Curso completo de Python 3 de la A a la Z

Por Juan Gabriel Gomila

Notad de Roberto Negro

15 de abril de 2021

Índice

1 Introducción

1.1 Sobre el curso

Enlace al Curso completo de Python 3 de la A a la Z.

1.2 Uso de Discord

Discord es un servicio de mensajería instantánea. Se trata de una comunidad para resolver dudas.

1.3 Uso de Google Colaboratory

`Google Colaboratory` es una herramienta utilizada en el curso para programar en Python. Se trata de un fichero notebook. Evita tener que instalar las librerías de Python a mano, tarea compleja para un principiante.

1.4 Uso de Jupyter Notebooks en VSCode

En lugar de usar Google Colaboratory, se puede utilizar el editor `VSCode`, configurándolo para poder usar Jupyter Notebooks. Para ello, consultar los apuntes de python.

El modo de trabajar consistirá:

1. Abrir el recurso subidos al curso de Udemy (enlace a Google Colaboratory)
2. Descargar el archivo `.ipynb`
3. Abrir el archivo `.ipynb` en VSCode.

1.5 Uso de Python en VSCode

Consultar los apuntes de python.

2 Tema 1. Python básico

En los Jupyter Notebook se utiliza markdown. El notebook hace uso de celdas: doble clic para editarlas.

Nota. Al programar en Python se recomienda, o tabular o utilizar espacio, nunca mezclar ambos.

2.1 El concepto de variable

Una `variable` (en python) es un identificador/etiqueta asociado a un tipo de dato. Un mismo dato puede estar asociado a varias variables. Para dar un valor a una variable se utiliza el operador de asignar `=`

```
<nombre_variable> = <valor>
```

Podemos hacer

```
x = 1
y = "hola"
```

Se adelanta que `x` es una variable de tipo numérico e `y` es una variable de tipo string. En este último caso da igual utilizar comillas simples que dobles.

Restricciones sobre nombres de variables:

- No colocar nada que no sean letras y números.
- No se pude comenzar por un número.
- No es buena práctica utillizar acentos.
- No utilizar palabras reservadas de Python.
- No utilizar espacios.

- Es una buena práctica utiliza nombres coherentes.

Consejos sobre la nomenclatura:

1. Utilizar `nombre_Mascota` para variables
2. Utilizar `Nombre_Mascota` para clases
3. Utilizar `nombre_mascota` para funciones

2.2 Palabras clase en Python

Para listarlas se ejecuta

```
import keyword
keyword.kwlist
```

2.3 Declaración de múltiples variables

Podemos declarar múltiples variables en una sola línea:

```
age, name = 30, "Marta"
```

2.4 Operaciones con variables de tipo numérico

Algunas ordenes interesantes

- `x = 1`
- `x = x + 1` sobrescribe el valor original
- `x += 1` equivale a la orden 2

Del mismo modo se utiliza `-=` , `*=` , `/=` , `%=` , `**=` para la resta, el producto, el cociente, el resto de la división entera y la potencia repecticamente.

2.5 Comentarios

En Python `# esto es un comentario`

Consejos:

- No abusar de los comentarios
- Comentar, preferiblemente, antes del bloque de código, con la salvedad de líneas de código "raras".

2.6 La función import

En primer lugar, algo de nomenclatura:

- Un `algoritmo` es una receta de cocina.
- Una `función` es una máquina que "come" argumentos y "devuelve" valores o tareas.
- Un `script` es un archivo preparado para ser ejecutado.
- Un `módulo` o `librería` es un script que contiene funciones, programas, etc.

Algunos comandos:

- `import <module>` importa las funciones del modulo.
- `import <module> as <alias>` importa las funciones del modulo y le asigna un alias.
- `from <module> import <function>` importa una función del modulo.
- `from <module> import <function> as <alias>` hace lo anterior estableciendo un alias.

Algunos ejemplos:

```
import math
```

Las funciones de `math` se ejecutan con el formato `math.<function>()` y las variables con el formato `math.<variable>`

```
import math as mt
```

Con el alias anterior, las funciones de `math` se ejecutan con el formato `mt.<funcion>()` y las variables con el formato `mt.<variable>` .

Nota. Es recomendable utilizar alias de modo sistemático aunque los nombres sean cortos.

```
from math import pi
```

```
from math import pi as numero_pi
```

Ordenando

```
from math import pi, log, exp
```

podemos utilizar la variable `pi` y las funciones `log()` , `exp()` .

Nota. Podemos utilizar `from math import *` para llamar a la funciones de `math` únicamente por su nombre, pero no es recomendable porque pueden

existir funciones de distintos módulos con el mismo nombre.

3 Tema 2: Números en Python

3.1 Tipos de números en Python

Hay dos tipos de números:

1. `int` : número entero
2. `float` : número de coma flotante

Nota. En Python, la "coma decimal" viene dada por el punto. Además, `3.` y `3.0` es lo mismo.

Algunos comandos útiles:

- `type()` nos da el tipo de número.
- `int()` transforman a número entero.
- `float()` transforman a número en coma flotante.

Algunos ejemplos:

```
type(5)
type(5.0)
```

3.2 Operaciones aritméticas

Disponemos de las operaciones `+` , `-` , `*` , `/` , `**` , `//` , `%` . Las dos últimas son la división entera y el resto de la división.

Algunas notas:

1. Si hacemos opoeraciones mezclando números enteros con números en coma flotante, el resultado viene dado en coma flotante.
2. Para evitar problemas (relacionados con división normal o entera) se recomienda ordenar `6/5.0` en lugar de `6/5` .
3. La potencia n-ésima puede ejecutarse como `pow(5,3)` en lugar de `5**3`
4. Se utiliza la jerarquía de operaciones clásica, con el criterio de izquierda a derecha.

3.3 Números complejos en Python

Algunos comandos útiles:

- `z = 2+5j` declara el número complejo (2,5)
- `z.real` devuelve la parte real de `z`
- `z.imag` devuelve la parte imaginaria de `z`
- `abs()` devuelve el valor absoluto de un real y el módulo de un complejo.

Algunas notas:

1. Se utiliza `j` en lugar de `i`
2. Para utilizar el complejo j o $-j$ debemos ordenar `1j` y `-1j`

De la librería `cmath` obtenemos las funciones:

- `cmath.phase` pcalcula el argumento del complejo
- `cmath.polar()` pasa a forma polar
- `cmath.rect()` pasa a forma binómica.

Nota. En calculo numérico arrastramos errores. El número $1.2246...e-16$ claramente es un cero.

4 Tema 3: Strings en Python

4.1 Variable de tipo string

Un `string` es una cadena ordenada de caracteres. Una variable de tipo `string` es aquella que guarda un string. Se declara con comillas dobles o simples.

4.2 String literals

Los `string literals` son caracteres especiales. Algunos de los más utilizados son:

Código	Significado
`\\`	Backslash, `\`
`\'`	Comilla simple, `'`

Código	Significado
`\"`	Comilla doble, `"`
`\n`	Salto de línea
`\t`	Tabulación horizontal

Para más información ir a la documentación oficial.

4.3 Concatenación, repetición y visualización de strings

Algunos comandos útiles son:

- `<string> = <string 1> + <string 2>` permite concatenar o unir dos o más strings en uno solo.
- `<string> = <string> * 5` repite el string 5 veces (conmutativo)
- `print(<string>)` para imprimir el string en pantalla

Algunas notas:

1. Se recomienda utilizar un mismo criterio para añadir espacios en blanco a una variable string, al principio o al final.
2. Más adelante veremos como eliminar espacios en blanco sobrantes.

4.4 Funciones str, format, saltos de línea y tabulaciones

Algunos comandos útiles son:

- el uso de `str(<numeric type>)` permite concatenar "un tipo numérico" con otros strings.
- `"... {} ... {} ...".format(<string>, <numeric type>)` permite concatenar distintos tipos de datos

Ejemplos:

```
nombre = "Paco"
numero_hijos = 3
print("Mi abuelo se llama {} y tiene {} hijos".format(nombre,
 ↪  numero_hijos))
```

Nota. Lo anterior está `DEPRECATED` , mejor utilizar lo siguiente:

```
nombre = "Paco"
numero_hijos = 3
print(f"Mi abuelo se llama {nombre} y tiene {numero_hijos}
 ↪  hijos")
```

4.5 Substrings

Supongamos que definimos la variable `string = "Esto es un string"` .

Podemos acceder a los distintos substrings de `string` con la siguientes órdenes:

- `string[0]` accede al primer carácter, `string[1]` accede al segundo carácter, etc
- `string[-1]` accede al último carácter, `string[-2]` accede al penúltimo carácter, etc
- `string[2:5]` accede a los carácteres del tercero al quinto, es decir, posiciones 2, 3, 4
- `string[:5]` accede a los carácteres del primero al quinto
- `string[2:]` accede a los carácteres del tercero al último
- `string[-10:]` accede a los carácteres desde el décimo por atrás hasta el final (accede a los 10 últimos)
- `string[:-5]` accede a todos los carácteres desde el inicio hasta el quinto por atrás (elimina los 5 últimos)

4.6 Métodos para trabajar con strings

Supongamos que definimos la variable `string = "Esto es un string"` .

Tenemos los siguientes métodos sobre `string` :

- `string.lower()` transforma todos los caracteres de `string` en minúsculas.
- `string.upper()` transforma todos los caracteres de `string` en mayúsculas.
- `string.count(<substring>)` cuenta las veces que aparece el substring en `string` .
- `string.capitalize()` convierte a mayúscula el primer carácter de `string` .

- `string.casefold()` convierte a minúscula el primer carácter de `string` .
- `string.title()` convierte a mayúscula el primer carácter de cada palabra de `string` .
- `string.swapcase()` convierte las mayúsculas de `string` en minúsculas y viceversa.
- `string.replace(<old substring>, <new substring>)` reemplaza un substring por el otro.
- `string.split(<substring>)` rompe `string` cada vez que aparece el substring, eliminado este último (podemos indicar un espacio)
- `string.rstrip()` elimina los espacio sobrantes al final de `string` .
- `string.lstrip()` elimina los espacio sobrantes al principio de `string` .
- `string.strip()` elimina los espacio sobrantes al principio y al final de `string`
- `string.find(<substring>)` busca el substring en `string` y devuelve la posición de su primer carácter. Si el substring no existe, devuelve -1.
- `string.find(<substring>, 10)` idem anterior comenzando la búsqueda en la posición 10.
- `string.find(<substring>, 30, 40)` idem anterior comenzando la búsqueda en la posición 30 y terminando en la posición 40
- `string.index(<substring>)` , `string.index(<substring>, 10)` , `string.index(<substring>, 30, 40)` idem anteriores pero si el substring no existe, devuelve un error.

4.7 Funciones para trabajar con strings

Supongamos que definimos la variable `string = "Esto es un string"` . Algunas funciones interesantes:

- `len(string)` devuelve el número de caracteres (espacios incluídos) en `string`

- `<variable> = input()` sirve para pedir al usuario un dato a través de la consola y guardarlo en la variable.
- `<variable> = input("Esto es una etiqueta")` idem anterior añadiendo una etiqueta.

Algunos ejemplos de peticiones de datos por consola al usuario:

```
print("Introduce tu nombre: ")
name = input("Nombre: ")

print("Introduce tu edad: ")
age = int(input("Edad: "))

print("Introduce tu altura: ")
height = float(input("Altura (en m): "))

print("La edad de {} es {} y mide {}m".format(name, age, height))
```

Algunas notas sobre la práctica anterior:

- Se utiliza `int` para convertir a `numero entero` y `float` para convertir a `número en coma flotante` .
- Para los decimales hay que utilizar un punto `.` , si se utiliza una coma `,` da error.
- Como programador, hay que contemplar los errores que pueda cometer un usuario. Esto se verá más adelante.

5 Tema 4: operadores de decisión

5.1 Variables booleanas

Un **dato booleano** es aquel que solo puede tomar 2 valores: `True` o `False` . Una **variable lógica** es una variable que almacena un dato booleano.

Algunas notas:

- Python (a diferencia de R) solo admite los booleanos escritos de la forma indicada (en mayúscula la primera letra).
- Al ordenar `type(False)` se obtiene `bool` .

5.2 Tablas de verdad

Son conocidas las tablas de verdad de la Negación, la Conjunción y la Disyunción. Consultar cualquier libro de lógica.

5.3 Operadores lógicos en python

Algunas notas:

- El operador de negación es `not` .
- El operador de conjunción es `and` .
- El operador de distunción es `or` .
- Observar la utilización del paréntesis en el comando `A and (not B)`

5.4 Operadores de comparación

Los operadores de comparación de python son:

Operador	Significado
`>`	Estrictamente mayor
`>=`	Mayor o igual
`<`	Estrictamente menor
`<=`	Menor o igual
`==`	Igual
`!=`	Diferente

Algunas notas:

- `7 != 7.0` es False
- `7 != "7"` es True

5.5 Múltiples comparaciones simultáneas

Se combinan operadores lógicos y operadores de comparación.

5.6 Compararaciones de strings

Algunas notas:

- Se pueden comparar variables de tipo string.
- Se utiliza el orden lexicográfico, por ejemplo, `"Mallorca" < "Dubai"` nos devuelve `False` , porque la D aparece primero que la M en el abecedario.

5.7 Métodos booleanos de strings

Supogamos que definimos la variable `sting = "Esto es un string` . Tenemos los siguientes métodos sobre `string` :

- `string.startswith(<substring>)` devuelve `True` si comienza con el substring indicado.
- `string.endswith(<substring>)` devuelve `True` si termina con el substring indicado.
- `string.isalnum()` devuelve `True` si todos los caracteres de `string` son alfanuméricos.
- `string.isalpha()` devuelve `True` si todos los caracteres de `string` son del alfabeto (A-Z, a-z).
- `string.isdigit()` devuelve `True` si todos los caracteres de `string` son dígitos.
- `string.isspace()` devuelve `True` si todos los caracteres de `string` son espacios en blanco.
- `string.islower()` devuelve `True` si todos los caracteres de `string` están en minúscula.
- `string.isupper()` devuelve `True` si todos los caracteres de `string` están en mayúscula.
- `string.istitle()` devuelve `True` si todas las palabras de `string` empiezan en mayúscula y el resto de letras están en minúscula.

Algunas notas:

- `"365".isalnum()` devuelve `True` .
- `"Soy Roberto".isalnum()` devuelve `False` por que el espacio no se considera un caracter alfanumérico.

5.8 Operadores de decisión

Algunas estructuras de control con las siguientes:

```
# (if) statement
if <test expression>:
    <body of statement>
```

```
# (if ... else) statement
if <test expression>:
    <body of statement>
else:
    <body of statement>
```

```
# (if ... elif) statement
# Cada elif se evalua sólamente si la condición anterior es falsa
if <test expression>:
    <body of statement>
elif <test expression>:
    <body of statement>
elif <test expression>:
    <body of statement>
# También se puede terminar con else
# si todo lo anterior es falso
else:
    <body of statement>
```

Tenemos una versión corta del (if)

```
# short hand (if)
if <test expression>: <statement>
```

Y también tenemos el llamado `operador ternario` . Esta estructura es exclusiva de python:

```
# short hand (if ... else)
# El primer statement se ejecuta si el test es verdadero
# En caso contrario, se ejecuta el segundo statement
<statement> if <test expression> else <statement>
```

Algunas notas:

- Hacer un buen uso de la identación (o bien tabular, o bien colocar 4 espacios).
- Los operadores de decisión se pueden anidar.
- La función `chr()` nos da el código ASCCI (el número) de los caracteres recogidos en este código. Los número del codigo del 65 al 90 se corresponden con las letras mayúsculas de la A a la Z.

6 Tema 5: operadores de iteración

6.1 El bucle (while)

El bucle while se ejecuta miestras se cumple una determinada condición.

```
# (while) loop
# El bucle while se ejecuta miestras se cumple una condición
# Suele acompañarse de un índice contador i = 0 que va creciendo
# Pueden aparecer también contadores count = 0
while <condition>:
    <body of the loop>
```

6.2 El comando break

El comando `break` detiene la ejecución del bucle y salta a la primera línea del programa tras el bucle. Se utiliza dentro del cuerpo del bucle.

6.3 El bucle (while) con (else)

Se puede añadir un (else) después de un bucle (while).

```
while <condition>:
    <body of the loop>
else:
    <statement>
```

6.4 El bucle (for)

El bucle for se ejecuta un número predeterminado de veces. Se recorre una clave.

Algunas notas previas:

- En python, un iterable es un tipo de objeto que se puede iterar, es decir, que puede dar sus elementos uno por uno.
- Veremos varios tipo de iterables como las listas o los rangos. Una secuencia es un tipo de iterable.

```
# (for) loop
# El bucle for se ejecuta un número predeterminado de veces.
# <variable> no tiene por qué estar definida
for <variable> in <iterable>:
    <body of the loop>
```

6.5 La función range

La función `range()` nos devuelve una sucesión (sequence) de números enteros, en particular, una sucesión aritmética. Tiene tres posibles argumentos:

- `start` es el número inicial de la sucesión. Si se omite se considera el 0.
- `stop` es el numero final de la sucesión.
- `step` es la diferencia entre términos. Si se omite se considera el 1.

Por ejemplo:

- `range(6)` nos da 0, 1, … , 5
- `range(2,6)` nos da 2, 3, … , 5
- `range(1,6,2)` nos da 1, 3, 5
- `range(10,3,-1)` nos da 10, 9, … , 2
- `type(range(6))` nos devuelve `range` .

6.6 El comando continue

El comando `continue` detiene la iteración actual y salta a la siguiente iteración del bucle sin salir de este.

Por ejemplo, el siguiente código imprime todos los números entre el 0 y el 100 que no son divisibles por 2 ni por 5.

```
for i in range(101):
  if i % 2 == 0 or i % 5 == 0:
    continue
  print(i)
```

6.7 Bucles anidados

Los bucles se pueden anidar. Veamos el famoso ejemplo de las tablas de multiplicar:

```
for i in range(1, 4):
  print("\nTabla de multiplicar del {}".format(i))
  for j in range(1, 6):
    print("{} x {} = {}".format(i, j, i * j))
```

7 Tema 6: estructura de datos - listas

7.1 Introducción a las listas

Una lista es un vector (importa el orden) cuyos elementos son tipos. Los elementos se pueden modificar.

Las listas se definen con corchetes y comas:

- `l = ["Juan", 2, 3.5, True]` define la lista con los elementos indicado.
- `l = []` define una lista vacía.

Algunas comandos fundamentales:

- `type(<list>)` nos devuelve `list` , que es el nombre de tipo de dato.
- `len(<list>)` nos devuelve la longitud de la lista.

Algunas notas:

- El primer indice de una lista es el 0.
- Lo que en python se llama listas, en otros lenguajes se llama arrays. En python, los arrays serán otra cosa.

7.2 Elementos de una lista

Sintaxis para accder a los distintos elementos de una lista:

- `<list>[2]` accede al tercer elemento de la lista.
- `<list>[-1]` accede al último elemento de la lista.
- `<list>[-1]` accede al último elemento de la lista.
- `<list>[2:4]` accede a los elementos del tercero al cuarto (posiciones 2 y 3).
- `<list>[:5]` accede a los elementos del primero al cuarto.
- `<list>[3:]` accede a los elementos del tercero al último.

Obsevar que el índice indicado a la derecha de los `:` nunca es incluído.

Podemos modificar los elementos de una lista:

- `<list>[2] = <string>` añade el string en la posición 2 de la lista (será el tercer elemento).
- `<list>[0] = <int>` añade el número entero en la primera posición de la lista.

- `<list>.append(<string>)` añade el string en la última posición de la lista.
- `<list>.insert(<position>, <string>)` añade el string en la posición de la lista indicada con un número.

7.3 Bucle con listas

Es muy común tener que recorrer en bucle los elementos de una lista.

Algunos ejemplos muy utilizados:

```
names = ["Mario", "Cristian", "Juan"]

for i in range(len(names)):
  print(names[i])
```

```
names = ["Mario", "Cristian", "Juan"]

for name in names:
  print(name)
```

Algunas notas:

- Los dos códigos anteriores hacen la misma tarea.
- El primer código es más complejo, pero tiene la ventaja de pode utilizar el índice i de cada elemento.

7.4 Concatenación y repetición de listas

Se puede concatenar y repetir listas.

Comandos utilizados:

- `<list> = <list> + <list>` concatena las dos listas unidas por el signo `+` .
- `<list> = <list> * n` repite `n` veces los elementos de la lista y forma con ellos una nueva lista.

7.5 Métodos con listas

Tenemos varios métodos para listas:

- `<list>.count(<element>)` cuenta el número de veces que aparece el elemento en la lista
- `<list>.extend(<iterable>)` extiende la lista agregando al final el interable indicado como parámetro.

- `<list>.index(<element>)` devuelve la posicón del elemento en la lista.
- `<list>.pop()` devuelve el último elemento en la lista y lo borra de la misma.
- `<list>.remove(<element>)` borra la primera aparición del elemento en la lista.
- `<list>.reverse()` devuelve la lista invirtiendo el orden.
- `<list>.sort()` devuelve la lista con la relación de orden creciente.
- `<list>.sort(reverse = True)` devuelve la lista con la relación de orden decreciente.

7.6 Conversión a listas

Python permite convertir cualquier objeto iterable en una lista utilizando la función `list(<iterable>)` .

Un ejemplo imprimido:

```
print(list(range(0, 100, 10)))
```

7.7 Listas anidadas

Podemos tener listas dentro de listas. ¡Hola matrices!

Un ejemplo:

```
l = [["Cristina", "Juan", "Marta"],
    ["Pedro", [1, 2, 3, 4, 5], 20],
    10]
```

Notar que los elementos están separados por intros para una mejor visualización.

Algunos comandos útiles:

- `<nested list>[0][2]` accede al tercer elemento de la primera lista anidada. En otras palabras, salta al primer elemento anidado y, desde aquí dentro, salta al tercer elemento.
- `<nested list>[1][0][4]` salta a la posición 1, aquí dentro salta a la posición 0, aquí dentro salta a la posición 4.

7.8 Matrices con listas

Algunos conceptos:

- Una matriz es un tipo de lista muy utilizado.

- Se trata de una lista de `m` listas anidadas que tienen el mismo número `n` de elementos.
- No tienen por contener datos de tipo numérico.
- Se recuerda que en python, los índices comienzan en 0.

Algunos ejemplos:

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

print(matrix[0][2])

for row in matrix:
  print(row)
```

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

for row in matrix:
  for element in row:
    print(element)
```

```
# m y n es el número de filas y columnas respectivamente
m = len(matrix)
n = len(matrix[0])

for i in range(m):
  for j in range(n):
    print(matrix[i][j], end = " ")
  print("")

# la alternatva es
for row in matrix:
  for element in row:
    print(element, end = " ")
  print("")
```

Algunas notas:

- Por defecto, tras ejecutar la función `print()` se produce un salto de línea.
- El parámetro `end` de la función `print()` se utiliza cuando queremos hacer otras cosa inmediatamente después de la ejecución de `print()` .

En este caso, hemos indicado que queremos un espacio en blanco.
- Hay que repasar el álgebra de matrices.
- En el notebook del curso se pueden ver los códigos para la suma y el producto de matrices utilizando un bucle for.

7.9 Matrices con numpy

La librería numpy nos permite trabajar con matrices de una forma mucho más sencilla. Para cargar la librería ordenaremos

```
import numpy as np
```

Algunos métodos interesantes son las siguientes:

- `np.empty((2,3))` crea una matriz "cualquiera" de dimensiones 2x3. Se puede añadir un parámetro adicional indicando el tipo de dato para sus elementos.
- `np.empty_like(<matrix>)` crea una matriz "cualquiera" de dimensiones las dimensiones de la matriz indicada.
- `np.zeros((3,5))` crea una matriz nula de dimensiones 3x5.
- `np.zeros_like(<matrix>)` crea una matriz nula de dimensiones las dimensiones de la matriz indicada.
- `np.zeros((3,5))` crea una matriz de unos de dimensiones 3x5.
- `np.zeros_like(<matrix>)` crea una matriz de unos de dimensiones las dimensiones de la matriz indicada.
- `np.matrix(<list of lists>)` crea una matriz a partir de la lista de listas indicada.
- `<matrix>[i,j]` accede al elemento de posición (i,j). Tener en cuenta que los índices en python comienzan en cero.

Podemos pedir una matriz al usuario con el siguiente código:

```
import numpy as np
n = int(input("Introduce el número de filas: "))
m = int(input("Introduce el número de columnas: "))

A = np.empty((n,m))

for i in range(n):
    for j in range(m):
        A[i,j] = float(input("Introduce el elemento ({},{}):
 ↪  ".format(i+1,j+1)))
```

```
print(A)
```

Respecto a las dimensiones y las operaciones, tenemos los siguientes comandos:

- `<matrix>.shape` nos devuelve las dimensiones de la matriz.
- `<matrix> + <matrix>` nos devuelve la matriz suma.
- `<matrix>.dot(<matrix>)` nos devuelve la matriz producto.

Con `numpy` , para mostrar una matriz en forma de tabla basta con hacer uso de la función `print()` .