



**Universidad Nacional  
Autónoma de México**  
Facultad de Ingeniería



# **Laboratorios de computación salas A y B**

**PROFESOR:** M.I. Marco Antonio Martínez Quintana

**ASIGNATURA:** Estructura de Datos y Algoritmos I

**GRUPO:** 17

**NO DE PRÁCTICA:** 10

**NOMBRE:** Reyes Mendoza Miriam Guadalupe

**SEMESTRE:** 2020-2

**FECHA DE ENTREGA:** 14/04/2020

**OBSERVACIONES:**



**CALIFICACIÓN:**

# INTRODUCCIÓN A PYTHON (II)

## OBJETIVO

Aplicar las bases del lenguaje de programación Python en el ambiente de Jupyter notebook.

## INTRODUCCIÓN

### ¿QUÉ ES PYTHON?

Python es un lenguaje de programación orientado a objetos, es decir, un idioma que la computadora conoce y nos sirve para ordenarle distintas acciones.

Pero Python no es sólo un lenguaje de programación, sino además es *interpretado*, esto quiere decir que la computadora por sí sola no conoce el lenguaje y lo comprende, si no que necesita un *intérprete*. Esto sería similar a visitar un país del cual no conocemos ni una palabra de la lengua nativa, pero viajamos con una persona que conoce nuestra lengua y la de ese país, y nos hace de traductor para que logremos entendernos con las demás personas.

El mundo de la programación se divide en dos clases de lenguajes: ***interpretados*** y ***compilados***. La diferencia es que, en un lenguaje compilado, el programador escribe el código en el lenguaje de su preferencia, siempre y cuando sea un lenguaje que se compile, y luego, por medio del compilador, ese código se “traduce” al lenguaje ensamblador que entiende la computadora. La ventaja de esto es que el código ya se encuentra totalmente traducido y su ejecución es veloz.

En cambio, con un lenguaje interpretado el intérprete traduce el código para que la computadora lo comprenda a medida que se va ejecutando, y esto puede llegar a resultar un poco más lento en la ejecución que un código compilado.

Una de las principales ventajas es que se puede ir probando el código a medida que lo vamos escribiendo, un intérprete no sabe ni le importa cuando termine el código para hacer su trabajo. En cambio, un compilador realiza su tarea hasta que encuentra la instrucción de *fin*, por lo que, si el código que queremos compilar no está completo, no lograremos compilar y ejecutar nuestro programa. Esta diferencia hace que programar en un lenguaje interpretado sea mucho más dinámico, y de esta forma se optimiza el tiempo de programación y depuración del código escrito.

## ESTRUCTURAS DE CONTROL

Un bloque condicional o estructura de control, es un bloque de código que se ejecuta sí y solamente si la instrucción de control que lo contiene ve cumplida su condición cuando uno de los bloques se ejecuta en seguimiento de las instrucciones condicionales se termina.

Nada evita que un bloque se encuentre dentro de una función de un método de un módulo, o incluso de una clase. Una estructura de control Puede ubicarse en bucle o incluso dentro de otra estructuras de control.

Una condición es la evaluación de una expresión que transforma de forma determinista dicha expresión por uno de los dos valores *True* o *False*.

Las condiciones utilizan con frecuencia operadores de comparación, aunque no es la única forma de crearlas. Es posible utilizar cualquier objeto y evaluarlo en una función de los principios de evaluación booleana.

Un operador de comparación devuelve simplemente un valor booleano, que es Asimismo un objeto.

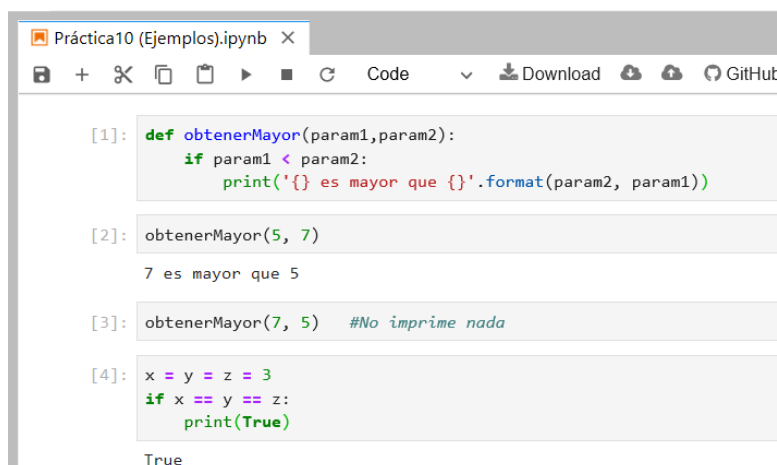
La instrucción *is* permite realizar una comparación sobre la identidad del objeto, y no sobre su valor. La instrucción *not* es, simplemente, una instrucción que interviene una condición, cuál sea.

## SELECTIVAS

### IF

La instrucción *if* es una palabra reservada el Python usada para ejecutar un bloque de sentencias si se cumple determinada condición representada por una expresión booleana.

En el caso del *if* simple se evalúa la expresión booleana y si el resultado es *True* se ejecuta el bloque de instrucciones asociado con él (su cuerpo) siguiéndose a continuación el flujo del programa. Si el resultado es *False* no ejecutaría el cuerpo del *if* y seguiría directamente con el flujo del programa.



```
Práctica10 (Ejemplos).ipynb X
[1]: def obtenerMayor(param1,param2):
    if param1 < param2:
        print('{} es mayor que {}'.format(param2, param1))

[2]: obtenerMayor(5, 7)
7 es mayor que 5

[3]: obtenerMayor(7, 5) #No imprime nada

[4]: x = y = z = 3
    if x == y == z:
        print(True)
True
```

## IF - ELSE

La instrucción combinada *if-else* añade al *if* la posibilidad de ejecutar un bloque de instrucciones cuando la condición es falsa.

Se evalúa la expresión lógica; si el resultado es *True* ejecuta un bloque de instrucciones y en caso contrario será otro bloque. La expresión booleana siempre es verdadero o falsa, por lo que está asegurada la ejecución de alguno de los dos bloques.

```
Práctica10 (Ejemplos).ipynb X
[5]: def obtenerMayorv2(param1,param2):
      if param1 < param2:
          return param2
      else:
          return param1

[6]: print ("El mayor es {}".format( obtenerMayorv2(4, 20) ))
      El mayor es 20

[7]: print ("El mayor es {}".format( obtenerMayorv2(11, 6) ))
      El mayor es 11

[8]: def obtenerMayor_idiom(param1,param2):
      #La variable valor va a tener el valor de param2 is el if es verdadero
      #de lo contrario tendra el valor de param1
      valor = param2 if (param1 < param2) else param1
      return valor

[9]: print ("El mayor es {}".format( obtenerMayor_idiom(11, 6) ))
      El mayor es 11
```

## IF – ELIF - ELSE

Mediante *if-elif-else* podemos tener varias expresiones booleanas, que se evaluarán solamente si las anteriores han devuelto *Falso*, ya que en el momento en que alguna devuelve *True* se ejecuta el bloque de instrucciones asociado a ella y se sale de la instrucción combinada *if-elif-else*.

En cuanto una expresión booleana es verdadera, tras ejecutar las instrucciones asociadas a ella, se seguirá el flujo del programa. Si ninguna de ellas es verdadera, se ejecuta el bloque de instrucciones asociado al *else*.

```
Práctica10 (Ejemplos).ipynb X
[10]: def numeros(num):
      if num==1:
          print ("tu numero es 1")
      elif num==2:
          print ("el numero es 2")
      elif num==3:
          print ("el numero es 3")
      elif num==4:
          print ("el numero es 4")
      else:
          print ("no hay opcion")

[11]: numeros(2)
      el numero es 2

[12]: numeros(5)
      no hay opcion

[13]: def numeros_idiom(num):
      #La tupla tiene las opciones válidas
      if num in (1,2,3,4):
          print("tu numero es {}".format(num))
      else:
          print ("{} no es una opcion".format(num))

[14]: numeros_idiom(2)
      tu numero es 2

[15]: numeros_idiom(5)
      5 no es una opcion
```

```
Práctica10 (Ejemplos).ipynb X
[16]: def obtenerMasGrande(a, b, c):
      if a > b:
          if a > c:
              return a
          else:
              return c
      else:
          if b > c:
              return b
          else:
              return c

[17]: print ("El mas grande es {}".format(obtenerMasGrande(7,13,1) ))

El mas grande es 13
```

## REPETITIVAS

En nuestros programas necesitaremos ejecutar de forma reiterada (en bucle) bloques de código. Habrá casos en los que sabremos el número exacto de repeticiones, y otros en los que conoceremos la condición de salida del bucle. Python nos proporcionan instrucciones para implementar bucles:

- **For**, donde el bucle es controlado mediante la variable contador.
- **While**, donde el bucle lo controla una condición lógica.

Con *for* tendremos, en los casos en que se pueda usar, un elemento más simple y cómodo, mientras que *while* nos permitirá más versatilidad.

### WHILE

Mediante *while* entraremos en un bucle si se cumple una determinada condición lógica, y nos mantendremos en él mientras la citada condición sea verdadera.

La condición lógica de mantenimiento en el bucle es una expresión booleana que devolverá *True* o *False* computarse. En el primer caso de que resultara el bloque (lo que se denomina el cuerpo del *while*), tras lo cual volverá a comprobar la condición lógica, y así sucesivamente hasta que el resultado sea *False*, se salga del bucle *while*, y continúe el programa en la siguiente instrucción. Es importantísimo que en el cuerpo del *while* se modifiquen elementos que haga que la condición de mantenimiento en bucle sea falsa, dado que de lo contrario podríamos entrar en un bucle infinito, algo que “colgaría” nuestro programa.

```
Práctica10 (Ejemplos).ipynb X
[18]: #Ejemplo 1
      def cuenta(limite):
          i = limite
          while True:
              print(i)
              i = i - 1
              if i == 0:
                  break # Rompiendo el ciclo

[19]: cuenta(10)

10
9
8
7
6
5
4
3
2
1
```

```
Práctica10 (Ejemplos).ipynb X
[20]: #Ejemplo 2
      def factorial(n):
          i = 2
          tmp = 1
          while i <= n:
              tmp = tmp * i
              i = i + 1
          return tmp

[21]: print (factorial(4))

24

[22]: print (factorial(6))

720
```

## FOR

Mediante *for* creamos un bucle controlado por una variable contador que recorre la serie de valores que contiene una secuencia.

En ella se repite en ejecución el bloque de instrucciones (lo que se denomina el cuerpo del *for*) mientras el va pasando por los valores contenidos en la secuencia. Cuando se han recorrido todos, se termina el bucle.

Los 3 parámetros entrada para la primera sintaxis son:

- **INICIO:** Entero desde el cual comenzará la secuencia.
- **FINAL:** Entero posterior al que finalizara la secuencia, es decir, está llegará hasta final – 1
- **PASO:** Entero que determina el incremento (o decremento puede ser un número negativo) entre los valores de la secuencia.

## ITERACIÓN DE LISTAS

```
Práctica10 (Ejemplos).ipynb X
+ ✕ 📄 ▶ ⌂ Code ⌵ Download 📁 🌐 GitHub 🌐 Binder

[23]: for x in [1,2,3,4,5]:
      print(x)

1
2
3
4
5

[24]: #La función range() sirve para generar una lista
      for x in range(5): #este caso es equivalente a range(0,5)
          print(x)

0
1
2
3
4

[25]: #También se puede inicializar desde números negativos
      for x in range(-5,2):
          print(x)

-5
-4
-3
-2
-1
0
1

[26]: for num in ["uno", "dos", "tres", "cuatro"]:
      print(num)

uno
dos
tres
cuatro
```

## ITERACIÓN DE DICCIONARIOS

```
Práctica10 (Ejemplos).ipynb X
+ ✕ 📄 ▶ ⌂ Code ⌵ Download 📁 🌐 GitHub 🌐 Binder

[27]: #Creando un diccionario
      elementos = { 'hidrogeno': 1, 'helio': 2, 'carbon': 6 }

      for llave, valor in elementos.items():
          print(llave, " = ", valor)

hidrogeno = 1
helio = 2
carbon = 6

[28]: #Obteniendo sólo las llaves
      for llave in elementos.keys():
          print(llave)

hidrogeno
helio
carbon

[29]: #Obteniendo sólo los valores
      for valor in elementos.values():
          print(valor)

1
2
6

[30]: #Si se necesita iterar utilizando un índice
      for idx, x in enumerate(elementos):
          print("El índice es: {} y el elemento: {}".format(idx, x))

El índice es: 0 y el elemento: hidrogeno
El índice es: 1 y el elemento: helio
El índice es: 2 y el elemento: carbon

Práctica10 (Ejemplos).ipynb X
+ ✕ 📄 ▶ ⌂ Code ⌵ Download 📁 🌐 GitHub 🌐 Binder

[31]: def cuenta_idiom(limite):
      for i in range(limite, 0, -1):
          print(i)
      else: #Corresponde al for, NO al IF
          print("Cuenta finalizada")

[32]: cuenta_idiom(5)

5
4
3
2
1
Cuenta finalizada

[33]: #Se rompe el ciclo y la sentencia else del for no se ejecuta
      def cuenta_idiomv2(limite):
          for i in range(limite, 0, -1):
              print(i)
              if i == 3:
                  break #Se rompe el ciclo
          else: #Corresponde al FOR, NO al IF
              print("Cuenta finalizada")

[34]: cuenta_idiomv2(5)

5
4
3
```

## BIBLIOTECAS

Todas las funcionalidades de Python son proporcionadas a través de bibliotecas que se encuentran en la colección de The Python Standard Library, la mayoría de estas bibliotecas son multiplataforma. Las bibliotecas más usadas son:

- **NumPy (Numerical Python):** Es una de las bibliotecas más populares de Python, es usado para realizar operaciones con vectores o matrices de manera eficiente. Contiene funciones de álgebra lineal transformadas de Fourier, generación de números aleatorios e integración con Fortran, C y C++.
- **SciPy (Numerical Python):** Es una biblioteca que hace uso de Numpy y es utilizada para hacer operaciones más avanzadas como transformadas de Fourier, Álgebra Lineal, Optimización, entre otras.
- **Matplotlib:** Es una biblioteca usada para generar una variedad de graficad en 2D y 3D, done una de las configuraciones de la gráfica es programable. Se puede usar comando de Latex para agregar ecuaciones matemáticas a las gráficas.
- **Scikit Learn (Machine Learning):** Esta biblioteca basada en los anteriores y contiene algoritmos de aprendizaje de máquina, reconocimiento de patrones y estadísticas para realizar clasificación, regresión, clustering, entre otros.
- **Pandas (Manipulación de datos):** Esta biblioteca es utilizada para manipulación de datos, contiene estructuras de datos llamadas data frames que se asemejan a las hojas de cálculo y a los cuales se le puede aplicar una gran cantidad de funciones.

```
Práctica10 (Ejemplos).ipynb ×  
+ ✕ 📄 ▶ ⏮ ⏭ Code ▾ 📁 Download 🔒 GitHub 🔍 Binder No Kernel  
  
[35]: #Para utilizar una biblioteca, ésta se debe de importar  
import math  
  
x = math.cos(math.pi)  
  
print(x)  
  
-1.0  
  
[36]: #También se pueden importar todas las funciones de la bibliotecas, de esta manera no se tiene que usar el prefijo  
#de la biblioteca, que en el ejemplo anterior fue math  
from math import *  
  
x = cos(pi) #No se utiliza el prefijo math  
  
print(x)  
  
-1.0  
  
[37]: #Otra manera es importar sólo las funciones que se necesitan  
from math import cos, pi  
  
x = cos(pi)  
  
print(x)  
  
-1.0  
  
[38]: #Una vez que la biblioteca está importada, se pueden conocer las funciones que éste contiene  
print(dir(math))  
  
['_doc_', '_file_', '_loader_', '_name_', '_package_', '_spec_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',  
'copyrig', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'exp1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'gamma',  
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isin', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',  
'radians', 'random', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

```
Práctica10 (Ejemplos).ipynb X
+ ✂ 📄 ▶ ■ 🔄 Code ⌵ ⬇ Download 📁 📁 📁 GitHub 🔄 Binder

[39]: #Para conocer cómo utilizar las funciones, se puede utilizar la función help
      help(math.log)

      Help on built-in function log in module math:

      log(...)
          log(x, [base=math.e])
          Return the logarithm of x to the given base.

          If the base not specified, returns the natural logarithm (base e) of x.

[40]: #Se puede definir un alias para llamar a las funciones que tiene la biblioteca math.
      #Esta es la forma más recomendada para importar módulos, ya que de esta manera se sabe de qué módulo proviene la función.
      import math as ma

      x = ma.cos(ma.pi)

      print(x)

-1.0
```

## NUMPY

```
Untitled1.ipynb
+ ✂ 📄 ▶ ■ 🔄 Code ⌵ ⬇ Download 📁 📁 📁 GitHub 🔄 Binder

[2]: #Esta línea se utiliza para que las gráficas que se creen queden embebidas dentro de la notebook.
      %pylab inline

      Populating the interactive namespace from numpy and matplotlib

[3]: #Los comentarios en Python empiezan con el signo #
      #Cargar módulo de numpy
      import numpy as np
```

## CREAR VECTORES Y MATRICES

```
Untitled1.ipynb
+ ✂ 📄 ▶ ■ 🔄 Code ⌵ ⬇ Download 📁 📁 📁

[4]: vector = array([1,2,3,4,5,6])
      matriz = array([[1, 2], [3, 4], [5, 6]])

[5]: #imprimir contenido y tamaño del vector y la matriz
      print(vector)
      print() #Línea vacía
      print(matriz)
      print()
      print("Tamaño del vector {}".format(vector.shape))
      print("Tamaño de la matriz {}".format(matriz.shape))

[1 2 3 4 5 6]

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

Tamaño del vector (6,)
Tamaño de la matriz (3, 2)
```



```
Untitled1.ipynb
[6]: #Los valores del tamaño del vector o matrices se pueden acceder de la siguiente manera

v_elem = vector.size

#Forma 1
m_reng, m_column = matriz.shape

#Forma 2
m_reng = matriz.shape[0]
m_column = matriz.shape[1]

#Forma 3
m_reng, m_column = matriz.shape[0], matriz.shape[1]

print("El vector tiene {} elementos".format(v_elem))
print("La matriz es de {},{} con un total de {} elementos".format(m_reng, m_column, matriz.size))

El vector tiene 6 elementos
La matriz es de 3,2 con un total de 6 elementos
```

La diferencia entre shape y size es que shape indica el número de renglones y columnas que componen al vector o matriz. Mientras size indica el total de elementos.

```
Untitled1.ipynb
[7]: #También se pueden crear vectores y matrices de la siguiente forma
v_ceros = zeros(4)
v_unos = ones(4)
v_random = np.random.rand(8) #Se generan números aleatorios

m_ceros = zeros([4,3])
m_unos = ones([4,5])
m_random = np.random.rand(6,4) #Se generan números aleatorios
```

```
Untitled1.ipynb
[8]: print("Imprimiendo los vectores")
print(v_ceros)
print(v_unos)
print(v_random)
print()
print("Imprimiendo las matrices")
print(m_ceros)
print(m_unos)
print(m_random)

Imprimiendo los vectores
[0. 0. 0. 0.]
[1. 1. 1. 1.]
[0.09695719 0.30591264 0.00982457 0.17927148 0.36213727 0.08967954
 0.62206004 0.08666265]

Imprimiendo las matrices
[[0. 0. 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.]
 [1. 1. 1. 1. 1.]]
[[0.32842397 0.54033051 0.81927942 0.19438556]
 [0.46272606 0.52930856 0.83382422 0.26766381]
 [0.26692025 0.82713967 0.75349851 0.69599504]
 [0.52776521 0.50599707 0.55998161 0.58623193]
 [0.41167668 0.21512933 0.06526197 0.34581586]
 [0.26232902 0.19335971 0.39007851 0.68570492]]
```

## ACCEDER A LOS ELEMNTOS DE LOS VECTORES Y MATRICES

```
Untitled1.ipynb
[9]: # Los índices de los vectores y matrices empiezan en cero
print(v_random[1])
print(v_random[3]) #EL índice 3 no existe, por lo que marca un error al tratar de acceder a él
0.30591263899714316
0.17927148460455156

[10]: #Los índices de las matrices están organizados de la forma [renglones, columnas]
print("Renglo 1 {}".format(m_random[1,:]))
Renglo 1 [0.46272606 0.52930856 0.83382422 0.26766381]
```

## RENGLONES, COLUMNAS Y ELEMENTOS

```
Untitled1.ipynb
[10]: #Los índices de las matrices están organizados de la forma [renglones, columnas]
print("Renglo 1 {}".format(m_random[1,:]))
Renglo 1 [0.46272606 0.52930856 0.83382422 0.26766381]

[11]: print("Columna 3 {}".format(m_random[:,3]))
Columna 3 [0.19438556 0.26766381 0.69599504 0.58623193 0.34581586 0.68570492]
```

## CONJUNTOS DE RENGONES

```
Untitled1.ipynb
[12]: print("Primero dos renglones {}".format(m_random[:2]))
Primero dos renglones [[0.32842397 0.54033051 0.81927942 0.19438556]
[0.46272606 0.52930856 0.83382422 0.26766381]]

[13]: print("Del renglón 3 en adelante {}".format(m_random[3:]))
Del renglón 3 en adelante [[0.52776521 0.50599707 0.55998161 0.58623193]
[0.41167668 0.21512933 0.06526197 0.34581586]
[0.26232902 0.19335971 0.39007851 0.68570492]]

[14]: print("Últimos dos renglones {}".format(m_random[-2:]))
Últimos dos renglones [[0.41167668 0.21512933 0.06526197 0.34581586]
[0.26232902 0.19335971 0.39007851 0.68570492]]

[15]: print("Últimos dos renglones {}".format(m_random[-2:]))
File "<ipython-input-15-9ae9467d4c3d>", line 1
print("Últimos dos renglones {}".format(m_random[-2:]))
                                         ^
SyntaxError: invalid syntax
```

## SUBCONJUNTOS

```
Untitled1.ipynb
[16]: print("Original: {}".format(v_random))
#Se hace un sub-vector de 3 elementos empezando por el que está en el índice 3 y hasta el 5.
#Como se observa, el primer índice la posición del primer elemento que se va a tomar, mientras que el segundo
#índice indica que se va a tomar hasta el elemento anterior a ese índice.

sub_v = v_random[3:6]
print("sub_1 del 3-5: {}".format(sub_v))

sub_v2 = v_random[5:9]
print("sub_2: del 5-8 {}".format(sub_v2))

Original: [0.09695719 0.30591264 0.00982457 0.17927148 0.36213727 0.08967954
0.62206004 0.08666265]
sub_1 del 3-5: [0.17927148 0.36213727 0.08967954]
sub_2: del 5-8 [0.08967954 0.62206004 0.08666265]

[17]: print("Original")
print(m_random)
print("\n Sub-conjunto [1-2-3, 1-2]")
print(m_random[1:4,1:3])

Original
[[0.32842397 0.54033051 0.81927942 0.19438556]
[0.46272606 0.52930856 0.83382422 0.26766381]
[0.26692025 0.82713967 0.75349851 0.69599504]
[0.52776521 0.50599707 0.55998161 0.58623193]
[0.41167668 0.21512933 0.06526197 0.34581586]
[0.26232902 0.19335971 0.39007851 0.68570492]]

Sub-conjunto [1-2-3, 1-2]
[[0.52930856 0.83382422]
[0.82713967 0.75349851]
[0.50599707 0.55998161]]
```

## CICLOS

```
Untitled1.ipynb
[18]: #Los arreglos y vectores también se pueden recorrer mediante ciclos
#Recorriendo la matriz usando índices
for i in range(0, m_random.shape[0]):
    print(m_random[i,:]) #Se imprime el renglón completo
    for j in range(0, m_random.shape[1]):
        print(m_random[i,j]) #Se imprime el elemento de cada renglón

[0.32842397 0.54033051 0.81927942 0.19438556]
0.32842397362219655
0.5403305063226352
0.8192794211852154
0.1943855644670608
[0.46272606 0.52930856 0.83382422 0.26766381]
0.4627260583074576
0.5293085610659869
0.833824219711227
0.267663806107727
[0.26692025 0.82713967 0.75349851 0.69599504]
0.2669202456441524
0.8271396743590895
0.7534985057776606
0.6959950388268975
[0.52776521 0.50599707 0.55998161 0.58623193]
0.5277652112401842
0.505997074558485
0.5599816127458772
0.5862319323398693
[0.41167668 0.21512933 0.06526197 0.34581586]
0.41167668065754026
0.21512933249424912
0.06526197015197654
0.3458158630959598
[0.26232902 0.19335971 0.39007851 0.68570492]
0.26232902289231397
0.19335971444818245
0.39007851102484314
0.6857049162536234
```

## GUARDAR Y CARGAR VARIABLES

```
Untitled1.ipynb
[19]: print("Original:")
print(m_random)

Original:
[[0.32842397 0.54033051 0.81927942 0.19438556]
 [0.46272606 0.52930856 0.83382422 0.26766381]
 [0.26692025 0.82713967 0.75349851 0.69599504]
 [0.52776521 0.50599707 0.55998161 0.58623193]
 [0.41167668 0.21512933 0.06526197 0.34581586]
 [0.26232902 0.19335971 0.39007851 0.68570492]]

[20]: #Guardando la matriz m_random en el archivo m_random.csv usando como delimitador ','
np.savetxt("m_random.csv",m_random, delimiter=",")

[21]: #Cargando la matriz guardada en m_random.csv
m_load = np.loadtxt("m_random.csv",delimiter=",")

[22]: print("Matriz guardada y cargada:")
print(m_load)

Matriz guardada y cargada:
[[0.32842397 0.54033051 0.81927942 0.19438556]
 [0.46272606 0.52930856 0.83382422 0.26766381]
 [0.26692025 0.82713967 0.75349851 0.69599504]
 [0.52776521 0.50599707 0.55998161 0.58623193]
 [0.41167668 0.21512933 0.06526197 0.34581586]
 [0.26232902 0.19335971 0.39007851 0.68570492]]
```

## OPERACIONES CON MATRICES

```
Untitled1.ipynb
[23]: #Simplemente se multiplica la matriz por el escalar deseado
resultado = m_unos * 9
print(resultado)

[[9. 9. 9. 9.]
 [9. 9. 9. 9.]
 [9. 9. 9. 9.]
 [9. 9. 9. 9.]]

[24]: #Para hacer la transpuesta de una matriz, sólo se utiliza el operador T a la matriz
result_T = resultado.T
print(result_T)

[[9. 9. 9. 9.]
 [9. 9. 9. 9.]
 [9. 9. 9. 9.]
 [9. 9. 9. 9.]
 [9. 9. 9. 9.]]

[25]: #Para la multiplicación de matrices se necesitan que el índice las columnas de la primera matriz coincida
#con el índice de renglones de la segunda matriz
print("Índices de las matrices:")
print(m_unos.shape)
print(result_T.shape)

#Para realizar la operación se utiliza la función dot()
print("\nResultado de la multiplicación m_unos * result_T")
print(np.dot(m_unos, result_T))

Índices de las matrices:
(4, 5)
(5, 4)

Resultado de la multiplicación m_unos * result_T
[[45. 45. 45. 45.]
 [45. 45. 45. 45.]
 [45. 45. 45. 45.]
 [45. 45. 45. 45.]]
```

# MATPLOTLIB

```

[27]: #Esta línea se ocupa para que las gráficas que se generen queden embebidas dentro de la página
      %pylab inline

      Populating the interactive namespace from numpy and matplotlib

[28]: #Importando las librerías
      import matplotlib.pyplot as plt
      from mpl_toolkits.mplot3d import Axes3D

[29]: #Datos de entrada
      X = linspace(0, 5, 10) #Generando 10 puntos entre 0 y 5
    
```

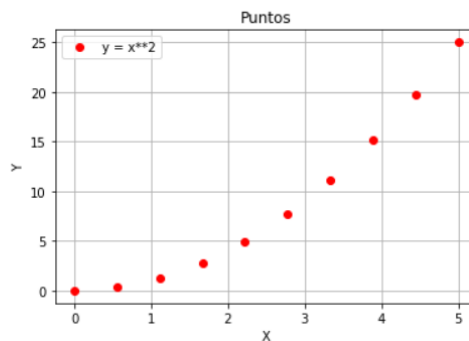
```

[30]: fig, ax = plt.subplots(facecolor='w', edgecolor='k')
      ax.plot(X, X**2, marker="o", color="r", linestyle='None')

      ax.grid(True)
      ax.set_xlabel('X') #Etiqueta del eje x
      ax.set_ylabel('Y') #Etiqueta del eje y
      ax.grid(True)
      ax.legend(["y = x**2"])

      plt.title('Puntos')
      plt.show()

      fig.savefig("gráfica.png") #Guardando la gráfica
    
```



```

[31]: fig, ax = subplots()

      ax.plot(X, X**2, label="y = x**2", markers="o", color="r")
      ax.plot(X, X**3, label="y = x**3", markers="x", color="b")
      ax.set_xlabel('X')
      ax.set_ylabel('Y')
      ax.grid(True)
      ax.legend(locs=2);

      plt.title('Dos gráficas')
      plt.show()
    
```

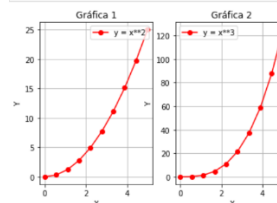


```

[32]: fig, ax = subplots(nrows=1, ncols=2) #Se genera un renglón con dos columnas donde se generarán dos gráficas
      ax[0].plot(X, X**2, label="y = x**2", markers="o", color="r")
      ax[0].set_xlabel('X')
      ax[0].set_ylabel('Y')
      ax[0].grid(True)
      ax[0].legend(locs=1);
      ax[0].set_title('Gráfica 1');

      ax[1].plot(X, X**3, label="y = x**3", markers="o", color="r")
      ax[1].set_xlabel('X')
      ax[1].set_ylabel('Y')
      ax[1].grid(True)
      ax[1].legend(locs=2);
      ax[1].set_title('Gráfica 2');

      plt.show()
    
```



## GRAFICACION

**Matplotlib** una biblioteca usada para generar gráficas en 2D y 3D, donde cada una de las configuraciones de la gráfica es programable. En el siguiente ejemplo se mostrará la configuración básica de una gráfica.

```
Práctica10 (Ejemplos).ipynb X
+ 🔍 📄 ▶ ⏏ Code ⌵ Download 📁 🌐 GitHub 🔗 Binder Python 3

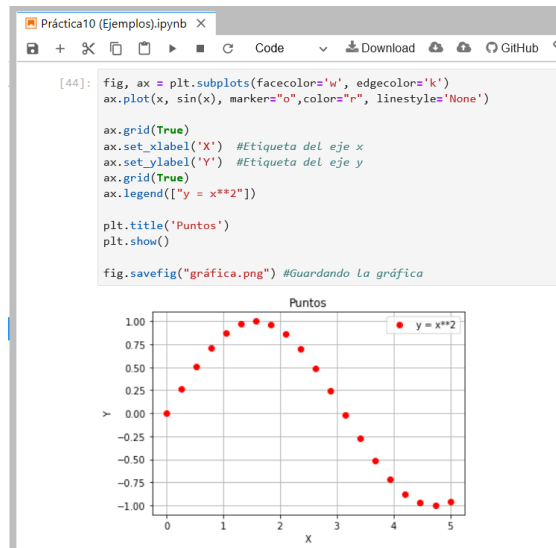
[41]: #Esta línea se ocupa para que las gráficas que se generen queden embebidas dentro de la página
      %pylab inline

      Populating the interactive namespace from numpy and matplotlib

      /srv/conda/envs/notebook/lib/python3.7/site-packages/IPython/core/magics/pylab.py:160: UserWarning: pylab import has clobbered these variables:
      ['fmod', 'hypot', 'sin', 'log10', 'floor', 'frexp', 'remainder', 'ldexp', 'tan', 'cosh', 'isinf', 'log2', 'pi', 'radians', 'trunc', 'ma', 'degrees', 'isfinite', 'cos', 'sqrt', 'gamma', 'exp', 'copysign', 'inf', 'tanh', 'log', 'modf', 'fabs', 'log1p', 'isclose', 'sinh', 'ceil', 'e', 'nan', 'gcd', 'expm1', 'isnan']
      '%matplotlib' prevents importing * from pylab and numpy
      "\n'%matplotlib' prevents importing * from pylab and numpy"

[42]: #Importando las bibliotecas
      import matplotlib.pyplot as plt
      from mpl_toolkits.mplot3d import Axes3D

[43]: #Datos de entrada
      x = linspace(0, 5, 20) #Generando 10 puntos entre 0 y 5
```



## EJECUCIÓN DESDE VENTANAS DE COMANDOS

Todo el código que se ha visto hasta el momento puede ser guardado en archivos de texto plano con la extensión `.py`. Para ejecutarlo desde la ventana de comandos se escribe el comando:

**`python nombre_archivo.py`**

## ENTRADA DE DATOS

Al igual que en otros lenguajes, también se puede pedir al usuario que introduzca ciertos datos de entrada cuando se ejecute un programa. Esto no se puede hacer desde la notebook, ya que los datos se introducen en las celdas que se van agregando a lo largo de la página, tal y como se ha venido manejando hasta ahora.

Como ejemplo se va a ejecutar el archivo `lectura_datos.py` desde una ventana de comandos.

*`python lectura_datos.py`*

## CONCLUSIONES

Con esta práctica he podido observar que Python es un lenguaje de programación extremadamente versátil y con muchas funciones que se vuelven muy prácticas al escribir el código e igualmente en su ejecución. Además, he tenido la oportunidad de investigar que cuenta con más de 300 módulos de biblioteca estándar que contienen módulos y clases para una amplia variedad de tareas de programación.

Incluso pienso que cuando uno ha dominado Python uno puede acostumbrarse tanto a sus características, particularmente su modelo dinámico de enlace en tiempo de ejecución y sus muchas bibliotecas, que puede ser difícil de aprender y sentirse cómodo en otros lenguajes de programación. Específicamente, la necesidad de declarar valores de variable "tipos" y "cast" de un tipo a otro y los requisitos sintácticos para agregar punto y coma y llaves utilizados por otros lenguajes de programación pueden ser vistos como tediosos.

## BIBLIOGRAFÍA

- Cuevas, A. (2019). *Programar con Python 3* (1ª edición). Madrid, España: RA-MA.
- Guagliano, C. (2019). *Programación en Python 1: Entorno de Programación- Sintaxis. Estructuras de Control* (1ª edición). Buenos Aires, Argentina: Six Ediciones.