

VERSIÓN PRELIMINAR

Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

7 de noviembre de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-???-???-?
© Facundo Batista & Manuel Carlevaro
Primera Edición (versión preliminar)
Escrito con X₃TEX.

Licencia: [Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional \(CC BY-NC-SA 4.0\)](https://creativecommons.org/licenses/by-nc-sa/4.0/)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: <http://pyciencia.taniquetil.com.ar/>

10 9 8 7 6 5 4 3 2 1

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científicos, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos" que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquivó la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de los científicos para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Tanto los textos como el código fuente, ejemplos e imágenes son Copyright de Facundo Batista y Manuel Carlevaro y están compartidos bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) [1], salvo que se especifique puntualmente lo contrario.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Prefacio	2
Índice general	3
I Python	4
II Herramientas fundamentales	5
III Temas específicos	6
1. Elementos de estadística	7
1.1. Introducción	7
1.2. Números aleatorios	7
1.3. Variables aleatorias y distribuciones	12
1.3.1. Medidas de centralidad	13
1.3.2. Medidas de dispersión	16
1.4. Test de hipótesis	24
1.5. Estimación no paramétrica de la FDP	33
1.6. Lecturas recomendadas	37
IV Apéndices	38
A. Zen de Python	39
Bibliografía	40

Parte I

Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

Parte II

Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Parte III

Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte [II](#).

1 | Elementos de estadística

1.1. Introducción

La estadística comprende la colección, organización y análisis de datos para comprender fenómenos y tomar decisiones. Los métodos estadísticos son necesarios cuando tenemos información incompleta sobre un fenómeno, debido a que es imposible obtener datos de todos los miembros de una “población”, o porque las mediciones que realizamos poseen intrínsecamente una incerteza. Cuando no podemos determinar completamente los valores de una población, podemos obtener una “muestra” elegida aleatoriamente (muestreo sistemático, estratificado, de conglomerados, etc.) y por medio de métodos estadísticos obtener parámetros que permitan hacer inferencias sobre las propiedades de la población, en forma sistemática y con estimaciones.



Módulo	Versión
Matplotlib	3.9.2
NumPy	1.26.4
SciPy	1.14.1

[Código disponible](#)

Los métodos estadísticos se fundamentan en la teoría de la probabilidad, con la cual podemos modelar incertezas e información incompleta usando variables aleatorias. Como un ejemplo muy común, a partir de la selección aleatoria de una muestra de una población podemos inferir las propiedades de esta última. En la teoría de la probabilidad, cada salida posible de una observación tiene una probabilidad dada, y la probabilidad de todas las posibles salidas constituye la distribución de probabilidad. Dada una función de distribución de probabilidad, podemos calcular las propiedades de la población tales como la media y la varianza, pero si solo conocemos los valores para una muestra obtenida aleatoriamente podemos estimar los valores esperados, o medios, de la población.

1.2. Números aleatorios

La biblioteca estándar de Python provee el módulo `random` que implementa generadores de números pseudo aleatorios para diversas distribuciones, utilizando el algoritmo Mersenne-Twister como generador [2]. Por otra parte, el módulo `random` de NumPy brinda una funcionalidad similar con métodos que generan arrays de NumPy, además de ofrecer un número mayor de distribuciones de probabilidad.

Las rutinas para generar números pseudo aleatorios de NumPy utilizan una combinación

de un `BitGenerator`, que son objetos que proveen secuencias de bits aleatorios¹ compuestos de “palabras” de enteros sin signo con secuencias de 32 o 64 bits, y `Generators`, que transforman las secuencias de bits aleatorios de un `BitGenerator` en secuencias de números con una distribución de probabilidad específica (uniforme, normal o binomial, por ejemplo), en un intervalo dado.

Un detalle importante es la semilla utilizada para iniciar la secuencia de números aleatorios. Esta semilla se pasa como argumento al objeto `BitGenerator` en forma de entero o array de enteros. En caso de no pasar ninguna semilla al inicializar el `BitGenerator`, se tomará un valor del generador de entropía del sistema operativo². En las aplicaciones científicas o tecnológicas es importante emular un proceso aleatorio, pero también lo es que estas secuencias sean reproducibles, por lo que el control y registro de las semillas conforman una buena práctica.

Veamos entonces cómo se utilizan estas herramientas. Como muestra el *jupyter-notebook* a continuación, en la celda 1 importamos `Generator` y `PCG64` desde `numpy.random`, además de `NumPy` para la utilización de arrays. `PCG64` es el objeto `BitGenerator` que inicializa el generador pseudo aleatorio PCG-64, que a su vez implementa el generador congruencial de permutación de O’Neill [3]. Luego establecemos primero la semilla del generador utilizando el número de Hardy-Ramanujan³, de modo de hacer reproducible los resultados, e instanciamos el objeto `rng` de `Generator`. Luego obtenemos el primer número de la secuencia con el método `random` de `rng`, que devuelve un real con distribución uniforme en $[0, 1)$.

```

CELL 01

import numpy as np
from numpy.random import Generator, PCG64

np.random.seed(1729)
rng = Generator(PCG64())
rng.random()

0.9000273730257627

```

Si queremos obtener un número aleatorio con distribución uniforme en $[a, b)$ es necesario reescalar la salida de `random` multiplicándola por $(b-a)$ y sumándole a . Por ejemplo, si el intervalo es $[-3, 0)$:

```

CELL 02

(0 + 3) * rng.random() - 3

-0.8286282006850128

```

Tal como anticipamos, podemos obtener no solo números aleatorios aislados sino arrays con las dimensiones deseadas:

¹ Sabiendo que los algoritmos solo son capaces de generar números “pseudo aleatorios”, utilizaremos indistintamente el término “aleatorio” para referirnos a los que estrictamente son números “pseudo aleatorios”.

² En los sistemas operativos del tipo Unix, estas semillas se originan de los archivos especiales `/dev/random`, `/dev/urandom` y `/dev/arandom`, que coleccionan ruido ambiental de los dispositivos de la computadora. Se puede ver la implementación en Linux [aquí](#).

³ 1729 es el número natural más pequeño que puede expresarse como suma de dos cubos positivos, de dos formas diferentes: $1729 = 1^3 + 12^3 = 9^3 + 10^3$. También es conocido como el “número Taxi”, por la anécdota que le da nombre.

CELL 03

```
rng.random((2,3))

array([[0.65059468, 0.96076472, 0.5634431 ],
       [0.75887512, 0.7184905 , 0.8458265 ]])
```

El objeto Generator también ofrece la posibilidad de generar números enteros en un intervalo dado $[a, b)$ (por defecto semiabierto) mediante el método `integers()`. En el ejemplo siguiente generamos un array de cinco elementos en el intervalo cerrado $[10, 20]$, estableciendo que el extremo superior del intervalo puede ser generado con la opción `endpoint=True`.

CELL 04

```
rng.integers(low=10, high=20, size=5, endpoint=True)

array([16, 12, 14, 16, 18])
```

Otra funcionalidad muy útil que brinda Generator es la generación de una muestra aleatoria de un determinado array, usando para esto el método `choice()`. En el ejemplo a continuación simulamos una serie de 5 lanzamientos de tres dados. Para ello, en la celda 5 asignamos en la variable `dado` los posibles resultados del lanzamiento de un dado. En la celda 7 invocamos el método `choice` sobre el objeto `rng`, generando los 5 lanzamientos con tres dados, asumiendo que el dado es “honrado” y que cada cara tiene probabilidad $1/6$:

CELL 05

```
dado = np.arange(6) + 1
dado

array([1, 2, 3, 4, 5, 6])
```

CELL 06

```
rng.choice(dado, (5, 3), replace=True)

array([[1, 2, 4],
       [3, 3, 5],
       [4, 6, 4],
       [3, 5, 2],
       [5, 6, 1]])
```

El argumento `replace` permite establecer si el muestreo es con o sin reemplazo. La opción por defecto es `True`, lo que significa que los elementos del array `dado` pueden ser elegidos en múltiples oportunidades.

Por supuesto, podemos asignar probabilidades diferentes para cada elemento del array del cual queremos tomar la muestra. Para ello es necesario especificar mediante otro array la distribución deseada, cumpliendo la condición obvia que la suma debe ser 1. Este array de probabilidades se pasa como argumento `p` del método `choice`. En las celdas 7–8 siguientes “cargamos” el dado en la cara 4 asignándole mayor probabilidad que al resto de las caras, y repetimos el experimento de los cinco lanzamientos de los tres dados:

CELL 07

```
pesos = np.array([3/20, 3/20, 3/20, 1/4, 3/20, 3/20])
pesos.sum()
```

1.0

CELL 08

```
rng.choice(dado, (5,3), p=pesos, replace=True)
```

```
array([[5, 3, 2],
       [3, 4, 4],
       [2, 4, 5],
       [3, 3, 2],
       [1, 4, 5]])
```

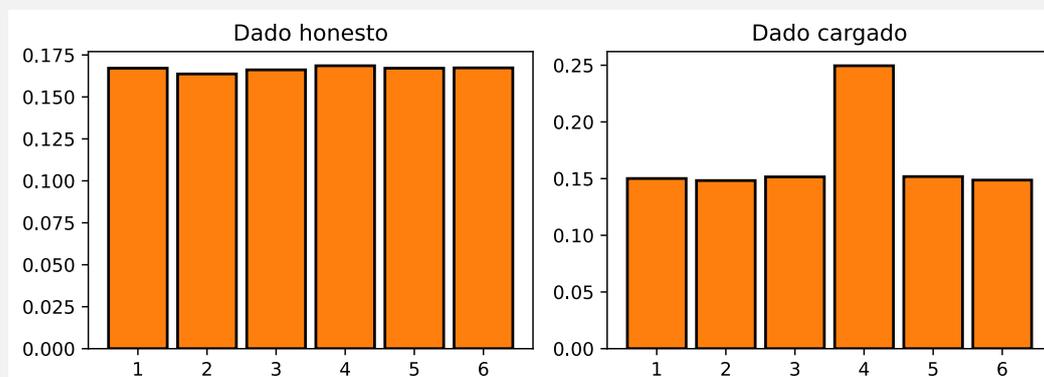
Podemos comparar la frecuencia con la que son seleccionados los diferentes elementos del array `dado` para ambos casos, generando un número relativamente grande de selecciones y graficando el histograma correspondiente, tal como vemos en la celda siguiente:

CELL 09

```
import matplotlib.pyplot as plt

num_lanzamientos = 100000 # Cantidad de veces que seleccionamos una cara del dado
dado_honesto = rng.choice(dado, num_lanzamientos, replace=True)
dado_cargado = rng.choice(dado, num_lanzamientos, p=pesos, replace=True)
bins = [1, 2, 3, 4, 5, 6, 7] # Bins para construir el histograma

fig, ax = plt.subplots(1,2, figsize=(8,3))
ax[0].hist(dado_honesto, bins=bins, density=True, align='left', rwidth=0.85, ec='black',
           fc='tab:orange', linewidth=1.5)
ax[0].set_title("Dado honesto")
ax[0].set_xticks(bins[:-1])
ax[1].hist(dado_cargado, bins=bins, density=True, align='left', rwidth=0.85, ec='black',
           fc='tab:orange', linewidth=1.5)
ax[1].set_title("Dado cargado")
ax[1].set_xticks(bins[:-1])
plt.tight_layout()
plt.show()
```



Por último, Generator provee métodos para reordenar o permutar elementos de una secuencia: `shuffle` y `permutation`. La principal diferencia entre estos métodos consiste en que `shuffle` reordena los elementos de la secuencia modificando ésta (*in place*), por lo que la secuencia debe ser un objeto “mutable”, mientras que `permutation` devuelve una copia si como argumento se le pasa un array, o un rango reordenado si como argumento se le pasa un entero. Podemos ver cómo se reordenan arrays, listas o rangos en las siguientes celdas:

```

array = np.arange(10)
rng.shuffle(array)
array
-----
array([2, 8, 1, 5, 7, 4, 0, 6, 9, 3])

```

```

rng.permutation([1, 2, 3, 4, 5])
-----
array([3, 2, 1, 5, 4])

```

```

rng.permutation(10)
-----
array([1, 5, 2, 3, 4, 8, 9, 7, 0, 6])

```

En el caso que necesitemos reordenar arrays multidimensionales, podemos establecer sobre qué dimensión hacer el reordenamiento indicándolo en el argumento `axis`. Por ejemplo, si queremos ordenar “por columna”:

```

x = np.arange(0, 12).reshape(3,4)
x
-----
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

```

```

y = rng.permutation(x, axis=1)
y
-----
array([[ 1,  0,  2,  3],
       [ 5,  4,  6,  7],
       [ 9,  8, 10, 11]])

```

mientras que si queremos hacerlo “por fila”:

```

CELL 15
z = rng.permutation(x, axis=0)
z
-----
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [ 0,  1,  2,  3]])

```

1.3. Variables aleatorias y distribuciones

En la teoría de la probabilidad, el conjunto de todos los resultados de un experimento aleatorio se denomina “espacio muestral”, y cada uno de los resultados se llama “punto muestral”. Por lo general habrá más de un espacio muestral para describir los resultados de un experimento, pero usualmente hay solo uno que suministra la mayoría de la información (por ejemplo, para el experimento de lanzar un dado podríamos tener como espacio muestral el número sobre la cara superior, o si esa cara representa un número par o impar).

Si a cada punto muestral le asignamos un número, podemos definir una “función aleatoria” (o “estocástica”) que mapea desde el espacio muestral al conjunto de números reales o enteros, que de este modo constituye una “variable aleatoria” o “estocástica”. Por ejemplo, nuestro experimento podría ser lanzar una moneda dos veces de tal forma que nuestro espacio muestral es $E = \{CC, CS, SC, SS\}$ donde C representa cara y S , sello. Con cada punto muestral podemos asociar un número X que sea el número de caras. Nuestra variable aleatoria queda definida entonces por la siguiente tabla:

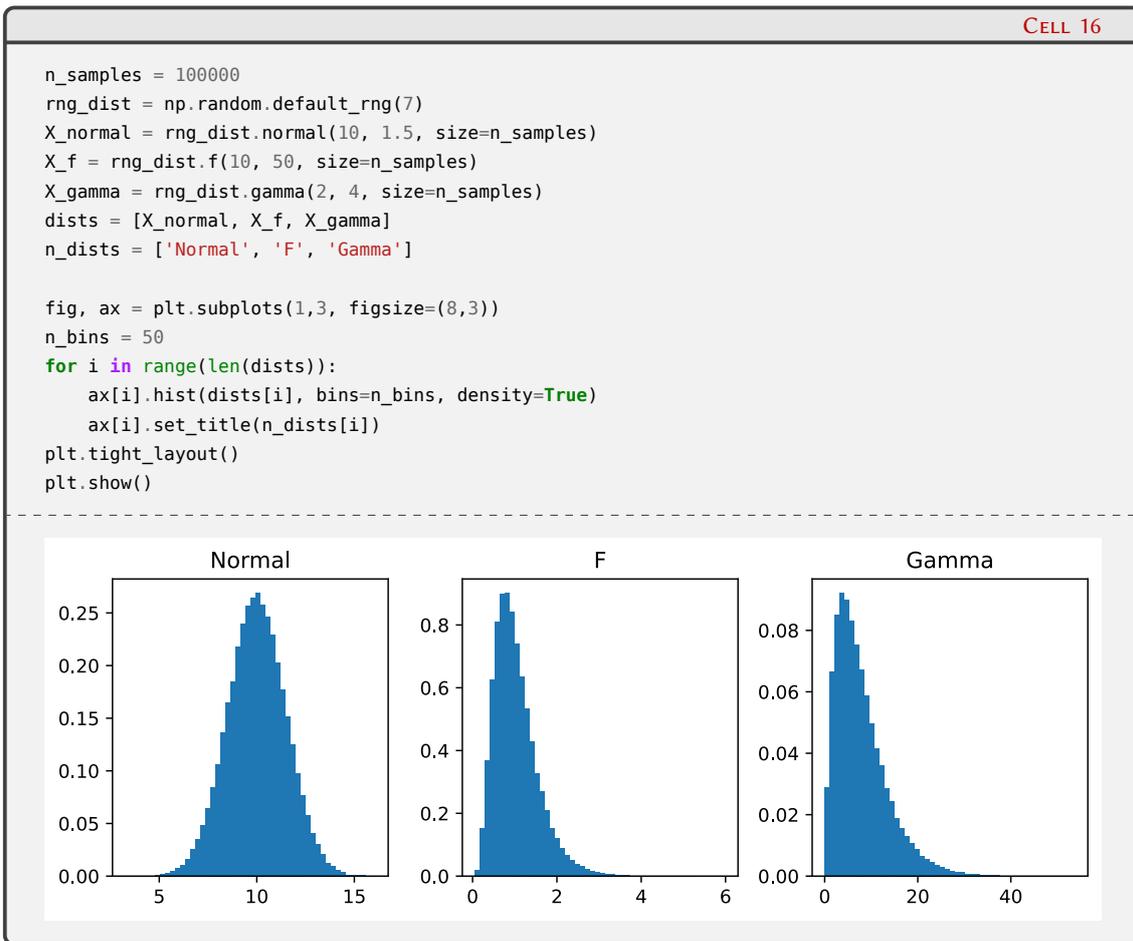
Punto muestral	CC	CS	SC	SS
X	2	1	1	0

Una variable aleatoria que toma un número finito o infinito contable de valores se denomina “variable aleatoria discreta”, mientras que una que toma un número infinito no contable de valores se llama “variable aleatoria continua”. Un problema estadístico usual consiste en mapear el resultado de experimentos en valores numéricos y determinar la distribución de probabilidad de esos valores. En consecuencia, una variable aleatoria se caracteriza por los posibles valores que puede tomar (el “recorrido” de la variable aleatoria) y por su función de distribución de probabilidad, y en la práctica esto significa trabajar con estas distribuciones. Según la notación tradicional, en el caso discreto se denomina “función de distribución de probabilidades” mientras que para el caso continuo la denominación es “función de densidad de probabilidad”. En lo que sigue usaremos indistintamente la notación FDP según el caso sea discreto o continuo.

Los objetos Generator de NumPy permiten generar secuencias de números aleatorios a partir de numerosas distribuciones⁴. Como ejemplo, en la celda siguiente construimos un objeto Generator invocando (a diferencia del ejemplo previo) el constructor por defecto `default_random()` que lo inicializa con el BitGenerator PCG64, y al que le pasamos la semilla del generador (lo cual es opcional⁵). Luego utilizamos ese objeto para generar arrays con un número `n_samples` de elementos con distintas distribuciones, que se muestran como histogramas en la figura.

⁴ Se puede ver la lista completa de distribuciones disponibles en la [documentación](#) de NumPy.

⁵ En este ejemplo utilizamos el [primer primo feliz](#).



Como vimos, un objeto Generator de NumPy es capaz de producir una secuencia de números aleatorios con una distribución dada. Esto es, de algún modo, equivalente a realizar experimentos estocásticos con esa distribución y obtener salidas de las variables aleatorias que resultan de ese experimento. Una descripción más completa a nivel estadístico se obtiene a partir del submódulo stats de SciPy.

El módulo `scipy.stats` provee clases para representar variables aleatorias con una extensa lista de distribuciones de probabilidad. Tiene dos clases base para variables aleatorias continuas y discretas: `rv.continuous` y `rv.discrete`. Estas clases no son utilizadas directamente sino que se usan como clases base para variables estocásticas con distribuciones específicas, y definen una interfaz común para todas las variables aleatorias en este módulo. A continuación haremos una revisión breve de los principales conceptos que se utilizan para caracterizar una distribución, y la forma de obtener descriptores utilizando convenientemente métodos de NumPy o SciPy.

1.3.1. Medidas de centralidad

Cuando tenemos un conjunto de datos como muestra de una distribución, podemos caracterizar el centro de la distribución con diferentes parámetros: según su valor o por su rango en función del orden en una lista de acuerdo a una magnitud.

1.3.1.1. Media

Por la media de una distribución nos estamos refiriendo a la media aritmética \bar{x} :

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

Tomemos como ejemplos para analizar las distribuciones obtenidas en la celda 16. Con el método `mean()` de NumPy podemos obtener las medias:

CELL 17
<code>X_normal.mean(), X_f.mean(), X_gamma.mean()</code>

<code>(9.998010521368922, 1.039862209298918, 8.001001879189166)</code>

Un problema común que acontece es que el conjunto de datos con el que trabajamos tiene valores perdidos (por errores de medición, registro, etc.) que se representan por NaN (que significa “not a number”, o un no-número). En estos casos, NumPy posee métodos que pueden obtener resultados ignorando estos NaN. Por ejemplo, en la celda 18 agregamos un NaN (que numpy representa con `numpy.nan`) a nuestro array con distribución normal. Si calculamos la media de esta nueva distribución, obtenemos un NaN como resultado, pero con el método `nanmean()` recuperamos el resultado correcto:

CELL 18
<code>X_normal_Nan = np.hstack((X_normal, np.nan))</code> <code>X_normal_Nan.mean()</code>

<code>nan</code>

CELL 19
<code>np.nanmean(X_normal_Nan)</code>

<code>9.998010521368922</code>

1.3.1.2. Mediana

La mediana es el valor que separa la mitad superior de la mitad inferior de una lista ordenada de valores. En caso en que dicha lista tenga un número par de valores, la mediana es la media aritmética de los dos valores centrales.

En la celda siguiente podemos ver el resultado de obtener la mediana para dos arrays con números par e impar de elementos:

CELL 20

```
valores_par = np.arange(10)
valores_impar = np.arange(11)
print(f'Mediana par: {np.median(valores_par)} | Mediana impar: {np.median(valores_impar)}')
print(f' Media par: {valores_par.mean()} | Media impar: {valores_impar.mean()}')
```

```
Mediana par: 4.5 | Mediana impar: 5.0
Media par: 4.5 | Media impar: 5.0
```

Se puede observar que cuando las distribuciones de valores son simétricas, la media y la mediana coinciden, pero no así cuando las distribuciones no lo son:

CELL 21

```
print(f' Media Normal: {X_normal.mean()}')
print(f'Mediana Normal: {np.median(X_normal)}')
print(f' Media F: {X_f.mean()}')
print(f'Mediana F: {np.median(X_f)}')
```

```
Media Normal: 9.998010521368922
Mediana Normal: 9.996573186158033
Media F: 1.039862209298918
Mediana F: 0.9448978239181103
```

En el caso en que tengamos una función de densidad de probabilidad (variable continua), la mediana es el valor tal que si integramos dicha función desde $-\infty$ hasta la mediana, la integral da 0,5.

1.3.1.3. Moda

La moda de un conjunto de datos es el valor que aparece con mayor frecuencia (si la distribución es continua, es el valor máximo). Por ejemplo, si hacemos cinco experimentos lanzando cuatro dados cargados según la FDP que mostramos en la celda 7, podemos obtener la moda en cada eje del array resultante, o de todas las salidas, utilizando el método `mode` de `scipy.stats` como se muestra en la celda 23:

CELL 22

```
from scipy import stats

samples = rng.choice(dado, (5, 4), p=pesos, replace=True)
print(samples) # Salida del experimento
print(stats.mode(samples)) # Por defecto axis=0 (conteo por columnas)
print(stats.mode(samples, axis=None)) # Todos los datos
```

```
[[4 1 4 5]
 [5 6 2 4]
 [3 6 4 1]
 [4 6 1 2]
 [3 5 3 6]]
ModeResult(mode=array([3, 6, 4, 1]), count=array([2, 3, 2, 1]))
ModeResult(mode=4, count=5)
```

En la primera salida del cálculo de la moda obtenemos un array que muestra cuál es el valor más frecuente en cada columna (`axis=0` por defecto) y un array con el número de veces que aparece el valor correspondiente. Si queremos obtener la moda de todos los valores que contiene el array `samples`, pasamos la opción `axis=None` como argumento.

En caso que el conjunto de datos a analizar incluya uno o varios NaN, es posible establecer la respuesta de `mode` mediante el parámetro opcional `nan_policy`. Si este parámetro es `'propagate'` el método `mode` devuelve NaN (y ésta es la opción por defecto). Si le pasamos `'omit'` obtenemos la moda de la distribución omitiendo los NaN y si el valor de este parámetro es `'raise'` se lanza una excepción. Cuando existen varias modas en el conjunto, `mode()` devuelve la menor.

1.3.1.4. Media geométrica

En algunos casos es útil la media geométrica como medida de centralidad de una distribución. Esta magnitud se puede calcular mediante la media aritmética del logaritmo de los valores:

$$\bar{x}_g = \left(\prod_{i=1}^n x_i \right)^{1/n} = \exp \left(\frac{\sum_i \ln(x_i)}{n} \right)$$

Esta función se puede evaluar con el método `gmean` de `scipy.stats`. Obviamente, los valores deben ser positivos para que se pueda calcular la media geométrica. En la celda 23 podemos ver las distintas medidas de centralidad sobre la distribución F .

```

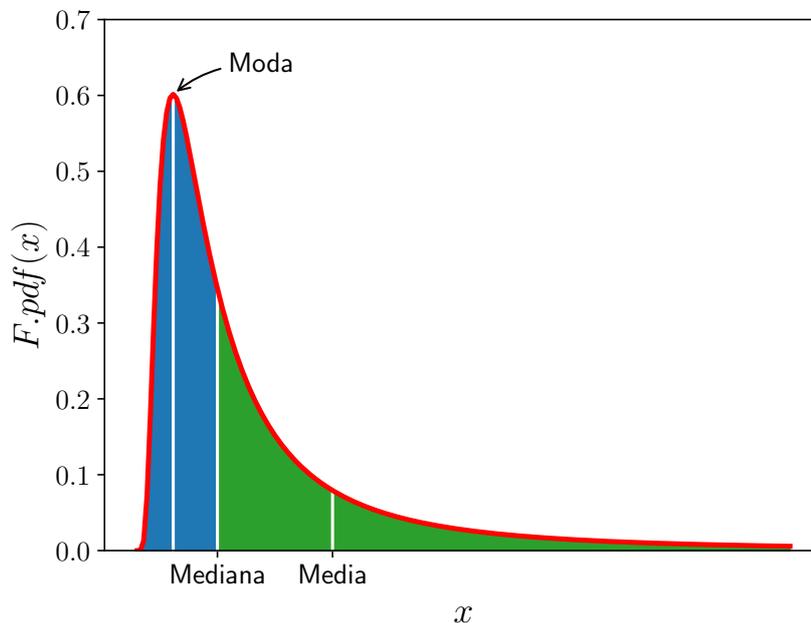
CELL 23
print(f'      Media F: {X_f.mean()}')
print(f'      Mediana F: {np.median(X_f)}')
print(f' Media geom. F: {stats.gmean(X_f)}')
print(f'      Moda F: {stats.mode(X_f)}')
-----
Media F: 1.039862209298918
Mediana F: 0.9448978239181103
Media geom. F: 0.9189045833992728
Moda F: ModeResult(mode=0.05411090116941826, count=1)

```

En la figura 1.1 vemos una comparación gráfica de la moda, la mediana y la media para la distribución F cuya asimetría permite distinguir las tres magnitudes. Cabe enfatizar aquí que estas medidas de centralidad corresponden a muestras específicas, si cambiamos la semilla que usamos para el generador, obtendremos otros valores dado que estaríamos analizando una salida de esas mismas variables aleatorias.

1.3.2. Medidas de dispersión

Además de las medidas de centralidad que nos informan acerca de los valores centrales de una distribución de datos, es necesario también conocer alguna cantidad que represente la dispersión de los valores alrededor de las medidas centrales. Veremos a continuación algunas medidas frecuentes.

FIGURA 1.1: Moda, mediana y media para la distribución F .

1.3.2.1. Rango

El rango de una distribución de datos es la diferencia entre los valores máximo y mínimo de dicha distribución, y se obtiene con el método `ptp` de NumPy (que proviene del inglés *peak-to-peak*):

```

CELL 24
for name, distribucion in zip(n_dists, dists):
    print(f'Rango de {name}: {np.ptp(distribucion)}')

Rango de Normal: 12.845742827626554
Rango de F: 5.944760304157928
Rango de Gamma: 53.84036147711152

```

Esta medida de dispersión incluye los valores atípicos, u *outliers*, que difieren notoriamente del resto de los valores de nuestro conjunto de datos, y pueden ser originados por errores en la medición o registro. Existen criterios para detectar estos *outliers*, tal como el considerar que un valor es atípico cuando se encuentra a $1,5 IQR$ de distancia de los cuartiles Q_1 y Q_3 , siendo $IQR = Q_3 - Q_1$ el rango intercuartílico (que veremos a continuación).

1.3.2.2. Cuantiles

Los cuantiles son valores específicos de la variable aleatoria que dividen el rango de la distribución de probabilidad en intervalos continuos de igual probabilidad. Los cuantiles más usuales tienen nombre propio: cuartiles (cuatro grupos), deciles (diez grupos) y percentiles (100 grupos). Un caso particular es la mediana, que divide la distribución en dos regiones de igual probabilidad.

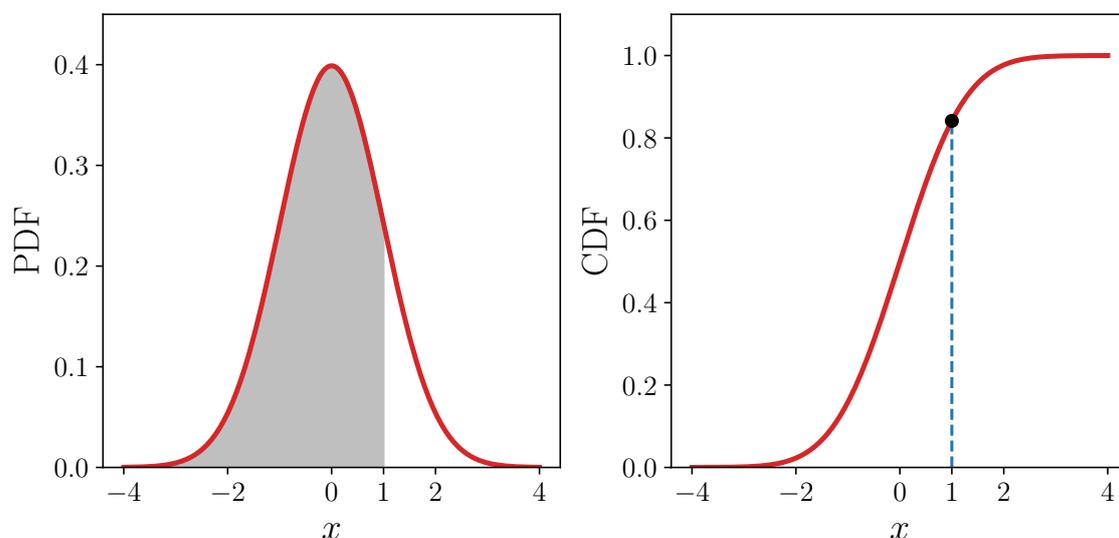


FIGURA 1.2: Función de densidad de probabilidad (FDP, panel izquierdo) y función de distribución acumulada (CDF, panel derecho), para una distribución normal con media $\mu = 0$ y desviación estándar $\sigma = 1$.

Para analizar cuantitativamente los cuantiles es útil definir la “función de distribución acumulada” CDF (por su sigla en inglés) como función de la integral de la FDP:

$$\text{CDF}(x) = \int_{-\infty}^x \text{FDP}(x') dx' \quad (1.1)$$

Conocer la CDF facilita el cálculo de la probabilidad de encontrar un valor de x en el intervalo (a, b) , dado que

$$P(a \leq X \leq b) = \int_a^b \text{FDP}(x) dx = \text{CDF}(b) - \text{CDF}(a)$$

Naturalmente, para distribuciones discretas, las integrales deben reemplazarse por sumas. La figura 1.2 muestra la representación gráfica de la $\text{PDF}(x)$ y $\text{CDF}(x)$ para una distribución normal, destacando en ambos paneles el valor de $\text{CDF}(x = 1)$.

Dado que la CDF es una función monótona creciente, su inversa CDF^{-1} existe, y a partir de ésta se determinan los cuantiles. Por ejemplo, la ya mencionada mediana es simplemente $\text{CDF}^{-1}(0,5)$. Del mismo modo se pueden encontrar rangos que incluyan un porcentaje dado de los datos. Si queremos obtener el rango $(x_{\text{inf}}, x_{\text{sup}})$ que incluye el 95 % de la distribución, tenemos:

$$x_{\text{inf}} = \text{CDF}^{-1}(0,025), \quad x_{\text{sup}} = \text{CDF}^{-1}(0,975)$$

Como mencionamos, otros cuantiles muy usados son los cuartiles Q_1 , Q_2 y Q_3 , que corresponden a los percentiles 25 ($\text{CDF}^{-1}(0,25)$), 50 ($\text{CDF}^{-1}(0,5)$) y 75 ($\text{CDF}^{-1}(0,75)$), respectivamente. Estos cuartiles se utilizan para generar los gráficos de caja (*boxplots*), y la diferencia $IQR = Q_3 - Q_1$ se denomina “rango intercuartílico”, que se utiliza en uno de los métodos para identificar *outliers*.

El submódulo `stats` de SciPy tiene la función `iqr()` que calcula, por defecto, el rango intercuartílico de una distribución. Sin embargo, podemos modificar este comportamiento estableciendo el rango correspondiente a otros percentiles. Por ejemplo, para calcular el rango de una

distribución podemos pasar como argumento `rng=(0, 100)` que equivale a utilizar la función `ptp`. En el ejemplo mostrado en la celda 25 vemos cómo podemos calcular la mediana como el percentil 50, y el rango intercuartílico como diferencia de los percentiles 75 y 25.

```

CELL 25
IQR = stats.iqr(X_gamma)
p25 = stats.iqr(X_gamma, rng=(0, 25)) # Percentil 25
p50 = stats.iqr(X_gamma, rng=(0, 50)) # Percentil 50
p75 = stats.iqr(X_gamma, rng=(0, 75)) # Percentil 75
ptp = stats.iqr(X_gamma, rng=(0, 100)) # rango
print(f'      IQR = } \np75 - p25 = {p75 - p25} \nmediana = {p50} \n {   ptp = }')

-----
IQR = 6.916758838426253
p75 - p25 = 6.916758838426253
mediana = 6.721041139175036
ptp = 53.84036147711152

```

El uso de cuartiles permite representar distribuciones mediante los populares gráficos de caja o *boxplots*. En la celda 26 generamos nuevas distribuciones (normal, lognormal y Gumbel) y representamos los valores obtenidos mediante cajas que abarcan el 50 % de los datos, desde el primer cuartil Q_1 hasta el tercer Q_3 (es decir, el rango intercuartílico *IQR*). La línea horizontal dentro de la caja representa la mediana o segundo cuartil Q_2 . Los “bigotes” (líneas negras en ambos lados la caja) se extienden hasta $Q_1 - 1,5 IQR$ y $Q_3 + 1,5 IQR$, y establecen el rango fuera del cual los valores de la distribución se consideran *outliers*, que en la figura se muestran con cruces. Estos bigotes pueden representar valores alternativos como: los valores mínimo y máximo, una desviación estándar alrededor de la media de los datos, los percentiles 9 y 91 o los percentiles 2 y 98⁶.

⁶ Para más información sobre cómo construir estos gráficos, invitamos a leer la [documentación de Matplotlib](#).

CELL 26

```

n2_samples = 500 # Número de muestras

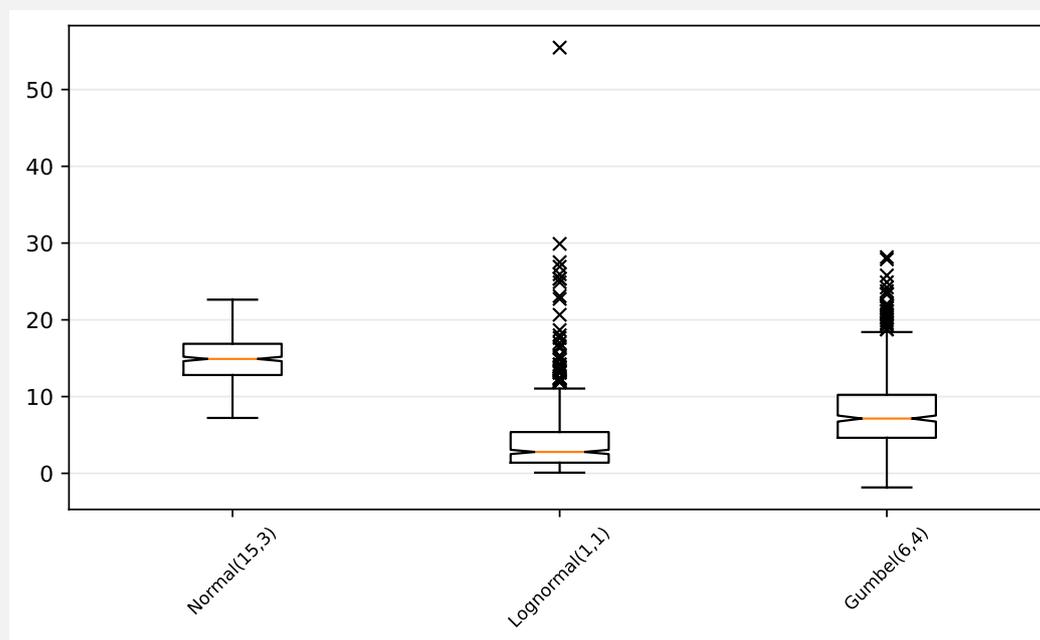
# Tomamos muestras de tres distribuciones
X2_normal = rng_dist.normal(15, 3, size=n2_samples)
X2_logn = rng_dist.lognormal(1, 1, size=n2_samples)
X2_gumb = rng_dist.gumbel(6, 4, size=n2_samples)
dists_2 = [X2_normal, X2_logn, X2_gumb]
nombres_dist = ['Normal(15,3)', 'Lognormal(1,1)', 'Gumbel(6,4)']

fig2, ax2 = plt.subplots(figsize=(8,4))
bp = plt.boxplot(dists_2, notch=True, sym='+', vert=True, whis=1.5)
plt.setp(bp['boxes'], color='black')
plt.setp(bp['whiskers'], color='black')
plt.setp(bp['fliers'], marker='x')

# Trazamos una grilla horizontal de color tenue para no distraer
# la lectura de los datos.
ax2.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
               alpha=0.5)

# Etiquetamos los boxplots
xtickNames = plt.setp(ax2, xticklabels=nombres_dist)
plt.setp(xtickNames, rotation=45, fontsize=8)
plt.show()

```



1.3.2.3. Varianza y desviación estándar

El cálculo de la varianza de una población con n valores discretos se define como

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (1.2)$$

y la correspondiente desviación estándar es simplemente σ . En caso en que los valores disponibles no correspondan a toda la población, sino solamente a una *muestra* de la misma, se sabe que la ecuación (1.2) es un estimador sesgado⁷ que subestima la varianza de la población. Se puede demostrar que un estimador no sesgado está dado por

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (1.3)$$

que se conoce como la varianza muestral, y la correspondiente raíz cuadrada como la desviación estándar muestral. La notación usual consiste en representar con letras del alfabeto griego los parámetros poblacionales y con letras latinas los correspondientes muestrales. La diferencia con la varianza poblacional es que dividimos por $n-1$ en vez de n . La razón de esto es que se ha eliminado un grado de libertad de la muestra al calcular el valor medio \bar{x} , por lo que quedan disponibles $n-1$ grados de libertad. Esta diferencia en la forma de calcular varianzas y desviaciones estándar se refleja en la forma en que invocamos funciones en NumPy. En la celdas 27 y 28 vemos cómo utilizar las funciones `var` y `std` de Numpy para calcular estas magnitudes en ambos casos.

```

CELL 27
# Varianza y desviación estándar poblacional
var = np.var(X2_normal)
std = np.std(X2_normal)
print(f'σ² = {var}, σ = {std}')
-----
σ² = 7.6544508825934985, σ = 2.7666678301873353

```

```

CELL 28
# Varianza y desviación estándar muestral
var_m = np.var(X2_normal, ddof=1)
std_m = np.std(X2_normal, ddof=1)
print(f's²(m) = {var_m}, s(m) = {std_m}')
-----
s²(m) = 7.669790463520539, s(m) = 2.7694386549480634

```

Al igual que en otras funciones, NumPy ofrece también versiones de estas funciones que ignoran la presencia de NaN en los datos (`nanvar()` y `nanstd()`).

1.3.2.4. Error estándar

El error estándar (EE) es la estimación de la desviación estándar de un parámetro que se calcula a partir de una dada muestra. Por ejemplo, si estimamos la media de una distribución a partir de las medias de diversas muestras obtenidas de esa distribución, la desviación estándar de esas medias muestrales es el error estándar de la muestra (EEM).

En las celda 29 construimos una distribución normal con media 5 y desviación estándar 1, y a continuación tomamos `n_muestras = 100` muestras de esa distribución, con `n_valores = 60` valores cada una, y guardamos en el array `medias` los valores medios de cada muestra. El valor

⁷ Un estimador es una función que se calcula a partir de los valores muestrales, utilizado para estimar un parámetro desconocido de la población. Un estimador es sesgado cuando la diferencia entre su esperanza matemática y el valor numérico es diferente de cero.

medio de este array es una buena estimación de la media de la distribución. La dispersión de estos valores se representa en estas figuras como la línea anaranjada que están a una desviación estándar de la media. Para valores normalmente distribuidos, el error estándar de la media (EEM) se define como

$$\text{EEM} = \frac{s}{\sqrt{n}} = \frac{1}{\sqrt{n}} \sqrt{\frac{\sum_{i=1}^n (x - \bar{x})^2}{n - 1}} \quad (1.4)$$

que se representa con las líneas de trazos verdes en la figura.

CELL 29

```

from math import sqrt

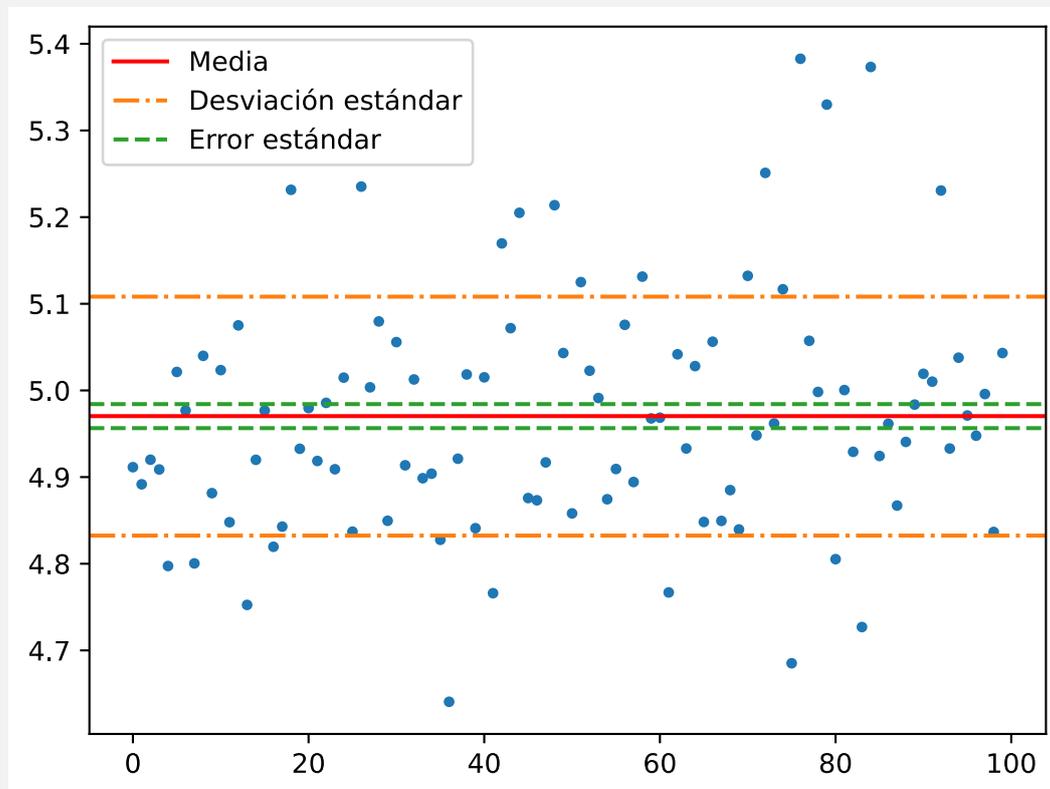
rv = stats.norm(5, 1)

n_muestras = 100
n_valores = 60
medias = np.zeros(n_muestras)
for i in range(n_muestras):
    sample = rv.rvs(n_valores)
    medias[i] = sample.mean()
media = medias.mean()
std = medias.std()
EEM = np.std(medias, ddof=1) / sqrt(n_muestras)
print(f'Media de medias: {media}')
print(f'Error estándar de la media: {EEM}')
plt.plot(medias, '.')
plt.axhline(y = media, color='red', label='Media')
plt.axhline(y = media - std, color='tab:orange', linestyle='dashdot', label='Desviación estándar')
plt.axhline(y = media + std, color='tab:orange', linestyle='dashdot')
plt.axhline(y = media - EEM, color='tab:green', linestyle='--', label='Error estándar')
plt.axhline(y = media + EEM, color='tab:green', linestyle='--')
plt.legend()
plt.show()

```

Media de medias: 4.970356897826361

Error estándar de la media: 0.013853918499575835



El submódulo `stats` de SciPy tiene una función especializada para el cálculo del error estándar de la media (`sem`), que permite operar sobre los diferentes ejes del array de entrada y configurar la manipulación de NaN:

CELL 30

```
stats.sem(medias)
0.013853918499575835
```

1.3.2.5. Intervalos de confianza

Cuando se realiza un análisis estadístico de datos es usual informar el intervalo de confianza de la estimación de un parámetro. El intervalo de confianza (IC) del α % representa el rango que contiene el valor verdadero de ese parámetro con una probabilidad del α %.

Si la distribución muestral es simétrica y unimodal, es posible aproximar el intervalo de confianza de la media por:

$$\text{IC} = \bar{x} \pm Z_{\alpha} \frac{s}{\sqrt{n}}$$

Para determinar el valor de Z_{α} es necesario conocer la distribución teórica del parámetro que se intenta determinar. Sin embargo, para tamaños de muestras n grandes la media muestral está distribuida normalmente, por lo cual la determinación de Z_{α} consiste en encontrar el cuantil $(1 - \alpha)/2$ de la distribución normal. Por ejemplo, para el intervalo de confianza del 95 %:

CELL 31

```
print(stats.norm.interval(.95, np.mean(medias), scale=stats.sem(medias)))
(4.943203716522439, 4.997510079130282)
```

En caso que la muestra sea pequeña ($n < 30$) suele usarse como distribución teórica la distribución t de Student⁸.

CELL 32

```
print(stats.t.interval(.95, np.mean(medias), scale=stats.sem(medias)))
(-0.03567660913926219, 0.03567660913926219)
```

1.4. Test de hipótesis

Es frecuente la necesidad de tomar decisiones sobre poblaciones a partir de la información muestral disponible de las mismas. En estos casos conviene hacer determinados supuestos o hipótesis acerca de las poblaciones que se estudian. Incluso en algunas ocasiones se formulan hipótesis con el solo propósito de rechazarlas. Por ejemplo, si se quiere demostrar que una moneda está “cargada”, se formula la hipótesis de que es “honesta”, lo que significa que la probabilidad

⁸ Se pueden ver más detalles de la distribución t de Student en la [entrada](#) de Wikipedia.

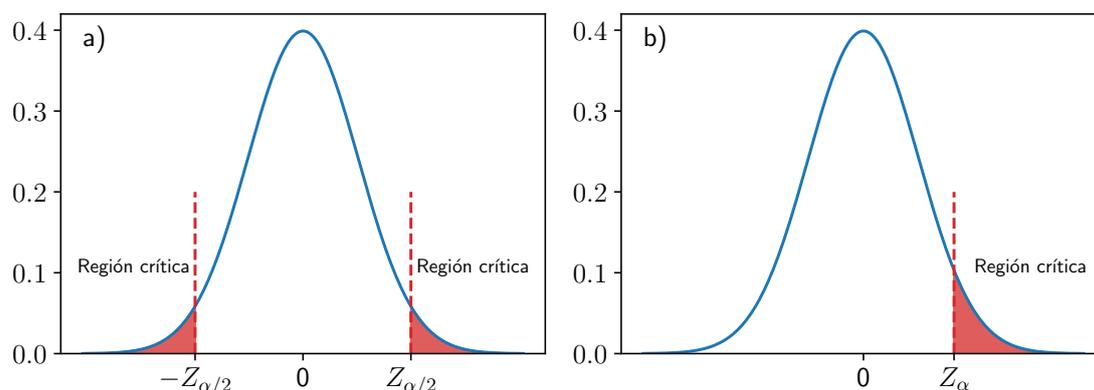


FIGURA 1.3: Test de hipótesis para una distribución normal ($\mu = 0$, $\sigma = 1$). a) Ensayo de dos colas. b) Ensayo de una cola. Las sumas de las áreas en rojo correspondientes a valores en las regiones críticas representan el nivel de significancia del test.

de que un lanzamiento resulte cara o sello es 0,5, y se hace un ensayo para validar (o rechazar) estadísticamente esta hipótesis.

El proceso de ensayo o *test* de hipótesis involucra entonces la evaluación de dos afirmaciones mutuamente excluyentes utilizando para ello los datos de una muestra. La idea principal es que siempre existe una situación que puede ser considerada por defecto, de no cambio (respecto de la opinión actual), de no diferencia, no mejora, *statu quo*, etc., que se denomina “hipótesis nula” o H_0 . Por otro lado, la hipótesis alternativa H_1 asume que hay algún cambio respecto de la situación actual. El propósito del test de hipótesis es determinar si los datos muestrales respaldan la hipótesis nula o la alternativa.

En virtud de la naturaleza estadística del método de ensayo, las decisiones que se toman con un número limitado de datos (la muestra) pueden ser erróneas. Si se rechaza la hipótesis nula cuando debería ser aceptada se produce un “error tipo I”, mientras que si se no se rechaza H_0 siendo esta falsa en la población, se está cometiendo un “error tipo II”. Los ensayos de hipótesis deben diseñarse entonces de forma que minimicen los errores de decisión, pero esto no es tan simple ya que en general, un intento de disminuir un tipo de error produce un incremento en el otro. En la práctica un tipo de error suele tener más importancia que el otro (generalmente el error tipo I), por lo que se suele poner una limitación al error de mayor importancia. La única forma de reducir efectivamente ambos errores es aumentar el tamaño de la muestra, pero esto no siempre es posible.

La máxima probabilidad con la que estaríamos dispuestos a arriesgarnos a un error tipo I se denomina “nivel de significancia del test”. Esta probabilidad se suele establecer antes de generar la muestra, de modo que los resultados obtenidos no tengan influencia en esta decisión. Por lo general se utiliza un nivel de significancia de 0,05 o 0,01. Si, por ejemplo, decidimos usar un nivel de 0,05 o 5 %, habrá 5 oportunidades en 100 de que rechazemos la hipótesis cuando debería ser aceptada. Es decir, siempre que la hipótesis nula sea verdadera, tenemos un 95 % de confianza en que tomaremos la decisión correcta.

Otro aspecto a tener en cuenta se refiere al tipo de comparación a realizar. Por ejemplo, podríamos establecer como hipótesis nula que el tiempo promedio que tarda una persona en llegar a su trabajo es de 30 minutos, o que este tiempo es superior a 30 minutos. En estos casos,

los valores críticos que definen el rechazo de H_0 se toman como los valores extremos de una distribución de dos colas (con el correspondiente nivel de significancia), o como el valor extremo de una sola cola (que en este caso es la superior). En el primer caso, si el nivel de significancia es del 5 %, los valores críticos son los que limitan la distribución en 2,5 % de su área a cada lado, mientras que en el segundo el 5 % corresponde solo al límite superior de la distribución, tal como se ejemplifica en la figura 1.3.

Una vez que se definieron H_0 y H_1 , se deben coleccionar los datos de la muestra (por ejemplo a través de mediciones, encuestas, etc.). El paso siguiente consiste en encontrar un estadístico que se pueda calcular a partir de la muestra y cuya distribución de probabilidad sea conocida asumiendo la hipótesis nula. Luego calculamos, a partir de los datos, la probabilidad de obtener el valor observado del estadístico utilizando la función de distribución que implica asumir la hipótesis nula. Esta probabilidad se denomina p -valor. Si el p -valor es menor que el umbral predeterminado establecido por el nivel de significancia, podemos concluir que los datos observados probablemente no son descriptos por la distribución correspondiente a H_0 , y en este caso podemos rechazar la hipótesis nula en favor de H_1 . El procedimiento para realizar el *test* de hipótesis consiste entonces en los siguientes pasos:

1. Formular la hipótesis nula y la alternativa.
2. Seleccionar un estadístico tal que la distribución de la muestra bajo la hipótesis nula sea conocida (en forma exacta o aproximada).
3. Obtener los datos de la muestra.
4. Calcular el estadístico a partir de los datos y obtener su p -valor bajo la hipótesis nula.
5. Si el p -valor es menor que el nivel de significancia α predeterminado, rechazamos la hipótesis nula. Si es mayor, H_0 no puede ser rechazada.

En el procedimiento anterior, la mayor dificultad la plantea la determinación de la distribución de probabilidad del estadístico. Sin embargo, muchos ensayos de hipótesis pertenecen a unas pocas categorías usuales para las cuales estas distribuciones son conocidas. En la Tabla 1.1 se muestran algunos casos comunes para H_0 , las distribuciones asociadas y las funciones de SciPy que calculan el estadístico y el p -valor.

Veamos un ejemplo. Uno de los casos más frecuentes es el de afirmar que la media μ de una población tiene el valor μ_0 . Obtenemos entonces una muestra de esa población, y utilizamos la media muestral \bar{x} para construir el estadístico

$$Z = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}}$$

donde como es usual, n es el tamaño de la muestra. Si la población es grande y se conoce su varianza σ , es razonable asumir que el estadístico Z está distribuido normalmente. Si la varianza es desconocida podemos reemplazarla por la varianza muestral s . Entonces el estadístico sigue una distribución t de Student, que en el límite de grandes números se aproxima a una distribución normal.

En la celda 33 generamos una distribución normal X con media μ y desviación estándar σ , y luego tomamos una muestra de n valores de esta distribución que almacenamos en `X_muestra`. Entonces, nuestras hipótesis son:

TABLA 1.1: Diversos ensayos de hipótesis y sus distribuciones asociadas.

Hipótesis nula	Distribuciones	Funciones de SciPy
Evaluar si la media de una población tiene un valor dado	Normal o t de Student	<code>stats.ttest_1samp</code>
Evaluar si la media de dos variables aleatorias son iguales	t de Student	<code>stats.ttest_ind</code> , <code>stats.ttest_rel</code>
Evaluar la calidad de ajuste de una distribución continua a datos	Kolmogorov-Smirnov	<code>stats.kstest</code>
Evaluar la frecuencia de ocurrencia de datos categóricos	χ^2	<code>stats.chisquare</code>
Evaluar la igualdad de varianza en muestras de dos o mas variables	F	<code>stats.barlett</code> , <code>stats.levene</code>
Evaluar la no correlación entre dos variables	Beta	<code>stats.pearsonr</code> , <code>stats.spearmanr</code>
Evaluar si dos o mas variables tienen la misma media poblacional (análisis de varianza - ANOVA)	F	<code>stats.f_oneway</code> , <code>stats.kuskal</code>

- $H_0: \mu = \mu_0$
- $H_1: \mu \neq \mu_0$

```

mu_0, mu, sigma = 2.0, 1.82, 0.5
X = stats.norm(mu, sigma)
n = 100
X_muestra = X.rvs(n)
    
```

A continuación calculamos los estadísticos Z (asumiendo una distribución normal con σ conocida) y t (con desviación estándar muestral s). Si definimos que nuestro nivel de significancia es del 5% ($\alpha = 0,05$), calculamos el valor crítico con una distribución normal de dos colas, utilizando para ello la función `norm.ppf()` que devuelve el valor de $Z_{\alpha/2}$, como se muestra en la figura 1.3 (panel a).

```

Z = (X_muestra.mean() - mu_0) / (sigma / np.sqrt(n))
Z
-2.720519876366154
    
```

```

t = (X_muestra.mean() - mu_0) / (X_muestra.std(ddof=1) / np.sqrt(n))
t
-2.7205218379193887
    
```

CELL 36

```
valor_critico = stats.norm().ppf(0.025)
valor_critico
```

```
-1.9599639845400545
```

Podemos ver que los valores de Z y t caen fuera del intervalo $(-1,96, 1,96)$ que corresponde a la zona de aceptación de H_0 , lo cual nos permite rechazar la hipótesis nula con nivel de significancia del 5%. Podemos calcular también los p -valores utilizando para ello tanto la distribución normal como la t de Student, para calcular el área de las dos colas (por eso el factor 2) limitadas por los estadísticos correspondientes. Vemos que ambos valores están por debajo del nivel de significancia preestablecido, lo que confirma la decisión anterior de rechazar H_0 .

CELL 37

```
p_valor_n = 2 * stats.norm().cdf(-abs(Z))
p_valor_t = 2 * stats.t(df=(n-1)).cdf(-abs(t))
p_valor_n, p_valor_t
```

```
(0.0065179353263593535, 0.007700231797543923)
```

Hemos hecho estos cálculos “manualmente” para comprender los detalles de cómo tomar la decisión de aceptar o rechazar la hipótesis nula, pero podemos ahorrarnos este trabajo utilizando directamente las funciones provistas por SciPy para ello. En la celda 38 utilizamos la función `ttest_1samp` de `stats` para comparar la media de la muestra con μ_0 tal como afirma H_0 . Esta función nos devuelve el valor del estadístico y el p -valor:

CELL 38

```
e, p = stats.ttest_1samp(X_muestra, mu_0)
e, p
```

```
(-2.7205218379193883, 0.00770023179754393)
```

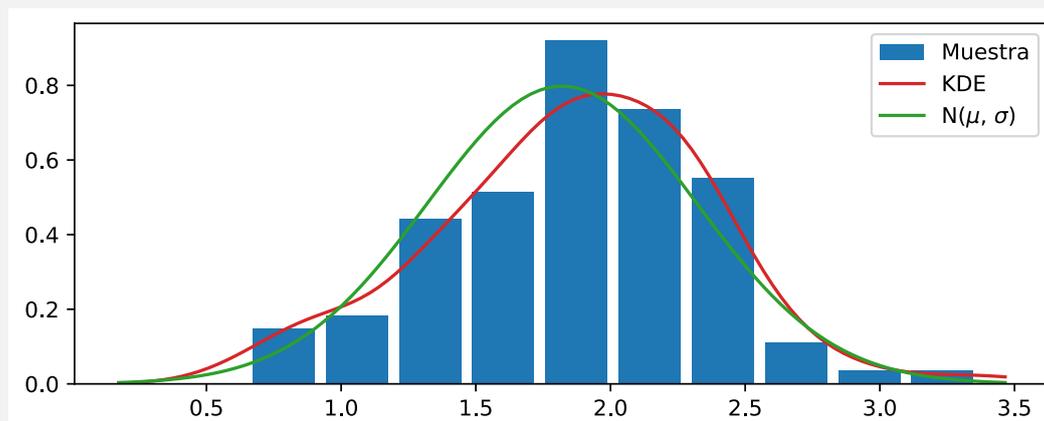
Tal como vimos previamente, el p -valor devuelto nos permite rechazar la hipótesis nula. En la celda siguiente comparamos la distribución de datos de la muestra (histograma), con la distribución real (línea verde) y una distribución aproximada a partir de los datos de la muestra (línea roja, ver sección siguiente 1.5). Pese a que las distribuciones son muy similares, el *test* de hipótesis con solo 100 valores nos permite afirmar (con una probabilidad menor al 5% de cometer un error tipo I) que la media muestral no es μ_0 .

CELL 39

```

from scipy.stats import gaussian_kde
fig, ax = plt.subplots(figsize=(8,3))
ax.hist(X_muestra, density=True, rwidth=0.85, label='Muestra')
kde = gaussian_kde(X_muestra)
x = np.linspace(*X.interval(0.999), num=100)
ax.plot(x, kde(x), color='tab:red', label='KDE')
ax.plot(x, stats.norm(loc=mu, scale=sigma).pdf(x), color='tab:green', label=r'N($\mu$, $\sigma$)')
ax.legend()
plt.show()

```



Otro caso muy frecuente consiste en la comparación de las medias de dos muestras diferentes en la cual la hipótesis nula consiste en asumir que las medias poblacionales de las dos variables aleatorias representadas por las muestras son iguales. Para realizar el *test* de hipótesis, generaremos dos variables aleatorias con distribución normal y medias elegidas aleatoriamente. De cada una de estas distribuciones seleccionaremos muestras de 30 valores:

CELL 40

```

n, sigma = 30, 1.0
mu_1, mu_2 = np.random.random(2)
X_1 = stats.norm(mu_1, sigma)
X_2 = stats.norm(mu_2, sigma)
X_1_muestra = X_1.rvs(n)
X_2_muestra = X_2.rvs(n)

```

La función `ttest_ind()` realiza un ensayo para la hipótesis nula de que dos muestras independientes tienen la misma media asumiendo por defecto que tienen idéntica varianza. La distribución utilizada en este caso es la *t* de Student con un *test* de dos colas, y obtenemos el valor del estadístico y el *p*-valor:

CELL 41

```

t, p = stats.ttest_ind(X_1_muestra, X_2_muestra)
t, p

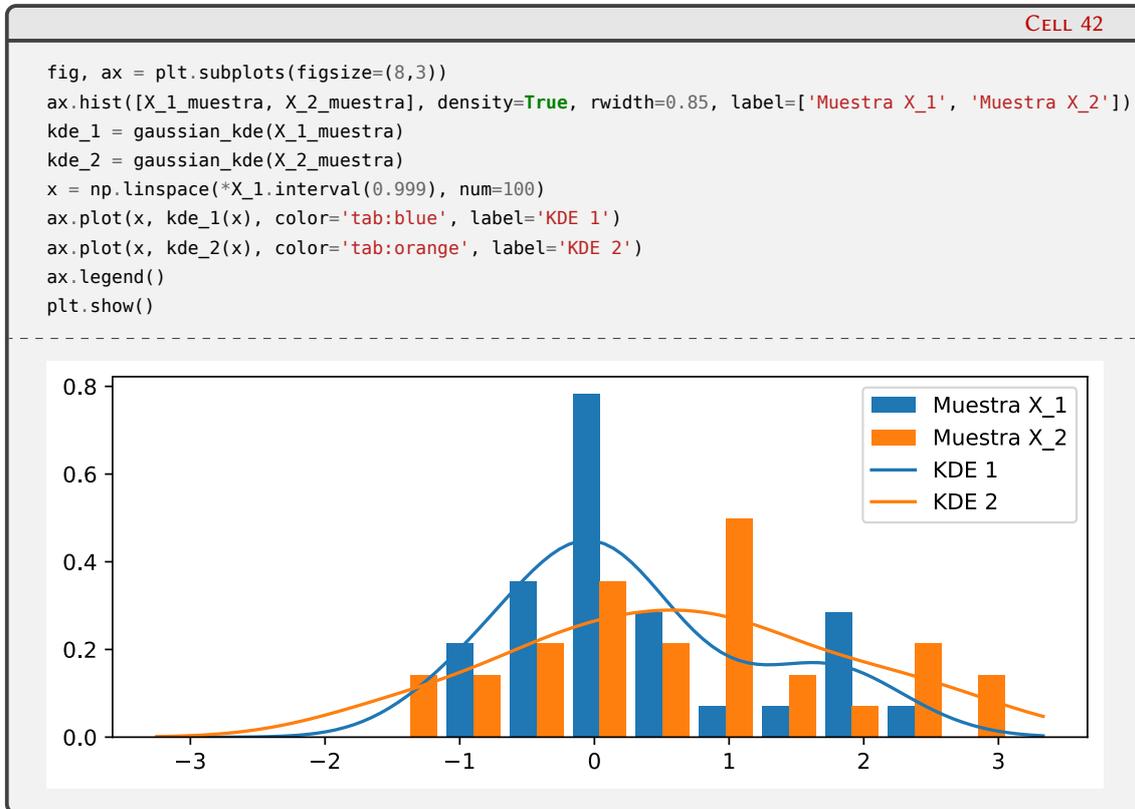
```

```

(-1.2353053231429025, 0.2216974276096801)

```

Vemos que el p -valor está muy por encima de 0,05 (considerando un nivel de significancia del 5%), lo cual no nos permite rechazar la hipótesis nula. En la celda siguiente graficamos los histogramas correspondientes a cada muestra, así como sus distribuciones aproximadas. La escasa cantidad de valores no permite afirmar estadísticamente que las diferencias sean significativas.



Sin embargo, si mostramos los valores elegidos estocásticamente para μ_1 y μ_2 vemos que son bien diferentes:

CELL 43

```

mu_1, mu_2

(0.04154576848611169, 0.4911986914774764)

```

Incrementando la muestra 50 valores, el p -valor obtenido es menor a 0,05, permitiendo en ese caso rechazar H_0 . Esto muestra la importancia del muestreo en la toma de decisiones basada en el *test* de hipótesis.

Por último mostraremos un ejemplo de *test* de hipótesis que generaliza el test anterior a dos o más poblaciones, siendo uno de los casos de análisis de varianza (ANOVA: “ANalysis Of VAriance”), y que se basa en la idea de dividir la varianza total en la varianza entre diferentes grupos y la varianza dentro de cada grupo, y ver si estas distribuciones satisfacen la hipótesis nula de que todos los grupos pertenecen a la misma distribución. Las variables que distinguen los diferentes grupos se denominan habitualmente “factores” o “tratamientos”. En el caso en que cada grupo corresponda a un valor diferente de una sola variable categórica, se utiliza un ANOVA “de una vía” o unidireccional (*one-way ANOVA*), mientras que si se intenta determinar el efecto

de dos variables categóricas, se utiliza un ANOVA bidireccional (o *two-way ANOVA*). Es posible realizar también estudios de tres o más variables categóricas pero son poco comunes y puede ser difícil interpretar los resultados si se utilizan demasiados factores.

Un estudio ANOVA básico consiste en realizar los siguientes pasos. Primeramente se calcula la media de cada uno de los grupos, y a continuación se compara la varianza de estas medias (“intervarianza”) con la varianza promedio dentro de cada grupo (“intravarianza”, no explicada por la variable grupo, intervarianza). Según la hipótesis nula, todas las observaciones proceden de la misma población con lo que la intervarianza será igual a la varianza promedio dentro de los grupos. Se asumen las siguientes suposiciones:

- Las distribuciones están normalmente distribuidas.
- Homogeneidad de varianza, las variaciones dentro de cada grupo son similares en todos los grupos.
- Las observaciones de cada grupo son independientes entre sí, y las observaciones dentro de cada grupo se obtuvieron en forma aleatoria.

En estas condiciones, para un ANOVA unidireccional las hipótesis son:

- H_0 (hipótesis nula): $\mu_1 = \mu_2 = \dots = \mu_k$ (las medias de los k grupos son iguales).
- H_1 (hipótesis alternativa): al menos una media es diferente del resto.

Se construye entonces el estadístico F , que es el cociente entre la varianza de las medias de los grupos y el promedio de la varianza dentro de los grupos, y que tiene una distribución “F de Fisher-Snedecor”. Si H_0 se satisface, F adquiere un valor cercano a 1 debido a que la intervarianza será igual a la intravarianza. Por otro lado, si las medias de los grupos difieren, mayor será la varianza entre medias en comparación con el promedio de las varianzas dentro de los grupos, generando valores de F mayores a 1 y en consecuencia, menor es la probabilidad de que la distribución adquiera valores tan extremos (obteniéndose de este modo un p -valor menor).

Veamos cómo realizar un ANOVA unidireccional mediante la función `f_oneway` del módulo `scipy.stats`. Para ello queremos comparar el rendimiento en un examen de un grupo de 30 alumnos, en función de dónde se sientan cuando asisten a clase: adelante, al medio o atrás. La hipótesis nula consiste en afirmar que la ubicación en el aula no afecta al rendimiento. La hipótesis alternativa indica entonces que la ubicación afecta al rendimiento.

En la celda 44 agrupamos el resultado de una evaluación en tres grupos según la ubicación habitual en el aula durante las clases de 30 alumnos, con 10 alumnos por grupo. Para tener una idea gráfica de estos rendimientos, generamos un gráfico con los *boxplots* de cada grupo a los que hemos superpuesto los resultados individuales, lo que nos da una idea de la dispersión de los resultados.

A continuación, en la celda 45 importamos e invocamos la función `f_oneway` (pasando como argumentos los grupos de datos) que realiza todos los cálculos necesarios para obtener el estadístico F y calcular el p -valor a partir de los resultados muestrales. Vemos que el valor de F es cercano a 1, y el p -valor es suficientemente grande como para evitar el rechazo de H_0 . Podemos interpretar esto afirmando que para los datos disponibles, no hay evidencia estadística que

permita distinguir el efecto de la ubicación en las clases con el rendimiento de los alumnos⁹.

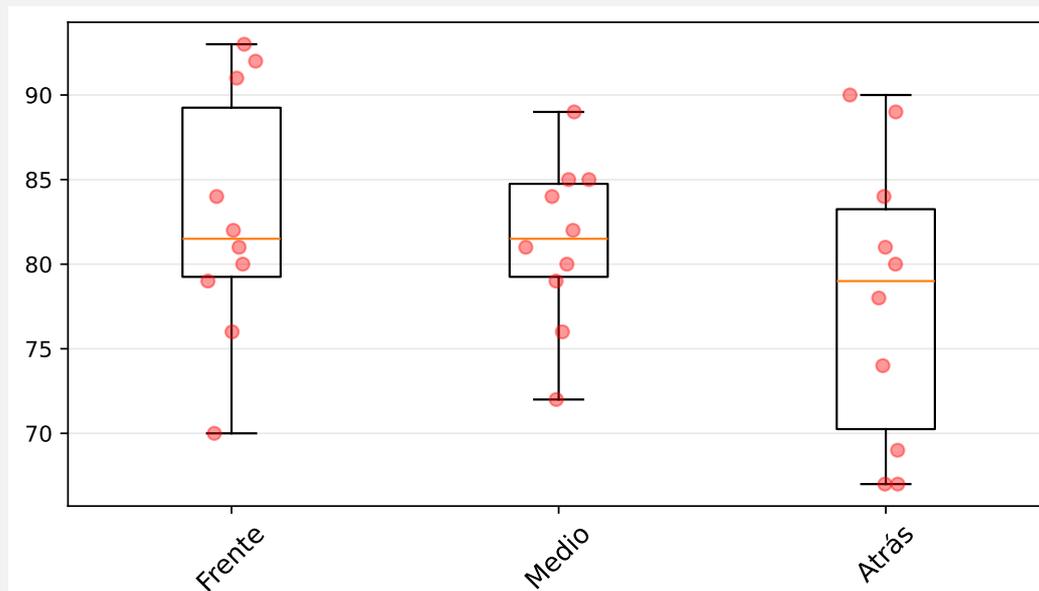
CELL 44

```
frente = [92, 79, 70, 76, 91, 80, 93, 82, 81, 84]
medio = [80, 84, 76, 72, 85, 79, 81, 85, 82, 89]
atras = [74, 67, 81, 78, 69, 80, 89, 84, 90, 67]
grupos = [frente, medio, atras]
nombres_grupos = ['Frente', 'Medio', 'Atrás']

fig, ax = plt.subplots(figsize=(8,4))
bp = plt.boxplot(grupos, sym='+', vert=True, whis=1.5)
plt.setp(bp['boxes'], color='black')
plt.setp(bp['whiskers'], color='black')

for i, g in enumerate(grupos):
    x = np.random.normal(i + 1, 0.05, size=len(g))
    plt.plot(x, g, 'ro', alpha=0.4)
ax.yaxis.grid(True, linestyle='--', which='major', color='lightgrey', alpha=0.5)

# Etiquetamos los boxplots
xtickNames = plt.setp(ax, xticklabels=nombres_grupos)
plt.setp(xtickNames, rotation=45, fontsize=12)
plt.show()
```



CELL 45

```
from scipy.stats import f_oneway
f_oneway(frente, medio, atras)

F_onewayResult(statistic=1.2490092470277412, pvalue=0.30283594892105703)
```

⁹Por supuesto, esto es solo un ejemplo de utilización de la función `f_oneway()` y no representa un estudio real (y serio) del rendimiento en los exámenes según la ubicación de los alumnos en clase.

1.5. Estimación no paramétrica de la FDP

Dada una muestra, podemos asumir que representan valores aleatorios que siguen una determinada distribución de probabilidad caracterizada por ciertos parámetros (por ejemplo, media y varianza de una distribución normal). Estos parámetros pueden ser ajustados a partir de los valores muestrales usando, por ejemplo, una optimización de máxima verosimilitud. De este modo, los métodos estadísticos basados en estas funciones de distribución (tales como los *tests* de hipótesis) representan “métodos paramétricos”. En estos casos estamos realizando una suposición fuerte al determinar la función de distribución que sigue nuestro proceso aleatorio.

Una forma alternativa consiste en intentar describir la distribución a partir de los datos sin hacer ninguna suposición sobre su distribución. Tal vez la manera más frecuente es la de realizar un histograma que divide el rango de valores que contiene la muestra en intervalos iguales (o *bins*) y calcula la cantidad de datos en cada intervalo, o la frecuencia si normalizamos el histograma de modo que el área total de todos los *bins* sume 1. Como resultado obtenemos una descripción empírica discontinua de la distribución que depende del número de intervalos y de sus localizaciones, por lo que la elección de estos parámetros pueden conducir a representaciones cuantitativamente diferentes de la distribución.

Una *kernel density estimation* (KDE) es una función cuyo propósito es el de estimar la función de distribución de probabilidad de una manera no paramétrica, que puede expresarse como:

$$f(x) = \frac{1}{nh} \sum_{i=0}^n K\left(\frac{x - x_i}{h}\right) \quad (1.5)$$

donde como es usual, n es el tamaño de la muestra (el número de valores), x_i son los valores que componen la muestra, K es una función *kernel* y h es el ancho de banda o *bandwidth*. Un kernel es una función de distribución de probabilidad que debe satisfacer tres condiciones:

1. Simetría: $K(x) = K(-x)$.
2. No negativa: $\forall x, K(x) \geq 0$.
3. Área igual a 1: $\int_{-\infty}^{\infty} K(x) dx = 1$.

Las funciones kernel determinan la forma en que se distribuye la influencia de cada observación. En la gran mayoría de los casos se usa un kernel gaussiano, pero existen otras posibilidades como las que se enumeran en la Tabla 1.2 y sus correspondientes representaciones gráficas en la figura 1.4. En SciPy está disponible el kernel gaussiano, mientras que todas las que aparecen en la Tabla 1.2 pueden utilizarse con el módulo `scikit-learn`¹⁰

El ancho de banda h controla el ancho de las distribuciones que se ubican alrededor de cada x_i , lo que afecta en forma importante el suavizado del KDE. Para muestras pequeñas, es recomendable usar un h grande de modo que la inclusión de más puntos provea un contexto adecuado. Para muestras grandes, es mejor usar un h pequeño ya que cada punto está rodeado de muchos otros, lo cual brinda un buen contexto. Existen diversos criterios para obtener el h óptimo, pero para KDE gaussianos existen algunas “reglas del pulgar” como la regla de Silverman:

$$h = \frac{0,9}{\sqrt[5]{n}} \min\left(s, \frac{IQR}{1,349}\right)$$

¹⁰Se puede ver la documentación [aquí](#).

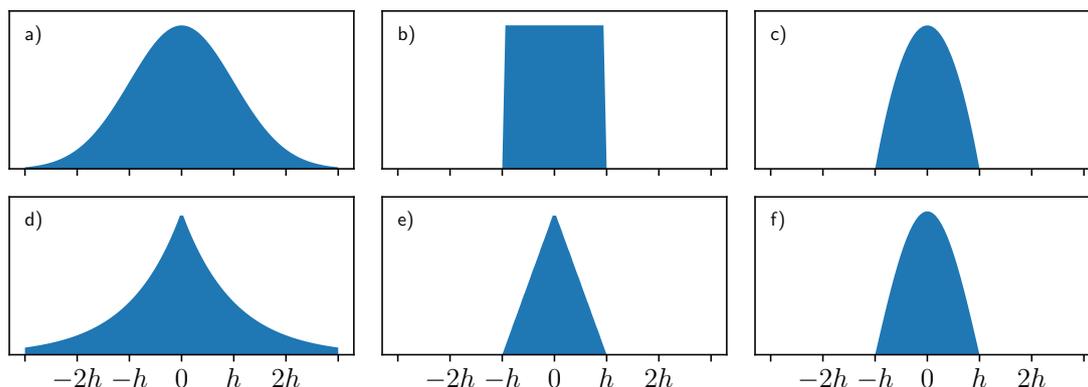


FIGURA 1.4: Algunos kernels más utilizados. a) Gaussiano; b) uniforme; c) Epanechnikov; d) exponencial; e) lineal y f) coseno.

TABLA 1.2: Kernels usuales.

Kernel	Ecuación
Gaussiano	$\frac{1}{h\sqrt{2\pi}} \exp(-x^2/(2h^2))$
Uniforme	$\frac{1}{2h}, \quad x \leq h$
Epanechnikov	$\frac{3}{4h}(1 - x^2/h^2), \quad x \leq h$
Exponencial	$\frac{1}{2h} \exp(- x /h)$
Lineal	$\frac{1}{2h}(1 - x /h), \quad x \leq h$
Coseno	$\frac{\pi}{4h} \cos(\pi x/2h), \quad x \leq h$

(recordemos que *IQR* es el rango intercuartílico, ver subsección 1.3.2.2). Veamos cómo generamos un KDE gaussiano a partir de una muestra y el efecto que tiene la elección de h .

En la celda 46 definimos la función `k_gauss()` que utilizaremos para graficar las funciones gaussianas que componen la KDE (notar que no están normalizadas, ya que las usaremos solo con fines de visualización) y una función auxiliar, `hacer_figura()`, para construir los gráficos de las celdas siguientes. Esta última función grafica un histograma que agrupa los datos en `n_bins`, traza la KDE y por defecto traza también los kernels sobre los valores individuales (que podemos desactivar con `plt_kernels=False`). Adicionalmente, esta función añade una pequeña marca en cada valor de la muestra sobre el eje horizontal (lo que se denomina un *rug plot*). En la celda 49 generamos una muestra pequeña (`x_chica`), establecemos el valor de $h = 0,05$ y calculamos la KDE sobre la muestra).

CELL 46

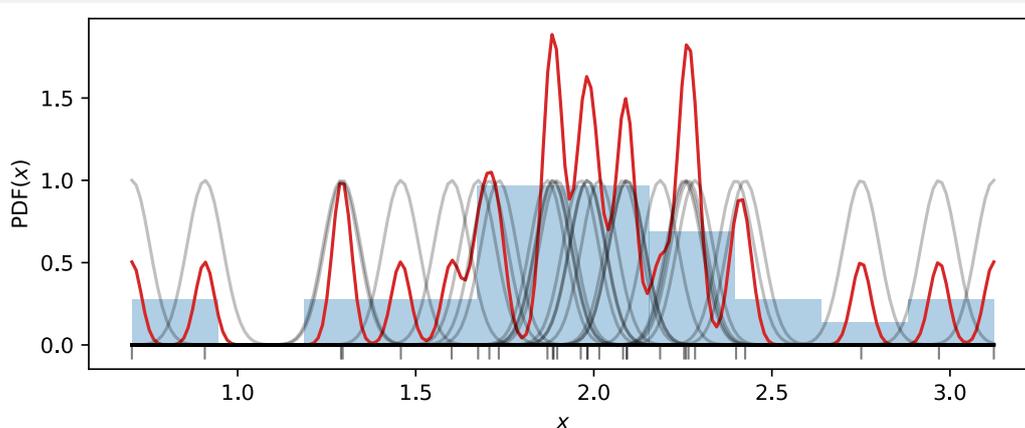
```
def k_gauss(x, xi, h):
    return np.exp(-(x-xi)**2/(2*h**2))

def hacer_figura(datos, kde, n_bins, plot_kernels=True):
    """Función que grafica un histograma de los datos y el KDE asociado,
    junto con las funciones gaussianas de cada valor y un rug plot."""
    fig, ax = plt.subplots(figsize=(8,3))
    ax.hist(datos, bins=n_bins, density=True, alpha=0.35)
    x_c = np.linspace(datos.min(), datos.max(), num=200)
    ax.plot(datos, np.full_like(datos, -0.05), 'k|', markeredgewidth=1.0, alpha=0.5)
    ax.plot(x_c, kde(x_c), color='tab:red', label='KDE')
    if plot_kernels:
        for x_i in datos:
            ax.plot(x_c, k_gauss(x_c, x_i, h), color='k', alpha=0.25)

    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'PDF($x$)')
    plt.show()
```

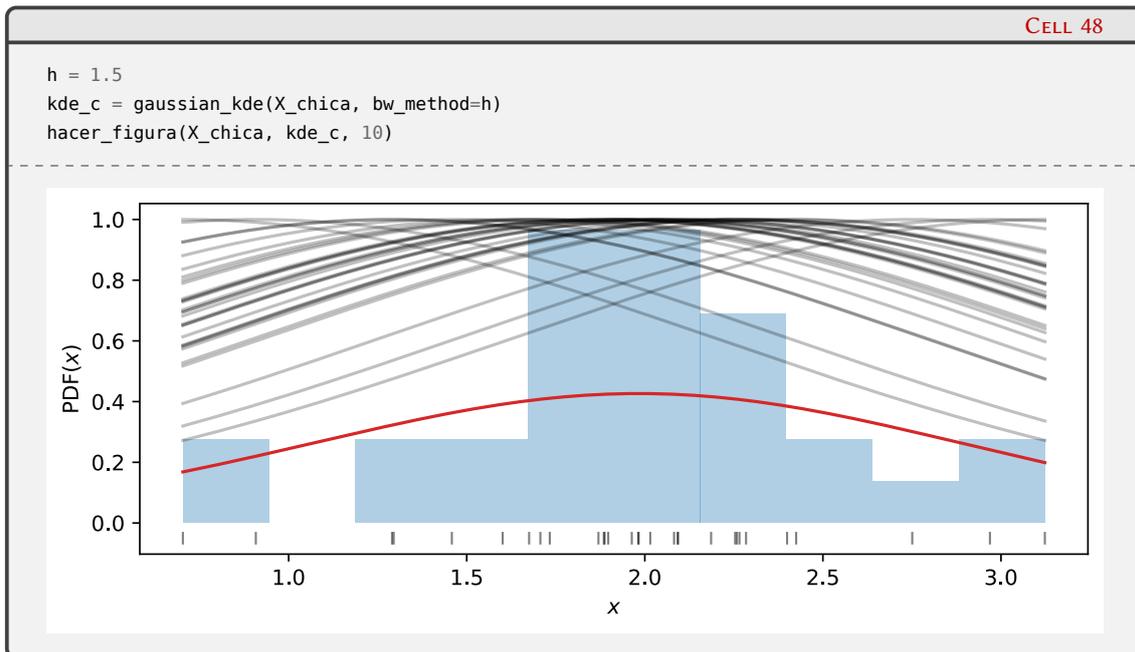
CELL 47

```
n_chico = 30
X_chica = X.rvs(n_chico)
h = 0.05
kde_c = gaussian_kde(X_chica, bw_method=h)
hacer_figura(X_chica, kde_c, 10)
```

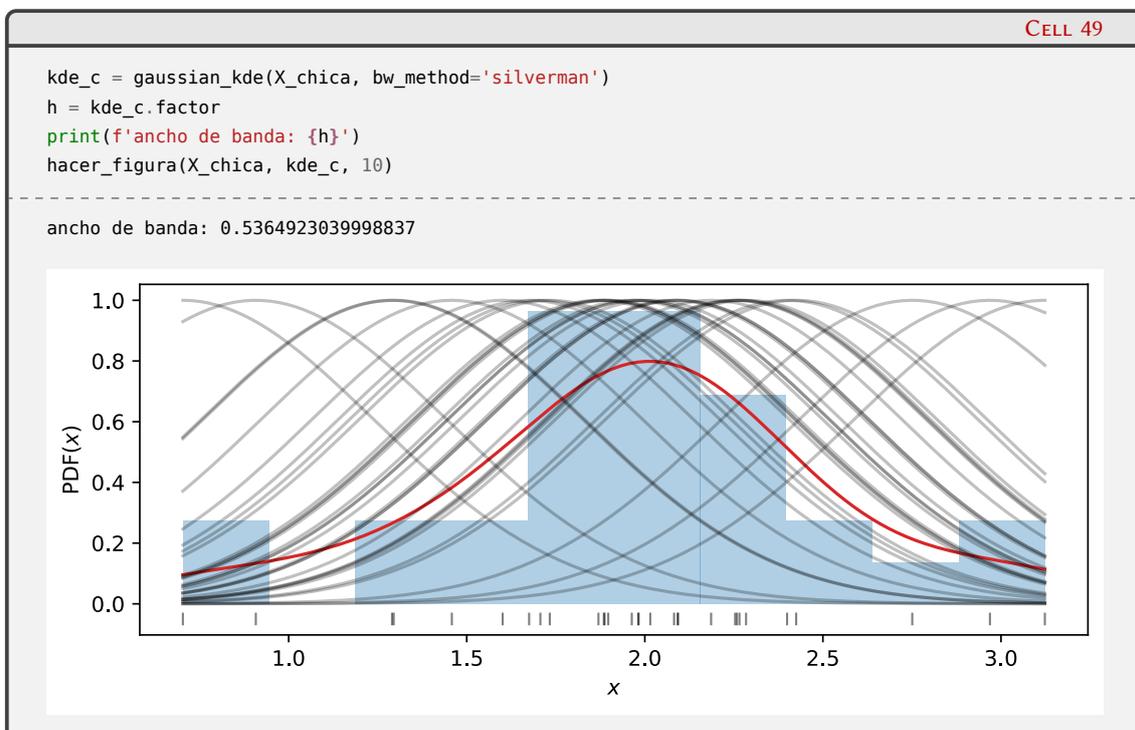


En la figura generada, las curvas grises tenues representan cada gaussiana centrada en los valores de la muestra, que componen (cuando están correctamente normalizadas, no como en la figura) la función KDE que se muestra en línea roja. Dado que usamos un valor de h muy chico, cada función gaussiana abarca una zona angosta alrededor de los valores x_i , dando como resultado una KDE excesivamente estructurada.

Si ahora elegimos un valor de h exageradamente grande, como en la celda 48, vemos que las gaussianas individuales son tan anchas que todas abarcan al conjunto entero que compone la muestra, y en consecuencia la KDE resulta muy “plana”.



Finalmente, en la celda 49 seleccionamos como argumento de `bw_method` la cadena `'silverman'`, que aplicará esa regla para determinar el h óptimo. Podemos mostrar el valor de h obtenido mediante el atributo `factor` de la KDE, que ahora resulta una distribución que representa mejor la distribución de los datos. Tal como dijimos previamente, dado que la muestra es chica resulta un ancho de banda relativamente grande de modo que las gaussianas individuales son relativamente anchas.



Para finalizar, veamos un ejemplo en el que tenemos una muestra bimodal de $n = 1000$

valores que construimos en la celda 50. En este caso, el método por defecto para el cálculo del ancho de banda óptimo es el de Scott, utilizado en el cálculo de la KDE de la celda 51. Se puede ver que la KDE obtenida representa de una manera suave el histograma de los datos.

```
CELL 50
n = 1000
X_1 = stats.norm(2.5, 0.5)
X_muestra = X.rvs(n)
X_muestra[int(0.3 * n):] += 2
```

```
CELL 51
kde = gaussian_kde(X_muestra)
print(f'Ancho de banda: {kde.factor}')
hacer_figura(X_muestra, kde, 50, plot_kernels=False)
```

Ancho de banda: 0.251188643150958

1.6. Lecturas recomendadas

- Thomas Haslwanter. *An Introduction to Statistics with Python*. 1.^a ed. Statistics and Computing. Cham, Switzerland: Springer International Publishing, 2016. DOI: [10.1007/978-3-319-28316-6](https://doi.org/10.1007/978-3-319-28316-6).
- John J. Schiller, R. Alu Srinivasan y Murray R. Spiegel. *Schaum's outline of probability and statistics*. 4.^a ed. Schaum's outline series. New York, NY: McGraw-Hill Professional, dic. de 2012.

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [6].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tenemos más de esos!

Bibliografía

- [1] URL: <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>.
- [2] Makoto Matsumoto y Takuji Nishimura. «Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator». En: *ACM Trans. Model. Comput. Simul.* 8.1 (ene. de 1998), págs. 3-30. DOI: [10.1145/272991.272995](https://doi.org/10.1145/272991.272995). URL: <https://doi.org/10.1145/272991.272995>.
- [3] Melissa E. O’Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Inf. téc. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, sep. de 2014.
- [4] Thomas Haslwanter. *An Introduction to Statistics with Python*. 1.^a ed. Statistics and Computing. Cham, Switzerland: Springer International Publishing, 2016. DOI: [10.1007/978-3-319-28316-6](https://doi.org/10.1007/978-3-319-28316-6).
- [5] John J. Schiller, R. Alu Srinivasan y Murray R. Spiegel. *Schaum’s outline of probability and statistics*. 4.^a ed. Schaum’s outline series. New York, NY: McGraw-Hill Professional, dic. de 2012.
- [6] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: <https://www.python.org/dev/peps/pep-0020/>.