Manual de Python

Docente

2026-02-08

Table of contents

Manual de Python

Presentación del Manual

Este manual te enseña a programar en **Python desde cero**, paso a paso, con un enfoque práctico. No necesitas haber programado antes.

Aquí vas a ver **código real**, explicado con claridad: qué hace, cómo se usa y qué errores evitar.

¿Qué vas a obtener después de este manual?

Al finalizar, vas a poder:

- Leer y escribir código en Python con confianza.
- Trabajar con datos usando `pandas` y `numpy`.
- Hacer gráficos básicos con `matplotlib`.
- Entender la lógica de programación (condicionales, bucles y funciones).
- Usar distribuciones, intervalos de confianza y simulación de Monte Carlo desde Python.
- Ejecutar y adaptar código en tu propio entorno (VSCode) o en la nube (Google Colab).

Lecturas recomendadas (HTML, PDF y Colab)

- **HTML:** navegación rápida, lectura cómoda y capítulos ordenados.
- **PDF:** ideal para imprimir o leer offline.
- **Google Colab:** ejecutas el código sin instalar nada (recomendado para practicar).

Recomendación práctica: **lee el capítulo en HTML/PDF y practica en Colab**.

Índice del Manual

Cada capítulo tiene su botón para abrirlo en **Google Colab**.

Capítulo 1 — Empezando en Python

Aprendes a ejecutar código, usar variables, importar módulos y entender tu directorio de trabajo.

Leer Capítulo 1 Abrir en Google Colab

Capítulo 2 — Objetos en Python

Entiendes tipos de datos, operaciones, estructuras básicas (listas/diccionarios) y cómo inspeccionar objetos.

Leer Capítulo 2 Abrir en Google Colab

Capítulo 3 — Datos externos

Practicas operadores, comparaciones, manejo de textos, y estructuras comunes con ejemplos claros.

Leer Capítulo 3 Abrir en Google Colab

Capítulo 4 — Gráficos básicos con Matplotlib

Aprendes a graficar de forma profesional: líneas, dispersión, etiquetas, títulos y leyendas.

Leer Capítulo 4 Abrir en Google Colab

Capítulo 5 — Estadísticas descriptivas

Generas resúmenes, medidas descriptivas y tablas usando Python (enfoque 100% en programación).

Leer Capítulo 5 Abrir en Google Colab

Capítulo 6 — Distribuciones de probabilidad

Aprendes a usar distribuciones (PMF/PDF/CDF), cuantiles (`ppf`) y simulación desde distribuciones.

Leer Capítulo 6 Abrir en Google Colab

Capítulo 7 — Intervalos de confianza e inferencia (t-test)

Construyes intervalos de confianza y pruebas t paso a paso (manual y automático), evitando errores típicos.

Leer Capítulo 7 Abrir en Google Colab

Capítulo 8 — Python avanzado: condicionales, bucles, funciones y SymPy

Dominas patrones de programación (if/for/while/funciones) y tienes una sección opcional de SymPy.

Leer Capítulo 8 Abrir en Google Colab

Capítulo 9 — Simulación de Monte Carlo

Simulas promedios muestrales (TLC), cobertura de intervalos y error Tipo I con Monte Carlo.

Leer Capítulo 9 Abrir en Google Colab

Capítulo 1 — Empezando en Python

Abrir este capítulo en Google Colab

Objetivos del capítulo

Al terminar este capítulo vas a poder:

- escribir y ejecutar tus primeras líneas de Python,
- crear variables y usarlas en cálculos,
- imprimir resultados con `print()` y f-strings,
- importar módulos (`math`, `os`) y usarlos correctamente,
- entender qué es el **directorio de trabajo** y por qué importa.

---

1. Comentarios, texto y variables

Comentarios

Los comentarios son líneas que **Python no ejecuta**. Sirven para explicar el código o dejar recordatorios.

```
# Esto es un comentario: Python lo ignora.
# Úsalo para explicar qué hace tu código.
```

Glosario

- **comentario**: texto que Python ignora (`# ...`).
- **string**: texto entre comillas (`"hola"`).
- **variable**: nombre que guarda un valor (`x = 10`).
- **módulo**: "paquete" con código reutilizable (`math`, `os`).
- **import**: traer un módulo para usarlo (`import math`).
- **directorio de trabajo**: carpeta actual donde Python busca/guarda archivos.

> **Tip**
>
> **Tip rápido** Si una línea empieza con `#` y "no pasa nada", no es un error: es un comentario.

Texto (strings)

En Python, el texto se escribe entre comillas simples o dobles.

```
print("Hola mundo")
```

```
Hola mundo
```

- **Qué hace:** muestra el texto en pantalla.
- **Cómo se usa:** `print("...")` o `print('...')`.

> **Warning**
>
> **Error típico** Escribir texto sin comillas: `print(Hola)` → Python cree que `Hola` es una **variable** y lanza `NameError`.

Operaciones matemáticas

Python funciona como una calculadora.

```
1 + 1
```

```
2
```

> **Tip**
>
> **Prioridad de operaciones** Python respeta la prioridad matemática: `()` → `**` → `* /` → `+ -`.

Variables

Una variable es un nombre que "apunta" a un valor.

```
var1 = 1 + 1
var2 = 5 * (4 - 1) ** 2
```

Para ver el contenido de una variable:

```
var1
```

```
2
```

```
var2
```

```
45
```

> **Note**
>
> **Idea clave** En Python, `=` no significa "igualdad matemática". Significa **asignación**: "guarda esto con este nombre".

Imprimir resultados con `print()` y f-strings

Cuando quieras mostrar texto + variables de forma clara, usa **f-strings**:

```
print(f"Uno más uno: {var1}")
print(f"El valor de var2 es: {var2}")
```

```
Uno más uno: 2
El valor de var2 es: 45
```

Sintaxis importante - `f"..."` activa el modo *f-string*. - `{var1}` inserta el valor de `var1` en el texto.

También puedes forzar un salto de línea con `\n`:

```
print(f"Uno más uno:\n{var1}")
```

```
Uno más uno:
2
```

---

2. Módulos y directorio de trabajo

¿Qué es un módulo?

Un **módulo** es código ya escrito y listo para usar (funciones y constantes). Python trae muchos módulos "de fábrica".

Importar y usar `math`

Primero importamos el módulo `math`. Usaremos un **alias** (`m`) para escribir menos.

```
import math as m
```

Ahora puedes usar funciones/constantes con `m.`:

```python
raiz = m.sqrt(16)   # raíz cuadrada
pi = m.pi           # constante pi
e = m.e             # constante e

print(f"sqrt(16) = {raiz}")
print(f"pi = {pi}")
print(f"e = {e}")
```

```
sqrt(16) = 4.0
pi = 3.141592653589793
e = 2.718281828459045
```

> **Warning**
>
> **Errores típicos con módulos** - Usar `m.sqrt(...)` sin importar antes → `NameError: name 'm' is not defined`. - Escribir `m.pi()` → error, porque `pi` **no es una función** (no lleva paréntesis).

Directorio de trabajo con `os`

Para trabajar con carpetas y archivos usamos el módulo `os`.

```python
import os
```

El **directorio de trabajo** es la carpeta desde donde Python "mira" para leer/guardar archivos.

Ver el directorio actual:

```python
os.getcwd()
```

```
'C:\\Users\\USUARIO\\Documents\\QUARTO\\Manual de Python\\manual\\chapters'
```

Listar archivos de la carpeta actual (limitamos a los primeros 10 para que se vea limpio):

```
archivos = sorted(os.listdir())
archivos[:10]
```

```
['01-empezando-en-python.qmd',
 '01-empezando-en-python.quarto_ipynb',
 '01-empezando-en-python_files',
 '02-objetos-en-python.qmd',
 '02-objetos-en-python.quarto_ipynb',
 '03-datos-externos.qmd',
 '03-datos-externos.quarto_ipynb',
 '04-graficos-basicos-matplotlib.qmd',
 '04-graficos-basicos-matplotlib.quarto_ipynb',
 '05-estadisticas-descriptivas.qmd']
```

Cambiar de directorio (ejemplo, **no lo ejecutes** si no sabes lo que haces):

```
os.chdir("C:/ruta/a/tu/carpeta")
```

> Tip
>
> **Tip docente** Si Python "no encuentra un archivo", casi siempre es porque: 1) estás en el directorio equivocado, o 2) la ruta/nombre del archivo está mal.

---

3. Instalación de librerías (opcional)

En capítulos futuros verás librerías que pueden no venir instaladas por defecto.

> **Note**
>
> **Regla práctica** - En **VSCode** instalas con `pip install ...` en la terminal. - En **Colab** instalas con `!pip install ...` dentro de una celda.

VSCode (terminal)

```
#| eval: false
pip install wooldridge yfinance pandas_datareader
```

Google Colab

```
!pip -q install wooldridge yfinance pandas_datareader
```

Ejercicios propuestos

1) **Tu primera impresión con f-string** Crea dos variables: `nombre` (texto) y `edad` (número). Imprime una frase usando f-string.

   - **Pista:** `print(f"...{nombre}...{edad}...")`
   - **Respuesta esperada:** una frase bien formada con tu nombre y tu edad.

2) **Operaciones básicas** Crea `a = 10` y `b = 3`. Calcula e imprime:

   - `a + b`
   - `a * b`
   - `a ** b`
   - **Pista:** `**` es potencia.

- **Respuesta esperada:** `13`, `30`, `1000`.

3) **Usa `math`** Calcula la raíz cuadrada de 81 usando `math`.

   - **Pista:** `m.sqrt(81)`.
   - **Respuesta esperada:** `9.0`.

4) **Chequea tu directorio de trabajo** Muestra tu directorio actual y lista los primeros 10 archivos.

   - **Pista:** `os.getcwd()` y `sorted(os.listdir())[:10]`
   - **Respuesta esperada:** una ruta + una lista con nombres de archivos/carpetas.

Glosario (para Colab)

- **comentario**: texto que Python ignora (`# ...`).
- **string**: texto entre comillas (`"hola"`).
- **variable**: nombre que guarda un valor (`x = 10`).
- **módulo**: "paquete" con código reutilizable (`math`, `os`).
- **import**: traer un módulo para usarlo (`import math`).
- **directorio de trabajo**: carpeta actual donde Python busca/guarda archivos.

Capítulo 2 — Objetos en Python (variables, listas, numpy y pandas)

Abrir este capítulo en Google Colab

Objetivos del capítulo

Al terminar este capítulo vas a poder:

- identificar tipos de datos comunes (`str`, `int`, `float`, `bool`),
- crear y manipular **listas** y **diccionarios**,
- trabajar con arreglos `numpy` (vectores y matrices) usando índices y condiciones,
- crear y explorar tablas con `pandas` (`DataFrame`) y seleccionar datos con `[]`, `.loc` y `.iloc`.

---

1. Variables y objetos

En Python, **todo es un objeto**. Lo que cambia es el **tipo** (*type*), y el tipo determina qué operaciones puedes hacer.

Texto, números y booleanos

Glosario

- **objeto**: cualquier "cosa" en Python (número, texto, lista, tabla…).
- **tipo (type)**: la categoría de un objeto (`int`, `float`, `str`, `bool`).
- **índice (index)**: posición de un elemento. En Python empieza en `0`.
- **método**: función "pegada" a un objeto, por ejemplo `lista.sort()`.
- **mutabilidad**: si un objeto puede cambiar por dentro (listas sí; tuplas no).
- **alias**: dos nombres apuntando al mismo objeto (común con listas).

```python
texto = "Hola"
entero = 10
decimal = 3.14
bandera = True  # también existe False

type(texto), type(entero), type(decimal), type(bandera)
```

```
(str, int, float, bool)
```

> **Sintaxis clave (variables)**
>
> - `nombre = valor` guarda un valor con un nombre.
> - Las comillas (`"..."` o `'...'`) indican **texto**.
> - `True` y `False` son booleanos (no llevan comillas).

1.1 Listas

Una **lista** guarda varios elementos en orden.

```python
lista_texto = ["Ana", "Luis", "María"]
lista_numeros = [10, 20, 30, 40]
```

> **Sintaxis clave (listas)**
>
> - Se crean con corchetes: `[...]`
> - Los elementos se separan con comas: `[10, 20, 30]`
> - Una lista puede guardarse en una variable: `lista = [...]`

Indexación (empieza en 0)

```
lista_numeros[0]   # primer elemento
```

```
10
```

```
lista_numeros[2]   # tercer elemento
```

```
30
```

> **Sintaxis clave (indexación)**
>
> - `lista[i]` devuelve el elemento en la posición `i`.
> - El primer elemento es `i = 0`.
> - El último elemento se puede pedir con `lista[-1]`.

> **Warning**
>
> **Error típico**
> Pedir un índice que no existe da `IndexError`.

Slicing (rangos)

```
lista_numeros[1:4]  # del segundo al cuarto (4 no se incluye)
```

```
[20, 30, 40]
```

> **Sintaxis clave (slicing)**
>
> - `lista[inicio:fin]` toma un rango.
> - `inicio` se incluye y `fin` **no** se incluye.
> - Ejemplo: `[1:4]` toma posiciones `1, 2, 3`.

Modificar un elemento

```
lista_numeros[2] = 99
lista_numeros
```

```
[10, 20, 99, 40]
```

> **Sintaxis clave (modificar)**
>
> - `lista[i] = nuevo_valor` reemplaza el elemento en la posición `i`.

1.2 Funciones vs métodos

```
max(lista_numeros)
```

```
99
```

```
lista_numeros.sort()
lista_numeros
```

```
[10, 20, 40, 99]
```

> **Sintaxis clave (función vs método)**
>
> - **Función:** `funcion(objeto)` → ejemplo: `max(lista)`
> - **Método:** `objeto.metodo()` → ejemplo: `lista.sort()`

1.3 Cuidado: alias al copiar listas

```
a = [1, 2, 3]
b = a        # alias (mismo objeto)
b[0] = 999

a, b
```

```
([999, 2, 3], [999, 2, 3])
```

```
a = [1, 2, 3]
b = a.copy() # copia real
b[0] = 999

a, b
```

```
([1, 2, 3], [999, 2, 3])
```

> **i** Sintaxis clave (copiar)
>
> - `b = a` crea un **alias** (no copia).
> - `b = a.copy()` crea una copia (superficial) de la lista.

1.4 Diccionarios

```
estudiante = {"nombre": "Juan", "nota": 14}
estudiante
```

```
{'nombre': 'Juan', 'nota': 14}
```

Sintaxis clave (diccionario)

- Se crea con llaves: `{...}`
- Dentro van pares `clave: valor`
- Ejemplo: `{"nombre": "Juan", "nota": 14}`

Acceder por clave:

```
estudiante["nota"]
```

```
14
```

Modificar / agregar claves:

```
estudiante["nota"] = estudiante["nota"] + 6
estudiante["edad"] = 20
estudiante
```

```
{'nombre': 'Juan', 'nota': 20, 'edad': 20}
```

Sintaxis clave (usar diccionarios)

- `dicc["clave"]` devuelve el valor de esa clave.
- `dicc["clave"] = valor` crea o actualiza una clave.

---

2. Arreglos con numpy

```
import numpy as np
```

Sintaxis clave (import)

- `import paquete` importa un paquete.
- `import paquete as alias` crea un alias para es-

cribir menos.
- Ejemplo: `import numpy as np` → usamos `np` en lugar de `numpy`.

Vectores (array 1D)

```
x = np.array([10, 20, 30, 40])
x
```

```
array([10, 20, 30, 40])
```

Sintaxis clave (crear array)

- `np.array([ ... ])` convierte una lista en un `array` de numpy.
- Para 1D: `np.array([1,2,3])`

Indexación y selección:

```
x[2]      # un elemento
```

```
np.int64(30)
```

```
x[[0, 2]]  # varios índices
```

```
array([10, 30])
```

```
x[x > 20]  # filtro booleano
```

```
array([30, 40])
```

Sintaxis clave (selección en numpy)

- `x[i]` un elemento
- `x[[i, j]]` varios índices (lista de índices)
- `x[condicion]` filtra por condición (devuelve solo los que cumplen)

Matrices (array 2D)

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

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Sintaxis clave (2D)

- Para 2D usas listas dentro de listas: `[[...], [...]]`
- `A.shape` devuelve `(filas, columnas)`
- `A[fila, columna]` accede a un elemento

```
A.shape
```

```
(3, 4)
```

```
A[1, 2]
```

```
np.int64(7)
```

```
A[:, 1:3]
```

```
array([[ 2,  3],
       [ 6,  7],
       [10, 11]])
```

> **Sintaxis clave (`:` en numpy)**
>
> - `:` significa "todo"
> - `A[:, 1:3]` = todas las filas, columnas 1 a 2 (porque 3 no se incluye)

Operaciones con matrices

```
B = np.array([[1, 1, 1],
              [2, 2, 2]])
C = np.array([[10, 10, 10],
              [20, 20, 20]])

B + C
```

```
array([[11, 11, 11],
       [22, 22, 22]])
```

```
B * C
```

```
array([[10, 10, 10],
       [40, 40, 40]])
```

```
M = np.array([[1, 2],
              [3, 4]])
N = np.array([[10, 0],
              [0, 10]])

M @ N
```

```
array([[10, 20],
       [30, 40]])
```

⚠ Sintaxis clave (`*` vs `@`)

- `*` = multiplicación **elemento a elemento**
- `@` = multiplicación **matricial**

---

3. Tablas con pandas

```
import pandas as pd
```

Crear un DataFrame

```
df = pd.DataFrame({
    "ventas": [100, 120, 90, 130, 110],
    "clima":  ["sol", "sol", "lluvia", "sol", "lluvia"],
    "clientes": [20, 25, 18, 30, 22]
})
df
```

	ventas	clima	clientes
0	100	sol	20
1	120	sol	25
2	90	lluvia	18
3	130	sol	30
4	110	lluvia	22

Sintaxis clave (DataFrame)

- `pd.DataFrame({...})` crea una tabla.
- Dentro usas un diccionario: cada clave es una **columna**.
- Las listas deben tener el mismo tamaño (mismo número de filas).

```
df.index = ["d1", "d2", "d3", "d4", "d5"]
df
```

	ventas	clima	clientes
d1	100	sol	20
d2	120	sol	25
d3	90	lluvia	18
d4	130	sol	30
d5	110	lluvia	22

Selección de columnas y filas

```
df["ventas"]
```

```
d1    100
d2    120
d3     90
d4    130
d5    110
Name: ventas, dtype: int64
```

```
df[["ventas", "clientes"]]
```

	ventas	clientes
d1	100	20
d2	120	25
d3	90	18
d4	130	30
d5	110	22

Sintaxis clave (seleccionar columnas)

- `df["col"]` una columna
- `df[["col1", "col2"]]` varias columnas (lista de nombres)

```
df.loc["d2":"d4"]
```

	ventas	clima	clientes
d2	120	sol	25
d3	90	lluvia	18
d4	130	sol	30

```
df.iloc[1:4]
```

	ventas	clima	clientes
d2	120	sol	25
d3	90	lluvia	18
d4	130	sol	30

```
df.loc["d3", "clientes"]
```

```
np.int64(18)
```

```
df.iloc[2, 2]
```

```
np.int64(18)
```

Sintaxis clave (`loc` vs `iloc`)

- `.loc[filas, columnas]` usa **etiquetas** (nombres)
- `.iloc[filas, columnas]` usa **posiciones** (0,1,2,…)
- Un rango `1:4` incluye 1,2,3 (4 no se incluye)

Explorar rápidamente

```
df.head()
```

	ventas	clima	clientes
d1	100	sol	20
d2	120	sol	25
d3	90	lluvia	18
d4	130	sol	30
d5	110	lluvia	22

```
df.tail()
```

	ventas	clima	clientes
d1	100	sol	20
d2	120	sol	25
d3	90	lluvia	18
d4	130	sol	30
d5	110	lluvia	22

```
df.select_dtypes(include="number").describe()
```

	ventas	clientes
count	5.000000	5.000000
mean	110.000000	23.000000

	ventas	clientes
std	15.811388	4.690416
min	90.000000	18.000000
25%	100.000000	20.000000
50%	110.000000	22.000000
75%	120.000000	25.000000
max	130.000000	30.000000

i Sintaxis clave (resumen)

- `head()` muestra las primeras filas
- `tail()` muestra las últimas filas
- `select_dtypes(include="number")` se queda solo con columnas numéricas
- `describe()` resume estadísticas (conteo, media, etc.)

Groupby y rezagos

```
df["clima"] = df["clima"].astype("category")
df.dtypes
```

```
ventas          int64
clima        category
clientes        int64
dtype: object
```

```
df.groupby("clima")["ventas"].mean()
```

```
clima
lluvia    100.000000
sol       116.666667
Name: ventas, dtype: float64
```

Sintaxis clave (`groupby`)

- `df.groupby("col")` agrupa por una columna
- `["ventas"]` selecciona la variable a resumir
- `.mean()` calcula el promedio por grupo

```
df["ventas_lag1"] = df["ventas"].shift(1)
df
```

	ventas	clima	clientes	ventas_lag1
d1	100	sol	20	NaN
d2	120	sol	25	100.0
d3	90	lluvia	18	120.0
d4	130	sol	30	90.0
d5	110	lluvia	22	130.0

Sintaxis clave (`shift`)

- `shift(1)` mueve los valores 1 fila hacia abajo
- sirve para crear **rezagos** (valor anterior)

Ejercicios propuestos

1) **Índices y slicing con intención**
Crea la lista `L = [2, 4, 6, 8, 10, 12]`.
Construye una nueva lista `R` que contenga **solo** los elementos de `L` que están en posiciones pares (0, 2, 4, …) y luego **reemplaza** el último elemento de `R` por `999`.

   Respuesta esperada: `R` debe quedar como `[2, 6, 999]`.

2) **Alias vs copia (comprobación)**
Crea `a = [1, 2, 3]`. Luego crea `b = a` y `c = a.copy()`. Cambia `b[0] = 100` y `c[1] = 200`. Imprime `a`, `b`, `c`.

   Respuesta esperada:

- `a` y `b` deben ser iguales y empezar con `100` (porque comparten el mismo objeto).

- `c` debe ser distinto (solo `c` debe tener `200` en la segunda posición).
  Ejemplo coherente: `a=[100, 2, 3]`, `b=[100, 2, 3]`, `c=[1, 200, 3]`.

3) **Diccionario como "registro"**
   Crea un diccionario `persona` con `{"nombre": "Ana", "edad": 19}`.
   Luego actualiza `edad` sumándole 1 y agrega una nueva clave `activo` con valor `True`.

   Respuesta esperada: `{"nombre": "Ana", "edad": 20, "activo": True}` (el orden puede variar).

4) **numpy: filtro + transformación**
   Crea `x = np.array([3, 6, 9, 12, 15])`.
   Crea un nuevo array `y` con **solo** los valores de `x` que son múltiplos de 6 y luego súmales 1.

   Respuesta esperada: `y` debe ser `array([7, 13])`.

5) **numpy: diferencia entre `*` y `@`**
   Define:

   - `M = np.array([[1, 2], [3, 4]])`
   - `N = np.array([[2, 0], [1, 2]])`
     Calcula `M * N` y `M @ N`.

   Respuesta esperada:

   - `M * N` debe ser `[[2, 0], [3, 8]]` (elemento a elemento).

   - `M @ N` debe ser `[[4, 4], [10, 8]]` (matricial).

6) **pandas: selección + columna nueva + resumen**
   Con el `df` del capítulo:

   - crea una columna `ventas_por_cliente = ventas / clientes`,
   - filtra las filas donde `clima == "sol"`,

- calcula el promedio de `ventas_por_cliente` en ese subconjunto.

Respuesta esperada: un único número (float).
Con los datos del capítulo debe ser aproximadamente `5.0` (puede verse como `5` o `5.0` según formato).

Glosario (para Colab)

- **objeto**: cualquier "cosa" en Python (número, texto, lista, tabla…).
- **tipo (type)**: la categoría de un objeto (`int`, `float`, `str`, `bool`).
- **índice (index)**: posición de un elemento. En Python empieza en `0`.
- **método**: función "pegada" a un objeto, por ejemplo `lista.sort()`.
- **mutabilidad**: si un objeto puede cambiar por dentro (listas sí; tuplas no).
- **alias**: dos nombres apuntando al mismo objeto.

Capítulo 3 — Datos externos (leer y guardar archivos con pandas)

Abrir este capítulo en Google Colab

Objetivos del capítulo

Al terminar este capítulo vas a poder:

- leer datos desde archivos comunes (`.txt`, `.csv`, `.xlsx`) usando `pandas`,
- entender los parámetros más importantes de lectura: `sep`, `header`, `names`,
- guardar tus datos desde Python (`to_csv`, `to_excel`) de forma reproducible,
- distinguir claramente lo **local** (sin internet) de lo **online** (opcional, depende de conexión).

---

1. La idea general: "archivo → DataFrame"

En la práctica, casi todo flujo de datos es:

1) **leer** un archivo en un `DataFrame`

2) **transformar** (limpiar, filtrar, crear variables)

3) **guardar** el resultado

Glosario

- **DataFrame**: tabla de datos en `pandas` (filas y columnas).
- **CSV**: archivo de texto con valores separados por comas.
- **delimiter / sep**: separador entre columnas (coma, espacio, tab, etc.).
- **header**: indica si el archivo tiene fila de nombres.
- **ruta (path)**: ubicación del archivo (ej.

2. Leer archivos de texto (`.txt`) y CSV (`.csv`)

Vamos a usar ejemplos **locales** (sin internet). Para simular un archivo, usaremos `StringIO` (un "archivo en memoria").

```
import pandas as pd
from io import StringIO
```

> **Sintaxis clave (leer archivos)**
>
> - `pd.read_table(...)` se usa para texto con separador (por ejemplo espacios o tabulaciones).
> - `pd.read_csv(...)` se usa para CSV (separado por comas).
> - En ambos, el separador se controla con `sep=` (o `delimiter=`).

2.1 Texto con separador por espacios

```
texto = '''year product1 product2 product3
2010  3        1        2
2011  4        2        2
2012  6        3        1
'''

df_txt = pd.read_table(StringIO(texto), sep=r"\s+")
df_txt
```

	year	product1	product2	product3
0	2010	3	1	2
1	2011	4	2	2
2	2012	6	3	1

Qué significa `sep=r"\s+"`
- `\s+` es "uno o más espacios" (sirve si hay varios espacios entre columnas).

2.2 CSV simple

```
csv = '''year,product1,product2,product3
2010,3,1,2
2011,4,2,2
2012,6,3,1
'''

df_csv = pd.read_csv(StringIO(csv))
df_csv
```

	year	product1	product2	product3
0	2010	3	1	2
1	2011	4	2	2
2	2012	6	3	1

2.3 Cuando el archivo NO tiene encabezados

A veces un archivo viene sin nombres de columnas. En ese caso:

- `header=None` para decir "no hay encabezado"

- `names=[...]` para asignar nombres

```
csv_sin_header = '''2010,3,1,2
2011,4,2,2
2012,6,3,1
'''

df_csv2 = pd.read_csv(
    StringIO(csv_sin_header),
    header=None,
    names=["year", "product1", "product2", "product3"]
)
df_csv2
```

	year	product1	product2	product3
0	2010	3	1	2
1	2011	4	2	2
2	2012	6	3	1

---

3. Guardar datos (exportar)

Guardar es importante por 2 razones:

- para no perder trabajo,
- para poder compartir resultados.

> **Sintaxis clave (guardar)**
>
> - `df.to_csv("archivo.csv", index=False)` guarda CSV.
> - `df.to_excel("archivo.xlsx", index=False)` guarda Excel.
> - `index=False` evita guardar la columna de índice como una columna extra.

3.1 Guardar CSV (ejemplo)

```
df_csv2.to_csv("salida_ejemplo.csv", index=False)
```

> En el manual (Quarto) no lo ejecutamos para no crear archivos en el render.
> En VSCode o Colab sí puedes ejecutarlo sin problema.

3.2 Guardar Excel (ejemplo)

```
df_csv2.to_excel("salida_ejemplo.xlsx", index=False)
```

---

4. Leer Excel (`.xlsx`) sin depender de internet

Para practicar la sintaxis sin archivos externos, vamos a:

1) crear un `DataFrame`,
2) guardarlo a un Excel en memoria,
3) volverlo a leer con `read_excel`.

```
from io import BytesIO
```

```
df_base = pd.DataFrame({"A": [1, 2, 3], "B": [10, 20, 30]})
buffer = BytesIO()

with pd.ExcelWriter(buffer, engine="openpyxl") as writer:
    df_base.to_excel(writer, sheet_name="Hoja1", index=False)

buffer.seek(0)
```

```
df_excel = pd.read_excel(buffer, sheet_name="Hoja1")
df_excel
```

	A	B
0	1	10
1	2	20
2	3	30

Tip

Por qué hacemos esto
Porque te permite aprender `read_excel()` sin depender de archivos externos ni internet.

---

5. Aplicación con datos reales (opcional, online)

Todo lo de esta sección **depende de internet** y puede fallar en renderización local.
Por eso, en el manual lo dejamos **comentado** (`eval: false`).
En **Google Colab** suele funcionar mejor.

5.1 Bases de ejemplo (wooldridge)

```
# En VSCode: pip install wooldridge
# En Colab:  !pip -q install wooldridge
import wooldridge as woo

wage1 = woo.dataWoo("wage1")
wage1.head()
```

5.2 Datos financieros y macro (FRED / Stooq / Yahoo Finance)

```
# En Colab:
# !pip -q install pandas_datareader yfinance

import pandas_datareader as pdr
import yfinance as yf

wti = pdr.data.DataReader(
    name="DCOILWTICO",
    data_source="fred",
    start="2014-01-01",
    end="2024-12-31"
)

aapl = pdr.data.DataReader(
    name="AAPL",
    data_source="stooq",
    start="2017-01-01",
    end="2024-12-31"
)

btc = yf.download(
    "BTC-USD",
    start="2017-01-01",
    end="2024-12-31",
    auto_adjust=False
)

wti.tail(), aapl.head(), btc.head()
```

5.3 Datos desde un enlace (Google Drive)

```
df_online = pd.read_table(
    "https://drive.google.com/uc?id=TU_ID",
    sep=r"\s+"
)
df_online.head()
```

Ejercicios propuestos

1) **Lectura sin encabezado**
Crea un CSV sin encabezado con 4 columnas (puedes copiar el ejemplo del capítulo). Léelo con `read_csv` y asigna los nombres `["a","b","c","d"]`.

   Respuesta esperada: un `DataFrame` con columnas exactamente `a, b, c, d` y 3 filas.

2) **Separadores "raros"**
Crea un texto donde las columnas estén separadas por varios espacios (no uno). Léelo con `read_table` usando `sep=r"\s+"`.

   Respuesta esperada: un `DataFrame` con las columnas correctas (sin que se junte todo en una sola columna).

3) **Exportación consistente**
Toma un `DataFrame` cualquiera del capítulo y guarda:

   - un CSV sin índice,
   - un Excel sin índice.

   Respuesta esperada: dos archivos creados (`.csv` y `.xlsx`) sin una columna extra de índice.

4) **Excel con hojas**
Crea un Excel en memoria con dos hojas (`Hoja1`, `Hoja2`), cada una con un DataFrame distinto. Luego lee solo `Hoja2`.

Respuesta esperada: `read_excel(..., sheet_name="Hoja2")` devuelve exactamente la tabla de `Hoja2`.

5) **Mini-proyecto**
Crea un `DataFrame` con columnas `ventas` y `clientes`. Calcula `ventas_por_cliente`, luego guarda el resultado en CSV.

Respuesta esperada: un `DataFrame` con 3 columnas (`ventas`, `clientes`, `ventas_por_cliente`) y un CSV con esas mismas columnas.

Glosario (para Colab)

- **DataFrame**: tabla de datos en `pandas`.
- **CSV**: archivo de texto con valores separados por comas.
- **sep / delimiter**: separador entre columnas.
- **header**: indica si el archivo trae nombres de columnas.
- **ruta**: ubicación del archivo en tu computador.
- **relativa**: ruta desde la carpeta del proyecto.
- **absoluta**: ruta completa (más frágil).

Capítulo 4 — Gráficos básicos con matplotlib

Abrir este capítulo en Google Colab

Objetivos del capítulo

Al terminar este capítulo vas a poder:

- crear gráficos básicos (`plot`, `scatter`, `hist`) con **matplotlib**,
- entender la sintaxis mínima de un gráfico: ejes, títulos, etiquetas y leyenda,
- sobreponer varios gráficos en una misma figura,
- guardar figuras a un archivo (opcional).

---

1. ¿Qué es matplotlib?

`matplotlib` es una librería para hacer gráficos en Python. La forma más común de usarla es:

```
import matplotlib.pyplot as plt
```

Sintaxis clave (import)

- `import matplotlib.pyplot as plt` crea el alias `plt`.
- Luego casi todo se hace con `plt.algo(...)` (por ejemplo `plt.plot(...)`).

Glosario

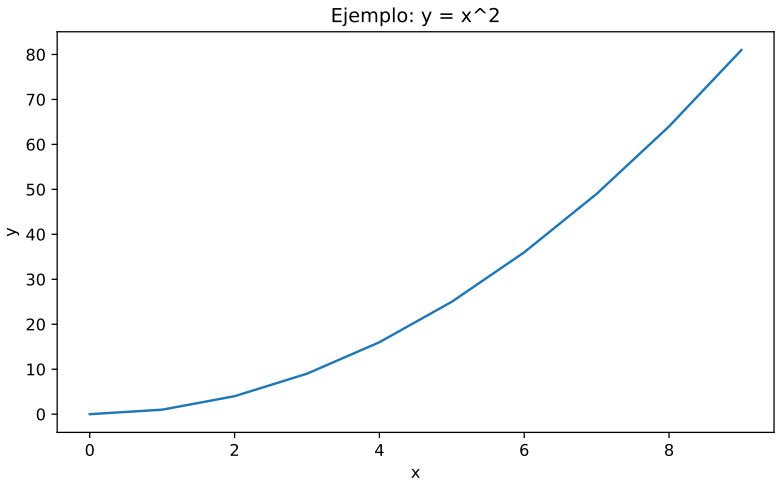
- **figura (figure)**: el "lienzo" completo del gráfico.
- **ejes (axes)**: el área donde se dibuja (x e y).
- **plot**: línea.
- **scatter**: nube de puntos.
- **histograma (hist)**: barras para mostrar distribución.
- **leyenda (legend)**: cuadro con el nombre de cada serie.

2. Nuestro primer gráfico (línea)

Vamos a crear datos simples y graficarlos.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.arange(0, 10)       # 0,1,2,...,9
y = x**2                   # y = x^2

plt.plot(x, y)
plt.title("Ejemplo: y = x^2")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Sintaxis clave (líneas)

- `plt.plot(x, y)` dibuja una línea.
- `plt.title(...)` pone un título.
- `plt.xlabel(...)` y `plt.ylabel(...)` etiquetan los ejes.

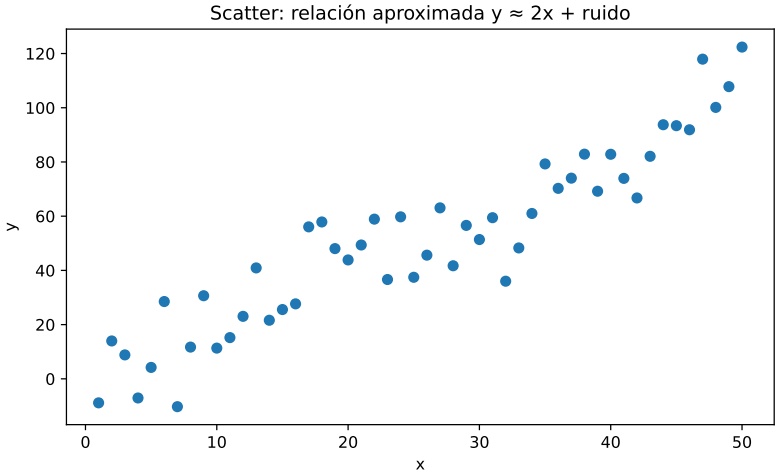
- `plt.show()` muestra la figura (es buena práctica en notebooks).

---

3. Gráfico de dispersión (scatter)

Un `scatter` muestra puntos individuales. Es típico cuando quieres ver relación entre dos variables.

```
np.random.seed(123)

x = np.arange(1, 51)
y = 2*x + np.random.normal(0, 10, size=len(x))  # relación con ruido

plt.scatter(x, y)
plt.title("Scatter: relación aproximada y   2x + ruido")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Sintaxis clave (scatter)

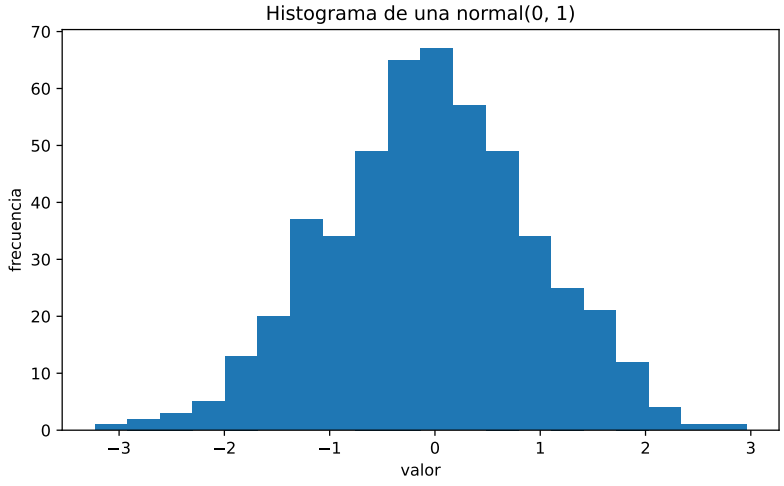
- `plt.scatter(x, y)` dibuja puntos (no línea).
- Es útil para ver patrones, tendencias y valores atípicos.

---

4. Histograma (hist)

Un histograma sirve para ver "cómo se distribuyen" los valores.

```
datos = np.random.normal(loc=0, scale=1, size=500)

plt.hist(datos, bins=20)
plt.title("Histograma de una normal(0, 1)")
plt.xlabel("valor")
plt.ylabel("frecuencia")
plt.show()
```



Sintaxis clave (hist)

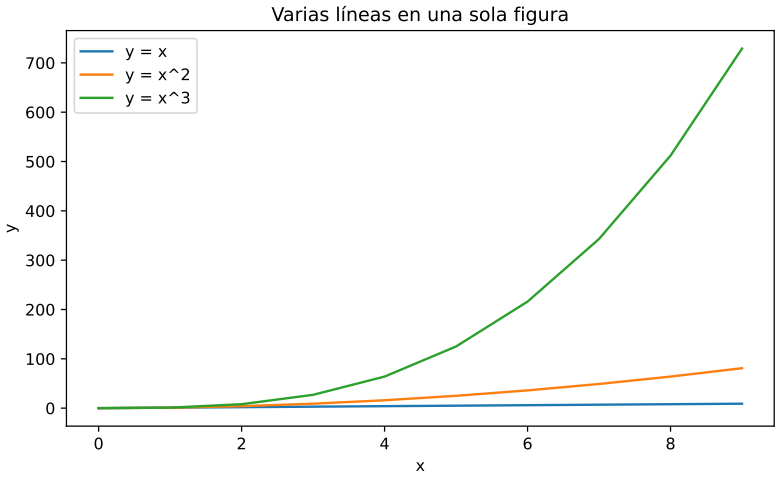
- `plt.hist(datos, bins=20)` crea un histograma con 20 barras (bins).
- Más `bins` = más detalle (pero puede verse ruidoso).

---

5. Sobreponer varios gráficos (varias líneas)

Cuando quieres comparar series, puedes dibujar varias líneas en la misma figura.

```
x = np.arange(0, 10)
y1 = x
y2 = x**2
y3 = x**3

plt.plot(x, y1, label="y = x")
plt.plot(x, y2, label="y = x^2")
plt.plot(x, y3, label="y = x^3")

plt.title("Varias líneas en una sola figura")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```



Sintaxis clave (leyenda)

- `label="..."` nombra cada serie.
- `plt.legend()` muestra la leyenda.

---

6. Guardar un gráfico (opcional)

Guardar gráficos es útil para informes. En Quarto lo dejamos desactivado para no crear archivos en el render.

```
plt.plot([1, 2, 3], [1, 4, 9])
plt.title("Figura para guardar")
plt.savefig("mi_figura.png", dpi=150, bbox_inches="tight")
```

Tip

Recomendación
En proyectos reales, guarda figuras en una carpeta como `figuras/` para tener todo ordenado.

Ejercicios propuestos

1) **Línea con transformación**
Crea `x = np.arange(0, 11)` y `y = 3*x + 2`. Grafica `y` contra `x` con título y etiquetas.

Respuesta esperada: una línea recta creciente que cruza `y=2` cuando `x=0` y llega a `y=32` cuando `x=10`.

2) **Scatter con dos grupos**
Genera dos nubes de puntos:

- Grupo A: `x` entre 1 y 30, `y = x + ruido`
- Grupo B: `x` entre 1 y 30, `y = 2x + ruido`
  Dibuja ambos en el mismo scatter (dos llamadas a `plt.scatter`) con leyenda.

Respuesta esperada: dos "bandas" de puntos: una alrededor de la recta `y=x` y otra alrededor de `y=2x`.

3) **Histograma comparativo**
Genera 1000 datos normales y 1000 datos uniformes. Haz dos histogramas (uno a la vez) usando el mismo número de bins.

Respuesta esperada:

- el histograma normal se concentra en el centro (forma de "campana" aproximada),
- el uniforme se ve "plano" (barras relativamente parejas).

4) **Tres líneas con leyenda**
Grafica en una misma figura `sin(x)`, `cos(x)` y `sin(x)+cos(x)` para `x` entre 0 y `2 `. Usa leyenda.

Respuesta esperada: tres curvas suaves y periódicas con una leyenda que las identifique.

5) **Guardar figura**
Crea un gráfico y guárdalo en `figuras/ejemplo.png`.

Respuesta esperada: el archivo `figuras/ejemplo.png` existe y al abrirlo se ve el gráfico guardado.

Glosario (para Colab)

- **figura (figure)**: el “lienzo” completo del gráfico.
- **ejes (axes)**: el área donde se dibuja (x e y).
- **plot**: gráfico de líneas.
- **scatter**: gráfico de puntos.
- **hist**: histograma.
- **leyenda**: nombres de series en el gráfico.

Capítulo 5 — Estadísticas descriptivas (tablas, gráficos y resumen)

Abrir este capítulo en Google Colab

Objetivos del capítulo

Al terminar este capítulo vas a poder:

- construir **tablas de frecuencia** y **porcentajes** con `value_counts`,
- crear **tablas de contingencia** con `pd.crosstab` y normalizarlas,
- graficar categorías con **pastel**, **barras**, **apiladas** y **agrupadas**,
- visualizar variables numéricas con **histograma** (frecuencia o densidad) y **boxplot**,
- construir una **ECDF** (distribución acumulada empírica) con `numpy`,
- calcular estadísticos y matrices con `describe`, `var`, `std`, `cov`, `corr`.

---

1. Qué haremos (enfocado en Python)

Este capítulo no es para "explicar estadística".
Aquí nos concentramos en **cómo escribir el código** para resumir y visualizar datos.

Glosario

- **frecuencia**: cuántas veces aparece un valor.
- **proporción**: frecuencia / total.
- **crosstab**: tabla cruzada (contingencia).
- **densidad**: histograma escalado para que el área total sea 1 (`density=True`).
- **KDE**: estimación de densidad (Kernel Density Estimation).

2. Datos de ejemplo (local)

Usamos datos pequeños **locales** para practicar la sintaxis (sin depender de internet).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

np.random.seed(123)

df = pd.DataFrame({
    "marriage": np.random.choice(["mala", "regular", "buena"], size=80, p=[0.2, 0.35, 0.45]),
    "haskids":  np.random.choice(["sin hijos", "con hijos"], size=80, p=[0.55, 0.45]),
    "roe":      np.clip(np.random.normal(loc=12, scale=6, size=80), -10, 40),
    "salary":   np.clip(np.random.normal(loc=120, scale=35, size=80), 20, 300)
})
df.head()
```

	marriage	haskids	roe	salary
0	buena	sin hijos	21.113194	93.354701
1	regular	con hijos	9.875999	120.127580
2	regular	con hijos	7.059412	76.051596
3	buena	sin hijos	12.781290	100.682209
4	buena	con hijos	19.603792	111.417883

Sintaxis clave (crear datos)

- `pd.DataFrame({...})` crea una tabla (DataFrame).
- `np.random.choice([...], size=n)` genera categorías aleatorias.
- `np.random.normal(loc, scale, size)` genera números.
- `np.clip(x, a, b)` fuerza a que `x` quede entre `a` y `b`.

3. Categorías: frecuencias y gráficos

3.1 Frecuencias y porcentajes (`value_counts`)

```
freq = df["marriage"].value_counts()
freq
```

```
marriage
buena      36
regular    32
mala       12
Name: count, dtype: int64
```

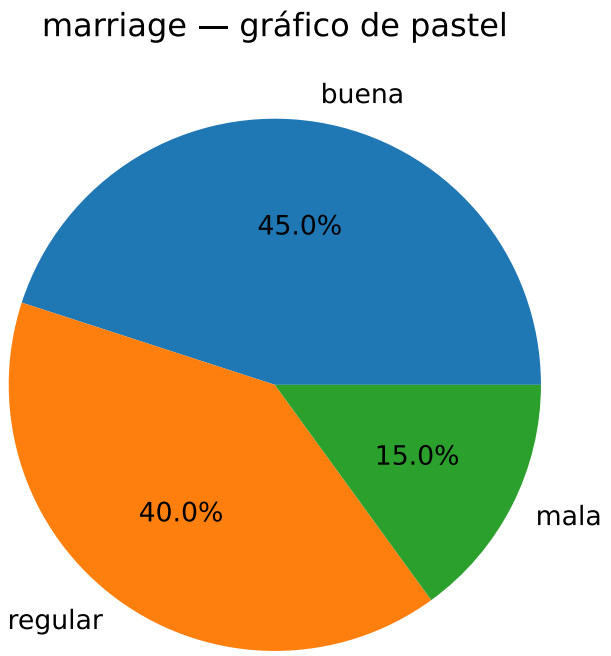
```
df["marriage"].value_counts(normalize=True) * 100
```

```
marriage
buena      45.0
regular    40.0
mala       15.0
Name: proportion, dtype: float64
```

> **Sintaxis clave (`value_counts`)**
>
> - `serie.value_counts()` → conteos por categoría.
> - `normalize=True` → proporciones (suman 1). Multiplica por 100 si quieres porcentaje.

3.2 Pastel (pie chart)

```
freq.plot(kind="pie", autopct="%1.1f%%")
plt.ylabel("")  # quita etiqueta del eje y
plt.title("marriage - gráfico de pastel")
plt.show()
```

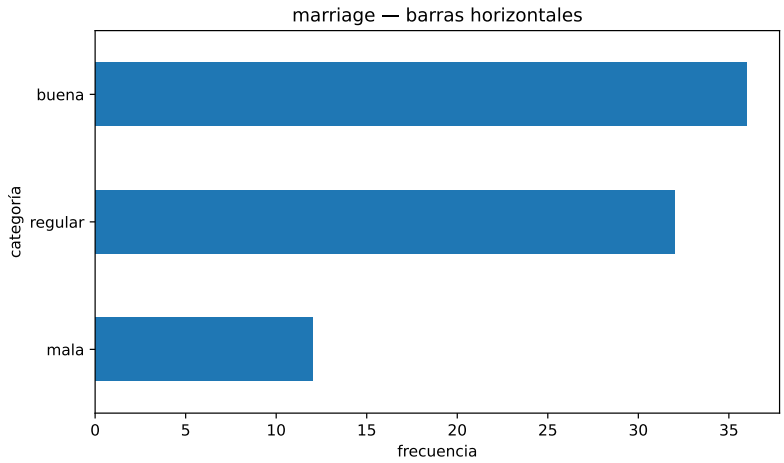


Sintaxis clave (pie)

- `freq.plot(kind="pie")` grafica desde una Serie.
- `autopct="..."` muestra porcentajes dentro del pastel.
- `plt.show()` "cierra" y muestra la figura (buena práctica).

3.3 Barras horizontales

```
freq.sort_values().plot(kind="barh")
plt.title("marriage - barras horizontales")
plt.xlabel("frecuencia")
plt.ylabel("categoría")
plt.show()
```



---

4. Tablas por subgrupos y contingencia (`crosstab`)

4.1 Tabla cruzada (conteos)

```
tabla = pd.crosstab(df["marriage"], df["haskids"])
tabla
```

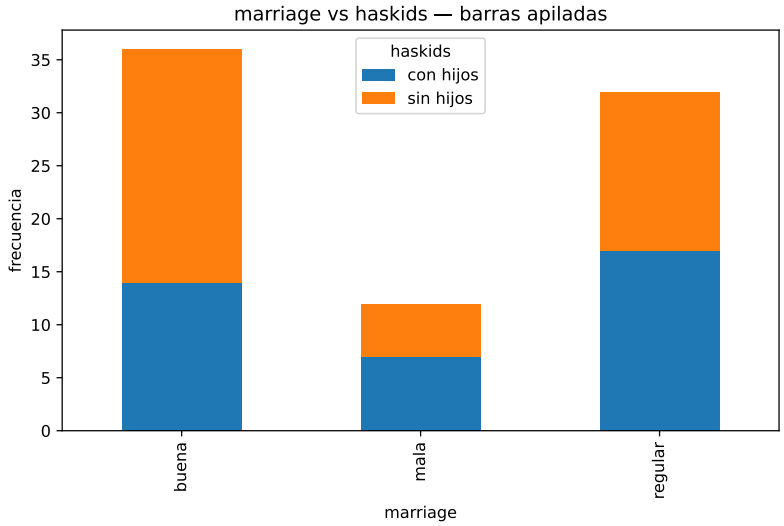
haskids marriage	con hijos	sin hijos
buena	14	22
mala	7	5
regular	17	15

Sintaxis clave (`crosstab`)

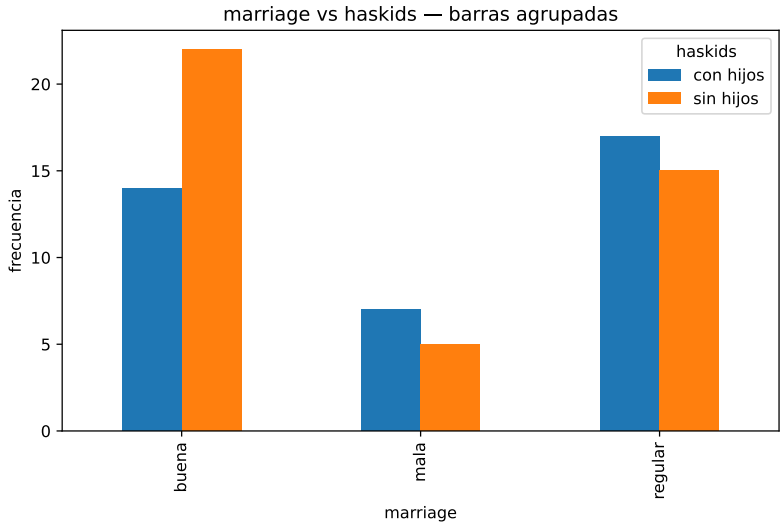
- `pd.crosstab(filas, columnas)` → tabla de conteos.
- Funciona perfecto para variables categóricas.

4.2 Barras apiladas y agrupadas

```
tabla.plot(kind="bar", stacked=True)
plt.title("marriage vs haskids - barras apiladas")
plt.xlabel("marriage")
plt.ylabel("frecuencia")
plt.show()
```



```
tabla.plot(kind="bar", stacked=False)
plt.title("marriage vs haskids - barras agrupadas")
plt.xlabel("marriage")
plt.ylabel("frecuencia")
plt.show()
```



4.3 Tablas normalizadas (proporciones)

```
pd.crosstab(df["marriage"], df["haskids"], normalize=True)
```

haskids marriage	con hijos	sin hijos
buena	0.1750	0.2750
mala	0.0875	0.0625
regular	0.2125	0.1875

```
pd.crosstab(df["marriage"], df["haskids"], normalize="index")
```

haskids marriage	con hijos	sin hijos
buena	0.388889	0.611111
mala	0.583333	0.416667
regular	0.531250	0.468750

```
pd.crosstab(df["marriage"], df["haskids"], normalize="columns")
```

haskids marriage	con hijos	sin hijos
buena	0.368421	0.523810
mala	0.184211	0.119048
regular	0.447368	0.357143

i Sintaxis clave (normalize)

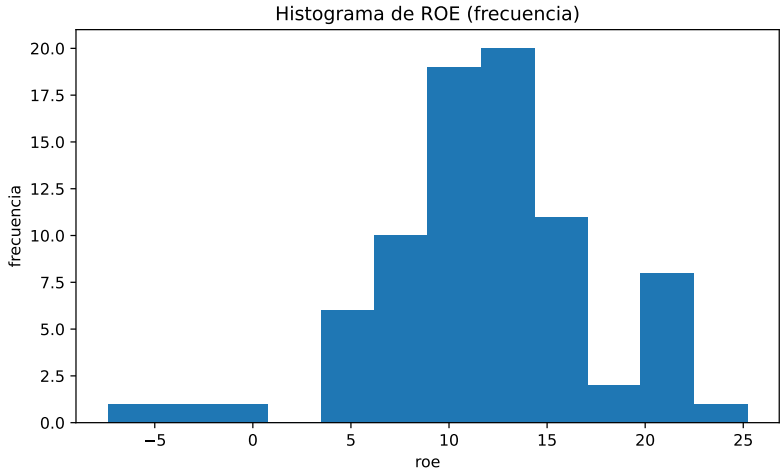
- `normalize=True` → proporción del total (toda la tabla suma 1).
- `normalize="index"` → proporciones por fila (cada fila suma 1).
- `normalize="columns"` → proporciones por columna (cada columna suma 1).

---

5. Numéricas: histogramas, densidad y ECDF

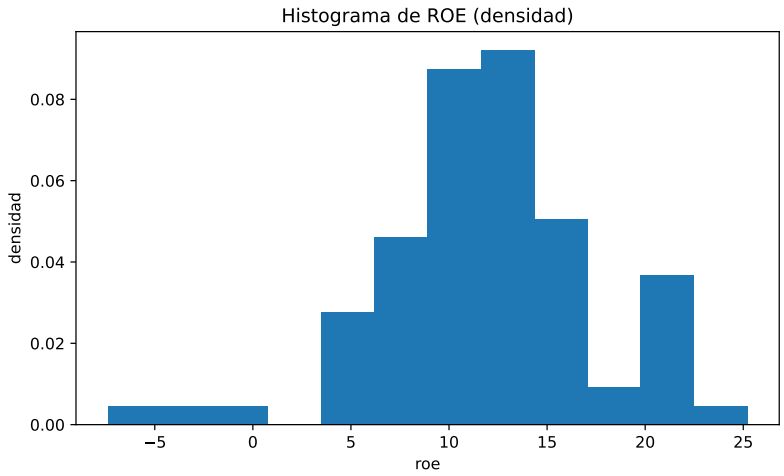
5.1 Histograma (frecuencia)

```
plt.hist(df["roe"], bins=12)
plt.title("Histograma de ROE (frecuencia)")
plt.xlabel("roe")
plt.ylabel("frecuencia")
plt.show()
```



5.2 Histograma (densidad)

```
plt.hist(df["roe"], bins=12, density=True)
plt.title("Histograma de ROE (densidad)")
plt.xlabel("roe")
plt.ylabel("densidad")
plt.show()
```



Sintaxis clave (`density=True`)

- `density=True` cambia la escala: el área total del histograma 1.
- Útil para comparar distribuciones entre grupos o muestras distintas.

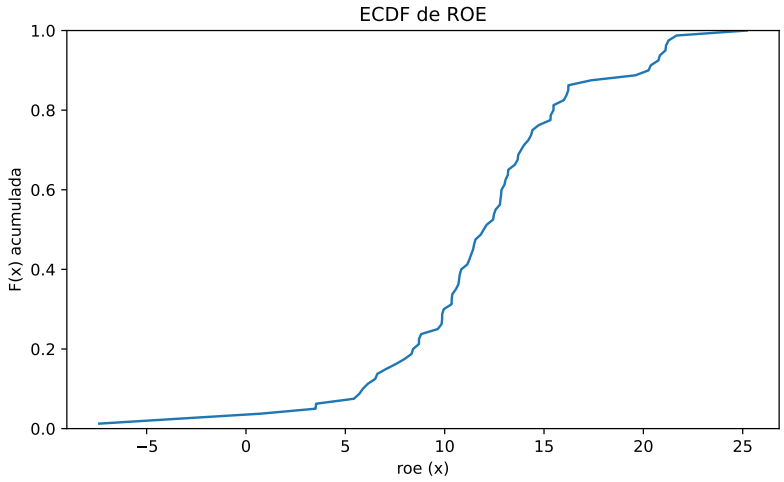
5.3 Densidad KDE (opcional)

La KDE puede depender de paquetes extra. En el manual (Quarto) la dejamos desactivada.
En Colab puedes activarla si tienes lo necesario.

```
df["roe"].plot(kind="density")
plt.title("KDE de ROE (opcional)")
plt.xlabel("roe")
plt.show()
```

5.4 ECDF (acumulada empírica)

```
x = np.sort(df["roe"].to_numpy())
y = np.arange(1, len(x) + 1) / len(x)

plt.plot(x, y)
plt.title("ECDF de ROE")
plt.xlabel("roe (x)")
plt.ylabel("F(x) acumulada")
plt.ylim(0, 1)
plt.show()
```



Sintaxis clave (ECDF)

- `to_numpy()` convierte la columna a arreglo.
- `np.sort(...)` ordena de menor a mayor.
- `np.arange(1, n+1)/n` crea el acumulado: 1/n, 2/n, …, 1.

6. Resumen numérico: estadísticos y matrices

6.1 Resumen rápido con `describe`

```
df[["roe", "salary"]].describe()
```

	roe	salary
count	80.000000	80.000000
mean	12.003661	113.816654
std	5.418543	34.735772
min	-7.386330	44.662729
25%	9.797359	88.437450
50%	12.042026	113.114536
75%	14.498293	136.610401
max	25.204213	223.551891

6.2 Media, mediana, varianza, desviación estándar

```
df["roe"].mean(), df["roe"].median(), df["roe"].var(), df["roe"].std()
```

```
(np.float64(12.003661003911507),
 12.042026241962258,
 29.360611235450484,
 5.418543276144473)
```

6.3 Covarianza y correlación

```
df[["salary", "roe"]].cov()
```

	salary	roe
salary	1206.573859	8.807864
roe	8.807864	29.360611

```
df[["salary", "roe"]].corr()
```

	salary	roe
salary	1.000000	0.046796
roe	0.046796	1.000000

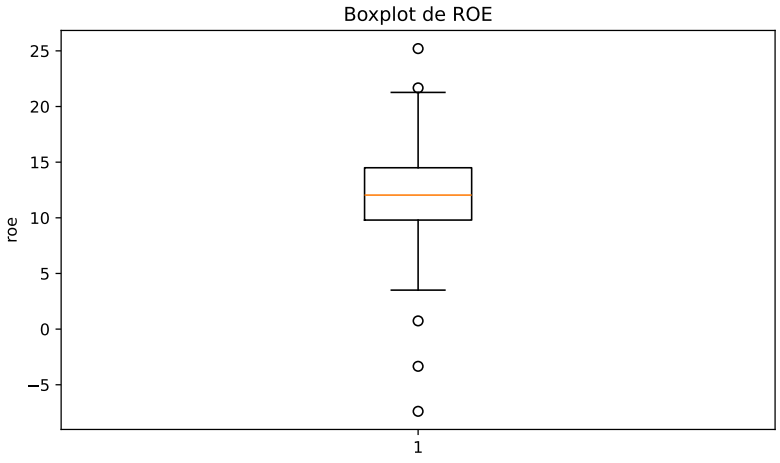
i Sintaxis clave (cov/corr)

- `df[cols]` selecciona columnas.
- `.cov()` y `.corr()` devuelven matrices (2×2 si son 2 variables).

---

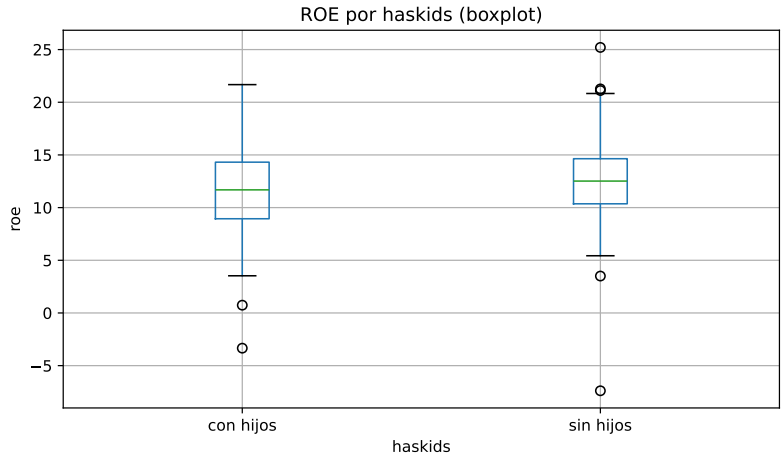
7. Boxplots

```
plt.boxplot(df["roe"].dropna())
plt.title("Boxplot de ROE")
plt.ylabel("roe")
plt.show()
```



Por grupo:

```
df.boxplot(column="roe", by="haskids")
plt.title("ROE por haskids (boxplot)")
plt.suptitle("")  # quita el título automático
plt.xlabel("haskids")
plt.ylabel("roe")
plt.show()
```



---

8. Aplicación con datos reales (opcional, online)

Si quieres practicar con **bases reales**, puedes usar conjuntos de datos educativos (por ejemplo, paquetes como `wooldridge`). Esto requiere instalación y/o internet, por eso en Quarto no se ejecuta.

```
# En VSCode: pip install wooldridge
# En Colab:  !pip -q install wooldridge
import wooldridge as woo
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
affairs = woo.dataWoo("affairs")

# Frecuencias y tablas
affairs["ratemarr"].value_counts()
pd.crosstab(affairs["ratemarr"], affairs["kids"], normalize="index")

ceosal1 = woo.dataWoo("ceosal1")

# Histograma (frecuencia y densidad)
plt.hist(ceosal1["roe"], bins=20)
plt.title("Histograma de ROE (ceosal1)")
plt.show()

plt.hist(ceosal1["roe"], bins=20, density=True)
plt.title("Histograma de ROE (densidad, ceosal1)")
plt.show()

# ECDF
x = np.sort(ceosal1["roe"].to_numpy())
y = np.arange(1, len(x) + 1) / len(x)
plt.plot(x, y)
plt.title("ECDF de ROE (ceosal1)")
plt.ylim(0, 1)
plt.show()

# Estadísticos y matrices
ceosal1[["salary", "roe"]].describe()
ceosal1[["salary", "roe"]].cov()
ceosal1[["salary", "roe"]].corr()

# Boxplot
ceosal1.boxplot(column="roe")
plt.title("Boxplot de ROE (ceosal1)")
plt.show()
```

Ejercicios propuestos

1) **Frecuencias + barras**
Calcula `value_counts()` de `marriage` y grafica barras horizontales.

Respuesta esperada: 3 barras con los conteos por categoría.

2) **Crosstab normalizado por filas**
Crea `pd.crosstab(marriage, haskids, normalize="index")`.

Respuesta esperada: cada fila suma 1.

3) **Frecuencia vs densidad**
Grafica dos histogramas de `roe`: uno con `density=False` y otro con `density=True`.

Respuesta esperada: cambia la escala del eje y (frecuencia vs densidad).

4) **ECDF y percentil**
Construye la ECDF de `roe` y estima el valor del percentil 90.

Respuesta esperada: un número cercano a `df["roe"].quantile(0.9)`.

5) **Cov y corr**
Calcula `cov()` y `corr()` para `salary` y `roe`.

Respuesta esperada: dos matrices 2×2.

6) **Boxplot por grupo**
Crea un boxplot de `roe` por `haskids`.

Respuesta esperada: 2 cajas (una por cada categoría de `haskids`).

Glosario (para Colab)

- **frecuencia**: conteos.
- **proporción**: conteos / total.
- **crosstab**: tabla cruzada.
- **densidad**: histograma escalado (`density=True`).
- **KDE**: densidad estimada (kernel).
- **boxplot**: gráfico de caja.
- **cov / corr**: covarianza / correlación.

Capítulo 6 — Distribuciones de probabilidad (scipy.stats)

Abrir este capítulo en Google Colab

Objetivos del capítulo

Vas a aprender a usar `scipy.stats` para:

- calcular **PMF/PDF**, **CDF** y **cuantiles** (`ppf`),
- graficar PMF/PDF y CDF de forma simple con `matplotlib`,
- simular variables aleatorias con `rvs` y comparar **muestra vs teoría**,
- entender (por código) qué significa "promedio acumulado" y por qué converge a la esperanza.

---

1. `scipy.stats`: el patrón de uso

En `scipy.stats`, muchas distribuciones siguen este patrón:

- Discretas: `pmf(x, ...)` y `cdf(x, ...)`
- Continuas: `pdf(x, ...)` y `cdf(x, ...)`
- Cuantiles: `ppf(q, ...)` (por ejemplo, `q=0.95`)
- Simulación: `rvs(..., size=n, random_state=...)`

Glosario

- **PMF**: *probability mass function* (masa) → discretas (`pmf`).
- **PDF**: *probability density function* (densidad) → continuas (`pdf`).
- **CDF**: *cumulative distribution function* (acumulada) → `cdf`.
- **PPF**: *percent point function* (cuantil) → inversa de la CDF (`ppf`).
- **rvs**: *random variates* → simular una muestra (`rvs`).
- **loc / scale**: parámetros estándar de `scipy` (inicio y escala).

Sintaxis (plantilla)

- Discreta: `stats.binom.pmf(x, n, p)` / `stats.binom.cdf(x, n, p)`
- Continua: `stats.norm.pdf(x, mu, sigma)` / `stats.norm.cdf(x, mu, sigma)`
- Cuantil: `stats.norm.ppf(0.975, mu, sigma)`
- Simular: `stats.norm.rvs(mu, sigma, size=1000, random_state=123)`

---

2. Distribuciones discretas

2.1 Uniforme discreta (`randint`)

```
# Parámetros
a = 1  # primer valor
b = 6  # último valor

# Valores de X (para tabla) y valores para gráfica CDF
x = np.arange(a, b + 1)
xg = np.arange(a - 1, b + 2)

# PMF y CDF
fx = stats.randint.pmf(x, a, b + 1)
Fx = stats.randint.cdf(x, a, b + 1)
Fxg = stats.randint.cdf(xg, a, b + 1)

tabla_dist = pd.DataFrame({"x": x, "PMF": fx, "CDF": Fx})
tabla_dist
```

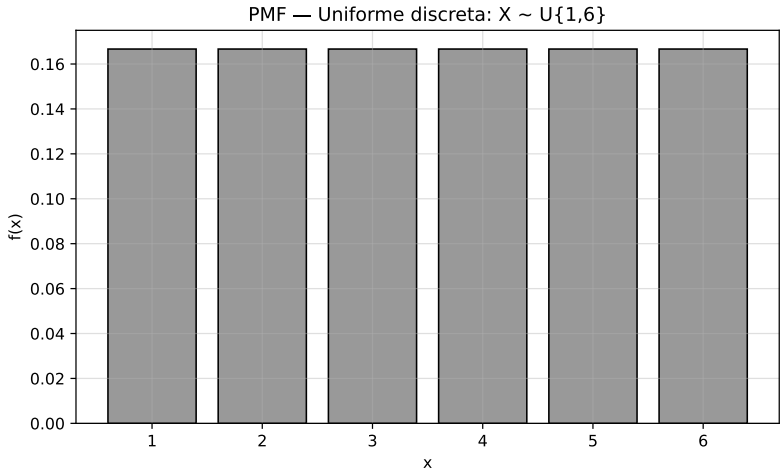
	x	PMF	CDF
0	1	0.166667	0.166667
1	2	0.166667	0.333333
2	3	0.166667	0.500000

	x	PMF	CDF
3	4	0.166667	0.666667
4	5	0.166667	0.833333
5	6	0.166667	1.000000

i Sintaxis clave (`randint`)

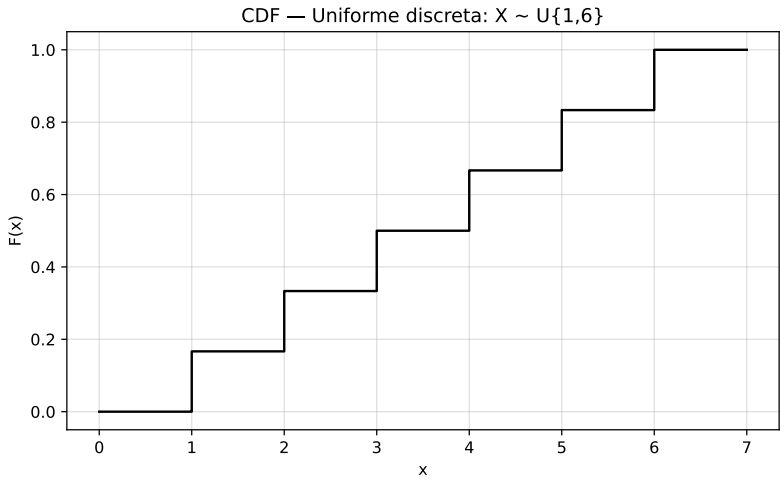
- En `scipy`, `randint(a, b+1)` representa enteros en **[a, b]**.
- Por eso verás `b+1` en la función.

```
plt.bar(x, fx, color="0.6", edgecolor="black")
plt.title(f"PMF - Uniforme discreta: X ~ U{{{a},{b}}}")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.step(xg, Fxg, where="post", color="black")
plt.title(f"CDF - Uniforme discreta: X ~ U{{{a},{b}}}")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
```

```
plt.show()
plt.close()
```



2.2 Bernoulli (`bernoulli`)

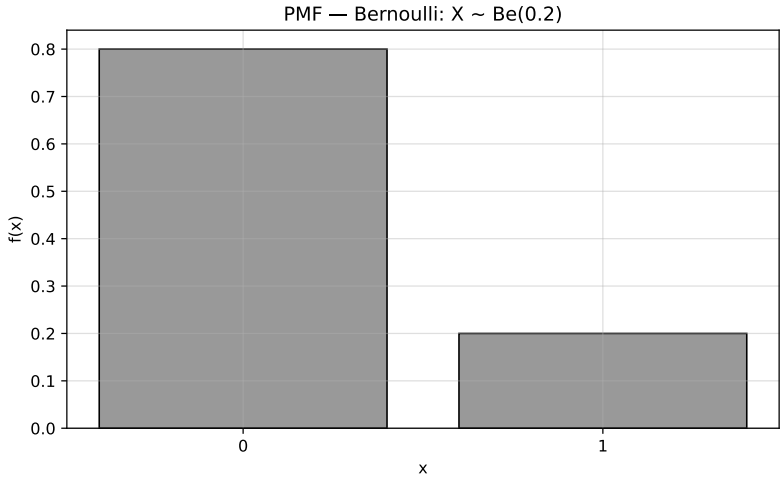
```
x = [0, 1]
xg = np.linspace(-0.1, 1.1, num=1000)

p = 0.2

fx = stats.bernoulli.pmf(x, p)
Fx = stats.bernoulli.cdf(x, p)
Fxg = stats.bernoulli.cdf(xg, p)

tabla_dist = pd.DataFrame({"x": x, "PMF": fx, "CDF": Fx})
tabla_dist
```

	x	PMF	CDF
0	0	0.8	0.8
1	1	0.2	1.0

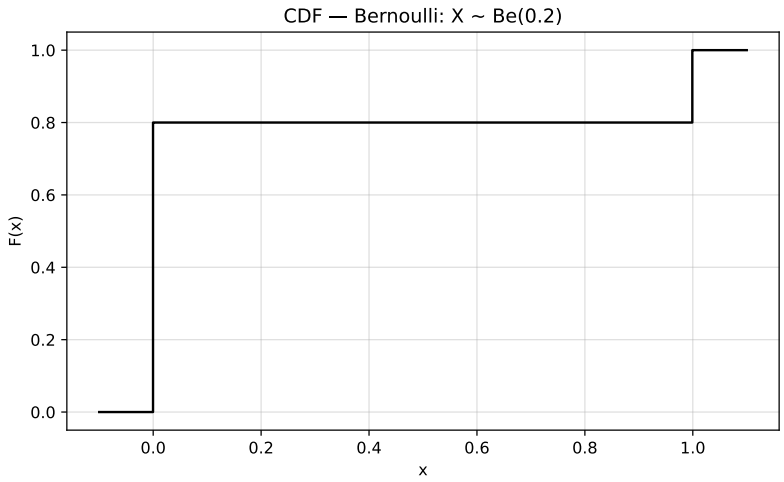
x	PMF	CDF

> **Sintaxis clave (Bernoulli)**
>
> - `x` solo puede ser 0 o 1.
> - `p` es la probabilidad de éxito (`X=1`).

```
plt.bar(x, fx, color="0.6", edgecolor="black")
plt.xticks(x, ["0", "1"])
plt.title(f"PMF - Bernoulli: X ~ Be({p})")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.step(xg, Fxg, color="black")
plt.title(f"CDF - Bernoulli: X ~ Be({p})")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



2.3 Binomial (`binom`)

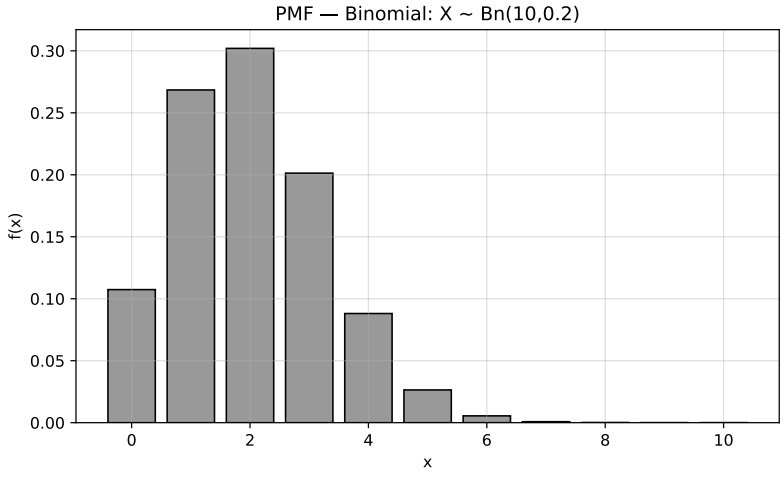
Ejemplo típico: n ensayos Bernoulli independientes con probabilidad de éxito p.

```
x = np.linspace(0, 10, num=11)
xg = np.linspace(-0.5, 10.1, num=1000)

p = 0.2
n = 10

fx = stats.binom.pmf(x, n, p)
Fx = stats.binom.cdf(x, n, p)
Fxg = stats.binom.cdf(xg, n, p)

tabla_dist = pd.DataFrame({"x": x, "PMF": fx, "CDF": Fx})
tabla_dist.head()
```

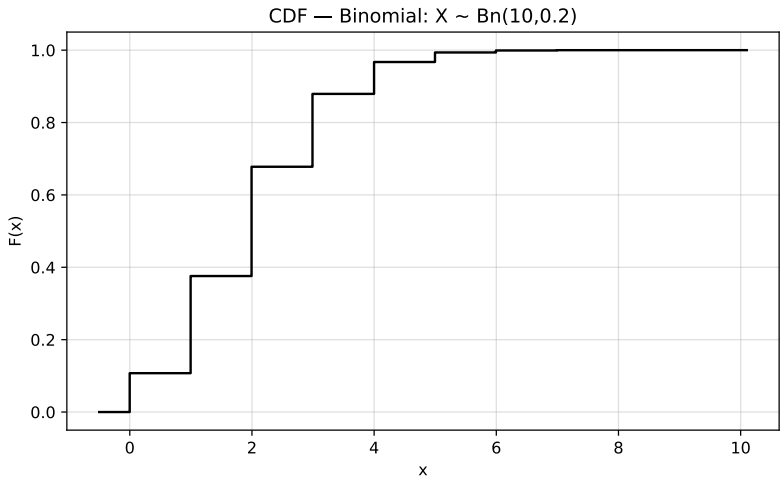
	x	PMF	CDF
0	0.0	0.107374	0.107374
1	1.0	0.268435	0.375810

	x	PMF	CDF
2	2.0	0.301990	0.677800
3	3.0	0.201327	0.879126
4	4.0	0.088080	0.967207

```
plt.bar(x, fx, color="0.6", edgecolor="black")
plt.title(f"PMF - Binomial: X ~ Bn({n},{p})")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.step(xg, Fxg, color="black")
plt.title(f"CDF - Binomial: X ~ Bn({n},{p})")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



Probabilidad puntual (ejemplo): `P(X=2)` con `n=10`, `p=0.2`.

```
stats.binom.pmf(2, 10, 0.2)
```

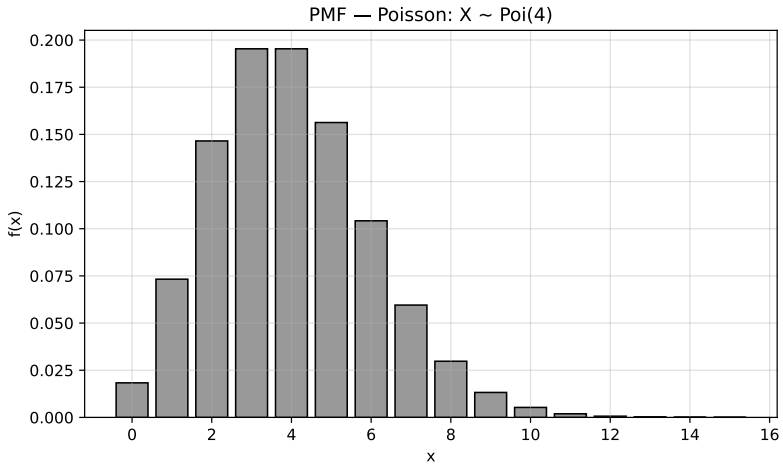
```
np.float64(0.30198988800000004)
```

2.4 Poisson (`poisson`)

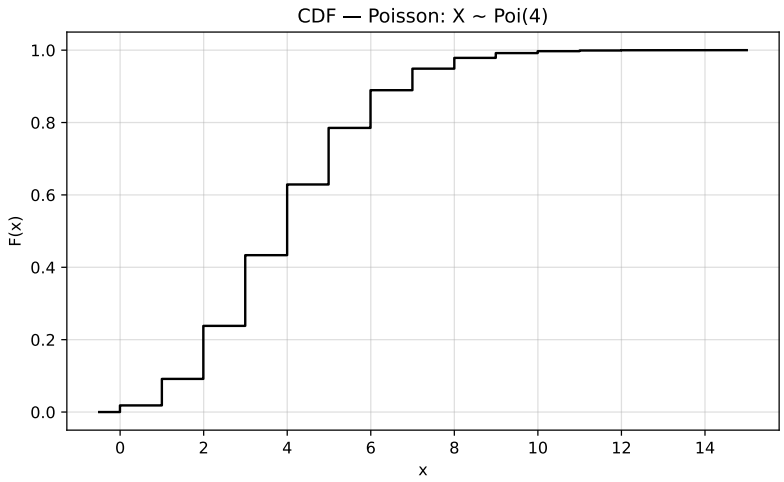
```
x = np.linspace(0, 15, num=16)
xg = np.linspace(-0.5, 15, num=1500)

plambda = 4

fx = stats.poisson.pmf(x, plambda)
Fx = stats.poisson.cdf(x, plambda)
Fxg = stats.poisson.cdf(xg, plambda)

tabla_dist = pd.DataFrame({"x": x, "PMF": fx, "CDF": Fx})
tabla_dist.head()
```

	x	PMF	CDF
0	0.0	0.018316	0.018316
1	1.0	0.073263	0.091578
2	2.0	0.146525	0.238103
3	3.0	0.195367	0.433470
4	4.0	0.195367	0.628837

```
plt.bar(x, fx, color="0.6", edgecolor="black")
plt.title(f"PMF - Poisson: X ~ Poi({plambda})")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.step(xg, Fxg, color="black")
plt.title(f"CDF - Poisson: X ~ Poi({plambda})")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



2.5 Geométrica (`geom`)

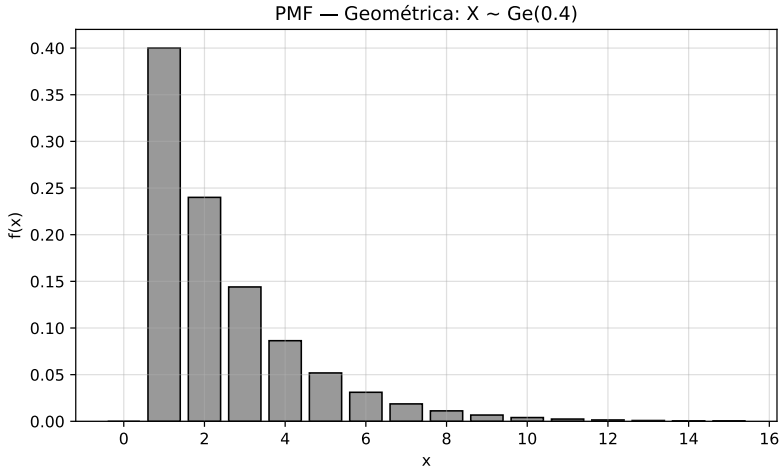
En `scipy.stats.geom`, el soporte empieza en 1, pero es común graficar desde 0 para ver que `P(X=0)=0`.

```
x = np.linspace(0, 15, num=16)
xg = np.linspace(-0.5, 15, num=1500)

p = 0.4

fx = stats.geom.pmf(x, p)
Fx = stats.geom.cdf(x, p)
Fxg = stats.geom.cdf(xg, p)

tabla_dist = pd.DataFrame({"x": x, "PMF": fx, "CDF": Fx})
tabla_dist.head()
```

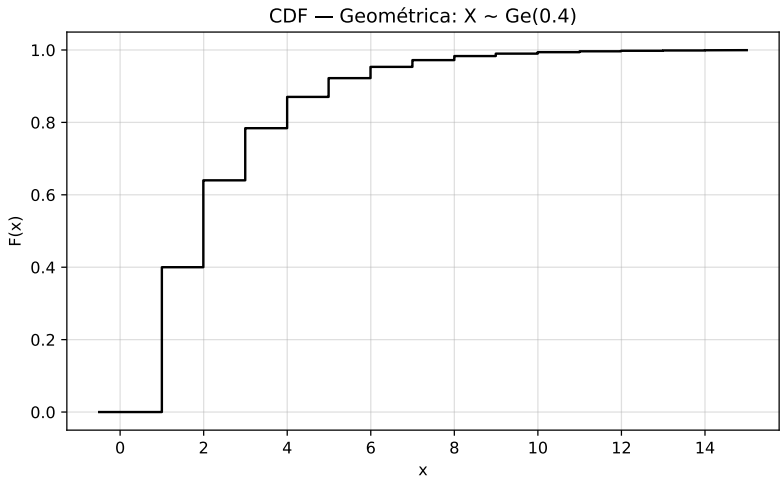
	x	PMF	CDF
0	0.0	0.0000	0.0000
1	1.0	0.4000	0.4000
2	2.0	0.2400	0.6400

	x	PMF	CDF
3	3.0	0.1440	0.7840
4	4.0	0.0864	0.8704

```
plt.bar(x, fx, color="0.6", edgecolor="black")
plt.title(f"PMF - Geométrica: X ~ Ge({p})")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.step(xg, Fxg, color="black")
plt.title(f"CDF - Geométrica: X ~ Ge({p})")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



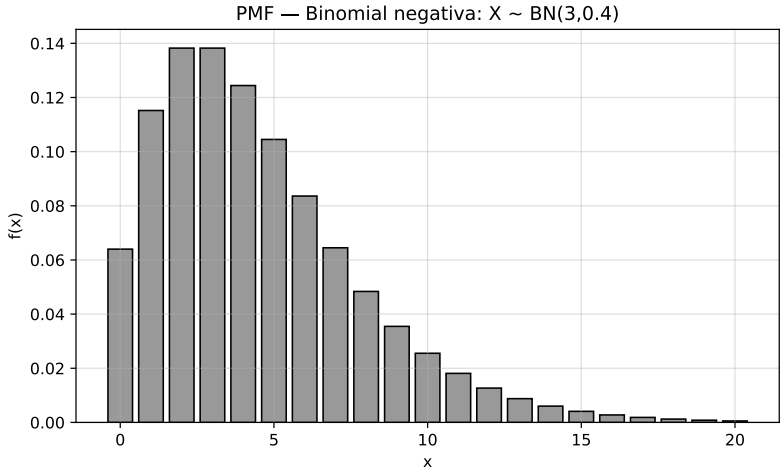
2.6 Binomial negativa (`nbinom`)

```
x = np.linspace(0, 20, num=21)
xg = np.linspace(-0.5, 20, num=2000)

p = 0.4
r = 3

fx = stats.nbinom.pmf(x, r, p)
Fx = stats.nbinom.cdf(x, r, p)
Fxg = stats.nbinom.cdf(xg, r, p)

tabla_dist = pd.DataFrame({"x": x, "PMF": fx, "CDF": Fx})
tabla_dist.head()
```

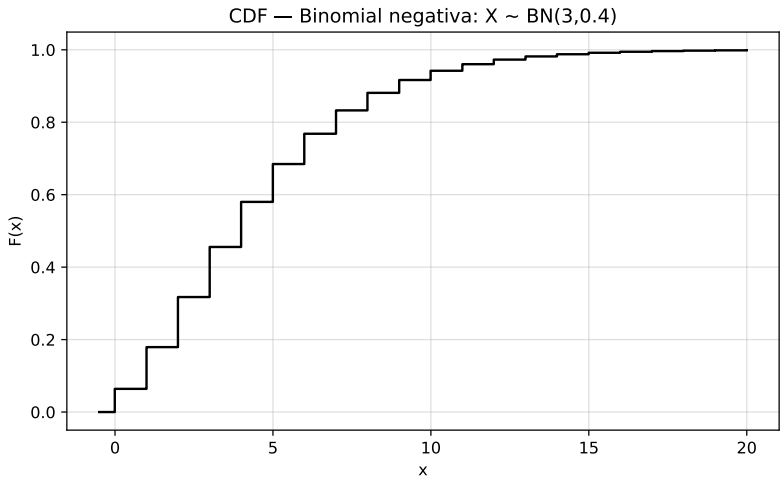
	x	PMF	CDF
0	0.0	0.064000	0.064000
1	1.0	0.115200	0.179200
2	2.0	0.138240	0.317440
3	3.0	0.138240	0.455680

	x	PMF	CDF
4	4.0	0.124416	0.580096

```
plt.bar(x, fx, color="0.6", edgecolor="black")
plt.title(f"PMF - Binomial negativa: X ~ BN({r},{p})")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.step(xg, Fxg, color="black")
plt.title(f"CDF - Binomial negativa: X ~ BN({r},{p})")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



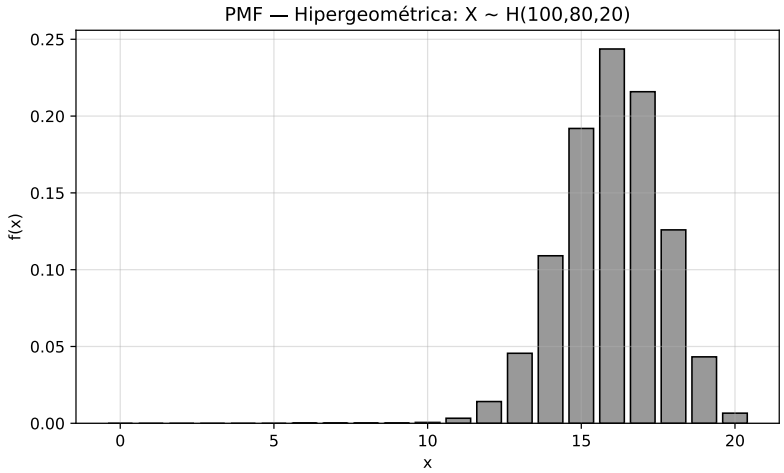
2.7 Hipergeométrica (`hypergeom`)

```
x = np.linspace(0, 20, num=21)
xg = np.linspace(0, 20, num=21)

N = 100  # población
m = 80   # éxitos en la población
n = 20   # muestra

fx = stats.hypergeom.pmf(x, N, m, n)
Fx = stats.hypergeom.cdf(x, N, m, n)
Fxg = stats.hypergeom.cdf(xg, N, m, n)

tabla_dist = pd.DataFrame({"x": x, "PMF": fx, "CDF": Fx})
tabla_dist.head()
```

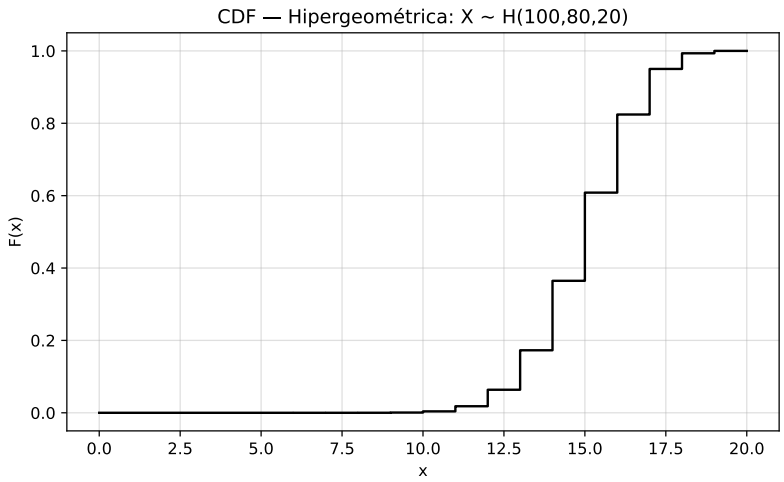
	x	PMF	CDF
0	0.0	1.865730e-21	1.865730e-21
1	1.0	2.985167e-18	2.987033e-18
2	2.0	1.120184e-15	1.123171e-15

	x	PMF	CDF
3	3.0	1.747487e-13	1.758719e-13
4	4.0	1.429663e-11	1.447250e-11

```
plt.bar(x, fx, color="0.6", edgecolor="black")
plt.title(f"PMF - Hipergeométrica: X ~ H({N},{m},{n})")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.step(xg, Fxg, color="black")
plt.title(f"CDF - Hipergeométrica: X ~ H({N},{m},{n})")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



---

3. Distribuciones continuas

3.1 Uniforme continua (`uniform`)

```
x = np.linspace(-4, 4, num=200)
xt = np.linspace(-4, 4, num=9)

alpha = -3
beta = 3

fx = stats.uniform.pdf(x, alpha, beta - alpha)
Fx = stats.uniform.cdf(x, alpha, beta - alpha)
Fxt = stats.uniform.cdf(xt, alpha, beta - alpha)

tabla_dist = pd.DataFrame({"x": xt, "CDF": Fxt})
tabla_dist
```

	x	CDF
0	-4.0	0.000000
1	-3.0	0.000000

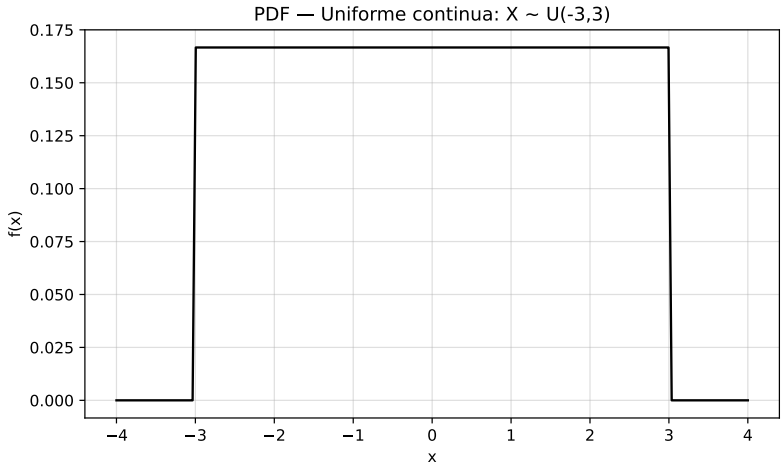
	x	CDF
2	-2.0	0.166667
3	-1.0	0.333333
4	0.0	0.500000
5	1.0	0.666667
6	2.0	0.833333
7	3.0	1.000000
8	4.0	1.000000

Sintaxis clave (uniform)

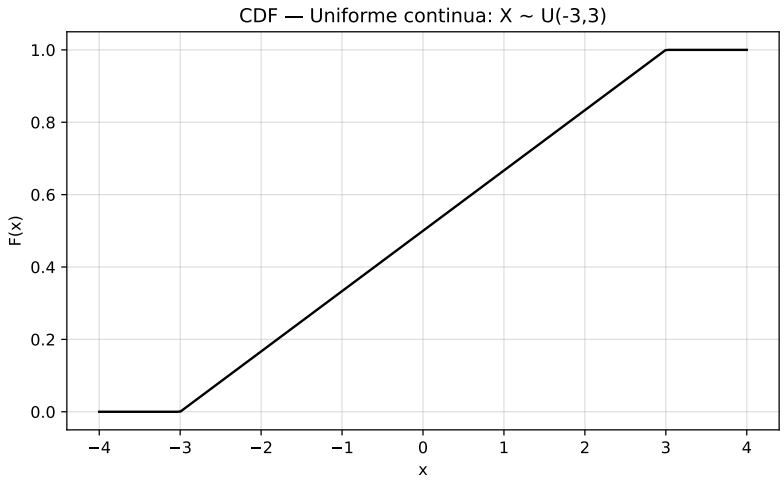
En `scipy`:

- `loc = alpha`
- `scale = beta - alpha`

```
plt.plot(x, fx, linestyle="-", color="black")
plt.title(f"PDF - Uniforme continua: X ~ U({alpha},{beta})")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



```
plt.plot(x, Fx, linestyle="-", color="black")
plt.title(f"CDF - Uniforme continua: X ~ U({alpha},{beta})")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



Cuantiles (ejemplo): intervalo central 95%.

```
q025 = stats.uniform.ppf(0.025, alpha, beta - alpha)
q975 = stats.uniform.ppf(0.975, alpha, beta - alpha)
q025, q975
```

```
(np.float64(-2.85), np.float64(2.8499999999999996))
```

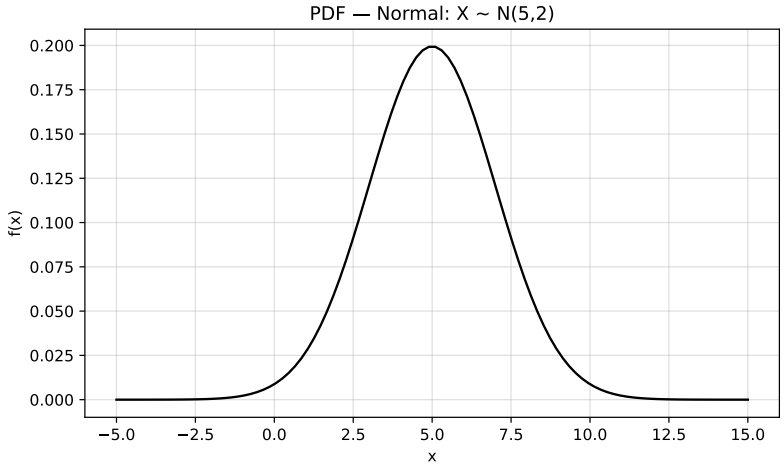
3.2 Normal (`norm`)

```
x = np.linspace(-5, 15, num=100)
xt = np.linspace(-5, 15, num=11)
```

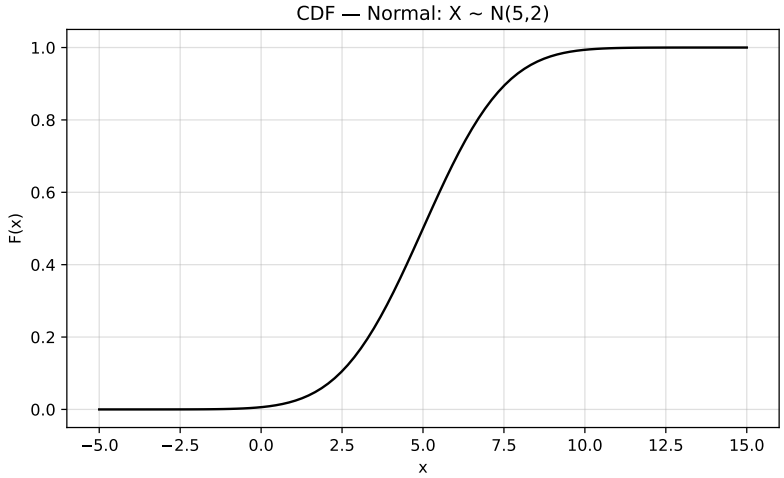
```
mu = 5
sigma = 2

fx = stats.norm.pdf(x, mu, sigma)
Fx = stats.norm.cdf(x, mu, sigma)
Fxt = stats.norm.cdf(xt, mu, sigma)

tabla_dist = pd.DataFrame({"x": xt, "CDF": Fxt})
tabla_dist
```

	x	CDF
0	-5.0	2.866516e-07
1	-3.0	3.167124e-05
2	-1.0	1.349898e-03
3	1.0	2.275013e-02
4	3.0	1.586553e-01
5	5.0	5.000000e-01
6	7.0	8.413447e-01
7	9.0	9.772499e-01
8	11.0	9.986501e-01
9	13.0	9.999683e-01
10	15.0	9.999997e-01

```
plt.plot(x, fx, linestyle="-", color="black")
plt.title(f"PDF - Normal: X ~ N({mu},{sigma})")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.plot(x, Fx, linestyle="-", color="black")
plt.title(f"CDF - Normal: X ~ N({mu},{sigma})")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



Cuantiles (intervalo central 95%):

```
q025 = stats.norm.ppf(0.025, mu, sigma)
q975 = stats.norm.ppf(0.975, mu, sigma)
q025, q975
```

```
(np.float64(1.080072030919891), np.float64(8.919927969080108))
```

Ejemplo de probabilidad: si `X ~ N(4, 9)`, calcula `P(2 < X 6)`.

```
stats.norm.cdf(6, 4, 3) - stats.norm.cdf(2, 4, 3)
```

```
np.float64(0.4950149249061542)
```

3.3 Normal estándar: estandarización

Aquí hacemos dos cosas:

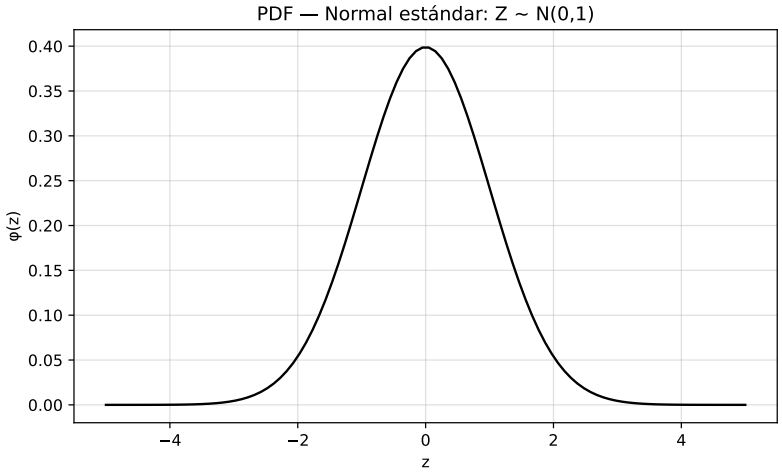
1) estandarizar valores: `z = (x - mu)/sigma`

2) usar la normal estándar `N(0,1)`.

```
x = np.linspace(-5, 15, num=100)
xt = np.linspace(-5, 15, num=11)

mu = 5
sigma = 2

z = (x - mu) / sigma
zt = (xt - mu) / sigma

fz = stats.norm.pdf(z)
Fz = stats.norm.cdf(z)
Fzt = stats.norm.cdf(zt)

tabla_dist = pd.DataFrame({"z": zt, "CDF": Fzt})
tabla_dist
```

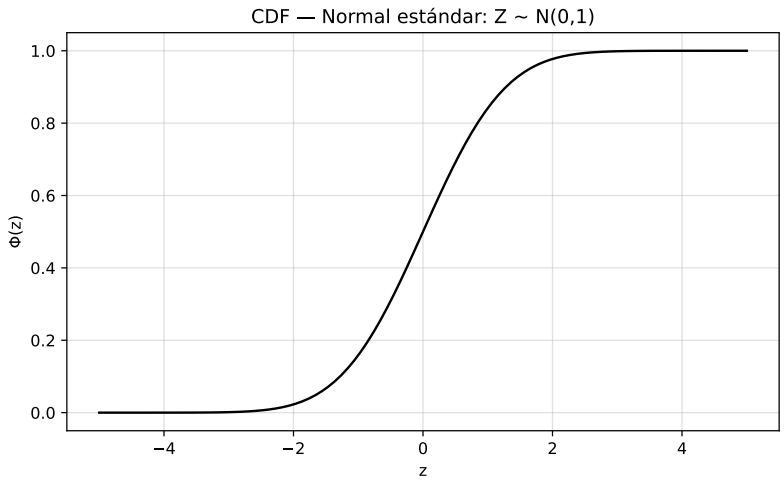
	z	CDF
0	-5.0	2.866516e-07
1	-4.0	3.167124e-05
2	-3.0	1.349898e-03
3	-2.0	2.275013e-02
4	-1.0	1.586553e-01
5	0.0	5.000000e-01
6	1.0	8.413447e-01
7	2.0	9.772499e-01
8	3.0	9.986501e-01
9	4.0	9.999683e-01
10	5.0	9.999997e-01

```
plt.plot(z, fz, linestyle="-", color="black")
plt.title("PDF - Normal estándar: Z ~ N(0,1)")
plt.xlabel("z")
plt.ylabel(" (z)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.plot(z, Fz, linestyle="-", color="black")
plt.title("CDF - Normal estándar: Z ~ N(0,1)")
plt.xlabel("z")
```

```
plt.ylabel("Φ(z)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



Cuantiles de la normal estándar (95% central):

```
stats.norm.ppf(0.025), stats.norm.ppf(0.975)
```

```
(np.float64(-1.9599639845400545), np.float64(1.959963984540054))
```

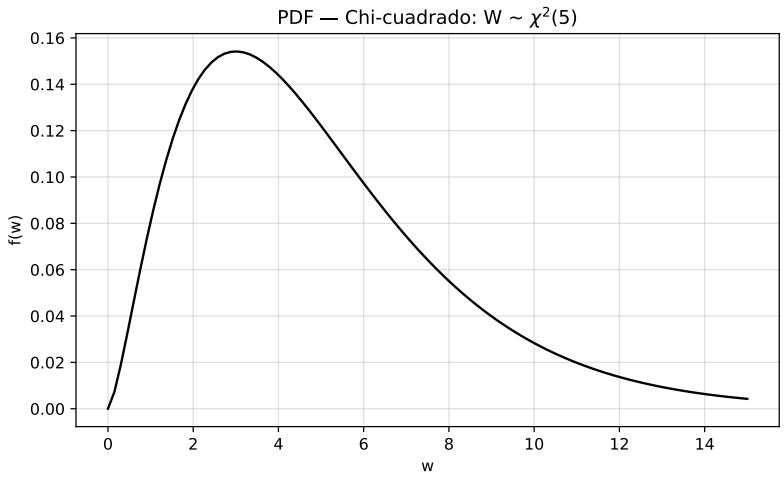
3.4 Chi-cuadrado (`chi2`)

```
w = np.linspace(0, 15, num=100)
wt = np.linspace(0, 15, num=16)

m = 5

fw = stats.chi2.pdf(w, m)
Fw = stats.chi2.cdf(w, m)
Fwt = stats.chi2.cdf(wt, m)
```

```
tabla_dist = pd.DataFrame({"w": wt, "CDF": Fwt})
tabla_dist.head()
```

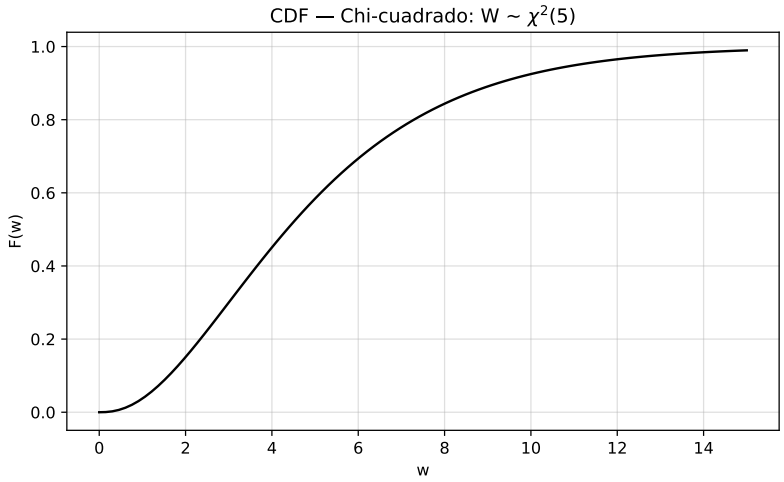
	w	CDF
0	0.0	0.000000
1	1.0	0.037434
2	2.0	0.150855
3	3.0	0.300014
4	4.0	0.450584

```
plt.plot(w, fw, linestyle="-", color="black")
plt.title(rf"PDF - Chi-cuadrado: W ~ $\chi^2$({m})")
plt.xlabel("w")
plt.ylabel("f(w)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.plot(w, Fw, linestyle="-", color="black")
plt.title(rf"CDF - Chi-cuadrado: W ~ $\chi^2$({m})")
plt.xlabel("w")
plt.ylabel("F(w)")
plt.grid(alpha=0.4)
```

```
plt.show()
plt.close()
```



Cuantil 0.95:

```
stats.chi2.ppf(0.95, m)
```

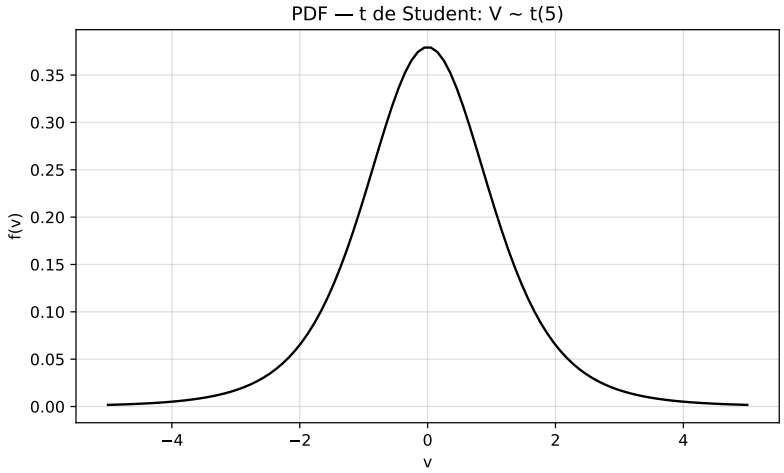
```
np.float64(11.070497693516351)
```

3.5 t de Student (`t`)

```
v = np.linspace(-5, 5, num=100)
vt = np.linspace(-5, 5, num=11)

m = 5

fv = stats.t.pdf(v, m)
Fv = stats.t.cdf(v, m)
Fvt = stats.t.cdf(vt, m)

tabla_dist = pd.DataFrame({"v": vt, "CDF": Fvt})
tabla_dist
```

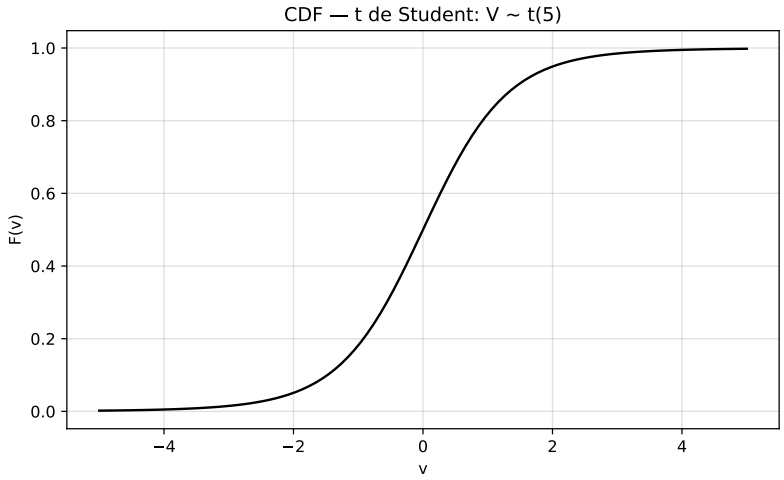
	v	CDF
0	-5.0	0.002052
1	-4.0	0.005162
2	-3.0	0.015050
3	-2.0	0.050970
4	-1.0	0.181609
5	0.0	0.500000
6	1.0	0.818391
7	2.0	0.949030
8	3.0	0.984950
9	4.0	0.994838
10	5.0	0.997948

```
plt.plot(v, fv, linestyle="-", color="black")
plt.title(f"PDF - t de Student: V ~ t({m})")
plt.xlabel("v")
plt.ylabel("f(v)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.plot(v, Fv, linestyle="-", color="black")
plt.title(f"CDF - t de Student: V ~ t({m})")
plt.xlabel("v")
```

```
plt.ylabel("F(v)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



Cuantiles 0.025 y 0.975:

```
stats.t.ppf(0.025, m), stats.t.ppf(0.975, m)
```

```
(np.float64(-2.5705818356363146), np.float64(2.570581835636314))
```

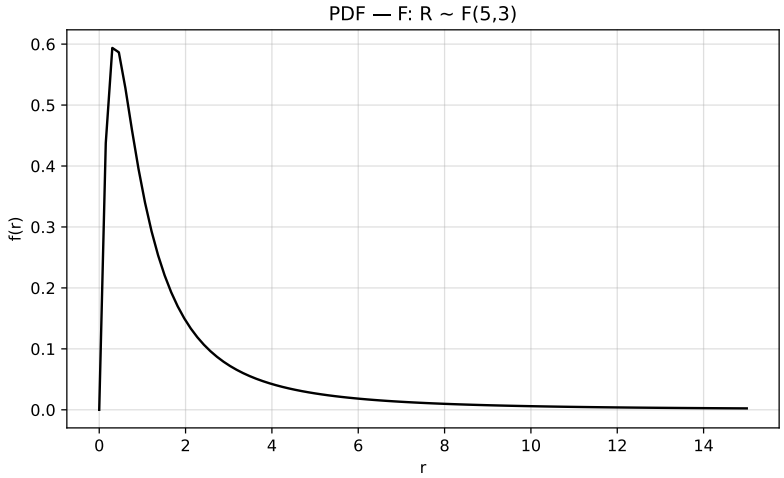
3.6 F de Fisher (`f`)

```
r = np.linspace(0, 15, num=100)
rt = np.linspace(0, 15, num=16)

m1 = 5
m2 = 3

fr = stats.f.pdf(r, m1, m2)
Fr = stats.f.cdf(r, m1, m2)
```

```
Frt = stats.f.cdf(rt, m1, m2)

tabla_dist = pd.DataFrame({"r": rt, "CDF": Frt})
tabla_dist.head()
```

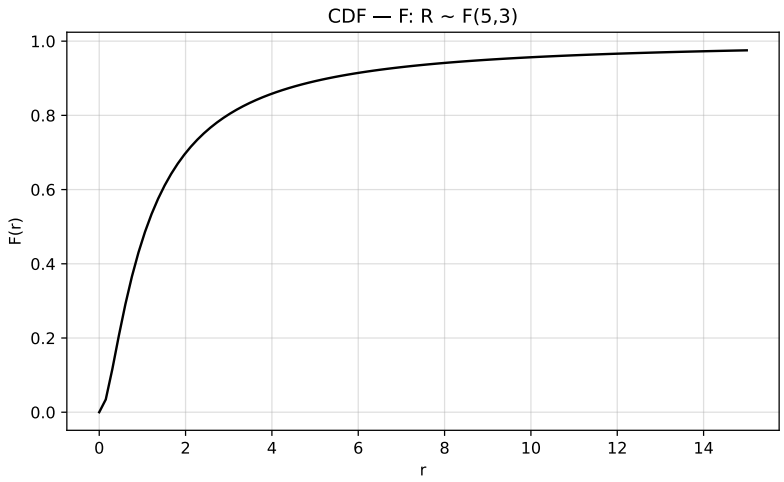
	r	CDF
0	0.0	0.000000
1	1.0	0.464855
2	2.0	0.698453
3	3.0	0.802577
4	4.0	0.858387

```
plt.plot(r, fr, linestyle="-", color="black")
plt.title(rf"PDF - F: R ~ F({m1},{m2})")
plt.xlabel("r")
plt.ylabel("f(r)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.plot(r, Fr, linestyle="-", color="black")
plt.title(rf"CDF - F: R ~ F({m1},{m2})")
plt.xlabel("r")
```

```
plt.ylabel("F(r)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



Cuantil 0.95:

```
stats.f.ppf(0.95, m1, m2)
```

```
np.float64(9.013455167522583)
```

3.7 Exponencial (`expon`)

```
x = np.linspace(-0.5, 6, num=150)
xt = np.linspace(-0.5, 6, num=9)

plambda = 1

fx = stats.expon.pdf(x, scale=1 / plambda)
Fx = stats.expon.cdf(x, scale=1 / plambda)
Fxt = stats.expon.cdf(xt, scale=1 / plambda)
```

```
tabla_dist = pd.DataFrame({"x": xt, "CDF": Fxt})
tabla_dist
```

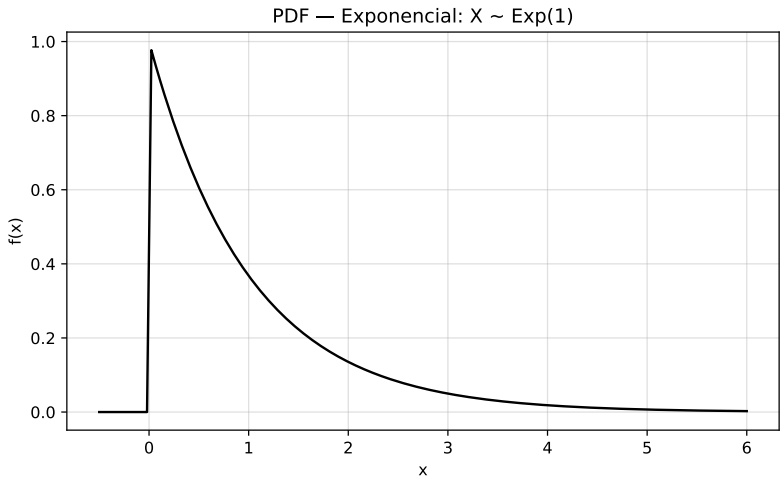
	x	CDF
0	-0.5000	0.000000
1	0.3125	0.268384
2	1.1250	0.675348
3	1.9375	0.855936
4	2.7500	0.936072
5	3.5625	0.971632
6	4.3750	0.987412
7	5.1875	0.994414
8	6.0000	0.997521

Sintaxis clave (expon)

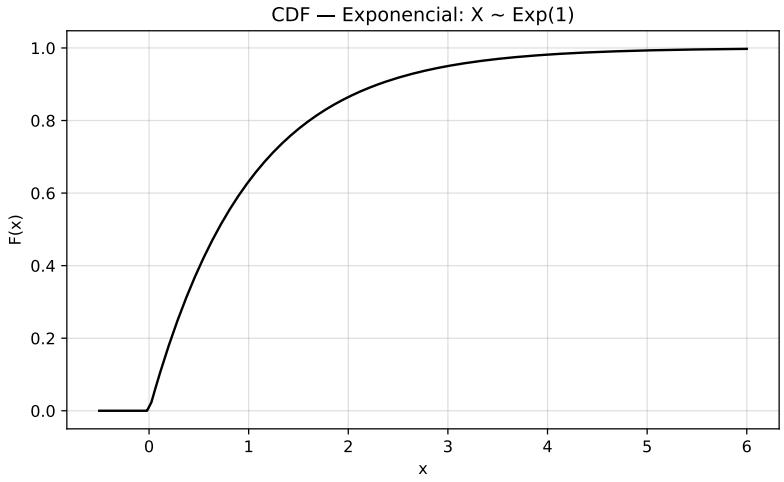
Si tú trabajas con tasa , en `scipy` se usa:

- `scale = 1/`

```
plt.plot(x, fx, linestyle="-", color="black")
plt.title(f"PDF - Exponencial: X ~ Exp({plambda})")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```




```
plt.plot(x, Fx, linestyle="-", color="black")
plt.title(f"CDF - Exponencial: X ~ Exp({plambda})")
plt.xlabel("x")
plt.ylabel("F(x)")
plt.grid(alpha=0.4)
plt.show()
plt.close()
```



Cuantil 0.95:

```
stats.expon.ppf(0.95, scale=1 / plambda)
```

```
np.float64(2.99573227355399)
```

---

4. Simulación (muestra vs teoría)

La simulación tiene dos ideas útiles:

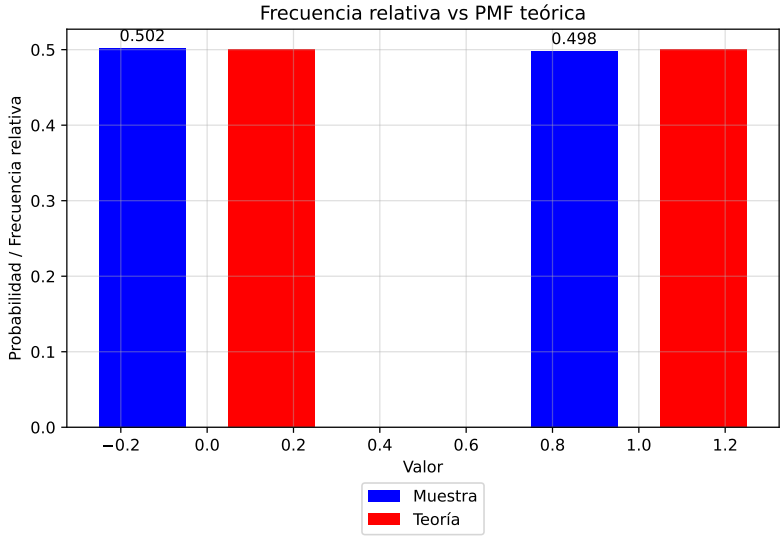
1) comparar "lo que salió" (frecuencia/histograma) con "la teoría" (PMF/PDF),

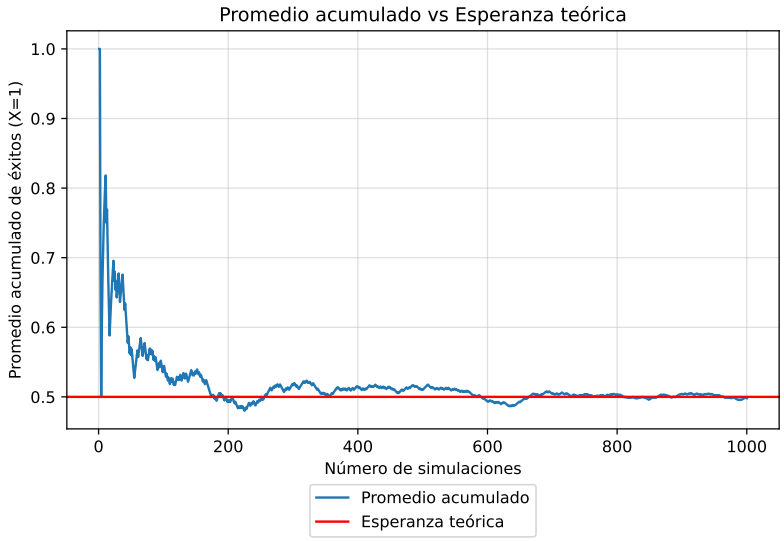
2) ver cómo el **promedio acumulado** converge a la esperanza teórica.

4.1 Bernoulli simulada

```
semilla = None  # None -> resultados diferentes cada ejecución
ns = 1000
p = 0.5

muestra = stats.bernoulli.rvs(p, size=ns, random_state=semilla)

# Frecuencias
valores, freq_abs = np.unique(muestra, return_counts=True)
freq_rel = freq_abs / ns

# PMF teórica
pmft = stats.bernoulli.pmf(valores, p)

# Comparación: muestra vs teoría
plt.bar(valores - 0.15, freq_rel, width=0.2, color="blue", label="Muestra")
plt.bar(valores + 0.15, pmft, width=0.2, color="red", label="Teoría")
```

```
for xx, yy in zip(valores - 0.15, freq_rel):
    plt.text(xx, yy + 0.005, f"{yy:.3f}", ha="center", va="bottom")
plt.xlabel("Valor")
plt.ylabel("Probabilidad / Frecuencia relativa")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Frecuencia relativa vs PMF teórica")
plt.show()
plt.close()

# Promedio acumulado
cum_ns = np.arange(1, ns + 1)
plt.plot(cum_ns, muestra.cumsum() / cum_ns, label="Promedio acumulado")
plt.axhline(y=p, color="red", label="Esperanza teórica")
plt.xlabel("Número de simulaciones")
plt.ylabel("Promedio acumulado de éxitos (X=1)")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Promedio acumulado vs Esperanza teórica")
plt.show()
plt.close()
```

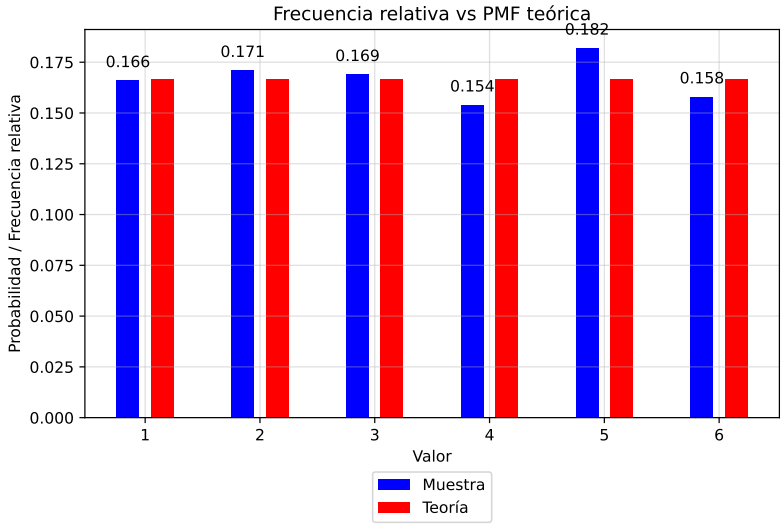


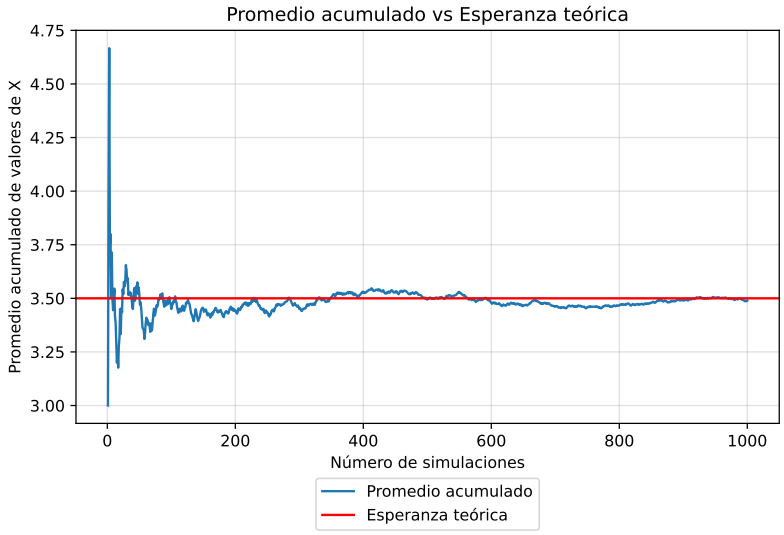


4.2 Uniforme discreta simulada

```
semilla = None
ns = 1000
a, b = 1, 6

muestra = stats.randint.rvs(a, b + 1, size=ns, random_state=semilla)

valores, freq_abs = np.unique(muestra, return_counts=True)
freq_rel = freq_abs / ns

pmft = stats.randint.pmf(valores, a, b + 1)

plt.bar(valores - 0.15, freq_rel, width=0.2, color="blue", label="Muestra")
plt.bar(valores + 0.15, pmft, width=0.2, color="red", label="Teoría")
for xx, yy in zip(valores - 0.15, freq_rel):
    plt.text(xx, yy + 0.005, f"{yy:.3f}", ha="center", va="bottom")
plt.xlabel("Valor")
plt.ylabel("Probabilidad / Frecuencia relativa")
plt.grid(alpha=0.4)
```

```
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Frecuencia relativa vs PMF teórica")
plt.show()
plt.close()

cum_ns = np.arange(1, ns + 1)
plt.plot(cum_ns, muestra.cumsum() / cum_ns, label="Promedio acumulado")
plt.axhline(y=(a + b) / 2, color="red", label="Esperanza teórica")
plt.xlabel("Número de simulaciones")
plt.ylabel("Promedio acumulado de valores de X")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Promedio acumulado vs Esperanza teórica")
plt.show()
plt.close()
```

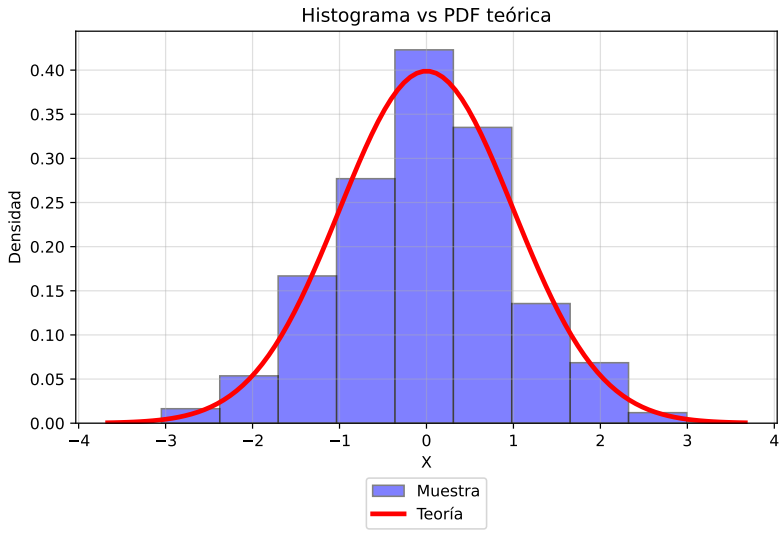


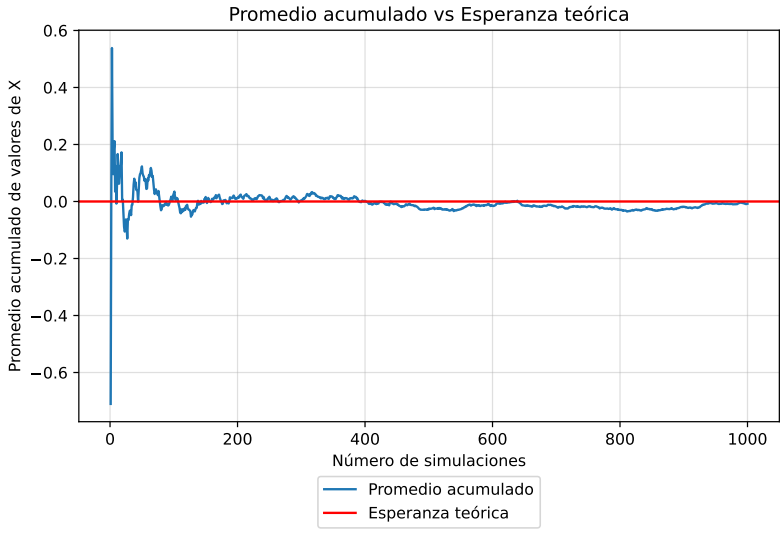


4.3 Normal simulada

```
semilla = None
ns = 1000
mu, sigma = 0, 1

muestra = stats.norm.rvs(mu, sigma, size=ns, random_state=semilla)

lim = max(abs(muestra.min()), abs(muestra.max()))
x = np.linspace(-lim, lim, num=100)
pdft = stats.norm.pdf(x, mu, sigma)

plt.hist(muestra, bins=10, density=True, edgecolor="black", color="blue", alpha=0.5, label="Mue
plt.plot(x, pdft, color="red", linewidth=3, label="Teoría")
plt.xlabel("X")
plt.ylabel("Densidad")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Histograma vs PDF teórica")
plt.show()
```

```
plt.close()

cum_ns = np.arange(1, ns + 1)
plt.plot(cum_ns, muestra.cumsum() / cum_ns, label="Promedio acumulado")
plt.axhline(y=mu, color="red", label="Esperanza teórica")
plt.xlabel("Número de simulaciones")
plt.ylabel("Promedio acumulado de valores de X")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Promedio acumulado vs Esperanza teórica")
plt.show()
plt.close()
```



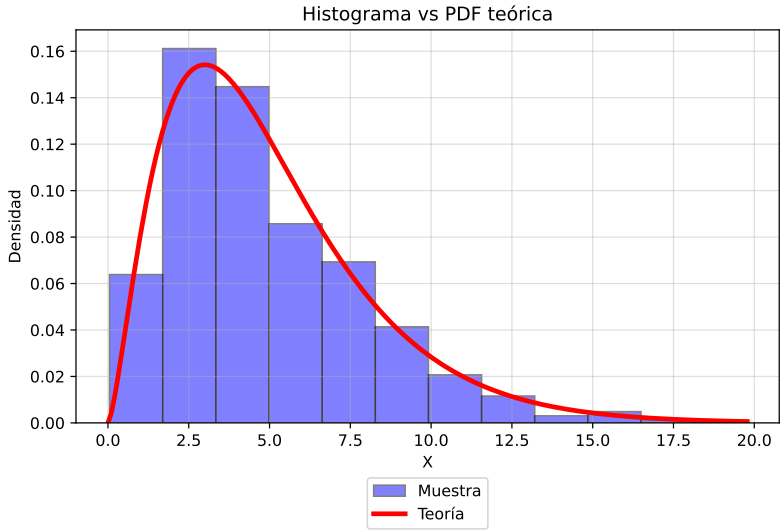


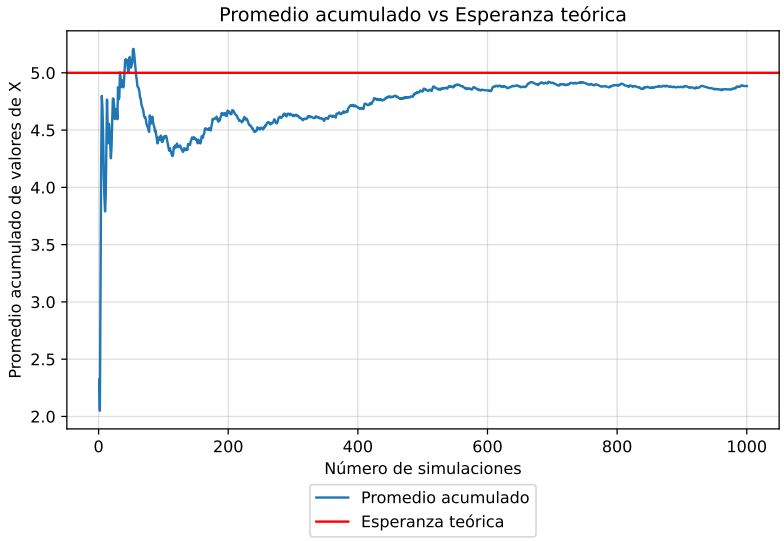
4.4 Chi-cuadrado simulada (por partes)

La chi-cuadrado puede verse como suma de cuadrados de normales estándar.

```
semilla = None
ns = 1000
m = 5

sim_partes = True
if sim_partes:
    sim_var2 = []
    for i in range(1, m + 1):
        semilla2 = None if semilla is None else semilla + i
        sim_var = stats.norm.rvs(size=ns, random_state=semilla2)  # Z ~ N(0,1)
        sim_var2.append(sim_var**2)
    muestra = sum(sim_var2)
else:
    muestra = stats.chi2.rvs(m, size=ns, random_state=semilla)

# PDF teórica
```

```
lim = max(abs(muestra.min()), abs(muestra.max()))
x = np.linspace(0, lim, num=200)
pdft = stats.chi2.pdf(x, m)

plt.hist(muestra, bins=12, density=True, edgecolor="black", color="blue", alpha=0.5, label="Mue
plt.plot(x, pdft, color="red", linewidth=3, label="Teoría")
plt.xlabel("X")
plt.ylabel("Densidad")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Histograma vs PDF teórica")
plt.show()
plt.close()

cum_ns = np.arange(1, ns + 1)
plt.plot(cum_ns, muestra.cumsum() / cum_ns, label="Promedio acumulado")
plt.axhline(y=m, color="red", label="Esperanza teórica")
plt.xlabel("Número de simulaciones")
plt.ylabel("Promedio acumulado de valores de X")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Promedio acumulado vs Esperanza teórica")
plt.show()
plt.close()
```



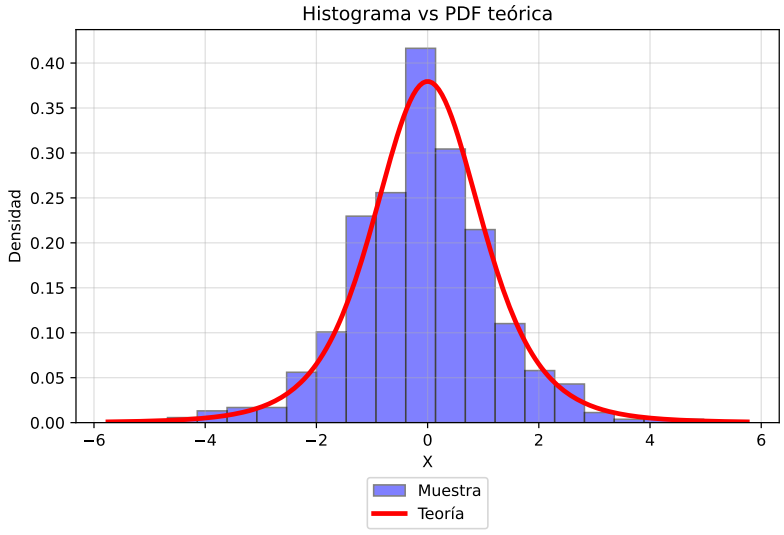


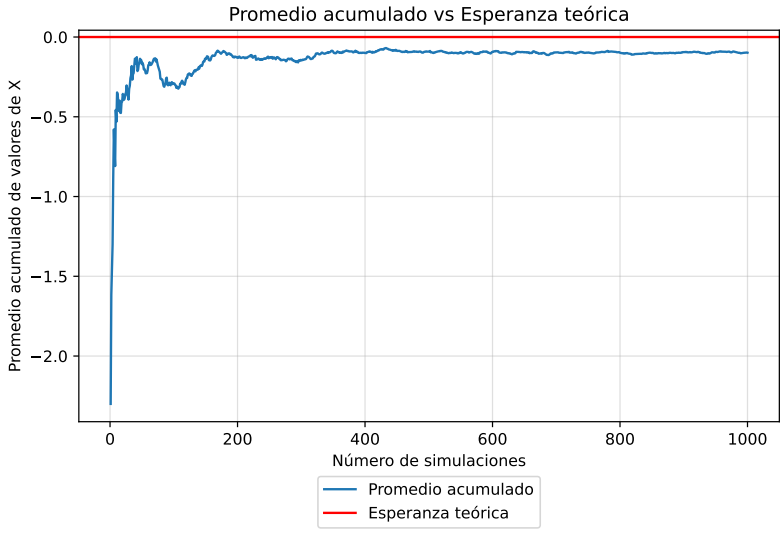
4.5 t de Student simulada (por construcción)

Una forma de construir una t es: `Z / sqrt(W/m)` con `Z ~ N(0,1)` y `W ~ chi^2(m)` independientes.

```
semilla = None
ns = 1000
m = 5

sim_partes = True
if sim_partes:
    semilla2 = None if semilla is None else semilla + 1
    sim_var_Z = stats.norm.rvs(size=ns, random_state=semilla)
    sim_var_W = stats.chi2.rvs(m, size=ns, random_state=semilla2)
    muestra = sim_var_Z / np.sqrt(sim_var_W / m)
else:
    muestra = stats.t.rvs(m, size=ns, random_state=semilla)

# PDF teórica
lim = max(abs(muestra.min()), abs(muestra.max()))
x = np.linspace(-lim, lim, num=200)
```

```
pdft = stats.t.pdf(x, m)

plt.hist(muestra, bins=20, density=True, edgecolor="black", color="blue", alpha=0.5, label="Mue
plt.plot(x, pdft, color="red", linewidth=3, label="Teoría")
plt.xlabel("X")
plt.ylabel("Densidad")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Histograma vs PDF teórica")
plt.show()
plt.close()

cum_ns = np.arange(1, ns + 1)
plt.plot(cum_ns, muestra.cumsum() / cum_ns, label="Promedio acumulado")
plt.axhline(y=0, color="red", label="Esperanza teórica")
plt.xlabel("Número de simulaciones")
plt.ylabel("Promedio acumulado de valores de X")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Promedio acumulado vs Esperanza teórica")
plt.show()
plt.close()
```





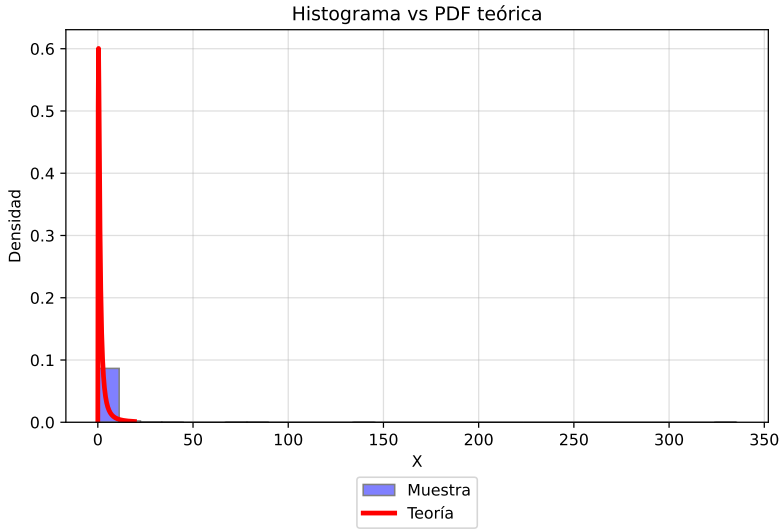
4.6 F de Fisher simulada (por construcción)

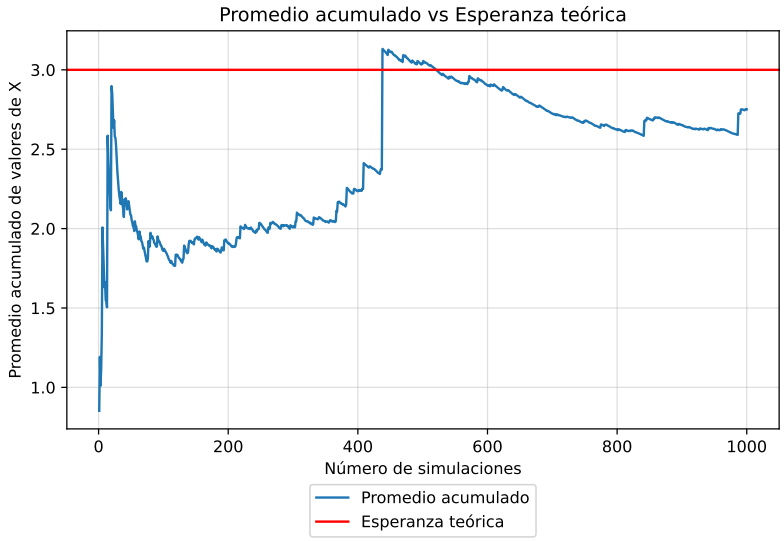
Construcción típica:

[R = (W_1/m_1) / (W_2/m_2),] donde `W1 ~ chi^2(m1)` y `W2 ~ chi^2(m2)` independientes.

```
semilla = None
ns = 1000
m1, m2 = 5, 3

sim_partes = True
if sim_partes:
    semilla2 = None if semilla is None else semilla + 1
    sim_var_W1 = stats.chi2.rvs(m1, size=ns, random_state=semilla)
    sim_var_W2 = stats.chi2.rvs(m2, size=ns, random_state=semilla2)
    muestra = (sim_var_W1 / m1) / (sim_var_W2 / m2)
else:
    muestra = stats.f.rvs(m1, m2, size=ns, random_state=semilla)

# PDF teórica
lim = np.quantile(muestra, 0.99)  # evita que un extremo gigante "rompa" la gráfica
```

```
x = np.linspace(0, lim, num=200)
pdft = stats.f.pdf(x, m1, m2)

plt.hist(muestra, bins=30, density=True, edgecolor="black", color="blue", alpha=0.5, label="Mue
plt.plot(x, pdft, color="red", linewidth=3, label="Teoría")
plt.xlabel("X")
plt.ylabel("Densidad")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Histograma vs PDF teórica")
plt.show()
plt.close()

cum_ns = np.arange(1, ns + 1)
plt.plot(cum_ns, muestra.cumsum() / cum_ns, label="Promedio acumulado")
plt.axhline(y=m2 / (m2 - 2), color="red", label="Esperanza teórica")
plt.xlabel("Número de simulaciones")
plt.ylabel("Promedio acumulado de valores de X")
plt.grid(alpha=0.4)
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.title("Promedio acumulado vs Esperanza teórica")
plt.show()
plt.close()
```





Ejercicios propuestos

1) **Binomial (PMF y CDF)**
Para `X ~ Bn(10, 0.2)`, calcula `P(X=2)` y `P(X 2)`.

   Respuesta esperada: dos números en (0, 1).

2) **Cuantiles (normal)**
Para `X ~ N(5,2)`, calcula el intervalo central del 95% usando `ppf(0.025)` y `ppf(0.975)`.

   Respuesta esperada: dos valores (límite inferior y superior).

3) **Normal estándar**
Estandariza `x=9` si `X ~ N(5,2)` y calcula `P(X 9)` usando `Φ(z)` (CDF estándar).

   Respuesta esperada: un valor cercano a `0.977` (aprox.).

4) **Simulación Bernoulli**
Simula `ns=5000` con `p=0.3` y grafica el promedio acumulado.

Respuesta esperada: el promedio acumulado se acerca a `0.3`.

5) **Exponencial**
Con `=1`, calcula el cuantil 0.95 con `ppf`.

Respuesta esperada: un valor cercano a `3`.

6) **F de Fisher**
Para `F(5,3)`, calcula el cuantil 0.95.

Respuesta esperada: un número positivo mayor que 1.

Glosario (para Colab)

- **PMF/PDF/CDF**: masa / densidad / acumulada.
- **PPF**: cuantil (inversa de la CDF).
- **rvs**: simulación (muestra aleatoria).
- **loc/scale**: parámetros estándar de `scipy`.

Capítulo 7 — Intervalos de confianza e inferencia (t-test)

Abrir este capítulo en Google Colab

Objetivos del capítulo

En este capítulo vas a programar (en Python) el flujo completo de inferencia con una muestra:

- calcular un **intervalo de confianza (IC)** para un promedio,
- construir una **prueba t** "a mano": estadístico t, valor crítico y valor p,
- comparar con la versión automática `ttest_1samp`,
- evitar errores típicos: `ddof`, **una cola vs dos colas**, y **cuantiles mal puestos**.

---

1. Una muestra: datos y objetivo

Vamos a trabajar con una muestra pequeña (sin depender de internet).
Tenemos una medida en dos años y queremos analizar la **variación**.

Glosario

- **estimador**: cantidad calculada con la muestra (ej. promedio).
- **SE (error estándar)**: variabilidad del estimador; típico `s / sqrt(n)`.
- **ddof=1**: ajuste de "grados de libertad" para la desviación estándar muestral.
- **cuantil / ppf**: "punto de corte" de una distribución; `ppf(q, df)`.
- **valor crítico**: cuantil que delimita la región de rechazo.
- **valor p**: probabilidad (bajo H0) de ver un estadístico igual o más extremo.
- **una cola / dos colas**: define qué significa "extremo".

2. Construcción de la base y variable de interés

```
TD87 = np.array([10, 1, 6, .45, 1.25, 1.3, 1.06, 3, 8.18, 1.67, .98, 1, .45, 5.03, 8, 9, 18, .
TD88 = np.array([3, 1, 5, .5, 1.54, 1.5, .8, 2, .67, 1.17, .51, .5, .61, 6.7, 4, 7, 19, .2, 5,

variacion = TD88 - TD87
n = len(variacion)

baseTD = pd.DataFrame({
    "Empresa": np.arange(1, n + 1),
    "Tasa (1987)": TD87,
    "Tasa (1988)": TD88,
    "Variación": variacion
})

baseTD.head()
```

	Empresa	Tasa (1987)	Tasa (1988)	Variación
0	1	10.00	3.00	-7.00
1	2	1.00	1.00	0.00
2	3	6.00	5.00	-1.00
3	4	0.45	0.50	0.05
4	5	1.25	1.54	0.29

i Sintaxis clave (lo mínimo que debes dominar)

- `np.array([...])`: crea un arreglo numérico.
- `TD88 - TD87`: resta **elemento a elemento** (vectorizada).
- `np.arange(1, n+1)`: números 1,2,…,n.
- `pd.DataFrame({...})`: arma una tabla por columnas.

Tip rápido

Si `TD87` y `TD88` no tienen el mismo tamaño, la resta falla. Chequea con `len(TD87)` y `len(TD88)` antes de restar.

---

3. Intervalo de confianza para el promedio (con t)

3.1 Promedio, desviación estándar y error estándar

```
promedio = np.mean(variacion)
s = np.std(variacion, ddof=1)        # desviación estándar muestral
se = s / np.sqrt(n)                  # error estándar del promedio

promedio, s, se
```

```
(np.float64(-1.1544999999999999),
 np.float64(2.4006391144640498),
 np.float64(0.5367992249386514))
```

Cómo leer esta sintaxis

- `np.mean(x)` calcula el promedio de `x`.
- `np.std(x, ddof=1)` usa **ddof=1**: esto es la desviación estándar **muestral** (no poblacional).
- `se = s / sqrt(n)` es el error estándar del promedio.

Error común #1: olvidar `ddof=1`

Si usas `np.std(x)` sin `ddof=1`, obtienes la desviación estándar **poblacional**.
Para inferencia típica con muestra, eso te cambia el SE y el IC.

3.2 Cuantil t y el IC 95% (bilateral)

Para un IC 95% bilateral:

- $\quad = 0.05$
- colas: $\quad /2 = 0.025$ en cada cola
- por eso el cuantil es **0.975** $(1 - 0.025)$

```
c = stats.t.ppf(0.975, n - 1)
LI95 = promedio - c * se
LS95 = promedio + c * se
LI95, LS95
```

```
(np.float64(-2.2780336901843095), np.float64(-
0.030966309815690485))
```

> **Sintaxis clave (ppf)**
>
> - `stats.t.ppf(q, df)` devuelve el cuantil `q` de una t con `df` grados de libertad.
> - Para IC bilateral al 95%: `q = 0.975`, `df = n - 1`.

> **Error común #2: usar 0.95 en vez de 0.975**
>
> Para IC bilateral necesitas 0.975, no 0.95.
> Usar 0.95 te da un IC **más estrecho** de lo correcto.

---

4. Prueba t "manual" (una cola a la izquierda)

Ahora probamos si el promedio de la variación es **menor que 0**:

- H0: $\quad = 0$
- H1: $\quad < 0$

4.1 Estadístico t

```
mu0 = 0
t_manual = (promedio - mu0) / se
t_manual
```

```
np.float64(-2.150711003973493)
```

> **Sintaxis clave (estadístico t)**
>
> - La forma es `(promedio - mu0) / SE`.
> - Si `promedio` es negativo, `t_manual` tiende a ser negativo.

4.2 Valor crítico (cola izquierda al 5%)

```
val_critico = stats.t.ppf(0.05, n - 1)
val_critico
```

```
np.float64(-1.7291328115213678)
```

> **Regla de decisión (cola izquierda)**
>
> Rechazas H0 si:
>
> - `t_manual < val_critico`

4.3 Valor p (cola izquierda)

En cola izquierda, el valor p se puede calcular con la CDF:

```
p_una_cola = stats.t.cdf(t_manual, n - 1)
p_una_cola
```

```
np.float64(0.02229062646839212)
```

> **Por qué esto funciona (en código)**
>
> - `stats.t.cdf(t, df)` te da `P(T  t)`.
> - En cola izquierda, "más extremo" significa "más pequeño".

> **Error común #3: usar dos colas por accidente**
>
> Si tu hipótesis es < 0, el valor p es de **una cola**.
> Usar dos colas te da un p-value distinto y puede cambiar tu conclusión.

---

5. Prueba t automática (`ttest_1samp`)

```
test = stats.ttest_1samp(variacion, popmean=0)  # por defecto es bilateral
t_auto = test.statistic
p_bilateral = test.pvalue

t_auto, p_bilateral
```

```
(np.float64(-2.150711003973493), np.float64(0.04458125293678424))
```

Para una cola (izquierda), si `t_auto` es negativo:

```
p_una_cola_auto = p_bilateral / 2
p_una_cola_auto
```

```
np.float64(0.02229062646839212)
```

Tip: valida el signo antes de dividir entre 2

Dividir entre 2 solo tiene sentido si el estadístico va "en la dirección" de tu H1.
Para H1: < 0, esperas `t` **negativo**.

---

6. Ejemplo práctico con una variable `y` en {-1, 0, 1} (local)

Aquí usamos un ejemplo tipo "auditoría" con una variable de decisión:

- `-1`, `0`, `1` (tres resultados posibles).

```
audit_like = pd.DataFrame({
    "y": [-1, 0, 0, -1, 1, 0, -1, 0, 0, 1,
          0, 0, -1, 0, 1, 0, 0, -1, 0, 0,
          1, 0, -1, 0, 0, 0, -1, 0, 0, 1]
})
audit_like["y"].value_counts()
```

```
y
 0    18
-1     7
 1     5
Name: count, dtype: int64
```

6.1 IC 95% y 99% usando cuantiles normales

```
y = audit_like["y"]
n = len(y)

promedio_y = np.mean(y)
s_y = np.std(y, ddof=1)
```

```
se_y = s_y / np.sqrt(n)

c95 = stats.norm.ppf(0.975)
c99 = stats.norm.ppf(0.995)

IC95 = (promedio_y - c95 * se_y, promedio_y + c95 * se_y)
IC99 = (promedio_y - c99 * se_y, promedio_y + c99 * se_y)

IC95, IC99
```

```
((np.float64(-0.29557037914190526), np.float64(0.1622370458085719)),
 (np.float64(-0.36749713852006516), np.float64(0.23416380518673185)))
```

> **Sintaxis clave (normal)**
>
> - `stats.norm.ppf(0.975)` 1.96 (IC 95% bilateral).
> - `stats.norm.ppf(0.995)` 2.576 (IC 99% bilateral).

6.2 Prueba t (dos colas) para y

```
test_y = stats.ttest_1samp(y, popmean=0)
test_y.statistic, test_y.pvalue
valc_2colas = stats.t.ppf(0.025, n - 1)
print(f"t (automático): {test_y.statistic:.4f}")
print(f"valor crítico (2 colas): {valc_2colas:.4f}")
print(f"valor p (2 colas): {test_y.pvalue:.4f}")
```

```
t (automático): -0.5708
valor crítico (2 colas): -2.0452
valor p (2 colas): 0.5725
```

---

7. Aplicación con datos reales (opcional, online)

```
# En VSCode: pip install wooldridge
# En Colab:  !pip -q install wooldridge
import wooldridge as woo
import numpy as np
from scipy import stats

audit = woo.dataWoo("audit")
y = audit["y"]

promedio_y = np.mean(y)
n = len(y)
se_y = np.std(y, ddof=1) / np.sqrt(n)

c95 = stats.norm.ppf(0.975)
print("IC 95%:", promedio_y - c95*se_y, promedio_y + c95*se_y)

print(stats.ttest_1samp(y, popmean=0))
```

Ejercicios propuestos

1) **IC 95% (t)**
   Con `variacion`, calcula el IC 95% usando `ppf(0.975, n-1)`.

   Respuesta esperada: dos números (LI, LS).

2) **t manual**
   Calcula `t_manual = (promedio - 0) / se`.

   Respuesta esperada: un número (signo depende del promedio).

3) **Valor crítico (cola izquierda)**
Calcula `stats.t.ppf(0.05, n-1)`.

Respuesta esperada: un número negativo.

4) **Valor p (cola izquierda)**
Calcula `stats.t.cdf(t_manual, n-1)`.

Respuesta esperada: número entre 0 y 1.

5) **Automático**
Calcula `stats.ttest_1samp(variacion, 0).statistic`.

Respuesta esperada: muy cercano a `t_manual`.

6) **IC 99% (normal)**
Para `audit_like["y"]`, calcula IC 99% con `stats.norm.ppf(0.995)`.

Respuesta esperada: intervalo más ancho que el IC 95%.

Glosario (para Colab)

- **SE**: error estándar del promedio.
- **ddof=1**: ajuste muestral en desviación estándar.
- **ppf**: cuantil (inversa de CDF).
- **valor crítico**: punto de corte para decisión.
- **valor p**: probabilidad bajo H0.

Capítulo 8 — Python avanzado: condicionales, bucles, funciones y SymPy

Abrir este capítulo en Google Colab

Objetivos del capítulo

- Usar **condicionales** (`if/elif/else`) para tomar decisiones sin "romper" el programa.
- Escribir **bucles** (`for` y `while`) con acumuladores y contadores, sin errores de lógica.
- Crear **funciones** reutilizables con validación de argumentos.
- (Opcional) Usar **SymPy** para resolver ecuaciones y graficar funciones implícitas sin depender de "cálculo manual".

---

1. Condicionales y bucles (patrones que vas a usar siempre)

1.1 Bucle + condicional (ejemplo básico)

```
# Ejemplo básico de un bucle y un condicional
i = 1
for x in [1, 7, 20, 12, 9, 17]:
    if x % 2 == 0:
```

Glosario

- **condicional**: bloque que se ejecuta si se cumple una condición (`if`).
- **bucle**: repite un bloque de código (ej. `for`, `while`).
- **acumulador**: variable que "suma"/"junta" resultados dentro de un bucle.
- **índice**: posición dentro de una secuencia (0, 1, 2, …).
- **return**: salida de una función (lo que la función "devuelve").
- **argumento**: valor que le pasas a una función (ej. `raizp(9)`).
- **SymPy**: librería para matemáticas simbólicas (resolver/derivar/ecuaciones).
- **función implícita**: ecuación del tipo `F(x, y) = 0` (no despejada).

```
        print(i, x, "es par")
    else:
        print(i, x, "es impar")
    i += 1
```

```
1 1 es impar
2 7 es impar
3 20 es par
4 12 es par
5 9 es impar
6 17 es impar
```

Sintaxis (en simple)

- `for x in [...]`: recorre elementos de una lista (uno por uno).
- `if x % 2 == 0`: condición (aquí: "¿x es divisible para 2?").
- `i += 1`: abreviación de `i = i + 1` (incrementa el contador).

Error común

Olvidar `i += 1` deja el contador "congelado" y el resultado se repite mal.

1.2 Bucle con secuencia no numérica y acumulador

```
# Ejemplo de bucle con secuencia no numérica y variable que se acumula
lista = ["Economía", "Estadística", "Python", "Datos"]

acum = ""
for palabra in lista:
    acum = acum + palabra + " | "

acum
```

```
'Economía | Estadística | Python | Datos | '
```

> **Para qué sirve esto**
>
> Un acumulador es útil cuando: - construyes texto, - construyes una lista de resultados, - sumas valores (ej. suma total, promedio, etc.).

> **Tip**
>
> Para textos muy largos, a veces es mejor acumular en lista y al final usar `"".join(...)`. Aquí lo hacemos simple a propósito.

1.3 Índices en un `for` (cuando necesitas posición)

```
# Ejemplo de bucle donde el contador se usa como índice
x = [10, 2, 7, 15]
y = [0] * len(x)  # crea una lista de ceros del mismo tamaño

for i in range(len(x)):
    y[i] = x[i] ** 2

x, y
```

```
([10, 2, 7, 15], [100, 4, 49, 225])
```

> **Sintaxis clave**
>
> - `range(len(x))` produce 0, 1, 2, …, `len(x)-1`.
> - `y = [0] * len(x)` reserva espacio para escribir resultados.
> - `x[i]` es “el elemento i-ésimo” (ojo: Python empieza en 0).

> ⚠ Error común
>
> Confundir índices: el primer elemento es `x[0]`, no `x[1]`.

1.4 El mismo ejemplo usando `while`

```
# Ejemplo anterior usando "while"
x = [10, 2, 7, 15]
y = [0] * len(x)

i = 0
while i < len(x):
    y[i] = x[i] ** 2
    i += 1

x, y
```

```
([10, 2, 7, 15], [100, 4, 49, 225])
```

> ℹ Cuándo usar `while`
>
> Úsalo cuando no sabes cuántas repeticiones necesitas de antemano, y la condición "de parada" depende de lo que ocurre dentro del bucle.

> ⚠ Error común (grave)
>
> Un `while` sin `i += 1` (o sin actualizar la condición) puede volverse un **bucle infinito**.

---

2. Funciones (para no copiar/pegar código)

Una función te permite empaquetar un procedimiento y reutilizarlo:

- Recibe **argumentos**
- Hace un proceso
- Devuelve un resultado con `return`

2.1 Ejemplo: raíz cuadrada solo para positivos

```
# Definimos función que obtiene la raíz cuadrada pero solo de números positivos
def raizp(x):
    if x < 0:
        return None
    return np.sqrt(x)

raizp(9), raizp(-9)
```

```
(np.float64(3.0), None)
```

Sintaxis clave

- `def nombre(parametro):` define una función.
- `return ...` devuelve un resultado y termina la función.
- `return None` es una forma simple de decir "no hay resultado válido".

Tip de calidad

En proyectos reales, muchas veces conviene lanzar un error con `raise ValueError(...)`. Aquí usamos `None` para mantenerlo amigable para principiantes.

2.2 Función lógica: ¿es potencia de 3?

```
# Definimos función para saber si un número n es potencia de 3
def es_potencia_de_3(n):
    if n < 1:
        return False
    while n % 3 == 0:
        n = n // 3  # división entera
    return n == 1

for k in [1, 3, 9, 12, 27, 28, 81]:
    print(k, es_potencia_de_3(k))
```

```
1 True
3 True
9 True
12 False
27 True
28 False
81 True
```

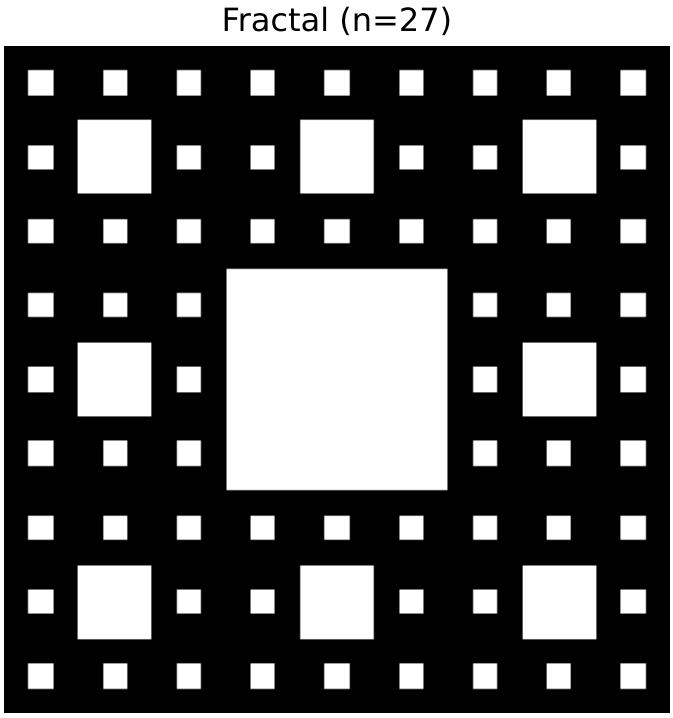
Sintaxis clave

- `n // 3` divide y se queda con la parte entera.
- `while n % 3 == 0` sigue mientras "sea divisible por 3".
- al final preguntamos si quedó `n == 1`.

2.3 Función aplicada: fractal tipo "tapete" (Sierpiński) (visual)

En este ejemplo, **n debe ser potencia de 3**.

```python
# Definimos función para crear un fractal (tapete de Sierpiński) (n debe ser potencia de 3)
def ej_fractal(n):
    if not es_potencia_de_3(n):
        raise ValueError("n debe ser potencia de 3 (ej. 3, 9, 27, 81, ...)")

    A = np.ones((n, n), dtype=int)
    step = n

    while step > 1:
        step = step // 3
        for i in range(0, n, step * 3):
            for j in range(0, n, step * 3):
                A[i + step:i + 2 * step, j + step:j + 2 * step] = 0

    plt.imshow(A, cmap="gray_r")
    plt.axis("off")
    plt.title(f"Fractal (n={n})")
    plt.show()
    plt.close()

ej_fractal(27)
```



Tips y errores comunes

- Si usas un `n` que no sea potencia de 3, este ejemplo **debe fallar**: eso es intencional (validación).
- Si tu entorno no muestra la figura, revisa que estés ejecutando en un notebook (Colab/Jupyter) o en Quarto con ejecución habilitada.

---

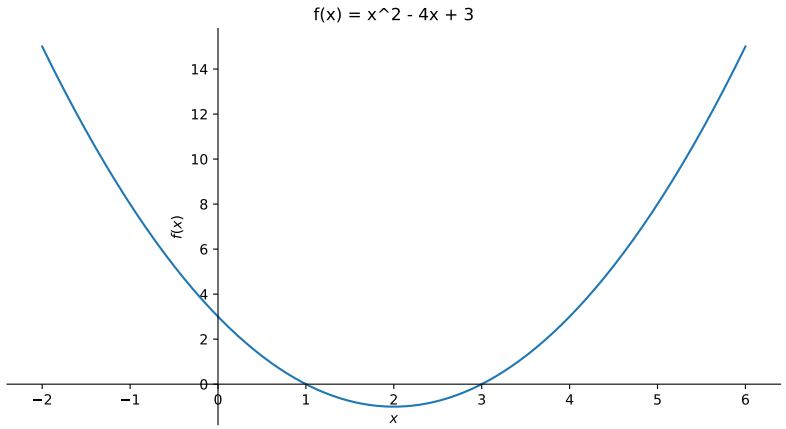
3. SymPy (opcional): ecuaciones, derivadas y gráficas implícitas

Si SymPy no está disponible, el manual no se rompe: simplemente no ejecutamos esta parte.

```
if sp is None:
    print("SymPy no está instalado en este entorno. En Colab puedes instalar con: !pip -q insta
```

3.1 Expresiones simbólicas y graficar una función

```
if sp is not None:
    # Importamos módulo SymPy (Symbolic Python: biblioteca para matemáticas simbólicas)
    x = sp.Symbol("x")

    # Ejemplo: parábola
    f = x**2 - 4*x + 3

    # Graficar con SymPy (rápido)
    sp.plot(f, (x, -2, 6), title="f(x) = x^2 - 4x + 3", show=True)
```



Sintaxis clave (SymPy)

- `sp.Symbol("x")` crea una variable simbólica.
- `f = ...` crea una expresión simbólica.
- `sp.plot(...)` grafica en un rango.

3.2 Resolver un sistema de ecuaciones (dos ecuaciones)

```
if sp is not None:
    x, y = sp.symbols("x y")

    eq1 = sp.Eq(x + y, 10)
    eq2 = sp.Eq(2*x - y, 3)

    sol = sp.solve([eq1, eq2], [x, y], dict=True)
    sol
```

> **Tip**
>
> `dict=True` te devuelve la solución como diccionario, más cómodo para leer y reemplazar.

3.3 Graficar funciones implícitas (una o varias)

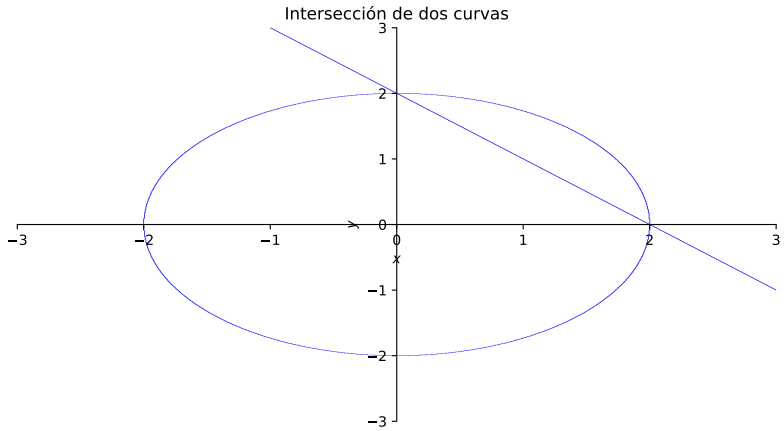
A veces tienes ecuaciones del tipo:

- `F(x, y) = 0`

y quieres graficar su curva.

```
if sp is not None:
    x, y = sp.symbols("x y")

    # Función para graficar varias funciones implícitas
    def plot_implicitas(ecuaciones, xlim=(-5, 5), ylim=(-5, 5), titulo="Funciones implícitas")
        p = sp.plot_implicit(ecuaciones[0], (x, xlim[0], xlim[1]), (y, ylim[0], ylim[1]),
                             show=False)
        for eq in ecuaciones[1:]:
            p2 = sp.plot_implicit(eq, (x, xlim[0], xlim[1]), (y, ylim[0], ylim[1]),
                                  show=False)
            for s in p2:
                p.append(s)
```

```
        p.title = titulo
        p.show()

    # Planeando y resolviendo un sistema de dos ecuaciones (y graficando)
    eq1 = sp.Eq(x + y - 2, 0)
    eq2 = sp.Eq(x**2 + y**2 - 4, 0)

    plot_implicitas([eq1, eq2], xlim=(-3, 3), ylim=(-3, 3), titulo="Intersección de dos curvas"
```



> **Error común**
>
> En `plot_implicit` las ecuaciones deben quedar como igualdad a cero (o con `Eq(..., 0)`). Si no, SymPy a veces interpreta mal lo que quieres graficar.

Ejercicios propuestos

1) **Par o impar (bucle + condicional)**
   Crea una lista con 10 números enteros y escribe un bucle que imprima si cada uno es par o impar.

   Respuesta esperada: 10 líneas, cada una con “par” o “impar”.

2) **Acumulador de texto**
Dada una lista de palabras, construye un string final separado por comas.

Respuesta esperada: un string tipo `"a, b, c, d"`.

3) **Cuadrados con índice**
Dada una lista `x`, crea `y` con los cuadrados, pero esta vez usando un `while`.

Respuesta esperada: `y[i] == x[i]**2` para todo i.

4) **Función de validación**
Escribe una función `solo_positivos(x)` que devuelva `x` si `x>0` y `None` si no.

Respuesta esperada: `solo_positivos(3)=3` y `solo_positivos(-1)=None`.

5) **Potencia de 3**
Prueba tu función `es_potencia_de_3` con los valores 1, 2, 3, 9, 18, 27.

Respuesta esperada: `True` solo para 1, 3, 9, 27.

6) *(Opcional)* **SymPy: sistema**
Resuelve un sistema 2x2 distinto al del ejemplo y muestra la solución.

Respuesta esperada: una solución `{x: ..., y: ...}`.

Glosario (para Colab)

- **if/elif/else**: decisiones por condición.
- **for / while**: bucles para repetición.
- **acumulador**: variable que agrega resultados.
- **índice**: posición en una lista/array (empieza en 0).
- **return**: salida de una función.
- **SymPy**: librería simbólica (resolver/derivar/graficar).

Capítulo 9 — Simulación de Monte Carlo

Abrir este capítulo en Google Colab

Objetivos del capítulo

- Entender **qué significa simular** en Python: repetir un experimento aleatorio muchas veces.
- Programar una simulación para visualizar el **Teorema del Límite Central (TLC)** usando promedios muestrales.
- Programar una simulación de Monte Carlo para estudiar:
  - la frecuencia de **rechazos** en una prueba t cuando H0 es verdadera (error Tipo I),
  - la **cobertura** de intervalos de confianza.

---

1. ¿Qué es una simulación de Monte Carlo (en código)?

En Python, una simulación de Monte Carlo casi siempre se ve así:

1) defines una distribución o un mecanismo aleatorio,

2) repites `r` veces (un bucle),

Glosario

- **Monte Carlo**: repetir muchas veces un experimento aleatorio para aproximar un resultado.
- **replicación**: una corrida/iteración de la simulación (una vez que "repites").
- **semilla**: número que fija el generador aleatorio para reproducibilidad.
- **TLC / CLT**: el promedio muestral tiende a ser normal cuando n crece.
- **IID**: independientes e idénticamente distribuidas.
- **cobertura**: proporción de intervalos que contienen el parámetro verdadero.
- **error Tipo I**: rechazar H0 siendo H0 verdadera.

3) guardas resultados en arrays,

4) resumes con una tabla, proporción o gráfico.

La parte clave para programar bien no es “estadística”, sino:

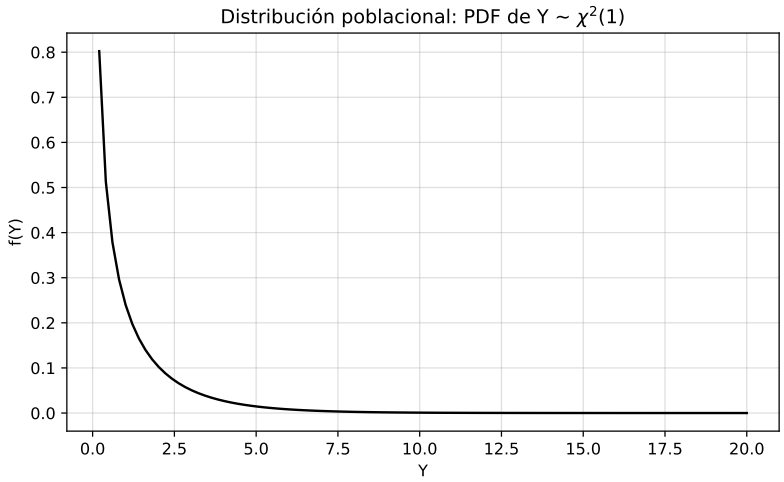
- **organizar resultados** (arrays),
- **reproducibilidad** (semilla),
- **evitar errores de lógica** (índices, condiciones, acumuladores).

---

2. Teorema del Límite Central (TLC) con una población asimétrica

2.1 Población (asimétrica): chi-cuadrado

```
# Ejemplo de distribución poblacional asimétrica (chi-cuadrado)
y = np.linspace(0, 20, num=100)

m = 1  # grados de libertad
fy = stats.chi2.pdf(y, m)

plt.plot(y, fy, linestyle="-", color="black")
plt.title(rf"Distribución poblacional: PDF de Y ~ $\chi^2$({m})")
plt.xlabel("Y")
plt.ylabel("f(Y)")
plt.grid(alpha=0.4)
plt.show()
plt.close()

# Media y varianza poblacionales (para chi-cuadrado con m g.l.)
EYi = m
VYi = 2 * m
print(f"Media poblacional: E(Y_i) = {EYi}\n")
print(f"Varianza poblacional: Var(Y_i) = {VYi}\n")
```




```
Media poblacional: E(Y_i) = 1
```

```
Varianza poblacional: Var(Y_i) = 2
```

Sintaxis clave

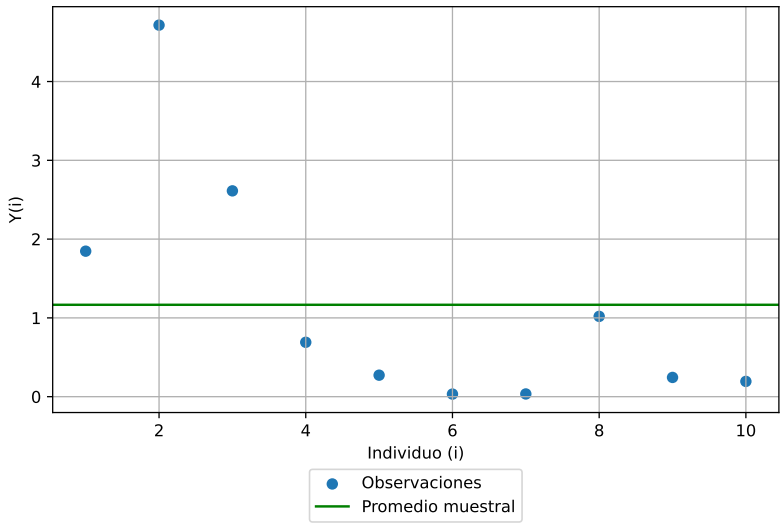
- `np.linspace(a, b, num=k)` crea k puntos entre a y b (útil para graficar).
- `stats.chi2.pdf(y, m)` evalúa la PDF en cada punto de `y`.
- `plt.plot(...)` grafica líneas; `plt.grid()` agrega grilla.

2.2 Una muestra y su promedio

```
# Definimos el valor semilla (None = resultados distintos cada ejecución)
semilla = None

# Tamaño de muestra
n = 10
```

```python
# Muestreo (una muestra desde la población chi-cuadrado)
muestra = stats.chi2.rvs(m, size=n, random_state=semilla)

# Promedio de la muestra
y_promedio = muestra.mean()
print(f"Promedio de la muestra (línea verde): {y_promedio:.4f}")

# Gráfico: observaciones y promedio
plt.scatter(np.arange(1, n + 1), muestra, label="Observaciones")
plt.axhline(y_promedio, color="green", label="Promedio muestral")
plt.ylabel("Y(i)")
plt.xlabel("Individuo (i)")
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.grid()
plt.show()
plt.close()
```

```
Promedio de la muestra (línea verde): 1.1663
```



> **Error común**
>
> Si quieres resultados **reproducibles**, no dejes `semilla = None`.

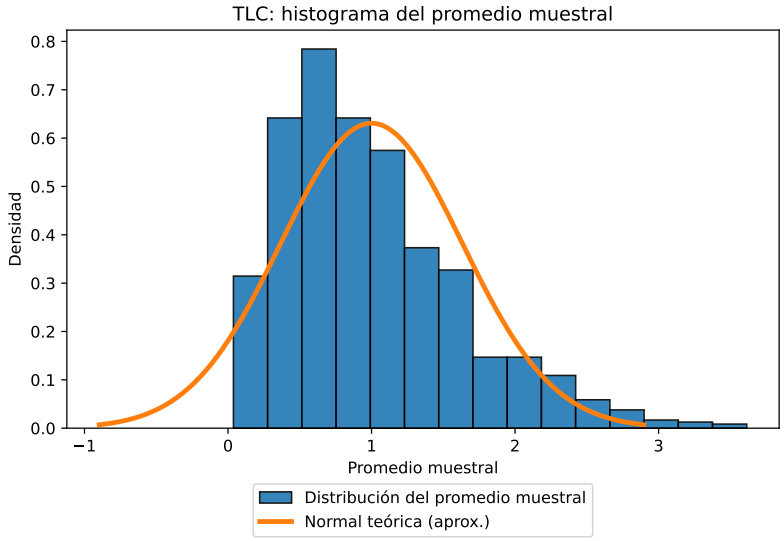
En ese caso, usa un entero (ej. `semilla = 1`).

2.3 Muchas muestras: distribución del promedio muestral

Aquí repetimos el muestreo `r` veces y guardamos los promedios en un array.

```
campana = True          # si True, sobrepone una normal teórica
limites = None          # si quieres un rango fijo, usa limites=(a, b)
semilla = None          # semilla base
n = 5                   # tamaño de muestra
r = 1000                # número de replicaciones

promedios = np.zeros(r)

for j in range(r):
    semilla2 = semilla if semilla is None else semilla + j
    muestra = stats.chi2.rvs(m, size=n, random_state=semilla2)
    promedios[j] = muestra.mean()

# Histograma de promedios
plt.hist(promedios, bins=15, density=True, edgecolor="black", alpha=0.9,
         label="Distribución del promedio muestral")

# Normal teórica aproximada (por TLC): N(E(Y), Var(Y)/n)
if campana:
    EYp = EYi
    VYp = VYi / n
    if limites is None:
        x = np.linspace(EYp - 3*np.sqrt(VYp), EYp + 3*np.sqrt(VYp), num=1000)
    else:
        x = np.linspace(*limites, num=1000)
    pdfx = stats.norm.pdf(x, EYp, np.sqrt(VYp))
    plt.plot(x, pdfx, linewidth=3, label="Normal teórica (aprox.)")

plt.ylabel("Densidad")
```

```
plt.xlabel("Promedio muestral")
plt.title("TLC: histograma del promedio muestral")
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.show()
plt.close()
```



Sintaxis clave (Monte Carlo)

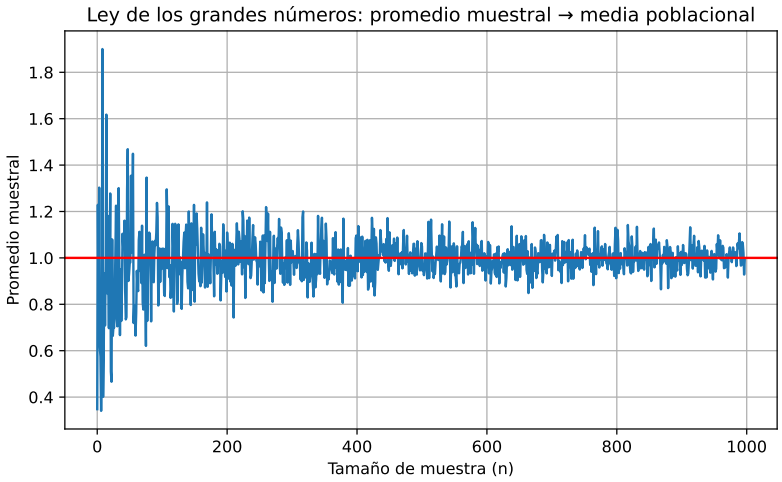
- `np.zeros(r)` reserva un array para guardar `r` resultados.
- Dentro del bucle:
  - generas una muestra aleatoria,
  - calculas su promedio,
  - lo guardas en `promedios[j]`.

Tip: controla semillas por iteración

La línea `semilla2 = semilla + j` sirve para que cada réplica sea distinta, pero reproducible.

2.4 Ley de los grandes números (promedio vs n)

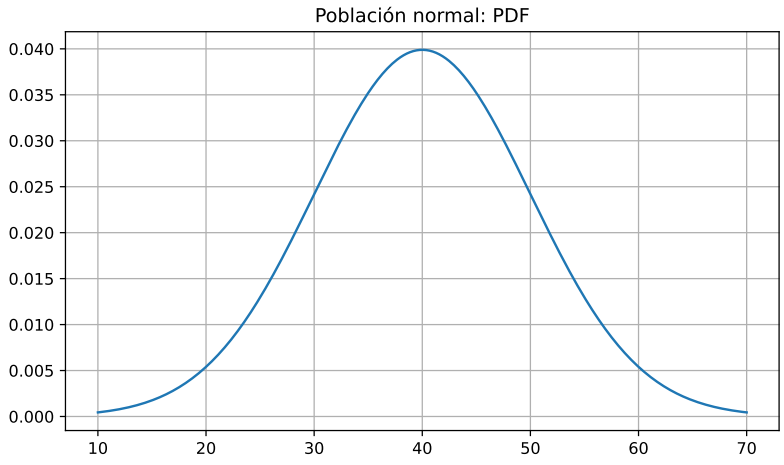
```
semilla = None
nmax = 1000

promedios_nmax = np.zeros(nmax - 2)

for j in range(2, nmax):
    muestra = stats.chi2.rvs(m, size=j, random_state=semilla)
    promedios_nmax[j - 2] = muestra.mean()

plt.plot(promedios_nmax)
plt.axhline(EYi, color="red")
plt.grid()
plt.xlabel("Tamaño de muestra (n)")
plt.ylabel("Promedio muestral")
plt.title("Ley de los grandes números: promedio muestral → media poblacional")
plt.show()
plt.close()
```



3. Simulación de pruebas t e intervalos de confianza

3.1 Población normal (para simular)

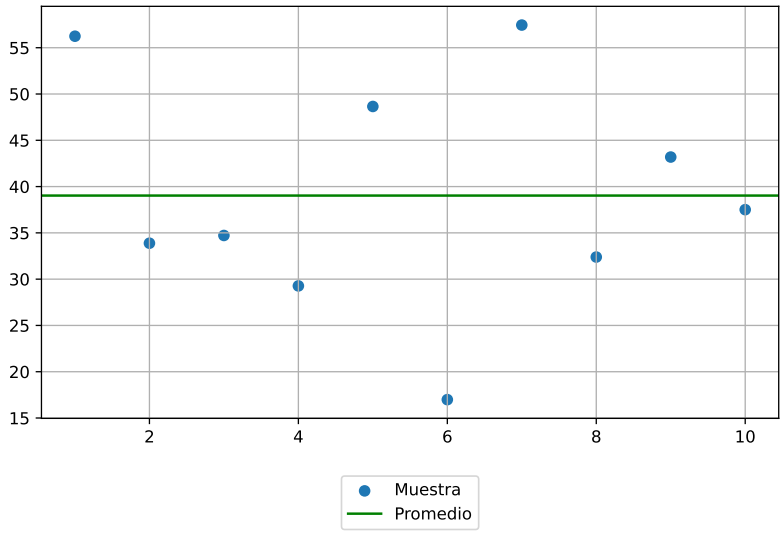
```
mu = 40                  # media poblacional
sigma = np.sqrt(100)     # desviación estándar poblacional

y = np.linspace(10, 70, num=400)
fy = stats.norm.pdf(y, mu, sigma)

plt.plot(y, fy)
plt.grid()
plt.title("Población normal: PDF")
plt.show()
plt.close()
```



3.2 Una muestra: promedio, desviación y SE

```python
semilla = 1
n = 10

muestra = stats.norm.rvs(mu, sigma, size=n, random_state=semilla)

promedio_m = np.mean(muestra)
desv_est_m = np.std(muestra, ddof=1)
error_est = desv_est_m / np.sqrt(n)

print(f"Promedio muestral: {promedio_m:.4f}\n")
print(f"Desviación estándar muestral: {desv_est_m:.4f}\n")
print(f"Error estándar: {error_est:.4f}")

plt.scatter(np.arange(1, n + 1), muestra, label="Muestra")
plt.axhline(promedio_m, color="green", label="Promedio")
plt.grid()
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12))
plt.show()
plt.close()
```

```
Promedio muestral: 39.0286

Desviación estándar muestral: 12.5532

Error estándar: 3.9697
```



> ⚠ Error común: `ddof`
>
> Para inferencia con muestras, usa `np.std(..., ddof=1)` (desviación estándar muestral).

3.3 Prueba t (bilateral) y valor crítico

```
alpha = 0.05
mu0 = mu  # H0 verdadera en este experimento

prueba_t = stats.ttest_1samp(muestra, popmean=mu0)
valor_crit = stats.t.ppf(1 - alpha/2, df=n - 1)

estadistico_t = prueba_t.statistic
valorp = prueba_t.pvalue

print(f"Estadístico t: {estadistico_t:.4f}\n")
print(f"Valor crítico ({(1-alpha)*100:.2f}% confianza): {valor_crit:.2f}\n")
print(f"Valor p: {valorp:.4f}")
```

```
Estadístico t: -0.2447

Valor crítico (95.00% confianza): 2.26

Valor p: 0.8122
```

> **Sintaxis clave**
>
> - `ttest_1samp(muestra, popmean=mu0)` devuelve `.statistic` y `.pvalue`.
> - `t.ppf(1 - alpha/2, df=n-1)` es el cuantil bilateral para el valor crítico.

3.4 Intervalo de confianza e indicador "¿contiene mu?"

```python
LI = promedio_m - valor_crit * error_est
LS = promedio_m + valor_crit * error_est

print(f"IC al {(1-alpha)*100:.2f}%: [{LI:.2f}, {LS:.2f}]\n")
print(f"¿El intervalo contiene a mu? {'Sí' if (mu >= LI and mu <= LS) else 'No'}")
```

```
IC al 95.00%: [30.05, 48.01]

¿El intervalo contiene a mu? Sí
```

3.5 Monte Carlo: repetir r veces (p-values, error tipo I y cobertura)

```
mu = 40
sigma = np.sqrt(100)

semilla = 1
n = 10
alpha = 0.05
mu0 = mu
r = 20

resultados_valores_p = np.zeros(r)
resultados_error1 = np.zeros(r)
resultados_LI = np.zeros(r)
resultados_LS = np.zeros(r)
resultados_intervalo_mal = np.zeros(r)

for i in range(r):
    semilla2 = None if semilla is None else semilla + i

    muestra = stats.norm.rvs(mu, sigma, size=n, random_state=semilla2)

    promedio_m = np.mean(muestra)
    desv_est_m = np.std(muestra, ddof=1)
    error_est = desv_est_m / np.sqrt(n)

    prueba_t = stats.ttest_1samp(muestra, popmean=mu0)
    valor_crit = stats.t.ppf(1 - alpha/2, df=n - 1)

    valorp = prueba_t.pvalue
    resultados_valores_p[i] = valorp

    if valorp < alpha:
        resultados_error1[i] = 1

    LI = promedio_m - valor_crit * error_est
    LS = promedio_m + valor_crit * error_est
    resultados_LI[i] = LI
    resultados_LS[i] = LS

    if (mu < LI) or (mu > LS):
        resultados_intervalo_mal[i] = 1
```

```python
tabla_valores_p = pd.DataFrame({
    "Valores p": resultados_valores_p,
    "Error Tipo 1": resultados_error1,
    "Lim inferior": resultados_LI,
    "Lím superior": resultados_LS,
    "¿No contiene mu?": resultados_intervalo_mal
})

print(f"{tabla_valores_p}\n" if r <= 20 else "Más de 20 simulaciones: omitimos tabla\n")
print(f"Errores tipo I: {sum(resultados_error1)} de {r} ({sum(resultados_error1)/r*100:.2f}%)")
print(f"Intervalos que NO contienen mu: {sum(resultados_intervalo_mal)} de {r} ({sum(resultados
```

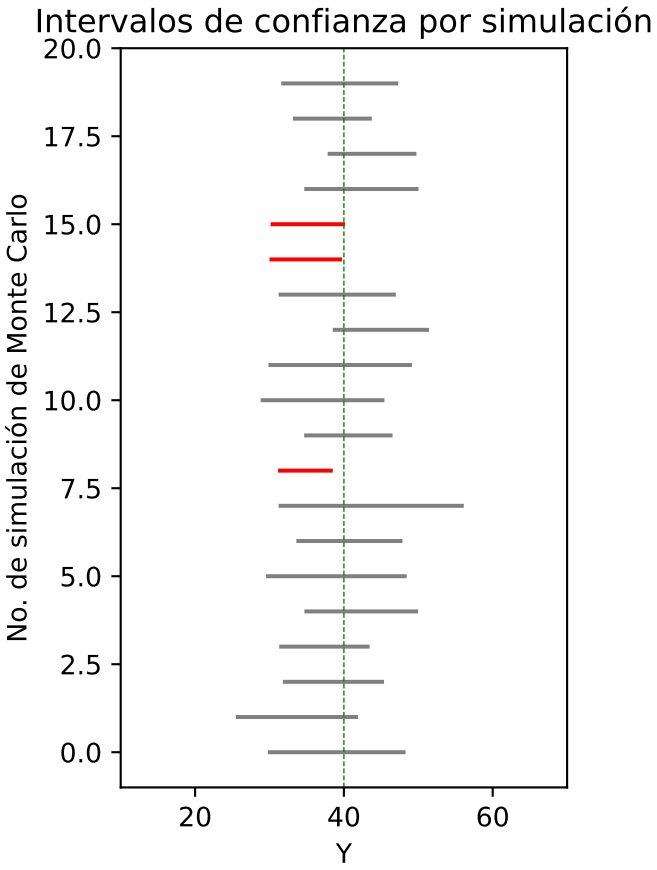
```
    Valores p  Error Tipo 1  Lim inferior  Lím superior  ¿No contiene mu?
0    0.812170           0.0     30.048592     48.008590                0.0
1    0.106021           0.0     25.735345     41.637656                0.0
2    0.638300           0.0     32.061999     45.128436                0.0
3    0.334230           0.0     31.574147     43.187785                0.0
4    0.491500           0.0     34.962626     49.713357                0.0
5    0.808156           0.0     29.800708     48.168869                0.0
6    0.812628           0.0     33.877092     47.604126                0.0
7    0.512978           0.0     31.488820     55.843483                0.0
8    0.007517           1.0     31.413388     38.239187                1.0
9    0.816756           0.0     34.927793     46.268373                0.0
10   0.441022           0.0     29.081732     45.181874                0.0
11   0.906862           0.0     30.128664     48.874181                0.0
12   0.102848           0.0     38.776779     51.164886                0.0
13   0.797112           0.0     31.495786     46.721771                0.0
14   0.033067           1.0     30.271250     39.486101                1.0
15   0.046063           1.0     30.417601     39.894965                1.0
16   0.490989           0.0     34.941927     49.761617                0.0
17   0.167283           0.0     38.086729     49.477681                0.0
18   0.505837           0.0     33.422185     43.492977                0.0
19   0.871146           0.0     31.840531     47.038261                0.0

Errores tipo I: 3.0 de 20 (15.00%)
Intervalos que NO contienen mu: 3.0 de 20 (15.00%)
```

Interpretación en modo “código”

- `sum(resultados_error1)/r` es una **proporción** (frecuencia relativa).
- `resultados_intervalo_mal` marca cuando el IC **falla** en cubrir `mu`.

3.6 Visualizar intervalos de confianza (rojo = no cubre mu)

```
r_ej = 20  # debe ser <= r

plt.figure(figsize=(3, 5))
plt.xlim(10, 70)
plt.ylim(-1, r_ej)

for i in range(r_ej):
    color = "grey" if (mu > resultados_LI[i] and mu < resultados_LS[i]) else "red"
    plt.plot([resultados_LI[i], resultados_LS[i]], [i, i], linestyle="-", color=color)

plt.axvline(mu, linestyle="--", color="green", linewidth=0.5)
plt.ylabel("No. de simulación de Monte Carlo")
plt.xlabel("Y")
plt.title("Intervalos de confianza por simulación")
plt.show()
plt.close()
```



Ejercicios propuestos

1) **Semillas y reproducibilidad**
   Ejecuta dos veces una simulación con `semilla=None` y luego con `semilla=1`.

   Respuesta esperada: con `None` obtienes resultados diferentes; con `1` obtienes resultados idénticos.

2) **TLC: cambia n**
   Repite la simulación de promedios con `n=5`, `n=30` y `n=100`.

Respuesta esperada: el histograma del promedio se vuelve cada vez más parecido a una normal.

3) **Ley de grandes números**
Aumenta `nmax` y mira el gráfico del promedio.

Respuesta esperada: la serie se estabiliza alrededor de `E(Y)=m`.

4) **Error tipo I con más r**
Cambia `r` a 500 o 1000 y calcula la proporción `sum(error1)/r`.

Respuesta esperada: la proporción se acerca a `alpha` (0.05).

5) **Cobertura del IC**
Con `r=1000`, calcula `1 - sum(intervalo_mal)/r`.

Respuesta esperada: la cobertura se acerca a `1-alpha` (0.95).

6) **Exploración**
Cambia `n` de 10 a 30 en la simulación t.

Respuesta esperada: los intervalos se vuelven (en promedio) más estrechos y la cobertura se mantiene cercana a 0.95.

Glosario (para Colab)

- **Monte Carlo**: repetir muchas veces un experimento aleatorio.
- **semilla**: hace reproducible el azar.
- **TLC/CLT**: el promedio muestral tiende a normal.
- **cobertura**: proporción de intervalos que contienen el valor verdadero.
- **error Tipo I**: rechazar H0 siendo verdadera.