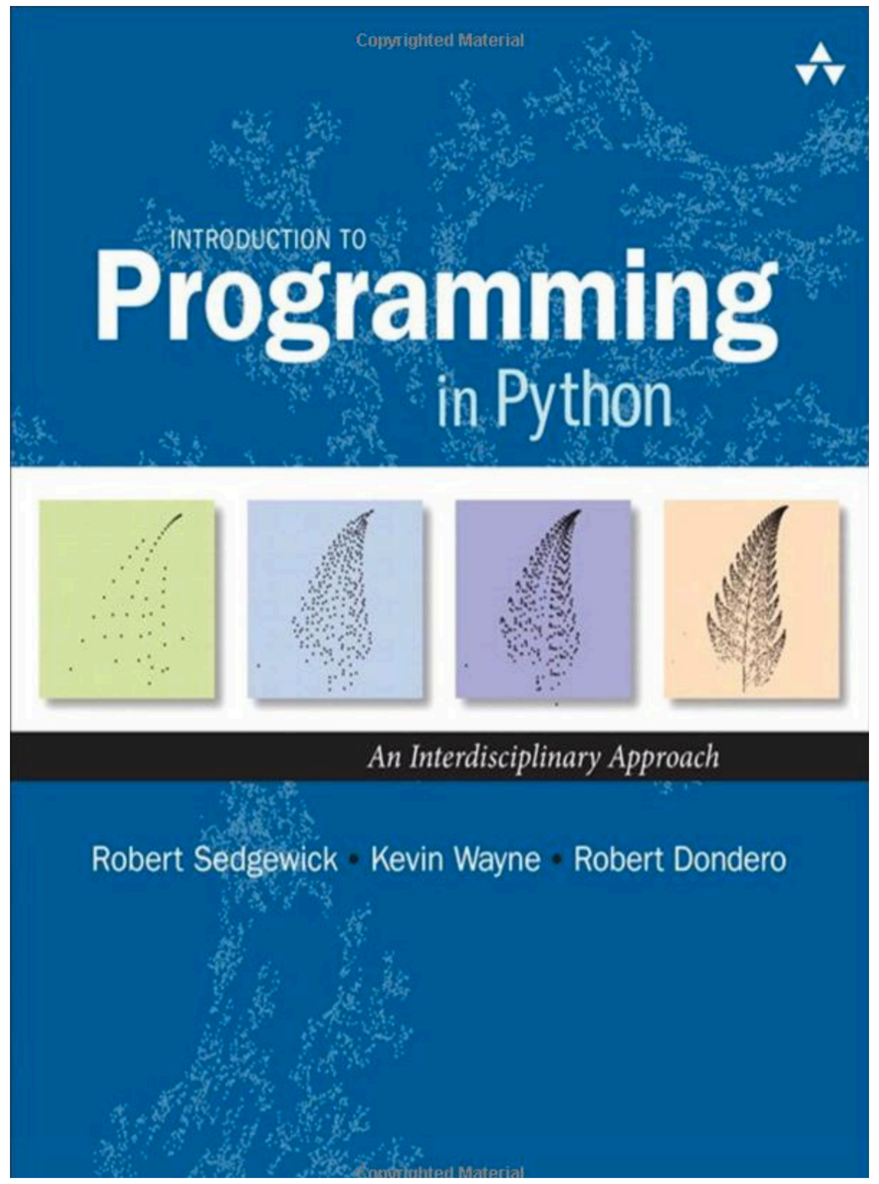
Parte II: Computación científica

Clase 13: Scientific computing with Python

Diego Caro
dcaro@udd.cl



Basada en presentaciones oficiales de libro Introduction to Programming in Python (Sedgewick, Wayne, Dondero).

Disponible en https://introcs.cs.princeton.edu/python

Aclaración Alias

```python
class Matrix:
    """Create a matrix of n rows and m columns."""
    def __init__(self, n, m):
        self.cols = m
        self.rows = n
        self.m = [ [0]*m ]*n

    def __str__(self):
        s = '\n'
        for row in self.m:
            s += '[ '
            for elem in row: s += str(elem) + ' '
            s += ']\n'
        return s

    def get(self, row, col):
        assert row < self.rows
        assert col < self.cols
        return self.m[row][col]

    def set(self, row, col, value):
        assert row < self.rows
        assert col < self.cols
        self.m[row][col] = value
```

Obtiene celda en posición row, col

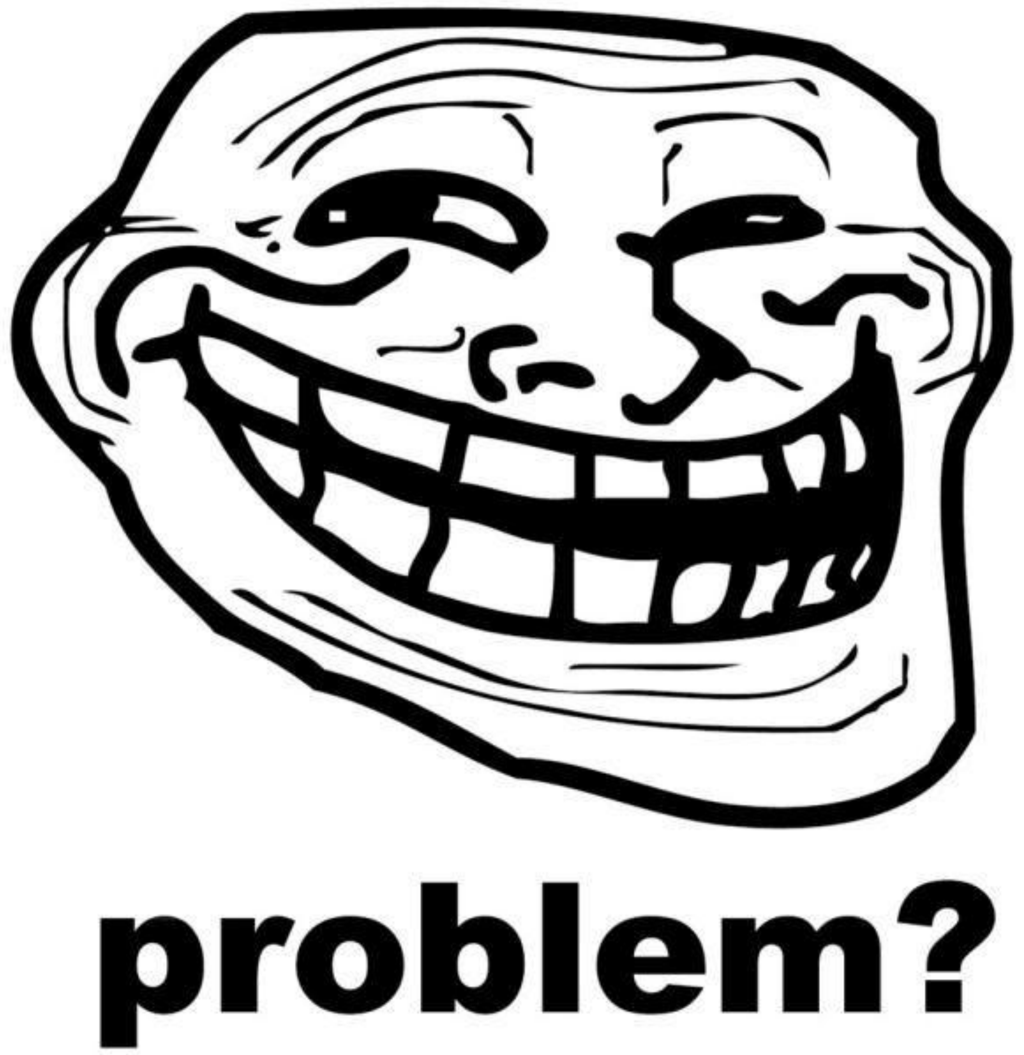
Asigna valor a celda en posición row, col

```python
m = Matrix(4, 5)
print('type(m):', type(m))
print('m:', m)
m.set(0, 1, 99)
print('m:', m)
```

Asigna 99 a celda en (0,1)

```
type(m): <class '__main__.Matrix'>
m:
[ 0 0 0 0 0 ]
[ 0 0 0 0 0 ]
[ 0 0 0 0 0 ]
[ 0 0 0 0 0 ]

m:
[ 0 99 0 0 0 ]
[ 0 99 0 0 0 ]
[ 0 99 0 0 0 ]
[ 0 99 0 0 0 ]
```



Memoria en Python

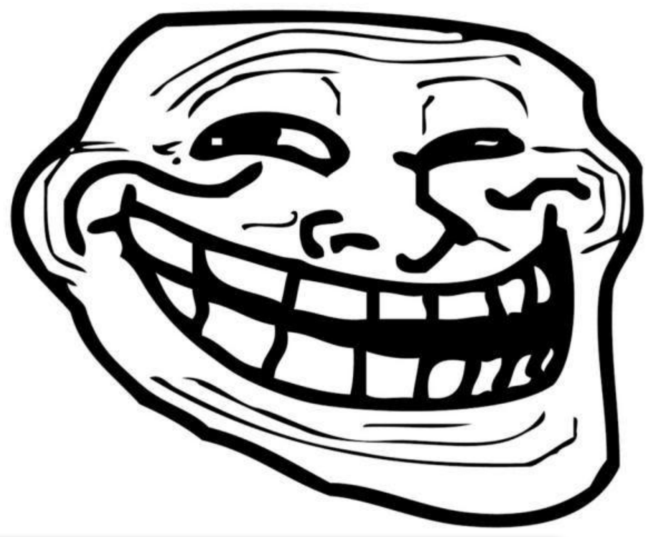
```python
class Matrix:
    """Create a matrix of n rows and m columns."""
    def __init__(self, n, m):
        self.cols = m
        self.rows = n
        self.m = [ [0]*m ]*n

    def id_elem(self):
        s = '\n'
        for row in self.m:
            s += str(id(row))
            s += ' -> [ '
            for elem in row: s += str(id(elem)) + ' '
            s += ']\n'
        return s
```

Ese es el problema!
Se están creando alias de [0] muchas veces!

Identificador de Matriz m en la memoria del PC

```python
print(id(m))
print(m.id_elem())
```



?

```
4522769824

4520562568 -> [ 4519530448 4519533616 4519530448 4519530448 4519530448 ]
4520562568 -> [ 4519530448 4519533616 4519530448 4519530448 4519530448 ]
4520562568 -> [ 4519530448 4519533616 4519530448 4519530448 4519530448 ]
4520562568 -> [ 4519530448 4519533616 4519530448 4519530448 4519530448 ]
```

Todos los elementos las filas son alias!!!!!!

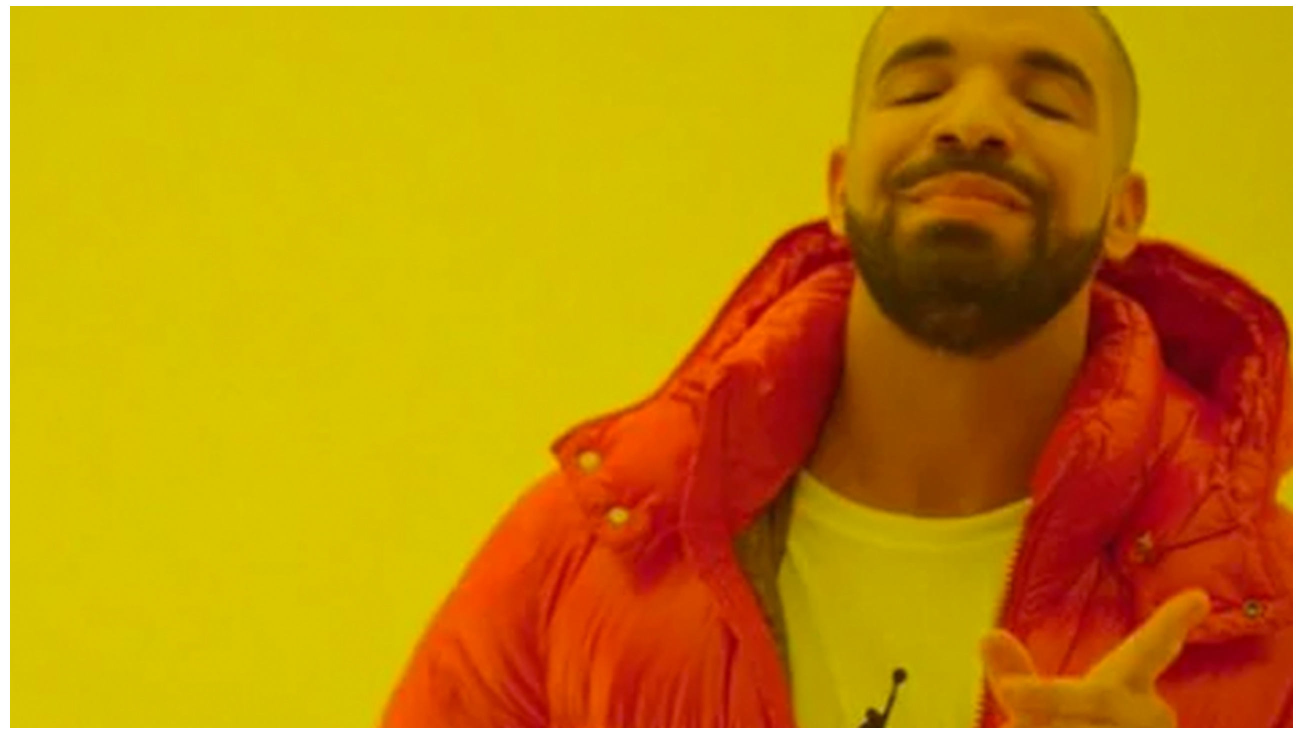
0	1	2	3	4	5	6	7	8	9	...	...	...	4520562567	4520562568	4520562569	4520562570

****Imaginar que esta tabla es la memoria del computador****

Como resolver la creación de alias

```
class Matrix:
    """Create a matrix of n rows and m columns."""
    def __init__(self, n, m):
        self.cols = m
        self.rows = n
        self.m = [ [0]*m ]*n
```

```
class Matrix:
    """Create a matrix of n rows and m columns."""
    def __init__(self, n, m):
        self.cols = m
        self.rows = n
        self.m = []
        for i in range(self.rows):
            self.m.append([0]*self.cols)
```



Intro a módulo Numpy

http://www.numpy.org/

- Python no incluye un tipo de dato para matrices, por lo tanto solo queda la alternativa de implementarla con listas....
- Pero existe numpy, un módulo para representar vectores, matrices y tensores.
  - Además incluye muchísimas operaciones para trabajar con matrices!
- Puede representar bool, int, float y complex.

```python
import numpy as np
# integer array:
intarr = np.array([1, 4, 2, 5, 3])
print(intarr)

mixedarr = np.array([3.14, 4, 2, 3])
print(mixedarr)

print(np.zeros(10, dtype=int))

print(np.ones((3, 5), dtype=float))
```

Se pueden mezclar de datos numéricos

Vector de ceros

matriz de 3 filas y 5 cols lleno con 1s

```
[1 4 2 5 3]
[3.14 4.   2.   3.  ]
[0 0 0 0 0 0 0 0 0 0]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

Operaciones básicas

```python
import numpy as np
np.random.seed(0)  # seed for reproducibility

# One-dimensional array (vector)
x1 = np.random.randint(10, size=6)

# Two-dimensional array (matrix)
x2 = np.random.randint(10, size=(3, 4))

print(x1)
print(x2)

print("x2 ndim: ", x2.ndim)
print("x2 shape:", x2.shape)
print("x2 size: ", x2.size)
print("dtype:", x2.dtype)

print('x1[0]:', x1[0])
print('x1[4]:', x1[4])
print('x1[-1]:', x1[-1])

print('x2[0, 0]:',x2[0, 0])
print('x2[2, -1]:',x2[2, -1])

x2[0, 0] = 99
print('x2[0, 0]:',x2[0, 0])
```

Se puede acceder con índices

En matrices se debe indicar col, row

También se pueden actualizar valores

```
[5 0 3 3 7 9]
[[3 5 2 4]
 [7 6 8 8]
 [1 6 7 7]]
x2 ndim:  2
x2 shape: (3, 4)
x2 size:  12
dtype: int64
x1[0]: 5
x1[4]: 7
x1[-1]: 9
x2[0, 0]: 3
x2[2, -1]: 7
x2[0, 0]: 99
```

https://docs.scipy.org/doc/numpy-1.15.4/reference/generated/numpy.random.randint.html

https://github.com/jakevdp/PythonDataScienceHandbook/blob/be23269c7eb119e093a6d5ce91e464f5e686d9ab/notebooks/02.02-The-Basics-Of-NumPy-Arrays.ipynb

Subvector / submatrices y acceso

```python
x2 = np.random.randint(10, size=(3, 4))
print('x2[:2, :3]:', x2[:2, :3])  # two rows, three columns

print('x2[:, 0]:', x2[:, 0])  # first column of x2

print('x2[0, :]:', x2[0, :])  # first row of x2

x2_sub = x2[:2, :2]
print('x2[:2, :2]:',x2_sub)

# update the array
x2_sub[0, 0] = 99
print('x2_sub:', x2_sub)
print('x2:',x2)

x2_sub_copy = x2[:2, :2].copy()
print('x2_sub_copy:', x2_sub_copy)

x2_sub_copy[0, 0] = 42
print('x2_sub_copy:', x2_sub_copy)
print('x2', x2)
```

Alerta de alias!

```
x2[:2, :3]: [[5 0 3]
 [7 9 3]]
x2[:, 0]: [5 7 2]
x2[0, :]: [5 0 3 3]
x2[:2, :2]: [[5 0]
 [7 9]]
x2_sub: [[99  0]
 [ 7  9]]
x2: [[99  0  3  3]
 [ 7  9  3  5]
 [ 2  4  7  6]]
x2_sub_copy: [[99  0]
 [ 7  9]]
x2_sub_copy: [[42  0]
 [ 7  9]]
x2 [[99  0  3  3]
 [ 7  9  3  5]
 [ 2  4  7  6]]
```

Concatenación

Axis = 0 significa concatenar por filas
Axis = 1 significa concatenar por columnas

```python
import numpy as np
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
c = np.concatenate([x, y])
print('c:', c)

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

# concatenate along the first axis
print('axis=0:',np.concatenate([grid, grid]))

# concatenate along the second axis (zero-indexed)
print('axis=1:',np.concatenate([grid, grid], axis=1))

```

```
c: [1 2 3 3 2 1]
axis=0: [[1 2 3]
 [4 5 6]
 [1 2 3]
 [4 5 6]]
axis=1: [[1 2 3 1 2 3]
 [4 5 6 4 5 6]]
```

Operaciones básicas vectores

- Aritmética con vectores, vector & escalar, vector & vector
  - + - * / %

Versión numpy

```python
import numpy as np

a = np.array([5, 0, 3, 3, 7, 9])
b = np.array([3, 5, 2, 4, 7, 6])

c = 2 + a # scalar + vector
print('c:', c)

c = a + b # vector + vector (also -,/,*,%)
print('c:', c)
```

```
c: [ 7  2  5  5  9 11]
c: [ 8  5  5  7 14 15]
```

Versión listas

```python
a = [5, 0, 3, 3, 7, 9]
b = [3, 5, 2, 4, 7, 6]
c = [0]*len(a)

# scalar and vector
for i in range(len(a)): c[i] = 2 + a[i]
print('c:', c)

# vector and vector
for i in range(len(a)): c[i] = a[i] + b[i]
print('c:', c)
```

```
c: [7, 2, 5, 5, 9, 11]
c: [8, 5, 5, 7, 14, 15]
```

Operaciones básicas matrices

- Aritmética con matrices + - * / %

```python
import numpy as np

a = np.array([[5, 0, 3, 3, 7, 9],[1, 1, 1, 1 ,1, 1]])
b = np.array([3, 5, 2, 4, 7, 6])

c = 2 + a # scalar + matrix
print('c:', c)

c = b + a # vector + matrix
print('c:', c)

c = a + a # matrix + matrix
print('c:', c)
```

```
c: [[ 7  2  5  5  9 11]
 [ 3  3  3  3  3  3]]
c: [[ 8  5  5  7 14 15]
 [ 4  6  3  5  8  7]]
c: [[10  0  6  6 14 18]
 [ 2  2  2  2  2  2]]
```

Más operaciones

```python
import numpy as np
m = np.array([[1, -2, 3],
                   [4, 5, -6]])
# matrix transpuesta
print('m.T:', m.T)

# dot product
print('m.dot(m.T):', m.dot(m.T))

# inverse matrix
a = np.array([[1., 2.], [3., 4.]])
ainv = np.linalg.inv(a)
print('ainv:', ainv)
print('a * ainv:', a.dot(ainv))
```

```
m.T: [[ 1  4]
 [-2  5]
 [ 3 -6]]
m.dot(m.T): [[ 14 -24]
 [-24  77]]
ainv: [[-2.   1. ]
 [ 1.5 -0.5]]
a * ainv: [[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]
```

https://docs.scipy.org/doc/numpy-1.15.1/reference/routines.array-manipulation.html

Aplicaciones: resolver sistemas de ecuaciones

De acuerdo con la primera ley de Kirchhoff (ley de los nodos), tenemos:

$$i_1 - i_2 - i_3 = 0$$

La segunda ley de Kirchhoff (ley de las mallas), aplicada a la malla según el circuito cerrado s_1, nos hace obtener:

$$-R_2 i_2 + \epsilon_1 - R_1 i_1 = 0$$

La segunda ley de Kirchhoff (ley de las mallas), aplicada a la malla según el circuito cerrado s_2, por su parte:

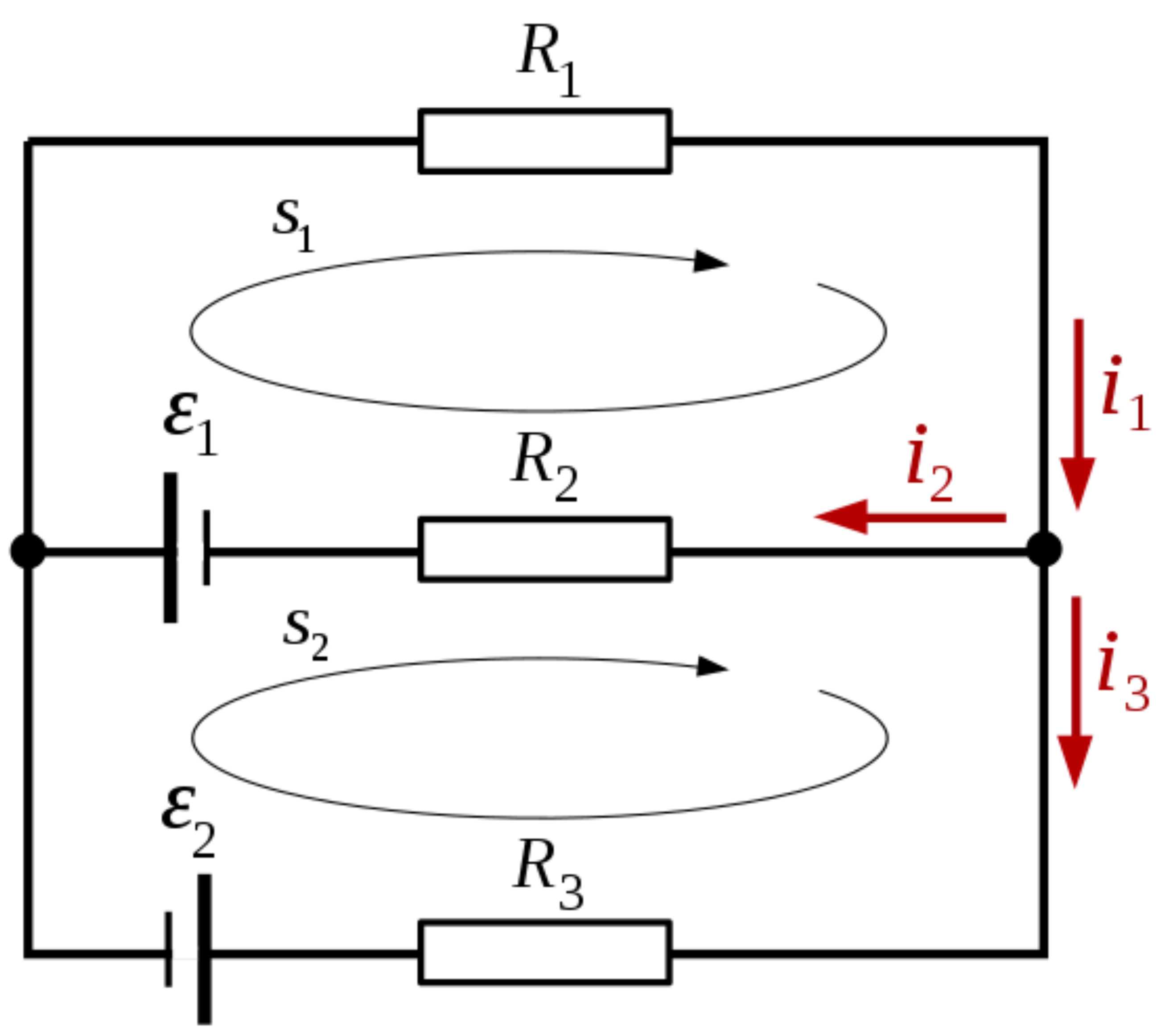
$$-R_3 i_3 - \epsilon_2 - \epsilon_1 + R_2 i_2 = 0$$

Debido a lo anterior, se nos plantea un sistema de ecuaciones con las incógnitas i_1, i_2, i_3:

$$\begin{cases} i_1 - i_2 - i_3 & = 0 \\ -R_2 i_2 + \epsilon_1 - R_1 i_1 & = 0 \\ -R_3 i_3 - \epsilon_2 - \epsilon_1 + R_2 i_2 & = 0 \end{cases}$$

Dadas las magnitudes:

$$R_1 = 100,\ R_2 = 200,\ R_3 = 300,\ \epsilon_1 = 3,\ \epsilon_2 = 4\,,$$



$$Ax = b$$

$$\begin{bmatrix} 1 & -1 & -1 \\ -1 & -R_2 & 0 \\ o & R_2 & -R_3 \end{bmatrix} \times \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} 0 \\ -e_1 \\ e_2 + e_2 \end{bmatrix}$$

Aplicaciones: resolver sistemas de ecuaciones

```python
import numpy as np

A = np.array([[1, -1, -1],[-100, -200, 0], [0, 200, -300]])
b = np.array([0, -3, 3 + 4])
x = np.linalg.solve(A, b)
print(x)
```

```
[ 0.00090909  0.01454545 -0.01363636]
```

Dadas las magnitudes:

$$R_1 = 100,\ R_2 = 200,\ R_3 = 300,\ \epsilon_1 = 3,\ \epsilon_2 = 4,$$

la solución definitiva sería:

$$\begin{cases} i_1 = \frac{1}{1100} \\ i_2 = \frac{4}{275} \\ i_3 = -\frac{3}{220} \end{cases}$$

```
>>> 1/1100
0.0009090909090909091
>>> 4/275
0.014545454545454545
>>> -3/220
-0.013636363636363636
```