

Seminario PYTHON - MCAF (Sesión I)

Francisco Gárate Santiago

UCM, 2022

Introducción a Python

Mínimas reglas

```
1 | print('Hola Mundo')
```

```
Hola Mundo
```

- ▶ La sintaxis es concisa y se utiliza de forma coherente en todas las bibliotecas
- ▶ La puntuación se utiliza con moderación para separar los bloques de código, lo que hace que el código Python sea muy legible
- ▶ El código Python se interpreta a la vez que se ejecuta, en vez de ser compilado*
- ▶ No hay necesidad de definir las variables previamente
- ▶ Una misma instrucción puede escribirse de varias formas (lo que se conoce como azúcar sintáctico)

Tipos de objetos

Variables y funciones

En Python no hace falta declarar el **tipo de variable** al darle un valor o leerla de una fuente externa, ya que interpreta y asigna el tipo de variables más adecuado. Las más comunes son:

- ▶ **str**: String o cadena de caracteres
- ▶ **int**: Números enteros
- ▶ **float**: Número con decimales
- ▶ **bool**: Booleano: Verdadero o falso (True/False)
- ▶ **datetime**: Fecha

Para conocer el tipo de variable se usa **type()**, que nos devolvería: int, str, float, datetime o bool
Del mismo modo, podemos utilizar las siguientes funciones para hacer **conversiones** entre los diferentes tipos de variables:

- ▶ **str()**: convierte a carácter
- ▶ **int()**: convierte a número entero
- ▶ **float()**: convierte a número con decimales

Tipos de objetos

Variables

```
1 a = 5  
2 b = 25  
3 print(a * b)
```

125

```
1 a = 5  
2 b = 25.  
3 print(a * b)
```

125.0

```
1 a = 5  
2 b = 'c'  
3 print(a * b)
```

CCCCC

Operaciones básicas

Son fácilmente adivinables:

- ▶ + Suma
- ▶ - Resta
- ▶ * Multiplicación
- ▶ / División
- ▶ ** Exponencial
- ▶ // Cociente de la división
- ▶ % Resto de la división

```
1 | a = 5
2 | b = 2
3 | print(a + b)
4 | print(a - b)
5 | print(a * b)
6 | print(a / b)
7 | print(a ** b)
8 | print(a // b)
9 | print(a % b)
```

7
3
10
2.5
25
1
2

Listas

Creación de listas

En Python, al igual que en otros lenguajes de programación, **se empieza a contar a partir del 0**.

Las listas pueden ser construidas de varias maneras:

- ▶ Usando un par de corchetes para denotar una lista vacía: []
- ▶ Usando corchetes, separando los elementos con comas: [a], [a, b, c]
- ▶ Usando la función **list()** sobre un objeto que sea iterable (por ejemplo un rango de valores).

```
1 | x = []
2 | y = [50, 51, 52, 53, 54, 55]
3 | z = list(range(10))
```

Listas

Operar con listas

- ▶ **s[i] = x** el item i de la lista s es reemplazada por x
- ▶ **s[i:j] = t** El contenido del segmento de s (desde i a j) es reemplazado por el contenido de t
- ▶ **s.append(x)**: añade x a la lista s

```
1 | s = []
2 | s.append(10)
3 | print(s)
```

10

- ▶ **s.clear()**: Elimina todos los items de s
- ▶ **s.copy()**: crea una copia de s. Ejemplo a = s.copy()
- ▶ **s.insert(i, x)**: inserta x dentro de s en el indice dado por i

Listas

Operar con listas

- ▶ **len(s)**: longitud s
- ▶ **min(s)**: El valor más pequeño (mínimo) de s
- ▶ **max(s)**: El valor más grande (máximo) de s
- ▶ **sort()**: ordena los elementos de la lista.
- ▶ **s.count(x)**: número total de ocurrencias de x en s

```
1 | x = [25, 45, 36, 12, 25, 65]  
2 | print(x.count(25))
```

2

- ▶ **abs()**: Devuelve el valor absoluto de un número

Secuencias y selecciones

Para seleccionar los valores de una lista se utilizan los corchetes, teniendo en cuenta que en Python **se empieza a contar por cero**.

- ▶ **lista[n]**: Selecciona el valor de la n-ésima posición
- ▶ **lista[n:m]**: Selecciona el valor de la n-ésima a la m-ésima posición.
- ▶ **lista[n:m:i]**: Selecciona el valor de la n-ésima a la m-ésima posición con saltos de i

En caso de omitirse **n** y/o **m**, se entenderá que la selección es desde el principio hasta el final.
Si fueran negativos, se entiende que dicho valor negativo se resta al valor del inicio o final.

```
1 | lista = list(range(10))  
2 | print(lista[5:9])  
3 | print(lista[::-1])  
4 | print(lista[:-1])
```

```
[5, 6, 7, 8]
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

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

Comentarios

Una buena costumbre a la hora de programar es **comentar el código** para facilitar el entendimiento de una tercera persona, o incluso de nosotros mismo si ha pasado tiempo desde que lo programamos.

Con `#` y `'''(triple apóstrofe)` se puede comentar una línea o un grupo de líneas respectivamente:

```
1 #!/usr/bin/python
2 # Esto es un comentario en una linea
3 a = 5
4 b = 2
5 ''' Esto son varias
6 lineas comentadas
7 en python '''
8
9 a + b # Resultado
```

Operadores lógicos

Los operadores lógicos se utilizan principalmente en funciones y sirven para interactuar con las variables o listas:

- ▶ == Igual que
- ▶ != No es igual que
- ▶ <> Diferente que (equivalente a !=)
- ▶ > Mayor que
- ▶ < Menor que
- ▶ >= Mayor o igual que
- ▶ <= Menor o igual que

Que a su vez pueden enlazarse con la condición OR y AND:

- ▶ or
- ▶ and

Condicionales

if...else

- El funcionamiento de if es fácil: se evalúa la condición, si es verdadera se ejecuta el código, si es falsa, no se ejecuta nada o se ejecuta otro código.

```
1 | if True:  
2 |     print('Se cumple la condicion')
```

- En cualquier función if .. else, los : (**dos puntos**) y el **espaciado** son necesarios.
- La instrucción **pass** es una operación nula y no pasa nada al ejecutarse. No es obligatorio utilizar **else** en un condicional. Suele usarse para llenar funciones inacabadas.

```
1 | if x >= 65:  
2 |     print('El valor es mayor o igual a 65')  
3 | else:  
4 |     pass
```

- La instrucción **continue** finaliza el bucle hasta ese punto, aunque continua ejecutándose el bucle (es decir, ignora todo lo que venga a continuación de continue).

Condicionales

if...else

- ▶ Puede haber tantos if como se quiera con la expresión **elif** (*else if*)
- ▶ La ejecución o lectura del código finalizará en la primera condición que se cumpla.

```
1  x = 60
2  if x >= 65:
3      print('El valor es mayor o igual a 65')
4  elif x > 50:
5      print('El valor es mayor que 50')
6  else:
7      pass
```

El valor es mayor que 50

Iteraciones

for

- ▶ Los bucles **for** son estructuras que repiten un bloque de instrucciones un número determinado de veces. El bloque de instrucciones que se repiten se llaman cuerpo del bucle y cada repetición se llama **iteración**.
- ▶ Los bucles son un recurso muy utilizado en la programación de python. Por ejemplo, utilizar el bucle for para anexar datos a una lista es una funcionalidad muy utilizada:

```
1 s = []
2 for i in range(5):
3     s.append(i)
4
5 print(s)
```

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

Creación de funciones

- ▶ Podemos crear nuestras propias funciones fácilmente con la palabra reservada **def**:
`def nombre-funcion(argumentos) :`
- ▶ Los **argumentos** van entre paréntesis y son opcionales e incluso se pueden especificar valores por defecto.
Ej.: `def nombrefun(x, y=100)` o `def nombrefun(x, y , *args)`
- ▶ A continuación, se introduce el cuerpo de la función con una tabulación (o 4 espacios según guía de estilo PEP 8).
- ▶ El resultado de la función deberá devolverse con la instrucción **return**.

Ejemplo de función que elevada al cuadrado el número dado:

```
1 | def nombrefuncion (x) :  
2 |     m = x ** 2  
3 |     return m  
4 |  
5 | print(nombrefuncion(5))
```

Manejo de errores y excepciones

Los errores nunca deberían pasar silenciosamente, a menos que se silencien explícitamente

La declaración **try .. except** puede resultar bastante útil en nuestras funciones:

```
1 def divide(x, y):  
2     try:  
3         return (x / y)  
4     except:  
5         return 0.
```

Puede especificarse el mensaje o instrucción por tipos de errores (ej. `except ZeroDivisionError:`)

- ▶ `ZeroDivisionError`: Error de división por cero.
- ▶ `TypeError`: la función se aplica a un objeto de tipo inapropiado.
- ▶ `ValueError`: el valor no está en el rango de valores posibles
- ▶ `NameError`, `RuntimeError`
- ▶ `KeyboardInterrupt`: cuando se interrumpe durante la ejecución (p.e. Control-C)
- ▶ Otros: `OSError`, `BufferError`, `MemoryError`, `LookupError`, `IndexError`, `KeyError`,
- ▶ Lista completa: <https://docs.python.org/3/library/exceptions.html#bltin-exceptions>

Librerías

Importar librerías

- ▶ Podemos hacer nuestras propias funciones o bien importarlas de librerías (o bibliotecas) específicas para cada ámbito.
- ▶ Estas librerías pueden venir instaladas de serie (standard library) como `os`, `sys`, `math`, `decimal`, `urllib` o pueden ser instaladas desde repositorios externos.
- ▶ Normalmente las librerías suelen ser, al igual que Python, de código abierto y puedes consultar y modificar su código.
- ▶ Donde conseguir librerías:
 - ▶ PyPi: repositorio oficial. Se instalan desde la línea de comandos:
`pip install LIBRARY`
 - ▶ Github: principal repositorio de librerías open-source. Facebook, el CERN o la NASA han publicado y liberado sus propias librerías de python en Github. Se descargan desde github o desde linea de comandos:
`git clone https://github.com/USERNAME/REPOSITORY`

Standard Library

Baterías incluidas

Vienen incluidas en la instalación de python. Las más utilizadas:

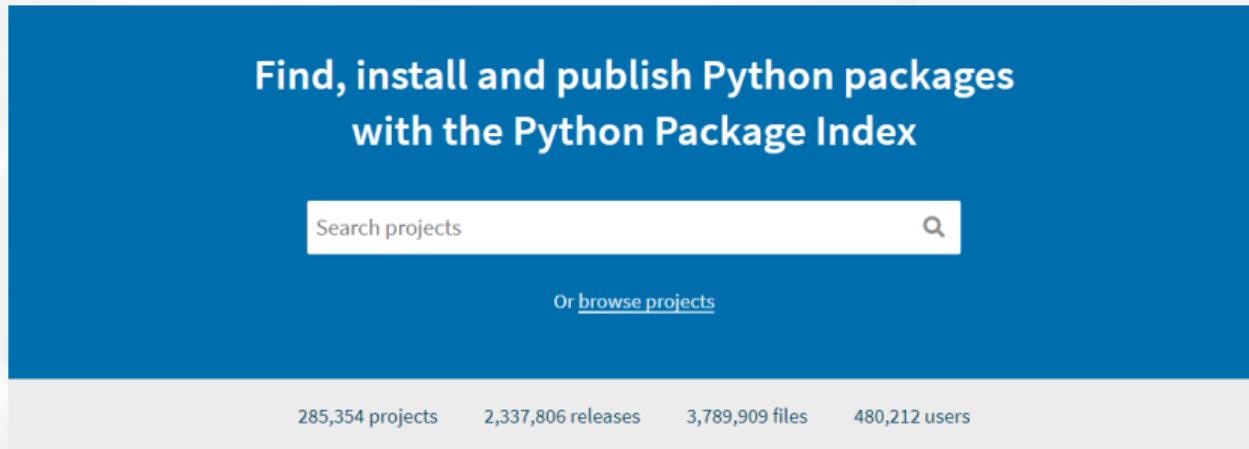
- ▶ **csv**: Lectura y escritura de ficheros csv
- ▶ **datetime**: Operar con fechas
- ▶ **decimal**: Aporta mayor precisión a operaciones con decimales (usa código C)
- ▶ **gzip, zlib, bz2, zipfile, tarfile**: para manejo de ficheros comprimidos
- ▶ **math**: alternativa ligera de numpy (exp, sqrt...)
- ▶ **itertools**: Permite crear combinaciones y permutaciones fácilmente.
- ▶ **os**: Acceso directo al sistema operativo (OS) a través de comandos.
- ▶ **sys**: Obtiene información del sistema. Útil para añadir seguridad al código.
- ▶ **tempfile**: manejo de ficheros temporales que pueda requerir nuestro código
- ▶ **urllib**: Acceso al código html de cualquier url pública.

Documentación: <https://docs.python.org/3/library/>

Importar librerías

pypi.org

- Existe un repositorio oficial de librerías (pypi.org), que actualmente posee más de 285.000 proyectos.
- Para instalarlas tan sólo hay que ejecutar: `pip install NOMBRE_LIBRERIA`



The Python Package Index (PyPI) is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community. [Learn about installing packages ↗](#)

Package authors use PyPI to distribute their software. [Learn how to package your Python code for PyPI ↗](#)

Añadir librerías

PyPi: Repositorio de paquetes

- ▶ Muchas librerías de python tienen dependencias a otras. En ese caso, deben instalarse conforme van siendo demandadas.
- ▶ Para desinstalar una librería: `pip uninstall numpy`
- ▶ Para actualizar una librería: `pip install numpy --upgrade` o `numpy -U`
- ▶ Para saber las librerías que tenemos instaladas: `pip list`
- ▶ Para instalar una versión específica (downgrade) Ej.: `pip install numpy==1.19.1`

version y dir()

Mostrar la versión y funciones incluidas

También puede consultarse la versión de la librería y las funciones incluidas con los siguientes comandos:

```
1 import numpy  
2 print(numpy.__version__)  
3 print(dir(numpy.random))
```

1.20.0

'Lock', 'RandomState', 'absolute_import', 'beta', 'binomial', 'bytes', 'chisquare', 'choice', 'dirichlet',
'division', 'exponential', 'f', 'gamma', 'geometric', 'get_state', 'gumbel', 'hypergeometric', 'info',
'laplace', 'logistic', 'lognormal', 'logseries', 'mtrand', 'multinomial', 'multivariate_normal',
'negative_binomial', 'noncentral_chisquare', 'noncentral_f', 'normal', 'np', 'operator', 'pareto',
'permutation', 'poisson', 'power', 'print_function', 'rand', 'randint', 'randn', 'random',
'random_integers', 'random_sample', 'ranf', 'rayleigh', 'sample', 'seed', 'set_state', 'shuffle',
'standard_cauchy', 'standard_exponential', 'standard_gamma', 'standar_normal', 'standard_t',
'test', 'triangular', 'uniform', 'vonmises', 'wald', 'warnings', 'weibull', 'zipf'

Importar librerías

Una vez instalada una librería, se la puede llamar de varias formas:

```
1 import numpy  
2 print(numpy.pi)  
3 print(numpy.log(2))
```

Importar sólo las funciones que vas a necesitar:

```
1 from numpy import pi, log  
2 print(pi)  
3 print(log(2))
```

Usando un alias (la más común):

```
1 import numpy as np  
2 print(np.pi)  
3 print(np.log(2))
```

3.141592653589793

0.6931471805599453

Principales librerías matemáticas y estadísticas

- ▶ **Numpy**: librería matemática
- ▶ **Pandas**: Análisis de datos (dataframes, vectores). Similar a R.
- ▶ **Matplotlib**: librería gráfica
- ▶ **Scipy**: funciones estadísticas
- ▶ **Statsmodels**: funciones estadísticas
- ▶ **Scikit-learn**: librería de *machine learning*

NumPy 

 **pandas**

 **SciPy**

 **matplotlib**

 **jupyter**

Numpy

Librería NumPy

Numerical computation

- ▶ NumPy (*Numerical computation Python*) es la librería matemática de Python por excelencia, nacida de la unión de varias librerías matemáticas.
- ▶ **Principales utilidades:**
 - ▶ Operaciones con vectores y matrices (N-arrays)
 - ▶ Estadística básica
 - ▶ Manejo de fechas (datetime64)
 - ▶ Álgebra lineal (numpy.linalg)
 - ▶ Generación de números aleatorios (numpy.random)
- ▶ **Sitio:** <http://www.numpy.org>
- ▶ **Documentación:** <https://numpy.org/devdocs/reference/index.html>
- ▶ **Licencia:** open-source BSD-new

Vectores y matrices

N-arrays

- ▶ Trabajar con vectores y matrices (N-arrays) es la verdadera funcionalidad de NumPy.
- ▶ La estructura de datos son los "ndarray": n-dimensional array

```
1 import numpy as np
2 a = np.array([1, 2, 3, 5])
3 print(a * 3)
4
5 b = np.arange(10)
6 # arange = similar a range, pero devuelve un array, no un listado
7 print(b * 3)
```

```
[ 3 6 9 15]
```

```
[ 0 3 6 9 12 15 18 21 24 27]
```

Vectores y matrices

Secuencias

- ▶ **np.linspace(inicio, fin, n-pasos)**: Devuelve los números espaciados uniformemente durante un intervalo especificado. Por defecto: 50 números.

```
1 import numpy as np  
2 c = np.linspace(5, 18, 6)  
3 print(c)
```

```
[ 5.  7.6 10.2 12.8 15.4 18. ]
```

- ▶ **np.zeros(n) y np.ones(n)**: Creación de vectores n-dimensionales con ceros o unos:

```
1 print(np.zeros(5))  
2 print(np.ones(5))
```

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

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

Indexación

Arrays indexing

```
1 import numpy as np  
2 x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

La sintaxis básica de las secciones es $i:j:k$, donde i es el índice inicial, j es el índice final y k son los n -pasos.

- ▶ $x[1:7:2]$ es [1 3 5]
- ▶ $x[-2:10]$ es [8 9] *al igual que $x[-2:]$*
- ▶ $x[-3:3:-1]$ es [7 6 5 4]
- ▶ $x[::-1]$ es [9 8 7 6 5 4 3 2 1 0]

Manejar este concepto es importante para comprender más adelante el funcionamiento de muchos enfoques.

Operaciones con vectores

Sumas, productos, diferencias

Con np.array se pueden utilizar las funciones incluidas en Python:

```
1 import numpy as np  
2 x = np.array([1, 2, 3, 5])  
3 print(len(x))
```

4

Operaciones con vectores

Sumas, productos, diferencias

Con np.array se pueden utilizar las funciones incluidas en Python:

```
1 import numpy as np  
2 x = np.array([1, 2, 3, 5])  
3 print(len(x))
```

4

Así como las funciones que el propio NumPy incorpora:

- ▶ **np.sum(x)**: Suma de los valores del vector. En este caso: 11
- ▶ **np.prod(x)**: Producto de los valores del vector. En este caso: 30
- ▶ **np.cumsum(x)**: Suma acumulada de los valores del vector. En este caso: [1 3 6 11]
- ▶ **np.cumprod(x)**: Producto acumulado. En este caso: [1 2 6 30]
- ▶ **np.diff(x)**: Diferenciales o valores añadidos en cada salto. En este caso: [1 1 2]

Operaciones con vectores

statistics

```
1 import numpy as np  
2 x = np.array([1, 2, 3, 5])
```

Estadísticos de centralización y dispersión:

- ▶ **np.median(x)** es 2.5
- ▶ **np.mean(x)** es 2.75
- ▶ **np.std(x)** es 1.479019945774904
- ▶ **np.var(x)** es 2.1875

Operaciones con vectores

statistics

```
1 import numpy as np  
2 x = np.array([1, 2, 3, 5])
```

Estadísticos de posición:

- ▶ **np.amin(x)** es 1
- ▶ **np.amax(x)** es 5
- ▶ **np.percentile(x, 50)** es 2.5
- ▶ **np.quantile(x, 0.5)** es 2.5

Operaciones con vectores

Las fórmulas incluidas en NumPy Pueden aplicarse de dos formas. Ejemplo:

```
1 import numpy as np  
2 x1 = np.array([1, 2, 3, 5]).sum()  
3 x2 = sum(np.array([1, 2, 3, 5]))  
4 print(x1, x2)
```

```
11 11
```

Operaciones con vectores

Exponenciales, logaritmos, constantes

Exponenciales y logaritmos:

- `np.exp(1.25)` es 3.4903429574618414
- `np.log(1.25)` es 0.22314355131420976
- `np.log10(1.25)` es 0.09691001300805642

Operaciones con vectores

Exponenciales, logaritmos, constantes

Exponenciales y logaritmos:

- ▶ `np.exp(1.25)` es 3.4903429574618414
- ▶ `np.log(1.25)` es 0.22314355131420976
- ▶ `np.log10(1.25)` es 0.09691001300805642

Principales constantes:

- ▶ `np.e` es 2.718281828459045
- ▶ `np.pi` es 3.141592653589793
- ▶ `np.nan` es NaN (*Not a Number*)

Operaciones con vectores

Exponenciales, logaritmos, constantes

Exponenciales y logaritmos:

- ▶ `np.exp(1.25)` es 3.4903429574618414
- ▶ `np.log(1.25)` es 0.22314355131420976
- ▶ `np.log10(1.25)` es 0.09691001300805642

Principales constantes:

- ▶ `np.e` es 2.718281828459045
- ▶ `np.pi` es 3.141592653589793
- ▶ `np.nan` es NaN (*Not a Number*)

Otras funciones útiles:

- ▶ `np.power(1.25, 2)` es 1.5625
- ▶ `np.sqrt(1.5625)` es 1.25

Operaciones con vectores

Multiplicación

En el caso de querer multiplicar dos vectores, existe la función **np.vdot(a, b)**:

```
1 import numpy as np  
2  
3 a = np.array ([1 , 2 , 3 , 4 , 5])  
4 b = np.array ([3 , 6 , 8 , 9 , 10])  
5  
6 print(np.sum(a * b))  
7 print(np.vdot(a, b))
```

125

125

Matrices

Creación de matrices

```
1 import numpy as np  
2 x = np.array ([[1, 2, 3], [4, 5, 6]])  
3 print(x)  
4 print(x.shape)
```

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

```
(2, 3)
```

Seleccionar fila, puede indicarse con corchetes. Ejemplo: **x[0]** devuelve **[1 2 3]**

Y para seleccionar un valor dentro de la matriz, indicando su posición.

Ejemplo: **x[0][0]** o su equivalente **x[0,0]** devuelve **1**.

Selecciones

```
1 import numpy as np  
2 x = np.array([1, 2, 3, 4, 5, 6])  
3 print(x[x>2])
```

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

```
1 import numpy as np  
2 x = np.array([[1, 2, 3], [4, 5, 6]])  
3 x[0,0] = 5  
4 print(x)
```

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

```
1 print(x[0:2, 1])
```

```
[2 5]
```

Operaciones con matrices

Toda matriz posee dos ejes. El axis=0 corresponde con el eje horizontal (lectura por defecto) y axis=1 con el vertical.

- ▶ **np.sum(x,y)**: Devuelve la suma de los elementos del array sobre el eje definido
- ▶ **np.prod(x,y)**: Devuelve el producto de los elementos del array sobre un eje dado.

```
1 import numpy as np
2 a = np.prod([[1.,2.],[3.,4.]], axis=0)
3 print(a)
4
5 b = np.sum([[1.,2.],[3.,4.]], axis=1)
6 print(b)
```

[3. 8.]

[3. 7.]

Del mismo modo, en el caso de cumsum() y cumprod(), si queremos acumular la suma o multiplicación de izquierda a derecha debemos especificar axis=1

Operaciones con matrices

Multiplicación de matrices.

- ▶ **np.dot(x, y)**: Multiplica dos matrices, donde la matriz identidad y debe tener el mismo orden que x, es decir, que el número de columnas de x debe coincidir con el número de filas de y.

```
1 import numpy as np
2 a = np.array ([[1, 2, 3], [4, 5, 6]])
3 b = np.array ([[7, 8], [9, 6], [5, 1]])
4
5 print(a.shape, b.shape)
6 print(np.dot(a, b))
```

(previamente comprobamos que el número de filas de A sea igual que el número de columnas de B)

(2, 3) (3, 2)

```
[[ 40 23]
 [103 68]]
```

Operaciones con matrices

Otras funciones útiles con matrices

- ▶ **.transpose()** o **.T**: Devuelve la matriz transpuesta.
- ▶ **.inverse()** o **.I**: Devuelve la matriz inversa
- ▶ **.diagonal()**: Devuelve la diagonal principal.

Reordenación de elementos:

- ▶ **flip(m[, axis])**: Invierte el orden de los elementos de un array según el eje dado.
- ▶ **flplr(m)**: Voltea la matriz sobre el eje vertical (izquierda/derecha). Ejemplo
`np.flplr(A)`
- ▶ **flipud(m)**: Voltea la matriz sobre el eje horizontal (arriba/abajo).

Ejercicio 1

Ejercicio 1

Dados los siguientes valores de los diferentes submódulos de SCR:

Mercado	Crédito	Vida	Salud	No Vida
95 000	65 000	25 000	35 000	125 000

Se pide:

- ▶ Calcular el BSCR aplicando la matriz de correlaciones del Reglamento Delegado de Solvencia II:

1	0.25	0.25	0.25	0.25
0.25	1	0.25	0.25	0.5
0.25	0.25	1	0.25	0
0.25	0.25	0.25	1	0
0.25	0.5	0	0	1

- ▶ Calcular el efecto de diversificación.

NumPy Financial

Operaciones financieras

- ▶ NumPy Financial es el reemplazo de las funciones financieras originales de NumPy.
- ▶ Al ser el *spin off* de NumPy, hereda sus funciones financieras elementales con objeto implementar nuevas funcionalidades a futuro.
- ▶ Las funciones financieras en NumPy (npv, irr, etc..) han sido eliminadas en la versión 1.20 de NumPy.
- ▶ **Funciones:** (versión 1.0.0):
 - ▶ Descuento de flujos: fv, pv, npv
 - ▶ Tipo de interés: irr, mirr, rate
 - ▶ Cálculo de pagos y periodos de pago: pmt, ppmt, ipmt, nper
- ▶ **Sitio:** <https://numpy.org/numpy-financial/>
- ▶ **Documentación:**
<https://numpy.org/doc/1.19/reference/routines.financial.html>
- ▶ **Licencia:** open-source BSD-new

NumPy Financial

IRR, NPV

- ▶ **npf.irr(valores)**: Devuelve la TIR o IRR.

```
1 import numpy_financial as npf  
2 payments = [-100, 39, 59, 55, 20]  
3 tir = npf.irr(payments)  
4 print(tir)
```

```
0.2809484211599611
```

- ▶ **npf.npv(i, payments)**: net present value

```
1 import numpy_financial as npf  
2 payments = [100, 100, 100, 100, 100, 100, 100, 100, 100, 100]  
3 net_present_value = npf.npv(0.02, payments)  
4 print(net_present_value)
```

```
916.2236706367081
```

Simulaciones y números aleatorios

numpy.random

- ▶ Las **simulaciones** requieren el uso de muestras aleatorias para encontrar una solución a un problema matemático.
- ▶ **Numpy** posee el módulo `numpy . random` que implementa **generadores de números pseudo-aleatorios** para varias distribuciones.
- ▶ Ofrece números aleatorios de distribuciones discretas y continuas, así como algunas utilidades relacionadas con los números aleatorios.
- ▶ Usa PCC64 como generador interno¹ (uno de los nuevos generadores más avanzados que existen). Esta basado en C que utiliza los ciclos del reloj del procesador.

¹<https://www.pcg-random.org/index.html>

Números aleatorios

numpy.random

Principales funciones de **numpy.random**:

- ▶ **np.random.rand(x,y)**: Devuelve valores aleatorios en una forma dada, entre 0 y 1.
- ▶ **np.random.randn(x,y)**: Devuelve una muestra (o muestras) de la distribución normal estándar (Gaussina), entre 0 y 1.
- ▶ **np.random.randint(low[, high, size])** números enteros aleatorios desde una base (inclusive) a un tope (exclusivo).
- ▶ **np.random.random(size)** Devuelve numeros *float* aleatorios en el intervalo medio-aberto [0.0, 1.0)
- ▶ **np.random.choice(a[, size, replace])** Genera una muestra aleatoria de un array 1-D dado.
 - ▶ a: 1-D array
 - ▶ size: tamaño de la muestra
 - ▶ replace: True (con reemplazamiento), False (sin reemplazamiento)