Breve introducción a Python aplicado a probabilidad

March 5, 2021

1 Brevisima introducción a Python

El objetivo de estas notas es explicar brevemente como utilizar Python como una herramienta a la materia de Probabilidad y estadística. En el mismo se va a explicar como armar vectores, hacer algunos cálculos básicos relevantes a la materia como por ejemplo cálculo de medias y desvíos muestrales, así como también calculo de cuantiles. Además se explicará muy brevemente las herramientas para graficar y cálculo de histogramas.

1.1 Preliminares

1.1.1 Instalación de Python

Antes de empezar a trabajar con Python, es necesario verificar que tengamos una distribución de Python instalada en nuestra computadora.

- Windows:

  Abrimos una consola de Windows (Command Prompt - cmd) y escribimos

  ```
  python
  ```

  Si el sistema reconoce el comando correctamente debería aparecer algo del estilo:

  ```
  Python 3.7.0 (v3.7.0:1bf9cc5093, Jan 25 2019, 07:44:31) [MSC v.1914 64 bit (AMD64)] on wir
  Type "help", "copyright", "credits" or "license" for more information
  ```

  Si no se posee instalación de Python recomiendo instalarlo a través de la distribución Anaconda (https://www.anaconda.com/products/individual).

- Linux: viene con una versión de Python nativa. También se puede descargar Anaconda para mantener separadas las instalaciones. Además de esta forma podemos elegir la versión de Python que deseamos.

Si descargan Anaconda, ya incluye el programa Spyder, un IDE (*integrated development environment*) que provee interfaz gráfica, de forma que podemos escribir el código directamente en un archivo (¡y guardarlo!) para luego correrlo, todo en la misma plataforma. El Spyder se usa de forma análoga al RStudio. También existen otros IDEs como Pycharm (https://www.jetbrains.com/pycharm/), aunque no lo recomiendo para principiantes ya que hay que configurar varias cosas antes de poder arrancar.

1.1.2 Descarga de paquetes

Para poder sacarle el jugo a Python es necesario instalar varias librerías. Si te bajaste Anaconda, podes saltearte este paso ya que las librerías que vamos a necesitar ya vienen instaladas. Si no lo descargaste, vas a tener que hacer la instalación via Pip, para lo cual vas a tener que instalar el sistema:

- Ubuntu: Abrimos una terminal y escribimos

  ```
  apt install python3-pip
  ```

- Windows: (https://phoenixnap.com/kb/install-pip-windows)
  - Primero debemos bajar el archivo get-pip.py on pypa.io
  - Luego, abrimos una consola de Windows y escribimos

  ```
  python get-pip.py
  ```

  Si esta acción devuelve error, es posible que haya que abrir la consola como administrador.

Algunas de las librerías que vamos a utilizar:

- NumPy: da soporte para trabajar con vectores y matrices, así como diversas funciones matemáticas que operan sobre ellos. Instrucciones de instalación para distintos sistemas operativos: https://numpy.org/install/
- SciPy: se compone de herramientas y algoritmos matemáticos. Contiene módulos de estadística, optimización, álgebra lineal, entre muchas otras. Instrucciones de instalación: https://www.scipy.org/install.html
- Matplotlib: librería utilizada para graficar. Instrucciones de instalación: https://matplotlib.org/users/installing.html

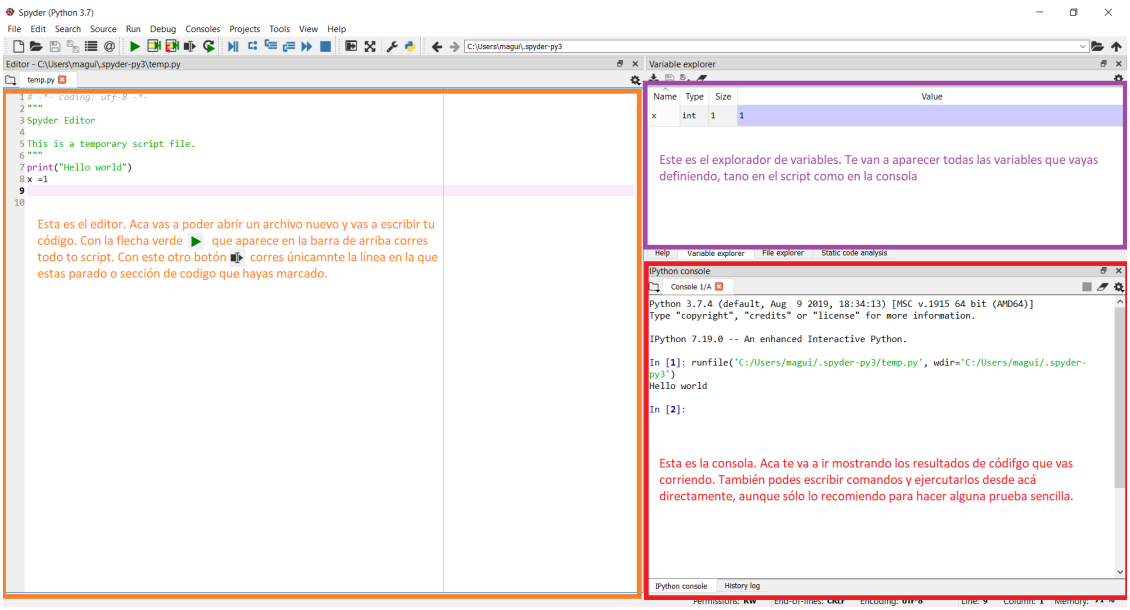
Existen muchas otras librarías muy útiles, pero estas son las básicas que vamos a necesitar para generar muestras aleatorias y poder graficar histogramas.

En este link hay un buen resumen de como arrancar a utilizar Numpy y los distintos tipos de datos.

1.2 Ahora si, comencemos a programar!

Si se desea se puede trabajar directamente en la consola para hacer los cálculos, pero lo más práctico es escribir un código, o *script*, con todos los comandos para poder usarlo todas las veces que haga falta. Un código o *script* es, entonces, un conjunto de comandos que uno puede generar y guardar como archivo de extensión ".py" para no tener la necesidad de volver a tipear todas las órdenes una y otra vez.

Si te bajase el Spyder, la ventana se va a ver algo así



Asignación de valores a una variable. En Python existen distintos tipos de datos, en este caso nos vamos a concentrar en definir variables constantes, es decir que contengan un único valor, listas, y en vectores, o *arrays*. Existen muchos otros tipos de datos que resultan útiles, pero que no vamos a estar necesitando por ahora.

Para definir constantes, basta con asignarle el valor a una variable utilizando el símbolo =

[487]:
```
x = 2
y = 1.56
z= 'a'
```

Las listas sirven para guardar distintos objetos en una única variable. Son nativas de Python y no necesitan de ninguna librería para poder usarlas. La lista se define a partir de una secuencia de elementos entre corchetes, separados por una ",":

[488]:
```
x = [1, 2, 1, 3, 5]
y = ["gato", "perro", "loro"]
z = [[1.1,1.2], [2.1,2.2], [0,1,2,3,6],["a", "b"]]
```

Las listas son muy útiles para definir conjuntos de objetos, sin embargo una desventaja que poseen es que no están definidas las funciones ariméticas básicas sobre listas, después de todo ¿que significaría decir "gato"+"perro"+"loro"?.

Para poder definir y operar con vectores y matrices, vamos a utilizar la librería Numpy. En este caso, los vectores (y matrices) se van a representar con lo que se llaman *arrays*. Estos objetos vienen previstos de las operaciones aritméticas básicas (suma, resta multiplicación y división) que operan componente a componente, junto con muchas otras funciones que permiten realizar diversas operaciones matriciales, entre muchas otras.

Lo primero que debemos hacer es importar la librería

[489]:
```
import numpy as np
```

La sintaxis general para importar cualquier librería (que tengamos instalada) es `import` [*libreria*] `as` [*nombre*] , donde [*libreria*] es en nombre de la librería que queremos importar y [*nombre*] es el nombre que vamos a utilizar para llamar a la librería en nuestro código. En verdad, basta con `import`[*libreria*], pero, como van a ir notando, adjudicarle un nombre más corto simplifica mucho la escritura y lectura del código.

Si queremos, por ejemplo, armar el vector $x = [1, 2, 3, 4]$ lo hacemos de la forma

[490]:
```
x = np.array([1,2,3,4])
```

o

[491]:
```
x = np.arange(1,7)
print(x)
```

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

la función `np.arange(start, stop)` es muy útil ya que genera una secuencia de números enteros que van desde `start` hasta `stop -1`. También habrán notado que usé la función `print`, la cual simplemente me imprime en la consola el valor de la variable.

Si en cambio quisiera definir una matriz, por ejemplo $A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$, puedo hacerlo como

[492]:
```
A = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

Para definir la matriz, se pasa una lista correspondiente a cada fila, separando cada lista por una coma `,`.

La librería Numpy posee diversas funciones. En la esta pagina, se encuentra un gráfico con un breve resumen de algunas de las funciones que trae Numpy y cómo operar con los vectores (arrays).

Para acceder a un elemento, o subconjunto de elementos, de los vectores (o matrices) simplemente indicamos los índices deseados entre corchetes `[]`. Recordar que, a diferencia de por ejemplo R, en Python los índices comienzan a contarse desde 0. Si queremos la primera posición de `x`

[493]:
```
x[0] # accedo a la primera posición
```

[493]:
```
1
```

[494]:
```
x[-1] # accedo a la última posición del vector
```

[494]:
```
6
```

Puedo pedir también un subconjunto de de índices,

[495]:
```
x[[0,1,5]]
```

[495]:
```
array([1, 2, 6])
```

obtener todos los elementos hasta una cierta posición,

[496]:
```
x[:2] # me devuelve hasta la posicion 2, es equivalente a  x[[0,1]]
```

[496]:
```
array([1, 2])
```

recuperar los elementos desde un índice en adelante,

[497]:
```
x[2:] # devuelve desde el índice 2 hasta el final del vector, equivale a
↪x[[2,3,4,5,5]]
```

[497]:
```
array([3, 4, 5, 6])
```

[498]:
```
x[-2:] # devuelve de la anteúltima posición hasta el final del vector
```

[498]:
```
array([5, 6])
```

o puedo querer ir desde un índice a otro.

[499]:
```
x[2:5] # devuelce desde el índice 2 hasta el 4, equivale a x[[2,3,4]]
```

[499]:
```
array([3, 4, 5])
```

Observar que al pedir `x[a:b]` me devuelve los índices `a` hasta `b-1`(inclusive)

Con las matrices la forma de acceder a una posición es equivalente, sólo que debemos indicar los números de filas y los de columnas. La sintaxis es `A[filas, columnas]` donde en `filas` y `columnas` podemos usar cualquiera de las sintaxis vistas para vectores. Por ejemplo:

[500]:
```
A[1:3,1:] #Accedemos a los elementos [1,1], [1,2], [1:3], [2,1],[2,2], [2,3] de
↪la matriz
```

[500]:
```
array([[ 6,  7,  8],
       [10, 11, 12]])
```

Para conocer las dimensiones de un vector una matriz basta con llamar a la función `shape`. Esto se puede hacer de dos formas

[501]:
```
np.shape(x)
```

[501]:
```
(6,)
```

o bien

[502]:
```
x.shape
```

[502]:
```
(6,)
```

[503]:
```
A.shape
```

[503]:
```
(3, 4)
```

Operaciones con vectores y matrices Las operaciones básicas, (+,-,*,/) que están definidas sobre los *arrays*, operan componente a componente.

[504]:
```
x = np.array([1,2,3,4])
y = np.array([5,1,6,4])
x+y
```

[504]:
```
array([6, 3, 9, 8])
```

[505]:
```
x*y
```

[505]:
```
array([ 5,  2, 18, 16])
```

[506]:
```
x**2 # elevo cada componene del vector al cuadrado
```

[506]:
```
array([ 1,  4,  9, 16], dtype=int32)
```

[507]:
```
A = np.array([[0,1,2],[3,4,5],[6,7,8]])
B = np.array([[0,10,20],[30,40,50],[60,70,80]])
print(A)
print(B)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[ 0 10 20]
 [30 40 50]
 [60 70 80]]
```

[508]:
```
A+B
```

[508]:
```
array([[ 0, 11, 22],
       [33, 44, 55],
       [66, 77, 88]])
```

[509]:
```
A*B
```

[509]:
```
array([[  0,  10,  40],
       [ 90, 160, 250],
       [360, 490, 640]])
```

Si queremos trasponer una matriz existen dos formas de escribirlo

[510]:
```
np.transpose(A)#forma 1
```

[510]:
```
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
```

[511]:
```
A.T # forma 2
```

[511]:
```
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
```

Si quisiera calcular el producto escalar entre dos matrices A y B, se utiliza la función `dot`

[512]:
```
np.dot(A,B)
```

[512]:
```
array([[ 150,  180,  210],
       [ 420,  540,  660],
       [ 690,  900, 1110]])
```

También podemos operar sobre un mismo vector, por ejemplo obteniendo la suma de todos sus elementos

[513]:
```
np.sum(x)
```

[513]:
```
10
```

[514]:
```
x.sum()
```

[514]:
```
10
```

[515]:
```
A.sum() #suma todos los elementos de la matriz A
```

[515]:
```
36
```

[516]:
```
A.sum(axis=1) # sumo las filas de la matriz
```

[516]:
```
array([ 3, 12, 21])
```

[517]:
```
A.sum(axis=0) # sumo las columnas de la matriz
```

[517]:
```
array([ 9, 12, 15])
```

[518]:
```
x.max() #obtengo el máximo de x
```

[518]:
```
4
```

[519]:
```
A.max(axis=1) # obtengo el máximo por fila de la matriz A
```

[519]:
```
array([2, 5, 8])
```

[520]:
```
x.mean() #calculo el promedio de los elementos de x
```

[520]:
```
2.5
```

Y muchísimas otras funciones más que vienen definidas en la librería. Pueden consultar tanto la documentación oficial, así como también una gran cantidad de tutoriales para que puedan ir arrancando.

Vectores lógicos A veces queremos hacer comparaciones. Por ejemplo, usando el vector `x` que definí previamente,

[521]:
```
x<2
```

[521]:
```
array([ True, False, False, False])
```

Esta operación me devuelve un vector de la misma dimensión que `x` que vale `True` cuando la componente es menor a 2 y `False` cuando no.

También puedo querer hacer comparaciones componente a componente entre dos vectores (o matrices)

[522]:
```
x>=y
```

[522]:
```
array([False,  True, False,  True])
```

[523]:
```
A == B # el == compara si los elementos son iguales
```

[523]:
```
array([[ True, False, False],
       [False, False, False],
       [False, False, False]])
```

[524]:
```
x!=y # != se usa para ver si los elementos son distintos
```

[524]:
```
array([ True,  True,  True, False])
```

Otra comparación que a veces resulta útil, es ver qué elementos de un vector (o matriz) pertenecen a un subconjunto de valores

[525]:
```
np.isin(x,[0,5,4,1])
```

[525]:
```
array([ True, False, False,  True])
```

[526]:
```
np.isin(A, [0,10,12,6])
```

[526]:
```
array([[ True, False, False],
       [False, False, False],
       [ True, False, False]])
```

[527]:
```
np.isin(A, [[10,8,2],[0,1,5]])
```

[527]:
```
array([[ True,  True,  True],
       [False, False,  True],
       [False, False,  True]])
```

Bucles Una herramienta muy útil presente en todos los lenguajes de programación son los bucles. Sirven por ejemplo para recorrer una matriz, o correr una porción de código utilizando distintos parámetros de simulación.

Posiblemente uno de los más conocidos sea el `for`, que se usa para iterar sobre una secuencia de valores. Esto sirve si sabemos cuántas veces queremos ejecutar cierta porción de código, o bien si quisiéramos evaluar una variable en distintos valores que conocemos a priori. La sintaxis es

```
for iterator in sequece:      comando 1                    ...
```

Así como en otro lenguajes como R o C, el comienzo y final de las instrucciones se da encerrándolas entre llaves, `{ comandos }`, en python se indica por la indentacion. Observá que el `for` se encuentra con una indentación y las instrucciones (luego del `:`) están un nivel mas adentro. Cuando finalizas las sentencias que van dentro del for volvés a la indentación original.

[528]:
```
for k in np.arange(5):
    factorial = np.product(np.arange(1,k+1)) #computo el factorial de k,␣
↪armando un vector que vaya de 1 a k
                                             #y multiplicando los elementos␣
↪entre sí.
    print(factorial)
```

```
1
1
2
6
24
```

[529]:
```
for animal in np.array(['gato', 'perro', 'loro', 'elefante']):
    print(animal)
```

```
gato
perro
loro
elefante
```

Otra sentencia super importante es el `if else`. Esto sirve para ejecutar un comando si ocurre una condición (y otro en caso que no, esto ultimo es optativo). La sintaxis es de la forma:

```
if (condicion):      comando 1        ...          else:      comando 1
...
```

Nuevamente, el el comienzo y el fin de cada conjunto de instrucciones se indica con un nuevo nivel de indentación.

[530]:
```
animal = np.array(['gato' , 'perro', 'elefante', 'loro'])
for a in animal:
    if a == 'loro':
        print('Vuela')
    else:
        print('Camina')
```

```
Camina
Camina
Camina
Vuela
```

Se pueden anidar tanto `if-else` como se desee. Para ello, los `else` intermedios son reemplazados por un `elif`

[531]:
```
for a in animal:
    if a == 'loro':
        print('Vuela')
    elif a == 'elefante':
        print('Es enorme')
    else:
        print('Es doméstico')
```

```
Es doméstico
Es doméstico
Es enorme
Vuela
```

Funciones Algo sumamente útil son las funciones. Ya que las definimos una única vez y luego las podemos llamar cuantas veces querramos en nuestro código. La sintaxis para crear funciones es: `def nombre_de_la_funcion(x,y,...):`. Luego la función se ejecuta como `nombre_de_la_función(x,y,...): instruccion 1 instruccion 2 ... return valor1, ...`

El `return` al final de la función no es obligatorio, pero es la forma que tenemos de acceder a los resultados de la función. Todas las variables que definamos dentro de la función son locales a la misma, y tanto las variables como los cómputos que no devolvamos en el return queden dentro de la función y no podemos acceder por fuera.

[532]:
```
def promedio(x):
    suma = x.sum()
    return suma/x.shape

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

[532]:
```
array([2.5])
```

En este caso, por ejemplo, una vez que llamamos a la función promedio, solo accedemos al valor del `return` y no tenemos acceso a `suma`. Si te fijas, la variable nunca te va a aparecer en el explorador de variables.

Las funciones que armes van a ser tan simples o tan complejas como necesites. También podés crear muchas funciones pequeñal y llamarlas una dentro de otra para crear un código más modularizado.

[533]:
```
def suma_cuadrado(x):
    return (x**2).sum()
def resto_media(x):
```

```
    y = x - promedio(x)
    return y

def varianza(x):
    sigma2 = suma_cuadrado(resto_media(x))/x.shape
    return sigma2

varianza(np.array([1,2,1,2,1,2]))
```

[533]:
```
array([0.25])
```

1.2.1 Probabilidad

Entre muchas de las funciones que tiene Numpy, se pueden generar valores aleatorios a partir de distintas distribuciones. Estas funciones se encuentran dentro del módulo `random`.

El caso más elemental es el de generar números aleatorios, es decir uniformes en el intervalo [0,1]. Para ello, usamos la función `rand()` del módulo `random`. Para generar un único número aleatorio basta con escribir

[534]:
```
np.random.rand()
```

[534]:
```
0.8814204638963415
```

Si en cambio queremos generar `n` elementos, basta con definir

[535]:
```
n=10
np.random.rand(n)
```

[535]:
```
array([0.56600262, 0.67568921, 0.92263965, 0.64234101, 0.07173451,
       0.08357854, 0.56823127, 0.11666522, 0.04288448, 0.38366199])
```

Si volviéramos a correr estas dos sentencias obtendríamos resulados distintos. Una forma de hacer que el código sea repetible es fijar lo que se llama la semilla. Para ellos se usa la función `seed`, y se le pasa un enero como parámetro. Cualquiera que te guste va a estar bien, pero si cambias el valor de la semilla te van a cambiar los números que obtengas. Es muy importante que este paso lo hagas antes de comenzar a generar datos.

[536]:
```
np.random.seed(0)
```

Otra función que nos va a ser muy útil es `choice`, que elige al azar tantos elementos de un vector como le digamos. Puede ser con o sin reposición.

[537]:
```
np.random.choice(np.arange(10), 5, replace=False)# del vector [0,1,2,...,9]␣
↪elijo al azas 5 elementos SIN reposicion
```

[537]:
```
array([2, 8, 4, 9, 1])
```

[538]:
```
np.random.choice(['perro', 'gato', 'oso', 'leon'], 3, replace=True)# de los␣
↪elementos perro, gato, oso, leon
                                                                    # elijo al␣
↪azas 5 elementos SIN reposicion
```

[538]:
```
array(['perro', 'leon', 'oso'], dtype='<U5')
```

Así como generamos número aleatorios, podemos obtener realizaciones de variables con distintas distribuciones.

[539]:
```
np.random.randn(n) # genera n muestras de una variable con distribución normal␣
↪estándar
```

[539]:
```
array([-0.05503512, -0.10731045,  1.36546718, -0.09769572, -2.42595457,
       -0.4530558 , -0.470771  ,  0.973016  , -1.27814912,  1.43737068])
```

[540]:
```
m=5
p=0.2
np.random.binomial(m,p, n)# genera n muestras de una variable con distribucion␣
↪binomial de parámetros m y p.
```

[540]:
```
array([3, 0, 2, 1, 2, 1, 1, 1, 1, 1])
```

[541]:
```
df = 3
np.random.chisquare(df, n) # genera n muestras de una variable con distribucion␣
↪chi cuadrado con df grados de libertad
```

[541]:
```
array([1.56153497, 2.95754703, 0.94347223, 4.21374028, 1.54552321,
       1.37377019, 0.81650911, 2.48178276, 2.80845353, 3.92985676])
```

Existen un montón de distribuciones disponibles, que podes ver en detalle en este link

La desventaja de este módulo es que sólo permite generar muestras y no tiene la posibilidad de calcular cuantiles, que muchas veces son necesarios. Sin embargo, existe otra librería, Scipy, que sí tiene estas capacidades. En particular vamos a utilizar el módulo stats. Nuevamente, debemos importarlo. En este caso vamos a importar directamente el módulo y no toda la librería

[542]:
```
import scipy.stats as stats
```

Para las distintas distribuciones continuas que están implementadas, existen distintos *metodos*, algunos de ellos son - rvs: genera números aleatorios - pdf: funcion de densidad de probabilidad - cdf: funcion de distribución - sf: función de supervivencia (1-cdf) - ppf: cuantiles (inversa de la cdf) Si queremos trabajar con la distribución normal, por ejemplo

[543]:
```
stats.norm.rvs() # genero un valor de la normal estándar
```

[543]:
```
1.4296607664750955
```

[544]:
```
stats.norm.pdf(1)# evaluo la función de densidad en z=1
```

[544]:
```
0.24197072451914337
```

[545]:
```
stats.norm.cdf(0) # evalúo la funcion de distribucion en 2
```

[545]:
```
0.5
```

[546]:
```
stats.norm.ppf(0.5) # calculo el cuantil 0.5
```

[546]:
```
0.0
```

Podes probar estas funciones con los valores que quieras, o con muchos a la vez.

[547]:
```
stats.norm.pdf([-3,-2,-1,0,1,2,3]) # evaluo la funcion de densidad en distintos␣
↪valore sde z con un paso de 1
```

[547]:
```
array([0.00443185, 0.05399097, 0.24197072, 0.39894228, 0.24197072,
       0.05399097, 0.00443185])
```

[548]:
```
stats.norm.cdf([-3,-2,-1,0,1,2,3]) # evaluo la funcion de distribucion en␣
↪distintos valore sde z con un paso de 1
```

[548]:
```
array([0.0013499 , 0.02275013, 0.15865525, 0.5       , 0.84134475,
       0.97724987, 0.9986501 ])
```

En el caso de las variables discretas, los *métodos* son los mismos excepto que cambia la `pdf` por la `pmf` que representa la función de probabilidad

[549]:
```
stats.binom.rvs(m,p) #genera una muestra de una variable con distribución␣
↪binomial de parámetros m y p
```

[549]:
```
0
```

[550]:
```
stats.binom.rvs(m,p, size = [3,2])# genero una matriz de 3x2 de muestras␣
↪independientes de una variable con distribución bin(3,p)
```

[550]:
```
array([[1, 3],
       [1, 1],
       [4, 1]])
```

[551]:
```
stats.binom.pmf(0, m,p) #me devuelve al valor de la función de probabilidad en 0
```

[551]:
```
0.32768
```

Una vez que obtenemos las muestras, podemos sacar datos muestrales, como la media muestral o el desvío estándar muestral. Por ejemplo puedo generar 100 muestras de una distribución chi cuadraro con `df` grados de libertad

[552]:
```
x = stats.chi2.rvs(df, size=100) #genero las 10 muestras
x.mean() # calculo la media muestral
```

[552]:
```
3.032143057598038
```

[553]:
```
x.std() # calculo el desvio estandar muestral
```

[553]:
```
2.37073790419428
```

Gráficos Para generar gráficos en Python, la librería básica el Matplotlib, y utilizamos el módulo pyplot

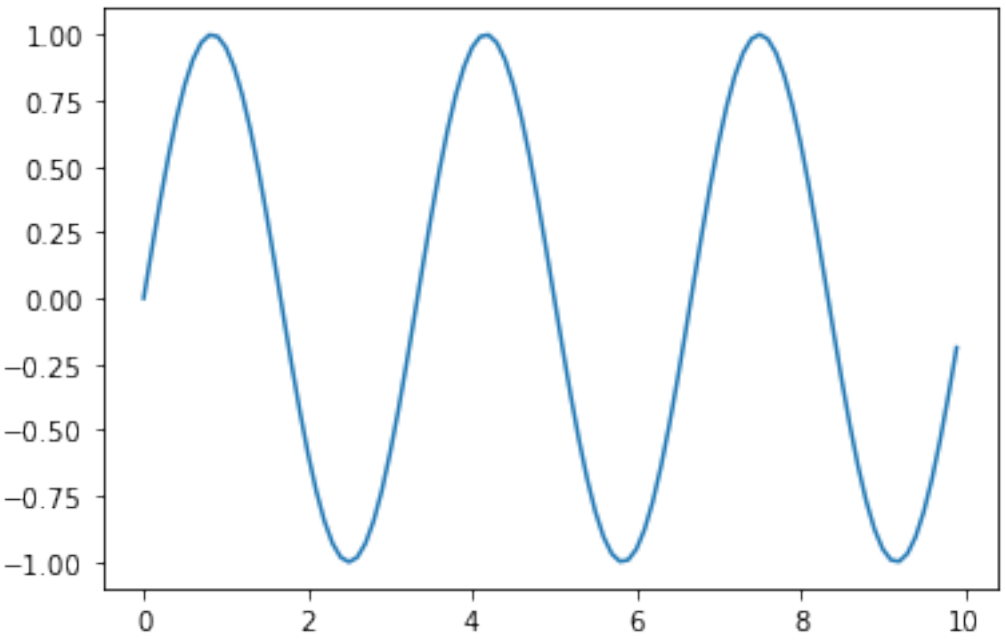
[554]:
```
import matplotlib.pyplot as plt
```

La función más común para generar un gráfico es `plot`

[555]:
```
t = 0.1*np.arange(100) #genero indices de tiempo con pasos de 0.1
x = np.sin(2*np.pi*0.3*t) #genero un seno de frecuencia 0.3Hz
```

[556]:
```
plt.plot(t,x)
```

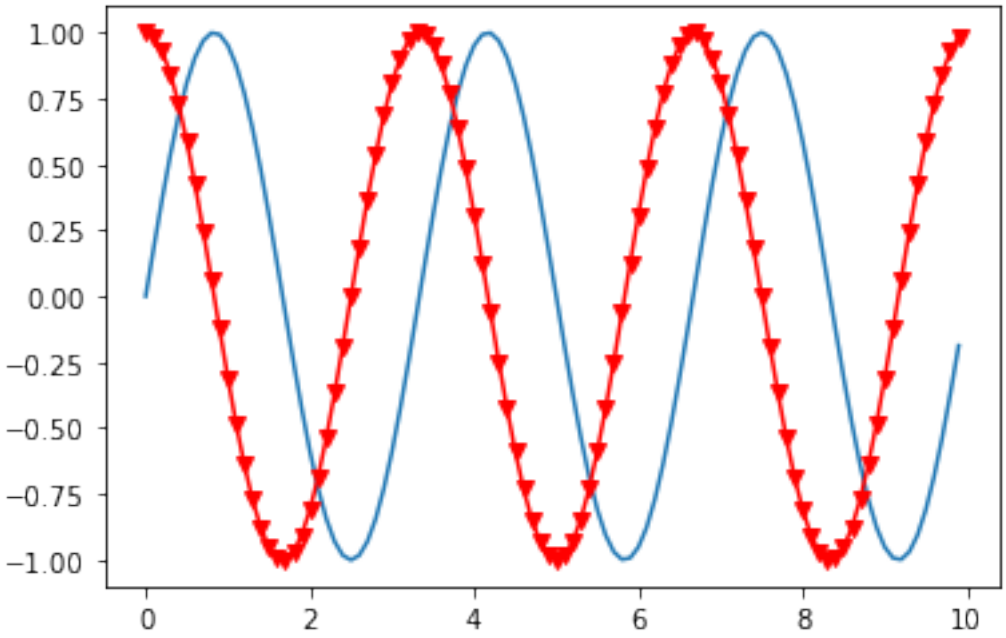
[556]:
```
[<matplotlib.lines.Line2D at 0x1eff44908c8>]
```



Este es el gráfico más simple. Se puede customizar el color, los marcadores, y agregar etiquetas a cada curva. Para agregar curvas al gráfico simplemente llamamos de nuevo a la función `plot`.

[557]:
```
y = np.cos(2*np.pi*0.3*t)
plt.plot(t,x, label='sin')
plt.plot(t,y, color='red', marker='v', label='cos')
```

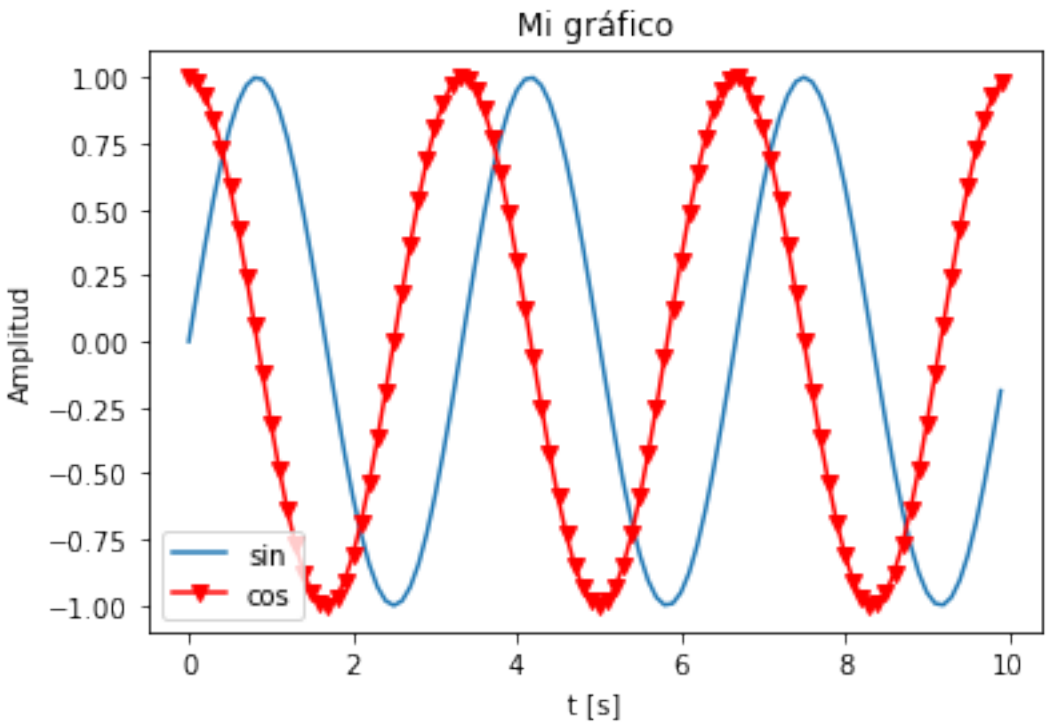
[557]:
```
[<matplotlib.lines.Line2D at 0x1eff4503288>]
```



con las funciones `xlabel` y `ylabel` podemos poner nombre a los ejes de coordenadas, mientras que con `title` elegimos un titulo. Con la función `lagend` transformamos las etiquetas de las curvas en leyendas

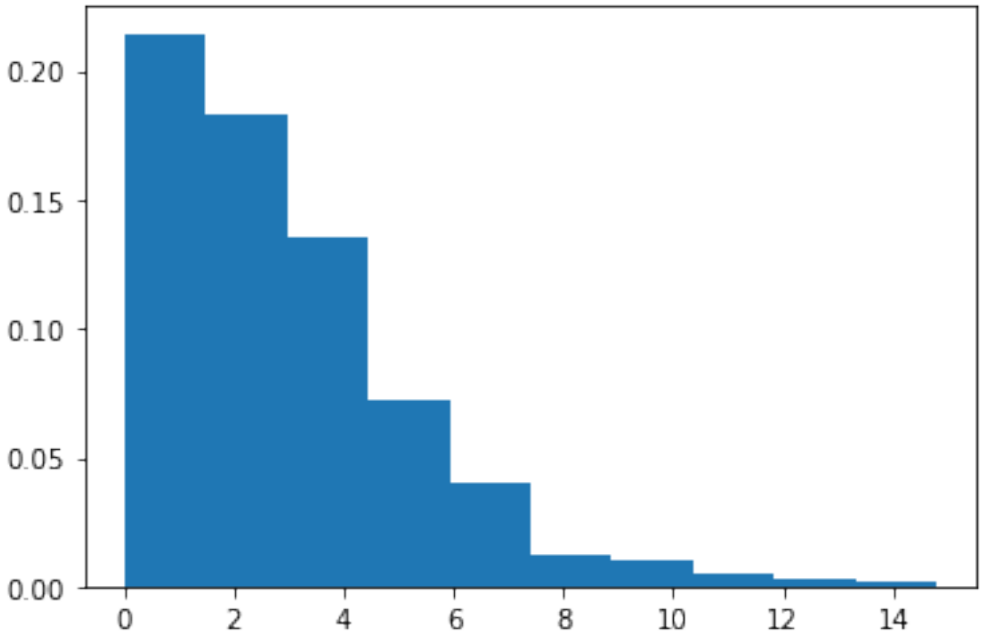
[558]:
```
plt.plot(t,x, label='sin')
plt.plot(t,y, color='red', marker='v', label='cos')
plt.xlabel('t [s]')
plt.ylabel('Amplitud')
plt.title('Mi gráfico')
plt.legend()
```

[558]:
```
<matplotlib.legend.Legend at 0x1eff45ab988>
```



Otra herramienta particularmente útil que trae `pyplot` es el cálculo de histogramas a partir de datos. Esto se hace con la función `hist`, que además de generar el gráfico nos devuelve las alturas de las barras del histograma y los intervalos (*bins*).

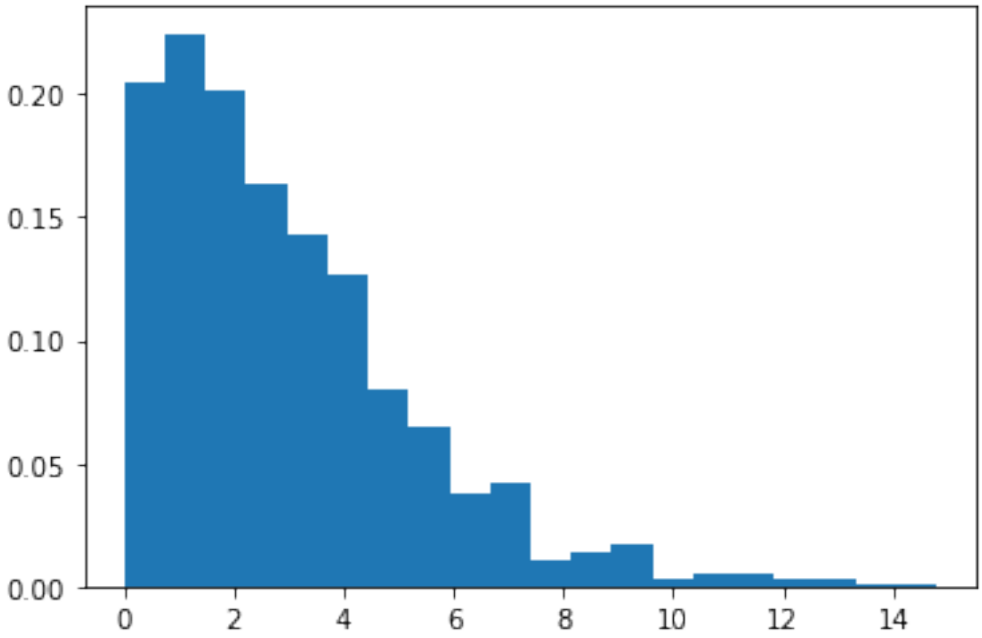
[559]:
```
n = 1000
chi = stats.chi2.rvs(df, size=n)
n, bins, _ = plt.hist(chi, density=True)
```



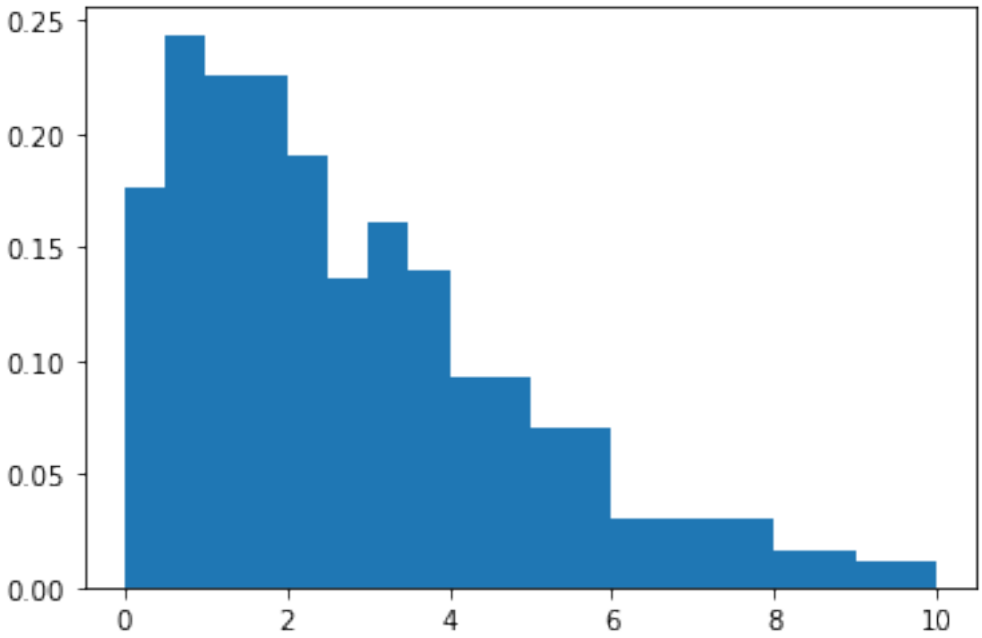
Este es el histograma generado automáticamente con 10 bins, lógicamente también podemos indicarle cuantos bins queremos o incluso cuáles queremos que sean los límites de dichos bins. Notar que para que los que obtengamos se corresponda con una densidad, a la función `hist` le pasamos el parámetro `density=True`, sino devuelve la cantidad de muestras en cada intervalo del histograma.

Obsérvese, que la función histograma en verdad de vuelve 3 elementos, como el tercero no lo quiero simplemente reemplazo el nombre de la variable por `_`. Esto puede hacerse con cualquier función que devuelva más de un resultado y solo estemos interesados en un subconjunto de los mismos.

[560]:
```
n,bins, _ = plt.hist(chi, density=True, bins=20) # si a bins le paso un entero,␣
↪es a cantidad de bins a usar
```



[561]:
```
n, bins,_ = plt.hist(chi, density=True, bins=np.array([0,0.5,1,1.5,2,2.5,3,3.
↪5,4,5,6,7,8,9,10]))
# si bins es un array, indica los límites de cada bin. Aca no es necesario que␣
↪tengan igual longitud.
```



Comentarios finales Este informe es super básico, y existe una cantidad gigante de cosas que se pueden hacer con Python. No tratamos cómo importar datos en nuestro código, que si bien es super importante, queda para una segunda version.

[ ]: