y matrices

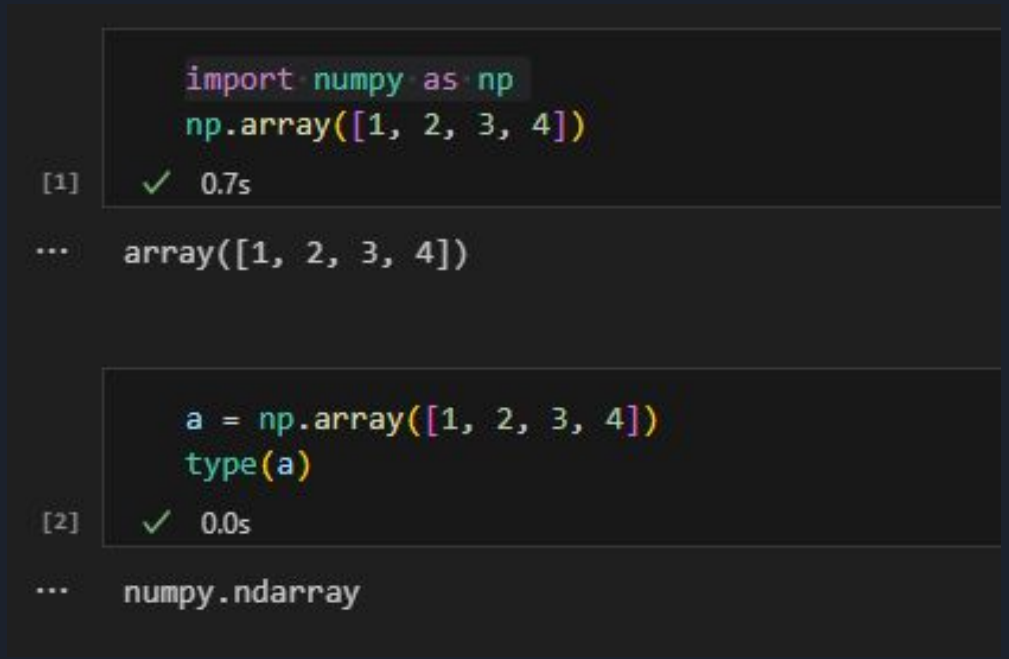
Sin embargo, existe una forma más eficiente de trabajar con estructuras de datos numéricos multi-dimensionales: la librería NumPy.

NumPy es una de las librerías más importantes para el cómputo numérico en Python. Trae incorporadas una gran variedad de funciones, facilita realizar operaciones vectorizadas y permite mejorar el tiempo de ejecución de los programas.

La estructura básica que utiliza NumPy es al **array n-dimensional** (ndarray).

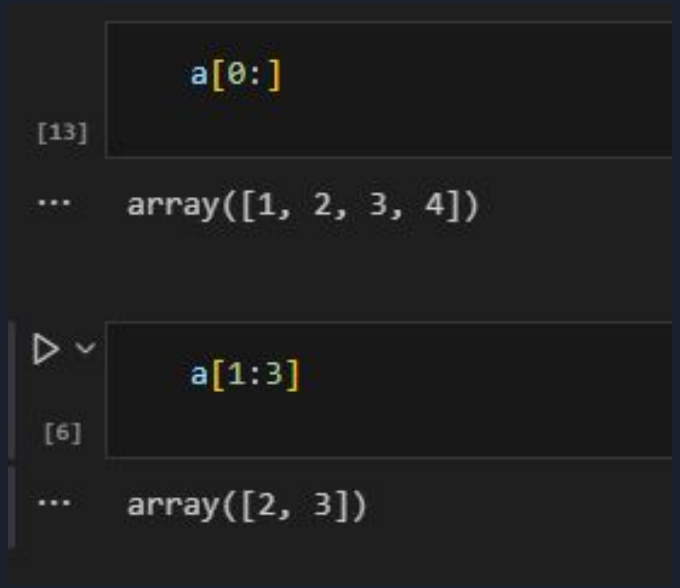
NumPy

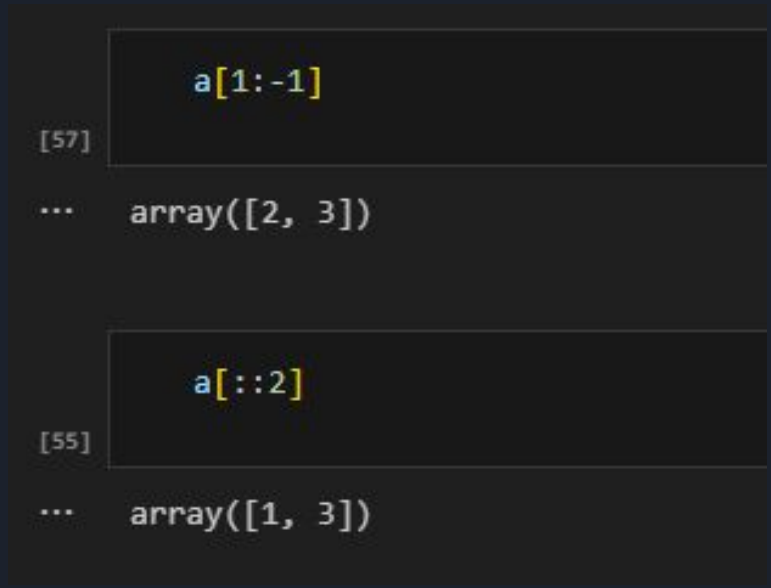
A diferencia de las listas, un array de NumPy está formado por elementos de un mismo tipo.



NumPy: slicing

El slicing se hace de manera similar a las listas y tuplas.





NumPy: tipos de datos

Dependiendo de los valores contenidos en el arreglo, NumPy le asigna un tipo de dato acorde.

```
[3]  a = np.array([1, 2, 3, 4])
     a.dtype
     ✓ 0.0s
...  dtype('int32')

[4]  b = np.array([0, 0.25, 0.5, 0.75, 1])
     b.dtype
     ✓ 0.0s
...  dtype('float64')
```

NumPy: tipos de datos

Sin embargo, también podemos determinar el tamaño de los datos.

```
[8]  np.array([1, 2, 3, 4], dtype=float)
     ✓ 0.0s
...  array([1., 2., 3., 4.])

[9]  c = np.array([1, 2, 3, 4], dtype=float)
     c.dtype
     ✓ 0.0s
...  dtype('float64')

[10] d = np.array([1, 2, 3, 4], dtype=np.int8)
     d.dtype
     ✓ 0.0s
...  dtype('int8')
```

NumPy: tipos de datos

- np.int8: (1 byte) - Para enteros entre -128 y 127.
- np.int16: (2 bytes) - Para enteros entre -32768 y 32767.
- np.int32: (4 bytes) - Para enteros entre -2147483648 y 2147483647.
- np.int64: (8 bytes) - Para números enteros entre -9223372036854775808 y 9223372036854775807.

```
    np.array([126, 127, 128, 129], dtype=np.int8)
[13] ✓ 0.0s
...  array([ 126,  127, -128, -127], dtype=int8)
```

NumPy: rendimiento

Podemos comparar el tiempo que tarda en sumarse los elementos de una lista de 10 millones de enteros, con lo que tarda en hacerse lo mismo con un array de NumPy.

```
array_as_list = list(range(int(10e6)))
array_as_ndarray = np.array(array_as_list)
```

```
%timeit sum(array_as_list)
#48 ms ± 203 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
172 ms ± 2.53 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit array_as_ndarray.sum()
#3.83 ms ± 4.84 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
4.02 ms ± 120 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

NumPy: formas y dimensiones

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

```
A.shape
```

```
(2, 3)
```

```
A.ndim
```

```
2
```

```
A.size
```

```
6
```

NumPy: formas y dimensiones

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

```
B
```

```
array([[[12, 11, 10],
        [ 9,  8,  7]],

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

```
B.shape
```

```
(2, 2, 3)
```

```
B.ndim
```

```
3
```

```
B.size
```

```
12
```

NumPy: formas y dimensiones

```
C = np.array([
    [
        [12, 11, 10],
        [9, 8, 7],
    ],
    [
        [6, 5, 4]
    ]
])
```

```
C.dtype
```

```
dtype('O')
```

```
C.shape
```

```
(2,)
```

```
C.size
```

```
2
```

```
type(C[0])
```

```
list
```

NumPy: selección

```
# Square matrix
A = np.array([
#.   0. 1. 2
    [1, 2, 3], # 0
    [4, 5, 6], # 1
    [7, 8, 9]  # 2
])
```

```
[85] A[1]
··· array([4, 5, 6])

[86] A[1][0]
··· 4

[87] A[1, 0]
··· 4

[88] A[0:2]
··· array([[1, 2, 3],
           [4, 5, 6]])
```

```
[90] A[:, :2]
··· array([[1, 2],
           [4, 5],
           [7, 8]])

[91] A[:2, :2]
··· array([[1, 2],
           [4, 5]])

[92] A[:2, 2:]
··· array([[3],
           [6]])
```

NumPy: selección

```
# Square matrix
A = np.array([
#.   0. 1. 2
    [1, 2, 3], # 0
    [4, 5, 6], # 1
    [7, 8, 9]  # 2
])
```

```
A[1] = np.array([10, 10, 10])
```

```
A
```

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

```
A[2] = 99
```

```
A
```

```
array([[ 1,  2,  3],
       [10, 10, 10],
       [99, 99, 99]])
```

NumPy: estadística

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

```
a.sum()
```

```
10
```

```
a.mean()
```

```
2.5
```

```
a.std()
```

```
1.118033988749895
```

```
a.var()
```

```
1.25
```

NumPy: estadística

```
A = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
```

```
A.sum()
```

```
45
```

```
A.sum(axis=0)
```

```
array([12, 15, 18])
```

```
A.mean()
```

```
5.0
```

```
A.sum(axis=1)
```

```
array([ 6, 15, 24])
```

```
A.std()
```

```
2.581988897471611
```

```
A.mean(axis=0)
✓ 0.0s
```

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

```
A.mean(axis=1)
✓ 0.0s
```

```
array([2., 5., 8.])
```

```
A.std(axis=0)
✓ 0.0s
```

```
array([2.44948974, 2.44948974, 2.44948974])
```

```
A.std(axis=1)
✓ 0.0s
```

```
array([0.81649658, 0.81649658, 0.81649658])
```

NumPy: operaciones vectorizadas

```
a = np.arange(4)
```
✓ 0.0s

```
a
```
✓ 0.0s

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

```
a + 10
```
✓ 0.0s

```
array([10, 11, 12, 13])
```

```
a * 10
```
✓ 0.0s

```
array([ 0, 10, 20, 30])
```

```
a
```
✓ 0.0s

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

```
a += 100
```
✓ 0.0s

```
a
```
✓ 0.0s

```
array([100, 101, 102, 103])
```

NumPy: operaciones vectorizadas

```
a
```

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

```
b
```

```
array([10, 10, 10, 10])
```

```
[123]  a + b
```

```
...    array([10, 11, 12, 13])
```

```
[124]  a * b
```

```
...    array([ 0, 10, 20, 30])
```

NumPy: funciones booleanas

```
a = np.arange(4)
```

```
a
```

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

```
a[[True, False, False, True]]
```

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

```
a >= 2
```

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

```
a[a >= 2]
```

```
array([2, 3])
```

NumPy: funciones booleanas

```
a.mean()
```

```
1.5
```

```
a[a > a.mean()]
```

```
array([2, 3])
```

```
a[~(a > a.mean())]
```

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

```
a[(a == 0) | (a == 1)]
```

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

```
a[(a <= 2) & (a % 2 == 0)]
```

```
array([0, 2])
```

NumPy: funciones booleanas

```
A = np.random.randint(100, size=(3, 3))
```

```
A
```

```
array([[27, 10, 14],
       [52, 11, 24],
       [32, 19, 37]])
```

```
A[np.array([
    [True, False, True],
    [False, True, False],
    [True, False, True]
])]
```

```
array([27, 14, 11, 32, 37])
```

```
A > 30
```

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

```
A[A > 30]
```

```
array([52, 32, 37])
```

NumPy: álgebra lineal

```
A = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
```

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

```
A.dot(B)
```

```
array([[20, 14],
       [56, 41],
       [92, 68]])
```

```
A @ B
```

```
array([[20, 14],
       [56, 41],
       [92, 68]])
```

```
[147]  B.T
```

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

```
[149]  B.T @ A
```

```
...  array([[36, 48, 60],
            [24, 33, 42]])
```

NumPy: álgebra lineal

```
matriz = np.array([
    [5, 2],
    [10, 4]
])
```

[32] ✓ 0.0s

```
np.linalg.det(matriz)
```

[33] ✓ 0.0s

```
0.0
```

```
matriz = np.array([
    [8, 1],
    [5, 2]
])
```

[34] ✓ 0.0s

```
np.linalg.det(matriz)
```

[35] ✓ 0.0s

```
10.999999999999996
```

NumPy: números aleatorios

random

```
[26]  np.random.random(size=2)
      #Valores en el rango [0,1)
```

✓ 0.0s

```
array([0.43135885, 0.77707062])
```

```
[30]  np.random.normal(1, 0.5, size=(2,2))
      #(media, desv standard, tamaño)
```

✓ 0.0s

```
array([[1.90234491, 0.96761374],
       [1.15418869, 1.58922878]])
```

```
[35]  np.random.rand(2, 4)
      #Valores en el rango[0,1) con distribucion uniforme
```

```
array([[0.39262813, 0.29642559, 0.21552866, 0.33818229],
       [0.29691797, 0.43996496, 0.22383564, 0.79252537]])
```

NumPy: rangos

```
### arange

[164]  np.arange(10)

...    array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

[165]  np.arange(5, 10)

...    array([5, 6, 7, 8, 9])

[166]  np.arange(0, 1, .1)

...    array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

NumPy: linspace

```
### linspace
```

```
[58]
np.linspace(0, 1, 5)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
[59]
np.linspace(0, 1, 20)
```

```
array([0.        , 0.05263158, 0.10526316, 0.15789474, 0.21052632,
       0.26315789, 0.31578947, 0.36842105, 0.42105263, 0.47368421,
       0.52631579, 0.57894737, 0.63157895, 0.68421053, 0.73684211,
       0.78947368, 0.84210526, 0.89473684, 0.94736842, 1.        ])
```

```
[60]
np.linspace(0, 1, 20, False)
```

```
array([0.  , 0.05, 0.1 , 0.15, 0.2 , 0.25, 0.3 , 0.35, 0.4 , 0.45, 0.5 ,
       0.55, 0.6 , 0.65, 0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95])
```

NumPy: zeros, ones

```
[61]  np.zeros(5)

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

```
[62]  np.zeros((3, 3))

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

```
[64]  np.ones(5)

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

```
[65]  np.ones((3, 3))

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

NumPy: matriz identidad

```
[69]  np.identity(3)

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

```
[70]  np.eye(3, 3)

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

```
[71]  np.eye(8, 4)

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

```
[72]  np.eye(8, 4, k=1)

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

```
[73]  np.eye(8, 4, k=-3)

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

Clases y objetos

Podemos definir un objeto como una entidad que tiene un conjunto de parámetros que lo caracterizan. Una cadena de caracteres o una lista es un ejemplo de objeto. De hecho, **en Python todo es un objeto**.

Las clases son los “prototipos” de información que definen los atributos o propiedades que se le pueden asignar a cada objeto.

Un objeto es una instancia o caso particular de una determinada clase. De la misma forma que “Curso” es un caso particular de un string y 2023 es un caso particular de un entero.

Clases y objetos

Vamos a definir una nueva clase y determinar sus atributos.

```
class Estudiante:
    #atributos de la clase
    nombre = ''
    edad=0

#Creamos dos objetos
estudiante1=Estudiante()
estudiante2=Estudiante()
```

✓ 0.0s

Clases y objetos

Definimos objetos y accedemos a su información.

```
#Damos valores a sus parámetros
estudiante1.nombre='Sofia'
estudiante2.nombre='Agustina'

estudiante1.edad = 28
estudiante2.edad= 27

print(f'{estudiante1.nombre} es estudiante y tiene {estudiante1.edad} años')
print(f'{estudiante2.nombre} es estudiante y tiene {estudiante2.edad} años')
```

✓ 0.0s

```
Sofia es estudiante y tiene 28 años
Agustina es estudiante y tiene 27 años
```

Clases y objetos

Para crear nuevos objetos es recomendable usar el método init:

```
[48]
class Perro:
    def __init__(self, breed, eyes):
        self.Raza = breed
        self.ColorDeOjos= eyes
✓ 0.0s

[49]
Rocko = Perro('Golden', 'Marron')
✓ 0.0s

[50]
Rocko.Raza
✓ 0.0s
... 'Golden'

[51]
Rocko.ColorDeOjos
✓ 0.0s
... 'Marron'
```

Clases y objetos

```python
class Numero:
    def __init__(self, value):
        self.value = value

    def imprimir(self):
        print(f'El número es {self.value}')

obj1 = Numero(17)
obj1.imprimir()
```

✓ 0.0s

```
El número es 17
```

```python
type(obj1)
```

✓ 0.0s

```
__main__.Numero
```

Clases y objetos

A partir de una determinada clase, se pueden definir nuevas clases. Las mismas tendrán los métodos y propiedades de la clase de la que provienen, lo que se conoce como herencia, además de las que se definan exclusivamente para ellas.

Clases y objetos

```python
#Clase general
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def Mensaje(self):
        print(f"{self.nombre} pertenece al reino animal.")

#Clase derivada
class Perro(Animal):
    def Ladrar(self):
        print(f'{self.nombre} puede ladrar.')
```

✓ 0.0s

```python
a1 = Animal('Dobbie')
a1.Mensaje()
```

[130] ✓ 0.0s

```
Dobbie pertenece al reino animal.
```

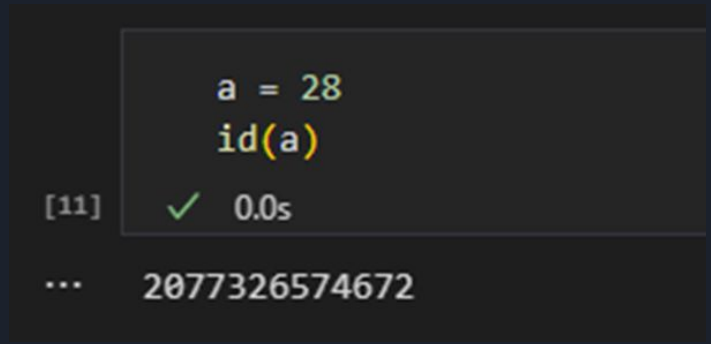
```python
a2 = Perro('Firu')
a2.Mensaje()
a2.Ladrar()
```

[135] ✓ 0.0s

```
Firu pertenece al reino animal.
Firu puede ladrar.
```

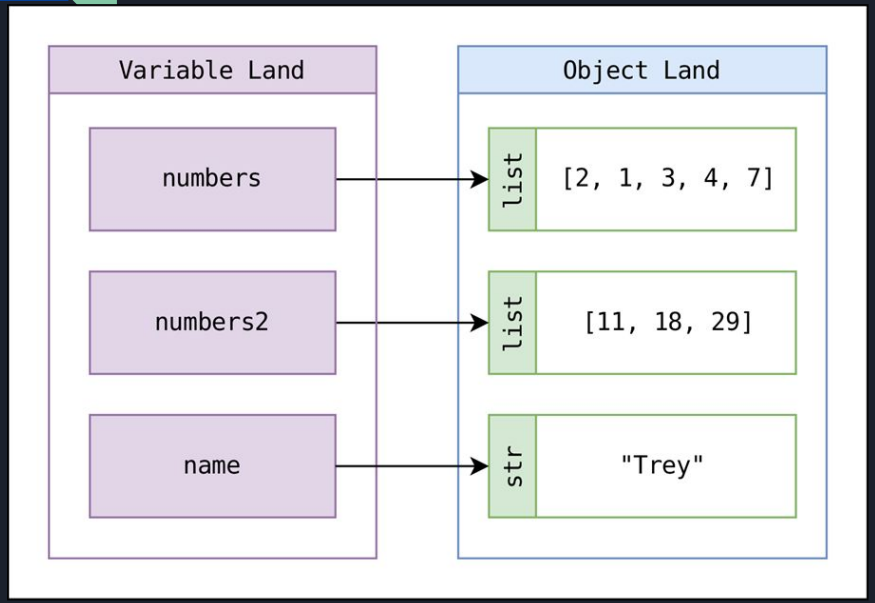
Clases y objetos

Cuando se crea un objeto en Python, este ocupa un lugar en la memoria. Si definimos una variable edad que tiene el valor 28, entonces 28 se define como un objeto que pertenece a una determinada clase (un número entero), y para conocer su lugar en la memoria se utiliza el comando id().

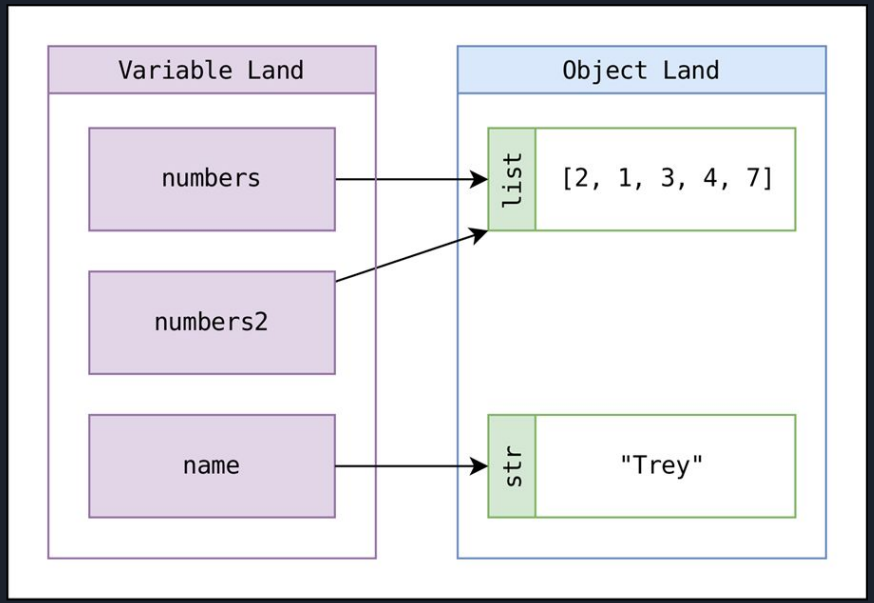


En Python, los nombres de las variables no son más que etiquetas o punteros que permiten acceder a los objetos o valores. Dos o más variables pueden apuntar al mismo objeto. Dos o más objetos pueden contener los mismos datos, pero ocupar lugares de memoria distintos.

Clases y objetos



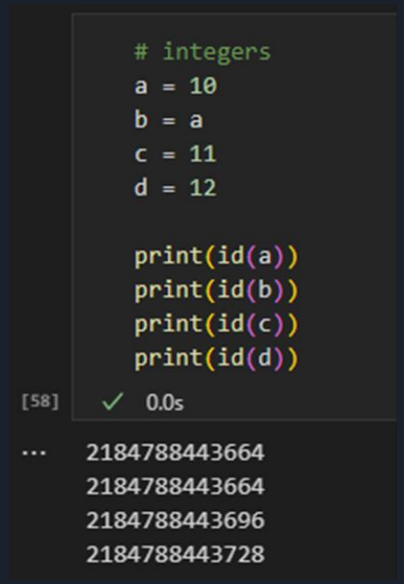

```
>>> numbers = [2, 1, 3, 4, 7]
>>> numbers2 = [11, 18, 29]
>>> name = "Trey"
```

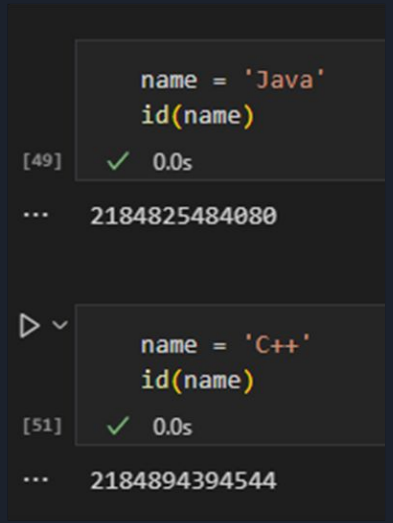


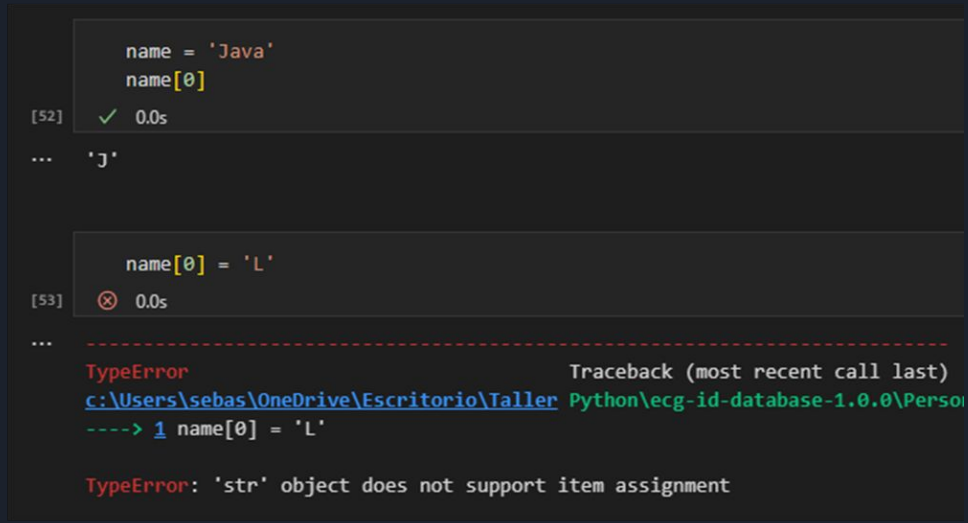

```
>>> numbers = [2, 1, 3, 4, 7]
>>> numbers2 = numbers
>>> name = "Trey"
```

Clases y objetos

Se dice que un objeto es inmutable cuando el mismo no se puede modificar.
Son objetos inmutables las tuplas y los tipos de datos primitivos (enteros, floats, booleanos, strings).







Clases y objetos

Las listas y diccionarios son objetos mutables. Si se definen dos objetos mutables, ocuparán lugares de memoria distintos, aunque los datos que contengan sean iguales.