Textos docentes | 200

Programación en Python: Estadística a través de problemas resueltos

Manuel José Martínez-Santaolalla Martínez
Isabel María del Águila Cano

edual

Programación en Python: Estadística a través de problemas resueltos

texto:
Manuel José Martínez-Santaolalla Martínez
Isabel María del Águila Cano

Textos Docentes n.º 200

edición:
Editorial Universidad de Almería, 2025
editorial@ual.es
www.ual.es/editorial
Telf/Fax: 950 015459
¤
ISBN: 978-84-1351-388-1

edual editorial UNIVERSIDAD DE ALMERÍA

En este libro puede volver al índice
pulsando el pie de la página

Índice

Prefacio

Después de varios años impartiendo la asignatura "Estadística e Informática" en la Universidad de Almería, en particular encargándonos del bloque correspondiente a Informática, hemos llegado a una conclusión que se ha ido reforzando curso tras curso: la necesidad de disponer de un material didáctico específico que, por una parte, sirva como introducción a la Un modo puede calcularse de foprogramación para estudiantes sin experiencia previa y, por otra, permita abordar la implementación de conceptos estadísticos mediante el lenguaje Python, ya sea de forma nativa o con el apoyo de librerías especializadas.

Este libro nace, por tanto, de una necesidad tanto práctica como docente. Queríamos ofrecer un recurso que integrara dos mundos que muchas veces se presentan por separado: la lógica y estructura de la programación, y la aplicación directa de estos conocimientos a problemas reales de estadística. En un contexto donde la capacidad de analizar datos y automatizar procesos se vuelve cada vez más relevante, aprender a programar ya no es solo una habilidad complementaria, sino una herramienta esencial.

Hemos elegido Python no solo por su creciente popularidad en la comunidad científica y académica, sino también por su sencillez, claridad sintáctica y su enorme ecosistema de librerías. Su curva de aprendizaje moderada lo convierte en un lenguaje ideal para introducirse en la programación, y su potencia lo hace igualmente válido para abordar problemas complejos de análisis de datos y es una herramienta clave en diversas disciplinas.

La estructura del libro refleja esta doble vocación. Los primeros capítulos se centran en una introducción progresiva a la programación estructurada: desde los tipos de datos básicos, pasando por estructuras de control, hasta llegar al diseño modular con funciones. Esta base permite que el lector pueda enfrentarse con confianza a problemas más complejos.

A medida que se avanza, el texto se adentra en la implementación de técnicas estadísticas, comenzando con cálculos simples sin librerías, para luego introducir herramientas especializadas como `itertools`, `statistics`, `numpy`, `pandas` o `matplotlib`. Al final, se presentan casos aplicados en los que se pone en práctica todo lo aprendido, abordando temas como la regresión lineal, la probabilidad, el Teorema Central del Límite, y los intervalos de confianza, entre otros.

Nuestro objetivo ha sido que este libro sea práctico y accesible, tanto para estudiantes universitarios

como para cualquier persona interesada en iniciarse en la programación y la estadística aplicada. Cada concepto va acompañado de ejemplos resueltos y simulaciones que permiten consolidar lo aprendido y, sobre todo, ver cómo la teoría se transforma en soluciones reales mediante código.

Esperamos que estas páginas sirvan no solo como material de estudio, sino también como una experiencia y punto de partida para que el lector descubra el valor de la programación como herramienta esencial para entender mejor el mundo a través de los datos.

Introducción a la Programación

Aunque no es estrictamente necesario para aprender a programar, es útil comprender qué significa realmente la computación o la programación, así como conocer los componentes y el funcionamiento básico de una computadora. Por ello, antes de adentrarse en la construcción de programas informáticos, este libro resume brevemente estos conceptos para luego pasar a los fundamentos propios de la programación de computadoras usando Python.

Conceptos básicos de la informática

La **informática** según la RAE es

> Conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de los ordenadores.

Se puede considerar la unión de dos elementos: INFORmación + autoMÁTICA.

En literatura anglosajona nos encontramos con los términos: Computer Science (ciencia de los computadores), Computer Engineering o IT Engineering. En literatura francófona: Informatique.

La **programación** según la RAE es

> Acción o efecto de programar

Programar según la RAE es:

1. Formar programas, previa declaración de lo que se piensa hacer y anuncio de las partes de que se ha de componer un acto o espectáculo o una serie de ellos.
2. Idear y ordenar las acciones necesarias para realizar un proyecto.
3. *Preparar ciertas máquinas o aparatos para que empiecen a funcionar en el momento y en la forma deseados.*
4. *Elaborar programas para su empleo en computadoras.*

Máquinas programables

Una **máquina** es un cierto dispositivo físico capaz de realizar un cierto trabajo u operación, bien de forma manual (una garrucha) o bien de forma automática (un ascensor).

Este concepto puede extenderse a cuando se consideran máquinas que no existen físicamente, pero en las que puede describirse, concebirse y simular su comportamiento, se denominan **máquinas virtuales**.

En el caso del ascensor su funcionamiento está fijado en el momento de su construcción dependiendo de los dispositivos y conexiones entre ellos.

Existen otras máquinas automáticas (que actúan por si solas) denominadas **máquinas programables**, cuyo comportamiento fijo se completa con un *programa* que permite modificar el comportamiento de la máquina base. Tal como se muestra en la siguiente figura, basta con cambiar el programa para tener una nueva máquina con un comportamiento diferente.

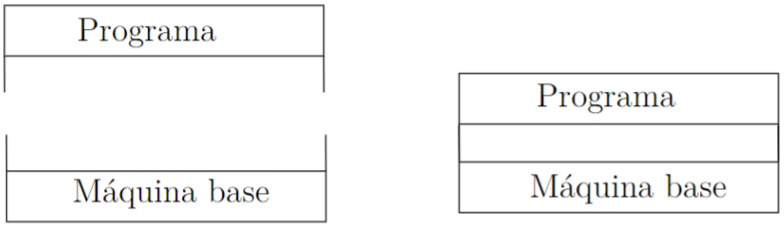

Figura 1: Máquinas programables

Concepto de cómputo

La definición de diccionario de cómputo indica que es el *cálculo destinado a obtener el resultado o valor de algo*.

Por lo tanto, un cómputo implica el procesamiento de información, que tradicionalmente es numérica, aunque puede extenderse a otro tipo de información como texto o sonido.

Cualquier cómputo se puede describir de diferentes maneras. Por ejemplo, el cómputo para **Calcular la nota final** de un alumno que ha realizado tres exámenes a lo largo del curso, pero sabiendo que si en uno de los exámenes tiene un cero, entonces su nota es cero podría hacerse de dos formas:

- Una opción es utilizar la siguiente fórmula

  $Nota \approx \sqrt[3]{nota_1 * nota_2 * nota_3}$

- Pero también se puede describir como un proceso

```
1. Nota = 0;
2. Si nota1 no es cero Nota = Nota + nota1
    - Sino  Nota = 0 e ir a paso 5
3. Si nota2 no es cero Nota = Nota + nota2
    - Sino  Nota = 0 e ir a paso 5
4. Si nota3 no es cero Nota = Nota + nota3
   - Sino  Nota = 0 e ir a paso 5
5. Nota = Nota / 3
```

Concepto de computador

Un **computador** es una máquina programable para el tratamiento de información, es decir realiza cómputos. Posee unos elementos **fijos** o máquina de base llamada **hardware** y otros modificables que son los programas o **software**.

Los computadores actuales son máquinas con programa almacenado de forma que la modificación de los programas no implica la modificación de los elementos físicos, y por tanto, se necesitan sistemas de almacenamiento y de comunicación del ordenador con el entorno.

Un **computador o computadora** es una máquina formada por elementos de tipo **electrónico**, capaces de aceptar unos datos de **entrada**, realizar con ellos gran variedad de tareas (**operaciones**) y proporcionar la información resultante a través de un medio de **salida**, bajo el control de un **programa** previamente **almacenado** en el propio computador. Esto puede incluir tanto dispositivos físicos como servidores, laptops, tablets, entre otros, así como también máquinas virtuales desplegadas en la nube.

Las **máquinas virtuales** en la nube son entornos computacionales virtuales que se ejecutan en infraestructuras de computación remota y están disponibles a través de Internet. Estas máquinas virtuales proporcionan capacidades informáticas similares a las de una computadora física, pero con la flexibilidad y escalabilidad adicionales ofrecidas por la computación en la nube. Por tanto, más que hablar de elementos electrónicos se tiene que hablar de **unidades funcionales**, ya que no tienen porque existir asignadas físicamente a una máquina y que son redimensionables. Se definen o alquilan componentes/unidades lógicas sobre servidores remotos para crear computadoras personalizadas virtuales. Es decir, la máquina base se define de forma virtual.

Un **programa** es la descripción de un cómputo en un lenguaje de programación, que como se ha visto en el ejemplo del cálculo de la nota puede tener muchas formas diferentes, aunque este libro se basa en la definición de programas como un conjunto de **instrucciones**, la llamada **programación estructurada**.

Estructura de una computadora

Los componentes electrónicos de una computadora son cada día más complejos, y muchos de ellos incluso se duplican. Existen numerosas combinaciones de dispositivos/componentes físicos que pueden ensamblarse para crear computadoras. Independientemente de las diferencias físicas, los ordenadores pueden dividirse en varias **unidades lógicas** o secciones, que pueden o no definirse en la nube como los componentes de una máquina virtual. Estas unidades son: Unidades de entrada, de salida, de memoria, de control, aritmético lógica y de memoria secundaria.

Unidad de entrada

Esta unidad obtiene la información (datos y programas informáticos) de los dispositivos de entrada y la pone a disposición de las demás unidades para su procesamiento. Los computadores reciben la mayoría de las entradas de los usuarios a través de teclados, pantallas táctiles, ratones y touchpads, aunque también pueden utilizar otros dispositivos como escáneres, micrófonos, cámaras web, sensores biométricos y lectores de códigos de barras.

Unidad de salida

Esta unidad envía la información que el computador ha procesado a uno o varios dispositivos de salida para que esté disponible fuera del ámbito del computador, y para ser usada directamente por el usuario o bien por otro computador. La mayor parte de la información sale de los ordenadores a través de las pantallas o se reproduce en audio o vídeo sobre teléfonos inteligentes, tabletas, ordenadores y pantallas gigantes en estadios deportivos, o muy habitualmente se transmite por Internet. Otras formas de salida son la vibración de teléfonos inteligentes, mandos de juegos y dispositivos de realidad virtual.

También es habitual que la información se transmita a dispositivos de almacenamiento secundario, como las unidades de estado sólido (SSD), los discos duros, memorias o lápices USB y unidades de DVD. Son unidades donde la información se hace **persistente**, es decir, no se pierde aunque se apague la máquina.

Unidad de memoria

Esta unidad es un almacén de acceso rápido y capacidad relativamente baja que mantiene la información introducida a través de la unidad de entrada, haciéndola disponible cuando sea necesario. La unidad de memoria también guarda la información procesada hasta que puede ser colocada en la unidad de salida.

La información de la unidad de memoria es volátil. Se pierde cuando se apaga el ordenador. La unidad de memoria suele denominarse memoria principal, memoria primaria o RAM (Random Access Memory).

Se divide en posiciones (**palabras de memoria**) de un determinado tamaño. Se accede a cada posición a través de un número (**dirección de memoria**).

Unidad Central aritmético lógica (ALU)

Esta unidad fabrica datos, es decir, realiza cálculos (suma, resta, multiplicación y división) y toma decisiones (por ejemplo, comparar dos elementos de la unidad de memoria para determinar si son iguales). En los sistemas actuales, la ALU forma parte de la siguiente unidad lógica.

Unidad central de proceso

Esta unidad es la que coordina y supervisa el funcionamiento de las demás unidades. La unidad central de proceso (CPU) indica:

- a la unidad de entrada, cuándo leer información y llevarla a la unidad de memoria,
- a la ALU cuándo utilizar la información de la unidad de memoria en los cálculos, y
- a la unidad de salida cuándo enviar datos desde la memoria a los dispositivos de salida,
- cuándo y cómo se accede a la memoria.

La mayoría de los ordenadores actuales tienen procesadores multinúcleo que implementan de forma económica múltiples procesadores en un único chip de circuito integrado. Estos procesadores pueden realizar muchas operaciones simultáneamente. Un procesador de doble núcleo tiene dos CPU, un procesador octa-core tiene ocho. Intel tiene algunos procesadores con hasta 72 núcleos.

Unidad de almacenamiento secundario

Es la unidad encargada de guardar la información de forma permanente, a largo plazo y con gran capacidad. Se considera tanto un dispositivo de entrada como de salida, ya que permite tanto almacenar como recuperar información. Los programas y datos que son utilizados por las demás unidades del sistema se transfieren a dispositivos de almacenamiento secundario (también conocidos como memoria secundaria) cuando no están en uso activo, y pueden permanecer allí durante horas, días, meses o incluso años antes de volver a ser requeridos. Por esta razón, también se le denomina memoria masiva.

La característica principal del almacenamiento secundario es su persistencia: la información no se pierde al apagar el computador, a diferencia de lo que sucede con la memoria primaria. Aunque acceder a esta información toma más tiempo debido a su menor velocidad de lectura y escritura, su gran ventaja es el bajo coste por unidad de almacenamiento.

Entre los dispositivos de almacenamiento secundario más comunes se encuentran las unidades de estado sólido (SSD), las memorias flash USB o lápices de memoria, los discos duros mecánicos (HDD), así como también las unidades ópticas como los discos Blu-Ray que permiten lectura y escritura de datos.

Además, en la actualidad, existen soluciones de almacenamiento secundario virtuales o en la nube, que permiten almacenar y acceder a datos sin necesidad de que estén físicamente en el computador. Estos servicios en línea hacen posible compartir, respaldar y recuperar archivos desde cualquier lugar con conexión a Internet.

Representación de la información

De las definiciones de **dato**, según la RAE, son del contexto de la computación la 1 y la 3, en concreto la tercera solamente es relativa a la informática.

1. m. Información sobre algo concreto que permite su conocimiento exacto o sirve para deducir las consecuencias derivadas de un hecho. A este problema le faltan datos numéricos.
2. m. Documento, testimonio, fundamento.
3. m. *Inform*. Información dispuesta de manera adecuada para su tratamiento por una computadora.

Más específicamente un **dato** en informática es: una representación formalizada de hechos o conceptos susceptible de ser comunicada o procesada.

Existen diversos **tipos de datos**, y por tanto, su representación es diferente:

- Numéricos (12, 28.5): reales o enteros
- Alfabéticos (Ana)
- Alfanuméricos: 23456X, M-6995
- Imágenes, sonido, video.

Se llama **representación de la información** a la forma en la que se almacenan y recogen los datos para poder ser procesados en la computadora. Se debe tener en cuenta que en los medios electrónicos de procesamiento solamente disponen de **dos estados**, interruptor abierto-cerrado.

Esto se traslada a que la información procesada por un computador son siempre unos y ceros. Lo que lleva a la necesidad de **traducir** cualquier dato a una combinación de esos dos símbolos es decir **notación binaria** o sistema de numeración binario.

Nº decimal	0	1	2	3	4	5	6	7
Nº binario	0	1	10	11	100	101	110	111

Observando la tabla se comprueba que, por ejemplo, para representar el "6" en decimal son necesarias tres posiciones en binario. Cada una de estas posiciones es un **bit**, siendo la unidad más pequeña de información un uno o un cero. Cabe decir que en la computación cuántica se amplía este concepto incorporando la idea de que cada bit puede ser cero, uno, o no conocerse su valor.

Veáse un ejemplo: representar en binario todos los posibles estados emocionales de una persona.

Estados = { Tranquilo, Feliz, Emocionado, Preocupado, Triste, Enojado, Sorprendido, Nervioso, Cansado, Motivado }

Son necesarias cuatro cifras binarias. 2^4 es el máximo número de elementos a representar, en este caso, ejemplo hay 10 estados.

Un **byte** es un conjunto de 8 bits. También se llama Octeto o Carácter (porque con un byte se codifica un carácter). Ésta es la unidad de almacenamiento de información en informática (por ejemplo, tamaño de memoria o tamaño de almacenamiento secundario) que siempre se mide en potencias de dos.

Los **qubits** (cúbit) pueden representar simultáneamente 0 y 1. El estado de un qubit puede depender del estado de otro, lo que proporciona una mayor capacidad de procesamiento y almacenamiento de información.

Múltiplos del byte

acrónimo	nombre	equivalencia
KB	Kilobyte	2^{10} bytes $= 1024$ bytes $\approx 10^3$ bytes
MB	Megabyte	2^{20} bytes $= 1048576$ bytes $\approx 10^6$ bytes
GB	Gigabyte	2^{30} bytes $= 1073741824$ bytes $\approx 10^9$ bytes
TB	Terabyte	2^{40} bytes $\approx 10^{12}$ bytes
PB	Petabyte	2^{50} bytes $\approx 10^{15}$ bytes
EB	Exabyte	2^{60} bytes $\approx 10^{18}$ bytes

Ejecución de programas. Lenguajes de Programación.

Desde el punto de vista del software almacenado en la memoria de una computadora se pueden observar distintas capas que definen su comportamiento.

- Software de base o programas del sistema, son programas que gestionan el funcionamiento de la computadora:
  - *Sistema Operativo* - Pertenece a los componentes fijos de un máquina dada. Porque si bien es un elemento lógico, es necesario para que la máquina funcione. Un móvil o un portátil sin sistema operativo no es más que un bonito pisapapeles.
  - *Utilidades para construcción/ejecución de aplicaciones*. Cabe destacar aquí el concepto de **lenguaje máquina**, puesto que cada modelo de computadora física tiene una forma de entender los programas que depende de cuál sea su arquitectura y sus componentes. El lenguaje máquina es el sistema de codificación que tiene cada computadora/procesador para almacenar los programas, que lo define el fabricante de la máquina y se almacena en la propia máquina.
- Software de aplicación son los programas que se cargan sobre la memoria para ejecutar una tarea.

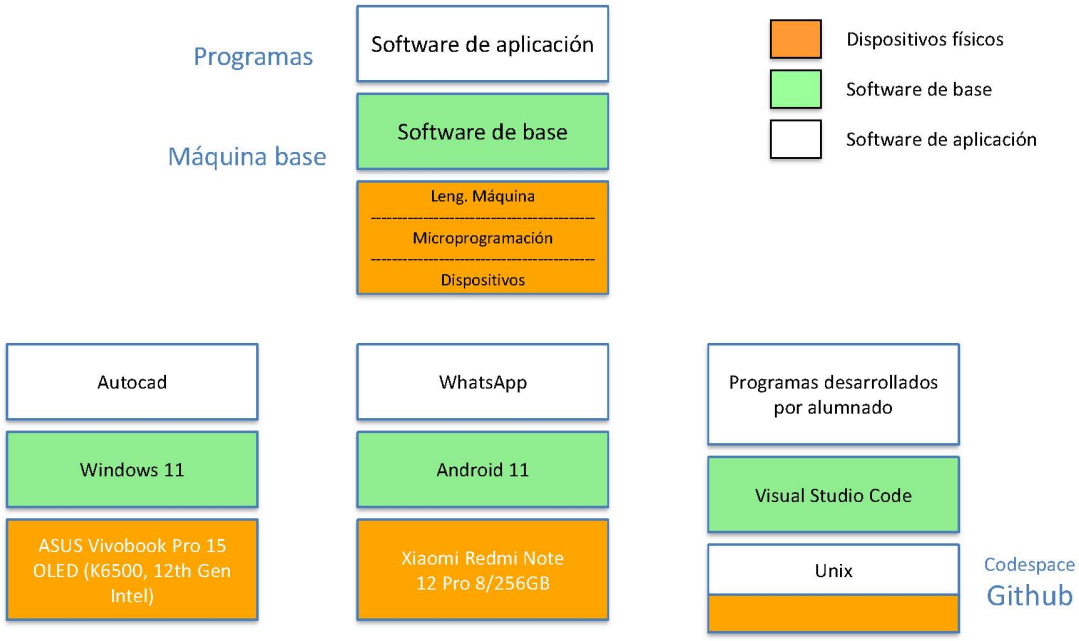

Figura 2: Distintas configuraciones de máquinas base y programas

En general cualquier software es una lista de instrucciones máquina que al ejecutarse producen un resultado concreto. Deben estar expresados en un lenguaje que entienda la máquina, o bien los programadores deben escribirlos en algún lenguaje de más alto nivel (lenguaje de programación) que ha de ser traducido a lenguaje máquina utilizando algún tipo de software de base como son los compiladores o los intérpretes.

Todo programa comienza con idea, algo que se quiere hacer. Generalmente ese algo se plantea como solución a un problema específico. La elaboración de esta solución requiere el diseño de un **algoritmo**.

Los algoritmos son soluciones abstractas a problemas, generalmente son codificados posterioremente en un lenguaje de programación, y luego se traducen al lenguaje máquina que es el único que una computadora puede ejecutar. Con sus resultados, este algoritmo en forma de programa solucionará el problema real para el que se concibió. Los algoritmos son independientes del lenguaje de programación y de la máquina que lo ejecute.

Una analogía de la vida real sería cuando un cocinero con vista cansada quiere hacer una receta. La receta de un plato de cocina (algoritmo) puede expresarse en inglés, francés o español (lenguaje), e indicará la misma preparación independientemente del cocinero (máquina). Después, el cocinero concreto que lea la receta, debe poder entender lo escrito y como el cocinero es corto de vista, necesita sus gafas (compilador) para que le traduzcan los garabatos a letras. Sin gafas el cocinero solamente ve garabatos y necesita que algo se los traduzca (gafas).

Un programa contiene la información codificada del comportamiento deseado o descrito en el algoritmo. Cada modelo de computadora podrá utilizar una forma particular de codificación dependiendo de sus dispositivos físicos, no es lo mismo un móvil que una computadora de sobremesa. La forma de codificar programas de una máquina particular se llama código máquina o lenguaje máquina. Los programas en lenguaje máquina no son legibles, y se suelen llamar **programas objeto** (obj), siendo el resultado de la traducción de un programa escrito en un lenguaje de programación.

Los lenguajes de programación sirven para representar programas de forma simbólica en forma de texto, abstrayéndose del lenguaje máquina que es lo realmente ejecutable. Un **lenguaje de programación** es un idioma artificial diseñado para que sea fácilmente entendible por un humano y traducible por una máquina (empleado una pieza de software de base que lo traduzca). También son llamados lenguajes de alto nivel.

Un programa escrito en un lenguaje de alto nivel está formado por símbolos tomados de un determinado repertorio (componentes **léxicos**). Se construyen siguiendo unas reglas precisas (**sintaxis**). Incorpora unas reglas de **semántica** que determinan el significado de cada construcción.

Tipos de lenguaje de programación

- Lenguajes de marcado: No son considerados lenguajes de programación como tal. Permiten añadir códigos en formato texto (marcas) para indicar cómo se debe mostrar la información. Ejemplos: markdown, latex, HTML.
- Lenguajes de programación. Es un conjunto de reglas y símbolos que permiten a un programador **escribir instrucciones** que una computadora puede entender y ejecutar. Hay muchos tipos y de muchas categorías. Se clasifican según su:
- *Nivel de abstracción* teniendo tanto lenguajes de **bajo nivel**, en los que se programa sobre el hardware (i.e. ensambladores); como lenguajes de **alto nivel**, donde hay una serie de símbolos y reglas que definen instrucciones más complejas.
- *Estilo de programación*: funcional, orientada a objetos, declarativa, guiada por eventos, concurrente e **imperativa**. En esta última es donde hay que definir el detalle de lo que hay que hacer siendo el estilo empleado en este libro sobre Python.
- *Dominio de aplicación*. Existiendo lenguajes especializados en aplicaciones: científico tecnológicas, gestión de la información, inteligencia artificial, programación de sistemas, o aplicaciones para la Web.
- *Ámbito de uso*: Web, móviles, tradicional (equipos fijos y procesamiento local) y/o hardware (Programación de sistemas, diseño de circuitos,...)
- *Forma de traducción*: Compiladores o Intérpretes.

Traductores: Compiladores e intérpretes

Hay diferentes estrategias para poder conseguir/ejecutar el lenguaje máquina a partir de un programa escrito en un lenguaje de programación (también llamado programa fuente). Se utiliza un software que ha de procesar el programa. Estos **procesadores de lenguajes o traductores** manipulan la descripción simbólica de un algoritmo escrito en un lenguaje de programación para obtener el código objeto que ya si es ejecutable en la máquina, como las gafas del cocinero. Existen dos categorías en los traductores:

- Compilador: una vez traducido el programa fuente, su ejecución (programa objeto) es independiente del compilador (se traduce una vez y se ejecuta múltiples veces).
- Intérprete: el programa fuente se traduce instrucción a instrucción cada vez que se ejecuta (no se crea el programa objeto).

Algunos lenguajes de programación

Python

- Se diseñó para ser un lenguaje muy legible, utilizando palabras clave en inglés donde otros lenguajes usan signos de puntuación, facilitando la comprensión y mantenimiento del código.
- Multiparadigma soportando por ejemplo la programación orientada a objetos, la programación imperativa y, en menor medida, la programación funcional.
- Multiplataforma porque es compatible con múltiples sistemas operativos, como Windows, macOS y diversas distribuciones de Linux, permitiendo su ejecución en diferentes entornos sin necesidad de modificaciones en el programa.
- Cuenta con una gran cantidad de librerías y paquetes que extienden sus funcionalidades. Una librería es un conjunto de módulos que ofrecen herramientas reutilizables para diversas tareas, como matemáticas, manipulación de datos, desarrollo web, inteligencia artificial, entre otras. Ejemplos destacados son *Numpy* para cálculo numérico y *Pandas* para análisis de datos.
- Interpretado: se requiere tener Python disponible para ejecutar las instrucciones.

C

- Utilizado para programación de sistemas.
- Gestión de memoria - Soporta la característica de asignación dinámica de memoria.
- Muy eficiente, de hecho la librería Numpy de Python está escrita en C.
- Estándar ISO/IEC 9899:2018 (C18)
- Familia de lenguajes C: C++ y C#
- Compilado. desde el archivo fuente`.c` se genera un ejecutable `.exe`. Aunque suele ser trasparente desde los entornos de construcción de programas, la generación del ejecutable tiene dos fases:

- Compilar: Crea los objetos (`.o` o `.obj`).
- Enlazar o Linkar: Une los objetos para crear ejecutables. Busca librerías y las añade si es necesario.

Java

- Lenguaje portable, basado en C.
- Utilizado para aplicaciones en internet.
- A la vez compilado e interpretado. Con el compilador se traducen archivos fuente `.java`, a un conjunto de instrucciones llamados bytecodes, en un archivo `.class` que son independientes del tipo de ordenador. El intérprete ejecuta estas instrucciones en un ordenador específico (máquina virtual java).

Hola Mundo

Un programa es por tanto un conjunto de **"texto"** o **bloque de código** escrito con unas normas de sintaxis marcadas por el lenguaje de programación que busca hacer algo y que se guarda en un archivo plano (`.c`, `.py`, `.java`). Sea el siguiente ejemplo:

Problema: Escribir un mensaje de saludo al usuario en la pantalla

Solución: Cada lenguaje tiene una forma de expresarlo. Nótese que cada lenguaje tiene su forma de organizar los bloques de código e incorporar comentarios. Por ejemplo, en C se utiliza `/* y */`, en Java se utiliza `//` para los comentarios y en el caso de Python se usa `#`.

Solución en ensamblador

```
section .data
    msg db "Hola Mundo!", 0ah
section .text
    global _start
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, 13
    syscall
    mov rax, 60
    mov rdi, 0
    syscall
```

Solución en C

```c
/* Programa que muestra un mensaje en pantalla */
#include <stdio.h>

int main() {
   printf("Hola Mundo!\n");
   return 0;
}
```

Solución en Java

```java
// Programa que muestra un mensaje en pantalla
import java.util.*;

public class Main {
    public static void main(String[] args) throws Exception {
      System.out.println("Hola Mundo!");
    }
}
```

Solución en Python

```python
# Programa que muestra un mensaje en pantalla
print('Hola Mundo!')
```

Pero ¿cómo se **ejecuta** este código? Tal como ya se ha dicho, algunos programas tienen que ser compilados y otros interpretados por un traductor del lenguaje concreto. En en el caso de C, se debe compilar el archivo que contiene el código para obtener un archivo ejecutable que se podrá usar sin que el compilador de C esté instalado en la misma máquina. Sin embargo, en Python, bien sea teniendo el código en un archivo `.py` o bien escribiendo directamente las instrucciones, para poder "saludar" es imprescindible tener un intérprete Python disponible en la máquina donde se ejecute.

El código Python (siempre con el intérprete presente) se ejecuta de diversas formas. Tres de los modos básicos de uso de Python son el modo interactivo, el modo script y los cuadernos.

En el **modo interactivo**, introduciendo directamente fragmentos de código Python viendo sus resultados inmediatamente. El símbolo `>>>` indica que el intérprete está cargado.

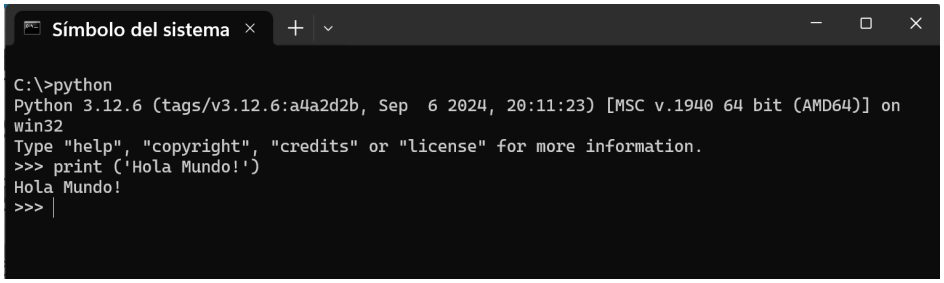

Figura 3: Ejecución en modo interactivo

En el modo **script**, se ejecuta el código cargando desde un archivo con la extensión `.py` que lo contiene, son los llamados scripts, que podrían traducirse como secuencia de instrucciones, aunque no suele hacerse y el nombre en inglés es el más utilizado. El código, en modo texto, se coloca en un archivo llamándolo por ejemplo `saludo.py`. Este archivo se puede lanzar bien desde entornos de programación como **Visual Studio Code**, **Anaconda** o **WinPython** o directamente desde la consola llamando al intérprete, `python saludo.py`.

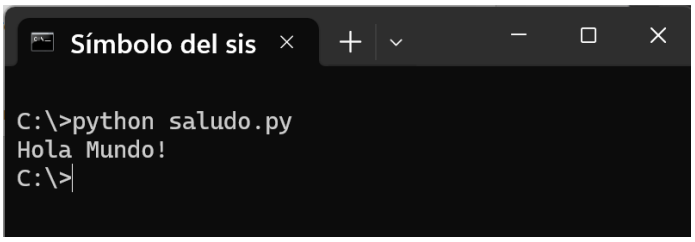

Figura 4: Ejecución en modo script sobre consola

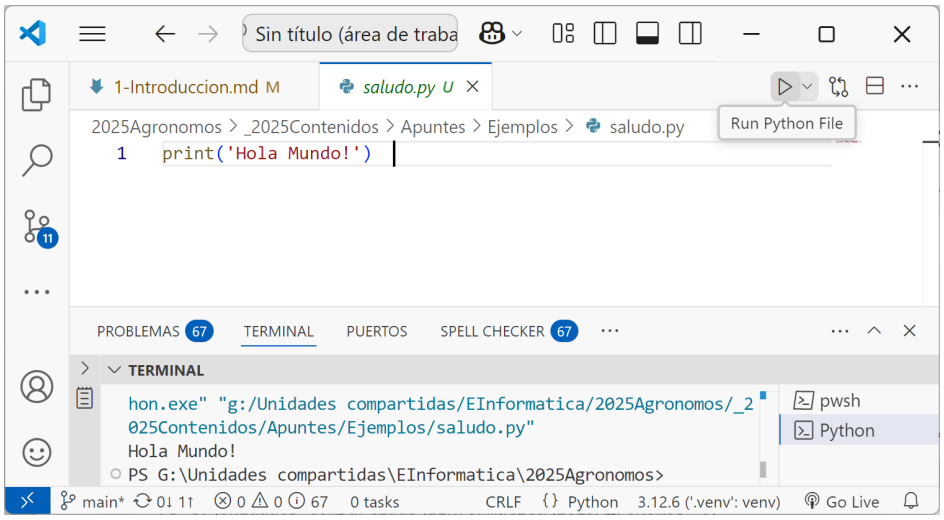

Figura 5: Ejecución en modo script desde Visual Studio Code

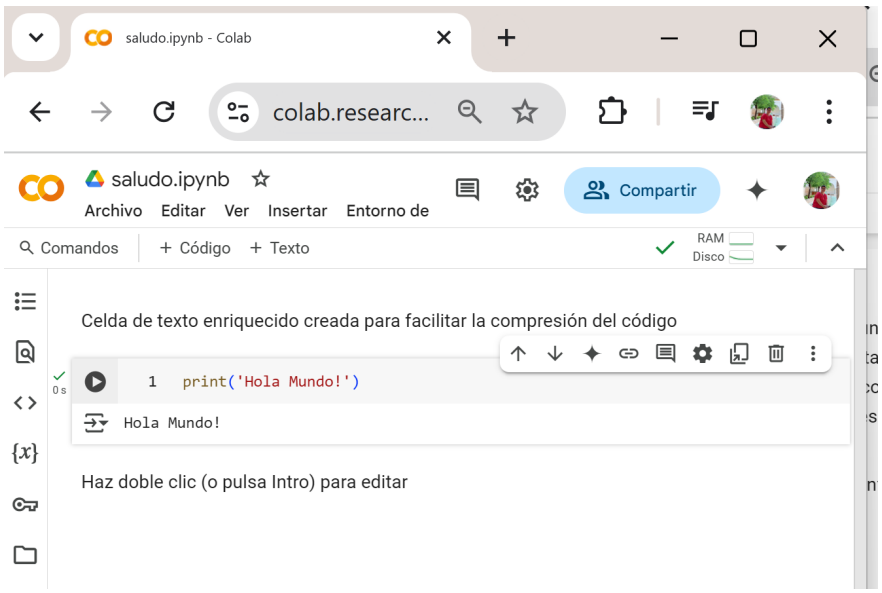

Figura 6: Cuaderno jupyter sobre Colab

Finalmente, sobre todo para soluciones finalistas, se pueden ejecutar fragmentos (chunks o snippets) de código Python sobre **cuadernos** Jupyter (Jupyter Notebook), en especial sobre **Colab de Google**, que es un servicio de alojamiento de cuadernos que no requiere configuración y que ofrece acceso sin coste a recursos de computación. Los cuadernos permiten combinar elementos de texto enriquecido (formato, tablas, figuras, ecuaciones, enlaces, etc.), código y el resultado de ejecutar el código en un único documento.

Instrucciones básicas

Un programa escrito en un lenguaje de alto nivel son varias **instrucciones** (órdenes) agrupadas, los grupos son llamados **bloques de código** y **bloques de código con nombre** (similar a una función matemática). Las instrucciones son las unidades elementales de un bloque y se utilizan para especificar las tareas que el programa debe llevar a cabo utilizando los **datos** guardados en la memoria. Los programas simples suelen tener un único bloque de código.

En el ejemplo *"Hola Mundo"* en C hay un bloque de código llamado `main`, mientras que en el ensamblador hay tres secciones identificadas por un tabulador y un nombre, mientras que en Python solamente se tiene una línea de código.

Cada instrucción en un lenguaje de programación realiza una tarea particular, que se categorizan en tres grupos:

- Asignación
- Entrada / Salida
- De control

Las **instrucciones** necesitan información para trabajar. Por ejemplo, si se quieren sumar dos números, esos números son los **datos** necesarios para ejecutar la instrucción `+`. De igual forma, en una instrucción `print` usada para imprimir en pantalla un texto, la cadena incluida entre paréntesis es el **dato** necesario para ejecutar la instrucción (por ejemplo, "Hola").

Instrucción	Datos	Como se escribe (sintaxis)	Resultado (semántica)
print	"Hola"	`print("Hola")`	Se muestra mensaje
+	5 y 6	`5+6`	Se suman 5 y 6
print, +	5 y 6	`print(5+6)`	Se suman 5 y 6, y se muestra el resultado

En C se usa `printf`, en Pyhton `print` y en java `System.print.out` para mostrar un mensaje en pantalla. Cada **lenguaje** de programación tiene su propio **léxico** (conjunto de palabras clave y símbolos) y sus propias reglas de combinación, es decir, su **sintaxis**. Por ejemplo, la cadena a mostrar con `print` debe ir entre paréntesis tras la instrucción. Gracias a estas reglas, es posible

trasladar el algoritmo a un programa fuente en un **lenguaje** para dar las órdenes a ejecutar una vez traducido el fuente sobre la máquina.

Realmente un lenguaje de programación es un **idioma** acotado a las cosas que el programador puede decir para que las entienda un computador y que vienen heredadas del inglés, algunas son:

- print input
- if elif else
- while for break continue
- True False None and
- import def

Para Python todas las palabras clave del lenguaje pueden verse si se ejecuta en el intérprete las líneas:

```
import keyword
print(keyword.kwlist)
```

Existen treinta y cinco palabras reservadas en Python.

```
False, None, True, and, as, assert, async, await, break, class, continue, def, del, elif, else,
except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass,
raise, return, try, while, with, yield
```

Identificadores y variables

Todo programa utiliza **datos** para poder realizar la tarea que tenga asignada, bien sea almacenándolos, modificándolos o mostrándolos. Los datos pueden ser números, caracteres u otros tipos que se verán después. Atendiendo a su uso estos datos se clasifican en:

- **Constante literal**: cualquier valor constante escrito directamente en un programa. Si se escribe `x + 3`, el número 3 es una constante literal; `Hola Mundo` también es una constante de cadena.
- **Constante simbólica (con nombre)**: valor de dato constante que se referencia mediante un nombre (identificador). Pero en Python no se declaran constantes como tales, realmente son variables. Por convenio se suelen poner en mayúsculas pero pueden ser reasignadas, por ejemplo: `PI = 3.1416`.
- **Variable**: zona de memoria central que se referencia mediante un nombre o identificador, en lugar de conocer la dirección física donde está en memoria, para usarla se usa el nombre que se le ha asignado. En una variable se almacena el valor de un dato que puede cambiar durante la ejecución del programa.

Cada lenguaje maneja las variables y las constates de forma diferente. En C/C++ se usa tipado estático, es decir, el tipo de cada variable debe declarase antes de usarlo (sino se produce un error) y los nombres hacen referencia permanente a localizaciones o bloques de memoria, no

cambian durante el ámbito de uso de la variable. Cada uno de estos bloques tiene una capacidad de almacenar información que depende de la representación interna del tipo de dato de la variable. Python es un lenguaje de tipado dinámico, donde no hay que declarar el tipo de las variables antes de usarlas, según el uso se le asigna un tipo, que además puede cambiar en la ejecución.

Una variable se puede ver como una *caja* dentro de la memoria, donde se almacena el valor de un dato, y que es conocida en el programa mediante el nombre que se ha dado a la caja (o **identificador**).

Las *cajas* de las variables pueden ser de distintos tamaños, en función del tipo de dato que se pueda almacenar en ellas. Así, si la variable va a contener valores de tipo carácter, su tamaño será de un byte. Si va a contener valores de tipo entero `int`, su tamaño será de dos bytes. Si va a contener una cadena de varios caracteres, necesitará ser de un tamaño igual al número de caracteres más uno (porque las cadenas añaden al final un carácter especial *fin de cadena*).

Hay lenguajes, como C, en los que las variables han de **declararse** indicando el tipo de dato de los valores que van a contener. La razón es que hay que reservar los espacios en memoria de todas las variables antes de comenzar la ejecución del programa, en Python no es necesario puesto que esta reserva se hace de forma dinámica.

Las variables en Python simplemente se definen al ser utilizadas por primera vez. Además, (si bien no es recomendable) pueden cambiar de tipo si se realiza una asignación con otro tipo:

```
x = "hola"
print(x)
```

Si cambia su asignación cambia su tipo,

```
x = 5
print(x)
```

o bien,

```
y = x + 2.5
print(y)
```

Todos los lenguajes estructurados se organizan en torno al concepto de **bloque de código** y **bloques de código con nombre**. Los script simples de Python no tienen por qué tener nombre, sin embargo, es importante tener en cuenta que por ejemplo, en C los bloques vienen definidos por el uso de delimitadores `{` para indicar el inicio del bloque y `}` para indicar el fin. En el caso de Python se usan las **indentanciones**, **saltos de línea** y **`:`** para marcar los bloques de código. Es decir, los **tabuladores** y los saltos de línea tienen significado. Otro elemento estructurador del código son los **paréntesis** `( )`, que indican cuales son los **datos** necesarios para ejecutar un bloque/función de código, a estos datos se les llama argumentos.

Un **identificador** es el nombre utilizado para representar variables, bloques de código y otros elementos dentro de un programa fuente. Sirve como una etiqueta que permite referenciar estos elementos de manera única dentro de un programa. Los identificadores deben seguir ciertas reglas establecidas por cada lenguaje, como comenzar con una letra o un guion bajo y no usar palabras reservadas. Elegir nombres descriptivos y claros para los identificadores es fundamental para mejorar la legibilidad y mantenibilidad del código.

Para Python las normas a cumplir para construir los identificadores son:

- Los nombres deben empezar por _ o por una **letra**
- El resto puede ser letras, números o _, sin espacios
- Se distingue entre mayúsculas y minúsculas.
- no se puede usar las palabras reservadas del lenguaje

```
# Válido
_variable = 10
vari_able = 20
variable10 = 30
variable = 60
variaBle = 10
Variable = 20
CONSTANTE = 0

# No válido

2variable = 10
var-iable = 10
var iable = 10
```

Primer ejemplo en Python

Python se desarrolló originalmente como lenguaje didáctico, pero su facilidad de uso y su sintaxis limpia han hecho que lo adopten tanto principiantes como expertos. Es mucho más fácil leer y entender un script en Python que leer un script similar escrito en otro lenguaje.

Algo que distingue a Python de otros lenguajes es que es **interpretado**, no compilado. Esto significa que se ejecuta línea a línea, lo que permite que la programación sea interactiva de una forma que no es directamente posible con lenguajes compilados.

Cuando se juntan varias instrucciones para ser ejecutados como una unidad se llama **script**, que puede o no estar en un archivo `.py`. Un script simple, pero que ilustra varios de los aspectos importantes de la sintaxis de Python es:

```
print ("Hola ")
print ("¿cómo estás?")
```

Se le indica al intérprete que utilice los datos `"Hola"` para ejecutar la funcionalidad definida en `print`, estos datos que necesita `print` se llaman argumentos y se colocan entre paréntesis. A esto se le llama **pasar** argumentos a `print`.

Si bien este bloque de dos instrucciones es directamente ejecutable en el intérprete Python, si se quiere poder utilizar este *trabajo* de forma repetida debe asignársele un identificador para usarlo en otras instrucciones. Suele guardase en un archivo `.py`. Cuando se quiere dar nombre a un bloque se utilizará la palabra clave `def` (definir), seguida por un identificador válido:

```
# código que permite mostrar un mensaje en pantalla
def saludo():
    print ("Hola", end=" ")
    print ("como estás?")
```

Los comentarios se marcan con

Aunque los comentarios no se ejecutan, por estilo y mantenibilidad, los scripts deben de comenzar con un comentario que indique su objetivo.

```
# código que permite mostrar un mensaje en pantalla
```

Los comentarios en Python se indican con la almohadilla (`#`). Desde la posición de la almohadilla hasta el final de línea el contenido es ignorado por el intérprete.

```
print ("Hola", end=" ")  # sustituye salto de linea final por un espacio
```

También hay comentarios multilínea, que se marcan con triples comillas bien sean simples ”’ o dobles “““, que además de documentar los bloques de código se usan en las utilidades de ayuda:

```
'''  código que permite mostrar
     un mensaje en pantalla
'''
```

El final de línea es el fin de instrucción y la indentación cuenta

Los saltos de línea marcan el cambio de instrucción.

```
print ("Hola ")
print ("¿cómo estas?")
```

El caso siguiente produce error:

```
print ("Hola ")
   print("¿cómo estás?")
```

También es erróneo:

```
print ("Hola ") print("como estás?")
```

Para que una instrucción continúe en la línea siguiente hay que utilizar el símbolo `\`. Solamente se permite la continuación implícita de líneas dentro de paréntesis `()`, corchetes `[]` y llaves `{}` para mejorar la legibilidad:

```
print("¿cómo \
    estás?")
```

Si bien es posible poner más de una instrucción en la misma línea, las normas de estilo no lo recomiendan y debe evitarse:

```
print ("Hola "); print("como estás?")   # No recomendable
```

Los bloques de código se marcan con *indentación*, también es un error:

```
def saludo():
print ("Hola", end=" ")
print ("como estas?")
```

Las dos instrucciones `print` están fuera de `saludo`, debe estar un tabulador indentado para que sean parte del bloque de código con nombre.

Ejecución del script

Con `def` se define un bloque de código con nombre o función pero no se ejecuta, solo se define. Para ejecutarla se debe **llamar** al bloque. De esta forma se separa la definición de la ejecución:

```
def saludo():
    print ("Hola", end=" ")
    print ("¿cómo estas?")
saludo()
```

Si se pone:

```
saludo()
saludo()
```

El resultado la ejecución será,

```
Hola ¿cómo estas?
Hola ¿cómo estas?
```

Para scripts con un solo bloque no es necesario usar `def`, es decir, no se le asigna un nombre. Su nombre es el del archivo `.py` donde esté.

La decisión de si definir un bloque de código con nombre o no, es decir usar `def`, dependerá del uso futuro de ese bloque. Por tanto, si se va a reutilizar desde otros scripts se le dará nombre, para poder lanzar su ejecución.

Tipos de datos simples

En un lenguaje de programación, la información (datos) con la que trabaja el programa se organizan en distintos tipos de datos. Además de los números enteros y decimales, existen otros tipos como las cadenas de texto, las listas y las tuplas. Cada lenguaje tiene su propia manera de representar y manipular estos datos de un determinado tipo, definiendo qué operaciones pueden realizarse con cada uno. Sin embargo, no todos los lenguajes ofrecen los mismos tipos de datos ni las mismas reglas para operarlos.

Mientras que un **dato** es el elemento de información concreto, que es objeto de operación por parte de la computadora y que puede ser simple o compuesto (formado por otros datos), un **tipo de dato** es la propiedad asociada a la representación del dato en la computadora. A continuación, se revisará como Python implanta y define estos tipos.

Cada tipo se caracteriza por la **representación interna** del tipo, que define el **rango** de valores permitidos, y por las **operaciones habilitadas** para esos tipos. Por ejemplo, dado el conjunto de **enteros** $\{\ldots$,-3,-2,-1,0,1,2,3,$\ldots\}$, cada valor del tipo se representará en binario en memoria, por tanto, la física de las celdas de memoria limita el tamaño máximo y mínimo del entero que puede manejarse para el tipo entero.

En Python, las limitaciones de la representación interna de los datos son menores que en otros lenguajes puesto que todos los tipos de datos son dinámicos, como se verá más tarde. En el caso del C estándar un entero debe estar en el rango -32767 a 32767. Las operaciones permitidas para esos valores **enteros** son: suma $(+)$, resta (-), división $(/)$, división entera, multiplicación, resto de la división entera o módulo, exponenciación y cambio de signo.

Se definen tres grandes tipos de datos: datos simples, compuestos y definidos por el usuario.

Datos simples: (indivisible o atómico): Son aquellos que no están compuestos de otros datos y se distingue entre estándar y no estándar.

- **Datos simples estándar**: Se llaman también tipos de datos **primitivos** o **predefinidos**, y forman parte de la sintaxis de la mayoría de los lenguajes de programación.
    - Numéricos:
        - Enteros – el número de tiradas de un dado.
        - Reales – la media de las notas de un alumno.
    - Lógico o booleano – Si se es o no familia numerosa.
    - Carácter – La letra del DNI.
    - Punteros, no es un dato en si mismo, es una dirección de memoria (un índice que referencia una celda física de la memoria). No todos los lenguajes permiten manejar las direcciones de memoria, curiosamente en Python *todo son punteros*, aunque sea trasparente para el usuario principiante.

- **Datos simples no estándar**: están definidos por el programador mediante especificación de los valores de su dominio (por **enumeración**) o mediante un subconjunto de un tipo (dando un **subrango**). Por ejemplo, los días de la semana o los estados git de un archivo (untracked, unstaged, staged, committed, synced, diverged, stashed), no están presentes en todos los lenguajes de programación aunque se suelen emular con enteros.

Datos Compuestos o estructurados o colecciones: Es un tipo de dato que consta de varios datos de tipos primitivos o compuestos, por eso también pueden ser llamados **colecciones**. Es un constructor genérico de tipos de datos que el programador ha de completar con significado, como por ejemplo, la colección de enteros que recoge las notas de un alumno en las actividades realizadas en un curso.

Una colección representa múltiples datos individuales, donde cada dato individual se puede referenciar de forma independiente a los demás. Las operaciones permitidas están relacionadas con almacenar y recuperar componentes individuales. Las colecciones clásicas en programación son: **Texto** (cadena de caracteres), **Colecciones indexadas** que son una colección de elementos del mismo tipo (habitualmente llamados vectores o arrays), y **Colecciones estructuradas** o registros que son un conjunto de datos de tipos distintos y habitualmente de tamaño fijo.

En Python estos tipos compuestos han evolucionado hacia constructores aun más genéricos (de hecho las listas son la generalización de colecciones estructuradas e indexadas) y se tienen unos tipos compuestos extendidos que están predefinidos sobre el propio lenguaje: **Textos**, **Listas** Mutables e inmutables (**Tuplas**), **Conjuntos** y **Diccionarios**, prescindiendo de los clásicos vectores (arrays) y registros.

Definidos por el usuario: Permiten crear alias para tipos de datos y ajustar las funciones y rango permitido. En C se utiliza `typedef` para indicar que se está definiendo un tipo de dato, permitiendo dar más contenido semántico a las colecciones. En Python aparece el concepto de **clase**, `class`, que no se explora en este libro pues implica niveles más avanzados en programación.

Python tiene los mismos datos primitivos que la mayoría de los lenguajes y algún tipo adicional que mejora la expresividad en los scripts. Los tipos de datos simples en Python se muestran en la siguiente tabla:

Tipo	Ejemplo	Descripción
`int`	`x = 1`	Enteros (i.e., todos los números)
`float`	`x = 1.0`	Números en punto flotante (i.e., números reales)
`complex`	`x = 1 + 2j`	Números complejos (i.e., con parte real y parte imaginaria)
`bool`	`x = True`	Booleanos: True/False
`char`	`x = 'a'`	Carácter individual (en Python, es una cadena de longitud 1)
`str`	`x = 'abc'`	Cadenas de caracteres
`NoneType`	`x = None`	Objeto especial que indica nulo

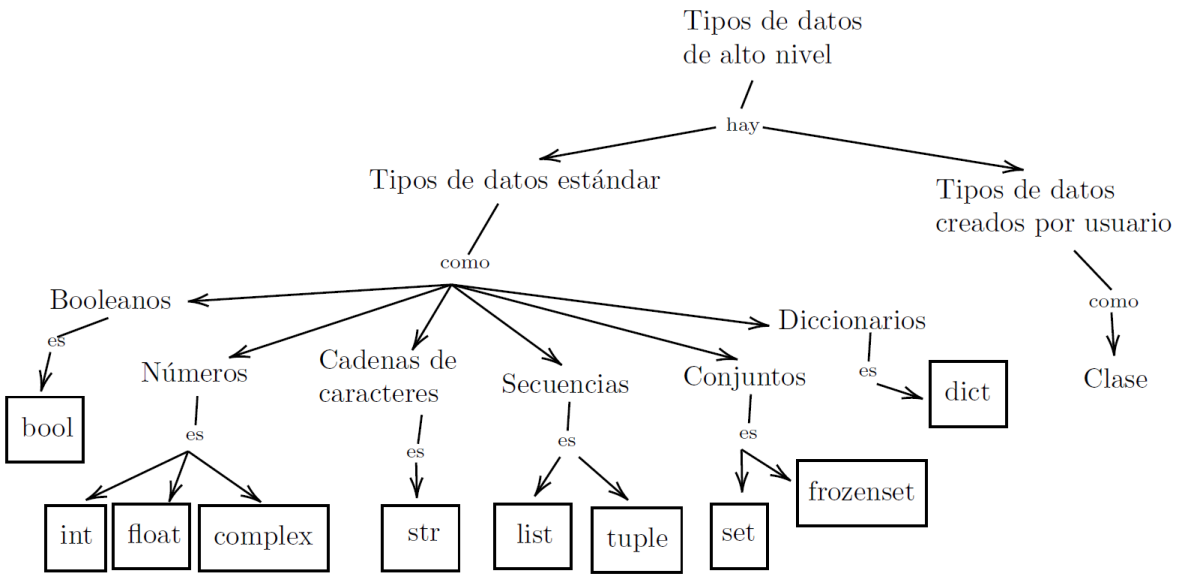

Figura 7: Tipos de datos en Python

Tipo entero

La cantidad y el rango de los números enteros y reales en matemáticas es un conjunto infinito. Una computadora, por contra, dispone de un número finito de posiciones de memoria en las que almacenar bits para permitir la representación **exacta**.

En lenguajes como C/C++, de forma nativa, hay un tamaño máximo de un valor entero. En Python el tipo entero (**int**), no tiene un límite predeterminado para el máximo valor absoluto. Los enteros de C son de precisión fija, y normalmente se desbordan en algún valor (a menudo cerca de 2^{31} o 2^{63}, dependiendo de tu sistema). El tipo `int` en Python es de precisión variable, por lo que se pueden hacer cálculos que se desbordarían en otros lenguajes como:

```
2 ** 200
```

Otra característica conveniente de los enteros de Python es que, por defecto, la división se convierte al tipo de punto flotante, es decir, un real. Incluso se pueden especificar en diferentes bases. Cualquier número sin punto decimal es un entero:

```
c = 0x7fa8      # Hexadecimal
d = 0o253       # Octal
e = 0b10001111  # Binario
```

Tipo real

Para los números reales, se utiliza una representación muy diferente a la de los enteros, denominada de punto (o coma) flotante que está definida en el estándar IEEE 754. La representación de números reales en las computadoras es limitada: no es posible expresar números infinitamente grandes, la precisión es finita, por lo que solo una cantidad limitada de números reales puede representarse exactamente.

Para los tipos `float` se pueden utilizar la notación decimal o exponencial. El caso de `1.4e6` se interpreta como $1{,}4 \times 10^6$.

Un real almacena entre 15 y 17 dígitos con un exponente entre -308 to 308,

```
x = 0.000005
y = 5e-6
# Estos dos números son iguales
```

Un entero puede convertirse explícitamente en un flotante con el constructor `float`:

```
a = 1         # entero
b = float(a) # b es ahora 1.0, tipo float

b = 1.8       # flotante
c = a + b     # c es 2.8, tipo float

a = 1.8       # flotante
b = int(a)    # b es 1, tipo int

a = 1         # entero
b = 2.0       # flotante
c = a + b     # c es 3.0, tipo float
```

Tipo lógico

Recoge la verdad o no de una situación o hecho. Los valores booleanos o lógicos se llaman así en honor al lógico inglés George Boole. Pueden tomar dos valores: `True` y `False` (verdadero, 1, <> 0 y falso, 0). Las operaciones asociadas a estos tipos de datos son los operadores lógicos(`and`, `or`, `not`) y relacionales (mayor, menor, igual, distinto), cuyo funcionamiento se describirá tarde.

Tipo carácter

Los computadores son conocidos por su capacidad para trabajar con valores numéricos. Sin embargo, el manejo de textos es una tarea recurrente (procesadores de texto, sistemas de mensajería, etc.). En todos estos casos, las cadenas de caracteres juegan un papel fundamental.

Si bien se puede trabajar con caracteres individuales, en la práctica, el manejo de textos se realiza principalmente mediante cadenas de caracteres. Éstas constituyen un tipo de dato compuesto en el que cada elemento es un carácter, como se verá más adelante.

En Python, los caracteres se almacenan como valores numéricos según su representación en la tabla ASCII o Unicode, dependiendo del sistema utilizado. Cada carácter se asocia a un código numérico que permite su almacenamiento y manipulación dentro de los programas.

Los **códigos de escape** (escape codes) son expresiones que comienzan con una barra invertida, `\` y se usan para representar caracteres especiales que no pueden ser fácilmente tecleados o que tienen un significado especial dentro de una cadena de texto. Estos códigos permiten, por ejemplo, insertar saltos de línea, tabulaciones o caracteres de comillas sin interferir con la sintaxis del lenguaje.

```
'\n'      Avanzar una línea
'\r'      Retorno de carro
'\t'      Tabulador
'\''      Comilla literal
'\"'      Comilla doble literal
'\\'      Barra invertida literal
```

Estos códigos facilitan la manipulación de textos en Python, permitiendo la inclusión de caracteres especiales sin necesidad de escribirlos directamente.

Tipo complejo

Los números complejos (`complex`) son números con dos elementos, la parte real y la parte imaginaria (ambas en coma flotante). Para crear un complejo en Python:

```
complex(1, 2)
```

Se puede utilizar el sufijo 'j' (representa la unidad imaginaria, que es igual a la raíz cuadrada de -1) en las expresiones para indicar la parte imaginaria. El acceso a las partes real e imaginaria puede verse en el siguiente ejemplo:

```
z = 1 + 2j

# Calcular el módulo usando las partes real e imaginaria - equivale a abs(z)
modulo = (z.real**2 + z.imag**2)**0.5
```

None

Existe un tipo especial `NoneType`, que solamente puede tener un valor: `None`. Se usa en muchas situaciones, pero quizás lo más común es usarla como valor devuelto por defecto de una función. Por ejemplo, `print()` no devuelve nada, pero se puede capturar su valor, que es `None`:

```
return_value = print('abc')
print(return_value)  # se muestra None
```

Expresiones y operadores

Una **expresión** es una combinación de constantes, variables, símbolos de operación y paréntesis que da como resultado un valor de un tipo de dato determinado.

```
-5/2+4*4          (x+y)/2        2*PI*radio
```

Una expresión combina operandos (como los literales enteros `2` o `5`) y operadores (como `+` para la suma) siguiendo unas reglas sintácticas similares a las de la aritmética clásica. Además, pueden contener signos de puntuación, variables y palabras clave del lenguaje, entendidos como los valores generados por la ejecución de esa funcionalidad (como `print` que devuelve `None`).

Un **operador** es un símbolo que determina la operación a realizar sobre los **operandos** (datos) a los que afecta. Una **operación** es el conjunto de un operador y sus operandos (que pueden ser el resultado de otra operación) y que genera un resultado único. Los operandos válidos dependen de los tipos de datos que pueden manejar los operadores. Una operación/expresión genera un resultado de un **tipo** concreto. Si un operador tiene un solo operando es un operador unario `operador operando`; y si tiene dos, binario ‘`operando1 operador operando2`.

Según *el tipo del resultado* las expresiones se clasifican en:

- Aritméticas, su resultado es numérico (entero o real).
- Relacionales, Comparan valores y su resultado es verdadero o falso.
- Lógicas, operan con valores lógicos y devuelven verdadero o falso.
- Carácter, su resultado es un carácter o una cadena de caracteres.

Operadores aritméticos básicos

Se dispone de operadores para las operaciones aritméticas básicas:

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de la división
//	División entera
**	Potenciación

Existe una forma de realizar la división sin abandonar los enteros. El operador de **división entera** en Python se representa por `//`, en otros lenguajes puede ser distinto. También existe el operador `%` que devuelve el **resto de la división** entera, también llamado módulo.

```
x = 3
print("- Valor de x:", x)        # Imprimir un valor
print("- x+1:", x + 1)           # Suma: imprime "4"
print("- x-1:", x - 1)           # Resta; imprime "2"
print("- x*2:", x * 2)           # Multiplicación; imprime "6"
print("- x^2:", x ** 2)          # Potenciación; imprime "9"
print("- x // 2:", x // 2)       # División entera (cociente): imprime "1"
print("- x% 2:", x % 2)          # Módulo (resto de la división)
x += 1                           # Modificación de x
print(x)                         # Imprime "4"
x *= 2
print(x)                         # Imprime "8"
print("- Varias operaciones en una línea:", 1, 2, x, 5*2)
```

Operadores relacionales y lógicos

Python implementa todos los operadores usuales de la lógica booleana, usando palabras en inglés (`and`, `or`, `not`) en lugar de símbolos (|, &, !). También se definen los operadores que permiten establecer la relación entre dos variables, `<`, `>`, `>=`, `<=`, `==`, `!=`.

Símbolo	Python	Descripción	tipo operandos	tipo resultado
$<$	$<$	Menor que	Numérico, carácter, texto	lógico
$>$	$>$	Mayor que	Numérico, carácter, texto	lógico
$=$	$==$	Igual que	Numérico, carácter, texto	lógico
$\leq$	$<=$	Menor o igual que	Numérico, carácter, texto	lógico
$\geq$	$>=$	Mayor o igual que	Numérico, carácter, texto	lógico
$\neq$	$!=$	Distinto	Numérico, carácter, texto	lógico

Símbolo	Python	Descripción	Tipos operando y resultado
no	`not`	negación	lógico
y	`and`	conjunción	lógico
o	`or`	disyunción	lógico

La tabla que muestra el resultado de las operaciones booleanas partiendo de dos valores lógicos `p` y `q` es (`true` y `false` en minúsculas no son correctos en Python):

p	q	p and q	p or q	not p
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

```
# Operaciones Booleanas
a = 15
b = False
c = 4 + True # 5
print(c)
b = (a < 20 and a >= 0)    # b es True si a esta en el intervalo [0,20)
print(b)
```

O por ejemplo:

```
v1 = True
v2 = False
print("- Valores de v1 y v2:", v1, v2)
print("- Tipo de v1:", type(v1)) # Imprime la clase de un  booleano ('bool')

print("- v1 y v2:", v1 and v2)        # y lógico; imprime False
print("- v1 o v2:", v1 or v2)         # o lógico; imprime True
print("- negación v1 ", not v1)       # negación lógica, imprime False

print(3 == 5)           # Imprime False ya que son distintos
print(3 != 5)           # Imprime True ya que son distintos
print(3 < 5)            # Imprime True ya que 3 es menor que 5
```

Los booleanos se pueden definir con el constructor de objetos `bool()`. Los valores de cualquier otro tipo pueden convertirse en booleanos mediante reglas predecibles. Los números son `False` si son 0 y `True` en cualquier otro caso. Las cadenas de texto son `False` si están vacías (“”) y `True` si contienen al menos un carácter. Las listas, tuplas, conjuntos y diccionarios, son `False` si están vacíos y `True` si contienen elementos. El valor especial `None` siempre se considera `False`.

```
# Conversión a booleano
print(bool(2014))           # True
print(bool(0))              # False
print(bool(-3.5))           # True
print(bool("Hola"))         # True
print(bool(""))             # False
print(bool(None))           # False
```

Una expresión booleana recibe también el nombre de **condición**, su resultado será `True` o `False` respondiendo a la situación establecida por sus operandos y operadores. Por ejemplo, si se desea saber si un alumno ha superado cierta asignatura sabiendo que para aprobar hay que obtener un 5 o más en la calificación, se usa la expresión: `Nota>=5`. De esta forma, esa expresión define la **condición** que debe cumplir un alumno para aprobar.

Reglas de precedencia

Cuando se combinan diferentes operadores, sobre todo los aritméticos, el valor determinado por la expresión se calcula en función de unas reglas de precedencia que indican qué operación se ejecuta antes. Estas reglas coinciden con lo esperable a partir de nuestros conocimientos aritméticos, es decir, la multiplicación y la división son prioritarios con respecto a la suma y la resta. La

precedencia habitual se puede alterar utilizando paréntesis. Los paréntesis pueden anidarse unos dentro de otros:

```
(3 + 2) * 5            --- su resultado es ---   25
((3 + 2) * 5 + 1) * 10 --- su resultado es ---  260
```

En presencia de operadores de igual precedencia, el cálculo del valor de una expresión se realiza típicamente de izquierda a derecha. En general, para evitar dudas y para mejorar la legibilidad, se deben usar los paréntesis.

A continuación se muestran los operadores de más prioritarios (en la parte superior de la tabla) a menos prioritarios (en la parte inferior):

Operador	Descripción
`**`	Exponenciación
`+x`, `-x`	Especificación de signo,
`*`, `/`, `//`, `%`	Multiplicación, división, división entera, resto
`+`, `-`	Adición y sustracción
`<`, `<=`, `>`, `>=`, `==`, `!=`	operadores relacionales
`not`	"no" lógico
`and`	"y" lógico
`or`	"o" lógico

Precisión en reales

Las operaciones sobre los reales son las operaciones numéricas básicas salvo la división entera y el resto o módulo de la división que no tienen sentido con reales. Una cosa que hay que tener en cuenta con la aritmética de coma flotante es que su precisión es limitada, lo que puede hacer que las pruebas de igualdad sean inestables. Esto se debe a la forma en que las computadoras almacenan los números de punto flotante. Por ejemplo, la siguiente expresión es falsa:

```
0.1 + 0.2 == 0.3
```

Operaciones con complejos

Los números complejos tienen diversos atributos y métodos que permiten manejarlos. Además, puede ampliarse información de funciones útiles viendo módulo `cmath` para el módulo de un número complejo, su argumento y su forma polar.

```
c = 3 + 4j
c.real              # parte real
c.conjugate()       # complejo conjugado
abs(c)              # modulo, i.e. sqrt(c.real ** 2 + c.imag ** 2)
```

Esta es la primera aparición de la notación punto `XXX.SS` con esto se indica que quiere ejecutar la funcionalidad `SS` con la información contenida en `XXX`, siendo `XXX` un objeto Python. Hay que recordar que todo son objetos en este lenguaje. Un complejo es un objeto con dos atributos, la parte real y la parte imaginaria.

Tipos de datos compuestos

Los tipos de datos compuestos predefinidos en Python son:

Nombre	Ejemplo	Descripción
`str`	`' Hola Mundo'`	Colección de ordenada de caracteres
`list`	`[1, 2, 3]`	Colección ordenada
`tuple`	`(1, 2, 3)`	Colección ordenada inmutable
`set`	`{1, 2, 3}`	Colección no ordenada con datos únicos
`dict`	`{'a':1, 'b':2, 'c':3}`	pares no ordenados (clave,valor)

```
print(type('Informática'))                # str- String - Cadena
print(type([1, 2, 3]))                    # list - Lista
print(type((9.8, 3.14, 2.7)))             # tuple - Tupla
print(type({9.8, 3.14, 2.7}))             # set - Conjunto
print(type({'asignatura':'Informática'}))  # dict - Dictionary - Diccionario
```

Una de las grandes aportaciones de Python es precisamente la incorporación de las listas, tuplas, diccionarios y conjuntos como tipos de datos propios del lenguaje.

De forma general las listas, las tuplas y las cadenas (`str`) son parte del conjunto de las **secuencias** Python que se caracterizan por estar ordenadas. Es decir, cada elemento individual que la compone, tiene asociada una posición (`i`) en la secuencia a partir de 0. Toda secuencia cuenta con las siguientes operaciones:

Operación	Resultado
`x in s`	Indica si el dato de la variable `x` se encuentra en `s`
`s + t`	Concatena las secuencias `s` y `t`
`s * n`	Con `n` un entero, concatena `n` copias de `s`
`s[i]`	Elemento `i` de `s`, empezando por 0
`s[i:j]`	Porción de la secuencia `s` desde `i` hasta `j` (no inclusive)
`s[i:j:k]`	Porción de la secuencia `s` desde `i` hasta `j` (no inclusive), con paso k
`len(s)`	Cantidad de elementos de la secuencia `s`
`min(s)`	Mínimo elemento de la secuencia `s`
`max(s)`	Máximo elemento de la secuencia `s`

El uso de `[]` para acceder a los elementos de las colecciones ordenadas se llaman indexación y segmentación. Su comportamiento se mostrará para el tipo de datos listas.

Cadenas de caracteres

Una cadena de caracteres, tipo de dato `str` (del ingles string), representa precisamente eso, una ristra de caracteres alfanuméricos que puede ser tratada como una unidad. Es realmente un tipo compuesto Python. Las cadenas en Python se crean con comillas simples o dobles:

```
mensaje = " ¿Qué te gusta?"
respuesta = 'spam'
```

Son muchas las operaciones para trabajar con cadenas (numerosas bajo la notación punto). Algunas están disponibles directamente sobre la base del lenguaje Python y otras muchas, están en librerías que se pueden importar, bien sean librerías estándar o bien de terceros (éstas requieren de una instalación adicional). Una librería no es más que un script Python con diversas funciones o bloques de código con nombre agrupadas según algún criterio de uso, también se le llama **paquetes**, **bibliotecas** o **módulos**.

```
a = 'Hola' + 'Mundo'     # ''
b = 'Di ' + a            # 'Di HolaMundo'
len(a)                   # 9
t = 'H' in a             # True
f = 'x' in a             # False
g = 'Di' not in s        # True
rep = a * 3              # 'HolaMundoHolaMundoHolaMundo'
s.upper()                # HOLAMUNDO
```

Una propiedad fundamental de las cadenas es que son **inmutables**. Si se intenta reasignar un carácter a una posición dentro de la cadena, no es posible:

```
palabra = "Python"
palabra[0] = "N"         # Esto da error
```

Con `str()` se crea y/o se convierte cualquier valor a cadena. El resultado es una cadena con el mismo contenido que hubiera mostrado el comando `print()` sobre la expresión entre paréntesis:

```
x = 76
b = str(x)
print(x," ", b)        # el resulado es:   76 76
```

f-Strings

Un f-String es una cadena que está asociada a una o varias variables, permitiendo generar una cadena formateada. Si se utiliza `f"XX {var} XXX"` se está indicando que esa cadena debe construirse poniendo el contenido de la variable `var` en donde aparece en la cadena.

```
mensaje = 'Gracias'
respuesta = 'De nada'
frase = f"Para el mensaje: {mensaje} hay una  respuesta: {respuesta}."
print(frase)
```

Otro ejemplo de **f-Strings** :

```
producto = 'Calabacín'
cajas = 1000
precio = 1.25

# Formateo de la información
detalle_producto = f"{producto:>10s} {cajas:10d} cajas"
costo_total = f"Costo total = {cajas * precio:,.2f} Euros"

# Impresión de la información formateada
print("Detalles del producto:")
print(detalle_producto)
print(costo_total)
```

Las cadenas ofrecen una amplia variedad de métodos para validar y manipular textos. Siendo `s` una cadena algunos de estos métodos se muestran en la tabla, incluyendo a continuación diversos ejemplos de uso:

Método	Descripción
`s.endswith(suffix)`	Verifica si termina con el sufijo
`s.find(t)`	Primera aparición de `t` en `s` (o -1 si no está)
`s.index(t)`	Primera aparición de `t` en `s` (error si no está)
`s.isalpha()`	Verifica si los caracteres son alfabéticos
`s.isdigit()`	Verifica si los caracteres son numéricos
`s.islower()`	Verifica si los caracteres son minúsculas
`s.isupper()`	Verifica si los caracteres son mayúsculas
`s.join(slist)`	Une una lista de cadenas usando s como delimitador
`s.lower()`	Convertir a minúsculas
`s.replace(old,new)`	Reemplaza texto
`s.split([delim])`	Parte la cadena en subcadenas
`s.startswith(prefix)`	Verifica si comienza con un prefijo
`s.strip()`	Elimina espacios en blanco al inicio o al final
`s.upper()`	Convierte a mayúsculas

Para **dividir** una cadena, generando una lista de palabra se utiliza `split`.

```
cadenaNumeros = "3, 4, 5, 2, 1"
palabras = cadenaNumeros.split(",")      # ['3', ' 4', ' 5', ' 2', ' 1']
```

Si se desea **unir** las cadenas de una lista `palabras` en una sola cadena, separando los elementos con un símbolo definido, en el ejemplo `-`.

```
unida = "-".join(palabras)
```

Para **verificar contenido** con funciones que devuelven un valor booleano

```
contiene = "4,5" in cadenaNumeros
empiezaCon = cadenaNumeros.startswith("3,")      # True
terminaCon = cadenaNumeros.endswith(",1")        # True
```

Tipo lista

Es un tipo compuesto que puede almacenar distintos valores (llamados ítems o componentes) y que pueden ser modificados. Para crear una lista basta con poner sus componentes entre `[ ]` (corchetes) y separarlos con `,` (comas).

Las listas son ordenadas, es decir, mantienen el orden en el que se definen independientemente de los valores de sus componentes. Pueden contener elementos de tipos arbitrarios, incluso otras listas, permitiendo la creación de estructuras anidadas. El acceso a los elementos se realiza por posición, utilizando índices entre corchetes `[i]` y como son dinámicas, se pueden añadir o eliminar elementos. Las listas son equivalentes a las colecciones indexadas y estructuradas clásicas, como los vectores de otros lenguajes.

Las listas especialmente útiles para almacenar y manipular colecciones de datos relacionados. Por ejemplo, se pueden utilizar para guardar los nombres de los clientes de un negocio, los precios de los productos en una tienda o las calificaciones de los estudiantes en una clase.

```
print("- Lista con 4 números:")
a = [57, 45, 7, 13] # una lista con cuatro números
print(a)

print("\n- Lista con 3 cadenas:")
b = ["Estadística", "Informática", "Buen día"] # una lista con tres strings
print(b)
```

Indexación y segmentación de listas

Python proporciona acceso a elementos en tipos compuestos a través de la indexación de elementos individuales. Se mostrarán los ejemplos sobre la siguiente la lista de los 5 primeros números primos: `L = [2, 3, 5, 7, 11]`.

Se utiliza la indexación basada en cero, por lo que se accede al primer y segundo elemento mediante de la lista `L` con:

```
print(L[0], L[1])
```

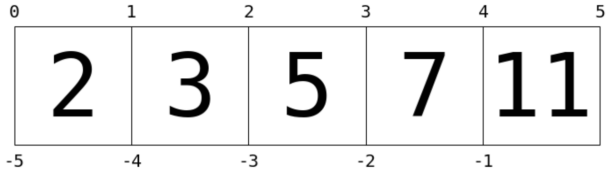

Figura 8: Índices positivos y negativos (números pequeños arriba y abajo)

Se puede acceder a los elementos contando del final al principio de la lista con números negativos, comenzando desde -1.

```
L[-2]   # Devuelve 7
L[ 2]   # Devuelve 5
```

La indexación es un medio para obtener un único valor de la lista y la segmentación es un medio para acceder a múltiples valores en **sublistas**. Se utilizan dos puntos para indicar el punto inicial (inclusive), y el punto final (no incluido) de la submatriz. Por ejemplo, para obtener los tres primeros elementos de la lista, se puede escribir:

```
n=L[0:3]
```

Se crea una lista `n` que toma los elementos desde el índice 0 hasta el índice 3 (sin incluirlo). `[2, 3, 5]` Pero se puede omitir el primer índice, `0`, escribendo su equivalente:

```
L[:3]   # Devuelve [2, 3, 5]
```

Del mismo modo, si se omite el último índice, el valor predeterminado es la longitud de la lista. Por lo tanto, se puede acceder a los últimos tres elementos de la siguiente manera:

```
L[-3:]   # Devuelve [5, 7, 11]
```

Finalmente, es posible especificar un tercer entero que represente el tamaño de paso/salto. Por ejemplo, para seleccionar cada segundo elemento de la lista, se escribe:

```
L[::2]   # equivalente a L[0:len(L):2]
```

Una versión muy útil es especificar un tamaño de paso negativo, lo que invertirá la lista:

```
L[::-1]
```

A modo de muestra de las operaciones de indexación:

```
nums = list(range(5)) # range  crea lista de enteros
print(nums)           # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])      # Desde el indice 2 al  4 (exclusivo); muestra "[2, 3]"
```

```
print(nums[2:])        # Desde el indice 2 hasta el final; muestra "[2, 3, 4]"
print(nums[:2])        # Desde el principio hasta 2 (exclusivo) muestra"[0, 1]"
print(nums[:])         # Trozo de toda la lista; muestra "[0, 1, 2, 3, 4]"
print(nums[:-1])       # Se pueden usar negativos; muestra "[0, 1, 2, 3]"
nums[2:4] = [8, 9]     # Asigna a nueva sublista al trozo indicado
print(nums)            # Muestra "[0, 1, 8, 9, 4]"
```

Otra peculiaridad, es que se pueden combinar asignación y segmentación para modificar en bloque varios ítems de la lista:

```
letras = ['a', 'b', 'c', 'd', 'e', 'f']
letras[:3]
letras[:3] = ['A', 'B', 'C']
print(letras)
```

Asignar una lista vacía equivale a borrar los ítems de la lista o sublista:

```
letras[:3] = []
```

Listas con datos de varios tipos

Las listas son un elemento muy abierto que permite mezclar elementos de distinto tipo:

```
a = [90, "Python", 3.87]
print(a[0])    # 90
print(a[1])    # Python
print(a[2])    # 3.87
```

Incluso listas dentro de listas (llamadas listas anidadas). Se pueden manipular fácilmente este tipo de estructuras utilizando múltiples índices, como si fuesen las filas y columnas de una tabla. Para representar una matriz se puede trabajar con listas de listas, es decir, listas anidadas `metro=[[1,2,3],[4,5,6],[7,8,9]]` y se puede acceder al elemento usando la notación `variable[fila][columna]`.

```
a = [1,2,3]
b = [4,5,6]
c = [7,8,9]
metro = [a, b, c]

print(metro[0])       # Salida: [1, 2, 3] - Primera sublista
print(metro[-1])      # Salida: [7, 8, 9] - Última sublista
print(metro[0][0])    # Salida: 1 - Primer elemento de la primera sublista
print(metro[1][1])    # Salida: 5 - Segundo elemento de la segunda sublista
print(metro[-1][-1]) # Salida: 9 - Último elemento de la última sublista
```

De esta forma, se puede representar información agrupada por entidades, muy similar a las clásicas colecciones estructuradas o registros. Por ejemplo, los nombres del los agricultores, su municipio y el total de hectareas de su explotación:

```
agricultores = [
    ["Miguel", "Almería", 1200],
    ["Ana", "Tahal", 1700],
    ["Lourdes", "Berja", 1500]
]

print(agricultores)
print("")

print(*agricultores)  # Separa por espacios el primer nivel
print("")

print(agricultores[1])
print(*agricultores[1])  # Separa por espacios datos del segundo agricultor
```

Operaciones sobre listas

Una lista es un objeto al que con la notación punto se le puede enviar un mensaje `objeto.metodo(parametro1,parametro2, ...)`. Para listas, por ejemplo, está el método para añadir un elemento a la lista. Si se tiene la lista `a` y se quiere que añadir el elemento `x`, se usa `a.append(x)`. El intérprete lo que hace es enviar un mensaje al objeto lista `a` con el argumento `x`.

Las listas disponen de diversos métodos útiles. Seguidamente se muestran solamente algunos ejemplos, en la documentación de Python se recogen todas las operaciones:

Método/operación	Descripción
`n=len(lista)`	Guarda en `n` la logitud de la `lista`
`lista.append(x)`	Añade el valor `x` a la `lista`
`lista + [13, 17]`	Genera una lista que concatena `lista` con `[13, 17]`
`lista.clear()`	Deja la lista vacía
`lista.remove(3)`	Eliminar elementos de `lista` con valor `3`
`lista.reverse()`	Invierte los elementos de `lista`
`del lista[3]`	Borra el elemento de la posición 3 de `lista`
`lista.extend(lista2)`	Concatena la primera lista con la segunda
`lista.insert(6,5.5)`	Inserta el valor 5.5 en la posición 6 de lista
`lista.pop([i])`	Elimina y devuelve el elemento en la posición `i` de `lista`. Si no se especifica `i`, elimina y devuelve el último elemento
`lista.clear()`	Elimina todos los elementos de lista, dejándola vacía
`lista.count(x)`	Devuelve el número de veces que el valor `x` aparece en `lista`
`lista.index(x[,`	Devuelve el índice de la primera aparición de `x` en `lista`.
`start[, end]])`	Opcionalmente, se puede especificar un rango con `start` y `end`
`sorted(lista)`	Devuelve una lista ordenada a partir de `lista`
`lista.sort()`	Ordena `lista`
`c = lista.copy()`	Devuelve una copia de lista

Para dejar la lista original intacta pero obtener una nueva lista ordenada a partir de ella, se usa la función `sorted`.

```
bs = [7, 4, 6, 8, 3]
cs = sorted(bs)
print(bs)
print(cs)
```

Para modificar la lista original se usa `sort()`, con notación punto:

```
ds = [6, 8, 7, 3, 1]
ds.sort()
```

Tipo tupla

Las tuplas son inmutables lo que las hace ideales para almacenar datos como coordenadas geográficas, configuraciones del sistema o puntos en un plano cartesiano. Son una excelente opción si se quiere que los datos de una colección sean solamente de lectura. Se crean con `()` (paréntesis):

```
semana = ('Lunes','Martes','Miércoles','Jueves','Viernes','Sábado','Domingo')
t = (1, 2, 3)
```

También se pueden definir sin ningún tipo de corchetes:

```
t = 1, 2, 3
print(t)
```

Se accede a los elementos con la indexación:

```
a = (1, 2, 57, 4)

print("- Una tupla de cuatro elementos:", a)
print("- El elemento con índice 2:", a[2])
print("- Los elementos entre los índices 0 y 2:", a[0:2])
```

Un caso particularmente común de uso de las tuplas es en funciones que tienen múltiples valores de retorno. Por ejemplo, el método `as_integer_ratio()` se usa con valores reales y devuelve un numerador y un denominador cuyo cociente se aproxima a ese real. Este doble valor de retorno viene en forma de una tupla:

```
x = 0.125
t = x.as_integer_ratio()

print(t)
print("numerador", t[0])
print("denominador", t[1])

t = list(t) # se puede convertir en una lista para se rmodificada
print(type(t))
```

Estos valores múltiples devueltos se pueden asignar individualmente de la siguiente manera:

```
x = 0.5
numerador, denominador = x.as_integer_ratio()
print(numerador, "/", denominador)
```

A las tuplas se le pueden aplicar algunos de los métodos de listas : `list()`, `sorted()`, o `count()`. Sin embargo otros no pueden ser utilizados directamente sobre las tuplas por ser inmutables: `append()`, `extend()`, `insert()`, `remove()`, `pop()` o `reverse()`. Para ser utilizados se debería crear una nueva tupla que contenga la copia de los elementos modificados:

```
tupla_original = (1, 2, 3, 4, 5)
tupla_invertida = tuple(reversed(tupla_original))

print("Tupla original:", tupla_original)
print("Tupla invertida:", tupla_invertida)
```

Tipo conjunto

A diferencia de las listas y las tuplas, los conjuntos no permiten elementos duplicados. Admiten las operaciones de la teoría de conjuntos como unión, intersección y diferencia, lo que los hace muy útiles para resolver problemas de lógica y teoría de conjuntos. Se puede verificar rápidamente si un elemento pertenece a un conjunto, lo que los convierte en una estructura de datos eficiente para realizar búsquedas.

Los conjuntos son útiles cuando se necesita trabajar con datos únicos, como palabras en un texto, usuarios en una red social o elementos de una colección. También pueden ser valiosos para realizar operaciones de filtrado y clasificación de datos. Se definen utilizando llaves, `{ }` :

```
primos = {2, 3, 5, 7}
impares = {1, 3, 5, 7, 9}
```

Cada operación tiene dos métodos equivalentes, uno usando un operador y otro con un método:

```
# union:
primos | impares      # con operador
primos.union(impares) # con método equivalente

# intersección
primos & impares             # con operador
primos.intersection(impares) # con método equivalente

# diferencia
primos - impares           # con operador
primos.difference(impares) # con método equivalente

# diferencia simétrica: items que aparecen en sólo uno de los conjuntos
primos ^ impares                     # con operador
primos.symmetric_difference(impares) # con método equivalente
```

Otra forma de crear conjuntos es utilizando la función `set()`, a la que se le pasa como argumento alguna otra colección (cadenas, listas, tuplas, conjuntos y diccionarios).

Una forma simple de quitar los duplicados de una lista aprovechando las propiedades de los conjuntos:

```
nombres = ["James", "Bob", "James", "Mark", "Kate", "Sarah", "Kate"]
print(list(set(nombres)))
```

Están disponibles muchos más métodos y operaciones de conjuntos que se pueden consultar en la documentación de Python. Algunos ejemplos son:

```
s1 = {1, 2, 3, 4}
s2 = {3, 4, 5, 6}
s3 = {1, 2, 3}
s4 = {1, 2, 3, 4, 5}
s5 = {1, 2, 3}
s6 = {1, 2, 3}

# Saber si es subconjunto
print(s1 <= s2)
print(s1.issubset(s2))

print(s3 <= s4)
print(s3.issubset(s4))

# Para saber si puede ser subconjunto
print(s3 < s4)
print(s5 < s6)
print(s5 < s5)

# Para saber si es súper conjunto
print(s4 >= s3)
print(s4.issuperset(s3))
print(s4 > s3)
print(s4 > s4)
```

Tipo diccionario

Los **diccionarios** son colecciones **no secuenciales y mutables** de elementos. Están compuestos por una **clave** (que identifica de modo único al elemento) y el valor que se desea almacenar. Esto permite acceder rápidamente a los datos mediante una clave única en lugar de una posición numérica. Los diccionarios son ideales para representar colecciones estructuradas (registros), como la información de usuarios o registros en bases de datos. Por ejemplo, se puede usar un diccionario para almacenar información sobre una persona, con claves como "nombre", "edad" y "ciudad". Gracias a su rapidez en la búsqueda y actualización de valores, los diccionarios son una opción potente para trabajar con grandes volúmenes de datos. Se pueden crear a través de una lista separada por comas de pares `key:value` dentro de llaves. No pueden aparecer elementos con la misma clave y son muy versatiles:

```
num = {1: 'uno', 2: 'dos', 3: 'tres', 4: 'cuatro'}

asig = {
    'asignatura': 'Fundamentos de Programación', 'créditos': 6,
    'tipos': ['teoría', 'prácticas']
}

nom = {
    'persona': {
      'nombre': 'Eva','primer apellido':'García', 'segundo apellido':'Pérez'
    },
    'edad': 30
}

print(num)
print(asig)
print(nom)
```

Se accede a los elementos y se modifican a través de la sintaxis de indexación utilizada para listas y tuplas, excepto que aquí el índice no es un orden basado en cero, sino una clave válida en el diccionario:

```
# Acceso via la clave
numeros['uno']
```

También se pueden agregar nuevos elementos al diccionario mediante la indexación:

```
numeros['dos'] = 22
```

Los diccionarios no mantienen ningún sentido de orden para los parámetros de entrada. Esta falta de ordenación permite que se implementen de manera eficiente, así el acceso rápido a elementos aleatorios, independientemente del tamaño del diccionario.

Por ejemplo, si se quiere guardar una base de datos de personas con el nombre, edad, teléfono y dirección de una persona se puede utilizar un diccionario:

```
nombre = input('¿Cómo te llamas? ')
edad = input('¿Cuántos años tienes? ')
direccion = input('¿Cuál es tu dirección? ')
telefono = input('¿Cuál es tu número de teléfono? ')

persona = {'nombre': nombre, 'edad': edad, 'direccion': direccion,
         'telefono': telefono}

print(persona['nombre'], 'tiene', persona['edad'], 'años, vive en', \
      persona['direccion'], 'y su número de teléfono es', persona['telefono'])
```

Otro caso de uso de un diccionario es almacenar tres productos fitosanitarios. Cada producto está representado por otro diccionario que contiene diferentes campos de información, como el nombre del producto, los ingredientes activos, el fabricante, el precio y la cantidad disponible. De esta forma, si se definen tres productos fitosanitarios insecticida1, fungicida1 y herbicida1, el diccionario en el que se guarda su información sería:

```
# Definir un diccionario de productos fitosanitarios
productos_fitosanitarios = {
    "insecticida1": {
        "nombre": "Insecticida A",
        "ingredientes_activos": ["Acetamiprid", "Imidacloprid"],
        "fabricante": "Fabricante X",
        "precio": 50.0,
        "cantidad": 100
    },
    "fungicida1": {
        "nombre": "Fungicida B",
        "ingredientes_activos": ["Tebuconazol", "Propiconazol"],
        "fabricante": "Fabricante Y",
        "precio": 75.0,
        "cantidad": 50
    },
    "herbicida1": {
        "nombre": "Herbicida C",
        "ingredientes_activos": ["Glifosato"],
        "fabricante": "Fabricante Z",
        "precio": 100.0,
        "cantidad": 25
    }
}
```

Para acceder a un elemento específico del diccionario, se utiliza la clave correspondiente. Por ejemplo, para acceder al nombre del primer producto, se utiliza la siguiente sintaxis:

```
nombre_producto1 = productos_fitosanitarios["insecticida1"]["nombre"]
print(nombre_producto1)  # Imprime "Insecticida A"
```

O para actualizar la cantidad de un producto tras una venta:

```
productos_fitosanitarios["insecticida1"]["cantidad"] -= 10
print(f"Nueva cantidad de {productos_fitosanitarios['insecticida1'] \
    ['nombre']}: {productos_fitosanitarios ['insecticida1']\
    ['cantidad']} unidades")
```

A modo de resumen

La elección del tipo de colección a utilizar depende en gran medida de la naturaleza del problema que se esté abordando y de las características específicas de los datos con los que se va a trabajar. Si se requiere almacenar información que pueda cambiar con el tiempo, agregando o eliminando elementos de manera dinámica, las **listas** son una excelente opción, ya que permiten modificaciones flexibles y un fácil recorrido secuencial.

Por otro lado, cuando los datos deben permanecer inmutables o se necesita asegurar que su posición se conserve, las **tuplas** son la opción más adecuada. Son especialmente útiles para almacenar pares de valores que deben mantenerse juntos sin cambios.

Si el objetivo es eliminar duplicados o realizar operaciones de conjuntos, como intersecciones o uniones, los **conjuntos** son ideales, ya que almacenan elementos únicos y permiten verificar

rápidamente la pertenencia de un dato.

Finalmente, para asociar valores con claves específicas y facilitar un acceso rápido y eficiente a la información, los **diccionarios** son la mejor opción, permitiendo organizar los datos mediante claves únicas para búsquedas ágiles y estructuradas.

Tipo	Usos Comunes	Ejemplo de uso en estadística y probabilidad
Lista Ordenado, Mutable	Cálculos secuenciales, series de datos, promedios, simulaciones, regresiones, etc.	Almacenar datos para calcular estadísticos, o realizar una simulación de crecimiento de un cultivo. Por ejemplo, `datos = [23, 45, 67, 12, 34]` para calcular la media y desviación estándar.
Tupla Ordenado, Inmutable	Parámetros constantes, relaciones fijas,espacios muestrales fijos, ecuaciones, combinaciones de eventos, etc.	Almacenar parámetros constantes de una distribución normal. Ejemplo: `normal = (0,1)` para media 0 y desviación estándar 1, o representar un espacio muestral del lanzamiento de un dado: `dado=(1, 2, 3, 4, 5, 6)`
Conjunto No ordenado, Único	Operaciones con conjuntos, trabajar con datos únicos (eliminar duplicados), y operacio- nes matemáticas de intersección, uniones, etc.	Almacenar valores únicos de una muestra. Para eliminar respuestas duplicadas y analizar los valores únicos de una encuesta: `valores_unicos = set([1, 2, 2, 3, 3, 4])`
Diccionario Clave-Valor	Mapeo de valores, acceso a datos asociados a claves únicas, análisis de categorías y estadísticas por grupo, etc.	Almacenamiento estadísticas en las que a cada resultado se le asocia una información adicional. Por ejemplo: estadísticas descriptivas por grupo: `estad = {'gr1': {'media': 50, 'desviacion': 5}, 'gr2': {'media': 45, 'desviacion': 4}}`

Instrucción de asignación

En programación, la **asignación** es la acción que consiste en dar o asignar el valor de una expresión a una variable, permitiendo almacenar datos en memoria para su uso posterior. Esta instrucción se ejecuta mediante el operador de asignación, que en muchos lenguajes de programación es el símbolo igual (`=`):

```
variable = expresión
```

El valor resultante al evaluar la `expresión` de la parte derecha es almacenado en la posición de memoria que representa la variable de identificador `variable` de la izquierda.

```
x = 4
y = 5
z = (2 * x + y) % 3
```

Una asignación es *destructiva* puesto que se pierde el valor anterior de la variable. Es distinta de la igualdad matemática. El tipo de dato del resultado de la expresión ha de coincidir con el de la variable en la que se almacena. En muchos lenguajes de programación se relaja esta regla, controlando el traductor algunas conversiones de tipos de datos, por ejemplo almacenar un valor entero en una variable real.

Se pueden utilizar expresiones con operadores que admiten varios tipos de datos (normalmente numéricos). En estas situaciones se hace la conversión del resultado y de los cálculos intermedios al tipo de dato con rango más amplio (en cuanto al espacio en memoria). :

```
x = 4
y = 5.5
z = (2 * x + y) % 3
```

En lenguajes como el C las variables son como contenedores o cajones donde se ponen datos:

```
// C code
int x = 4;
```

Se está creado una zona de memoria según tan grande como el tamaño de los enteros (2 bytes) llamada `x`, y se pone el valor `4` en ella. Por contra Python considera las variables como referencias o punteros. Si en Python se escribe `x = 4` esencialmente se define un *puntero* llamado `x` que guarda la dirección de memoria donde se ha puesto el valor `4`.

Al estar tipado dinamicamente en Python se pueden hacer cosas como:

```
x = 4.0        # s es un real
x = 1          # x es un entero
x = 'hello'    # ahora x una cadena
x = [1, 2, 3] # ahora x es una lista
```

Hay que tener cuidado porque una consecuencia del uso de las variables como puntero/referencia para tipos compuestos, es que si se trabaja con dos nombres de variable sobre el mismo objeto, cuando se cambia sobre una variable la otra también se ve afectada. Por suerte esto no pasa con los tipos simples. Esta situación también aparece en otros lenguajes como JAVA. Por ejemplo:

```
x = y = [7, 8, 9]    # x e y referencia la misma lista, [7, 8, 9]
x.append(4)
print(x)             # [7, 8, 9, 4]
print(y)             # [7, 8, 9, 4]
z = x                # z referencia la misma lista que x y que y
z[0] = 10
print(y)             # [10, 8, 9, 4]
```

Se crean dos variables `x` e `y` que apuntan al mismo objeto. Al añadir elementos a `x` afecta también a `y`.

No obstante para números, cadenas y otros tipos simples, es perfectamente seguro hacer operaciones como las siguientes:

```
x = 10
y = x
x = x + 5           # sumamos 5 al valor de x, y se le asigna a x
print("x =", x)     # x = 15
print("y =", y)     # y = 10
```

Cuando se ejecuta `x = x + 5`, no se está modificando el valor del objeto `10` apuntado por `x`; más bien se está cambiando la variable `x` para que apunte a un nuevo objeto entero con valor `15`. Por esta razón, el valor de `y` no se ve afectado por la operación.

Python admite la **asignación múltiple** poniendo una serie de variables separadas por comas y también el mismo número de valores o expresiones separadas por comas. La asignación en **cascada** o en tubería también está disponible.

```
a = b = c = 1       # Asignación en tubería todos valen 1
a, b, c = 1, 2, 3   # Asignación múltiple
```

Cualquier operador matemático puede ser utilizado antes de `=` para hacer una **operación embebida**, simplificando así la escritura:

Asignacion	
`-=`	disminuir la variable
`+=`	incrementar la variable
`*=`	multiplicar la variable
`/=`	divide la variable
`//=`	división entera de la variable
`%=`	devuelve sobre la variable su módulo
`**=`	elevar a una potencia

```
a += 1              # a = a + 1
print (a)
a *= 2              # a = a * 2
print (a)
```

Instruccion de entrada y salida

Las principales funciones para la interacción con el usuario en Python son `input()` para lectura (entra la información al programa desde el exterior) y `print()` para escritura (sale la información generada en el programa).

La función `input()` permite obtener texto escrito por teclado sobre variables. Al llegar a la función, el programa se detiene esperando que se escriba algo y se pulse la tecla $<$ *intro* $>$:

```
nace = int(input("Año nacimiento "))
nombre = input("¿Cómo se llama? ")
```

Su conjunción con los `f-string` y `print` permite generar la interacción con el usuario en la que se mezclan diferentes tipos de variables. Puesto que lo que se leen son cadenas, si se quieren leer números se tiene que hacer la **conversión**, muchas vesces sobre la misma línea:

```
eur_a_usd = 1.10
actual = 2025

nombre = input("Por favor, dame tu nombre: ")
edad = int(input(f"{nombre}, dame tu edad: "))
euros = float(input(f"{nombre}, dame una cantidad en € (hasta centimos): "))

print(f"\nHola, {nombre}.")
print(f"Tu edad es {edad} años,  naciste aproximadamente en {actual - edad}.")
print(f"{euros:.2f} € son {euros * eur_a_usd:.4f} dólares.")
```

También se pueden hacer lecturas múltiples con comprensión de listas (que se verán con detalle más adelante, puesto que suponen una instrucción de iteración `for`),

```
# Lectura sobre acoef, bcoef, ccoef de tres reales
# sobre la tupla  ('a', 'b', 'c'), coef toma estos tres valores

acoef, bcoef, ccoef = [float(input(f'Dame el coeficiente {coef}: ')) \
  for coef in ('a', 'b', 'c')]

print('    ',acoef,'x² +', bcoef,'x +', ccoef,'= 0\n')
```

Ya se han mostrado divesos ejemplos del uso de otra principal instrucción de salida, `print`, que se puede utilizar tanto en datos simples como en datos compuestos:

```
x = ("tomate", "pimiento", "pepino")
print(x)
print(x[0])
```

La inclusión de separadores ayuda a dar un mejor formato a las salidas:

```
print("Hola", "¿Cómo estás?", sep="---")
```

Se puede usar expresiones sobre la salida. Se aplica el operador y después se imprime el resultado:

```
print ("Bienvenidos")
print (8 * "\n")
print ("Bienhallados")
```

De forma predeterminada, la función de impresión en Python termina con una nueva línea si no se cambia con el parámetro `end`. En este caso, se usa un espacio mostrándose los dos mensajes en la misma línea y acabando con `!`.

```
print ("Bienvenidos", end = ' ')
print ("al maravilloso mundo de la PROGRAMACION", end = '!')
```

Para personalizar los mensajes se pueden usar las variables con distintas estrategias:

```
nombre = "Juan"
edad = 25

print("Hola, me llamo ",nombre, "y tengo", edad, "años." )

# Usando f-strings
print(f"Hola, mi nombre es {nombre} y tengo {edad} años.")

# Usando .format() para insertar valores en la cadena
print("Hola, mi nombre es {} y tengo {} años.".format(nombre, edad))

# Al estilo de C
print("Hola, me llamo %s y tengo %s años."%(nombre, edad))

# Usando índices
print("Hola, me llamo {0} y tengo {1} años.".format(nombre, edad))

# Usando nombres
print("Hola, me llamo {nom} y tengo {ed} años.".format(nom=nombre, ed=edad))
```

Para la entrada de datos es habitual la combinación de `input` y `split`, que permite separar en datos individuales una cadena leída:

```
entrada = input("Introduce tres números separados por espacios: ")
numeros = entrada.split()  # Se genera una lista con los datos

print("Número 1:", numeros[0])
print("Número 2:", numeros[1])
print("Número 3:", numeros[2])
```

Si se quieren usar comas como separador se da la coma `,` o el parámetro separador deseado a `split`.

```
entrada = input("Introduce tres números separados por comas: ")
numeros = entrada.split(",")
```

O por ejemplo para leer una fecha separada por `/`.

```
entrada = input("Introduce una fecha en formato día/mes/año: ")
fecha = entrada.split("/")  # Se genera una lista con los datos

print("Día:", fecha[0])
print("Mes:", fecha[1])
print("Año:", fecha[2])
```

Bloques de captura de errores

Al programar algunas veces se pueden anticipar errores de ejecución. Incluso en un programa sintáctica y lógicamente correcto, puede llegar a haber errores causados por entrada de datos inválidos o inconsistencias predecibles.

Se pueden usar los bloques **try** y **except** para manejar los errores como excepciones. Su estructura se divide en dos partes obligatorias y una opcional:

- **try**: Contiene el código donde puede ocurrir una excepción. Este código se ejecuta en un "modo protegido", es decir, se ejecuta con un mecanismo de seguridad que detecta errores sin detener el programa.
- **except**: Contiene el código que se ejecutará si ocurre una excepción dentro del bloque `try`.
- **finally**: Puede no incluirse y contiene código que se ejecutará siempre, haya ocurrido una excepción o no.

```
num = 10
div = 0  # Esto provocará una excepción de división por cero

try:
    res = num / div         # Aquí ocurre la excepción
    print(res)              # no se ejecutará si hay un error
except ZeroDivisionError:  # Captura la excep. concreta de división por cero
    print("Error: Intentaste dividir entre cero :( ")
```

Existen diferentes tipos de excepciones según el error que se produzca. Las que se generan por conversión se dice que son de tipo **ValueError** y las de división por cero son del tipo **ZeroDivisionError**.

```
try:
    numero = int(input("Introduce un número entero: "))
    print (numero)
    # procesar el número leido
    #....
except ValueError:
    print("Error: Debes ingresar solo números enteros, no letras \
        ni caracteres especiales.")
```

Funciones integradas e importadas

Además de las operaciones numéricas básicas, los lenguajes de programación incluyen otras operaciones que no son más que un bloque de código con nombre que usa unos valores. Algunas de ellas se han ido utilizando en los ejemplos previos, como `max()` o `len()`.

Estas funciones están disponibles en el **lenguaje base** o nativo. Esto significa que pueden utilizar directamente sin importarlas ni configurarlas dentro de los scripts Python. Algunas de las más utilizadas se muestran a continuación: `print()`, `len()`, `type()`, `int()`, `float()`, `str()`,

`input()`, `list()`, `dict()`, `min()`, `max()`, `sum()`, `sorted()`, `open()`, `file()`, `help()`, y `dir()`.

```
# Impresión
print ("hola")

# Longitud de una lista
len("no se que es eso ")
lista = [1, 2, 3, 4]
print(len(lista))  # Salida: 4

# Máximo
max(22,44)

# Conversión a entero
numero = int("20")
print(numero)  # Salida: 20

# Conversion de tupla a lista
tupla = (1, 2, 3)
lista = list(tupla)
print(lista)  # Salida: [1, 2, 3]
```

Existen otras funciones que para poder ser utilizadas requiren de la importación del paquete o librería donde estan definidas. Por ejemplo:

- Raíz cuadrada: **math.sqrt(x)**
- x elevado a y: **math.pow(x,y)**
- Coseno: **math.cos(x)**
- Seno: **math.sin(x)**
- Tangente: **math.tan(x)**
- Exponencial de x (e elevado a x): **math.exp(x)**
- Logaritmo natural (o neperiano) en base e: **math.log(x)**
- Logaritmo en base 10: **math.log10(x)**
- Obtener la carpeta actual del archivo: **os.getcwd()**
- Copiar un archivo: **shutil.copy('archivo.txt', 'copia_archivo.txt')**

Estas funciones **no son parte del lenguaje base**, sino del paquete **math**, **os**, o **shutil**, por lo que para utilizarlas, el programa deberá comenzar con la línea `import math`, como en el siguiente ejemplo:

```
#  Raíz cuadrada de un número
import math
n = float(input("Dime un número: "))
raiz = math.sqrt(n)
print( "Su raíz cuadrada es", raiz)

# Imprimir pi con tres decimales
print('El valor de pi es aproximadamente %5.3f.' % math.pi)
```

De todas las librerías que no requiren instalación adicional en Python, las llamadas **librerías estándar**, las más habituales son:

Librería	Descripción
`collections`	Estructuras de datos adicionales
`csv`	Procesamiento de los archivo separados por comas
`datetime`	Gestión de fechas y tiempos
`decimal`	Aritmética de punto fijo y de punto flotante, incluidos cálculos monetarios
`math`	Operaciones y constantes matemáticas habituales
`os`	Interacciones con el sistema operativo
`random`	Números pseudoaleatorios
`re`	Gestión de expresiones regulares
`statistics`	Funciones de estadística matemática como la media, la mediana, la moda y la varianza.
`sys`	Procesamiento de argumentos de línea de comandos; flujos de entrada estándar, salida estándar y error estándar
`timeit`	Análisis de rendimiento
`functools`	Herramientas para programación funcional, como lru_cache y partial
`itertools`	Generadores y combinaciones eficientes para trabajar con iteradores
`json`	Manejo de datos en formato JSON
`pathlib`	Manejo más moderno de rutas de archivos en comparación con os.path
`subprocess`	Ejecutar comandos del sistema operativo desde Python
`uuid`	Generación de identificadores únicos universales (UUID)

Diseño y Control de Flujo en Programación

Crear una solución software no es diferente de la resolución de problemas en general. Escribir un programa en un lenguaje de programación concreto es casi el último paso del proceso de determinar, primero cuál es el problema, y después el método que se usará para resolverlo.

Elaboración de soluciones a problemas. Algoritmos.

La disciplina de ingeniería que se ocupa de la producción de software se denomina **ingeniería del software**, que se define como la *aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación, y mantenimiento del software*.

El procedimiento de desarrollo de software, se ilustra en la figura, pero hay que entender que se trata de una versión simplificada. Buscando un símil en ingeniería de la construcción, no es igual el método para construir un rascacielos que una casa unifamiliar, pero se fundamentan en los mismos principios. Los ejemplos que se tratan en este libro utilizando este mismo símil son "casitas para pájaros".

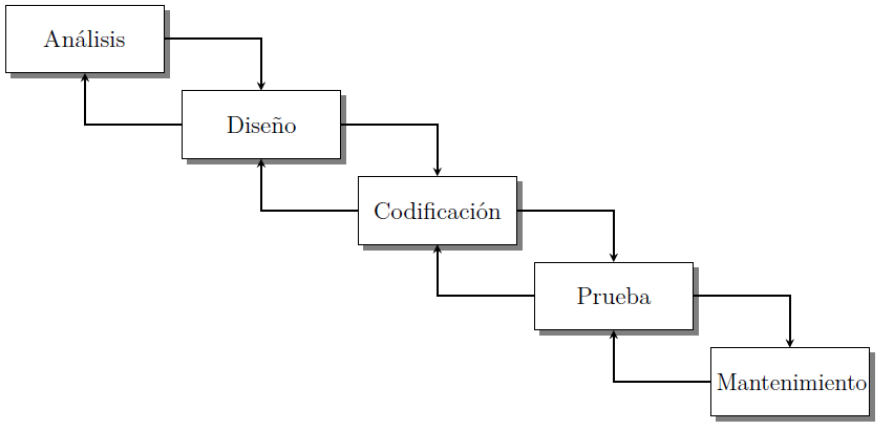

Figura 9: Ciclo de vida del software

El **análisis** representa el **Qué**. Conlleva la definición concreta del problema, la especificación de la información/datos de entrada y salida, la identificación de las tareas a realizar, así como la definición de los criterios de validación indicando cuáles son los casos para los que la solución que se construya debe resolver el problema.

El **diseño** supone la elaboración de la solución, es decir, el **Cómo**. Conlleva la definición de los algoritmos y datos específicos a utilizar incluyendo su tipo. Se trata de un proceso creativo en dos pasos, primero se hace el **diseño preliminar o global** donde se define la estrategia de resolución. Después durante el **diseño detallado** se definen los pasos detallados que recogen el algoritmo, utilizando una notación propia de la programación estructurada como puede ser el pseudocódigo o los diagramas de flujo.

La **codificación** es la traducción a un lenguaje de programación del diseño. Es decir, escribir el programa fuente en un lenguaje de programación de alto nivel que se traducirá a lenguaje máquina (programa ejecutable), utilizando un compilador o un intérprete.

La **prueba** o validación consiste en la ejecución del código para comprobar que las salidas generadas se corresponden con la solución esperada. Estos **casos de prueba** se construyen durante el análisis y representan las situaciones a las que se puede dar solución.

El **mantenimiento** consiste en realizar modificaciones al programa debido a los errores que puedan aparecer, o bien adaptaciones del código a nuevos entornos, o bien ampliaciones de las prestaciones añadiendo nuevas funciones al programa construido.

Estos pasos se muestran a continuación para un **Problema** concreto: Construir un programa que calcule e imprima en pantalla la velocidad de un cuerpo en movimiento en metros por segundo.

Análisis: Se debe decidir si se trata de un movimiento circular en cuyo caso hay que conocer la trayectoria o la velocidad angular o un movimiento rectilíneo uniforme con un movimiento entre dos puntos A y B, que será que se abordará en este caso. La información de la que se dispone es la velocidad inicial del cuerpo (v_0), la distancia entre A y B (dist) y el tiempo empleado (t). Según las leyes de la cinemática las posibles fórmulas a aplicar son: $v = v_0 + at$ o bien $v = dist/t$ para calcular la velocidad media.

Diseño: Para este problema, el diseño global consiste en decidir cual de las dos fórmulas se va a utilizar. En este caso puesto que no se tiene el valor de la aceleración se calculará la velocidad media usando el cociente entre la distancia y el tiempo.

en lenguaje natural

1. Obtener (por teclado) el valor la distancia recorrida en metros.
2. Obtener (por teclado) el valor del tiempo empleado en segundos.
3. Calcular distancia / tiempo.
4. Presentar (en pantalla) el resultado.

en Pseudocódigo

El pseudocódigo es un lenguaje artificial que ayuda a desarrollar algoritmos antes de convertirlos en programas. Se centra en la lógica del problema más que en los detalles sintácticos.

```
Algoritmo calcularVelocidad
      Escribir  "Introduce la distancia recorrida (m): ";
      Leer  distancia
      Escribir  "Introduce el tiempo empleado (s): ";
      Leer  tiempo;
      velocidad <- distancia / tiempo;
      Escribir  "La velocidad es: ", velocidad, " m/s";
FinAlgoritmo
```

La **codificación**, también llamada **implementación**, será construir un archivo `.py` en el que se escriben las instrucciones que reflejan el diseño. Existen pruebas a distintos niveles, pero el primer tipo de pruebas son las pruebas de unidad, que consisten en comprobar automáticamente el comportamiento del código utilizando otros programas Python que contienen los casos de prueba. Una de las herramientas de pruebas en Python más utilizada es **pytest**. Es fácil de usar y en combinación con Visual Studio Code, se pueden escribir, ejecutar y gestionar pruebas de forma eficiente. No obstante, el tratamiento de los casos de prueba no se incluirá en este libro.

Implementación en Python

La primera versión que resuelve el problema del **cálculo de velocidad** es:

```
print("CÁLCULO DE VELOCIDAD\n")
print("====================\n")

distancia = float(input("Introduce la distancia recorrida (m): "))
tiempo = float(input("Introduce el tiempo empleado (s): "))

try:
    velocidad = distancia / tiempo  # Si tiempo es 0,  ZeroDivisionError
    print(f"\nLa velocidad es: {velocidad:.2f} m/s")
except ZeroDivisionError:
    print("\nError:  El tiempo no puede ser 0.")
finally:
    print("\nCálculo finalizado.")
```

En este código hay algunas de las características sintácticas de Python que conviene revisar.

Los espacios en blanco dentro de las líneas no son importantes. Mientras que el mantra del *espacio en blanco importa* es válido *al principio* de las líneas, porque indican un bloque de código, el espacio en blanco *dentro* de las líneas de código Python no importa.

Por ejemplo, estas tres expresiones son equivalentes:

```
radio = diametro/2
radio = diametro / 2
radio =                    diametro          /                   2
```

El uso eficaz de los espacios en blanco puede dar lugar a un código mucho más legible. A la vista está, en este código donde se calcula la potencia con exponente de un número negativo:

```
x=10**-2          # Dificil de leer
x = 10 ** -2      # Mucho mas legible
```

Poner espacio a ambos lados de un operador binario simplifica la lectura. De igual forma, conviene separar las partes de los script mediante uso de líneas en blanco, como la separación del mensaje de salida, la lectura de datos de entrada y el cálculo, como en el script anterior.

Amigabilidad de los script. Es conveniente utilizar mensajes que faciliten la interacción con el usuario. Por ejemplo, los dos primeros `print`no son necesarios pero facilitan la ejecución. Además, es conveniente utilizar mensajes cuando se piden datos al usuario, el código funcionaría sin ellos, pero no es recomendable.

```
distancia = float(input())
tiempo = float(input())
```

Definición de un componente reutilizable. Si bien no es estrictamente necesario para facilitar la **reutilización** y la **extensibilidad** se suele recomendar enmarcar cualquier código Python dentro de un bloque de código con nombre, que no es más que una función como se verá más adelante en este libro. De esta forma, se trasforma el código en algo como:

```
def velocidad():
    print("CÁLCULO DE VELOCIDAD\n")
    print("====================\n")

    distancia = float(input("Introduce la distancia recorrida (m): "))
    tiempo = float(input("Introduce el tiempo empleado (s): "))

    try:
      velocidad = distancia / tiempo  # Si tiempo es 0,  ZeroDivisionError
      print(f"\nLa velocidad es: {velocidad:.2f} m/s")
    except ZeroDivisionError:
      print("\nError:  El tiempo no puede ser 0.")
    finally:
      print("\nCálculo finalizado.")

if __name__ == "__main__":  # indica por donde empieza la ejecución
    velocidad()
```

Cuando se quiera utilizar, es necesario importarlo:

```
import velocidad
velocidad.velocidad()
```

No obstante de cara a simplificar la lectura en la mayoría de los ejemplos de este libro no se incluye la asignación de nombre con `def`

Tipos de instrucciones de control del programa

Una estructura/instrucción de control permite especificar el orden en que se ejecutan las instrucciones de un algoritmo. DijKstra propone el uso de tres construcciones para diseñar cualquier programa que son el fundamento de la llamada programación estructurada. Dichas construcciones, tienen una estructura predecible con un único punto de entrada y de salida, facilitando al lector el seguimiento del flujo de acciones del programa:

- **Secuencia** implementa los pasos de procesamiento esenciales de cualquier algoritmo.
- **Selectiva** o **condicional** establece la posibilidad de seleccionar un conjunto de instrucciones u otro dependiendo de alguna ocurrencia lógica.
- **Iteración** o **repetición** proporciona la posibilidad de ejecutar repetidas veces un bloque de instrucciones.

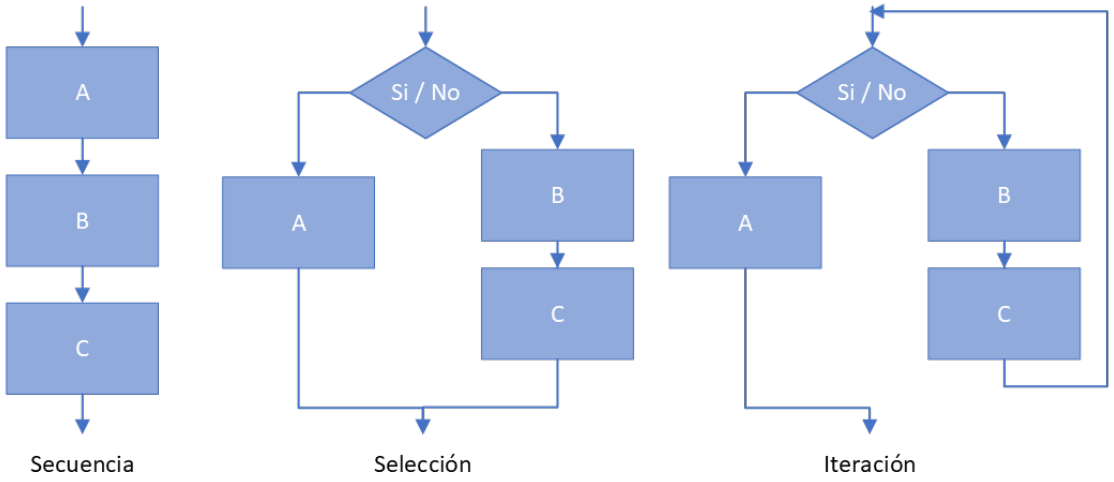

Figura 10: Instrucciones de control

La programación estructurada prohíbe la utilización de la instrucción para la transferencia incondicional del control (`goto` o `break`: instrucción de bifurcación incondicional), aunque cabe decir que Python intensifica su uso en determinadas situaciones, alejándose así un tanto de los principios estructurados.

Instrucción secuencial

Los bloques de código se ejecutan uno tras otro y las instrucciones dentro de cada bloque también, siguiendo el orden físico de escritura. En Python el fin de cada instrucción es el fin de línea. La salida de cada instrucción o cada bloque es la entrada de la siguiente. Un bloque de código es considerado como una única instrucción secuencial respecto al resto de bloques o instrucciones de su mismo nivel.

Ejercicio resuelto: Envasado

Una empresa de envasado automático de aceite dispone de diversos tipos de **envases** con capacidades de 50, 20, 10, 5, 2 y 1 litros, respectivamente. Construir un programa que dado por teclado un **número entero** de litros de aceite a envasar, determine el **menor número de envases** completos necesarios e indique el número de envases de cada tipo, presentándolos en pantalla.

Los datos son de **entrada** son: Número de litros

Los datos son de **salida** son:

- Número de envases de 50, n50
- Número de envases de 20, n20
- Número de envases de 10, n10
- Número de envases de 5, n5
- Número de envases de 2, n2
- Número de envases de 1, n1

Se estudian casos particulares para identificar las situaciones correctas e incorrectas en el envasado que definen la estrategia de solución del problema.

Por ejemplo, si los litros son 316 se presentan distintas situaciones:

- n50=n20=n10=n5=n2=0 y n1=316. No es correcto, lo es se busca el mínimo número de envases
- n50=6 n20=0 n10=1 n5=1 n2=0 n1=1 Solución correcta

Implementación

```
# Cálculo de número de envases

litros = int(input("Cuantos litros quieres envasar= "))

n50 = litros // 50
litros = litros % 50
n20 = litros // 20
litros = litros % 20
n10 = litros // 10
litros = litros % 10
n5 = litros // 5
litros = litros % 5
n2 = litros // 2
n1 = litros % 2

print("El número de envases de 50 litros es:", n50)
print("El número de envases de 20 litros es:", n20)
print("El número de envases de 10 litros es:", n10)
print("El número de envases de 5 litros es:", n5)
print("El número de envases de 2 litros es:", n2)
print("El número de envases de 1 litros es:", n1)
```

Ejercicio resuelto: Producto vectorial

Construir un programa que lea por teclado los componentes espaciales de dos vectores y que calcule e imprima por pantalla la suma de los dos vectores, su producto escalar y su producto vectorial.

Información de entrada:

- Componentes vector 1: (v1x,v1y,v1z) – reales
- Componentes vector 2: (v2x,v2y,v2z) – reales

Información de salida:

- Suma de los dos vectores: (sx,sy,sz) – real
- Producto escalar: e – real
- Producto vectorial: (px,py,pz) – real

Implementación:

```
# Cálculo del producto escalar y vectorial

print("Introduce los componentes del primer vector:")
v1x, v1y, v1z = [float(input(f'Dame la componente del vector {coef}: '))\
     for coef in ('eje X', 'eje Y', 'eje Z')]

print("Introduce los componentes del segundo vector")
v2x, v2y, v2z = [float(input(f'Dame la componente del vector {coef}: '))\
     for coef in ('eje X', 'eje Y', 'eje Z')]

sx = v1x + v2x
sy = v1y + v2y
sz = v1z + v2z

e = v1x * v2x + v1y * v2y + v1z * v2z

px = v1y * v2z - v1z * v2y
py = v1z * v2x - v1x * v2z
pz = v1x * v2y - v1y * v2x

print("La suma de los vectores es:",sx,",",sy,",",sx)
print("El producto escalar es:",e, end=" ")
print("y el vectorial es:",px,"i,",py,"j,",px,"k")
```

Instrucción selectiva. Condicionales

Es un tipo de estructura de control que ofrece la posibilidad de ejecutar las instrucciones en un orden lógico diferente del orden físico. Se evalúa una condición (expresión lógica) y en función del resultado de la misma se ejecuta un bloque de instrucciones u otro. Hay diversos tipos de estructuras selectivas: simple, doble, anidadas, encadenadas y múltiple (no existe como tal en Python, se emularía sobre las anteriores).

Selectiva simple

Se ejecuta una determinada acción cuando se cumple la condición especificada. Su funcionamiento comienza con la evaluación una condición, si ésta es verdadera entonces ejecuta un bloque de instrucciones; y si es falsa, no se ejecuta nada.

```
if (condición a evaluar): # Por ejemplo X >= 10
    # Bloque de instrucciones si se cumple la condición....
    # Solo se ejecutará si la condición es verdadera
    # Notesé que presenta indentación por ser un bloque
# Bloque de Instrucciones restante del programa....
# Se ejecuta siempre, pues está fuera de la condición
# se pueden quitar  paréntesis en condicion, no recomendable por claridad
```

Por ejemplo:

```
billete = 30              # Precio base del billete
edad = int(input("¿Cuál es tu edad? "))

if edad > 64:     # Aplicar descuento del 30% si tiene 65 años
    billete *= 0.7

print(f"El billete cuesta {billete:.2f} euros.")
```

Selectiva doble

Es la estructura que permite elegir entre dos opciones o alternativas posibles, en función del cumplimiento o no de una determinada condición. Primero se evalúa la condición y si ésta es verdadera, se ejecuta un bloque de instrucciones especificado justo después de la condición (habitualmente llamada parte `if`); y en caso contrario, es decir cuando la condición es falsa, se ejecuta otro bloque diferente de instrucciones, especificado después de la palabra `else`.

```
if (condición a evaluar): # Por ejemplo Z <= 50
    # Bloque de instrucciones si se cumple la condición....
    # Solo se ejecutará si la condición es verdadera
else:
    # Bloque de instrucciones si NO se cumple la condición....
    # Solo se ejecutará si la es falsa
# Bloque de Instrucciones restante del programa....
# Se ejecuta siempre, pues está fuera del if else
```

Por ejemplo:

```
password = input("Ingrese la contraseña: ")
if (password == "miClave"):
  print("Contraseña correcta. Te damos la bienvenida.")
else:
  print("Contraseña incorrecta.")
```

O bien:

```
edad = int(input("Ingrese su edad: "))
if edad >= 18:
    print("Eres mayor de edad. Bienvenido.")
else:
    print("Eres menor de edad. Acceso denegado.")
```

Selectivas anidadas

El funcionamiento se basa en evaluar la primera condición y, si se cumple, ejecutar su bloque de código donde puede haber otra estructura condicional que refine aún más la decisión. Permite manejar escenarios más complejos. Hay que cuidar la claridad del código para evitar confusión y saber exactamente qué bloque incluye, qué está dentro de cada parte. Por ejemplo:

```
password = input("Ingrese la contraseña: ")
if (len(password) >= 8):
    print('Tu contraseña es suficientemente larga.')

    if (password == 'miClaveSegura'):
        print("Además es la contraseña correcta.")
    else:
        print("Pero es incorrecta.")
else:
    print('Tu contraseña es muy corta e insegura.')

    if (password != 'miClaveSegura'):
        print("Además, es incorrecta (por supuesto).")
```

Selectivas encadenadas if-elif-else:

Permite evaluar múltiples condiciones de forma ordenada sin anidamiento de `if`. Se emplea cuando hay varias opciones posibles y solamente una de ellas debe ejecutarse. Por ejemplo:

```
x = -15

if x == 0:
    print(x, "es cero")
elif x > 0:
    print(x, "es positivo")
elif x < 0:
    print(x, "es negativo")
else:
    print(x, "es difícil que lo hayas visto alguna vez...")
```

Selectiva múltiple

Esta estructura, no presente en Python, evalúa una expresión que puede tomar n valores distintos (1, 2, 3,. . . , N) y según este valor el algoritmo seguirá un determinado camino entre n posibles. En C o Java se corresponde con la instrucción `switch`. En Python se pueden utilizar una cadena de `if` para recoger este comportamiento:

```
dia=int(input("Qué día es?"))

if dia == 1:
     print('lunes')
if dia == 2:
    print('martes')
if dia == 3:
    print('miércoles')
if dia == 4:
    print('jueves')
if dia == 5:
    print('viernes')
if dia == 6:
    print('sábado')
if dia == 7:
    print('domingo')
if dia < 1 or dia > 7:
    print('error')
```

Ejercicio resuelto: Límite de velocidad

Construir un programa que lea por teclado un valor de velocidad en Km/hora y que escriba en pantalla un mensaje si la velocidad supera los 120 Km/h y en cuánta cantidad los supera.

Información de entrada:

- Velocidad: v es real (>0)
- Límite de velocidad=120 es real (>0)

Información de salida:

- Mensaje que ha superado velocidad máxima
- Exceso de velocidad: exceso es real

Se usará una selectiva simple controlada por el dato lógico (v $>$ 120).

Implementación,

```
LIMITE=120     # Al ser constante por legibilidad se usan mayúsculas
v=float(input("la velocidad actual es: "))
if (v > LIMITE):
  exceso = v - LIMITE
  print("Sobrepasado limite de velocidad en", exceso," Km/h")
```

Implementación alternativa,

```
LIMITE = 120   # Al ser constante por legibilidad se usan mayúsculas
try:
    v = float(input("la velocidad actual es: "))
except ValueError:
        print("El valor ingresado no es un numero válido.")
else:
  if (v > LIMITE):
        exceso = v - LIMITE
        print("Sobrepasado limite de velocidad en", exceso," Km/h")
```

`ValueError` es una excepción generara automáticamente por el lenguaje para recoger que se ha producido un error en la trasformación de cadena a entero, por ejemplo si se escriben letras en lugar de números.

Validación de datos simple. Para evitar errores y mejorar la robustez del programa se desea que el código valide los datos de entrada, es decir, no se genere error y funcione correctamente cuando el usuario introduce una entrada incorrecta. Por ejemplo, si se da un número negativo cuando se espera un número positivo, o bien una entrada errónea escribiendo letras cuando se pide un número. Para gestionar posibles errores se deben incluir comprobaciones adicionales o incluso bloques `try/except`.

Dado el código que lee un entero positivo:

```
entero = int(input('Introduzca  un entero positivo: '))
print(entero)
```

Se puede validar con `is.numeric` o `is.digit`:

```
if entero.isdigit():  # Verificamos si es un número entero positivo
    entero = int(entero)
    if entero >= 0:  # Validamos que sea un número positivo
        print("Procesar entero")
    else:
        print("El número debe ser positivo")
else:
    print("Entrada no válida. Debe ser un número.")
```

O bien dentro de una excepción:

```
try:
     entero=int(input('Introduzca  un entero positivo: '))
     if entero >=0:
        ## procesar entero
        print  ("procesamos entero")
     else:
        print("dato no valido")
except ValueError:  # generada si la trasformación es errónea
    print("dato erróneo")
```

Las operaciones de validación no siempre se incluyen en los scripts de código. Se ha de llegar a un **compromiso** entre la capacidad de respuesta a errores (robustez) y complejidad. A más validación, mayor complejidad y menor mantenibilidad y legibilidad.

Ejercicio resuelto: Máximo de dos temperaturas

Construir un programa que **lea** por teclado **dos** valores de **temperatura** en grados centígrados y que determine e **imprima** en pantalla el **mayor** valor.

- Entrada: Dos temperaturas: t1, t2 reales
- Salida: El mayor valor

Dos alternativas controladas por un dato, es decir, una selectiva doble.

```
t1 = float(input("Introduce la primera temperatura:"))
t2 = float(input("Introduce la segunda temperatura:"))
if t1 < t2:
    tmax = t2
else:
    tmax = t1
print(" La mayor temperatura introducida es:", tmax)
```

En este ejemplo, se podría incluir la validación de la temperatura en dos sentidos: que sea un número sin caracteres alfanuméricos controlando la conversión (`ValueError`) o también que sea una temperatura válida mayor que el mínimo absoluto (-273º). Pero si se informa al usuario que este script solo funciona correctamente con números mayores de -272 no sería necesario incluir la validación, es responsabilidad del usuario dar los datos correctos.

```
t1 = input("Introduce la primera temperatura: ")
try:
    t1 = float(t1)
    if t1 <= -273:
        print("La temperatura debe ser mayor que -273°C.")
    else:
        t2 = input("Introduce la segunda temperatura: ")
        try:
            t2 = float(t2)
            if t2 <= -273:
                print("La temperatura debe ser mayor que -273°C.")
            else:
                if t1 < t2:
                    tmax = t2
                else:
                    tmax = t1
                print("La mayor temperatura introducida es:", tmax)
        except ValueError:
            print("Por favor, una la segunda temperatura válida")
except ValueError:
    print("Por favor, introduce un número válido para la primera temperatura")
```

Esta versión es mucho más compleja y en cierto sentido oculta la funcionalidad del script, tal como se ha indicado, se debe llegar a un compromiso que mejore la legibilidad y mantenibilidad.

Ejercicio resuelto: ¿Cuál es la nota?

Dada una nota numérica mayor que 0 de un alumno/a indicar su calificación según la siguiente tabla:

Nota	0 - 4.9	5 - 6.9	7 - 8.9	9 - 9.9	10
Calificación	Insuficiente	Aprobado	Notable	Sobresaliente	Matrícula de honor

```
nota = float(input("Introduce la nota del alumno: "))

if nota < 0:
    print("La nota no puede ser negativa.")
elif nota < 5:
    print("Insuficiente")
elif nota < 7:
    print("Aprobado")
elif nota < 9:
    print("Notable")
elif nota < 10:
    print("Sobresaliente")
elif nota == 10:
    print("Matrícula de honor")
else:
    print("La nota ingresada no es válida.")
```

Ejercicio resuelto: Ecuación de segundo grado

Construir un programa que calcule e imprima en pantalla las raíces de la ecuación de segundo grado: $Ax^2 + Bx + C = 0$, dados por teclado los coeficientes A, B y C.

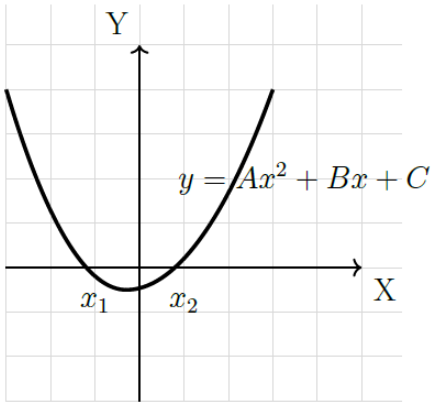

Figura 11: Representación de una parábola

La entrada serán los coeficientes del polinomio: A, B y C, reales.

Como salida se darán las soluciones: x1 y x2, reales

$x_1 = \frac{-B+\sqrt{(B^2-4AC)}}{2A}$, $x_2 = \frac{-B-\sqrt{(B^2-4AC)}}{2A}$

No obstante, existen diversas situaciones que se tienen que contemplar:

Se debe contemplar el valor del *discriminante* y el valor de A:

- Si *discriminante* menor que cero no hay solución real, pero sí compleja.
- Si A vale cero, tenemos una ecuación de primer grado con $x = -C/B$, que supone un problema si B es cero

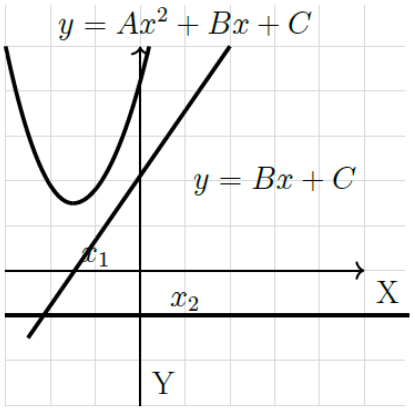

Figura 12: Casos a contemplar

La estrategia de solución es resolver el problema controlando las condiciones basadas en los valores de los coeficientes, siguiendo estos pasos:

- Se leen los coeficientes.
- Se estudia el valor de A para saber si se trata de en una ecuación de segundo grado.
- Si es de segundo grado se calcula el discriminante.
- Según el valor del discriminante se define el resultado.
- Si no es una ecuación de segundo grado — No esta especificado pero se puede optar o no por dar una solución, si no es de segundo grado no entra dentro del objetivo del programa

A	B	C	Resultado
<>0	-	-	Estudiar discriminante
0	-	-	No de segundo grado
0	0	0	No ecuación
0	0	<>0	Imposible

Implementación

```
from math import sqrt

print("¡Hola! Vamos a resolver una ecuación de segundo grado:")
print("-----------------------------------------------------")
print("    ax² + bx + c = 0\n")

a, b, c = [float(input(f"Dame el coeficiente {coef}: "))\
     for coef in ("a", "b", "c")]

if a == 0:
    if b == 0:
        if c == 0:
            print("0=0")
```

```
        else:
            print("Imposible")
    else:
        x = -c / b
        print("una única solución ", x)
else:
    discriminante = b * b - 4 * a * c
    if discriminante < 0:
        print(f"La ecuación no tiene soluciones reales.")
        x1 = (-b + complex(discriminante) ** (1 / 2)) / (2 * a)
        x2 = (-b - complex(discriminante) ** (1 / 2)) / (2 * a)
        print("La ecuación tiene dos soluciones complejas conjugadas:")
        print("x1 =", x1)
        print("x2 =", x2)
    else:
        raiz = sqrt(discriminante)
        x_1 = (-b + raiz) / (2 * a)
        if discriminante != 0:          # se comprueba si hay otra solución
            x_2 = (-b - raiz) / (2 * a)
            print(f"Las soluciones son {x_1} y {x_2}.")
        else:
            print(f"La única solución es x = {x_1}")
```

Pero ¿es necesario siempre contemplar todas las situaciones? En este caso contemplar a=0 o b=0 o c=0. Depende de la especificación (análisis) del problema, que define el "contrato" de uso del código. Si a quién use este código se le informa de que solo funciona correctamente con ecuaciones de segundo grado completas el resultado podría simplificarse:

```
from math import sqrt
# No se contemplan los casos de no tener una ecuación válida.

a, b, c = [float(input(f"Dame el coeficiente {coef}: ")) \
    for coef in ("a", "b", "c")]

discriminante = (b**2) - (4 * a * c)

if discriminante > 0:
    x1 = (-b + sqrt(discriminante)) / (2 * a)
    x2 = (-b - sqrt(discriminante)) / (2 * a)
    print("La ecuación tiene dos soluciones reales distintas:")
    print("real x1 =", x1)
    print("real x2 =", x2)
elif discriminante == 0:
    x = -b / (2 * a)
    print("La ecuación tiene una solución real doble:")
    print("x =", x)
else:
    x1 = (-b + complex(discriminante)**(1/2)) / (2*a)
    x2 = (-b - complex(discriminante)**(1/2)) / (2*a)
    print("La ecuación tiene dos soluciones complejas conjugadas:")
    print("x1 =", x1)
    print("x2 =", x2)
```

Esta misma situación se plantea para la validación de los datos de entrada, por ejemplo, si se quiere incluir la validación de que si se introducen caracteres alfanuméricos no se ejecuten los

cálculos :

```
from math import sqrt

print('¡Hola! Vamos a resolver una ecuación de segundo grado:')
print('-----------------------------------------------------')
print('    ax² + bx + c = 0\n')

try:
    a = float(input('Dame el coeficiente a: '))
    b = float(input('Dame el coeficiente b: '))
    c = float(input('Dame el coeficiente c: '))
except ValueError:
    print("Error: Los coeficientes deben ser valores numéricos.")
else:
    if a == 0:
        if b == 0:
            if c == 0:
                print("0=0")
            else:
                print("Imposible")
        else:
            x = -c / b
            print("una única solución ", x)
    else:
        discriminante = b * b - 4 * a * c
        if discriminante < 0:
            print(f'La ecuación no tiene soluciones reales.')
            x1 = (-b + complex(discriminante) ** (1 / 2)) / (2 * a)
            x2 = (-b - complex(discriminante) ** (1 / 2)) / (2 * a)
            print("La ecuación tiene dos soluciones complejas conjugadas:")
            print("x1 =", x1)
            print("x2 =", x2)
        else:
            raiz = sqrt(discriminante)
            x_1 = (-b + raiz) / (2 * a)
            if discriminante != 0:
                x_2 = (-b - raiz) / (2 * a)
                print(f'Las soluciones son {x_1} y {x_2}.')
            else:
                print(f'La única solución es x = {x_1}')
```

Ejercicio resuelto: Mayor de tres temperaturas

Construir un programa que calcule e imprima la mayor de tres temperaturas en grados Centigrados introducidas por teclado.

- Información de entrada: los valores de las tres temperaturas, t1, t2, t3 reales.
- Información de salida: La temperatura mayor

Una estrategia de solución es la utilización de una selectiva simple con condicionales dobles

- Si t1 es mayor o igual que t2 y t1 es mayor o igual que t3 el máximo es t1
- Si t2 es mayor o igual que t1 y t2 es mayor o igual que t3 el máximo es t2
- Si t3 es mayor o igual que t1 y t3 es mayor o igual que t2 el máximo es t3

Implementación

```
entrada = input("Dime tres temperaturas separadas por espacios (t1 t2 t3): ")
temperaturas = entrada.split()
t1 = float(temperaturas[0])
t2 = float(temperaturas[1])
t3 = float(temperaturas[2])

if (t1 >= t2) and (t1 >= t3):
  tmax = t1
if(t2 >= t1) and (t2 >= t3):
  tmax = t2
if (t3 >= t1) and (t3 >= t2):
  tmax = t3

print("\nTemperatura maxima: ", tmax);
```

Implementación alternativa con la utilización de una selectiva doble

```
if (t1 >= t2):
  if (t1 >= t3):
    tmax = t1
  else:
    tmax = t3
else:
  if (t2 >= t3):
    tmax = t2
  else:
    tmax = t3
print("\nTemperatura maxima: ", tmax);
```

Pero ¿qué pasa si aumenta el número de temperaturas? Se incrementa el número de **if** anidados, lo que hace un diseño poco escalable. Existe una alternativa de diseño que usa una misma variable **t** para ir almacenando sucesivamente el último dato introducido por teclado (los datos leídos anteriormente se pierden) y almacena en **tmax** la mayor temperatura de las introducidas hasta ese momento.

```
# La lectura no se hace al princicio para las tres variables
t=float(input("Dime una temperatura "))
tmax = t

# la lectura y el if es el que se repite en función del número de temperaturas
t = float(input("Dime una temperatura "))
if (t > tmax):
  tmax = t

t = float(input("Dime una temperatura "))
if (t > tmax):
  tmax = t

print(tmax)
```

Como mejora y como ejemplo de *variable indicador* si se incluye una variable se puede marcar cual es la mayor de las temperaturas en términos del orden de lectura.

```
t=float(input("Dime una temperatura "))
tmax=t
cual=1

t=float(input("Dime una temperatura "))
if (t>tmax):
  tmax=t
  cual=2

t=float(input("Dime una temperatura "))
if (t>tmax):
  tmax=t
  cual=3

print("máxima",tmax,"en posición: ", cual)
```

Finalmente, se incluye una solución muy al estilo Python, que aplica el principio de la reutilización de código. Se usa una lista aun a sabiendas de que se desperdicia espacio, puesto que no es necesario utilizarla. Se genera un código más corto pero más ineficiente, donde los condicionales están ocultos en la llamada a `max()`.

```
entrada = input("Dime tres temperaturas separadas por espacios (t1 t2 t3): ")
temperaturas = list(map(float, entrada.split(" ")))

tmax = max(temperaturas)
print("La mayor temperatura introducida es:", tmax)
```

Ejercicio resuelto: Tarifa de taxi

La tarifa de un taxi parte una bajada de bandera fija de 30€ si no se sobrepasan los 30 km de recorrido. Para más de 30 km, sé que si no se sobrepasan los 100 km se añaden 15€ por km que exceda de los 30 km. Si se sobrepasan los 100 km, se añade al anterior el precio de 10€ por cada km adicional que exceda de los 100 km. Construir un programa que pida por teclado los kilómetros recorridos y calcule e imprima en pantalla el total a pagar según la tarifa anterior.

- Información de Entrada:
  - Kilómetros recorridos: km, real (>0)
  - Cantidad fija: FIJO=30 (constante)
  - €/km entre 30-100 km: T1=15 (constante)
  - €/km más de 100 km: T2=10 (constante)
- Información de Salida: Total a pagar como real

```
FIJO = 30
T1 = 15
T2 = 10

km = float(input("Introduzca kms recorridos: "))
if (km <= 0):
  print("Error: datos de E no validos");
```

```
elif (km <= 30):
    total = FIJO
elif (km <= 100):
  total = FIJO + (km - 30) * T1
else:
  total = FIJO + (100 - 30) * T1 + (km - 100) * T2

print("\nTotal a pagar:", total)
```

Instrucción iterativa. Bucles

Los bucles o iteraciones son otra de las estructuras de control que permiten alterar el flujo normal (secuencial) de un programa. Se utiliza para **repetir una secuencia de instrucciones** un número de veces determinado o no. Esto no sólo hace que el código sea más limpio y manejable, sino que también ahorra tiempo y esfuerzo en la programación, evitando la duplicación de instrucciones y facilitando el mantenimiento del código. A cada ejecución de la secuencia de instrucciones se le denomina **iteración**.

Este concepto de iteración aparece, por ejemplo, si se está desarrollando un programa que necesita procesar grandes cantidades de datos, o realizar una operación como verificar cada número en una lista para encontrar los valores que cumplen con ciertas condiciones.

Una solución para escribir 5 veces `Hola Mundo` sería

```
print("Hola Mundo")
print("Hola Mundo")
print("Hola Mundo")
print("Hola Mundo")
print("Hola Mundo")
```

o bien podría hacerse aprovechando las propiedades que tiene la función `print`:

```
print(5*"Hola Mundo\n")
```

Pero esto sólo pasa con `print`, si cada iteración es un código complejo o bien supone mas de una instrucción, al solución utilizar alguna construcción que indique cuántas veces se ha de hacer:

```
for contador in range (0, 5):
    print("Hola Mundo")
```

Elementos de un bucle

Un bucle tiene dos elementos básicos: condición y cuerpo del bucle. La **condición** que es la expresión lógica que activa la repetición del bucle o la salida de este, por ejemplo, si se han ejecutado un cierto número de veces el cuerpo del bucle. El **cuerpo del bucle** son el conjunto de instrucciones que se ejecutan de forma repetitiva, cada ejecución del cuerpo del bucle se llamará

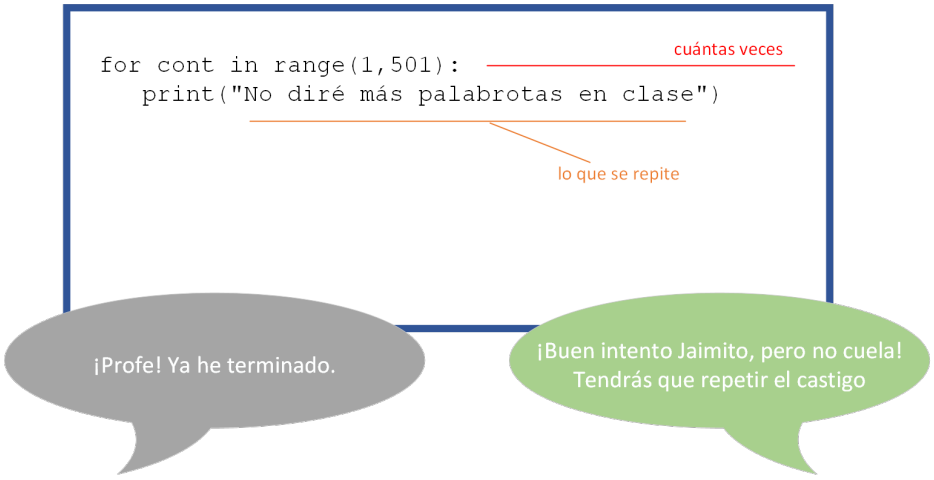

Figura 13: Solución inteligente de Jaimito

iteración. El funcionamiento consiste en evaluar una **condición**, y en función del resultado, se vuelve a repetir la secuencia de instrucciones o bien se acaba el bucle (salir del bucle).

Python incluye únicamente dos tipos de bucle: **while** y **for** de definen un bloque de código desde el inicio de los dos puntos (`:`).

```
while condicion:
    cuerpo_del_bucle

for elemento in iterable:
    cuerpo_del_bucle
```

Bucle while

La ejecución de una estructura de control `while` comienza evaluando la condición que sigue a la palabra clave. Si el resultado es `True`, se ejecuta el cuerpo del bucle, y una vez ejecutado, el proceso se repite volviendo a evaluar la condición y, si sigue siendo cierta, el cuerpo del bucle se ejecuta otra vez. Este ciclo continúa mientras la condición se mantenga verdadera. Si el resultado de una evaluación es `False`, el cuerpo del bucle no se ejecuta y se pasa a la siguiente parte del programa, es decir, la siguiente instrucción o bloque tras el bloque `while`.

La variable o las variables que aparezcan en la condición se suelen llamar **variables de control**. Las variables de control deben definirse antes del bucle while y modificarse en el cuerpo el bucle. En el bucle while el número de iteraciones no está definida antes de empezar el bucle, por ejemplo porque depende de los datos los proporciona el usuario o de los cambios realizados en el cuerpo del bucle.

Si se quiere realizar la raiz cuadrada solamente de números positivos:

```
numero = -10000                  # valor basura nunca se usará
while numero < 0 :
    numero = int(input("Dame un número positivo: "))

print(numero**0.5)
```

O bien, para calcular la temperatura máxima registrada hasta que el usuario introduce un valor por debajo del cero absoluto (-273.15°C), y siempre que se den valores válidos de temperaturas.

```
TLIM = -999                  # Valor centinela
temperatura = 0              # Valor basura, nunca se usará
temperatura_maxima = TLIM    # Inciialización  con un valor no válido

while temperatura > -273.15:
    temperatura = float(input("Ingrese una temperatura (°C): "))

    if temperatura > temperatura_maxima:
        temperatura_maxima = temperatura

# Si la temperatura máxima sigue siendo -999, no se registraron datos
if temperatura_maxima == TLIM:
    print("No se han registrado datos válidos.")
else:
    print(f"La temperatura máxima registrada fue: {temperatura_maxima}°C")
```

Ejercicio resuelto: Cálculo de velocidad para más de un caso

Construir un programa que calcule la velocidad para diversos valores de distancia y tiempo. El número de veces que se calcula dependerá de los deseos del usuario.

```
print("CÁLCULO DE VELOCIDAD\n")

repetir = True
while  repetir:
    distancia = float(input("Introduce la distancia recorrida (m): "))
    tiempo = float(input("Introduce el tiempo empleado (s): "))
    try:
        velocidad = distancia / tiempo  # lanza ZeroDivisionError
        print(f"\nLa velocidad es: {velocidad:.2f} m/s")
    except ZeroDivisionError:
        print("\nError: El tiempo no puede ser 0.")
    finally:
        print("\nCálculo finalizado.")

    respuesta = input("Desea realizar otra operación(s/n)?\n")
    repetir = (respuesta[0] != 'n') and (respuesta[0] != 'N')
    # salvo empiece por N o n repite el bucle
```

Ejercicio resuelto: Del revés

Programa que calcula un número escrito al revés. Dado n=12345, la salida será 54321.

```
numero = 10098765        # podría leerse de teclado
salida = 0
```

```
while numero != 0:
  resto = numero % 10
  salida = salida *10 + resto
  numero = int(numero/10)
print (salida)
```

Ejercicio resuelto: Sumar Pares

Construir un programa que pida al usuario una serie de números enteros y que sume los números pares acabando cuando se introduzca un 0.

```
suma = 0
numero = -2000    # cualquier valor no 0

while numero != 0:
  numero = int(input("Introduce el siguiente número "))
  if ((numero % 2) == 0):
    suma =suma + numero
print("suma", suma)
```

Para evitar la asignación inicial a un valor distinto de cero se puede utilizar lectura adelantada:

```
numero = int(input("Introduce el primer número "))
suma = 0

while numero != 0:
  if ((numero % 2) == 0):
    suma = suma + numero
  numero = int(input("Introduce en siguiente número "))
print("suma", suma)
```

Ejercicio resuelto: Sumar cubos

Construir un programa que pida al usuario el valor final sobre el que se calculen la suma de los cubos de los primeros números naturales hasta el número introducido.

```
# Suma de cubos
n = int(input("Ultimo número a considerar:"))
sumaAcumulada = 0
i = 1
while i <= n :
    c = i ** 3
    sumaAcumulada = sumaAcumulada + c
    i = i + 1
print("La suma es:", sumaAcumulada)
```

Bucle for

Un **iterable** es cualquier objeto en Python que se puede recorrer o iterar, es decir, que puede devolver uno de sus elementos a la vez en un bucle. Ejemplos comunes de iterables son las listas, las tuplas, los diccionarios y las cadenas de texto.

El bucle `for` se utiliza para tratar cada uno de los elementos de un objeto iterable y ejecutar el bloque de código definido en el cuerpo del bucle por cada elemento del iterable, se suele llamar **recorrer** un iterable. La condición del bucle está basada en el tamaño del iterable, lo que significa que el bucle terminará cuando haya recorrido todos los elementos del objeto iterable.

```
for elemento in iterable:
    cuerpo_del_bucle
```

En cada iteración `elemento` es la variable que toma el valor del miembro del iterable tratado ese momento:

```
for n in [2, 3, 5, 7]:
    print(f"Ahora n vale {n} y su cuadrado {n ** 2}")
```

Si un objeto es iterable, puede usarse a la derecha de un bucle `for`. La variable `elemento` toma automáticamente el valor de cada elemento en el iterable, eliminando la necesidad de acceder manualmente a los valores mediante índices (`[]`).

Esta construcción supone una gran ventaja frente a otros lenguajes de programación que obligan a la indexación en los recorridos, generando soluciones menos legibles. La sintaxis se asemeja bastante al lenguaje natural pero en inglés, sería algo así como decir "para cada elemento llamado c de cadena . . . ". No todo es iterable en Python, pero las colecciones (lista, tuplas, conjuntos y diccionarios) y las cadenas si lo son.

Por ejemplo, para recorrer una lista de palabras indicando cuantos caracteres tiene palabra en la lista:

```
palabras = ['gato', 'ventana', 'defenestrado']
for texto in palabras:
    print(texto, len(texto))
```

Finalmente, para tratar la lista de los días de la semana e indicar si es o no laborable, matizando y animando en miércoles.

```
semana = ['Lunes','Martes','Miércoles','Jueves','Viernes','Sábado','Domingo']

for dia in semana:
    if dia == 'Miércoles':
        print('Hoy es', dia + '.', '¡Tienes que madrugar más!')
    elif dia == 'Sábado' or dia == 'Domingo':
            print('Hoy es', dia + '.', '¡Puedes dormir más!')
    else:
        print('Hoy es', dia + '.', 'Es un día laborable.')
```

Ejercicio resuelto: Sumar serie

Calcular la suma de los siguientes números 5, 8, 17, 12; e imprimir en pantalla el resultado.

```
suma = 0
mis_numeros = [5, 8, 17, 12]

for numero in mis_numeros:
    suma = suma + numero
print("- La suma de los números es: ", suma, end="")
```

Solución alternativa usando listas

```
mis_numeros = [5, 8, 17, 12]
sum(mis_numeros)
```

Para evitar problemas si se va a modificar la lista sobre la que se itera en el cuerpo de bucle, puesto que se trabaja con punteros a memoria, se debe generar una copia que sea la que se recorre.

Por ejemplo, si se quiere construir un programa que elimine los números pares de una lista dada:

```
posicion = 0
mis_numeros = [5, 8, 17, 12]
for numero in mis_numeros.copy():
    if (numero%2) == 0:
        mis_numeros.pop(posicion)
    else:
        posicion = posicion + 1
print(mis_numeros)
```

O bien construyendo una nueva lista resultado de las operaciones definidas en el cuerpo del bucle:

```
posicion = 0
mis_numeros = [5, 8, 17, 12]
nuevo = []
for numero in mis_numeros:
    if (numero % 2) != 0:
        nuevo.append(numero)
print(nuevo)
```

Uso de `range` en `for`

Cuando no hay un iterable y se quiere hacer un `for` para ejecutar un número de veces las instrucciones del cuerpo del bucle, pero sin que exista de partida un iterable, se debe tener una variable **contador** que vaya cambiando (incrementándose o decrementándose) según la iteración en la que se está, pero sin estar ligado a un iterable.

En Python se utiliza la función `range` para generar este contador. Cuando solo se proporciona un argumento `range` devuelve una secuencia de números, incrementada en 1, comenzando desde 0 hasta el número anterior al argumento proporcionado.

```
# un for de 0 a 3, para imprimir esos valores
for i in range(4):
    print(i)
```

Es posible hacer que el rango comience en otro número que no sea cero. Cuando se pasan dos argumentos a la función `range()` se tiene el número inicial y el final de la secuencia (incluso con valores negativos). Con un tercer argumento se fija el incremento de salto entre los elementos.

```
print(list(range(5,10)))          # [5, 6, 7, 8, 9]
print(list(range(-50,-45)))       # [-50, -49, -48, -47, -46]
print(list(range(-4,4)))          # [-4, -3, -2, -1, 0, 1, 2, 3]
print(list(range(0,11,2)))        # [0, 2, 4, 6, 8, 10]
print(list(range(-50,50,10)))     # [-50, -40, -30, -20, -10, 0, 10, 20, 30, 40]
print(list(range(100,-10,-10))) # [100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 0]
```

Para iterar sobre los índices de una secuencia, se pueden combinar `range()` y `len()`.

```
frase = ['El', 'caballo', 'blanco', 'de', 'Santiago']
for i in range(len(frase)):
    print(i, frase[i])
```

El objeto devuelto por `range()` se comporta como si fuera una lista `range(10)`, pero no lo es. Es un objeto que devuelve los elementos sucesivos de la secuencia indicada cuando se itera sobre ella, de esta forma se ahorra espacio de memoria.

Si sólo se usa `range` para contar, es decir, no se necesita usar la variable contador en el cuerpo de bucle, se puede usar el símbolo _ ahorando la necesidad de almacenar una variable. Si se quiere escribr 5 veces "Hola Mundo"

```
for _ in range (0, 5) :
    print("Hola Mundo")
```

Ejercicio resuelto: Tabla multiplicar

Construir un programa que pregunte por la tabla de multiplicar que se desea generar y la presente en pantalla.

```
n = int(input("Introduzca el valor de la tabla de multiplicar a obtener: "))

print(" TABLA DEL", n);
print("==============");

for i in range(1, 11) :
   print(f" {n} * {i} = {n * i}")
```

Anidamiento de bucles

En el cuerpo de bucle puede aparecer cualquier instrucción o bloque de instrucciones, incluso otro bucle. Este anidamiento implica que el bucle interior debe acabar antes de seguir con la siguiente instrucción en el cuerpo del bucle exterior. Es decir, el bucle interior está totalmente incluido en el bucle exterior.

Ejercicio resuelto: Cuadrado de 5 X 5 asteriscos

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

```
for _ in range(0, 5):
    for _ in range(0, 5) :
        print("*", end=" ")
    print("\n", end="")
```

Alternativa preguntando el alto y el ancho de un rectángulo.

```
ancho=int(input("Ancho: "))
alto=int(input("Alto: "))

for _ in range(0, alto):
    for _ in range(0, ancho) :
        print("*", end=" ")
    print("\n", end="")
```

Uso de declaraciones `break`, `continue` y `else` en iteraciones

En Python, salir (`break`), continuar (`continue`) y si no (`else`) son herramientas para controlar el flujo de ejecución de los bucles. Sin embargo, no son del gusto de los programadores estructurados puesto que implican saltos incondicionales en la ejecución de un programa.

La instrucción `break` rompe el bucle sin importar cuántas repeticiones queden. Se usa típicamente cuando se cumple una condición que hace innecesario seguir ejecutando el bucle. Por ejemplo, el bucle siguiente busca números primos en el rango de 2 a 10:

```
for n in range(2, 11):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'igual', x, '*', n//x);
            break
    else:
        # bucle termina sin encontrar el factor
        print(n, 'es un número primo');
```

Los "buenos" programadores no abusan de `break` porque no está disponible en todos los lenguajes. Es mejor utilizar una variable booleana que marque la situación y se incorpore como condición al bucle, utilizando un bucle `while`.

La instrucción `continue` salta a la siguiente iteración del bucle, omitiendo el resto del código del cuerpo del bucle para esa iteración específica, y pasa a la siguiente iteración. Por ejemplo, cuando se encuentra un número par, `continue` salta a la siguiente iteración sin ejecutar el `print()`:

```
for num in range(10):
    if num % 2 == 0:
        continue  # Salta el resto del bucle si num es par
    print(num)
```

O para `while`:

```
i = 0
while i < 10:
    i += 1
    if i % 2 == 0:
        continue  # Salta los números pares
    print(i)
```

La cláusula `else` se ejecuta cuando el bucle termina por agotamiento de la iteración (`for`), o cuando la condición se vuelve falsa (`while`), pero no cuando se termina por una instrucción de interrupción `break` que aquí se usa para indicar que el bucle terminó normalmente:

```
for num in range(5):
    if num == 3:
        print("Se encontró el 3, rompiendo el bucle.")
        break
else:
    print("El bucle terminó sin usar break.")
```

Iteración para la validación de datos a la entrada

Tal y como se ha mostrado anteriormente, dado un dato no válido para el problema se podría avisar y volver a pedir el dato incluyendo la lectura en un bucle.

```
numero = int(input("Escriba un número positivo: "))
while numero < 0:
    print("¡Ha escrito un número negativo! Inténtelo de nuevo.")
    numero = int(input("Escriba un número positivo: "))
print("Gracias por su colaboración.")
```

No obstante, esta solución hace dos lecturas con `input` (una dentro y otra fuera del bucle) y la condición del bucle se "escribe en negativo", es decir, recoge la condición no válida de los datos, lo que dificulta la legibilidad del código. La alternativa es la utilización de un centinela/indicador lógico que controle la validez. Es una extensión de la versión anterior pero incluyendo una variable lógica que clarifica la notación, aunque se mantiene la duplicación de la lectura.

```
entero = int(input("Quiero un entero positivo "))
valido = entero > 0
while not valido:
   print("dato erroneo")
   entero = int(input("Quiero un entero positivo "))
   valido = entero > 0

print("aqui proceso el entero:", entero)
```

Otra alternativa, muy común en Python, es el uso de `while True` pero con salida abrupta del bucle. Esta solución es el equivalente en Python del bucle `repetir hasta que` o `do ... while` que está en otros lenguajes, no necesitando la salida incondicional porque no se usaría `break`:

```
while True:
    entero = int(input("Quiero un entero positivo "))
    if entero <= 0:    # condición de permanencia en bucle
        print("dato erroneo")
    else:
        break

print("aqui proceso el entero:", entero)
```

Siempre se entra en el bucle y solamente se sale cuando se cumple la condición dentro de un `if` en el cuerpo del bucle que dispara `break`, puesto que la condición del bucle es siempre verdadera.

o bien colocando en el `if` la condición de salida del bucle, lo que mejora la legibilidad:

```
while True:
    entero = int(input("Quiero un entero positivo "))
    if entero > 0:  # condición de salida del bucle
        break
    print("dato erroneo")

print("aqui proceso el entero:", entero)
```

Esta solución no gusta demasiado a los programadores de otros lenguajes por el abuso de `break`. Su utilización para principiantes se suele recomendar solo en estos bloques de lecturas de datos con `input`, puesto que realmente ahorran líneas de código. En general, no se aconseja su uso para operaciones de procesamiento, salvo por programadores avanzados.

Dentro de un bucle de lenctuar de datos con `while True` también se puede manejar la excepcion `ValueError`:

```
while True:
    try:
        numero = int(input("Introduce un número: "))
        print("El número introducido es:", numero)
        break  # Salir del bucle si se introduce un número válido
    except ValueError:
        print("Error: Por favor, introduce un número entero válido.")
```

Si también se quiere añadir una condición a la entrada, por ejemplo que el número sea un dígito entre 1 y 49, el código sería:

```
while True:
    numero_str = input("Introduce un número entre 1 y 49: ")
    try:
        numero = int(numero_str)
```

```
        if 1 <= numero <= 49:
            print("El número introducido es:", numero)
            break  # Salir del bucle si se introduce un número válido
        else:
            print("Error: Por favor, introduce un número entre 1 y 49.")
    except ValueError:
        print("Error: Por favor, introduce un número entero válido.")
```

Una solución de validación de números sin manejar las excepciones puede usar `is.digit` :

```
while True:
    numero_str = input("Introduce un número: ")
    if numero_str.isdigit():
        numero = int(numero_str)
        print("El número introducido es:", numero)
        break  # Salir del bucle si se introduce un número válido
    else:
        print("Error: Por favor, introduce un número entero válido.")
```

Pero un real tiene parte decimal, que al leerse como una cadena con `input` debe controlarse:

```
while True:
    try:
        numero_real = float(input("Introduce un número real: "))
        print("El número introducido es:", numero_real)
        break  # Salir del bucle si se introduce un número válido
    except ValueError:
        print("Error: Por favor, introduce un número real válido.")
```

O bien:

```
while True:
  numero_str = input("Introduce un número real: ")
  if all(c.isdigit() or c == '.' for c in numero_str) \
        and numero_str.count('.') <= 1:
    try:
        numero_real = float(numero_str)
        print("El número introducido es:", numero_real)
        break  # Salir del bucle si se introduce un número válido
    except ValueError:
        print("Error: Por favor, introduce un número real válido.")
  else:
    print("Error: Por favor, introduce un número real válido.")
```

No obstante, como ya se ha indicado, estos procesos de validación están sujetos a **contrato**. Lo que quiere decir que si el programa especifica en su modo de uso que no funcionará con valores negativos, si el usuario introduce un negativo tiene posibilidades de que el programa falle y no sería un error. Si se usa una sartén para cocer pasta puede que el resultado no sea del gusto de cocinero, pero el problema no es de la sartén.

En los ejercicios resueltos mostrados solo se incluye la validación cuando se especifica expresamente en el enunciado.

Problemas resueltos con iteraciones

Ejercicio resuelto: Indice de Masa Corporal (IMC)

Construir un programa que calcule, en el sistema métrico decimal y que permita realizar varios cálculos, el índice de masa corporal conocido el peso kg y la altura en metros mediante la división del peso entre la estatura al cuadrado. Complemente la información mostrada en pantalla añadiendo también la categoría del rango del IMC. Semánticamente no tiene sentido pesos y alturas negativas, habrá de validarse su valor de entrada.

Categoría	Rango IMC
Delgadez Severa	16 o menos
Delgadez Moderada	16 - 17
Delgadez Leve	17 - 18.5
Normal	18.5 - 25
Sobrepeso	25 - 30
Obesidad clase I	30 - 35
Obesidad clase II	35 - 40
Obesidad clase III	40 o más

```
eleccion = "s"
valido = ("S", "s", "n", "N")
si_lista = ("S", "s")

while eleccion in si_lista:
    peso = -1
    alto = -1

    while peso <= 0:
        peso = float(input("¿Cuál es su peso (kg)? "))
        if peso <= 0:
            print("El peso debe ser un número positivo.")

    while alto <= 0:
        alto = float(input("¿Cuál es su altura (m)? "))
        if alto <= 0:
            print("La altura debe ser un número positivo.")

    imc = peso / (alto * alto)
    print("Su IMC es:", round(imc, 2))

    if imc <= 16:
        print("Delgadez severa")
    elif imc <= 17:
        print("Delgadez moderada")
    elif imc <= 18.5:
        print("Delgadez leve")
    elif imc <= 25:
        print("Normal")
    elif imc <= 30:
```

```
        print("Sobrepeso")
    elif imc <= 35:
        print("Obesidad clase I")
    elif imc <= 40:
        print("Obesidad clase II")
    else:
        print("Obesidad clase III")

    eleccion = input("¿Desea realizar otra operación (S/N)? ")
    while eleccion not in valido:
        eleccion = input("Opción no válida. Introduce S o N: ")
```

Ejercicio resuelto: Interés del capital

Escribir un programa que pregunte al usuario una cantidad a invertir, el interés anual y el número de años, y muestre por pantalla el capital obtenido en la inversión cada año que dura la inversión.

```
while True:
    try:
        cantidad = float(input("¿Cantidad a invertir? "))
        if cantidad > 0:
            break
        else:
            print("La cantidad a invertir debe ser un número positivo.")
    except ValueError:
        print("Entrada no válida. Ingresa un número válido.")

while True:
    try:
        interes = float(input("¿Interés porcentual anual? "))
        # No es necesario validar que sea positivo,
        # ya que un interés negativo puede representar pérdidas
        break
    except ValueError:
        print("Entrada no válida. Ingresa un número válido.")

while True:
    try:
        tiempo = int(input("¿Años? "))
        if tiempo > 0:
            break
        else:
            print("El número de años debe ser un entero positivo.")
    except ValueError:
        print("Entrada no válida. Ingresa un número entero positivo.")

capital_final = cantidad * (1 + interes / 100) ** tiempo

print(f"Capital final tras {tiempo} años: {capital_final:.2f}")
```

Tal como se muestra a veces la validación son muchas más líneas de código que los cálculos a realizar. Esta es la razón por la que debe quedar claro el *contrato* de uso de un script, es decir, con qué tipo de datos y en qué circunstancias se generará un resultado (correcto o no) sin dar errores en la ejecución o bloquearse.

Ejercicio resuelto: Triángulo de asteriscos

Escribir un programa que pida al usuario un número entero y muestre por pantalla un triángulo rectángulo e isósceles, de altura el número introducido.

```
*
**
***
****
```

Solución inicial

```
while True:
    try:
        n = int(input("Introduce la altura del triángulo (entero positivo): "))
        if n > 0:
            break  # Si es un entero positivo, salimos del bucle
        else:
            print("Por favor, introduce un número entero positivo.")
    except ValueError:
        print("Entrada no válida. Debes ingresar un número entero positivo.")
for i in range(n):
    for _ in range(i + 1):
        print("*", end="")
    print("")
```

Solución alternativa sin validación

```
n = int(input("Introduce la altura del triángulo (entero positivo): "))
for i in range(n, 0, -1):
    for _ in range(i):
        print("*", end="")
    print("")
```

Otra solución

```
n = int(input("Introduce la altura del triángulo (entero positivo): "))
for i in range(n):
    for _ in range(n-i):
        print("*", end="")
    print("")
```

Solución para triangulo invertido

```
n = int(input("Introduce la altura del triángulo (entero positivo): "))
for i in range(n, 0, -1):
    # Imprimir espacios en blanco para alinear a la derecha
    for _ in range(n - i):
        print(" ", end="")
    # Imprimir asteriscos
    for j in range(i):
        print("*", end="")
    print("")
```

Solución alternativa para triangulo invertido

```
n = int(input("Introduce la altura del triángulo (entero positivo): "))
for i in range(n):
    # Imprimir espacios en blanco para alinear a la derecha
    for _ in range(n - i):
        print(" ", end="")
    # Imprimir asteriscos
    for _ in range(i+1):  # Imprimir solo un asterisco en la primera fila
        print("*", end="")
    print("")
```

Ejercicio resuelto: ¿Es primo?

Escribir un programa que pida al usuario un número entero positivo mayor que 2 y muestre por pantalla si es un número primo o no (no se incluye validación de la entrada).

```
n = int(input("Introduce un número entero positivo mayor que 2: "))
i = 2
while n % i != 0:
    i += 1

if i == n:
    print(str(n) + " es primo")
else:
    print(str(n) + " no es primo")
```

Solución alternativa que es más eficiente pero menos estructurada

```
n = int(input("Introduce un número entero positivo mayor que 2: "))
for i in range(2, n):
    if n % i == 0:
        break
if (i + 1)  == n:
    print(str(n) + " es primo")
else:
    print(str(n) + " no es primo")
```

Ejercicio resuelto: Suma de divisores

Calcular la suma de los divisores de cada número introducido por teclado. Se termina cuando el número ingresado sea negativo.

```
print("-----------------------------------------------------------")
print("Calcula la suma de divisores.")
print("-----------------------------------------------------------")

#Entradas
print("Introduce un número, y para acabar uno negativo:")  # Ojo con el 0
numero = int( input("Núm: "))

while numero > 0 :
  suma = 0
  for i in range(1,numero+1):
```

```
    if numero % i == 0 :
      suma = suma + i

  print("\nSALIDA: ")
  print("-------------------------------------------------------")
  print("La suma de los divisores del número es:", suma, "\n" )
  print("Introduce un número, y para acabar uno negativo:")
  numero = int( input("Núm: "))
```

Ejercicio resuelto: Algoritmo para la determinación del máximo común divisor (MCD).

Se trata de encontrar el mayor número que divida exactamente dos números enteros positivos dados. El algoritmo que resuelve el problema es uno de los más antiguos y famosos estando atribuido a Euclides (no comprobar la posibilidad de valores no numéricos).

Pasos del algoritmo:

Entrada: Dos números `a` y `b`, con `a > b`.

Mientras `b` distinto 0 se calcula el resto `r= a % b` y se reemplaza `a` por `b` y `b` por `r`. Cuando `b=0`, el MCD es el valor actual de `a`.

```
# Halla el MCD de dos números enteros naturales, incluido el 0

# ENTRADA DE DATOS
print('Introduzca dos enteros, para los que se calculará el MCD.')
dividendo = -1
divisor = -1
while dividendo < 0 or divisor < 0:
    dividendo = int(input('Introduzca el primer número (>=0): '))
    divisor = int(input('Introduzca el segundo número (>=0): '))

    if dividendo < 0 or divisor < 0:
        print("Ambos números deben ser iguales o mayores que 0.\n")

# ALGORITMO MCD
if dividendo < divisor:  # Asegurando  dividendo sea mayor que  divisor
    dividendo, divisor = divisor, dividendo

dividendo_copia = dividendo  # para el mensaje de salida
divisor_copia = divisor

if dividendo == divisor == 0:
    mcd = 0
elif divisor == 0:
    mcd = dividendo
else:
    resto = dividendo % divisor   #  resto adelantado fuera del bucle
    while resto != 0:
        dividendo = divisor
        divisor = resto
        resto = dividendo % divisor
    mcd = divisor

# SALIDA .
print(f'El MCD de {dividendo_copia} y {divisor_copia} es {mcd}.')
```

Ejercicio resuelto: Calcular producción

Calcular el promedio de peso de frutos por árbol a partir de la lista de producciones de un tipo de cultivo expresados como listas de listas:

```
arboles = [["Manzano", [10, 12, 15, 8]], ["Naranjo", [6, 8, 9, 5]],
           ["Pomelo", [4, 5, 6, 3]]]
resultados = []
for arbol in arboles:
    nombre = arbol[0]
    promedio = sum(arbol[1]) / len(arbol[1])
    resultados.append([nombre, promedio])

for nombre, promedio in resultados:
    print(f"Promedio de peso de frutos del árbol {nombre} es {promedio} kg.")
```

Técnicas y herramientas de iteración

La iteración es uno de los pilares fundamentales de la programación a continuación se exploraran de forma simple diversas técnicas y herramientas que enriquecen el uso de los bucles, expandiendo su funcionalidad y versatilidad. Estas técnicas no sólo optimizan el código, sino que también facilitan la resolución de problemas complejos de manera más expresiva y compacta.

Uso de funciones de iteración

Exiten diversas funciones que permiten facilitar y personalizar las iteraciones necesarias en un problema, como `enumerate` o `reverse`, siendo `zip` una de las más conocidas. Su explicación completa está fuera de nivel de iniciación en Python, se incluyen algunos ejemplos.

enumerate

Existen situaciones en las que no solo se quiere acceder al elemento i-ésimo de la colección.

Cuando se itera sobre una secuencia, se puede obtener el índice de posición junto a su valor correspondiente usando la función `enumerate()`.

```
for indice, valor in enumerate(['tic', 'tac', 'toe']):
    print(indice, valor)
```

Con la función se crea un objeto enumerador, que devuelve pares de valores `indice, valor` comenzando por 0. Con el `for` se desestructuran los pares que en este ejemplo solo se muestran con `print` pero podrían tratarse de cualquier otra manera.

reversed

Para iterar sobre una secuencia en orden inverso, se especifica primero la secuencia sin invertir y luego se llama a la función `reversed()`.

```
for i in reversed(range(1, 10, 2)):
    print(i)
```

sorted

Se utiliza la función `sorted()` para iterar sobre una secuencia ordenada `s` puesto que devuelve una nueva lista ordenada dejando a la original intacta.

```
cesta = ['manzana', 'naranja', 'tomate', 'tomate', 'naranja', 'plátano']
for i in sorted(cesta):
    print(i)
```

zip

Si se usan varias listas con `zip`, el resultado devuelto será una tupla donde cada elemento tendrá todos y cada uno de los elementos i-ésimos de las listas dadas como entrada a `zip`. Combinada con un `for` para iterar listas permite recorrer estas listas en paralelo:

```
a = [1, 2]
b = ["Uno", "Dos"]

for numero, texto in zip(a, b):
    print("Número", numero, "Letra", texto)
```

La salida es

```
Número 1 Letra Uno
Número 2 Letra Dos
```

Este versatilidad se verá sobre un ejemplo donde se tienen que guardar los datos de un agricultor: nombre, el municipio y el total de hectáreas de su explotación. Se puede pensar en tratar los datos de cada agricultor como una lista (`["Miguel", "Almería", 1200]`), por lo que los datos de todos los agricultores serán una lista de listas.

```
agricultor = [ ["Miguel", "Almería", 1200], ["Ana", "Tahal", 1700],
               ["Lourdes", "Berja", 1500]]
print(agricultor)
print("")

print(*agricultor)  # Separa por espacios el primer nivel
print("")
print(agricultor[1])
print(*agricultor[1])  # Separa por espacios los datos de primer agricultor

# Recorre la lista para mostrar los datos
for i in range(len(agricultor)):
    print("Nombre: ", agricultor[i][0])
    print("Muncipio: ", agricultor[i][1])
    print("Superficie parcela:", agricultor[i][2])
    print("-" * 24)
```

El uso de tres listas combinado con `zip()` mejora la legibilidad y mantenibilidad porque manejar el anidamiento puede ser complejo. Si bien, si se quiere tener la lista completa, se puede crear utilizando `list`.

```
etiquetas = ["Nombre", "Municipio", "Superficie_parcela"]
nombres = ["Miguel", "Ana", "Lourdes"]
municipios=["Almería", "Tahal", "Berja"]
superficies=[1200, 1700, 1500]

print(*etiquetas,sep="   |   ")
for nom, munic, sup in zip(nombres, municipios, superficies):
    print("-"*47)
    print(f"{nom:>8s} | {munic:^13} | {sup:>13.2f}")
print("-"*47)

lista_completa = list(zip(nombres, municipios, superficies))
```

```
Nombre   |   Municipio   |   Superficie_parcela
-----------------------------------------------
  Miguel |    Almería    |       1200.00
-----------------------------------------------
     Ana |     Tahal     |       1700.00
-----------------------------------------------
 Lourdes |     Berja     |       1500.00
-----------------------------------------------
```

iter

La función `iter()` se utiliza para crear un iterador a partir de un objeto iterable. Un iterador permite recorrer los elementos de un iterable sin necesidad de usar un bucle `for`. En lugar de almacenar toda la secuencia en memoria, los iteradores generan los elementos bajo demanda, optimizando el uso de recursos.

Una vez que se tiene un iterador, se pueden obtener sus elementos uno a uno utilizando la función `next()`:

```
texto = "python"
iterador = iter(texto)  # conversión  en un iterador

print(next(iterador))  # 'p'
print(next(iterador))  # 'y'
print(list(iterador))  # ['t', 'h', 'o', 'n']
```

Un iterador también puede usarse en un bucle, sin necesidad de llamar manualmente a `next()`:

```
iterador = iter(texto)

for elemento in iterador:
    print(elemento)  # Imprime cada letra de "python"
```

Una vez que se han recorrido todos los elementos de un iterador, este queda vacío y no puede reutilizarse, generándose una excepción `StopIteration`, ya que no quedan elementos para iterar.

Comprensión de listas

La comprensión de listas permite crear listas de forma compacta y eficiente. Se usa con corchetes `[ ]`, incluyendo una expresión seguida de un `for`, y opcionalmente, condiciones `if` o bien, más `for`. Esto permite generar listas de manera más clara y concisa.

```
variable = [<op_elementos> for <elementos> in <input_list> <if condicion>]
```

Para obtener la lista de los números pares de 0 a 50 sobre la variable `numeros`:

```
numeros = [x**2 for x in range(50) if x  % 2 == 0]
print(numeros)
```

La salida es:

```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676, 784, 900, 1024, 1156, 1296, 1444,
1600, 1764, 1936, 2116, 2304]
```

El bucle `for` genera el resultado correcto, pero la comprensión de listas es una notación mucho más compacta y legible. Por ejemplo, para convertir en mayúscula los elementos de una lista dada:

```
texto_minuscula = ['Un','día','vi','una','vaca','vestida','de','uniforme']
```

Solución sin comprensión de listas

```
texto_mayuscula = []
for palabra in texto_minuscula:
    texto_mayuscula.append(palabra.upper())
print(texto_mayuscula)
```

Solución con comprensión de listas

```
mayuscula = [palabra.upper() for palabra in texto_en_minuscula]
print(mayuscula)
```

O si se quiere transformar esta lista en esta otra lista con los valores al cuadrado:

```
xs = [0, 1, 2, 3, 4, 5]
```

Solución con `for`

```
for x in xs :
    print(x ** x)
```

Solución con comprensión de listas

```
print([x*x for x in xs])
```

Otra de las ventajas de la comprensión de listas es que se pueden aplicar condicionales tanto en el iterador como en los valores o en ambos:

```
# Condicional en el iterador (filtrando números pares)
print([x for x in range(10) if x % 2 == 0])

# Condicional en el valor (eleva al cuadrado los números menores de 5
# y deja los demás igual)
print([x**2 if x < 5 else x for x in range(10)])

# Condicional en ambos (filtra números pares y los eleva al cuadrado
# si son menores que 5)
print([x**2 if x < 5 else x for x in range(10) if x % 2 == 0])
```

Ejercicio previo con comprensión de listas: Calcular producción

Dado un conjunto de árboles frutales con los registros de producción en kilogramos, se quiere calcular el promedio de peso de frutos por árbol.

Solución con compresión de listas

```
arboles = [["Manzano", [10, 12, 15, 8]], ["Naranjo", [6, 8, 9, 5]],
           ["Pomelo", [4, 5, 6, 3]]]

resultados = [[nombre, sum(pesos) / len(pesos)] for nombre, pesos in arboles]

for nombre, promedio in resultados:
    print(f"Promedio de peso de frutos del árbol {nombre} es {promedio} kg.")
```

Ejercicio resuelto: Media cosecha

Construir un programa para calcular la media de la cosecha diaria de tomates mayores o iguales a 25 kg de la siguiente lista `[25, 32, 20, 18, 30, 35, 24]`

Solucion con un bucle tradicional

```
cosechas_diarias = [25, 32, 20, 18, 30, 35, 24]

cosechas_mas_25 = []
for cosecha in cosechas_diarias:
    if cosecha >= 25:
        cosechas_mas_25.append(cosecha)

suma_cosechas = sum(cosechas_mas_25)
cantidad_cosechas = len(cosechas_mas_25)

if cantidad_cosechas > 0:       # Controlar la división por cero
    media_cosechas_mas_25 = suma_cosechas / cantidad_cosechas
else:
    media_cosechas_mas_25 = 0

print(f"La media de las cosechas mayores o iguales a 25 kg es: \
     {media_cosechas_mas_25:.3f} kg.")
```

Una alternativa más eficiente al bucle de obtener las cosechas mayores o iguales a 25 kg sería utilizar `range` para iterar sobre los índices puesto que no tiene que realizar busqueda secuencial

```
for i in range(len(cosechas_diarias)):
    if cosechas_diarias[i] >= 25:
        cosechas_mas_25.append(cosechas_diarias[i])
```

Solución con comprensión de listas

```
cosechas_diarias = [25, 32, 20, 18, 30, 35, 24]
cosechas_mas_25 = [cosecha for cosecha in cosechas_diarias if cosecha >= 25]
media_cosechas_mas_25 = sum(cosechas_mas_25) / len(cosechas_mas_25)

print(f"La media de las cosechas mayores o iguales a 25 kg es: \
    {media_cosechas_mas_25:.3f} kg.")
```

Ejercicio resuelto: Espacio muestral

Escribir el espacio muestral del lanzamiento de una moneda y un dado.

```
moneda = ('c', 'x')
dado = (1, 2, 3, 4, 5, 6)

espacio_producto = []
for m in moneda:
    for d in dado:
        espacio_producto.append((m, d))
print(espacio_producto)
```

Solución con compresión de listas,

```
moneda = ('c', 'x')
dado = (1, 2, 3, 4, 5, 6)

espacio_producto = [(m, d) for m in moneda for d in dado]
print(espacio_producto)
```

Iterar en un diccionario

Si bien los diccionarios son iterables, puesto que usan pares clave-valor se puede iterar de tres modos, sobre las **claves**, los **valores**, y los pares **clave-valor** llamando a métodos específicos de diccionarios (`.value()` y `.items()`).

```
for <variable> in <variable_diccionario>:
    <código>
```

Por ejemplo:

```
dic = {"a": 1, "b": 2, "c": 3}

for clave in dic:           # Para iterar sobre las claves
    print(clave)

for valor in dic.values():  # Para iterar sobre los valores
    print(valor)
```

```
for clave, valor in dic.items():  # Para iterar sobre los pares clave-valor
    print(clave, valor)
    print (clave)
```

Recorrido de un diccionario

```
for t in dic.items():
    print(t[0], end= " ")
    print(t[1])

# Si sólo se usa una variable,  se crea una tupla con el par clave-valor
for par in dic.items():
    print(par)
```

Ejercicio resuelto: Producción por fecha

Se debe escribir un programa con un diccionario donde se recogan las fechas y los kilos de producto vendidos en esa fecha, calculando la media de las ventas en un rango de fechas.

```
ventas = {      '2022-01-01': 10,    '2022-01-02': 15,
                '2022-01-03': 20,    '2022-01-04': 25,
                '2022-01-05': 30,    '2022-01-06': 35,
                '2022-01-07': 40,}
fecha_inicio = input("Introduce la fecha de inicio (en formato AAAA-MM-DD): ")
fecha_fin = input("Introduce la fecha de fin (en formato AAAA-MM-DD): ")

total_ventas = 0
dias_ventas = 0

for fecha, ventas_dia in ventas.items():
    if fecha_inicio <= fecha <= fecha_fin:
        total_ventas += ventas_dia
        dias_ventas += 1

media_ventas = total_ventas / dias_ventas

print(f"La media de ventas entre {fecha_inicio} y {fecha_fin} \
    es de {media_ventas} kilos.")
```

Ejercicio resuelto: Cálculo de coste

Programa que use un diccionario donde se recojan las fechas y los kilos de productos vendidos en esa fecha, calculando la media de los ingresos por ventas de producto, teniendo en cuenta que el coste por unidad responde a la siguiente tabla, que además puede variar y, por lo tanto, se incluirá como un diccionario.

Unidades	Coste
0-10	20
11-25	15
mas de 26	12

```
costes = {  10:  20,  25: 15, }
MAX=12
ventas = {
    '2022-01-01': 6,     '2022-01-02': 12,
    '2022-01-03': 20,    '2022-01-04': 25,
    '2022-01-05': 30,    '2022-01-06': 35,
    '2022-01-07': 40,
}
total_ingresos = 0
dias_ventas = 0

for fecha, unidades_vendidas in ventas.items():
    coste_unidad = MAX
    for limite_inferior, coste in costes.items():
        if unidades_vendidas <= limite_inferior:
            coste_unidad = coste
            break
    ingresos_dia = unidades_vendidas * coste_unidad
    total_ingresos += ingresos_dia
    dias_ventas += 1

media_ingresos = total_ingresos / dias_ventas
print(f"La media de ingresos por ventas es de {media_ingresos} euros.")
```

Criterios de calidad de los programas

Un programa debe cumplir una serie de criterios de calidad. Los principales son: corrección, claridad, eficiencia, amigabilidad y robustez.

Se dice que un program es **correcto** si la entrada definida produce los resultados requeridos, es decir, el programa se ajusta a la especificación. Para comprobar si el funcionamiento es correcto se utilizan un conjunto de casos de prueba que permiten verificar la funcionalidad al ejecutarlos todos y verificar el resultado. Por ejemplo, para el programa anterior del cálculo de máximo común divisor se tendría:

Datos de prueba

Primer número	Segundo número	resultado
10	1	1
15	15	15
3	2	1
75	30	15

El programa debe generar el resultado correcto en todos los casos de prueba para ser **correcto**. Esto se puede hacer de forma manual ejecutando el programa tantas veces como casos de prueba y comprobando los resultados, o bien usando algún tipo de software que gestione estas pruebas de unidad.

Pytest es un marco de pruebas (testing framework) en Python que se utiliza para escribir y ejecutar tests de manera eficiente. Es especialmente útil para realizar pruebas automatizadas de funciones, módulos y aplicaciones en Python. Es fácil de utilizar porque todos los .py que contienen la palabra test se lanzan para validar el código. Lo habitual es tener una carpeta llamada `test` donde se colocarán todos los programas de prueba. Por ejemplo, el script `asigna.py`:

```
a=1
b=2
```

Tendrá asociado un script de prueba llamado por ejemplo "asigna_test.py"' que se lanzará con pytest cada vez que se realice una prueba.

```
import asigna as caso

def test_asigna():
    assert caso.a == 1
    assert caso.b == 2
```

Pero en este libro no se definirán ni usarán sistemas automáticos de prueba en los ejercicios resueltos.

Prácticamente todos los programas se modificarán en el futuro. Un programa será **claro** si su contenido es entendible por personas distintas a su autor o por el mismo autor pasado un tiempo.

Sobre el programa anterior se puede observar como el uso de comentarios, la separación en tres areas: Entrada - Proceso - Salida, y la utilización de idenficadores con significado para el problema mejoran la claridad. El mismo ejemplo menos claro sería:

```
print('Introduzca dos enteros, para los que se calculará el MCD.')
dd = -1
d = -1
while dd < 0 or d < 0:
    dd = int(input('Introduzca el primer número (>=0): '))
    d = int(input('Introduzca el segundo número (>=0): '))

    if dd < 0 or d < 0:
        print("Ambos números deben ser iguales o mayores que 0.\n")

    ddd = dd
    dddd = d
if dd < d:
    dd, d = d, dd
if dd == d == 0:
  a = 0
elif d == 0:
    a = dd
else:
    b = dd % d
    while b != 0:
      dd = d
      d = b
```

```
        b = dd % d
    a = d
print(f'El MCD de {ddd} y {dddd} es {a}.')
```

El consumo óptimo de recursos de computación (tiempo de CPU, memoria,...) definen la **eficiencia** de un programa:

```
# Halla el MCD de dos números enteros naturales,
# incluido el 0 (Versión Fuerza Bruta)

# ENTRADA DE DATOS
print('Introduzca dos enteros, para los que se calculará el MCD.')

dividendo = int(input('Introduzca el primer número (>=0): '))
divisor = int(input('Introduzca el segundo número (>=0): '))

# ALGORITMO MCD por fuerza bruta con bucle while descendente
if dividendo < divisor:  # Asegurando  dividendo sea mayor que  divisor
    dividendo, divisor = divisor, dividendo
dividendo_copia = dividendo
divisor_copia = divisor

if dividendo == divisor == 0:
    mcd = 0
elif divisor == 0:
    mcd = dividendo
else:
    resto= divisor
    while divisor > 0 and resto != 0:
        resto = dividendo % divisor
        dividendo = divisor
        divisor -= 1  # Desciende de 1 en 1
    mcd = dividendo

# SALIDA
print(f'El MCD de {dividendo_copia} y {divisor_copia} es {mcd}.')
```

La facilidad de uso para el usuario define la **amigabilidad**. En ciertos casos puede producir ineficiencia y se ha llegar a un compromiso. Para scripts simples basta con mostrar la información al usuario que facilite la interacción, por ejemplo mostrando mensajes, bien sea en cada orden de entrada o para explicar los resultados.

Se dice que un programa es **robusto** si es capaz de reaccionar ante entradas no definidas o al menos generar mensajes de aviso sin generar bloqueo. Este criterio se relaja en ciertas condiciones, dependiendo del contrato/compromiso de uso fijado al contruir el código, basta con recordar los ejemplos de los bucles de validación de las entradas de un programa o los bloques `try`.

```
# ENTRADA DE DATOS
print('Introduzca dos enteros, para los que se calculará el MCD.')
dividendo = -1
divisor = -1
while dividendo < 0 or divisor < 0:
    dividendo = int(input('Introduzca el primer número (>=0): '))
```

```
    divisor = int(input('Introduzca el segundo número (>=0): '))

    if dividendo < 0 or divisor < 0:
        print("Ambos números deben ser iguales o mayores que 0.\n")
```

O bien:

```
entero = int(input('Introduzca  un entero positivo: '))
while entero <= 0:
  print("dato erroneo")
  entero = int(input('Introduzca  un entero positivo: '))

print(f"continuamos procesado {entero}")
```

O incluso:

```
while True:
  entero = int(input('Introduzca  un entero positivo: '))
  if entero >=0:
     break

print(f"continuamos procesado {entero}")
```

También está la posibilidad de utilizar los bloques de captura de errores `try`.

Funciones. Diseño Modular.

En Python, al igual que en otros lenguajes de programación, una **función** es un bloque de código con nombre que encapsula un conjunto de instrucciones diseñadas para realizar una tarea específica. Su principal ventaja es la reutilización, puesto que una función puede ejecutarse tantas veces como sea necesario simplemente llamándola por su nombre. Además, su comportamiento puede adaptarse mediante argumentos, que permiten modificar su ejecución en función del contexto.

Un ejemplo mostrado previamente es la función `velocidad()`, que encapsulaba el *cálculo de la velocidad de un cuerpo* utilizando la palabra clave `def`. En Python, se puede usar `def` y `lambda` para definir funciones, ésta última es útil para definir funciones anónimas cortas y de una sola expresión.

Además de permitir la creación de funciones personalizadas, Python ofrece una gran cantidad de funciones integradas, así como muchas otras organizadas en librerías o paquetes. Por ejemplo, al importar una librería completa con import math, se tiene acceso inmediato a todas sus funciones, que pueden usarse a lo largo del código. Sin embargo, si solo se necesita una función específica (como `sqrt` para calcular raíces cuadradas), se puede usar una importación más precisa con `from math import sqrt`. Esta forma no solo hace el código más limpio y directo, sino que también evita cargar funciones innecesarias en memoria.

En el contexto de las funciones, se utilizan dos acciones clave: **definir** y **llamar** una función:

- Definir una función implica especificar las instrucciones que se van a reutilizar, junto con los datos necesarios para su ejecución. A este proceso también se le llama declarar o construir una función.

- Llamar a una función ocurre cuando se utiliza dicha función en el código, identificándola por su nombre y proporcionando los valores concretos que procesará en ese momento. También se conoce como invocar, lanzar o ejecutar la función.

Módulos y estructura general de un programa

Las funciones se **llaman** al escribir su nombre seguido de paréntesis. Dentro de estos paréntesis, se pueden incluir los argumentos, que son los datos necesarios para que la función realice su tarea. Estos argumentos pueden variar en cada llamada a la función. De hecho, ya se han utilizado funciones en ejemplos anteriores, por ejemplo, `print` es una función.

```
print('abc')
print ('Hola')
```

En este ejemplo, `print` es el nombre de la función, y `'abc'` es un argumento que la función utiliza para mostrar un mensaje en pantalla. En la segunda llamada, el argumento cambia a `'Hola'`.

Algunas funciones tienen parámetros con valores por defecto; por ejemplo, `sep` en `print` tiene como valor predeterminado `'\n'`, lo que hace que cada impresión ocurra en una nueva línea. Cuando una función define sus argumentos con nombres, éstos pueden especificarse en cualquier orden al llamarla, siempre que se indiquen sus claves explícitamente.

```
# son equivalentes no importa el orden
print('Hola', 'que', 'tal', sep=' ', end='!\n')
print('Hola', 'que', 'tal', end='!\n', sep=' ')
```

Python recomienda que cualquier código esté dentro de una función incluso cuando el script tiene un solo bloque de instrucciones. Eso sí, hay que de decirle al intérprete por donde tiene que empezar la ejecución para lo que se utiliza un nombre especial `__main__`, tal como se mostró en el script para calcular la velocidad.

Para **definir** una función se utiliza la palabra clave `def`, seguida del nombre que se quiere dar a la función que se ha de formar con las mismas normas que los identificadores de variables. A continuación, se añaden los paréntesis con los parámetros de entrada de la función seguidos de dos puntos (`:`) que marcan el inicio de un bloque. Después, se agregan las instrucciones o bloques de instrucciones que se quiere que se ejecuten cada vez que se invoque a la función. Dentro de este bloque de instrucciones (casi siempre al final y de forma opcional, aunque recomendable) se coloca una instrucción de retorno usando la palabra reservada `return` que indica el valor devuelto por la función. Además, siempre que sea posible, se deben definir valores por defecto `default` para todos los parámetros de entrada.

Si bien en otros lenguajes no siempre un bloque de código devuelve un valor (módulos o procedimientos), en Python una función siempre devolverá un valor, bien se puede hacer de forma explicita utilizando `return` o bien devolviendo el valor predeterminado `None` (como el caso de `print`).

Puesto que la indentación es fundamental para definir los bloques de código, es importante asegurarse de que todas las líneas que forman parte de una función estén correctamente indentadas.

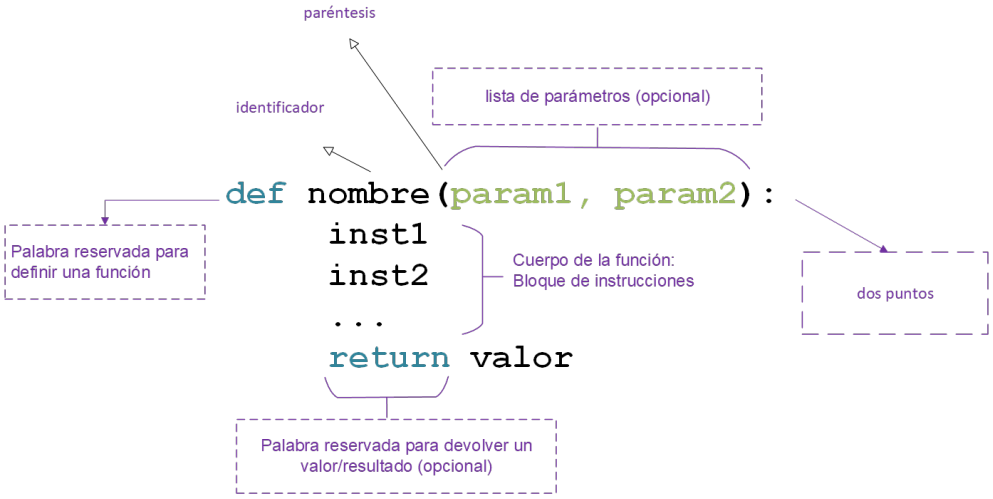

Figura 14: Elementos de una función

Por ejemplo, para construir una función llamada `mifuncion` que devuelva una cadena con un saludo se construye el siguiente código:

```
# Definición
def mifuncion():
    rsltd = '¡Hola!'
    return rsltd
```

Para utilizar la función desde cualquier otro bloque de código, se emplea el nombre `mifuncion` que puede ser otra función.

```
# Llamada
print(mifuncion())
print(mifuncion())
print(mifuncion())
# con esto se muestra tres veces: ¡Hola!
```

Al invocar una función, es importante asegurarse de que el número de argumentos proporcionados coincida con los parámetros definidos en su declaración. Si la función tiene parámetros con valores por defecto, se pueden omitir algunos argumentos, pero nunca se debe exceder la cantidad máxima de argumentos.

Sea la siguiente función que cuenta cuantos dígitos hay en una palabra, el argumento debe ser la palabra que se quiere contar.

```
def contardigitos(s = "1111"):
    digitos = 0
    for c in s:
        if c.isdigit():
            digitos += 1
    return digitos
```

```
# Ejemplos de uso
print(contardigitos("abc123"))  # Salida: 3
print(contardigitos())          # Salida: 4 (porque "1111" tiene 2 dígitos)
cadena = input()
print(contardigitos(cadena))
```

En el código se puede ver que la función necesita un dato de entrada, representado por la variable `s`, para realizar el conteo de dígitos. Al llamar a la función, éste dato se reemplaza por el valor específico que se desea analizar, como en el caso de la variable `cadena`. Si no se proporciona ningún valor al llamar a la función, se utilizará el valor predeterminado "1111".

Ejercicio resuelto: Fibonacci Construir una función que calcule los N primeros números de la sucesión de Fibonacci. La función recibirá un solo argumento `N`, y devolverá una lista de los primeros N números de Fibonacci:

```
def fibonacci(N):
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L

N = -1  # Valor inicial no válido
while N <= 0:
    N = int(input("Ingrese un número positivo: "))
L = fibonacci(N)
print(L)
```

Se puede devolver cualquier objeto de Python, simple o compuesto, lo que significa que las construcciones que pueden ser difíciles en otros lenguajes son sencillas en Python. Por ejemplo, cuando se quieren devolver múltiples valores simplemente se colocan en una tupla, que se construye con comas:

```
def real_imag_conj(val):
    return val.real, val.imag, val.conjugate()

r, i, c = real_imag_conj(3 + 4j)
print(r, i, c)
a=real_imag_conj(3 + 4j)
type(a)
```

En resumen, un programa Python son por tanto un conjunto de funciones que se comunican entre ellas, bien sea dentro del mismo script/archivo, o llamado a funciones que están en otros scripts o librerías importadas.

El bloque principal puede explicitarse como una función, habitualmente llamada `main()`, y asignar `__main__`, aunque en los scripts de iniciación como los de este libro no se recomienda. El resultado para el caso de la sucesión de Fibonacci sería:

```
def fibonacci(N):
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L

def main():
    N = -1  # Valor inicial no válido
    while N <= 0:
        N = int(input("Ingrese un número positivo: "))
    L = fibonacci(N)
    print(L)

if __name__ == "__main__":
    main()
```

Ejercicio resuelto: Lotería primitiva

Programa que pregunte al usuario los números ganadores de la lotería primitiva $[1, 49]$, los almacene en una lista y los muestre por pantalla ordenados de menor a mayor.

Se pide al usuario seis números que ha de validarse en el rango 1,49 pero no puede haber repeticiones. Es decir para cada número que se pide se tiene que comprobar que no haya sido introducido previamente.

Tiene sentido utilizar una función para leer un número y que diga si el número es válido en el rango de la lotería primitiva, porque se tiene que usar 6 veces para leer todos los números.

Implementación

```
# Validar que el número esté dentro del rango permitido y no se repita
def validar_numero(premiados):
    while True:
        try:
            numero = int(input("Introduce un número ganador (entre 1 y 49):"))
            if numero < 1 or numero > 49:
                print("Número fuera de rango. Debe estar entre 1 y 49.")
            elif numero in premiados:
                print("El número ya se introdujo. Introduce otro distinto.")
            else:
                return numero
        except ValueError:
            print("Entrada no válida. Introduzca un número entero.")

premiados = []

# Solicitar al usuario la introducción de los números ganadores
for _ in range(6):
    numero = validar_numero(premiados)
    premiados.append(numero)

premiados.sort()      # Ordenar la lista de números ganadores
print("Los números ganadores son:", premiados)
```

Aquí se pone de manifiesto una cuestión importante para el uso de funciones. Antes de su llamada tienen que estar "disponibles", es decir, definida antes en el archivo `.py`, igual que se hace con las instrucciones de importación, porque al importar se hacen visibles las funciones del paquete importado en el script actual, razón por la que se colocan los `import` al principio de los scripts.

Comunicación entre funciones

Los bloques de código deben intercambiar información y compartir contenidos para trabajar de manera eficiente. Existen tres mecanismos principales para lograr esta comunicación.

- *Importación de funciones, variables y datos entre módulos*: Permite que los elementos definidos en un script sean accesibles en otros scripts o módulos.
- *Paso de parámetros entre funciones*: Las funciones pueden recibir y devolver información a través de los parámetros.
- *Uso de variables globales*: Aunque es posible, este enfoque está TOTALMENTE desaconsejado debido a sus riesgos y a que puede dificultar el mantenimiento del código.

Importación

La palabra clave `import` se utiliza para acceder al contenido de otro script o módulo. Cuando las funciones se encuentran distribuidas en diferentes archivos, scripts o librerías, es necesario realizar una importación antes de hacer uso de cualquier función o elemento del módulo importado.

Ejercicio resuelto: Función máximo

Construir una función de devuelva el máximo de dos números.

```
def mayor(a, b):
    if a > b:
        return a
    else:
        return b
```

Para usarlo se debe escribir,

```
x = 33
y = 77
print( mayor(x, y))
print( mayor(y, x))
h = 12
j = 22
d = mayor(h, j)
print(d)
```

Si estos dos bloques de código están en el mismo archivo. No hay problema porque `mayor` es visible siempre que aparezca antes de la llamada.

Pero si no es así hay que *cargar* la función para que esté disponible, es decir para que el módulo *principal* vea la función, es necesario incorporarlo. Para usarlo se utilizará la notación punto `.`. Hay dos opciones siendo `maximo.py` el script que contiene la función `mayor`:

```
from maximo import mayor

a = int(input("primero: "))
b = int(input("segundo: "))
print(mayor(a, b))
```

O bien:

```
import maximo

a = int(input("primero: "))
b = int(input("segundo: "))
print(maximo.mayor(a, b))
```

O también finalmente, asignando un nuevo nombre para el script importado:

```
import maximo as m

a = int(input("primero: "))
b = int(input("segundo: "))
print(m.mayor(a, b))
```

En resumen hay al menos tres formas de realizar una importación:

- `import XXX` es el método más básico para importar un módulo. Cuando se usa esta forma, todo el módulo se importa y pudiendo acceder a sus funciones o clases usando la notación de punto (.).
- `from XXX import nombreenXXX` permite importar una función, clase o variable específica desde un módulo, en lugar de importar todo el módulo. Esto hace que no se necesite usar la notación de punto para acceder a esa función.
- `import XXX as nuevonombre` permite importar un módulo pero con un alias, renombrándolo. Esto es útil cuando el nombre del módulo es largo o si se prefiere usar un nombre más corto.

Definición del principal para un módulo

Si bien no es estrictamente necesario, porque los script que se incluirán en este libro no están pensados para ser importados (suelen ser finalistas), si es que se quiere ponerlos a disposición de otros programadores o scripts, es conveniente definir el comportamiento principal dentro de un módulo/función para evitar comportamientos no deseados.

Basta con mostrar una situación no planteada hasta ahora sobre el siguiente ejemplo.

```
def saludar():
    print("Hola, mundo")

# Esta llamada  se ejecuta inmediatamente al cargar el archivo
saludar()
```

Si ahora se importa este código para usar la función `saludar`:

```
import saludo

saludo.saludar()
```

El resultado de la ejecución de este segundo código será:

```
Hola, mundo
Hola, mundo
```

Lo que ocurre es que al importar todo el módulo cuando se ejecuta el importador lo primero que se hace es ejecutar el comportamiento que no está bajo ninguna función en el módulo importado (el bloque principal).

La construcción `if __name__ == "__main__":` permite ejecutar código solo cuando el archivo se ejecuta directamente, y no cuando es importado como un módulo en otro archivo. De esta forma el script de saludo, archivo `saludo.py`, se modificaría de esta forma:

```
def saludar():
    print("Hola, mundo")

if __name__ == "__main__":
    saludar()    # solo se ejecuta si el archivo es ejecutado directamente
```

Si el archivo es ejecutado directamente, `__name__` toma el valor `"__main__"`, y se ejecuta el código del bloque `if`. Pero si es importado desde otro script, el código dentro de `if __name__ == "__main__"` no se ejecuta.

```
import saludoconmain
saludoconmain.saludar()
```

Cuando el código de la parte principal a colocar en el bloque `if __name__` incorpora diversas instrucciones o es complejo, no se suelen colocar directamente en este bloque sino que se define una función que incluye éstas instrucciones. En este caso, se asigna a la variable especial `_name_` el nombre de esta función principal. Por herencia de otros lenguajes se le suele llamar `main`, pero podría usarse cualquier otro identificador tal como ya se ha mostrado para la función `fibonacci()`.

```
def saludar(nombre):
    print("Hola ",nombre)

def main():      ## puede llamarse 'principal'
    saludar('Pepe')
```

```
    saludar('Juan')
    for i in range (1,4):
        saludar('Carlos')

if __name__ == "__main__":
    main()
```

Sin embargo, para mantener la simplicidad, en la mayoría de los ejercicios de este libro no separara la lógica de importación de la ejecución, que es lo que permitiría que un archivo `.py` funcione tanto como un script ejecutable como un módulo importable. La parte principal del script no se muestra bajo una función sino en el nivel más alto de la indentación, siendo aquí donde empieza la ejecución.

Argumentos para comunicación entre funciones. Paso de parámetros.

Los argumentos o parámetros de una función son los **datos** que necesita una función para cumplir su objetivo. Por ejemplo, si se quiere una función que calcule la varianza de una lista hay que pasar a la función la lista como argumento, en este caso `t`. Esta acción de enviar datos a una función es llamada **paso de parámetros**.

```
def var(t):
    media = sum(t) / float(len(t))
    suma2 = sum(x**2 for x in t)
    var2 = suma2 / float(len(t)) - media**2
    return(var2)

print("Varianza:",var([1, 2, 33, 2, 4]))
```

El comportamiento de Python en el paso de parámetros se denomina **paso por asignación**, lo que es un poco diferente de las nociones tradicionales en otros lenguajes de programación de *paso por valor* (donde se pasa una copia del valor del argumento) y *paso por referencia* (se pasa una referencia o puntero al valor original).

El **paso por asignación** de un argumento a una función implica que lo que realmente se envía a la función es una referencia al objeto/variable en memoria, no el valor en sí, ni tampoco la referencia en el sentido clásico de otros lenguajes como C++, donde se pasa un puntero y que conlleva que las modificaciones realizadas dentro de la función afectarían directamente al valor original.

Este paso por asignación se comporta de manera diferente dependiendo de si el objeto es mutable o inmutable. Si el objeto es **inmutable** (como números, cadenas de texto, tuplas), no se modificará el objeto original dentro de la función. Si el objeto es **mutable** (como listas, diccionarios o tipos de datos compuestos/complejos/clases), las modificaciones dentro de la función afectan al objeto original.

A modo de ejemplo y para un dato inmutable:

```
def incrementar(valor):
    valor += 1
    print("Dentro de la función:", valor)

x = 10
incrementar(x)
print("Fuera de la función:", x)
```

La salida es:

```
Dentro de la función: 11
Fuera de la función: 10
```

Aquí, `x` no cambia fuera de la función. Esto se debe a que `x` es un entero (un objeto inmutable). Aunque la función recibe una referencia a `x`, cuando se hace `valor += 1`, se crea otro objeto y se asigna a `valor`, por lo que el objeto original no se modifica. Lo mismo pasa en el siguiente caso.

```
def doblar_valor(numero):
    numero *= 2
    return numero

n = 10
print(doblar_valor(n))
print(n)
```

Para modificar los tipos simples hay que devolverlos modificados y reasignarlos.

```
n = doblar_valor(n)
print(n)
```

Para un dato mutable, una lista en este caso, el comportamiento es distinto:

```
def agregar_elemento(lista):
    lista.append(4)
    print("Dentro de la función:", lista)

mi_lista = [1, 2, 3]
agregar_elemento(mi_lista)
print("Fuera de la función:", mi_lista)
```

La salida sería:

```
Dentro de la función: [1, 2, 3, 4]
Fuera de la función: [1, 2, 3, 4]
```

No solo se afecta si se añaden elementos a la lista sino también si se modifican:

```
def doblar_valores(numeros):
    for i,n in enumerate(numeros):
        numeros[i] *= 2
    return numeros

ns = [10,50,100]
print(doblar_valores(ns))
print(ns)
```

En el caso de los tipos compuestos mutables, se puede evitar la modificación enviando una copia del objeto mutable:

```
def doblar_valores(numeros):
    for i,n in enumerate(numeros):
        numeros[i] *= 2
    return numeros

ns = [10, 50, 100]
nn=doblar_valores(ns[:])  # Una copia al vuelo de una lista con [:]
print(nn)
print(ns)
```

Definición de parámetros por defecto

Al definir una función, a menudo hay ciertos valores que son comunes y se utilizan en la mayoría de los casos. Sin embargo, también puede ocurrir que, en algunas situaciones, esos valores necesiten cambiar. Para hacer que la función sea más flexible y adaptable, es posible asignar *valores predeterminados* a los argumentos.

En el caso de la función `fibonacci()` se podrían agregar valores predeterminados en los parámetros para permitir la modificación de los valores iniciales, ofreciendo más flexibilidad

```
def fibonacci(N=2, a=0, b=1):
    L = []
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

Con un solo argumento, el resultado de la llamada a la función es idéntico al anterior

```
fibonacci()
```

Sin embargo, ahora la función permite explorar nuevos escenarios, como por ejemplo el efecto de usar diferentes valores iniciales:

```
fibonacci(10, 0, 2)
```

Además, los valores también se pueden especificar por nombre. Al hacerlo, el orden de los parámetros no importa, lo que hace que la llamada a la función sea aún más flexible:

```
fibonacci(10, b=3, a=1)
```

Ámbito de las variables. Variables Globales.

El ámbito (scope) de una variable indica las partes del programa en las que esa variable es accesible. Las variables pueden clasificarse como **locales** o **globales**, dependiendo de dónde se les asigna valor y cómo se utilizan. A continuación, se describirán las diferencias entre ambos tipos

de variables, así como sus ventajas y desventajas. Es importante recordar que el uso de variables globales está desaconsejado en la mayoría de los casos.

Una **variable global** es aquella que se declara fuera de cualquier función y, por lo tanto, es accesible en todo el script, incluidas todas las funciones, a menos que haya una redefinición local dentro de una función. Para los ejemplos de este apartado se usará if `__name__ == "__main__"` para delimitar claramente los bloques de código.

```
x = 10  # Variable global

def imprimir_x():
    print(x)  # Accede a la variable global

if __name__ == "__main__":
    imprimir_x()  # Salida: 10
    print(x)
```

En este código, la variable `x` es global porque se ha asignado valor fuera de cualquier función. Cualquier bloque dentro del programa puede acceder a `x`, siempre y cuando no la sobrescriba dentro de su propio ámbito.

Una de las ventajas de las variables globales es que cualquier función o parte del código puede acceder a ellas sin tener que pasarlas como parámetros. Además, mantienen su valor durante toda la ejecución del programa. Se ahorra tener que incorporarla como argumento si se usa en múltiples llamadas a funciones.

El principal inconveniente es, al poderse cambiar en cualquier parte del código, que es difícil rastrear los problemas que las involucran. Pueden aparecer efectos secundarios si varias funciones modifican una variable global, al producirse cambios inesperados que afectarán a otras partes del programa.

Una **variable local** es aquella a la que se asigna valor dentro de una función, y sólo es accesible dentro de esa función. Fuera de ella, no es visible ni accesible.

```
def mi_funcion():
    y = 5  # Variable local
    print(y)
if __name__ == "__main__":
    mi_funcion()  # Salida: 5
    print(y)  # Error: 'y' no está definida fuera de la función
```

Aquí, la variable `y` es local a la función `mi_funcion`. Si se intenta acceder a `y` fuera de la función, se obtendrá un error porque su ámbito está restringido al interior de la función.

Las variables locales solo afectan a la función donde se asigna valor, lo que hace el código más seguro y predecible. Como solamente son accesibles dentro de una función, es más fácil rastrear los errores relacionados con su valor. Además, se puede usar el mismo nombre de variable en diferentes funciones sin que haya conflictos.

Uno de sus inconvenientes es que se requerirán más parámetros. Si es necesario que varias funciones compartan un dato, debe pasarse como argumento a cada función, lo que puede aumentar la complejidad de la llamada.

Si bien las variables globales son accesibles dentro de las funciones, modificarlas directamente dentro de una función requiere el uso de la palabra clave **global**. De lo contrario, Python creará una nueva variable local con el mismo nombre, sin modificar la global. Esto previene efectos no deseados sobre la variable global, pero supone otro gran problema para la depuración del código.

```
x = 10  # Variable global

def modificar_x():
    global x
    x = 20  # Modifica la variable global

if __name__ == "__main__":
    modificar_x()
    print(x)  # Salida: 20
```

Si no se usa la palabra clave `global`, el siguiente código NO modificará la variable global `x`:

```
x = 10

def modificar_x():
    x = 20  # Crea una variable local con el mismo nombre
    print(x)  # Salida: 20 (solo local)

modificar_x()
print(x)  # Salida: 10 (global no modificada)
```

Como recomendación es mejor usar variables globales con moderación o incluso no usarlas para evitar complicaciones en la depuración y mantenimiento del código. En lugar de depender de variables globales, se recomienda pasar los valores necesarios como argumentos a las funciones y devolver los resultados, lo que hace que el código sea más limpio, predecible y reutilizable.

Funciones importables

Desde el momento en que se importa o se define una función está disponible para ser usada. Si se quiere saber cuáles son todas las funciones propias de Python se usa `dir()`:

```
dir(__builtins__)
```

Al usar `dir` sin argumentos, se devuelve una lista con los nombres de funciones del ámbito actual. Aunque esto puede tener poco sentido dentro de un script, resulta muy útil cuando se está ejecutando Python en modo interactivo. Además, si se ha importado un paquete y se pasa el nombre del paquete como argumento, se muestran las funciones disponibles en dicho paquete ,

como en el caso de `dir(math)`. No obstante, se puede encontrar una lista exhaustiva de todas las funciones y de las funciones de las librerías estándar en la documentación oficial de Python.

Para conocer los detalles de cualquier función hay disponible una funcion de ayuda:

```
help(max)
```

Es posible incorporar ayuda específica a las funciones definidas por el programador utilizando los comentarios multilínea.

Ejercicio resuelto: Función media

Función que obtiene la media de una lista pasada como argumento

```
def media(t):
    """Función que devuelve la media de la lista pasada como argumento"""
    return(float(sum(t)) / len(t))
lista=[1, 2, 3, 4, 5]
print(media(lista))
```

Tras cargarla en el entorno interactivo, al hacer `help(media)`, el resultado mostrado en la consola será:

```
>>> help(media)
Help on function media in module __main__:

media(t)
    Función que devuelve la media de la lista pasada como argumento
>>>
```

Las **expresiones lambda** en Python son una alternativa concisa para definir funciones anónimas. Se utilizan cuando se necesita una función pequeña y rápida sin necesidad de asignarle un nombre formal. Aunque tienen limitaciones en cuanto a la cantidad de expresiones que pueden contener, son útiles en situaciones específicas, como la manipulación de listas. Así la función `media` del ejemplo anterior se reescribiría como:

```
media = lambda t: float(sum(t)) / len(t)

lista = [1, 2, 3, 4, 5]
print(media(lista))
```

`lambda t: float(sum(t)) / len(t)` define una función anónima que calcula la media. Se usa `sum(t)` para sumar los elementos y `len(t)` para obtener la cantidad de elementos. Se convierte el resultado a `float` para asegurar que sea decimal en divisiones exactas.

Generación de números aleatorios

La generación de números aleatorios desempeña un papel fundamental en numerosas disciplinas y aplicaciones dentro de la informática. Desde la simulación de fenómenos naturales hasta la

seguridad en los sistemas informáticos, la capacidad de obtener valores impredecibles es clave para el desarrollo de soluciones eficientes y robustas.

Los números aleatorios están estrechamente relacionados con la estadística porque muchos métodos estadísticos dependen de la aleatoriedad para modelar fenómenos reales y hacer inferencias sobre poblaciones. En el muestreo aleatorio, por ejemplo, se seleccionan datos de manera imparcial para garantizar que las conclusiones sean representativas. La aleatoriedad también es clave en pruebas de hipótesis y en la generación de distribuciones de datos sintéticos, lo que permite evaluar modelos y tomar decisiones informadas basadas en probabilidades.

Para generar números pseudo-aleatorios en Python se hace uso del módulo `random` de la librería estándar. Este módulo ofrece una serie de funciones que generan números aleatorios de maneras diferentes.

También existen algunas funciones para generar de datos siguiendo ciertas distribuciones estadísticas y otras que requieren un conocimiento estadístico más allá de la programación en Python.

La obtención de números pseudo-aleatorios de valor entero, se hace con la función `randint()`. La función `randint(a, b)` devuelve un número entero comprendido entre `a` y `b` (ambos inclusive) de forma aleatoria. Por ejemplo, se puede utilizar para determinar quién comienza una partida (jugador/PC); simular el dado del parchís, etc...

```
import random

# ¿Quién comienza?
comienza = random.randint(0, 1)
if comienza == 0:
    print('Comienza el jugador')
else:
    print('Comienza el PC')

# Número aleatorio del lanzamiento de un dado para jugar al parchís
numero = random.randint(1, 6)
print(numero)
```

La función `randrange(a, b, salto)` genera números enteros aleatorios comprendidos entre `a` y `b` separados entre sí con un `salto`. Por ejemplo, `randrange(5, 27, 4)` obtendría un valor aleatorio de entre los siguientes posibles: 5, 9, 13, 17, 21, 25.

```
random.randrange(5,27,4)
```

La función `random()` devuelve un real comprendido entre [0.0 y 1.0)

```
random.random()
```

La función `uniform(a,b)` devuelve un real aleatorio entre `a` y `b` (ambos inclusive).

```
random.uniform(0,1)
```

Ejercicio resuelto: Lanzar monedas

Simular el experimento que sirve para calcular la probabilidad de obtener cara en el lanzamiento de una moneda utilizando `randint()`.

```
import random

def lanzamiento_moneda():
    resultado = random.randint(0, 1)
    if resultado == 0:           # Devolver "cara" si es 0, "cruz" si es 1
        return "cara"
    else:
        return "cruz"

# Realizar 1000 lanzamientos de moneda y contar cuántas veces sale cara
lanzamientos = [lanzamiento_moneda() for i in range(1000)]
caras = lanzamientos.count("cara")

# Calcular la probabilidad de obtener cara
probabilidad = caras / len(lanzamientos)
print("La probabilidad de obtener cara  es:", probabilidad)
```

Una alternativa a la solución anterior, con menos código y más rápida al generar la lista en un solo paso, es usar `random.choices()`.

```
import random

# Realizar 1000 lanzamientos de moneda
lanzamientos = random.choices(["cara", "cruz"], k=1000)

# Calcular la probabilidad de obtener cara
probabilidad = lanzamientos.count("cara") / 1000

print("La probabilidad de obtener cara  es:", probabilidad)
```

Las funciones básicas de `random` son:

Método	Descripción
`seed(n)`	Inicia una secuencia de números aleatorios con la semilla `n` (entero positivo)
`choice(lista)`	Elige aleatoriamente un elemento de `lista`
`choices(lista, k=n)`	Devuelve una lista de `n` elementos seleccionados aleatoriamente con reemplazo
`shuffle(lista)`	Desordena aleatoriamente los elementos de `lista` (modifica la lista original)
`sample(lista, k=n)`	Devuelve una muestra de `n` elementos diferentes de `lista` (sin reemplazo)
`randint(a, b)`	Devuelve un número entero aleatorio entre `a` y `b` (incluidos)

Método	Descripción
`randrange(a, b, s)`	Devuelve un número entero aleatorio en el rango `[a, b)`, con paso `s`
`uniform(a, b)`	Devuelve un número flotante aleatorio entre `a` y `b`
`random()`	Devuelve un número flotante aleatorio en el rango `[0.0, 1.0)`

Algunos ejemplos de su uso son:

```
import random

# Imprime un número aleatorio impar entre 1 y 50
print(random.choice([i for i in range(1, 51) if i % 2 != 0]))

# Imprime un número aleatorio que sea múltiplo de 3 y 4, entre 10 y 100
print(random.choice([i for i in range(10, 101) if i % 3 == 0 and i % 4 == 0]))

# Imprime aleatoriamente una lista con 6 números impares entre 50 y 150
print(random.sample([i for i in range(50, 151) if i % 2 != 0], 6))

# Imprime una lista aleatoria con 4 números que sean múltiplos de 6 o 9,
#  entre 1 y 500
print(random.sample([i for i in range(1, 501) if i % 6 == 0 or i % 9 == 0],4))

# Imprime un número aleatorio primo entre 10 y 100
print(random.choice([i for i in range(10, 101) if all(i % j != 0 \
    for j in range(2, int(i**0.5) + 1))]))
```

Ejercicio resuelto: Dado cargado

Simular el experimento de un dado cargado de manera que la probabilidad de obtener seis es 0.25 y la del resto de números 0.15.

```
import random

def lanzamiento_dado_cargado():
    resultado = random.uniform(0, 1)

    # Asignar la probabilidad correspondiente a cada número
    if resultado < 0.15:
        return 1
    elif resultado < 0.30:
        return 2
    elif resultado < 0.45:
        return 3
    elif resultado < 0.60:
        return 4
    elif resultado < 0.75:
        return 5
    else:
        return 6

# Realizar 1000 lanzamientos del dado cargado y
```

```
# contar cuántas veces sale el número 6
lanzamientos = [lanzamiento_dado_cargado() for i in range(1000)]
seises = lanzamientos.count(6)

# Calcular la probabilidad de obtener el número 6
probabilidad = seises / len(lanzamientos)
print("La probabilidad de obtener el número 6  es:", probabilidad)
```

Las funciones que permiten generar distribuciones de probabilidad son

Función	Distribución	Descripción
`uniform(a, b)`	uniforme	Genera un número flotante aleatorio dentro del rango [a, b], con probabilidad uniforme en todo el intervalo
`gauss(mu, sigma)`	normal (Gaussiana)	Genera un número aleatorio con distribución normal de media `mu` y desviación estándar `sigma`
`normalvariate(mu, sigma)`	normal (Gaussiana)	Hace lo mismo que `gauss()`, pero con una implementación diferente
`expovariate(lambd)`	exponencial	Devuelve un número aleatorio siguiendo una distribución exponencial con parámetro `lambd` (la tasa de ocurrencia)
`weibullvariate(alpha, beta)`	Weibull	Genera un número aleatorio con distribución de Weibull, donde `alpha` es la forma y `beta` la escala
`gammavariate(alpha, beta)`	gamma	Devuelve un número aleatorio con distribución gamma, útil en modelos de tiempo de espera
`betavariate(alpha, beta)`	beta	Genera valores en el rango [0,1] siguiendo una distribución beta, usada en estadística bayesiana
`paretovariate(alpha)`	Pareto	Devuelve un número aleatorio siguiendo una distribución de Pareto
`vonmisesvariate(mu, kappa)`	von Mises	Genera valores siguiendo la distribución de von Mises, usada en datos angulares

Ejercicio resuelto. Generar normal

Generar una lista de 1000 números aleatorios que sigan una distribución normal con media 5 y desviación estándar 2. Luego, el programa debe calcular e imprimir la media y la desviación estándar de los números generados, redondeando ambos valores a 2 decimales.

```
import random
```

```
# Generar 1000 números aleatorios que sigan  una distribución normal de media
# 5 y desviación estándar 2. normalvariate(5, 2).

numeros = [random.normalvariate(5, 2) for i in range(1000)]

# Calcular la media y la desviación estándar de los números generados
media = sum(numeros) / len(numeros)
desviacion_estandar=((sum((x-media)**2 for x in numeros)/(len(numeros)))**1/2
print("La media es",round(media,2),"y la desviación típica es", \
    round(desviacion_estandar, 2))
```

Un script que combina los distintos métodos y que muestra cómo utilizarlos es el siguiente:

```
import random

# 1. random.randint(a, b): Genera un entero aleatorio N tal que a<= N<=b
# Ejemplo: Simular la cantidad de semillas que germinan en una parcela.
# Simula entre 50 y 100 semillas germinadas
cantidad_semillas = random.randint(50, 100)
print(f"Número de semillas germinadas: {cantidad_semillas}")

# 2. random.uniform(a, b): Genera un número de punto flotante
# aleatorio N tal que a <= N <= b.
# Ejemplo: Simular la cantidad de lluvia en milímetros.
# entre 10 y 50 mm de lluvia
cantidad_lluvia = random.uniform(10, 50)
print(f"Cantidad de lluvia (mm): {cantidad_lluvia:.2f}")

# 3. random.gauss(mu, sigma): Genera un número aleatorio de una
# distribución normal.
# Ejemplo: Simular el rendimiento de un cultivo.
# Media 5000 kg/ha, desviación estándar 500 kg/ha
rendimiento_cultivo = random.gauss(5000, 500)
print(f"Rendimiento del cultivo (kg/ha): {rendimiento_cultivo:.2f}")

# 4. random.choices(poblacion, pesos=None, k=1): Devuelve una
# lista de k elementos elegidos de la población con pesos específicos.
# Ejemplo: Simular la probabilidad de diferentes plagas en un cultivo.
tipos_plagas = ['pulgones', 'araña roja', 'trips']
probabilidades_plagas = [0.6, 0.3, 0.1]
# Simula 5 observaciones de plagas
plagas_observadas = random.choices(tipos_plagas, \
     weights=probabilidades_plagas, k=5)
print("Plagas observadas en el cultivo:", plagas_observadas)

# 5. random.shuffle(x): Baraja los elementos de una lista en su lugar.
# Ejemplo: Aleatorizar el orden de aplicación de diferentes
# fertilizantes a distintas parcelas.
fertilizantes = ['Fertilizante A', 'Fertilizante B', 'Fertilizante C']
random.shuffle(fertilizantes)
print("Orden de aplicación de fertilizantes:", fertilizantes)

# 6. random.sample(poblacion, k): Devuelve una lista de k elementos
# únicos elegidos de la población.
# Ejemplo: Seleccionar aleatoriamente un grupo de plantas para un experimento.
plantas = list(range(1, 101))  # Números del 1 al 100 son las plantas
muestra_plantas = random.sample(plantas, 10)  # Selecciona 10 plantas
print("Plantas seleccionadas para el experimento:", muestra_plantas)
```

```
# 7. random.random(): Genera un flotante aleatorio entre 0 y 1
# Ejemplo: Simular la probabilidad de ocurrencia de un evento.
# Probabilidad de sequía en un rango de 0 a 1
probabilidad_sequia = random.random()
print(f"Probabilidad de sequía: {probabilidad_sequia:.3f}")
```

La ejecución del programa anterior devuelve una salida similar a la siguiente:

```
Número de semillas germinadas: 53
Cantidad de lluvia (mm): 46.77
Rendimiento del cultivo (kg/ha): 4733.61
Plagas observadas en el cultivo: ['araña roja', 'araña roja', 'pulgones', 'pulgones', 'pulgones']
Orden de aplicación de fertilizantes: ['Fertilizante C', 'Fertilizante B', 'Fertilizante A']
Plantas seleccionadas para el experimento: [40, 52, 35, 34, 58, 30, 2, 71, 83, 75]
Probabilidad de sequía: 0.049
```

Validación de entrada con funciones y excepciones

Se busca construir una función que diga si la entrada es una entrada válida pero dentro de una función para que pueda reutilizarse. Se muestra el caso de la lectura de un número entero positivo:

```
def comprobar(entero):
    if entero > 0:
        return True
    else:
        return False

# se realizan diversas lecturas para probar la función
while True:
    entero = int(input("Quiero un entero positivo "))
    if comprobar(entero):
        break
    print("dato erroneo")
print("aqui proceso el entero:", entero)
```

O bien, para poder usarlo en una asignación al valor válido:

```
def dameentero(entero):
    if entero > 0:
        return entero
    else:
        return False

while True:
    entero = dameentero(int(input("Quiero un entero positivo ")))
    if entero:
        break
    print("dato erroneo")
print("aqui proceso el entero:", entero)
```

En estas funciones de lectura y validación de datos se pueden incorporar bloques `try` para gestionar la excepcion de error y si es preciso devuelve el entero.

```
def dameentero(entero):
    if entero > 0:
        return entero
    else:
        raise ValueError

while True:
    try:
        entero =dameentero(int(input("Quiero un entero ")))
        if entero>0:
              break
    except ValueError:
        print("dato erroneo")
print("aqui proceso el entero:", entero)
```

Otro ejemplo es una función dentro de la cual se lee un entero asegurándose que está en un rango y que si no es correcto devuelve `None`.

```
def comprobar(entero, ppio, fin):
  try:
    entero=int(entero)
    if entero<=fin and entero>=ppio:
      return  entero
    else:
      return None
  except ValueError:
      return None

numero_leido= comprobar(input("Quiero un entero "), 5, 10)
if numero_leido != None:
  print(numero_leido)
else:
  print("error")
```

Una versión alternativa usando `isnumeric`:

```
def comprobar(entero, ppio, fin):
  if entero.isnumeric():
    entero = int(entero)    # si es un número se convierte
    if entero<=fin and entero>=ppio:
      return  entero
    else:
      return None
  else:
      return None

numero_leido= comprobar(input("Quiero un entero "), 5, 10)
if numero_leido != None:
  print(numero_leido)
else:
  print("error")
```

Problemas resueltos con funciones

Ejercicio resuelto: Función es primo

Determinar si un número entero es primo usando una función.

```
# Determina si un número entero es primo, usando una función.
def es_primo(n):
    for div in range(2, n):
        if n % div == 0:
            return False
    return True

while True:
    numero = int(input('Deme un entero positivo mayor que 1: '))
    if es_primo(numero):
        print(f'El número {numero} es primo.')
    else:
        print(f'El número {numero} no es primo.')
    opcion = input("Desea salir (s/n):")
    if opcion == 's' or opcion == 'S':
            break
```

Ejercicio resuelto: Función producto cartesiano

Crear una función que devuelva una lista resultado del producto cartesiano de las dos listas pasadas como argumento.

```
def producto_cartesiano(lista1, lista2):
    result = []
    for elem1 in lista1:
        for elem2 in lista2:
            result.append((elem1, elem2))
    return result

lista1 = [1, 2, 3, 4]
lista2 = ['a', 'b', 'c']

producto_cartesiano(lista1, lista2)
```

Ejercicio resuelto: Combinatoria

Escribir un programa que calcule el resultado de un número combinatorio.

$\binom{m}{n}$, que se calcula con la fórmula

$\binom{m}{n} = \frac{m!}{n!(m-n)!}$

donde $n!$ es el producto de los números desde 1 a n:

$n! = \prod_{i=1}^{n} i$

Para construir un programa en Python para calcular el factorial se puede escribir:

```
a = int(input("Escribe un número para calcular su factorial: "))
fact = 1

# Cálculo del factorial
for b in range(a, 1, -1):
    fact *= b

print(f"El factorial de {a} es: {fact}")
```

Se debe construir una función que calcule el factorial a la que se llamará tres veces para calcular el combinatorio. Hay identifcar que el factorial es la tarea que se repite y para la que se debe contruir una función que devuelve un valor.

```
def factorial(n):
    """Calcula el factorial de un número de forma iterativa."""
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado

# Pedir y validar n y m
n = int(input("Introduce el valor de n: "))
while n < 0:
    print("n debe ser un número entero no negativo.")
    n = int(input("Introduce el valor de n: "))
m = int(input("Introduce el valor de m: "))
while m < 0 or m > n:
    print("m debe ser un número entero no negativo y no mayor que n.")
    m = int(input("Introduce el valor de m: "))

combinatorio= factorial(n) /(factorial(m)*factorial(n-m))

print("El número de combinaciones de", n, "objetos tomados de" \
        , m, "en", m, "es:", combinatorio)
```

En este caso, la validación de la entrada está mezclada con el cálculo en sí mismo y con la visualización de los datos. Sin embargo, si los datos se obtuviesen de otra manera diferente, la separación en entrada - proceso - salida facilitaría el mantenimiento del código, permitiendo otra forma de utilizar las funciones.

```
def factorial(n):
    """Calcula el factorial de un número de forma iterativa."""
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado

def leer_datos():
    """Solicita al usuario los valores de n y m con validaciones."""
    while True:
        try:
            n = int(input("Introduzca el número de objetos (n): "))
            if n > 0:
                break
```

```
            else:
                print("Error: Debe ser un número positivo.")
        except ValueError:
            print("Error: Debe ingresar un número entero.")

    while True:
        try:
            m = int(input(f"Introduzca m (1-{n}): "))
            if 1 <= m <= n:
                break
            else:
                print(f"Error: El número debe estar en el rango [1, {n}].")
        except ValueError:
            print("Error: Debe ingresar un número entero.")

    return n, m

while True:
    print("\nCOMBINACIONES DE n OBJETOS TOMADOS DE m EN m")
    print("============================================\n")

    n, m = leer_datos()
    combinatorio = factorial(n) / (factorial(m) * factorial(n - m))

    print(f"\nCombinaciones de {n} elementos tomados de {m} en {m} = \
        {int(combinatorio)}")

    continuar = input("\n\n¿Desea efectuar una nueva operación? \
            (s/n): ").strip().lower()
    if continuar == 'n':
        break
```

Ejercicio resuelto: El que falta

Escribir una función que permite buscar los números que faltan en una lista.

- Si se tiene una lista con `0236`.
- Se buscan los números que faltan, `145`.

```
def encuentra_numeros(nums):
    """Función que dada una lista de enteros devuelve los que
    faltan entre el máximo y el mínimo
    """
    min_num = min(nums)
    max_num = max(nums)
    faltan = []
    for i in range(min_num, max_num + 1):
        if i not in nums:
            faltan.append(i)
    return faltan

nums = [0, 2, 3, 6]
faltan = encuentra_numeros(nums)
print(faltan)
```

Ejercicio resuelto: Palabras duplicadas

Si se quiere saber qué palabras aparecen más de una vez en una lista, un conjunto es la mejor opción porque permite comprobar la existencia de un elemento de manera eficiente

```
def palabras_repetidas(texto):
    """Función que indica cuantas y cuales
    son las palabra duplicadas en un texto"""
    palabras = texto.lower().split()    # (sin distinguir mayúsculas)
    vistas = set()
    repetidas = set()
    for palabra in palabras:
        if palabra in vistas:
            repetidas.add(palabra)
        else:
            vistas.add(palabra)
    return repetidas

# Ejemplo de uso
texto = "el sol brilla en el cielo y el cielo es azul"
print(palabras_repetidas(texto))  # {'el', 'cielo'}
```

Ejercicio resuelto: Cálculo vectorial

Diseña un procedimiento para el cálculo vectorial en tres dimensiones que calcule la suma de dos vectores, el producto escalar de dos vectores, y que nos indique si son perpendiculares, así como el producto vectorial.

Para facilitar la entrada por pantalla de la lectura de los vectores, cree una función que permita la entrada de los elementos de cada vector. Para la salida cree otro que lo represente por componentes.

```
# Proporciona constantes y funciones para cálculo vectorial en 3 dimensiones

# Funciones de entrada
def v_lee_vector():
    return [float(input(f'Componente {com}: ')) for com in ['x', 'y', 'z']]

# Funciones de cálculo
def v_suma(u, v):
    return [ u[0] + v[0], u[1] + v[1], u[2] + v[2] ]

def v_producto_escalar(u, v):
    return u[0]*v[0] + u[1]*v[1] + u[2]*v[2]

def v_producto_vectorial(u, v):
    resultado_x = u[1]*v[2] - u[2]*v[1]
    resultado_y = u[2]*v[0] - u[0]*v[2]
    resultado_z = u[0]*v[1] - u[1]*v[0]
    return [resultado_x, resultado_y, resultado_z]

def v_son_perpendiculares(u, v):
    return v_producto_escalar(u, v) == 0

def menu() :
    opcion=0
```

```
    # SALIDA POR PANTALLA
    print("============================")
    print("1. Suma de los dos vectores")
    print("2. Producto escalar")
    print("3. Producto vectorial")
    print("4. Perpendicularidad")
    print("5. Terminar.")
    print("============================")
    print("\nIntroduce los dos vectores: ")
    u = v_lee_vector()
    v = v_lee_vector()

    while opcion != 5 :
        opcion = int(input("Elige una opción: "))

        if opcion == 1 :
            print("La suma de los vectores es ",v_suma(u,v))
        if opcion == 2 :
            print("El producto escalar de los vectores es ",\
                v_producto_escalar(u,v))
        if opcion == 3 :
            print("El producto vectorial de los vectores es ",\
                v_producto_vectorial(u,v))
        if opcion == 4 :
            if v_producto_escalar(u,v)==0 :
                print("Los vectores son perpendiculares")
            else : print("Los vectores no son perpendiculares")
    return opcion

s = menu()
print("Elegiste la opción ", s)
```

Ejercicio resuelto: Productos fitosanitarios

Construir un programa que muestre un sistema de inventario para los productos sanitarios que permita mediante un menú realizar las siguientes acciones:

- Ver el inventario
- Agregar un nuevo producto.
- Actualizar el precio de un producto existente.

```
# Función para mostrar el inventario actual de productos fitosanitarios
def mostrar_inventario():
    print("Inventario actual de productos fitosanitarios:")
    print("----------------------------------------------")
    for producto in productos_fitosanitarios:
        nombre = productos_fitosanitarios[producto]["nombre"]
        fabricante = productos_fitosanitarios[producto]["fabricante"]
        precio = productos_fitosanitarios[producto]["precio"]
        cantidad = productos_fitosanitarios[producto]["cantidad"]
        print(f"{nombre} ({fabricante}): {cantidad} unidades disponibles,\
              precio: {precio} €")

# Función para agregar un nuevo producto al inventario
def agregar_producto():
    print("Agregar un nuevo producto al inventario:")
```

```
    print("----------------------------------------")
    nombre = input("Nombre del producto: ")
    ingredientes_activos = input("Ingredientes activos (separados por \
        comas): ").split(",")
    fabricante = input("Fabricante del producto: ")
    precio = float(input("Precio del producto: "))
    cantidad = int(input("Cantidad disponible del producto: "))

    # Generar un nuevo ID para el producto
    nuevo_id = f"producto{len(productos_fitosanitarios) + 1}"

    productos_fitosanitarios[nuevo_id] = {
        "nombre": nombre,
        "ingredientes_activos": ingredientes_activos,
        "fabricante": fabricante,
        "precio": precio,
        "cantidad": cantidad
    }
    print(f"El producto {nombre} ha sido agregado al inventario con éxito.")

# Función para actualizar el precio de un producto existente en el inventario
def actualizar_precio():
    print("Actualizar el precio de un producto existente en el inventario:")
    print("----------------------------------------------------------------")
    producto_id = input("ID del producto: ")
    if producto_id not in productos_fitosanitarios:
        print("El ID del producto no es válido.")
        return
    nuevo_precio = float(input("Nuevo precio del producto: "))

    productos_fitosanitarios[producto_id]["precio"] = nuevo_precio
    print(f"El precio del producto ha sido actualizado con éxito.")

# Menu
while True:
    print("1. Ver el inventario")
    print("2. Agregar un nuevo producto.")
    print("3. Actualizar el precio de un producto existente")
    print("4. Salir")

    opcion = input("Elige una opción: ")

    if opcion == "1":
       mostrar_inventario()
    elif opcion == "2":
        agregar_producto()
    elif opcion == "3":
        actualizar_precio()
    elif opcion == "4":
        print("Saliendo del programa...")
        break
    else:
        print("Opción inválida, intenta de nuevo")
```

Recursividad

Un **concepto** es **recursivo** si se define en versiones más pequeñas de si mismo.

Por ejemplo:

$$\begin{cases} 1, & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Para estas definiciones no completas, se necesita saber cuándo se acaba la recursividad, es decir, cuando se detiene el proceso o la también llamada *condición de parada*.

En el ámbito de la programación una **función recursiva** es una función que se llama a sí misma durante su propia ejecución. Se comportan de forma similar a las iteraciones, pero hay que fijar la condición de parada o se tratará de una función recursiva infinita. Suele utilizarse para dividir una tarea en subtareas más simples de forma que sea más fácil abordar el problema y solucionarlo.

```
def cuenta_atras(num):
    num -= 1
    if num > 0:
        print(num)
        cuenta_atras(num)
    else:
        print("Boooooooom!")
    print("Fin de la función", num)

cuenta_atras(5)
```

O el cálculo del factorial de un número:

```
def factorial(num):
    print("Valor inicial ->",num)
    if num > 1:
        num = num * factorial(num -1)
    print("valor final ->",num)
    return num

print("\nResultado",factorial(5))
```

La **recursividad** en términos de diseño de programas es un método alternativo a a repetición. Se basa en resolver el mismo problema repetidamente pero sobre un conjunto de datos más pequeño (divide y vencerás), hasta que se satisfaga una condición de terminación.

La ventaja es que se obtiene una solución sencilla a problemas complejos (siempre que el problema admita un planteamiento recursivo que incluya una condición de parada). El inconveniente es que el resultado es una solución (en general) menos eficiente, con mayor tiempo de ejecución (a veces se repiten cálculos innecesarios) y mayor necesidad de espacio en el la memoria de trabajo. Si los datos de partida son grandes el problema se puede desbordar y llenar la memoria.

Ejercicio resuelto: Función máximo común divisor

Construir una función que dados dos números determine el máximo común divisor (MCD).

```
# Halla el MCD de dos números enteros naturales,
# incluido el 0 (Versión iterativa)

def mcd(dividendo, divisor):
    resto = dividendo % divisor      # primer resto fuera del bucle
    while resto != 0:
        dividendo = divisor
        divisor = resto
        resto = dividendo % divisor
    res = divisor
    return res

def lee_natural():
   while True:
       entrada = input("Escribe un número entero: ")
       try:
           entrada = int(entrada)
           if entrada > 0 :
                return entrada
       except ValueError:
           print ("Entrada es incorrecta: escribe un numero entero positivo")

# ENTRADA DE DATOS
print('Introduzca dos enteros, para los que se calculará el MCD.')

dividendo = lee_natural()
divisor = lee_natural()

if dividendo < divisor:  # Asegurar  dividendo  mayor que el divisor
    dividendo, divisor = divisor, dividendo

mcd = mcd(dividendo, divisor)

# SALIDA
print(f'El MCD de {dividendo} y {divisor} es {mcd}.')
```

Solución recursiva

```
# Halla el MCD de dos números enteros naturales,
# incluido el 0 (Versión recursiva)

def mcd(dividendo, divisor):
    if divisor == 0:
        return dividendo
    else:
        return mcd(divisor, dividendo % divisor)

def lee_natural():
   while True:
       entrada = input("Escribe un número entero: ")
       try:
           entrada = int(entrada)
           if entrada > 0:
                return entrada
       except ValueError:
```

```
            print ("Entrada es incorrecta: dame un numero entero positivo")

# ENTRADA DE DATOS
print('Introduzca dos enteros, para los que se calculará el MCD.')

dividendo = lee_natural()
divisor = lee_natural()

if dividendo < divisor:  # Asegurar  dividendo mayor que el divisor
    dividendo, divisor = divisor, dividendo

mcd = mcd(dividendo, divisor)

# SALIDA
print(f'El MCD de {dividendo} y {divisor} es {mcd}.')
```

Ejercicio resuelto: Binomial

Cálculo de la distribución binomial de forma recursiva. La fórmula de la función de probabilidad de la distribución binomial es:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

El coeficiente binomial se puede calcular de forma **recursiva** usando las siguientes dos propiedades de los números combinatorios:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\binom{n}{0} = 1, \quad \binom{n}{n} = 1$$

```
def coef_binomial(n, k):
    if k == 0 or k == n:
        return 1
    return coef_binomial(n - 1, k-1) + coef_binomial(n - 1, k)

def prob_binomial(n, k, p):
    return coef_binomial(n, k) * (p ** k) * ((1 - p) ** (n - k))

# Ejemplo: Probabilidad de obtener exactamente 3 éxitos
# en 5 intentos con p = 0.6
n, k, p = 5, 3, 0.6
print(f"P(X={k}) =", prob_binomial(n, k, p))
```

Herramientas para estadística, probabilidad y tratamiento de datos

El análisis de datos y la realización de operaciones estadísticas son fundamentales en diversas disciplinas, y Python se ha convertido en una de las herramientas más utilizadas para llevar a cabo estas tareas. Su sintaxis sencilla y sus potentes librerías hacen que la programación estadística sea accesible tanto para principiantes como para expertos. Incluso con las funciones integradas del lenguaje, como `sum()` o `len()`, es posible realizar cálculos rápidos y efectivos sobre colecciones de datos, lo que facilita el procesamiento de información y la obtención de resultados inmediatos.

Cuando los desafíos se vuelven más complejos Python también ofrece una amplia variedad de librerías de terceros que facilitan el análisis estadístico avanzado y la manipulación de grandes volúmenes de información. Además, su activa comunidad de usuarios ha desarrollado y compartido numerosos paquetes de código abierto que amplían y mejoran las funcionalidades disponibles.

Este capítulo presenta los conceptos básicos de la estadística descriptiva y la probabilidad, con un enfoque práctico y utilizando Python como herramienta principal. Se explorará cómo las estructuras de datos de Python, junto con librerías, tanto estándar como construidas por terceros, se integran con estos principios para construir una base sólida en el análisis de datos y, posteriormente, en la ciencia de datos.

Cálculo de estadísticos en Python sin uso de librerías

En la programación/cálculo con Python de los distintos estadísticos sobre datos se pueden utilizar diversas estrategias en dos aspectos, por una parte en donde se guardan los datos (estructura de datos) y y por otra en como realizan los cálculos (operaciones y funciones utilizadas).

Una estructura de datos es una forma organizada de guardar y acceder a los datos. Para realizar análisis estadísticos es esencial contar con estructuras de datos que faciliten su almacenamiento y manipulación eficiente. Python ofrece de forma estándar diversas de estas estructuras, como **listas, tuplas, conjuntos y diccionarios**, cada una con sus propias características y ventajas. Si bien la selección óptima y el uso eficiente de las estructuras de datos es un tema avanzado, fuera

del alcance de este libro. los ejemplos se centran en casos prácticos que usan todos estos tipos de datos sin pretender elegir el mejor de ellos en cada caso.

Para realizar los cálculos se pueden utilizar desde las soluciones de la programación clásica que implican el recorrido de las estructuras de datos, bien sea directamente o utilizando compresión de listas hasta la utilización de las operaciones y métodos ya implementados sobre las estructuras de datos como el método `sum()` que es aplicable a listas, tuplas, conjuntos (`set`) y diccionarios (`dict`). En el caso de los diccionarios, si no se especifica lo contrario, `sum()` operará sobre las claves. Sin embargo, para sumar los valores del diccionario, se debe utilizar `sum(diccionario.values())`. También se utiliza el método `sorted()`, teniendo en cuenta que, al aplicarlo a diccionarios, también se ordenarán las claves por defecto.

El enfoque principal de estos ejemplos es la práctica de la programación, utilizando estructuras de datos y métodos conocidos para realizar cálculos. Por lo tanto, no se priorizará la eficiencia computacional en este contexto, sino el desarrollo de habilidades de programación. Como primeros ejemplos se abordan el cálculo de estadísticos de una variable estadística como la media, mediana, moda, cuartiles, percentiles, deciles, quintiles, varianza y desviación típica.

Media:

```
sum(datos) / len(datos)
```

Mediana:

```
datos_ordenados = sorted(datos)
n = len(datos)
if n % 2 == 1:  # Si n es impar
    mediana = datos_ordenados[n // 2]
else:           # Si n es par
    mediana = (datos_ordenados[n // 2 - 1] + datos_ordenados[n // 2]) / 2
```

Moda:

```
frecuencia = {}
for num in datos:
    frecuencia[num] = frecuencia.get(num, 0) + 1
moda = [k for k, v in frecuencia.items() if v == max(frecuencia.values())]
```

Esta solución se basa en el método de un diccionario `frecuencia.get(num, 0)` que busca en el diccionario la clave `num` y devolverá 0 si la clave `num` no existe en el diccionario; y si está devuelve el valor asociado a la clave `num`. Por último, se aplica una comprensión de listas con la condición de que el valor sea el máximo de todas las valores del diccionario frecuencia. La ventaja de esta construcción del estadístico moda permite que si la distribución de frecuencias es bimodal, se devuelva una lista con dos valores para la moda, es decir, se tienen dos valores máximos en el diccionario, y por tanto, se han devuelto dos claves distintas.

Otra alternativa para el cálculo de la moda es:

```
unicos = set(datos)  #  valores únicos - sin considerar las frecuencias
frecuencia = [(num, datos.count(num)) for num in unicos] # (valor,frecuencia)
max_frec = max(frecuencia, key=lambda x: x[1])[1]  # Mayor frecuencia
moda = [num for num, frec in frecuencia if frec == max_frec]
```

Para esta segunda opción, se usa `lambda x: x[1]` tomando como argumento una tupla `x` y devuelve el segundo elemento `[1]` de la tupla, lo que es la *frecuencia* de cada valor. De este modo con `max()[1]` se encuentra la tupla con mayor frecuencia, y seguidamente devuelve su valor. En caso de no ser unimodal, la comprensión de listas devuelve la lista con todas las modas que pudiera tener la distribución de frecuencias.

Cuartiles

```
datos_ordenados = sorted(datos)
n = len(datos)
q1 = datos_ordenados[n // 4]
q2 = datos_ordenados[n // 2]              # Mediana
q3 = datos_ordenados[(3 * n) // 4]
```

Percentiles

```
datos_ordenados = sorted(datos)
n = len(datos)
posicion = (percentil / 100) * (n - 1) # el rango en python es de 0 a n-1
i = int(posicion)       # parte entera del índice de la posición
decimal = posicion - i      # parte decimal de la posición
if i + 1 < n:
    return datos_ordenados[i] + decimal * (datos_ordenados[i + 1] - \
        datos_ordenados[i])
else:
    return datos_ordenados[i]
```

Ordenados los datos, se interpolará si la posición del percentil no es un número entero.

Varianza

```
media = sum(datos) / len(datos)
varianza_poblacion = sum((x - media) ** 2 for x in datos) / len(datos)
varianza_muestral = sum((x - media) ** 2 for x in datos) / (len(datos) - 1)
```

Desviación estándar

```
desviacion_estandar = varianza ** 0.5
```

A continuación se plantean soluciones que se basan en el acceso a los elementos individuales de las estructuras de datos para realizar los cálculos de los estadísticos. Donde se incluyen soluciones con y sin comprensión de listas.

Ejercicio resuelto: Media y desviación

Escribir un programa que pregunte por una muestra de números, separados por comas, los guarde en una lista y muestre por pantalla su media y desviación típica.

Primera Solución

```
texto = input("Introduzca una muestra de números separados por comas: ")
texto = texto.split(',')

n = len(texto)
for i in range(n):
    texto[i] = int(texto[i])

suma = 0
sum2 = 0
for i in texto:
    suma += i
    sum2 += i**2

media = suma/n
stdev = (sum2/n-media**2)**(1/2)

print('La media es', media, ', y la desviación típica es', stdev)
```

Solución con comprensión de listas

```
import math
txt = input("Introduzca una muestra de números separados por comas: ")
txt = txt.split(',')

numeros = [float(numero) for numero in txt]
print(sum(numeros))

media = sum(numeros) / (len(numeros))
suma_cuadrados_diferencias = sum((x - media)**2 for x in numeros)
desviacion_tipica = (suma_cuadrados_diferencias / len(numeros))**0.5

print(f"Media: {media} y Desviación Típica: {desviacion_tipica}")
```

Solucion alternativa (usando operaciones sobre lista)

```
texto = input("Introduzca una muestra de números separados por comas: ")
muestra = [int(numero) for numero in texto.split(',')]

frecuencia_numeros = {}
for numero in muestra:
    if numero in frecuencia_numeros:
        frecuencia_numeros[numero] += 1
    else:
        frecuencia_numeros[numero] = 1

n = len(muestra)
suma = sum(muestra)
sum2 = sum(numero**2 for numero in muestra)
media = suma / n
stdev = ((sum2 / n) - (media**2))**(1/2)

print('La media es', media, ', y la desviación estándar es', stdev)
print('Frecuencia de cada número:', frecuencia_numeros)
```

Ejercicio resuelto. Espacio muestral

Construir un programa que defina el espacio muestral para experimentos de lanzamiento, como el lanzamiento de una moneda o un dado. El programa debe ser generalizable a diferentes tipos de experimentos y solicitar al usuario la siguiente información:

1. Número de elementos en el lanzamiento: Por ejemplo, si se lanzan dos monedas, el número de elementos sería 2.
2. Opciones posibles para cada elemento: Por ejemplo, para una moneda las opciones serían ("Cara", "Cruz"), y para un dado serían (1, 2, 3, 4, 5, 6).

Una vez que el usuario proporciona esta información, el programa debe generar y mostrar el espacio muestral completo del experimento.

Para generar el espacio muestral de un experimento tiene sentido utilizar tuplas para definir los posibles valores de cada suceso.

Solución recursiva

```
def obtener_entrada():
    elementos = int(input("¿Número de elementos en el lanzamiento? "))
    opciones = input("¿Opciones separadas por comas? ").split(',')
    return elementos, tuple(opciones)

def generar_espacio_muestral(opciones, elementos, actual=[]):
    if len(actual) == elementos:
        return [tuple(actual)]
    espacio_muestral = []
    for opcion in opciones:
        # Llamada recursiva para agregar la opción actual
        # y generar las siguientes secuencias
        espacio_muestral.extend(generar_espacio_muestral(opciones, elementos, actual + [opcion]))
    return espacio_muestral

def mostrar_resultado(espacio_muestral):
    print("Espacio muestral:")
    for resultado in espacio_muestral:
        print(resultado)

def main():
    elementos, opciones = obtener_entrada()
    espacio_muestral = generar_espacio_muestral(opciones, elementos)
    mostrar_resultado(espacio_muestral)

if __name__ == "__main__":
    main()
```

Solución iterativa (opción no recursiva de la función `generar_espacio_muestral)`.

En esta versión se inicializa el espacio muestral con una lista que contiene una lista vacía y con bucles anidados se itera `elementos` veces para seleccionar cada elemento, se itera sobre las secuencias existentes, sobre las opciones disponibles y se agrega a la secuencia añadiendo al nuevo espacio, convirtiéndola en tupla.

```
def generar_espacio_muestral(opciones, elementos):
    espacio_muestral = [[]]
    for _ in range(elementos):
        nuevo_espacio = []
        for secuencia in espacio_muestral:
            for opcion in opciones:
                nuevo_espacio.append(secuencia + [opcion])
        espacio_muestral = nuevo_espacio
    return [tuple(secuencia) for secuencia in espacio_muestral]
```

Se puede aprovechar la potencialidad de los conjuntos para generar los valores de una lista de ocurrencias.

Ejercicio resuelto. Conteo de frecuencias

Construye un programa para realizar el conteo de las frecuencias de una lista.

```
def valores_unicos(numeros):
    return list(set(numeros))

def contar(lista):
    listUnica = valores_unicos(lista)
    frec = []
    for valor in listUnica:
        frec.append(lista.count(valor))
    return listUnica, frec

# Entrada de datos
lista = [18, 32, 20, 18, 30, 35, 24, 33, 32, 30, 20, 20, 24, 20, 30]

# Obtener valores únicos y contar frecuencias
valores, frecuencias = contar(lista)

# Salida de resultados
print("xi | ni")
print("---+---")
for xi, ni in zip(valores, frecuencias):
    print(xi,"|", ni)
```

Ejercicio resuelto. Problema clásico de probabilidad

Se realiza el lanzamiento de un dado veinte veces y se obtienen los resultados: 1, 3, 6, 4, 2, 5, 6, 1, 3, 4, 5, 2, 6, 1, 3, 2, 4, 5, 6, 1. Se pide calcular la frecuencia de cada resultado utilizando diccionarios para almacenar los valores de la variable y sus frecuencias -pares (x_i, n_i)-, así como la probabilidad frecuentista de cada resultado.

La definición del espacio muestral del dado se hace utilizando `set` ya que no permite duplicados y es más eficiente.

```
# Definición del espacio muestral (posibles resultados del dado)
espacio_muestral = {1, 2, 3, 4, 5, 6}

# Lanzamiento del dado (supongamos que es justo)
resultados_lanzamiento = [1, 3, 6, 4, 2, 5, 6, 1, 3, 4, 5, 2, 6, 1, 3, 2, 4, 5, 6, 1]

# Calcula la frecuencia de cada resultado
frecuencia = {}
for elemento in espacio_muestral:
    frecuencia[elemento] = resultados_lanzamiento.count(resultado)

# Calcula la probabilidad de cada resultado
total_lanzamientos = len(resultados_lanzamiento)
probabilidad_resultados = {resultado: frec / total_lanzamientos \
    for resultado, frec in frecuencia.items()}

print("Frecuencia de cada resultado:", frecuencia)
print("Probabilidad de cada resultado:", probabilidad_resultados)
```

A continuación se incluye una solución más eficiente puesto que se mejora el cálculo de frecuencias en una única iteración. Así, con esta cambio, se hace en una sola pasada sobre la lista y se reduce la complejidad. Finalmente, se calcula la probabilidad de cada resultado en otro diccionario.

```
espacio_muestral = {1, 2, 3, 4, 5, 6}

resultados_lanzamiento = [1, 3, 6, 4, 2, 5, 6, 1, 3, 4, 5, 2, 6, 1, 3, 2, 4, 5, 6, 1]

# Inicializar el diccionario de frecuencias con ceros
frecuencia = {num: 0 for num in espacio_muestral}
```

```
# Recorrer los resultados para contar las frecuencias
for resultado in resultados_lanzamiento:
    frecuencia[resultado] += 1

total_lanzamientos = len(resultados_lanzamiento)
probabilidad_resultados = {num: frecuencia[num] / total_lanzamientos for num in espacio_muestral}

print("Frecuencia de cada resultado:", frecuencia)
print("Probabilidad de cada resultado:", probabilidad_resultados)
```

Este problema se puede hacer más eficiente utilizando *tuplas* en lugar de conjuntos y diccionarios. Las tuplas mejoran el rendimiento en algunos casos, especialmente cuando el acceso a los datos es constante y no se necesita modificar la estructura. Así pues, el código optimizado siguiente hace más rápido y ocupa menos memoria, sobre todo si el número de lanzamientos es muy grande. Además, en la versión final siguiente se utiliza una validación para evitar errores en caso de valores inesperados en los resultados de los lanzamientos.

```
espacio_muestral = (1, 2, 3, 4, 5, 6)

resultados_lanzamiento = (1, 3, 6, 4, 2, 5, 6, 1, 3, 4, 5, 2,  6, 1, 3, 2, 4, 5, 6, 1)

# Inicializar una lista de frecuencias (posición 0 para el 1,
# posición 1 para el 2, etc.)
frecuencia = [0] * len(espacio_muestral)

# Contamos las frecuencias recorriendo la tupla de resultados
for resultado in resultados_lanzamiento:
    if 1 <= resultado <= len(espacio_muestral):  # Validación del rango
        frecuencia[resultado - 1] += 1
    else:
        print(f"Aviso: valor {resultado} fuera del rango esperado.")

total_lanzamientos = len(resultados_lanzamiento)
probabilidad_resultados = tuple(f / total_lanzamientos for f in frecuencia)

print("Frecuencia de cada resultado:", list(zip(espacio_muestral, frecuencia)))
print("Probabilidad de cada resultado:", list(zip(espacio_muestral, probabilidad_resultados)))
```

El uso de las prestaciones de generación de números aleatorios es básica para el tratamiento de datos y sobre todo la simulación de experimentos. También se pueden utilizar para generar distintos tipos de datos para calcular la tabla de frecuencias.

Ejercicio resuelto. Experimento lanzamiento

Simular el experimento del conteo de la obtención del número 4 en el lanzamiento de un dado n-veces con el módulo `random`.

```
# Simular lanzamientos de un dado
from random import *

n=int(input('Cantidad de lanzamientos: '))
contador=0
for i in range(n):
    x=randint(1,6)
    if x==4:
        contador=contador+1
print('Conteo de resultados favorables: ',contador)

# Calcular la probabilidad frecuentista de obtener cuatro es
probabilidad = contador / n
print(f"Probabilidad frecuentista de obtener 4 en el lanzamiento de {n} dado es: {probabilidad}")
```

Ejercicio resuelto. Generación de datos para conteos

```
import random

# Generar datos aleatorios en el rango 1-10
datos_aleatorios_lista = [random.randint(1, 10) for _ in range(20)]

# Inicializar lista de frecuencias con ceros
frecuencia_lista = [0] * 10  # Índices de 0 a 9,  números del 1 al 10

# Contar frecuencias en una sola pasada
for dato in datos_aleatorios_lista:
    frecuencia_lista[dato - 1] += 1  # `dato - 1` ajusta al índice correcto

# Valores correctos en el rango 1-10
valores = list(range(1, 11))

print("Datos Aleatorios (Lista):", datos_aleatorios_lista)
print("Frecuencia (Lista):", frecuencia_lista)

# Imprimir la frecuencia por cada número
for i in range(len(valores)):
    print(f"{valores[i]}: {frecuencia_lista[i]}")
```

Con las soluciones anteriores se ha querido mostrar al lector una exploración de diversos tipos de datos (como listas, tuplas y conjuntos), así como distintas estrategias para calcular frecuencias de manera eficiente. Además, se ha comentado cómo optimizar el rendimiento de los cálculos utilizando estructuras de datos adecuadas y evitando operaciones innecesarias, así como el uso repetido de `.count()`.

Estas prácticas no solo mejoran la eficiencia del código, sino que también proporcionan una comprensión más profunda de las estructuras de datos y su aplicación en problemas reales.

Librería itertools: combinaciones y permutaciones

Habiendo explorado las estructuras de datos básicas de Python y su aplicación en el cálculo de frecuencias, ahora se utilizará la librería `itertools`, una herramienta poderosa que permite generar iteradores que representan estas estructuras combinatorias de forma eficiente, proporcionando funciones especializadas para generar:

- **Permutaciones:** Ordenaciones de un conjunto de elementos, donde el orden importa. Por ejemplo, las permutaciones de las letras "ABC" serían: "ABC", "ACB", "BAC", "BCA", "CAB", "CBA".
- **Combinaciones:** Subconjuntos de un conjunto de elementos, donde el orden no importa. Por ejemplo, las combinaciones de 2 letras del conjunto "ABC" serían: "AB", "AC", "BC".

A través de ejemplos prácticos, se ve cómo utilizar las funciones `permutations` y `combinations` de `itertools` para generar y manipular permutaciones y combinaciones. Esto permitirá resolver problemas que involucran la selección y ordenación de elementos, ampliando nuestras habilidades en el análisis de datos y la resolución de problemas con Python.

Además de los ejercicios que se han construido anteriormente utilizando recursividad, se puede

utilizar los módulos `math` y el módulo `itertools` tanto para el cálculo del número de ellas como para generarlas. Previo al desarrollo de algunos ejercicios resueltos se indica lo siguiente para el uso de `itertools`.

Tipo	Función en `itertools`	Fórmula
Permutaciones Sin Repetición	`permutations(it, k)`	$P(n,k) = \frac{n!}{(n-k)!}$
Permutaciones Con Repetición	`set(permutations(it))`	$\frac{n!}{n_1! \cdot n_2! \cdot \ldots \cdot 1!}$
Combinaciones Sin Repetición	`combinations(it, k)`	$C(n,k) = \frac{n!}{k!(n-k)!}$
Combinaciones Con Repetición	`combinations_with_replacement(it, k)`	$CR(n,k) = \frac{(n+k-1)!}{k! \cdot (n-1)!}$
Variaciones Sin Repetición	`permutations(it, k)`	$V(n,k) = \frac{n!}{(n-k)!}$
Variaciones Con Repetición	`product(it, repeat=k)`	$VR(n,k) = n^k$

Ejercicio resuelto. Dado un grupo de tres letras A, B, C ¿de cuántas maneras se pueden ordenar?

Como siempre se tienen dos formas para hacerlo, directamente codificando o bien utilizando un módulo ya desarrollado, en este caso `itertools`.

```
def permutar(elementos, inicio=0):
    if inicio == len(elementos) - 1:
        print(tuple(elementos))
        return

    for i in range(inicio, len(elementos)):
        # Intercambiar elementos
        elementos[inicio], elementos[i] = elementos[i], elementos[inicio]
        permutar(elementos, inicio + 1)  # Llamada recursiva
        # Deshacer el intercambio (backtracking)
        elementos[inicio], elementos[i] = elementos[i], elementos[inicio]

# Lista de elementos
elementos = ['A', 'B', 'C']

print("Permutaciones sin repetición:")
permutar(elementos)
```

Solución alternativa:

```
from itertools import permutations

elementos = ['A', 'B', 'C']
permutaciones = list(permutations(elementos))

print("Permutaciones sin repetición:")
for p in permutaciones:
    print(p)
```

Ejercicio resuelto. ¿Cuántas formas hay de ordenar la palabra "CASA"?

```
def factorial(n):
    resultado = 1
    for i in range(2, n + 1):
        resultado *= i
    return resultado

def permutaciones_con_repeticion(n, repeticiones):
    numerador = factorial(n)
    denominador = 1
    for r in repeticiones:
        denominador *= factorial(r)

    return numerador // denominador

# Datos del problema: 4 elementos (2,1,1)
n = 4
repeticiones = [2, 1, 1]

# Cálculo
resultado = permutaciones_con_repeticion(n, repeticiones)
print("Número de permutaciones con repetición:", resultado)
```

Una alternativa aquí es utilizar `math` para realizar el conteo, con lo que se omitiría la función `factorial(n)` y bastaría sólo llamar a `math.factorial()`.

```
import math

def permutaciones_con_repeticion(n, repeticiones):
    numerador = math.factorial(n)
    denominador = math.prod(math.factorial(r) for r in repeticiones)
    return numerador // denominador

n = 4
repeticiones = [2, 1, 1]

resultado = permutaciones_con_repeticion(n, repeticiones)
print("Número de permutaciones con repetición:", resultado)
```

Si además del conteo, se quiere calcular cuáles son éstas, se recurre a la libreria `itertools`.

```
from itertools import permutations  # Permutaciones con repetición

elementos = "CASA"
permutaciones = set(permutations(elementos))
print("Permutaciones únicas:")
for p in permutaciones:
    print("".join(p))
```

Ejercicio resuelto. En un examen de 5 preguntas de opción múltiple, ¿cuántas formas hay de elegir 2 preguntas para responder?

```
def factorial(n):
    resultado = 1
    for i in range(2, n + 1):
        resultado *= i
    return resultado

def combinaciones(n, k):
    return factorial(n) // (factorial(k) * factorial(n - k))

n, k = 5, 2                  # Ejemplo
print(f"C({n}, {k}) =", combinaciones(n, k))  # Debe imprimir C(5,2) = 10
```

Solución alternativa:

```
from math import comb
```

```
n, k = 5, 2
print(f"C({n}, {k}) =", comb(n, k))  # C(5,2) = 10
```

Si esas preguntas tienen tres opciones de respuesta, ¿cuántas posibles formas de mostrar las opciones de respuesta hay si se baraja el orden de presentación de cada una de las opciones?

```
from itertools import permutations

# Opciones de respuestas
respuestas = ["A", "B", "C"]

# Generar todas las permutaciones posibles de 3 elementos
# (ya que solo hay 3 respuestas únicas)
todas_las_parejas = list(permutations(respuestas, 3))

# Contar cuántas hay
total_parejas = len(todas_las_parejas)

# Mostrar las combinaciones generadas con etiquetas
print(f"Número total de parejas: {total_parejas}\n")
for i, pareja in enumerate(todas_las_parejas, 1):
    print(f"Pareja {i}: {''.join(pareja)}")
```

Ejercicio resuelto. En el lanzamiento de un dado, se necesita encontrar todas las combinaciones posibles de un número dado de lanzamientos de dados (por ejemplo, 3 lanzamientos), con la restricción de que la suma total de los lanzamientos debe ser mayor que un valor dado (por ejemplo, 10). Calcula cuántas de esas combinaciones cumplen con esta condición y calcula la **probabilidad** de que la suma de los lanzamientos sea mayor que el valor indicado.

```
import itertools

def calcular_combinaciones_y_probabilidad(num_dados, suma_superar):
    # Los resultados posibles de un lanzamiento de dado de 6 caras
    dados = [1, 2, 3, 4, 5, 6]

    # Generar todas las combinaciones posibles de `num_dados` lanzamientos
    # con repetición permitida)
    combinaciones = list(itertools.product(dados, repeat=num_dados))

    # Filtrar las combinaciones cuya suma sea mayor que la suma proporcionada
    comb_validas = [combinacion for combinacion in combinaciones \
        if sum(combinacion) > suma_superar]
    print("combinaciones validas: \n", comb_validas)

    # Número total de combinaciones posibles
    total_combinaciones = len(combinaciones)

    # Número de combinaciones válidas
    comb_validas_count = len(comb_validas)

    # Calcular la probabilidad
    probabilidad = comb_validas_count / total_combinaciones

    # Mostrar resultados
    print(f"Total de combinaciones posibles: {total_combinaciones}")
    print(f"Combinaciones cuya suma es mayor que {suma_superar}: {comb_validas_count}")
    print(f"Probabilidad de que la suma sea mayor que {suma_superar}:{probabilidad:.4f}\n")

    return comb_validas_count, probabilidad

def main():
    while True:
        try:
            num_dados = int(input("Deme el número de dados a lanzar: "))
            suma_superar = int(input("Deme la suma mínima a superar: "))
        except ValueError:
            print("Por favor, introduzca números enteros válidos.")
            continue
```

```
        # Llamar a la función para calcular combinaciones y probabilidad
        calcular_combinaciones_y_probabilidad(num_dados, suma_superar)

        # Preguntar al usuario si desea realizar otro cálculo
        continuar = input("¿Desea realizar otro cálculo? (s/n): ").lower()
        if continuar != 's':
            print("¡Gracias por usar el programa!")
            break

# Ejecutar el programa principal
main()
```

Ejercicio resuelto. Experimento dado

En el experimento aleatorio de lanzar un dado, calcular las probabilidades de ser par, impar y la probabilidad de la intersección de sucesos.

Solución sin comprensión de listas

```
E = list(range(1, 7))  # Espacio muestral
n = len(E)  # Total de la muestra

# Suceso A: Probabilidad de ser par
A = []
for i in E:
    if i % 2 == 0:
        A.append(i)
pA = len(A) / float(n)
print(f"La probabilidad de ser par es {pA}")

# Suceso B: Probabilidad de ser impar
B = []
for i in E:
    if i % 2 != 0:
        B.append(i)
pB = len(B) / float(n)
print(f"La probabilidad de ser impar es {pB}")

# Probabilidad de la intersección de sucesos A y B
# (sucesos mutuamente excluyentes)
pAB = 0  # Dado que A y B son mutuamente excluyentes
print(f"La probabilidad de la intersección de A y B es {pAB}")
```

Solución usando comprensión de listas

```
E = list(range(1, 7))  # Espacio muestral
n = len(E)  # Total de la muestra

# Suceso A: Probabilidad de ser par
A = [i for i in E if i % 2 == 0]
pA = len(A) / float(n)
print(f"La probabilidad de ser par es {pA}")

# Suceso B: Probabilidad de ser impar
B = [i for i in E if i % 2 != 0]
pB = len(B) / float(n)
print(f"La probabilidad de ser impar es {pB}")

# Probabilidad de la intersección de sucesos A y B
# (sucesos mutuamente excluyentes)
pAB = 0  # Dado que A y B son mutuamente excluyentes
print(f"La probabilidad de la intersección de A y B es {pAB}")
```

Librería statistics

El paquete `statistics` es un módulo útil, aunque poco conocido, en las librerías estándar de Python. Proporciona funciones que permiten calcular casi todos los valores estadísticos, como la media, la covarianza, etc. Para cálculos estadísticos sencillos, en lugar de instalar una biblioteca externa como `NumPy`, se puede usar este módulo integrado.

Se incluye un script de muestra de las funciones incorporadas en esta librería:

```
import random
import statistics as st

numeros = [random.randint(1, 100) for _ in range(10)]
print("Lista de números generada:", numeros)

# Promedios y medidas de ubicación central
print("Promedio:", st.mean(numeros))
print("Promedio para números:", st.fmean(numeros))
print("Media geométrica:", st.geometric_mean(numeros))
print("Media armónica:", st.harmonic_mean(numeros))

# Mediana o medida de tendencia central
numOrd = sorted(numeros)
print("Mediana:", st.median(numOrd))
print("Mediana inferior:", st.median_low(numOrd))
print("Mediana superior:", st.median_high(numOrd))
print("Mediana agrupada:", st.median_grouped(numbers))

# Moda y cuartiles**
print("Cuartiles:", st.quantiles(numOrd))
num = [1, 1, 1, 2, 3, 3, 4, 4, 4, 5, 5]
print("Moda:", st.mode(num))
print("Multimoda:", st.multimode(num))

# Varianza y desviación estándar
print("Varianza poblacional:", st.pvariance(numOrd))
print("Desviación estándar poblacional:", st.pstdev(numOrd))
print("Varianza muestral:", st.variance(numOrd))
print("Desviación estándar muestral:", st.stdev(numOrd))
```

Relación entre dos entradas

También se puede estudiar la relación entre dos valores mediante la correlación y la regresión. Un modelo de regresión lineal trata de encontrar una línea que describa cómo dato conocido afecta a la otro que se quiera predecir La regresión busca descubrir cómo una variable influye en otra para hacer predicciones.

```
import statistics as st

valores_x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
valores_y = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

print("Covarianza:", st.covariance(valores_x, valores_y))
print("Correlación:", st.correlation(valores_x, valores_y))

pendiente, interseccion = st.linear_regression(valores_x, valores_y)
print("Pendiente:", pendiente, "Intersección:", interseccion)
```

Ejercicio resuelto. Un agricultor, que en cada cosecha registra la cantidad de fertilizante (en kg/ha) aplicada a diferentes parcelas de un cultivo y el rendimiento obtenido (en toneladas/ha) en cada una de ellas, necesita un programa que le permita analizar la relación entre estas variables y predecir el rendimiento en futuras cosechas.

Los datos de una cosecha de ejemplo se encuentran en las siguientes listas:

- fertilizante_kg_ha =[100, 150, 200, 250, 300, 350, 400, 450, 500, 550]
- rendimiento_ton_ha =[2.5,3.0,3.2,3.5,3.7,3.8,3.9,3.95,4.0,4.0]

Se ha de diseñar un programa que:

1. Calcule la correlación entre la cantidad de fertilizante aplicado y el rendimiento del cultivo.
2. Determine la recta de regresión que mejor representa la relación entre estas variables.
3. Implemente una función que interprete el valor de la correlación según la escala:
   - mayor que 0.95: Correlación muy fuerte y positiva.
   - entre 0.85 - 0.95: Correlación fuerte y positiva.
   - entre 0.70 - 0.85: Correlación moderadamente fuerte y positiva.
   - entre 0.50 - 0.70: Correlación moderada y positiva.
   - entre 0.30 - 0.50: Correlación débil y positiva.
   - entre -0.30 - 0.30: Correlación muy débil o nula.
   - entre -0.50 y -0.30: Correlación débil y negativa.
   - entre -0.70 y -0.50: Correlación moderada y negativa.
   - entre -0.85 y -0.70: Correlación moderadamente fuerte y negativa.
   - entre -0.95 y -0.85: Correlación fuerte y negativa.
   - menor que -0.95: Correlación muy fuerte y negativa.
4. Utilice un diccionario para el tratamiento de los datos.
5. Presente los resultados de forma clara e interprete la correlación obtenida, ya que el agricultor utilizará este programa para analizar los datos de cada cosecha.

```
import statistics as st
datos = {
    'fertilizante_kg_ha': [100, 150, 200, 250, 300, 350, 400, 450, 500, 550],
    'rendimiento_ton_ha': [2.5, 3.0, 3.2, 3.5, 3.7, 3.8, 3.9, 3.95, 4.0, 4.0]
}

# 1. Calcular la covarianza y la correlación
covarianza = st.covariance(datos['fertilizante_kg_ha'], datos['rendimiento_ton_ha'])
correlacion = st.correlation(datos['fertilizante_kg_ha'], datos['rendimiento_ton_ha'])

# 2. Regresión lineal
pendiente, interseccion = st.linear_regression(datos['fertilizante_kg_ha'], \
    datos['rendimiento_ton_ha'])

# 3. Imprimir resultados con formato y claridad
print(f"Análisis de la relación entre fertilizante y rendimiento:\n")
print(f"Covarianza: {covarianza:.4f}")
print(f"Correlación: {correlacion:.4f}")
print(f"Pendiente de la regresión: {pendiente:.4f}")
print(f"Intersección de la regresión: {interseccion:.4f}\n")

# 4. Interpretación de la correlación (usando una función para modularidad)
def interpretar_correlacion(correlacion):
    if correlacion > 0.95:
        return "muy fuerte y positiva, indicando una relación casi perfecta entre las variables."
    elif 0.85 <= correlacion <= 0.95:
        return "fuerte y positiva, indicando una relación significativa  entre las variables."
    elif 0.70 <= correlacion < 0.85:
        return "moderadamente fuerte y positiva, indicando una relación notable entre \
            fertilizante y rendimiento."
    elif 0.50 <= correlacion < 0.70:
        return "moderada y positiva, indicando una relación perceptible entre fertilizante \
```

```
            y rendimiento."
    elif 0.30 <= correlacion < 0.50:
        return "débil y positiva, indicando una relación poco clara entre fertilizante y \
             rendimiento."
    elif -0.30 < correlacion < 0.30:
        return "muy débil o nula, indicando que la relación entre fertilizante y rendimiento \
             es prácticamente inexistente."
    elif -0.50 <= correlacion <= -0.30:
        return "débil y negativa, indicando una ligera tendencia a que el rendimiento \
             disminuya al aumentar el fertilizante."
    elif -0.70 <= correlacion < -0.50:
        return "moderada y negativa, indicando una tendencia notable a  que el rendimiento \
             disminuya al aumentar el fertilizante."
    elif -0.85 <= correlacion < -0.70:
        return "moderadamente fuerte y negativa, indicando una tendencia significativa a que \
             el rendimiento disminuya al aumentar  el fertilizante."
    elif -0.95 <= correlacion < -0.85:
        return "fuerte y negativa, indicando una fuerte tendencia a que el rendimiento \
            disminuya al aumentar el fertilizante."
    elif corre lacion <= -0.95:
        return "muy fuerte y negativa, indicando una relación casi perfecta inversa entre \
            fertilizante y rendimiento."

interpretacion_correlacion = interpretar_correlacion(correlacion)

# 5. Imprimir interpretación completa
print(f"Interpretación de los resultados:")
print(f"---------------------------------")
print(f"Un valor positivo indica una relación directa (a mayor fertilizante, mayor \
     rendimiento).")
print(f"En este caso, la covarianza es {covarianza:.4f} y la correlación es {correlacion:.4f},\
     indicando una correlación {interpretacion_correlacion}")
```

Para completar este apartado sería conveniente utilizar un gráfico que representara los datos, es aquí donde cabe mostrar un diagrama de punto y la recta de regresión, pero aún no se han detallado las librerías que contienen elementos gráficos. Se verá más tarde.

Ejercicio resuelto. Con este ejercicio se analiza la relación entre el período orbital (en días) y la distancia al Sol (en millones de kilómetros) de los planetas del sistema solar.

Primero, se ha de verificar si existe una relación monótona perfecta entre ambas variables utilizando la correlación por rangos. Luego, se observa que, aunque la correlación lineal es muy alta, no es perfecta. Finalmente, se aplica la tercera ley de Kepler, que establece que el cuadrado del período orbital es proporcional al cubo de la distancia al Sol, demostrando que esta relación es lineal cuando se ajustan los valores adecuadamente.

Planeta	Período Orbital (días)	Distancia al Sol (millones de km)
Mercurio	88	58
Venus	225	108
Tierra	365	150
Marte	687	228
Júpiter	4331	778
Saturno	10756	1400
Urano	30687	2900
Neptuno	60190	4500

Planeta enano	Período Orbital(días)	Distancia real al Sol (millones de km)
Plutón	90560	5906
Eris	204199	10152
Makemake	111845	6796
Haumea	103410	6450
Ceres	1680	414

```
import statistics as st

# Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano y Neptuno
periodo_orbital_planetas = [88,225,365,687,4331,10756,30687,60190]
distancia_al_sol_planetas = [58,108,150,228,778,1400,2900,4500]

# Observar que la relación lineal (correlación de Pearson) no es perfecta
correlacion_pearson = st.correlation(periodo_orbital_planetas, distancia_al_sol_planetas)
print(f"Correlación de Pearson (periodo orbital y distancia al sol): {correlacion_pearson:.4f}")
# Resultado: 0.9882

# Calcular el cuadrado del periodo orbital
periodo_orbital_al_cuadrado = [p * p for p in periodo_orbital_planetas]
# Calcular el cubo de la distancia al sol
distancia_al_sol_al_cubo = [d * d * d for d in distancia_al_sol_planetas]

# Demostrar la tercera ley de Kepler: Existe una correlación lineal
# entre el cuadrado del período orbital y el cubo de la distancia al sol.
correlacion_kepler = st.correlation(periodo_orbital_al_cuadrado, distancia_al_sol_al_cubo)
print(f"Correlación entre el cuadrado del periodo orbital y el cubo de la distancia: \
        {correlacion_kepler:.4f}")  # Resultado: 1.0

# --- Predicción de distancias para planetas enanos ---
# Define una función para la regresión lineal proporcional
def linear_regression(x, y, proportional=False):
    # similar a la de statistics pero con opción de proportional=True
    # para indicar si es proporcional (y = mx)
    n = len(x)
    sum_x = sum(x)
    sum_y = sum(y)
    sum_x2 = sum(xi * xi for xi in x)
    sum_xy = sum(xi * yi for xi, yi in zip(x, y))

    if proportional:
        pendiente = sum_xy / sum_x2
        interseccion = 0
    else:
        pendiente = (n*sum_xy - sum_x*sum_y) / (n*sum_x2 - sum_x*sum_x)
        interseccion = (sum_y - pendiente * sum_x) / n
    return pendiente, interseccion

# Datos de planetas enanos: Plutón, Eris, Makemake, Haumea, Ceres
# Períodos orbitales en días
periodo_orbital_planetas_enanos = [90_560, 204_199, 111_845, 103_410, 1_680]
# Usamos la regresión lineal proporcional para obtener la pendiente
pendiente, _ = linear_regression(periodo_orbital_al_cuadrado, distancia_al_sol_al_cubo, \
     proportional=True)

# Predecir las distancias al sol para los planetas enanos usando la tercera
# ley de Kepler. Se usa una función lambda y round en lugar de math.cbrt
distancia_al_sol_predicciones = [round((pendiente * (p * p))**(1/3)) \
    for p in periodo_orbital_planetas_enanos]

print("Distancias al sol predichas para planetas enanos (millones de km):", \
     distancia_al_sol_predicciones)    # [5912, 10166, 6806, 6459, 414]

# Distancias reales al sol en millones de km
distancias_reales = [5_906, 10_152, 6_796, 6_450, 414]
print("Distancias reales al sol para planetas enanos (millones de km):", distancias_reales)
```

Librerías de terceros

Los principales paquetes de terceros que ofrece Python para trabajar con estadística y probabilidad son los siguientes, algunas se han convertido en un estándar de facto como NumPy y Pandas:

- `NumPy`, Numerical Python, es uno de los paquetes más importantes para la computación numérica en Python. Siendo las matrices numéricas su principal aportación.
- `scipy.stats`, este submódulo del paquete científico `Scipy` una librería para el análisis estadístico. Proporciona herramientas para la estimación de parámetros, la generación de muestras aleatorias, las pruebas de hipótesis y la modelización de distribuciones estadísticas.
- `Statsmodels`, es una librería para el modelado estadístico. Proporciona funciones para la estimación de modelos lineales, modelos de series de tiempo, modelos de regresión y modelos de supervivencia. También tiene funciones para la evaluación del ajuste del modelo y la inferencia estadística.
- `Pandas`, centrada en el análisis de datos su principal aportación son los tipos de datos Series y DataFrame. Adopta partes significativas del estilo basada en matrices de NumPy. Posee algunas funciones muy útiles para realizar estadística descriptiva sobre datos. También tiene funciones para la lectura y escritura de datos en diferentes formatos, como CSV, Excel y bases de datos SQL.
- `Matplotlib`, es la librería que se centra en las visualizaciones y gráficos que junto a `Seaborn` proporciona opciones avanzadas para la creación de gráficos estadísticos complejos, como diagramas de violín, gráficos de regresión y mapas de calor.

Librería	Ventajas	Diferencias
NumPy	- Permite trabajar con grandes conjuntos de datos en formato de array eficientemente - Incluye funciones para realizar cálculos matemáticos complejos - Ofrece herramientas para el procesamiento de imágenes y señales. - Permite trabajar con datos en formato de texto, imagen o audio.	- Para la manipulación de vectores multidimensionales. - No tiene herramientas para el análisis estadístico
SciPy.stats	- Proporciona herramientas para el análisis estadístico - Ofrece una amplia variedad de distribuciones de probabilidad y funciones de ajuste - Permite realizar pruebas de hipótesis y análisis de correlación.	- No es una herramienta de visualización de datos - Para el análisis estadístico y de probabilidad, sin herramientas para la manipulación de arrays.

Librería	Ventajas	Diferencias
	- Incluye herramientas para el análisis de regresión y ANOVA.	
Statsmodels	- Ofrece herramientas para el análisis estadístico avanzado - Proporciona herramientas para el modelado y análisis de series temporales - Incluye herramientas para el análisis de regresión y ANOVA - Permite trabajar con datos en formato de tabla	- No es una herramienta de visualización - Para análisis estadístico sin herramientas para la manipulación de arrays multidimensionales
Pandas	- Permite trabajar con grandes conjuntos de datos en formato de tabla de manera eficiente - Proporciona herramientas para la manipulación de datos y la limpieza de datos - Ofrece herramientas para el análisis estadístico y de series temporales - Permite realizar operaciones de agrupación y combinación - Proporciona herramientas para la visualización	- No enfocada en el análisis estadístico avanzado - No proporciona herramientas para el procesamiento de señales o imágenes
Matplotlib	- Proporciona herramientas para la creación de gráficos y visualización 2D - Ofrece una amplia variedad de opciones de personalización de gráficos - Muy flexible y permite la creación de gráficos de alta calidad	- Sin herramientas para el análisis estadístico - No enfocada en la manipulación de arrays multidimensionales

Si no están instalados en necesario utilizar desde la consola `pip` (el gestor de paquetes que permite instalar, actualizar y administrar librerías y paquetes de terceros):

```
pip install <nombre_librería>
```

Cada una de estas librerías tiene ventajas y limitaciones, por lo que la elección dependerá de las necesidades específicas del proyecto. En términos generales, **NumPy** es la base para el análisis numérico en Python, mientras que **Pandas** y **Matplotlib** son fundamentales para la manipulación y visualización de datos. Por otro lado, **Scipy.stats** y **Statsmodels** resultan especialmente

útiles para el análisis estadístico y el modelado, y **Seaborn** ofrece herramientas avanzadas para representar datos estadísticos de manera más intuitiva.

Especificamente **NumPy** y **Pandas**, son librerías que ofrecen mejoras significativas en términos de eficiencia, facilidad de manipulación y escalabilidad con los vectores que otros tipos de datos incorporados en Python como las listas.

Una de las grandes aportaciones Numpy es la la definición de un tipo de datos más eficiente que las listas para las grandes cantidades de datos. De forma general, un vector o array es una estructura de datos ordenada de elementos homogéneos cuyo orden viene definido por su posición (índice) no por su contenido. Los elementos se distribuyen en filas (1D), filas y columnas (2D), o incluso más dimensiones. Estas dimensiones son los índices que permiten acceder por posición a los elementos guardados en esta estructura de datos. Si bien esto ya se puede hacer con las listas Python se apoya en diversas librerías especializadas para el manejo de colecciones indexadas grandes y complejas. Especificamente **NumPy** y **Pandas**, son librerías que ofrecen mejoras significativas en términos de eficiencia, facilidad de manipulación y escalabilidad de los vectores que los tipos de datos incorporados en Python como las listas.

NumPy (Numerical Python)

En el ámbito de la programación para estadística y probabilidad, **NumPy** (Numerical Python) se presenta como una herramienta fundamental. Su eficiencia, versatilidad e integración con otras herramientas la convierten en un componente crucial para el desarrollo de aplicaciones científicas y de análisis de datos. A lo largo de este bloque, se aprende a utilizar las principales funcionalidades de NumPy para resolver problemas de estadística y probabilidad de forma efectiva.

Esta librería de Python proporciona una estructura de datos esencial: el **vector multidimensional** (o *ndarray*), junto con una amplia gama de funciones que facilitan la manipulación y el análisis de datos numéricos. Está implementada en C, lo que la hace más rápida en cálculos numéricos. **NumPy** se utiliza tanto que mucha gente ya lo considera parte integral del lenguaje. Proporciona un objeto de matriz multidimensional de alto rendimiento y herramientas para trabajar con matrices. Esta librería destaca por su eficiencia en el manejo de grandes conjuntos de datos y por su capacidad para realizar operaciones matemáticas de forma vectorial, lo que se traduce en un código más conciso y rápido.

Estas son solo algunas de las funciones disponibles en NumPy sobre los `ndarray` para el análisis estadístico y de probabilidad.

Función	Descripción
`np.mean()`	Calcula la media aritmética de los elementos de un array
`np.median()`	Calcula la mediana de los elementos de un array

Función	Descripción
`np.std()`	Calcula la desviación estándar de los elementos de un array
`np.var()`	Calcula la varianza de los elementos de un array
`np.max()`	Encuentra el valor máximo en un array
`np.min()`	Encuentra el valor mínimo en un array
`np.percentile()`	Calcula el percentil especificado de los elementos de un array
`np.histogram()`	Calcula un histograma de los elementos de un array
`np.random.normal()`	Genera muestras aleatorias de una distribución normal
`np.random.binomial()`	Genera muestras aleatorias de una distribución binomial
`np.random.poisson()`	Genera muestras aleatorias de una distribución de Poisson
`np.random.exponential()`	Genera muestras aleatorias de una distribución exponencial
`np.random.uniform()`	Genera muestras aleatorias de una distribución uniforme
`np.random.randint(n)`	Devuelve un array con números aleatorios desde 0 hasta n-1
`np.random.rand(n)`	Devuelve un array de `n` números aleatorios desde 0.0 hasta 1.0 (exlcluido)
`np.random.choice()`	Genera una muestra aleatoria de un array
`np.corrcoef()`	Calcula la matriz de coeficientes de correlación de Pearson
`np.cov()`	Calcula la matriz de covarianza de un array
`np.polyfit()`	Ajusta una función polinomial a un conjunto de datos utilizando el método de mínimos cuadrados
`np.polyval()`	Evalúa una función polinomial en un conjunto de valores
`np.gradient()`	Calcula el gradiente de un array multidimensional
`np.fft.fft()`	Calcula la transformada de Fourier discreta de un array
`np.fft.ifft()`	Calcula la transformada inversa de Fourier discreta de un array

Numpy da soporte para **álgebra lineal** ofreciendo un conjunto completo de herramientas para realizar operaciones de álgebra lineal, como la multiplicación de matrices, la resolución de sistemas de ecuaciones lineales, el cálculo de autovalores y autovectores, etc. También facilita la **manipulación de datos científicos**, incluyendo la lectura y escritura de archivos en diferentes formatos, la gestión de datos faltantes y la realización de transformaciones de datos.

Otra de las grandes fortalezas de esta librería es que es la **base para otras** , como **Pandas** que e utiliza NumPy para la representación y manipulación de datos en formato tablas de datos, **SciPy** que extiende NumPy con funciones adicionales para optimización, integración, ecuaciones diferenciales. **TensorFlow y Scikit-learn** que utilizan NumPy para la representación y procesamiento de datos en tareas de aprendizaje automático e inteligencia artificial, finalmente **Matplotlib:** se combina con la librería Numpy para declarar los datos de entrada y generar visualizaciones de datos.

En el núcleo de **NumPy** está el `ndarray`, donde *nd* es por n-dimensional. Un `ndarray` es una

estructura de datos multidimensional de elementos del mismo tipo que permite representar datos de forma natural como matrices y vectores, estructuras comunes en problemas estadísticos y probabilísticos. Son más rápidos y consumen menos memoria que las listas de Python pudiendose realizar operaciones matemáticas en vectores completos sin necesidad de bucles que los recorran, lo que mejora el rendimiento.

Un ejemplo básico muestra las operaciones sobre vectores. En concreto se genera un vector aleatorio de 2x3, imprime el array original y luego lo imprime para después de sumarlo a sí mismo y que empieza siempre por la importación de la librería.. Esto demuestra la manipulación básica y las operaciones numéricas dentro de NumPy, una habilidad fundamental para la programación y el análisis estadístico en Python.

```
import numpy as np

datos = np.random.randn(2, 3)
print(datos)
print("\n",datos + datos)
```

por ejemplo la salida podría ser:

```
[[-1.19852637  0.31035518  0.62839176]
 [-0.56442117  1.19460937 -0.47389261]]

 [[-2.39705274  0.62071036  1.25678352]
 [-1.12884234  2.38921875 -0.94778522]]
```

Hay muchas otras funciones para manipular un `ndarray` y realizar cálculos matemáticos complejos en Python. A continuación se muestran algunos ejemplos:

```
import numpy as np

# Crear un vector de NumPy
vector1 = np.array([1, 2, 3, 4, 5])
vector2 = np.array([1, 2, 3, 4, 5])

# Suma de dos vectores
vector = vector1 + vector2
print(vector)  # Resultado: [ 2  4  6  8 10]

# Sumar un número a cada elemento del vector
vector1 = vector1 + 7

# Matriz 2x3: [[1 2 3] [4 5 6]]
matriz1 = np.arange(1, 7).reshape(2, 3)

# Matriz 3x2: [[1 2] [3 4] [5 6]]
matriz2 = matriz1.reshape(3, 2)

# Convertir una lista anidada a una matriz NumPy
matriz3 = np.array([[1,2,3],[4,5,6]])

# Matriz identidad 4x4
imatriz = np.identity(4)

# Multiplicar cada elemento en array por un número
matriz4 = 4 * matriz2

# Elevar al cuadrado cada elemento del array
matriz4 = matriz2 ** 2
```

```
# Multiplicación de matrices mA y mB
mA = [[1,2], [5,3]]
mB = [[4,5], [2,3]]
matrizC = mA.dot(mB)

# Transpuesta de una matriz
mAT = mA.transpose()
```

Otras funciones aplicables son ayuda a conocer la información sobre la estructura de datos:

```
import numpy as np
x = np.array([1, 2, 3, 4, 5])

print(x.ndim) # dimensión: 1
print(x.size) # tamaño total del array: 5
print(x.shape) # forma: (5,)
print(x.dtype) # tipo de sus elementos: int64
```

Es posible crear un vector con valores de distintos tipos de datos al contrario que en otros lenguajes. Lo que realmente se produce (de forma implícita) se convierte un tipo de datos a otro sin tener en cuenta la comprobación de tipos, hay que recordar que esa escrita en C, que tiene tipificación dinámica.

¿`ndarray` o `list`? El uso de `ndarray` frente a una lista está justificado por cuestiones de rendimiento. En el siguiente ejemplo que suma 10 millones de valores enteros se clarifica el tiempo de cómputo de cálculo para ambas estructuras.

```
array_as_list = list(range(int(10e6)))
%timeit sum(array_as_list) # Captura el tiempo
```

En este caso el resultado es: 91 ms ± 19.7 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
array_as_ndarray = np.array(array_as_list)
%timeit array_as_ndarray.sum() # Captura el tiempo
```

Cuando se usa NumPy es mas rápido: 7.64 ms ± 436 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

En cualquier caso, existe la posibilidad de convertir a lista cualquier `ndarray` mediante el método `tolist()`.

Ejercicio resuelto Producción

Se quiere analizar el rendimiento de un cultivo en una finca utilizando la librería ‘ Numpy de Python. Se han recogido datos de 50 parcelas diferentes, donde se han registrado las toneladas de producción por parcela. Por simplicidad los datos no se leen de teclado se les da un valor fijo

```
3.2, 4.5, 5.1, 2.8, 3.9, 4.1, 2.5, 4.7, 4.2, 4.9, 3.6, 4.4, 3.8, 4.3, 3.7, 2.9, 4.0, 4.5, 3.6,
3.2, 2.7, 3.3, 3.9, 3.5, 3.0, 3.7, 4.0, 4.2, 4.6, 3.5, 4.1, 2.8, 4.4, 4.8, 4.6, 4.2, 4.1, 4.2,
4.5, 4.7, 4.4, 4.0, 4.3, 3.8, 3.5, 3.6, 4.5, 4.1, 3.2, 3.9
```

Muestre un resumen estadístico descriptivo.

```
import numpy as np
lista_datos=[3.2, 4.5, 5.1, 2.8, 3.9, 4.1, 2.5, 4.7, 4.2, 4.9, 3.6, 4.4, 3.8, 4.3, 3.7,
             2.9, 4.0, 4.5, 3.6, 3.2, 2.7, 3.3, 3.9, 3.5, 3.0, 3.7, 4.0, 4.2, 4.6, 3.5,
             4.1, 2.8, 4.4, 4.8, 4.6, 4.2, 4.1, 4.2, 4.5, 4.7, 4.4, 4.0, 4.3, 3.8, 3.5,
             3.6, 4.5, 4.1, 3.2, 3.9]
```

```
# Toneladas de producción por parcela
produccion = np.array(lista_datos) # transformamos la lista en objeto array

# Calcular la media de producción
media = np.mean(produccion)
print("Media:", media)

# Calcular la mediana de producción
mediana = np.median(produccion)
print("Mediana:", mediana)

# Calcular la desviación estándar de producción
desviacion_estandar = np.std(produccion)
print("Desviación estándar:", desviacion_estandar)

# Calcular los cuartiles
q1 = np.percentile(produccion, 25)
q3 = np.percentile(produccion, 75)
print("Cuartil (q1):", q1)
print("Cuartil (q3):", q3)

# Calcular la probabilidad de obtener una producción de más de 4 toneladas
probabilidad_mas_4 = np.sum(produccion > 4) / len(produccion)
print("Probabilidad de producción mayor a 4 toneladas:", probabilidad_mas_4)
```

Ejercicio resuelto. Una finca produce tres tipos de productos: manzanas, naranjas y plátanos. La producción de cada producto se mide en kilogramos (kg). Se tienen los siguientes datos:

- Producción diaria:
  - Día 1: 100 kg de manzanas, 50 kg de naranjas, 20 kg de plátanos
  - Día 2: 120 kg de manzanas, 60 kg de naranjas, 30 kg de plátanos
  - Día 3: 90 kg de manzanas, 40 kg de naranjas, 25 kg de plátanos
- Precio por kilogramo:
  - Manzanas: 2€/kg
  - Naranjas: 1.5€/kg
  - Plátanos: 0.8€/kg

Calcular el ingreso total de la granja para cada día y el ingreso total para los tres días.

```
import numpy as np

# 1. Representar los datos en matrices
produccion_diaria = np.array([[100, 50, 20], [120, 60, 30],
                              [90, 40, 25]])  # Producción por día (kg)
precios = np.array([2, 1.5, 0.8])  # Precio por kg (euros)

# 2. Calcular el ingreso por día
ingreso_por_dia = produccion_diaria @ precios  # Multiplicación matricial
# ingreso_por_dia= np.matmul(produccion_diaria, precios) # Otra forma

# 3. Calcular el ingreso total para los tres días
ingreso_total = np.sum(ingreso_por_dia)  # Suma de los ingresos por día

# 4. Imprimir los resultados
print("Ingreso por día:")
for i, ingreso in enumerate(ingreso_por_dia):
    print(f"Día {i+1}: {ingreso:.2f} euros")

print(f"\nIngreso total para los tres días: {ingreso_total:.2f} euros")
```

Ejercicio resuelto. Simular el lanzamiento de cien tiradas de un dado y mostrar la distribución de frecuencias de los resultados.

```
import numpy as np

dado100 =  [np.random.randint(1,7) for x in range(0,1000)]
# print(dado100)
```

Si ahora se desea calcular la tabla de frecuencias, se recurre a `np.unique`. Para nuestro ejemplo bastará con encontrar los valores únicos que aparecen y devolver el conteo de éstos para calcular su frecuencia.

```
np.unique(array, return_index=True,  return_inverse=True, return_counts=True)
```

devuelve para el array pasado como primer algumento los valores observados únicos, sus correspondientes índices, el valor inverso de éstos, y el conteo de los valores observados únicos. Si se omite el argumento, se toma el valor `False`.

```
# unir a código anterior
# ---------------
unique_valores, unique_conteo = np.unique(dado100, return_counts=True)

print("Valores Únicos:", unique_valores)
print("Frecuencia de Valores Únicos:", unique_conteo)
N=unique_conteo.sum()


# Imprimir los valores y sus frecuencias emparejados según posición
print("Valor |   fi  ")
print("==============")
for value, count in zip(unique_valores, unique_conteo):
    print(f"  {value}   |  {count/N}")
```

La salida de este código es:

```
Valores Únicos: [1 2 3 4 5 6]
Frecuencia de Valores Únicos: [155 168 174 160 189 154]
Valor |   fi
==============
  1   |  0.155
  2   |  0.168
  3   |  0.174
  4   |  0.16
  5   |  0.189
  6   |  0.154
```

Como se acaba de decir **NumPy** es una librería fundamental para el cálculo numérico en Python, eficiente para almacenar y manipular datos numéricos multidimensionales teniendo que ser todos los elementos del mismo tipo de datos. Éstos no tienen etiquetas para los datos (solo índices numéricos) y es aquí, en los análisis de datos donde se necesitan almacenar datos de diferentes tipos (números, texto, fechas, etc.) donde es útil tener etiquetas (nombres) para las columnas o variables, en lugar de solo índices numéricos. NumPy no ofrece estas características de forma directa y es lo que provoca el nacimiento de la siguiente librería **Pandas**.

Desafortunadamente, no han explorado en detalle las funcionalidades de NumPy. Su documentación oficial proporciona una descripción detallada de sus capacidades: https://numpy.org.

Pandas (Panel Data Analysis)

Pandas es una librería esencial para la manipulación y el análisis de datos estructurados en Python. Construida sobre NumPy, amplía sus capacidades al permitir trabajar con datos heterogéneos y etiquetados de forma eficiente. Su flexibilidad permite almacenar distintos tipos de datos en una misma columna, utilizar índices numéricos, etiquetas de texto o fechas, y realizar operaciones de análisis de manera sencilla.

Así como los `ndarray` son la base de NumPy, en Pandas existen dos estructuras de datos principales: `Series` y `DataFrame`. Una Series es una estructura unidimensional similar a un vector de NumPy, pero con etiquetas para cada dato, mientras que un `DataFrame` es una estructura bidimensional similar a una tabla, con filas y columnas etiquetadas, ideal para el análisis de datos tabulares.

```
# Crear una Series
s = pd.Series([1, 2, 3, 4])

# Crear un DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
```

El siguiente script compara `ndarray` y `Series`.

```
import numpy as np
import pandas as pd

# Datos en un array NumPy
datos_planta = np.array([
    ['Planta A', 15.2, '2023-10-26', True],
    ['Planta B', 12.8, '2023-10-27', False],
    ['Planta C', 17.5, '2023-10-28', True]
])

# Datos en un objeto Pandas Series
serie_planta = pd.Series({
    'Nombre': ['Planta A', 'Planta B', 'Planta C'],
    'Altura (cm)': [15.2, 12.8, 17.5],
    'Fecha': pd.to_datetime(['2023-10-26', '2023-10-27', '2023-10-28']),
    'Germinó': [True, False, True]
})

print("Vector NumPy:\n", datos_planta)
print("\nPandas Series:\n", serie_planta)
```

La salida de este código es:

```
Array NumPy:
 [['Planta A' '15.2' '2023-10-26' 'True']
 ['Planta B' '12.8' '2023-10-27' 'False']
 ['Planta C' '17.5' '2023-10-28' 'True']]

Pandas Series:
 Nombre                           [Planta A, Planta B, Planta C]
Altura (cm)                                    [15.2, 12.8, 17.5]
Fecha          DatetimeIndex(['2023-10-26', '2023-10-27', '20...
Germinó                                        [True, False, True]
dtype: object
```

Como se observa el vector de NumPy `datos_planta` almacena todos los datos como cadenas de texto, perdiendo la información de tipos. Una serie `serie_planta` permite almacenar datos de diferentes tipos (texto, números, fechas, booleanos) y proporciona etiquetas para cada variable.

Pandas esta optimizada para la lectura/escritura en archivos de multiples formatos: CSV, Excel, JSON, SQL, HDF5, entre otros con funciones como `pd.read_csv()` y `df.to_csv()`. Se complementa con NumPy, Matplotlib, Seaborn y Scikit-learn para visualización y análisis de datos. Esta especialmente pensada para filtrar, ordenar, agrupar, fusionar y transformar grandes volúmenes de datos de manera sencilla, con operaciones como `df.groupby()`, `df.merge()`, `df.pivot_table()` y métodos como `df.dropna()` y `df.fillna()` facilitan el manejo de valores nulos.

Tipo Series

Una **Serie** en Pandas suele ser homogénea, es decir, sus elementos suelen ser del mismo tipo, aunque Pandas puede manejar tipos mixtos si es necesario. Su tamaño es flexible, en el sentido de que los datos pueden modificarse y se pueden generar nuevas series con más o menos elementos.

La forma más fácil de pensar en los objetos de la serie Pandas es como un contenedor para dos matrices Numpy, una para el índice o etiquetas y otra para los datos. Las matrices Numpy ya tienen indexación de enteros al igual que las listas regulares de Python.

Para crear una serie se utiliza `pd.Series` y puede tener diversos argumentos como los representan los datos, los indices, o el nombre. Los valores por defecto para los indices se usan los enteros del 0 al n-1, donde n es el tamaño de la serie

- `data`: Los datos que se van a almacenar en la serie. Pueden ser listas, arrays de NumPy, diccionarios, etc.
- `index`: Las etiquetas para el índice. Si no se especifica, Pandas crea un índice numérico por defecto (0, 1, 2, ...).
- `name`: Un nombre para la serie (opcional, pero útil para la organización).
- `dtype`: El tipo de dato de los elementos de la serie.

```
import pandas as pd
# Método constructor de series a partir de una lista, el indice de 0 a 5
x = pd.Series(data=[1, 2, 30, 0, 15, 6],
              dtype='int64',  # Especificando el tipo de dato
              name='Números')   # Añadiendo un nombre a la serie

# Crea una serie de nombre asignaturas con indice 0 a 4
s = pd.Series(data=['Matemáticas', 'Estadística', 'Economía', 'Programación', 'Inglés'], \
     index=['Mat', 'Est', 'Eco', 'Prog', 'Ing'],  # Índice personalizado
        name='Asignaturas', dtype='string')

# Serie llamada muestra con indice de a a f
y = pd.Series(data=[1, 2, 30, 0, 15, 6],
        index=['a', 'b', 'c', 'd', 'e', 'f'],
        name='Muestra',
        dtype='float64')  # Cambiando el tipo de dato a float

# Misma serie creada a partir de un diccionario
z = pd.Series(data={'a': 1, 'b': 2, 'c': 30, 'd': 0, 'e': 15, 'f': 6},
        name='Muestra desde Diccionario')

# Imprimir las series para ver los resultados
print("Serie x:\n", x, "\n")
print("Serie s:\n", s, "\n")
print("Serie y:\n", y, "\n")
print("Serie z:\n", z, "\n")
```

El acceso es similar a las colecciones de Python pero se puede acceder usando tanto la etiqueta como el indice. En ambos casos es puede hacer indexación (para un elemento) o segmentación (una lista de posiciones).

Acceso por posición

Utilizando la ubicación numérica de los elementos dentro de la serie, similar a cualquier indexación sobre estructuras de datos Python

- `y[i]`: Devuelve el elemento que ocupa la posición i+1 en la serie s.
- `y[posiciones]`: Devuelve otra serie con los elementos que ocupan las posiciones de la lista posiciones.

```
# Acceder al primer elemento (posición 0)
primer_elemento = y[0]
print(primer_elemento)  # Salida: 1.0

# Acceder al tercer elemento (posición 2)
tercer_elemento = y[2]
print(tercer_elemento)  # Salida: 30.0

# Acceder a un rango de elementos (posiciones 1 a 3)
subserie = y[1:4]
print(subserie)
# Salida:
# b     2.0
# c    30.0
# d     0.0
# Name: Muestra, dtype: float64
```

Acceso por índice (etiqueta)

El acceso por índice se realiza utilizando las etiquetas que se han asignado al índice de la serie. Esto permite acceder a los elementos por su nombre o etiqueta en lugar de su posición numérica.

- `y[nombre]`: Devuelve el elemento con el nombre nombre en el índice.
- `y[nombres]`: Devuelve otra serie con los elementos correspondientes a los nombres indicadas en la lista nombres en el índice.

```
# Acceder al elemento con la etiqueta 'b'
elemento_b = y['b']
print(elemento_b)  # Salida: 2.0

# Acceder a los elementos con las etiquetas 'a' y 'f'
subserie_af = y[['a', 'f']]
print(subserie_af)
# Salida:
# a     1.0
# f     6.0
# Name: Muestra, dtype: float64

# Acceder a un rango de elementos usando etiquetas
subserie_bcde = y['b':'e']
print(subserie_bcde)
# Salida:
# b     2.0
# c    30.0
# d     0.0
# e    15.0
# Name: Muestra, dtype: float64
```

Accediendo con `iloc` y segmentación

La indexación basada en posición (`iloc`) para acceder a una subserie de `x`, aplicando slicing (rebanado) con la sintaxis `x.iloc[inicio:fin:paso]`

- `inicio`: Selecciona la primera posición que se quiere (incluida).
- `fin`: Selecciona la última posición que se quiere (excluida).
- `paso`: El número de saltos entre cada selección.

```
import pandas as pd

# Muestra de Serie (elementos)
x = pd.Series([10, 20, 30, 40, 50, 60, 70])

# iloc con slicing para selección específicas
# ¡Desde la 1 hasta la 4 (sin incluirla), saltando de 2 en 2!
resultado = x.iloc[1:5:2]

print(resultado)
```

Genera como resultado:

```
1    20
3    40
dtype: int64
```

En caso de series con longitud considerable, se tienen los métodos `head` y `tail` para manejar las posiciones iniciales o finales de la serie. Permiten acceder a un trozo de los datos para visualizar como son.

```
x.head(3) # tres primeros elementos

x.tail(3) # tres últimos elementos
```

Existen varias propiedades o métodos para ver las características de una serie.

- `s.size`: Devuelve el número de elementos de la serie `s`.
- `s.index`: Devuelve una lista con los nombres de las filas de `s`.
- `s.dtype`: Devuelve el tipo de datos de los elementos de `s`.

```
x.values #valores
x.values[1:3]
x.index #índice de la serie
x.dtype #tipo de la serie
x.name #nombre de la serie
x.size #número de registros de la serie
```

Para **ordenar** una serie se utilizan los siguientes métodos:

- `serie.sort_values(ascending=booleano)`: Devuelve la serie que resulta de ordenar los valores la serie `s`. Si argumento del parámetro `ascending` es `True` el orden es creciente y si es `False` decreciente.
- `serie.sort_index(ascending=booleano)`: Devuelve la serie que resulta de ordenar el índice de la serie `s`. Si el argumento del parámetro ascending es `True` el orden es creciente y si es `False` decreciente.

```
x = pd.Series([1, 3, 7, 2, 4], index=['a', 'b', 'c', 'd', 'e'])
# Ordenación por valores
x.sort_values(False)
```

O bien

```
# Ordenación por índice
x.sort_index()
```

Ambos métodos admiten el parámetro `ascending` para indicar si la ordenación es ascendente (`True`) o descendente (`False`); y también admiten el parámetro `inplace` para indicar si se quiere modificar los valores de la serie (`True`) o devolver una nueva ya ordenada (`False`).

Para **filtrar** una serie y quedarse con los valores que cumplen una determinada condición se utiliza :

- `s[condicion]`: Devuelve una serie con los elementos de la serie s que se corresponden con el valor `True` de la lista booleana `condicion`. `condicion` debe ser una lista de valores booleanos de la misma longitud que la serie.

```
#Serie booleana con filtrado de sus elementos: x>3
x > 3

# Aplicación del filtrado de elementos con indexado:
x[x>3]
```

También es posible aplicar una función a cada elemento de la serie mediante el siguiente método:

- `s.apply(f)`: Devuelve una serie con el resultado de aplicar la función f a cada uno de los elementos de la serie s.

```
import pandas as pd
from math import log

# Aplicar la función logaritmo
s = pd.Series([1, 2, 3, 4])
print(s.apply(log))

# Aplicar la conversión a mayúsculas
s = pd.Series(['a', 'b', 'c'])
s.apply(str.upper)
```

Las siguientes funciones permiten resumir varios aspectos de una serie:

Función	Descripción
`s.count()`	Devuelve el número de elementos que no son nulos ni NaN en la serie `s`
`s.sum()`	Devuelve la suma de los datos de la serie s cuando los datos son de un tipo numérico, o la concatenación de ellos cuando son del tipo cadena `str`
`s.cumsum()`	Devuelve una serie con la suma acumulada de los datos de la serie s cuando los datos son de un tipo numérico
`s.value_counts()`	Devuelve una serie con la frecuencia de cada valor de la serie `s`
`s.min()`	Devuelve el menor de los datos de la serie
`s.max()`	Devuelve el mayor de los datos de la serie
`s.argmin()`	Devuelve la posición/índice del valor mínimo

Función	Descripción
`s.argmax()`	Devuelve la posición/índice del valor máximo de una serie
`s.idxmin()`	Devuelve la etiqueta/índice del valor mínimo de una serie
`s.idxmax()`	Devuelve la etiqueta/índice del valor máximo de una serie
`s.nsmallest(n)`	Devuelve los `n` valores menores de una serie
`s.nlargest(n)`	Devuelve los `n` valores mayores de una serie
`s.mean()`	Devuelve la media de los datos de la serie cuando los datos son de un tipo numérico
`s.var()`	Devuelve la cuasi-varianza de los datos de la serie cuando los datos son de un tipo numérico
`s.std()`	Devuelve la cuasi-desviación típica de los datos de la serie cuando los datos son de un tipo numérico
`s.describe()`	Devuelve una serie con un resumen que incluye el número de datos, su suma, el mínimo, el máximo, la media, la desviación típica y los cuartiles

Ejercicio resuelto. Análisis de producción

Se resolverá el problema anterior utilizando `Series` para analizar el rendimiento de un cultivo en una finca. Se han recogido datos de 50 parcelas diferentes, donde se han registrado las toneladas de producción por parcela.

```
3.2, 4.5, 5.1, 2.8, 3.9, 4.1, 2.5, 4.7, 4.2, 4.9,
3.6, 4.4, 3.8, 4.3, 3.7, 2.9, 4.0, 4.5, 3.6, 3.2,
2.7, 3.3, 3.9, 3.5, 3.0, 3.7, 4.0, 4.2, 4.6, 3.5,
4.1, 2.8, 4.4, 4.8, 4.6, 4.2, 4.1, 4.2, 4.5, 4.7,
4.4, 4.0, 4.3, 3.8, 3.5, 3.6, 4.5, 4.1, 3.2, 3.9
```

Se mostrará un resumen estadístico descriptivo y se calcularán los elementos con una producción de más de 4 toneladas.

```
import pandas as pd

produccion = [3.2, 4.5, 5.1, 2.8, 3.9, 4.1, 2.5, 4.7, 4.2, 4.9, 3.6, 4.4, 3.8,
        4.3, 3.7, 2.9, 4.0, 4.5, 3.6, 3.2, 2.7, 3.3, 3.9, 3.5, 3.0, 3.7, 4.0,
        4.2, 4.6, 3.5, 4.1, 2.8, 4.4, 4.8, 4.6, 4.2, 4.1, 4.2, 4.5, 4.7, 4.4,
        4.0, 4.3, 3.8, 3.5, 3.6, 4.5, 4.1, 3.2, 3.9]

serie = pd.Series(produccion, name="Producción")

# Muestra un resumen de los estadísticos predefinidos en Python
serie.describe()

# Calcular los elementos con una producción de más de 4 toneladas
probabilidad_mas_4 = sum(serie > 4) / len(produccion)
print("\nProbabilidad de producción mayor a 4 toneladas:", probabilidad_mas_4)

# Calcular la media de la altura
media=serie.mean()
print("Producción media", media)
```

Ejercicio resuelto. Análisis temperaturas

Se ha registrado en una serie de datos las temperaturas máximas diarias durante un mes, se quiere

seleccionar solo los días en los que la temperatura máxima superó los 30 grados Celsius. De igual forma, obtener la tabla de distribución de frecuencias como composición de Series para formar las columnas de la tabla.

Las temperaturas registradas son: `28, 30, 33, 29, 31, 35, 26, 27, 30, 32, 34, 28, 29, 30, 31, 32`

```
import pandas as pd

temperaturas = pd.Series([28, 30, 33, 29, 31, 35, 26, 27, 30, 32, 34, 28, 29, 30, 31, 32])

# Seleccionamos los días en los que la temperatura máxima superó los 30 grados Celsius
dias_calurosos = temperaturas[temperaturas > 30]

print(dias_calurosos)

# Obtención de las frecuencias absolutas: ni
""" Se realiza el conteo de valores y se clasifican por temperaturas ordenadas
    por el (índice) valor de las temperaturas."""
ni=temperaturas.value_counts().sort_index()

# Obtención frecuencias absolutas acumuladas: Ni
Ni=ni.cumsum()
```

Tipo DataFrame

Un **DataFrame** es una estructura de datos bidimensional etiquetada con filas y columnas (columnas de tipos potencialmente diferentes como enteros, cadenas, float, None, objetos Python, etc.). Su comportamiento y operaciones son parecidas a las series pero en dos dimensiones.

Se pueden crear `DataFrame` a partir de series, ndarrays, listas y diccionarios. Se puede relacionar a una hoja de cálculo o una tabla SQL, o un diccionario de objetos Series.

Es el objeto Pandas más utilizado, cada columna es de tipo `Series`, por lo que los datos de cada columna son del mismo tipo. Se puede considerar como una concatenación de series, donde cada una tiene a su vez un índice (`columns`). En los DataFrame se puede especificar tanto el `index` (el nombre de las filas) como `columns` (el nombre de las columnas)

`DataFrame(data=diccionario, index=indices)`: Devuelve un objeto de tipo `Series` con los valores del diccionario `diccionario` y las filas especificados en la lista `indices`. Si no se pasa la lista de índices se utilizan como índices las claves del diccionario.

```
import pandas as pd

# Crear un DataFrame
df = pd.DataFrame({'agricultor':['Bernardo','Sonia','Tomás'], 'edad':[20,30,40],
                   'cultivo_principal':['tomate', 'calabacín', 'sandía']},
                   index = ['agri1', 'agri2', 'agri3']
                  )

# lista de columnas
df.columns

# lista de índices
df.index

# Son expresiones equivalentes
print(df.cultivo_principal, end="\n\n")
print(df['cultivo_principal'])
```

Y si se quiere saber el tipo de dato de una columna en concreto:

```
print(df.edad.dtype)
```

Se puede asignar a una columna todos los valores iguales, por ejemplo, todas las personas con la edad a 10.

```
df.edad = 10
```

También crear edades aleatorias para éstas personas:

```
import numpy as np
df.edad = np.random.randint(15,40,3)
```

Acceso a los elementos mediante las posiciones:

- `df.iloc[i, j]` : Devuelve el elemento de la fila `i` y la columna `j` de `df`.
- `df.iloc[i]` : Devuelve una serie con los elementos de la fila `i` del DataFrame `df`.

Acceso a los elementos mediante nombres:

- `df.loc[fila, columna]` : Devuelve el elemento que se encuentra en la fila con nombre `fila` y la columna de con nombre `columna` del DataFrame `df`.
- `df[columna]` : Devuelve una serie con los elementos de la columna `columna` de `df`.

En el ejemplo anterior, se tendría:

```
# Acceso por posición:
print(df.iloc[0, 2])            # Salida: tomate
print(df.iloc[1])               # Salida: una Serie con la información de Sonia
print("-"*30)
# Acceso por nombres:
print(df.loc['agri2', 'edad'])   # Salida: 30
print(df['cultivo_principal'])   # Salida: Serie con los cultivos principales
```

Existen varias propiedades o métodos para ver las características de un DataFrame.

Función	Descripción
`df.info()`	Devuelve información (número de filas, número de columnas, índices, tipo de las columnas y memoria usado) sobre `df`.
`df.shape`	Devuelve una tupla con el número de filas y columnas de `df`.
`df.size`	Devuelve el número de elementos del DataFrame.
`df.columns`	Devuelve una lista con los nombres de las columnas de `df`.
`df.index`	Devuelve una lista con los nombres de las filas de `df`.
`df.dtypes`	Devuelve una serie con los tipos de datos de las columnas de `df`.
`df.head(n)`	Devuelve las n primeras filas de `df`.
`df.tail(n)`	Devuelve las n últimas filas de `df`.

Al igual que para las series, hay métodos que permiten resumir la información de un DataFrame:

Función	Descripción
`df.count()`	Devuelve una serie con el número de elementos que no son nulos ni `NaN` en cada columna de `df`.
`df.sum()`	Devuelve una serie con la suma de los datos de las columnas de `df` cuando los datos son de un tipo numérico, o la concatenación de ellos cuando son del tipo cadena `str`.
`df.cumsum()`	Devuelve un DataFrame con la suma acumulada de los datos de las columnas de `df` cuando los datos son de un tipo numérico.
`df.min()`	Devuelve una serie con los menores de los datos de las columnas de `df`.
`df.max()`	Devuelve una serie con los mayores de los datos de las columnas de `df`.
`df.mean()`	Devuelve una serie con las medias de los datos de las columnas numéricas de `df`.
`df.var()`	Devuelve una serie con las varianzas de los datos de las columnas numéricas de `df`
`df.std()`	Devuelve una serie con las desviaciones típicas de los datos de las columnas numéricas de `df`
`df.cov()`	Devuelve un DataFrame con las covarianzas de los datos de las columnas numéricas de `df`.
`df.corr()`	Devuelve un DataFrame con los coeficientes de correlación de Pearson de los datos de las columnas numéricas de `df`.
`df.describe (include = tipo)`	Devuelve un DataFrame con un resumen estadístico de las columnas de `df` del tipo `tipo`. Para los datos numéricos se calcula la media, la desviación típica, el mínimo, el máximo y los cuartiles. Para los datos no numéricos (`object`) se calcula el número de valores, el número de valores distintos, la moda y su frecuencia. Sin tipo solo se consideran las columnas numéricas

Ejercicio resuelto. Calificaciones

Se han recopilado las calificaciones de 20 estudiantes en dos asignaturas: "Estadística" y "Probabilidad". Las calificaciones se otorgan en una escala de 0 a 10. Realiza un análisis estadístico básico usando la librería Pandas de Python. El programa debe realizar las tareas:

- Crear un DataFrame: Crea un DataFrame de Pandas para almacenar las calificaciones de los estudiantes. El DataFrame debe tener las siguientes columnas:
  - "Estudiante": Contiene el nombre o identificador de cada estudiante (por ejemplo, "Estudiante 1", "Estudiante 2", etc.).
  - "Estadística": Contiene las calificaciones de cada estudiante en Estadística.
  - "Probabilidad": Contiene las calificaciones de cada estudiante en Probabilidad.
- Calcular estadísticas descriptivas: Para cada asignatura se calcula:
  - Calificación promedio.

    - Desviación estándar.
    - Calificación más alta.
    - Calificación más baja.
- Mostrar resultados: Imprime los resultados de los cálculos de forma clara y organizada.
- Utilizar describe(): Utiliza esta función de Pandas para obtener un resumen estadístico de las calificaciones en ambas asignaturas. Asegúrate de que los resultados se muestren con tres decimales de precisión.
- Análisis de correlación: Calcula e imprime el coeficiente de correlación entre las calificaciones de Estadística y Probabilidad. Interpreta el resultado brevemente.

```
import pandas as pd
import numpy as np

# 1. Crear el DataFrame (calificaciones entre 0 y 10)
datos = {'Estudiante': [f'Estudiante {i}' for i in range(1, 21)],
        'Estadística': [8.5, 7.8, 9.2, 8.8, 7.5, 9.0, 8.2, 7.9, 8.7, 9.1,
                        7.0, 8.3, 8.9, 7.6, 8.1, 9.3, 7.7, 8.4, 9.5, 6.8],
        'Probabilidad': [9.2, 8.8, 9.5, 9.0, 8.5, 9.3, 8.9, 8.6, 9.1, 9.4,
                         8.0, 8.7, 9.2, 8.3, 8.8, 9.6, 8.4, 9.0, 9.7, 7.5]}
df = pd.DataFrame(datos)

# 2. Calcular la calificación promedio
promedio_estadistica = df['Estadística'].mean()
promedio_probabilidad = df['Probabilidad'].mean()

# 3. Calcular la desviación estándar
desviacion_estadistica = df['Estadística'].std()
desviacion_probabilidad = df['Probabilidad'].std()

# 4. Encontrar las calificaciones más altas y más bajas
maxima_estadistica = df['Estadística'].max()
minima_estadistica = df['Estadística'].min()
maxima_probabilidad = df['Probabilidad'].max()
minima_probabilidad = df['Probabilidad'].min()

# 5. Mostrar los resultados
print("Análisis estadístico de calificaciones de exámenes (0-10):")
print("-"*58)
print(f"Calificación promedio en Estadística: {promedio_estadistica:.2f}")
print(f"Calificación promedio en Probabilidad: {promedio_probabilidad:.2f}")
print(f"Desviación estándar (Estadística): {desviacion_estadistica:.2f}")
print(f"Desviación estándar (Probabilidad): {desviacion_probabilidad:.2f}")
print(f"Calificación (Estadística) más baja {minima_estadistica} y más alta \
    {maxima_estadistica}")
print(f"Calificación (Probabilidad) más baja {minima_probabilidad} y más alta \
    {maxima_probabilidad}")

# 6. Usar describe() para obtener estadísticas descriptivas
print("\nEstadísticas Descriptivas:")
print(df.describe().map(lambda x: f'{x:.3f}')) # enlugar de apply.map

# 7. Calcular la correlación entre Estadística y Probabilidad
correlacion = df['Estadística'].corr(df['Probabilidad'])
print(f"\nCorrelación entre Estadística y Probabilidad: {correlacion:.4f}")
```

Matplotlib: visualización de datos

El paquete `matplotlib` es una extensa librería de funciones para generar gráficos 2D y 3D. Su uso es fundamental en el análisis de datos, ya que permite visualizar la información de manera clara y efectiva. estaca por su curva de aprendizaje suave, ya que ofrece funcionalidades simples

para usuarios ocasionales. Además, proporciona flexibilidad y personalización, permitiendo ajustar programáticamente todos los elementos de una ventana gráfica. Por último, cuenta con amplia compatibilidad, ya que admite una gran variedad de formatos de exportación para las figuras generadas.

Matplotlib está construido sobre el paquete **NumPy** y trabaja de forma natural con los vectores y matrices (arrays) asociados a él. Dado que en los ejemplos anteriores se han presentado datos numéricos sin su correspondiente representación gráfica, es conveniente introducir ahora esta herramienta para completar dichos ejemplos y mejorar la comprensión de los resultados.

Por el momento, se trabajará con **listas nativas** de Python, las cuales serán internamente transformadas a arrays de NumPy en las funciones de Matplotlib. Este proceso es transparente para el usuario, pero se debe tener en cuenta que las listas utilizadas deben estar formadas por valores del mismo tipo de datos.

Dentro del paquete `matplotlib` destaca el módulo `pyplot`. Este módulo oculta muchas de las funcionalidades de bajo nivel de la biblioteca, permitiendo el uso de sencillas funciones para los elementos gráficos más habituales, tales como creación de figuras, trazado de líneas, visualización de imágenes, inserción de texto, etc. El módulo `pyplot` emula el entorno de programación gráfica de MATLAB, herramienta software de pago muy popular en universidades y empresas.

Dado que en los ejemplos resueltos anteriores no se han incluido representaciones gráficas de los datos, a continuación se introducirá el uso de la función `.plot()`, que permite visualizar de manera efectiva las listas de valores proporcionadas. Esta incorporación facilitará la interpretación de los resultados y reforzará el vínculo entre los conceptos teóricos y su aplicación práctica.

Gráficos de linea con la función `.plot()`

Los gráficos de líneas son útiles para visualizar tendencias o patrones en datos secuenciales, como series temporales. En estadística, se utilizan para representar la evolución de una variable a lo largo del tiempo.

```
import matplotlib.pyplot as plt

# Datos: temperaturas diarias en una semana (en grados Celsius)
temperaturas = [14, 16, 12, 18, 20, 22, 19]
dias = ["Lun", "Mar", "Mié", "Jue", "Vie", "Sáb", "Dom"]

# Crear el gráfico de líneas
plt.plot(dias, temperaturas, marker='o', linestyle='-', color='b', label="Temperatura")

plt.xlabel("Día de la semana")  # Etiqueta del eje X
plt.ylabel("Temperatura (°C)")  # Etiqueta del eje Y
plt.title("Evolución de la temperatura en una semana")  # Título del gráfico
plt.legend()  # Mostrar la leyenda
plt.grid(True, linestyle=':', alpha=0.3)  # Añadir una rejilla punteada y transparente

# Mostrar el gráfico
plt.show()
```

Se ha de invocar la función `.show()`, que es la que de forma efectiva muestra la figura 15.

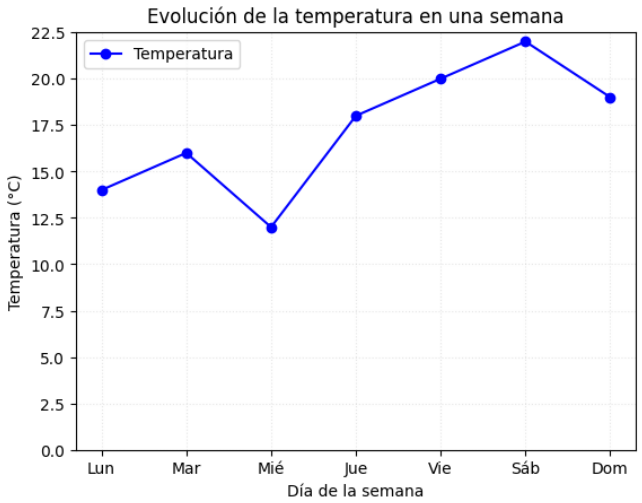

Figura 15: Gráfico de línea

En el caso de ser los datos numéricos se utilizan vectores para declarar los datos (figura 16):

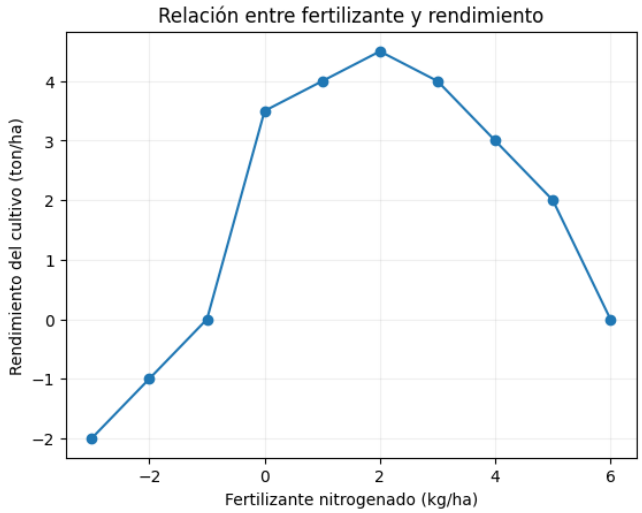

Figura 16: Gráfico sobre dos ejes

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

fertilizante = np.array([-3, -2, -1, 0, 1, 2, 3, 4, 5, 6])   # kg/ha
rendimiento = np.array([-2, -1, 0, 3.5, 4, 4.5, 4, 3, 2, 0])   # ton/ha

# Crear el gráfico
plt.plot(fertilizante, rendimiento, marker='o', linestyle='-')

# Personalizar el gráfico
plt.xlabel("Fertilizante nitrogenado (kg/ha)")
plt.ylabel("Rendimiento del cultivo (ton/ha)")
plt.title("Relación entre fertilizante y rendimiento")
plt.grid(True, alpha=0.2)

plt.show()     # Mostrar el gráfico
```

Representación de funciones matemáticas

También se pueden representar funciones matemáticas e incluso dos funciones en una misma gráfica: parábola y su recta tangente evaluada en un punto (figura 17). En el siguiente código se han añadido los ejes en la representación. Esto se hace con `axhline` y `axvline` teniendo en cuenta los distintos argumentos opcionales de palabra clave de la forma `kwarg=valor`:

- `axhline(y=valor, color='color', zorder=valor)` Línea horizontal en `y=valor`.
  - `color`: Especifica el color de la línea (por defecto es negro).
  - `zorder`: Controla la superposición con otros elementos del gráfico.
    - `zorder=0`: Nivel predeterminado.
    - `zorder=-1`: La línea queda detrás de otros elementos
- `avxline(x=valor, color='color', zorder=valor)`: Dibuja una línea vertical en `x=valor`. Tiene los mismos argumentos que `axhline`.

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

# 1. Definir la función de la parábola y su derivada
def parabola(x):
    return x**2 - 3

def derivada_parabola(x):
    return 2*x  # Derivada de x² - 3

# 2. Punto de tangencia y cálculo de la recta tangente
x0 = 1                              # Punto en el eje X donde queremos la tangente
y0 = parabola(x0)                   # Valor de Y en el punto de tangencia
pendiente = derivada_parabola(x0)   # Pendiente de la tangente

# Ecuación de la recta tangente: y = m(x - x1) + y1
def recta_tangente(x):
    return pendiente * (x - x0) + y0

# 3. Generar valores de X
x = np.linspace(-10, 10, 100)

# 4. Calcular valores de Y para la parábola y la recta tangente
y_parabola = parabola(x)
y_recta = recta_tangente(x)

# 5. Crear el gráfico
plt.plot(x, y_parabola, label='Parábola: y = x² - 3', color='red')
plt.plot(x, y_recta, label=f'Tangente en x = {x0}', color='blue')
plt.plot(x0, y0, marker='o', markersize=8, color='green',label=f'Punto de tangencia ({x0}, {y0})')

# 6. Ajustar la escala de los ejes para que sean proporcionales (opcional)
plt.ylim(-10,20)
plt.xlim(-15,15)
plt.axhline(color = 'gray', zorder=-1)
plt.axvline(color = 'gray', zorder=-1)


# 7. Personalizar el gráfico
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Parábola y su recta tangente")
plt.legend()
plt.grid(True, alpha=0.2)

# 8. Mostrar el gráfico
plt.show()
```

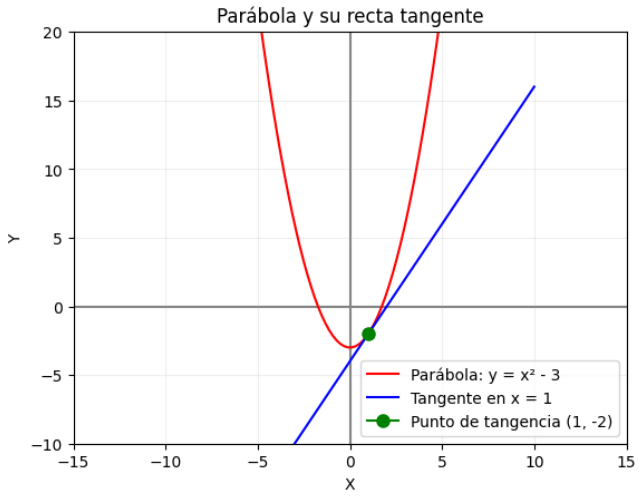

Figura 17: Gráfico sobre dos ejes

Gráfico de puntos o diagrama de dispersión

Si lo que se quiere es un gráfico de puntos donde se fijan etiquetas para los ejes y un título para el gráfico (figura 18).

```
temperatura =  [ 14.2, 16.4, 11.9, 15.2, 18.5, 22.1, 19.4, 25.1,
23.4, 18.1, 22.6, 17.2]
ventas = [ 215, 325, 185, 332, 406, 522, 412, 614, 544, 421, 445, 408]

plt.scatter(temperatura, ventas)
plt.xlabel('Temperatura(oC)')
plt.ylabel('Ventas')
plt.title('Temperatura vs Ventas de helados')
plt.show()
```

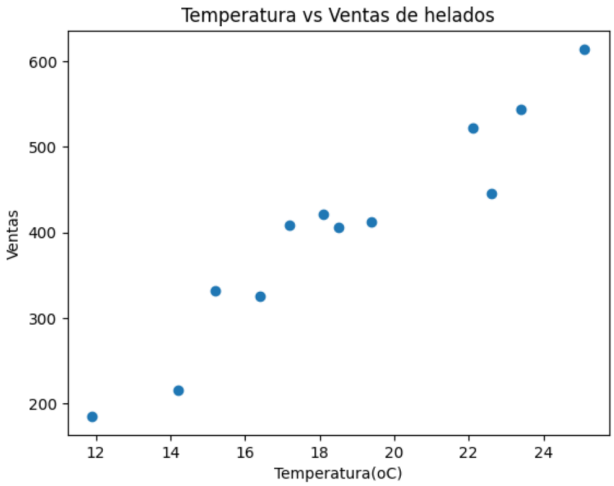

Figura 18: Gráfico de puntos

Gráfico de barras

Para generar un gráfico de barras se usará `bar()` y para personalizarlo se detallan algunas cuestiones a continuación tal como se muestra en la figura 19.

- `plt.xlabel(), plt.ylabel(), plt.title()`: Estas funciones agregan etiquetas al eje x, al eje y y al título del gráfico, respectivamente.
- `plt.xticks(x, etiquetas)`: Reemplaza las marcas predeterminadas del eje x (números) con las etiquetas de categoría de la lista etiquetas.
- Con el bucle `for` se agrega el valor numérico de cada barra encima de la barra para mayor claridad.
- `plt.grid(axis='y', linestyle='--', alpha=0.5)`: Agrega una cuadrícula horizontal al gráfico para facilitar la lectura de los valores, con un estilo de línea discontinua y transparencia media.

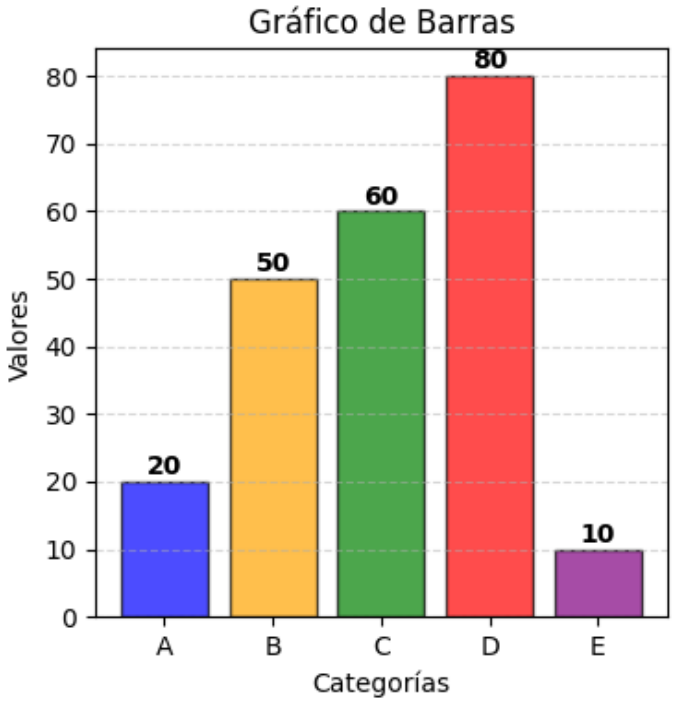

Figura 19: Gráfico de barras

```
import matplotlib.pyplot as plt

datos = [20, 50, 60, 80, 10]           # Valores de cada barra
x = [1, 2, 3, 4, 5]                    # Posición de las barras
etiquetas = ['A', 'B', 'C', 'D', 'E']  # Nombres para cada categoría

# Crear la figura y ajustar el tamaño
plt.figure(figsize=(4, 4))

# Crear el gráfico de barras con colores personalizados
plt.bar(x, datos, color=['blue','orange','green','red','purple'], alpha=0.7, edgecolor='black')
```

```
# Añadir etiquetas a los ejes y título
plt.xlabel("Categorías")
plt.ylabel("Valores")
plt.title("Gráfico de Barras")

plt.xticks(x, etiquetas)          # Colocar etiquetas en el eje X

# Añadir valores en cada barra
for i, valor in enumerate(datos):
    plt.text(x[i], valor + 1, str(valor), ha='center', fontsize=10, fontweight='bold')

# Mostrar cuadrícula horizontal para facilitar lectura
plt.grid(axis='y', linestyle='--', alpha=0.5)

# Mostrar el gráfico
plt.show()
```

Gráfico de sectores

Para el gráfico de sectores se utiliza `pie()` y se detallan los siguientes argumentos (ver figura 20):

- `autopct='%1.1f%%'`: Se encarga de mostrar los porcentajes en cada sector del círculo. Para formatear los porcentajes:
  - %: Indica el inicio de un especificador de formato.
  - 1.1f: Define el formato del número:
    - 1: Ancho mínimo de un carácter.
    - .1: Precisión de un decimal.
    - f: Tipo de dato: número punto flotante.
  - %%: Escapa el símbolo de porcentaje (%) para que se muestre en el gráfico
- `startangle=90`: Este argumento controla la rotación del círculo. Indica que se rotará 90 grados en sentido antihorario. Esto significa que el primer sector (primer valor en datos) comenzará en la posición de las 12 en punto (en la parte superior del gráfico).

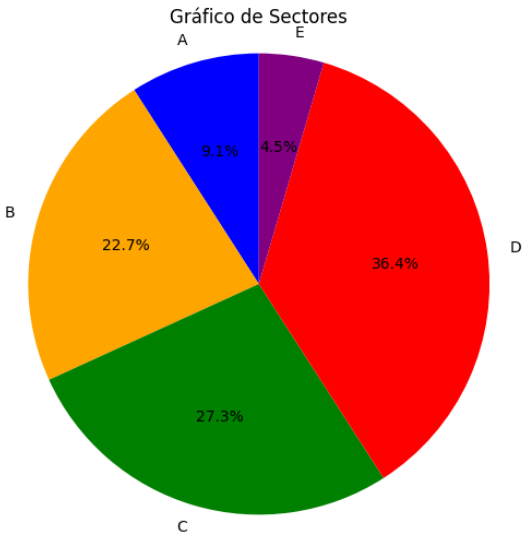

Figura 20: Gráfico de sectores

```
import matplotlib.pyplot as plt
# Datos para el gráfico de sectores
datos = [20, 50, 60, 80, 10]  # Valores de cada sector
etiquetas = ['A', 'B', 'C', 'D', 'E']  # Etiquetas para cada sector
colores = ['blue', 'orange', 'green', 'red', 'purple']

# Crear la figura y ajustar el tamaño (opcional)
plt.figure(figsize=(6, 6))

# Crear el gráfico de sectores
plt.pie(datos, labels=etiquetas, colors=colores, autopct='%1.1f%%', startangle=90)

# Ajustar el aspecto del gráfico
plt.axis('equal')  # Asegura que el gráfico sea un círculo
plt.title("Gráfico de Sectores")

# Mostrar el gráfico
plt.show()
```

Histograma

El histograma es una herramienta visual poderosa para explorar la distribución de tus datos. La elección del número de intervalos `(bins)` es crucial para una correcta interpretación. A continuación se muestra un ejemplo sobre cómo considerar las reglas teóricas de Sturges y del criterio de la raiz como punto de partida, ajustando el número de intervalos (`bins`) según la naturaleza de los datos y del objetivo del análisis.

En el código se crea una única figura con dos subgráficos (dos histogramas) en los que se compara el número de intervalos k a considerar, bien usando el criterio de la raiz de la muestra $k = \sqrt{n}$, bien usando la regla de Sturges $1 + log_2(n)$.

Se usará `figura, ejes = plt.subplots(filas,columnas, figsize=(ancho,alto))` para crear una única figura con dos subgráficos pudiendo ajustar el tamaño en pulgadas.

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

# Datos de ejemplo: Rendimientos de un cultivo (kg/ha)
rendimiento = [4500, 5200, 4800, 5100, 4900, 5000, 4700, 5300, 4600, 5100,
         4800, 4900, 5000, 5200, 4700, 4900, 5100, 5000, 4800, 5200]

# 1. Calcular el número de bins usando la regla de Sturges
num_bins_sturges = int(1 + np.log2(len(rendimiento)))
print(f"Número de bins (Regla de Sturges): {num_bins_sturges}")

# 2. Calcular el número de bins usando el criterio de la raíz cuadrada
num_bins_raiz = int(np.sqrt(len(rendimiento)))
print(f"Número de bins (Criterio de la raíz cuadrada): {num_bins_raiz}")

# 3. Crear una única figura con dos subgráficos (1 fila, 2 columnas)
#    Tamaño ajustado para mejor visualización
fig, axs = plt.subplots(1, 2, figsize=(9, 4))

# Histograma con la regla de Sturges
axs[0].hist(rendimiento, bins=num_bins_sturges, edgecolor='red', color='skyblue')
axs[0].set_xlabel("Rendimiento del cultivo (kg/ha)")
axs[0].set_ylabel("Frecuencia")
axs[0].set_title("Regla de Sturges")
axs[0].grid(True, alpha=0.5)

# Histograma con el criterio de la raíz cuadrada
axs[1].hist(rendimiento, bins=num_bins_raiz, edgecolor='black', color='lightcoral')
axs[1].set_xlabel("Rendimiento del cultivo (kg/ha)")
axs[1].set_ylabel("Frecuencia")
```

```
axs[1].set_title("Criterio de la raíz cuadrada")
axs[1].grid(True, alpha=0.5)

# Ajustar el espaciado entre gráficos
plt.tight_layout()

plt.show() # Mostrar la figura con ambos histogramas
```

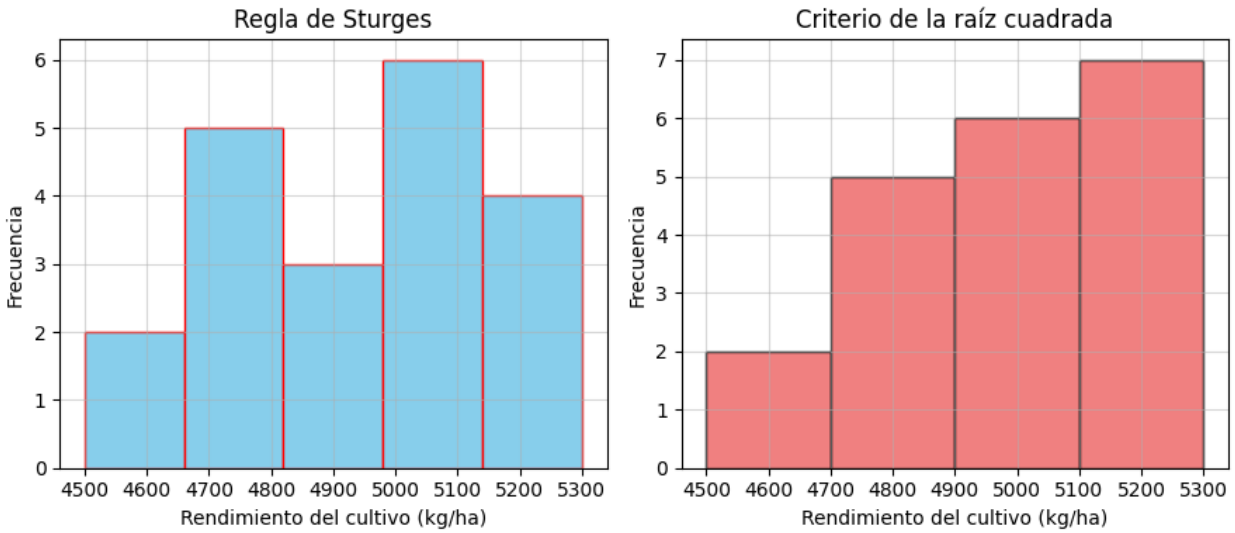

Figura 21: Histograma

En este caso, el criterio de la regla de Sturges (que generó un mayor número de intervalos) podría ser más adecuado para representar estos datos, ya que permite visualizar mejor la posible bimodalidad en la distribución de los rendimientos. Sin embargo, la elección del número de intervalos "ideal" es subjetiva y depende del objetivo del análisis. Si se busca una visión general de la distribución, el criterio de la raiz podría ser suficiente (redondeo al entero más cercano). Si se busca identificar detalles o variaciones, el criterio de la regla de Sturges podría ser más apropiado para obtener un equilibrio entre la visualización de la forma general de la distribución y la detección de posibles patrones.

Varias representaciones en un una ventana gráfica

Si ahora lo que se quiere es combinar algunas de estas gráficas en mayor número en una única ventana se puede añadir al código de las representaciones realizadas hasta ahora lo siguiente:

```
# Crear la figura y los subgráficos (1 fila, 2 columnas)
fig, axs = plt.subplots(1, 2, figsize=(12, 4)) # Ajustar figsize si necesario

# Gráfico 1: Fertilizante y rendimiento (en el primer subgráfico)
axs[0].plot(fertilizante, rendimiento, marker='o', linestyle='-')
axs[0].set_xlabel("Fertilizante nitrogenado (kg/ha)")
axs[0].set_ylabel("Rendimiento del cultivo (ton/ha)")
axs[0].set_title("Relación entre fertilizante y rendimiento")
axs[0].grid(True)

# Gráfico 2: Días y temperatura (en el segundo subgráfico)
```

```
axs[1].plot(dias, temperaturas, marker='o', linestyle='-', color='b')
axs[1].set_xlabel("Día de la semana")
axs[1].set_ylabel("Temperatura (°C)")
axs[1].set_title("Evolución de la temperatura en una semana")
axs[1].grid(True)

# Ajustar el espaciado entre los subgráficos
plt.tight_layout()

# Mostrar la figura con ambos gráficos
plt.show()
```

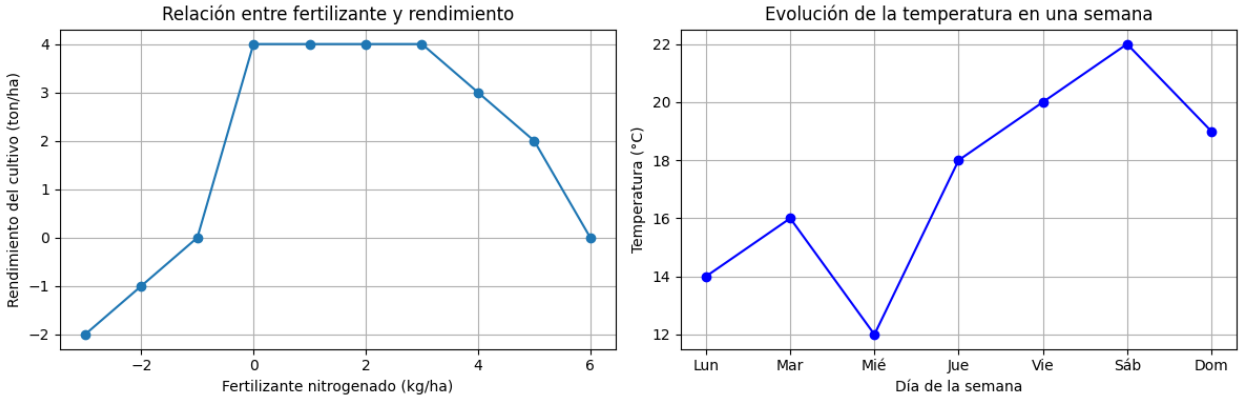

Figura 22: Dos gráficas en una fila

En el caso de más gráficos se ha de tener en cuenta que `fig` es el contenedor global de los gráficos. Si fueran más filas y columnas, `axs` sería una matriz en lugar de una lista, y se accedería con `axs[fila, columna]`. `axs` es un array con los subgráficos individuales, accesibles como axs[0], axs[1], etc. pasaría a ser `axs[0,0]` para la primera fila y primera columna. Tal como se muestre en la figura 23.

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

# 1. Crear la figura principal con 6 subgráficos
# organizados en una cuadrícula de 3x2
fig, axs = plt.subplots(3, 2, figsize=(12, 10))
# Ajustar figsize para una mejor visualización

# 2. Gráfico de línea (Subgráfico 1)

axs[0, 0].plot([1, 2, 3, 4], [5, 6, 7, 8], marker='o', linestyle='-', color='r')
axs[0, 0].set_title("Gráfico de Línea")  # Título del subgráfico
axs[0, 0].set_xlabel("Eje X")  # Etiqueta del eje X
axs[0, 0].set_ylabel("Eje Y")  # Etiqueta del eje Y

# 3. Gráfico de dispersión (Subgráfico 2)

axs[0, 1].scatter([1, 2, 3, 4], [5, 8, 7, 6], color='b')
axs[0, 1].set_title("Diagrama de Dispersión")
axs[0, 1].set_xlabel("Eje X")
axs[0, 1].set_ylabel("Eje Y")

# 4. Histograma (Subgráfico 3)
# Generamos 1000 datos aleatorios con distribución normal

datos = np.random.randn(1000)

axs[1, 0].hist(datos, bins=20, color='g', edgecolor='black')
```

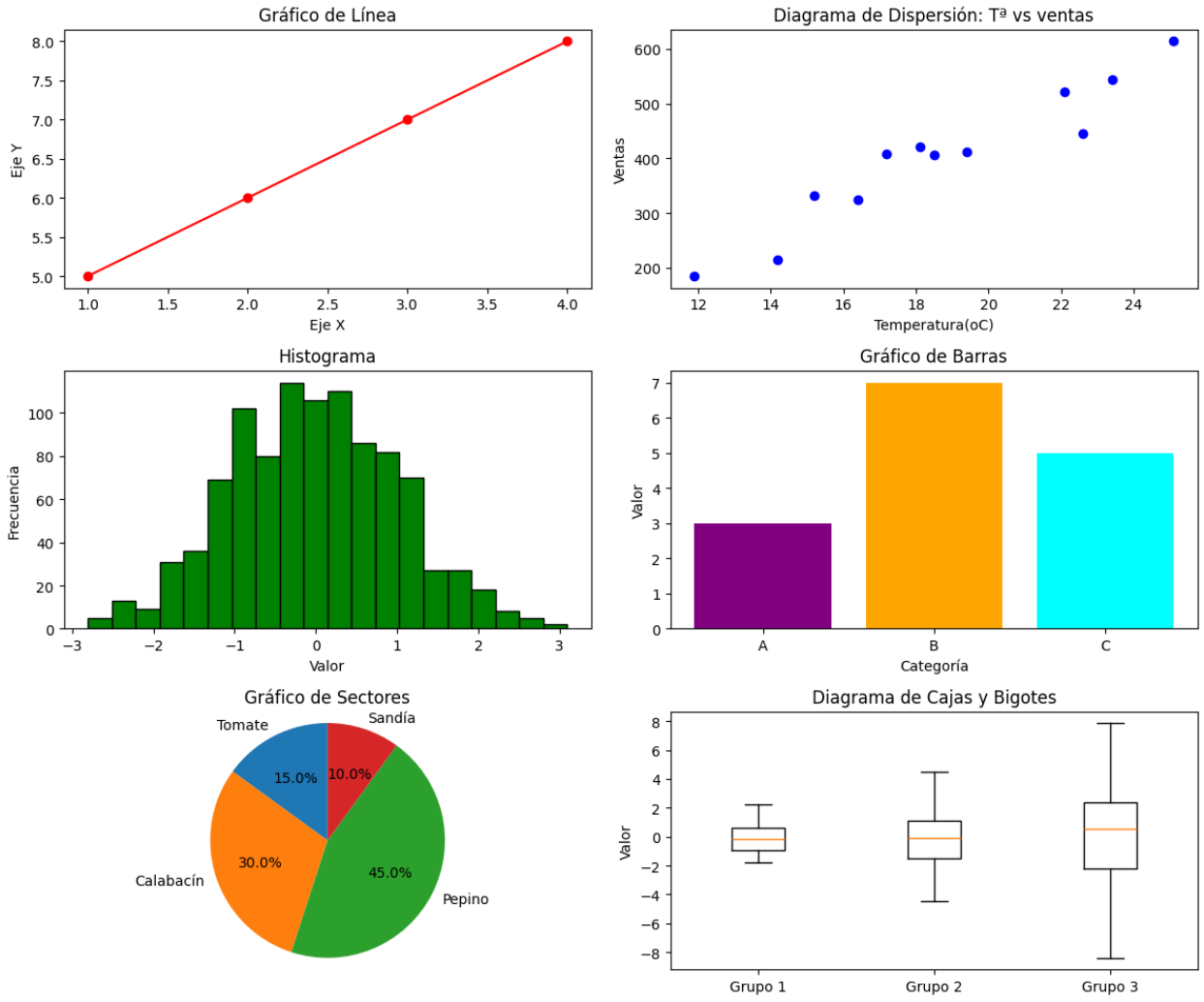

Figura 23: Gráficos combinados

```
axs[1, 0].set_title("Histograma")
axs[1, 0].set_xlabel("Valor")
axs[1, 0].set_ylabel("Frecuencia")

# 5. Gráfico de barras (Subgráfico 4)

categorias = ['A', 'B', 'C']
valores = [3, 7, 5]
axs[1, 1].bar(categorias, valores, color=['purple', 'orange', 'cyan'])

axs[1, 1].set_title("Gráfico de Barras")
axs[1, 1].set_xlabel("Categoría")
axs[1, 1].set_ylabel("Valor")

# 6. Gráfico de sectores (Subgráfico 5)

tamanos = [15, 30, 45, 10]
etiquetas = ['Tomate', 'Calabacín', 'Pepino', 'Sandía']
axs[2, 0].pie(tamanos, labels=etiquetas, autopct='%1.1f%%', startangle=90)

axs[2, 0].axis('equal')  # Asegura que el gráfico de sectores sea circular
axs[2, 0].set_title("Gráfico de Sectores")

# 7. Diagrama de cajas y bigotes (Subgráfico 6)
# Datos para el diagrama de cajas

datos_caja = [np.random.normal(0, std, 100) for std in range(1, 4)]

axs[2, 1].boxplot(datos_caja, tick_labels=['Grupo 1', 'Grupo 2', 'Grupo 3'])
axs[2, 1].set_title("Diagrama de Cajas y Bigotes")
axs[2, 1].set_ylabel("Valor")

# 8. Ajustar el diseño para evitar solapamientos
plt.tight_layout()

# 9. Mostrar la figura con todos los subgráficos
plt.show()
```

Diagrama de cajas y bigotes

El diagrama de cajas y bigotes `boxplot()` permite visualizar la distribución de una variable. Es útil para identificar valores atípicos, la dispersión y la simetría de los datos (figura 24). Un ejemplo sencillo es el siguiente:

```
import matplotlib.pyplot as plt
notas = [4, 8, 7.5, 6, 5.5, 5.2, 3.5, 7.7, 3.2, 9, 6.8, -2]
# se fueza un outliner incluyendo un -2
plt.boxplot(notas)
plt.title("Boxplot con Matplotlib")
plt.show()
```

Sin embargo, la función `boxplot` proporciona varios argumentos para personalizar el gráfico de cajas por defecto:

- `vert = False`: crea un box plot horizontal en lugar de vertical.
- `notch=True`: muestra el intervalo de confianza al 95 % para la mediana. El intervalo se mostrará con muescas (notch, en inglés) sobre la caja.
- `flierprops`: El símbolo por defecto para representar los outliers (datos atípicos) son círculos. Sin embargo, puede personalizarse el marcador y su color pasando un diccionario al argumento.
- `showmeans`: presenta la media aritmética en color verde si su valor es `True`.

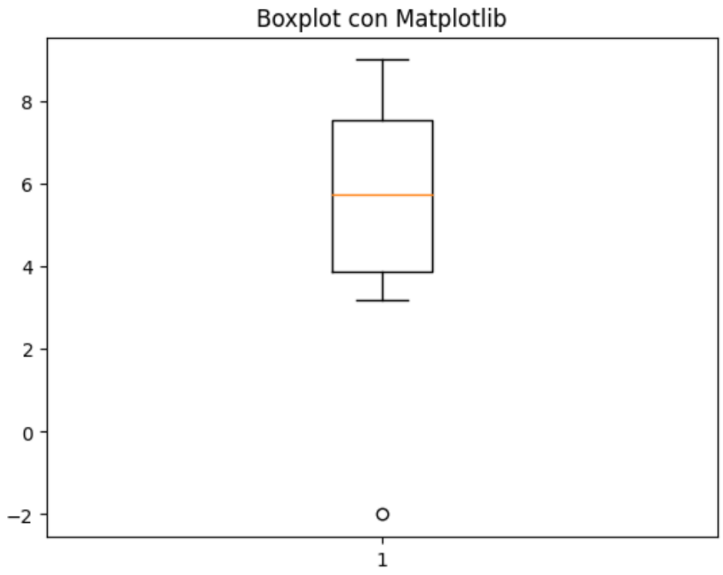

Figura 24: Diagrama de cajas y bigotes

- `meanline`: presenta la media aritmética como una línea de color verde por defecto si su valor es `True`.
- `showfliers = False`: hace que no aparezcan los valores atípicos en el gráfico.

Con respecto a los colores:

- `patch_artist = True`: modifica el color de las cajas de un gráfico de cajas que por defecto es blanco. Puede sobrescribirse estableciendo el valor a True y pasando un diccionario con `facecolor` al argumento `boxprops`, tal y como se muestra en el siguiente bloque de código.
- `medianprops`: El color por defecto de la línea que representa la mediana es naranja, pero puede cambiarse pasando un diccionario a `medianprops`. También se puede modificar el grosor de la línea de esta forma.

Así pues, otra representación de los datos anteriores es la siguiente, en donde se define el gráfico `bp` para aplicarle una personalización visual que se comenta en el propio código que sigue a continuación (figura 25):

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

# Datos de ejemplo (notas)
notas = [4, 8, 7.5, 6, 5.5, 5.2, 3.5, 7.7, 3.2, 9, 6.8, -2]

# Creación del boxplot mejorado
plt.figure(figsize=(8, 6))  # Ajusta el tamaño de la figura
bp = plt.boxplot(notas,
    vert=False,             # Boxplot horizontal
    patch_artist=True,      # Rellena las cajas con color
    notch=True,             # Muestra el intervalo de
                            # confianza para la mediana
    showmeans=True,         # Muestra la media
```

```
    meanline=True,          # Muestra la media como una línea
    flierprops={'marker': 'o', 'markerfacecolor': 'red', 'markersize': 8})
                            # Personaliza los outliers

# Personalización visual
 # Color de la caja
plt.setp(bp['boxes'], facecolor='lightblue', edgecolor='black')
# Color y grosor de la mediana
plt.setp(bp['medians'], color='red', linewidth=2)
# Color y grosor de la media
plt.setp(bp['means'], color='green', linewidth=2)
# Color y grosor de los bigotes
plt.setp(bp['whiskers'], color='gray', linewidth=1.5)
# Color y grosor de los límites
plt.setp(bp['caps'], color='black', linewidth=2)

# Agregar títulos y etiquetas
plt.title("Boxplot con Matplotlib", fontsize=14)
plt.xlabel("Notas", fontsize=12)
plt.grid(True, linestyle='--', alpha=0.6)  # Agregar cuadrícula

# Mostrar gráfico
plt.show()
```

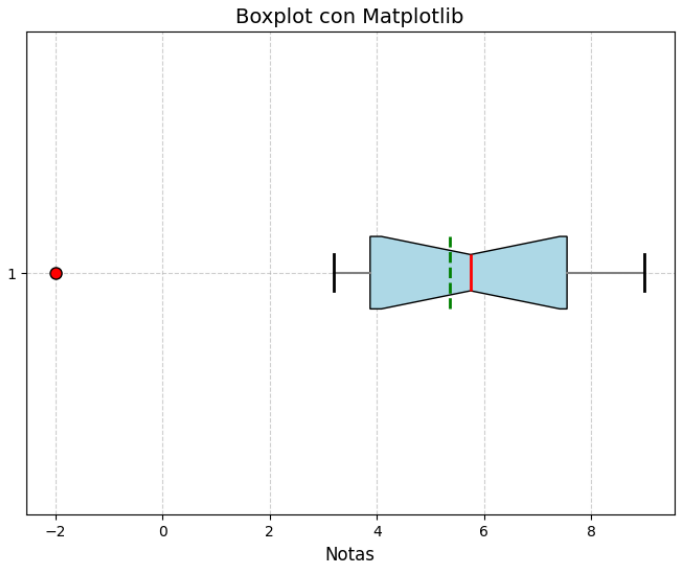

Figura 25: Diagrama de cajas y bigotes mejorado

Finalmente, para concluir este apartado se presenta el estudio comparativo de varias variables mediante este tipo de gráficos. Se presenta el caso del estudio de la producción para dos parcelas (extensible a más) (figura 26):

```
import matplotlib.pyplot as plt

# Datos de producción (kg/ha) para cada parcela
parcela_a = [3649, 3478, 3595, 3896, 3458, 3578, 3355, 3678, 3789, 3456, 3567,
             3456, 3678, 3789, 3567, 3456, 3345, 3567, 3678, 3789, 3456, 3567,
             3456, 3678, 3789, 3567, 3456, 3345, 3567, 3678]
parcela_b = [4349, 4178, 4295, 4596, 4158, 4278, 4055, 4378, 4489, 4156, 4489,
             4156, 4267, 4156, 4378, 4489, 4267, 4156, 4045, 4267, 4378, 4489,
             4156, 4267, 4156, 4378, 4489, 4267, 4156, 4045]

# Creación del boxplot
plt.figure(figsize=(8, 6))
plt.boxplot([parcela_a, parcela_b],
```

```
            tick_labels=['Parcela A', 'Parcela B'],
            patch_artist=True  # Rellena las cajas con color
            )
# Agregar títulos y etiquetas
plt.title("Distribución del Rendimiento de Trigo por Parcela")
plt.xlabel("Parcelas")
plt.ylabel("Producción (kg/ha)")
plt.grid(True, linestyle="--", alpha=0.4)  # Agrega una cuadrícula
plt.ylim(3000,5000)

plt.show()
```

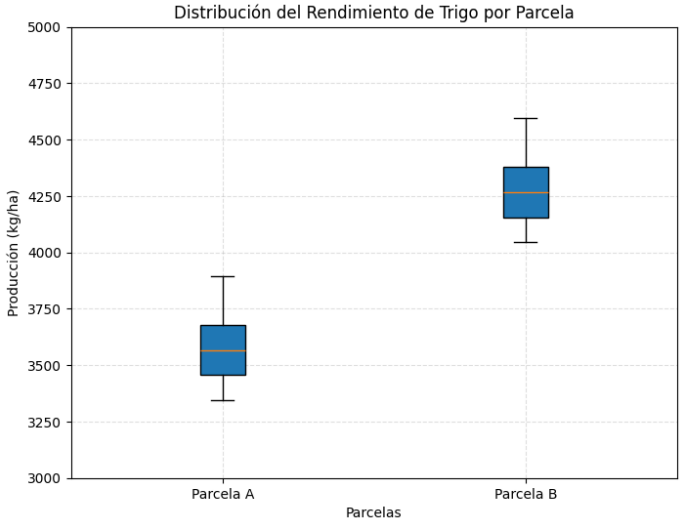

Figura 26: Diagrama de cajas y bigotes sobre dos valores

Gráficos de violín

Para la creación del gráfico de violin `violinplot()` se va a partir de 50 datos para cada parcela que siguen una distribución normal cada una. Entre los argumentos que se utilizan se tienen las opciones:

- `showmeans = True` → Muestra un marcador para la media.
- `showmedians = True` → Muestra un marcador para la mediana.
- `showextrema = True` → Muestra los valores mínimos y máximos.
- `bw_method = 0.5` → Controla la suavidad de la forma del violín. Y para personalizarlo:
- `plt.setp(vp['bodies'][0],facecolor='lightblue',edgecolor='black')`: primera variable de color azul claro.
- `plt.setp(vp['bodies'][1],facecolor='lightgreen',edgecolor='black')`: segunda variable de color verde claro.
- `plt.setp(vp['cmeans'], edgecolor='red', linewidth = 2)`: Color de la media en rojo
- `plt.setp(vp['cmedians'], edgecolor = 'green', linewidth = 2)`: Color de la mediana en verde

- `plt.setp(vp['cbars'], color = 'gray', linewidth = 1.5)`: Color de los valores extremos en gris

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

# Datos de producción (kg/ha) para cada parcela, modificados para mayor diferencia
# Media 3500, desviación estándar 300
parcela_a = np.random.normal(3500, 300, 50)
# Media 4500, desviación estándar 500
parcela_b = np.random.normal(4500, 500, 50)


# Creación del gráfico de violín (el resto del código es igual)
plt.figure(figsize=(8, 6))
vp = plt.violinplot([parcela_a, parcela_b],
                    showmeans=True,    # Mostrar la media
                    showmedians=True,  # Mostrar la mediana
                    showextrema=True,  # Mostrar los valores extremos
                    bw_method=0.5)     # Ajusta el ancho de banda

# Personalización visual para diferentes formas
# Color y forma  Parcela A
plt.setp(vp['bodies'][0], facecolor='lightblue', edgecolor='black')
# Color y forma  Parcela B
plt.setp(vp['bodies'][1], facecolor='lightgreen', edgecolor='black')

# Resto de la personalización visual
# Color y grosor de la media
plt.setp(vp['cmeans'], edgecolor='red', linewidth=2)
# Color y grosor de la mediana
plt.setp(vp['cmedians'], edgecolor='green', linewidth=2)
# Color y grosor de las barras
plt.setp(vp['cbars'], color='gray', linewidth=1.5)

# Agregar títulos y etiquetas
plt.title("Distribución del Rendimiento de Trigo por Parcela", fontsize=14)
# Etiquetas del eje x
plt.xticks([1, 2], ['Parcela A', 'Parcela B'], fontsize=12)
plt.ylabel("Producción (kg/ha)", fontsize=12)
# Agrega una cuadrícula
plt.grid(True, linestyle="--", alpha=0.6)
# Ajustamos los límites del eje y para visualizar mejor
plt.ylim(2000, 6000)
# Mostrar gráfico
plt.show()
```

El gráfico de violín es una herramienta útil porque combina información de un diagrama de cajas con una distribución de densidad, lo que permite visualizar tanto la dispersión como los valores centrales (media y mediana) en una misma representación.

El gráfico de violín permite visualizar con mayor claridad las diferencias en la distribución de la producción de trigo entre las dos parcelas. Se observa que la Parcela B tiene un rendimiento promedio superior al de la Parcela A, sin embargo, esta ventaja viene acompañada de una mayor variabilidad en la producción, lo que sugiere que sus rendimientos son menos predecibles.

Desde un punto de vista estadístico, la Parcela A muestra una distribución más concentrada alrededor de su media, lo que indica mayor estabilidad en la producción. En contraste, la Parcela B presenta una mayor dispersión en sus valores, lo que puede deberse a factores como variabilidad en el suelo, riego o condiciones climáticas.

Esta información es fundamental en la toma de decisiones agrícolas, permitiendo evaluar tanto

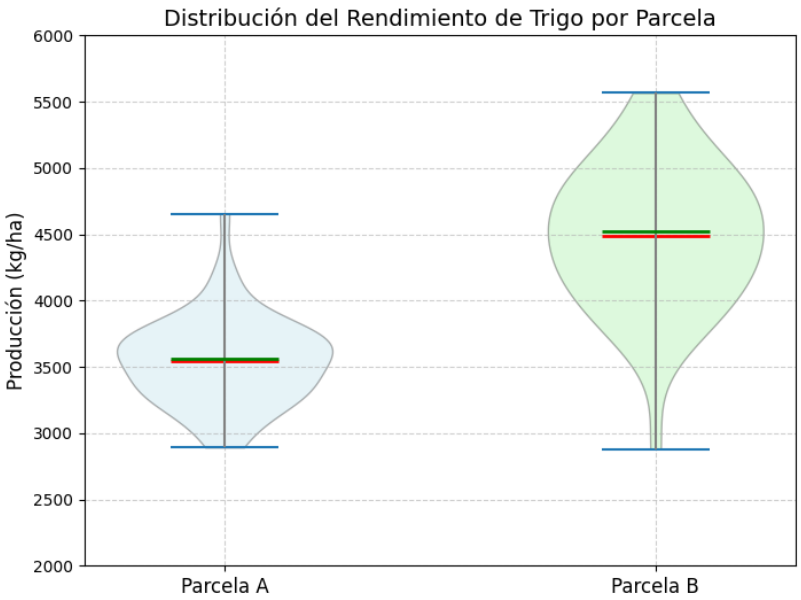

Figura 27: Gráfico de violin

el rendimiento esperado, como la estabilidad del cultivo en diferentes condiciones. Un agricultor podría preferir una parcela con menor variabilidad si busca resultados más predecibles, o bien, podría analizar estrategias para reducir la dispersión en la Parcela B y mejorar su eficiencia productiva.

Es importante recordar que en este caso los datos han sido generados aleatoriamente y no corresponden a un estudio real. En un análisis práctico, sería necesario trabajar con datos reales obtenidos mediante mediciones en campo para extraer conclusiones válidas y tomar decisiones informadas.

Gráfico de áreas apiladas

El uso de este tipo de gráfico es útil para representar series de tiempo con varias categorías. Permite observar tendencias generales y comparativas e la producción. La personalización de colores, leyendas y cuadrículas mejor la interpretación visual. Con `stackplot()` se va a crear un gráfico de áreas apiladas para visualizar la evolución de la producción de tres cultivos relevantes en la provincia de Almería durante un período de 5 años.

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

# Datos de producción (toneladas) para cada cultivo (aprox. para 2023-2027)
anio = ['2021', '2022','2023', '2024', '2025']
tomate = [1500000, 1550000, 1600000, 1650000, 1700000]
pimiento = [500000, 520000, 540000, 560000, 580000]
pepino = [300000, 310000, 320000, 330000, 340000]

# Creación del gráfico de áreas apiladas
plt.figure(figsize=(10, 6))
```

```
plt.stackplot(anio, tomate, pimiento, pepino,
              labels=['Tomate', 'Pimiento', 'Pepino'],
              colors=['#FF6347', '#90EE90', '#7FFFD4'])  # Colores

# Personalización del gráfico
plt.title('Producción de Cultivos en Almería (2021-2025)', fontsize=16)
plt.xlabel('Año', fontsize=12)
plt.ylabel('Producción (Toneladas)', fontsize=12)
plt.legend(loc='upper left', fontsize=10)
plt.grid(True, linestyle='--', alpha=0.6)

plt.show()
```

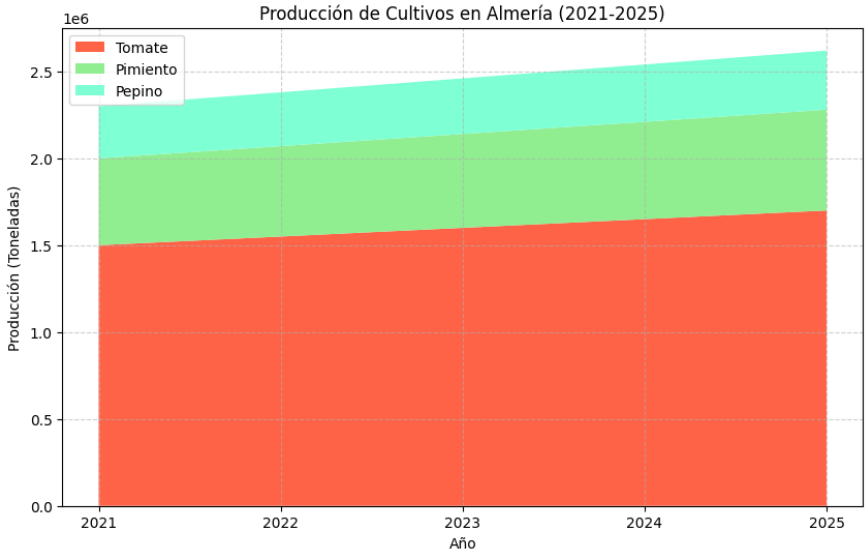

Figura 28: Superficies apiladas

Se observa un incremento progresivo en la producción de los tres cultivos entre 2021 y 2025. El tomate es el principal cultivo, seguido por el pimiento y el pepino. El crecimiento es aproximadamente lineal en los tres casos. Este monitoreo de producción permite visualizar la evoluación de la producción de cultivos.

- El crecimiento en los tres cultivos es uniforme, lo que sugiere estabilidad en la producción.
- Un crecimiento estable en la producción sugiere buenas condiciones de mercado y tecnología agrícola avanzada.

Casos aplicados de análisis de datos y la probabilidad: de estadística descriptiva a la inferencia estadística

Este capítulo se centra en la resolución de problemas relacionados con la estadística y la probabilidad, haciendo uso de diversas herramientas computacionales que potencian el análisis de datos. Para ello, se revisan estructuras de datos fundamentales en Python, como listas, diccionarios, arreglos `ndarray` de Numpy y `dataframes` de Pandas, que permiten organizar y manipular la información de forma eficiente. Estas estructuras no solo facilitan el manejo de datos, sino que también abren la puerta a una exploración más profunda de los conceptos estadísticos mediante la programación. La estadística tiene aplicaciones en múltiples áreas del conocimiento, como la biología, la economía, la ingeniería, la psicología y muchas más, lo cual la convierte en una herramienta transversal y poderosa. Comprender sus fundamentos es esencial para poder interpretar, modelar y tomar decisiones basadas en datos, independientemente del campo de aplicación. Por esta razón, los programas de software estadístico permiten obtener resultados precisos sin necesidad de desentrañar el desarrollo matemático de los algoritmos que subyacen a los procedimientos. No obstante, para lograr una comprensión sólida y crítica de los métodos estadísticos, es igualmente importante explorar la lógica y estructura de los algoritmos que los hacen posibles, y es en ese punto donde la programación adquiere un papel clave en el proceso de aprendizaje.

En este capítulo se abordarán los conceptos estadísticos desde una doble perspectiva: por un lado, desarrollando los algoritmos que los sustentan, y por otro, utilizando librerías especializadas que permiten evidenciar su potencia y facilitar su comprensión. El objetivo es que estos conceptos resulten accesibles y útiles para la formación del lector. En línea con el propósito general del libro, se ofrece un tratamiento elemental pero completo de la estadística, abarcando temas como la estadística descriptiva, tablas de frecuencias unidimensionales y bidimensionales, independencia y correlación, variables aleatorias, probabilidades e inferencia estadística.

Uso de librerías Python para la estadística

Se incluye una tabla con las fórmulas en Python para calcular estadísticas básicas de una distribución de una variable utilizando distintos enfoques con `numpy`, con `scipy.stats` y con `pandas`.

Estadísticos	**Código con `numpy`**
Moda	`np.bincount(datos).argmax()` *(solo datos enteros y discretos)*
Mediana	`np.median(datos)`
Media	`np.mean(datos)`
Cuartiles y percentiles *(con k de 1 a 100)*	`np.percentile(datos, k)`
Varianza	`np.var(datos, ddof=0)`
Desviación típica	`np.std(datos, ddof=0)`

Estadísticos	**Código con `scipy.stats`**
Moda	`stats.mode(datos, keepdims=True).mode[0]`
Cuartiles y percentiles *(con k de 1 a 100)*	`stats.scoreatpercentile(datos, k)`
Mediana	`stats.scoreatpercentile(datos, 50)`
Media	`stats.tmean(datos)`
Varianza	`stats.tvar(datos)`
Desviación típica	`stats.tstd(datos)`
Coeficiente de asimetría	`stats.skew(datos, bias=False)`
Curtosis	`stats.kurtosis(datos, fisher=False)`

Estadísticos	**Código con `pandas`**
Media	`df['col'].mean()`
Moda	`pd.Series(datos).mode().values` o `df.mode().iloc[]`
Varianza	`df['col'].var(ddof=0)`
Desviación típica	`df['col'].std(ddof=0)`
Coeficiente de asimetría	`df['col'].skew()`
Curtosis	`df['col'].kurtosis(fisher=False)`
Cuartiles y percentiles *(con k de 1 a 100)*	`pd.Series(datos).quantile(k/100)` o `df.quantile(k/100)`
Mediana	`df['col'].median()`

- Como se ha observado en las declaraciones el argumento `ddfof=0` en `numpy` y `pandas` indica que se calcula la varianza y la desviación típica con la fórmula para la población (dividido entre el total, sin la corrección de Bessel). Para la erión muestral, usar `ddof=1`
- En `scipy.stats`, `stats.kurtosis()` se devuelve la curtosis de Fisher por defecto (restando 3). Para obtener la curtosis de Pearson (sin restar 3), se debe usar `fisher=False`.
- **SciPy.stats** y **pandas** calculan varianza, curtosis y asimetría usando por defecto la corrección de Bessel (es decir, dividiendo por $n-1$ para obtener estimaciones muestrales), mientras que **NumPy** no.
  - Sin embargo, todas permiten cambiar este comportamiento pasando el argumento (`ddof=0`) o bien multiplicar por $\frac{n-1}{n}$ después de tener calculado previamente algún estadístico. Cuando se trabajen con datos grandes este factor de corrección no es recomendable utilizarlo (es mejor usar `ddof=0` por eficiencia y precisión).
  - En el caso de querer la muestral, cambiar `bias False`.

La librería `numpy` no tiene incorporadas funciones para el cálculo de la asimetría y la curtosis, se han de construir. Recuérdese que si el tamaño muestral es pequeño, la diferencia entre ambos métodos puede ser significativa. Así pues:

	Cálculo poblacional	**Muestral (ddof=1)**
Desviación estándar (σ)	$\sqrt{\sigma^2}$	`np.std(data, ddof=0)` o bien si ya lo tenemos calculado, se toma su raíz cuadrada `np.sqrt(data)`
Asimetría (γ_1)	$\frac{\frac{1}{n}\sum(x_i-\bar{x})^3}{\sigma^3}$	`asimetria = np.sum((data - mean_x) ** 3) / (n * std_x**3)`
Curtosis (γ_2)	$\frac{\frac{1}{n}\sum(x_i-\bar{x})^4}{\sigma^4} - 3$	`curtosis = np.sum((data - mean_x) ** 4) / (n * std_x**4) - 3`

Cada una de estas librerías tiene sus fortalezas y debilidades dependiendo del contexto de uso. NumPy es ideal para operaciones numéricas rápidas y eficientes sobre colecciones de datos, Pandas destaca en el manejo de datos estructurados, y SciPy es adecuada cuando se requieren técnicas estadísticas más avanzadas. La elección dependerá del tipo de análisis que se desee realizar, de la estructura de los datos y de las necesidades del proyecto. Sin embargo, dominar las tres librerías y entender cómo combinar sus capacidades permitirá realizar análisis estadísticos poderosos y flexibles en cualquier área, desde la ingeniería agronómica hasta la economía o la biología.

Una vez introducidas las principales ideas y casos de uso de las librerías, se procede a la resolución de distintos problemas. El enfoque se centrará en las medidas descriptivas, esenciales para resumir y comprender las características básicas de un conjunto de datos.

Asimetría y curtosis

Diseñar un programa en Python que calcule la asimetría y la curtosis de un conjunto de datos dado desde cuatro enfoques diferentes:

1. **Cálculo manual:** Implementar las fórmulas matemáticas de asimetría y curtosis directamente sin utilizar librerías especializadas.
2. **Utilizando NumPy:** Aprovechar las funciones de la librería NumPy para realizar los cálculos necesarios.
3. **Utilizando SciPy:** Emplear las funciones específicas para asimetría y curtosis disponibles en la librería SciPy.
4. **Utilizando Pandas:** Convertir los datos a una Serie de Pandas y usar sus métodos integrados para calcular la asimetría y la curtosis.

```
import numpy as np
import pandas as pd
import scipy.stats as stats

datos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# 1. Sin librerías (cálculos manuales)
def asimetria(datos):
    n = len(datos)
    media = np.mean(datos)
    s = np.std(datos, ddof=1)
    return (n / ((n - 1) * (n - 2))) * np.sum(((data - media) / s) ** 3)

def curtosis(datos):
    n = len(datos)
    media = np.mean(datos)
    s = np.std(datos, ddof=1)
    return (n * (n+1) / ((n-1) * (n-2) * (n-3))) * \
        np.sum(((data - media) / s) ** 4) - (3 * (n-1) ** 2) / ((n-2) * (n-3))

# 2. NumPy (funciones NumPy para cálculos)
def asimetria_numpy(datos):
    n = len(datos)
    media = np.mean(datos)
    s = np.std(data, ddof=1)
    return (n / ((n - 1) * (n - 2))) * np.sum(((data - media) / s) ** 3)

def curtosis_numpy(datos):
    n = len(datos)
    media = np.mean(datos)
    s = np.std(datos, ddof=1)
    return (n * (n + 1) / ((n - 1) * (n - 2) * (n - 3))) * \
        np.sum(((data - media) / s) ** 4) - (3 * (n-1) ** 2) / ((n-2) * (n-3))

# 3. SciPy (usando las funciones de SciPy)
asim_scipy = stats.skew(datos)
curt_scipy = stats.kurtosis(datos)

# 4. Pandas (usando los métodos de Pandas)
data_series = pd.Series(datos)
asim_pandas = data_series.skew()
curt_pandas = data_series.kurt()

print("Asimetría (Sin librerías):", asimetria(datos))
print("Curtosis (Sin librerías):", curtosis(datos))
print("Asimetría (NumPy):", asimetria_numpy(datos))
print("Curtosis (NumPy):", curtosis_numpy(datos))
print("Asimetría (SciPy):", asim_scipy)
print("Curtosis (SciPy):", curt_scipy)
print("Asimetría (Pandas):", asim_pandas)
print("Curtosis (Pandas):", curt_pandas)
```

Regresión lineal

En la siguiente sección de aplicación de un caso a la ciencia de datos se trabajará un caso parecido a este para trabajar con la regresión lineal. Aquí se va a resolver con la librería `statistics`.

```
import random
import statistics as st

# 1. Generar datos para las variables x e y
# Parámetros de la distribución normal para x
media_x = 10            # Media de la distribución de x
desviacion_x = 5        # Desviación estándar de la distribución de x
num_datos = 100         # Número de datos a generar

# Generar datos para x usando random.gauss()
x = [random.gauss(media_x, desviacion_x) for _ in range(num_datos)]

# Generar y como una función lineal de x más un componente aleatorio
# para lograr una correlación cercana a 1
y = [2 * xi + 5 + random.gauss(0, 5) for xi in x]       # y = 2x + 5 + ruido

# 2. Calcular la correlación con statistics
correlacion = st.correlation(x, y)
print(f"Correlación entre x e y: {correlacion:.4f}")

# 3. Visualizar los datos con la recta de regresión
# Calcular la recta de regresión
pendiente, interseccion = st.linear_regression(x, y)

# Generar puntos para la recta de regresión
x_recta = [min(x), max(x)] # interv. donde está el rango de datos de la recta
y_recta = [pendiente * xi + interseccion for xi in x_recta] # intervalo imagen

# Ecuación de la recta de regresión como string
ecuacion_recta = f'y = {pendiente:.2f}x + {interseccion:.2f}'
# Imprimir la ecuación de la recta en la salida
print(f"Ecuación de la recta de regresión: {ecuacion_recta}")
```

La salida de este programa mostrará algo similar a:

```
Correlación entre x e y: 0.8823
Ecuación de la recta de regresión: y = 1.99x + 5.31
```

Por último, se le puede agregar el código del gráfico sin necesidad de incorporar ninguna librería adicional utilizando `plt.scatter(x,y)` para crear el diagrama de dispersión para cada par de puntos (x_i, y_i). El código y el resultado podrían ser tal que así:

```
# Representar los puntos y la recta de regresión
plt.scatter(x, y)
plt.plot(x_recta, y_recta, color='red', label='Recta de regresión')

# Agregar la ecuación al gráfico
plt.text(min(x), max(y), ecuacion_recta, color='red', fontsize=12)

plt.title('Diagrama de dispersión de x e y con recta de regresión')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True, linestyle='dashed')
plt.show()
```

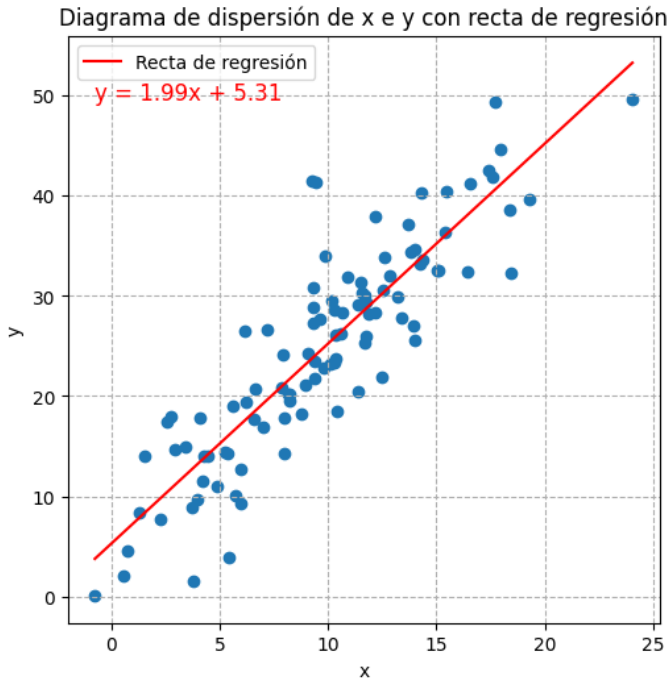

Figura 29: Recta de regresión

Probabilidad

Ejercicio resuelto: Experimento dado En el experimento aleatorio de lanzar un dado, calcular las probabilidades de ser par, impar y la probabilidad de la intersección de sucesos.

Solución sin comprensión de listas

```
E = list(range(1, 7))  # Espacio muestral
n = len(E)  # Total de la muestra

# Suceso A: Probabilidad de ser par
A = []
for i in E:
    if i % 2 == 0:
        A.append(i)
pA = len(A) / float(n)
print(f"La probabilidad de ser par es {pA}")

# Suceso B: Probabilidad de ser impar
B = []
for i in E:
    if i % 2 != 0:
        B.append(i)
pB = len(B) / float(n)
print(f"La probabilidad de ser impar es {pB}")

# Probabilidad de la intersección de sucesos A y B  (sucesos mutuamente excluyentes)
pAB = 0             # Dado que A y B son mutuamente excluyentes
print(f"La probabilidad de la intersección de A y B es {pAB}")
```

Solución usando comprensión de listas

```
E = list(range(1, 7))  # Espacio muestral
n = len(E)  # Total de la muestra
```

```
# Suceso A: Probabilidad de ser par
A = [i for i in E if i % 2 == 0]
pA = len(A) / float(n)
print(f"La probabilidad de ser par es {pA}")

# Suceso B: Probabilidad de ser impar
B = [i for i in E if i % 2 != 0]
pB = len(B) / float(n)
print(f"La probabilidad de ser impar es {pB}")

# Probabilidad de la intersección de sucesos A y B  (sucesos mutuamente excluyentes)
pAB = 0              # Dado que A y B son mutuamente excluyentes
print(f"La probabilidad de la intersección de A y B es {pAB}")
```

Ejercicio resuelto: Construir el programa que incluya un menú de calculadora de probabilidades, permitiendo calcular la probabilidad de la unión de dos sucesos A y B, y también calcule la probabilidad condicionada.

```
def validar_probabilidad(mensaje):
    """ Solicita una probabilidad entre 0 y 1,
        validando la entrada del usuario."""
    while True:
        try:
            valor = float(input(mensaje))
            if 0 <= valor <= 1:
                return valor
            else:
                print(" Error: La probabilidad debe estar entre 0 y 1.")
        except ValueError:
            print(" Error: Introduzca un número válido entre 0 y 1.")

def calcular_union(prob_A, prob_B, prob_interseccion):
    """ Calcula P( A U B) = P(A) +P(B) - P (A interseccion B), asegurando
        que el resultado esté entre 0 y 1."""
    resultado = prob_A + prob_B - prob_interseccion
    if 0<=resultado <=1:
        return resultado
    else:
        return None

def calcular_condicionada(prob_A_interseccion_B, prob_B):
    """Calcula P(A | B) = P(A interseccion B) / P(B),
        validando que la salida esté entre 0 y 1."""
    if prob_B > 0 and prob_A_interseccion_B <= prob_B:
        return prob_A_interseccion_B / prob_B
    else:
        return None

def main():
    print("Calculadora de Probabilidad")
    print("===========================")
    while True:
        print("\nSeleccione una opción:")
        print("1. Calcular la unión de dos eventos: P(A U B)")
        print("2. Calcular la probabilidad condicionada: P(A | B)")
        print("3. Salir")

        opcion = input("¿Qué desea hacer? Escriba el número de opción: ")

        if opcion == "1":
            prob_A = validar_probabilidad("Introduzca el valor de P(A): ")
            prob_B = validar_probabilidad("Introduzca el valor de P(B): ")
            prob_interseccion = validar_probabilidad("Introduzca el valor \
                de P(A interseccion B): ")
            union = calcular_union(prob_A, prob_B, prob_interseccion)
            if union is None:
                print("Error: Revise las probabilidades de los sucesos.")
            else:
                print(f"\n\tP(A U B) = {union:.4f} ")
```

```
        elif opcion == "2":
            prob_A_interseccion_B = validar_probabilidad("Introduzca \
                P(A interseccion B): ")
            prob_B = validar_probabilidad("Introduzca P(B): ")
            condicionada = calcular_condicionada(prob_A_interseccion_B, \
                prob_B)
            if condicionada is not None:
                print(f"\n\tP(A | B) = {condicionada:.4f} ")
            else:
                print("Error: Revise las probabilidades de los sucesos.")

        elif opcion == "3":
            print("¡Gracias por usar la calculadora de probabilidad!")
            break

        else:
            print("Opción no válida. Intente de nuevo.")

# Ejecutar el programa
main()
```

Sobre el uso de menús en Python

En muchos lenguajes de programación como Java, C o C++, cuando se quiere hacer un menú de opciones, se suele usar una estructura `switch` o `switch-case` para ejecutar diferentes bloques de código según el valor de una variable.

Sin embargo, en Python no existe `switch`. Aunque ya se ha usado a lo largo del contenido de este libro una estructura `if-elif-else` tradicional en los menús, esto puede hacer que el código crezca en complejidad y repetición. Una alternativa más elegante y permitida en Python es usar un diccionario de funciones, con lo que se mejora la legibilidad y escalabilidad del código, y es muy útil para programas educativos, menús interactivos, simulaciones, juegos de texto, etc.

Así pues, se utilizarán: - Las claves del diccionario para representar las opciones del menú. - Los valores del diccionario son las funciones que se deben ejecutar.

Cuando el usuario elija una opción, se llamará a la función asociada con esa clave. Es decir, el valor de la clave en el diccionario tiene que ser el mismo que el nombre de la función `def`.

De forma simplificada la estructura del código sería:

```
def f_opcion_1() :
    ...
    ...
def f_opcion_2() :
    ...

...
menu = {
    "1": f_opcion_1,
    "2": f_opcion_2,
    "0": salir
}

if opcion in menu:
    menu[opcion]()

    #al agregar paréntesis se está llamando a la función
```

El funcionamiento se ve en el siguiente ejercicio resuelto.

Ejercicio resuelto: Menú con diccionario

Diseñar un programa en Python que simule un conjunto de tareas básicas relacionadas con la agronomía y la estadística, como si formaran parte de un sistema de análisis agroambiental.

Para organizar el flujo del programa, deberás utilizar un menú interactivo basado en un diccionario de funciones, en lugar de estructuras if-elif-else tradicionales. La estructura del menú es la siguiente:

- Simular precipitación: genera un valor aleatorio entre 0 y 100 mm usando random.uniform, y lo muestra con 2 decimales.
- Estimar rendimiento del cultivo: genera un número a partir de una distribución normal (media 5000, desviación estándar 400) con random.gauss.
- Calcular media y desviación estándar: pide al usuario una serie de números separados por espacios y calcula su media y desviación usando el módulo statistics.
- Simular aparición de plagas: usa random.choices con pesos para simular qué tipo de plagas aparecen entre tres tipos posibles.
- Salir del programa.

```
import random
import statistics

def simular_precipitacion():
    lluvia = random.uniform(0, 100)  # mm de lluvia simulada
    print(f"\n Precipitación simulada: {lluvia:.2f} mm")

def estimar_rendimiento():
    rendimiento = random.gauss(5000, 400)  # media: 5000 kg/ha, sigma: 400
    print(f"\n Rendimiento estimado del cultivo: {rendimiento:.2f} kg/ha")

def calcular_estadisticas():
    datos = input("\nIntroduce una serie de datos separados por espacios: ")
    try:
        numeros = [float(x) for x in datos.split()]
        media = statistics.mean(numeros)
        desviacion = statistics.stdev(numeros)
        print(f"- Media: {media:.2f}")
        print(f"- Desviación estándar: {desviacion:.2f}")
    except:
        print("- Error: Asegúrate de introducir números válidos separados por espacios.")

def simular_plagas():
    plagas = ['pulgón', 'araña roja', 'trips']
    probabilidades = [0.5, 0.3, 0.2]
    observadas = random.choices(plagas, weights=probabilidades, k=5)
    print(f"\n- Plagas observadas: {observadas}")

def salir():
    print("\n---> Saliendo del programa. ¡Hasta pronto!")

# Diccionario de opciones
menu = {
    "1": simular_precipitacion,
    "2": estimar_rendimiento,
    "3": calcular_estadisticas,
    "4": simular_plagas,
    "0": salir
}

# Bucle del menú
while True:
    print("\n=== Menú Agronómico ===")
    print("1. Simular precipitación")
    print("2. Estimar rendimiento del cultivo")
```

```
    print("3. Calcular media y desviación estándar")
    print("4. Simular aparición de plagas")
    print("0. Salir")

    opcion = input("Elige una opción: ")

    if opcion in menu:
        menu[opcion]()
        if opcion == "0":
            break
    else:
        print("---> Opción no válida. Intenta de nuevo.")
```

Ejercicio resuelto: Implementar en Python un ejemplo que permita simular la Ley de los grandes números. La *Ley de los Grandes Números* establece que, al repetir un experimento aleatorio un gran número de veces, la *frecuencia relativa de un evento* tiende a estabilizarse en torno a su *probabilidad teórica*. Utiliza la simulación del lanzamiento de una moneda. A modo de ejemplo aquí se presenta una de las simulaciones:

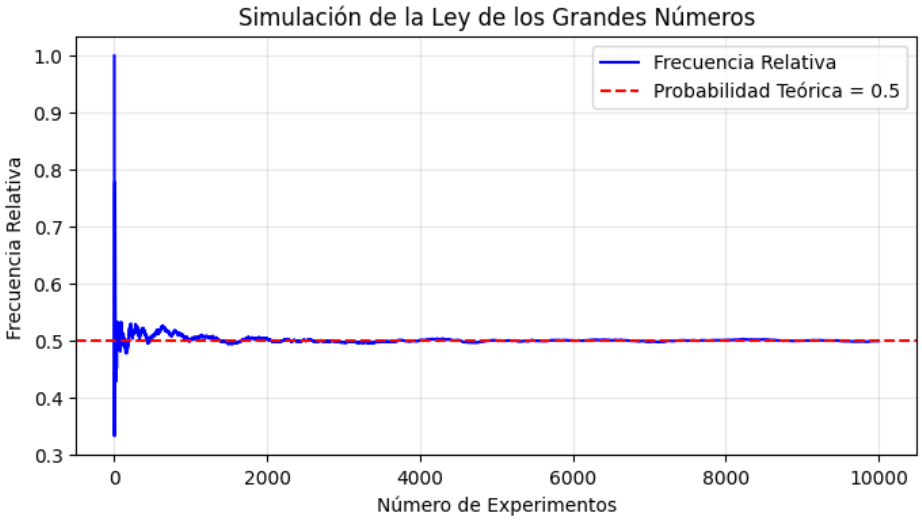

Figura 30: Ley de los grandes números

```
import random
import matplotlib.pyplot as plt

def simular_ley_grandes_numeros(probabilidad_evento, num_experimentos):
    """Simula la Ley de los Grandes Números con un evento
    de probabilidad dada."""

    exitos = 0  # Contador de veces que ocurre el éxito
    frecuencias_relativas = []
    for i in range(1, num_experimentos + 1):
        if random.random() < probabilidad_evento:
            exitos += 1  # Suma 1 si ocurre el evento
        frecuencia_relativa = exitos / i
        frecuencias_relativas.append(frecuencia_relativa)

    # Gráfica de la evolución de la frecuencia relativa
    plt.figure(figsize=(10, 5))
    plt.plot(range(1, num_experimentos + 1), frecuencias_relativas, \
        label="Frecuencia Relativa", color='blue')
    plt.axhline(y=probabilidad_evento, color='red', linestyle='dashed', \
        label=f"Probabilidad Teórica = {probabilidad_evento}")
    plt.xlabel("Número de Experimentos")
```

```
    plt.ylabel("Frecuencia Relativa")
    plt.title("Simulación de la Ley de los Grandes Números")
    plt.legend()
    plt.show()

    return frecuencias_relativas[-1]     # última frecuencia calculada

# Parámetros de la simulación
prob_evento = 0.5                        # Probabilidad de obtener "Cara"
num_lanzamientos = 10000                 # Cantidad de lanzamientos de la moneda

# Ejecutar la simulación
frecuencia_final = simular_ley_grandes_numeros(prob_evento, num_lanzamientos)
print(f"Frecuencia relativa final después de {num_lanzamientos} \
     experimentos: {frecuencia_final:.4f}")
```

Ejercicio resuelto: *Teorema de Bayes*. Crear un programa que calcule la probabilidad de que una planta esté realmente mente enferma sabiendo que la sensibilidad del test con el que se hizo haya dado positivo. El test de detección no es perfecto y se quiere calcular la probabilidad real de la planta. El programa debe pedir las entradas:

- $P(E) \rightarrow$ Probabilidad de que una planta esté enferma (prevalencia).
- $P(T+|E) \rightarrow$ Sensibilidad del test (probabilidad de que el test dé positivo si la planta está enferma).
- $P(T+|\overline{E}) \rightarrow$ Tasa de falsos positivos (probabilidad de que el test dé positivo si la planta NO está enferma).

```
def leer_probabilidad(mensaje):
    """
    Función para leer una probabilidad entre 0 y 1 con validación.
    Parámetro:
        - mensaje: Texto a mostrar en la entrada del usuario.
    Retorna:
        - Un valor float entre 0 y 1.
    """
    while True:
        try:
            valor = float(input(mensaje))
            if 0 <= valor <= 1:
                return valor
            else:
                print("\tError: La probabilidad debe estar entre 0 y 1.")
        except ValueError:
            print("\tError: Ingrese un valor numérico válido.")

def teorema_bayes(prob_enfermedad, sensibilidad, falso_positivo):
    """
    Calcula la probabilidad de que una planta esté realmente enferma dado que
    el test ha dado positivo.
    Parámetros:
        - prob_enfermedad: P(E) → Probabilidad de que una planta esté enferma.
        - sensibilidad: P( Tp | E) → Probabilidad de que el test sea positivo
    si la planta está enferma.
        - falso_positivo: P( Tp | nE) → Probabilidad de que el test sea positivo
    si la planta NO está enferma.
    Retorna:
      - P(E | Tp) → Probabilidad de que la planta esté enferma dado un test positivo
    """
    prob_no_enfermedad = 1 - prob_enfermedad  # P(nE)

    # Aplicación la fórmula de Bayes
    prob_enfermedad_dado_positivo = (sensibilidad * prob_enfermedad) / (
        (sensibilidad * prob_enfermedad) + (falso_positivo * prob_no_enfermedad)
    )
    return prob_enfermedad_dado_positivo
```

```
# Permitir al usuario ingresar valores personalizados:
while True:
    print("\n Simulación:")

    # Uso la nueva función para recoger las probabilidades
    p_enfermedad = leer_probabilidad("Introduzca la prevalencia de la enfermedad (entre 0 y 1): ")
    p_sensibilidad = leer_probabilidad("Introduzca la sensibilidad del test (entre 0 y 1): ")
    p_falso_positivo = leer_probabilidad("Introduzca tasa de falsos positivos (entre 0 y 1): ")

    # Calcular y mostrar el resultado
    probabilidad_real = teorema_bayes(p_enfermedad, p_sensibilidad, p_falso_positivo)
    print(f" Probabilidad  planta  enferma con test es positivo: {probabilidad_real:.4f}")

    # Opción para salir del bucle
    salir = input("¿Desea realizar otra simulación? (s/n): ").strip().lower()
    if salir != 's':
        break
```

Ejercicio resuelto: Desarrolla un programa en Python que permita clasificar variables en función de su tipo (cuantitativa o categórica), su escala de medida (nominal, ordinal, intervalo o razón) y su dependencia (dependiente o independiente). El programa funcionará como un juego interactivo en el que el usuario deberá responder correctamente a preguntas sobre diferentes variables relacionadas con la agronomía.

El programa trabajará con variables tales como:

- **Temperatura del suelo en grados Celsius** (cuantitativa, independiente)
- **Rendimiento de un cultivo en kg/ha** (cuantitativa, dependiente)
- **Tipo de fertilizante usado (A, B, C)** (cualitativa, independiente)
- **Calidad del suelo (buena, media, mala)** (cualitativa, independiente)
- **Nivel de pH del suelo** (cuantitativa, independiente)
- **Tiempo de germinación de las semillas (días)** (cuantitativa, dependiente)

Entre las reglas de juego se seleccionaran tres variables aleatorias, se formularán tres preguntas por variable, el usuario recibirá indicación de si es correcta o no su respuesta, calculará la puntuación total siendo:

- 9 puntos: ¡Excelente! Eres un experto en clasificación de variables.
- 6-8 puntos: ¡Muy bien! pero puedes mejorar.
- Menos de 6 puntos: sigue practicando para mejorar tu conocimiento en estadística.

```
# Definición de las variables y sus clasificaciones correctas
variables = [
    ("Temperatura del suelo en grados Celsius", "cuantitativa", "independiente"),
    ("Rendimiento de un cultivo en kg/ha", "cuantitativa", "dependiente"),
    ("Tipo de fertilizante usado (A, B, C)", "cualitativa", "independiente"),
    ("Calidad del suelo (buena, media, mala)", "cualitativa","independiente"),
    ("Nivel de pH del suelo", "cuantitativa", "independiente"),
    ("Tiempo de germinación de las semillas (días)", "cuantitativa", "dependiente")
]

def hacer_pregunta(variable, tipo_correcto, dependencia_correcta):
    """Formula preguntas al usuario y verifica las respuestas."""
    puntos = 0

    print(f"\nVariable: {variable}")

    tipo = input("¿Es cuantitativa o cualitativa? ").strip().lower()
    if tipo == tipo_correcto:
```

```
            print("\t¡Correcto!")
            puntos += 1
        else:
            print(f"\t¡Incorrecto! Es {tipo_correcto}.")

        dependencia = input("¿Es dependiente o independiente? ").strip().lower()
        if dependencia == dependencia_correcta:
            print("\t¡Correcto!")
            puntos += 1
        else:
            print(f"\t¡Incorrecto! Es {dependencia_correcta}.")

        return puntos

def jugar():
    """Función principal del juego."""
    print("\n\tBienvenido al juego 'Clasifica la Variable'")
    print("Responde correctamente sobre el tipo y la dependencia de cada variable.")

    # Seleccionamos 3 variables aleatorias
    seleccionadas = random.sample(variables, 3)
    puntuacion = 0

    for variable, tipo, dependencia in seleccionadas:
        puntuacion += hacer_pregunta(variable, tipo, dependencia)

    print(f"\nHas obtenido {puntuacion} puntos de 6 posibles.")

    # Evaluación de desempeño
    if puntuacion == 6:
        print("¡Excelente! Eres un experto en clasificación de variables.")
    elif puntuacion >= 4:
        print("¡Muy bien! ¡Pero aún puedes mejorar!")
    else:
        print("¡Sigue practicando para mejorar tu conocimiento sobre estadística!")

# Ejecución del juego
jugar()
```

Distribuciones de probabilidad

Ejercicio resuelto: Se desea contrastar los efectos de los tamaños de los parámetros n y p en la forma de representación de una distribución Binomial $B(n, p)$:

1. Para *p* pequeña y *n* pequeña, la distribución binomial es sesgada hacia la derecha.
2. Para *p* grande y *n* pequeña, la distribución binomial es sesgada hacia la izquierda.
3. Para *p*=0,5 y *n* grande y pequeña, la distribución binomial e simétrica.
4. Para *p* pequeña y *n* grande, la distribución binomial se acerca a la simetría.
5. A medida que `n` crece la distribución binomial se aproxima a una normal.

El programa que permite visualizar estas características en su forma puede ser el siguiente:

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

def funcion_masa_binomial(n, probabilidad, x):
    """Calcula la función de masa de probabilidad de la
    distribución binomial."""
    coeficiente = math.comb(n, x)  # Coeficiente binomial
    return coeficiente * (probabilidad ** x) * ((1 - probabilidad) ** (n - x))
```

```
def graficar_binomial_y_muestras(lista_n, lista_p, tamano=250):
    max_x = max(lista_n)
    valores_x = np.arange(0, max_x + 1, 1)
    plt.figure(figsize=(8, 6))

    for i in range(len(lista_n)):
        n = lista_n[i]
        p = lista_p[i]
        pmf = [funcion_masa_binomial(n, p, k) for k in valores_x]
        plt.plot(valores_x, pmf, marker='o', linestyle='-', label=f'n={n}, p={p}')

        muestra = np.random.binomial(n=n, p=p, size=tamano)
        plt.hist(muestra, bins=20, density=True, alpha=0.3, edgecolor='k')

    plt.legend()
    plt.title("Distribución Binomial y Muestras Simuladas")
    plt.xlabel("Número de éxitos")
    plt.ylabel("Probabilidad / Frecuencia relativa")
    plt.grid()
    plt.show()
    return None

def main():
    while True:
        try:
            cantidad_valores=int(input("¿Cuántas distribuciones Binomiales deseas introducir? "))
            if cantidad_valores <= 0:
                print("El número debe ser mayor que 0.")
                continue
        except ValueError:
            print("Por favor, introduce un número válido.")
            continue

        lista_n = []
        lista_p = []

        for i in range(cantidad_valores):
            try:
                n = int(input(f"Introduce el valor de n[{i+1}]: "))
                p = float(input(f"Introduce el valor de p[{i+1}]: "))
                if not (0 <= p <= 1):
                    print("El valor de p debe estar entre 0 y 1.")
                    continue
                lista_n.append(n)
                lista_p.append(p)

            except ValueError:
                print("Entrada no válida. Inténtalo de nuevo.")
                continue

        graficar_binomial_y_muestras(lista_n, lista_p)

       # Preguntar al usuario si desea repetir la ejecución
        repetir = input("¿Deseas realizar otra representación gráfica? (s/n): ").strip().lower()
        if repetir != 's':
            break  # Salir del bucle principal si la respuesta no es 's'

if __name__ == "__main__":
    main()
```

La comparación del histograma de frecuencias permite observar que, a medida que aumenta el tamaño de la muestra, la distribución empírica se va acercando cada vez más a la distribución teórica, lo que refleja el comportamiento descrito por la Ley de los Grandes Números. A medida que el número de observaciones crece, las frecuencias relativas tienden a estabilizarse y coincidir con las probabilidades teóricas. Además, una forma de verificar esta aproximación es superponer una curva normal sobre la distribución binomial. Esto permite visualizar cómo la distribución binomial se ajusta a una distribución normal con media $\mu = n \cdot p$ y varianza $\sigma^2 = n \cdot p \cdot q$.

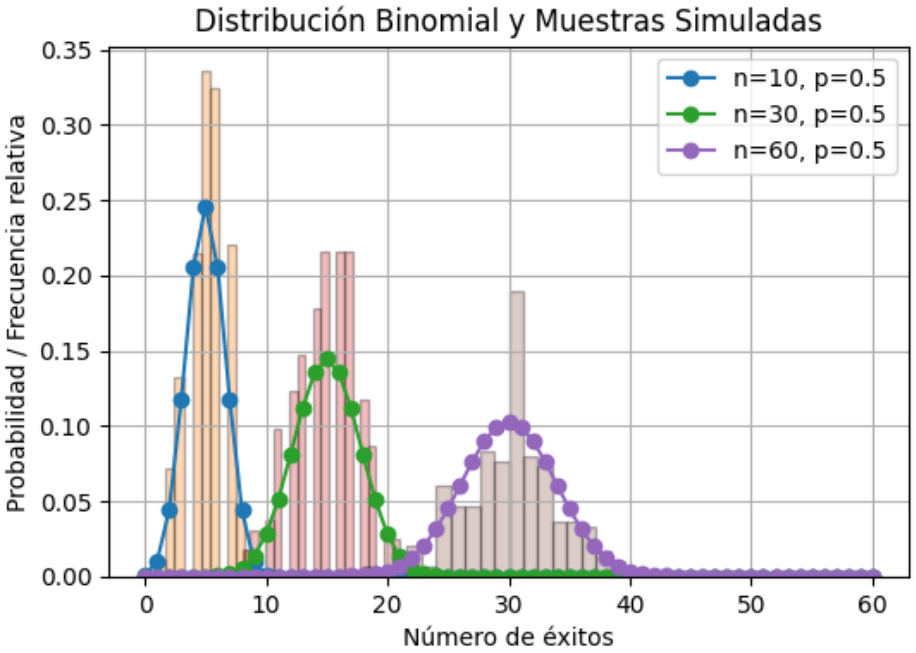

Figura 31: Distribución Binomial y muestras simuladas

Ejercicio resuelto: Construir un programa en Python para visualizar la **distribución de Bernoulli**, que modela experimentos con dos posibles resultados: éxito (**1**) o fracaso (**0**). Se trata de implementar la **función de masa de probabilidad (FMP)** de la distribución de Bernoulli, simular muestras aleatorias siguiendo una distribución de Bernoulli, representar gráficamente la distribución teórica junto con los datos simulados y permitir al usuario introducir distintos valores de p (probabilidad de éxito) y el **tamaño de la muestra** para observar cómo afectan a la distribución. Para ello, debe seguir las siguientes instrucciones:

1. El programa y proporcionar un valor de p (probabilidad de éxito) entre 0 y 1.
2. Indica el número de muestras que se quieren generar.
3. Observa la representación gráfica de la distribución teórica y de los datos simulados.
4. Decide si se quiere probar con otros valores o finalizar la ejecución.

Después de ejecutar el programa varias veces con distintos valores de p, ¿qué se observa en la forma de la distribución cuando p es cercano a 0 o a 1?

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

def funcion_masa_bernoulli(probabilidad, x):
    # Función de masa de probabilidad de la distribución de Bernoulli.
    if x in [0, 1]:
        return (probabilidad ** x) * ((1 - probabilidad) ** (1 - x))
    return 0

def mostrar_bernoulli(probabilidad, tamano=250):
    valores_x = [0, 1]
```

```
    plt.figure(figsize=(6, 5))

    pmf_bernoulli = [funcion_masa_bernoulli(probabilidad, k) for k in valores_x]
    plt.bar(valores_x, pmf_bernoulli, alpha=0.6, color='b', label='Bernoulli')

    muestra_bernoulli = np.random.binomial(n=1, p=probabilidad, size=tamano)
    plt.hist(muestra_bernoulli, bins=[-0.5, 0.5, 1.5], density=True, \
        alpha=0.3, color='b', edgecolor='k')
    plt.xticks([0, 1])
    plt.title("Distribución de Bernoulli")
    plt.xlabel("Éxito (1) / Fracaso (0)")
    plt.ylabel("Probabilidad / Frecuencia relativa")
    plt.legend()
    plt.grid()
    plt.show()

def main():
    while True:
        try:
            p = float(input("Introduce la probabilidad de éxito p (0 < p <= 1): "))
            if not (0 < p <= 1):
                print("El valor de p debe estar entre 0 y 1.")
                continue
            tamano = int(input("Introduce el número de muestras a generar: "))
            if tamano <= 0:
                print("El número de muestras debe ser mayor que 0.")
                continue
        except ValueError:
            print("Entrada no válida. Inténtalo de nuevo.")
            continue

        mostrar_bernoulli(p, tamano)

        cambiar_parametros = input("¿Quieres cambiar los valores de p y el tamaño de la \
            muestra? (s/n): ").strip().lower()
        if cambiar_parametros != 's':
            break
if __name__ == "__main__":
    main()
```

Ley de los sucesos raros o ley de las probabilidades pequeñas. (Aproximación de la Binomial a la Poisson)

Ejercicio resuelto. La distribución de Poisson se aplica en situaciones en que se tiene una distribución binomial con p muy pequeño $(p \to 0)$ y n grande $(n \to \infty)$, y el producto $n \cdot p$ se mantiene constante igual a $\lambda \quad (n\mathring{u}p \to \lambda)$.

La distribución de probabilidad se obtiene tomando el límite cuando n tiende a infinito en la distribución de probabilidad de la binomial, es decir:

$$p[X = k] = \lim_{n\to\infty} \binom{n}{k} p^k (1-p)^{n-k} = \frac{e^{-\lambda}\lambda^k}{k!}$$

Utilizar una distribución de Poisson en lugar de una Binomial (cuando se puede) es doble:

- La Binomial $B(n, p)$ depende de dos parámetros mientras que la Poisson $P(\lambda)$ sólo de uno.
- El cálculo de la función de probabilidad de la Poisson es más fácil de calcular que una binomial.

Implementar un programa que permita simular la aproximación anterior.

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

def validar_datos(num_ensayos, prob_exito):
    """Valida que los datos de entrada sean apropiados para la simulación."""
    if num_ensayos <= 0:
        raise ValueError("El número de ensayos debe ser un entero positivo.")
    if not (0 <= prob_exito <= 1):
        raise ValueError("La probabilidad de éxito debe estar entre 0 y 1.")
    if num_ensayos < 20 or prob_exito >= 0.05:
        print("Advertencia: Para una mejor aproximación, se recomienda n >= 20 y p < 0.05.")
    if num_ensayos * prob_exito > 10:
        print("Advertencia: Para n p grandes, la aproximación puede ser menos precisa.")
    return True

def simular_aproximacion_poisson(num_ensayos, prob_exito, num_simulaciones=10000):
    """Simula la aproximación de la distribución binomial a la de Poisson."""

    lambda_poisson = num_ensayos * prob_exito  # Parámetro lambda de la Poisson

    # Simulaciones
    muestras_binomial = np.random.binomial(num_ensayos, prob_exito, size=num_simulaciones)
    muestras_poisson = np.random.poisson(lambda_poisson, size=num_simulaciones)

    # Histograma
    plt.figure(figsize=(10, 6))
    bins = np.arange(min(muestras_binomial.min(), muestras_poisson.min()),
                     max(muestras_binomial.max(), muestras_poisson.max()) + 1.5) - 0.5

    plt.hist(muestras_binomial, bins=bins, alpha=0.6,color='b',label=f"Binomial (n={num_ensayos}, \
        p={prob_exito})", density=True)
    plt.hist(muestras_poisson, bins=bins, alpha=0.6, color='r', \
        label=f"Poisson (lambda={lambda_poisson})", density=True, histtype='step',\
         linewidth=2)

    # Distribución teórica de Poisson
    x = np.arange(0, max(muestras_binomial.max(), muestras_poisson.max()) + 1)
    poisson_pmf = stats.poisson.pmf(x, lambda_poisson)
    plt.plot(x, poisson_pmf, 'ro-', label="Poisson teórica")

    # Configuración del gráfico
    plt.xlabel("Número de éxitos")
    plt.ylabel("Frecuencia relativa")
    plt.title("Aproximación de la distribución binomial a una Poisson")
    plt.legend()
    plt.grid()
    plt.show()

def main():
    """Función principal para ejecutar la simulación."""
    print("-"*75)
    print("Recuerde que para que la aproximación sea válida: n>=20 y p<0.05")
    print("y que n.p permanece constante.")
    print("-"*75)

    while True:
        try:
            num_ensayos = int(input("Introduzca el número de ensayos (n): "))
            prob_exito = float(input("Introduzca la probabilidad de éxito (p): "))

            if validar_dato entre raiz tamaño muests(num_ensayos, prob_exito):
                simular_aproximacion_poisson(num_ensayos, prob_exito)
                break  # Salir del bucle si los datos son válidos
        except ValueError as e:
            print(f"Error: {e}")

if __name__ == "__main__":
    main()
```

Ejercicio resuelto: Construir un programa que represente la función de densidad de la distribución

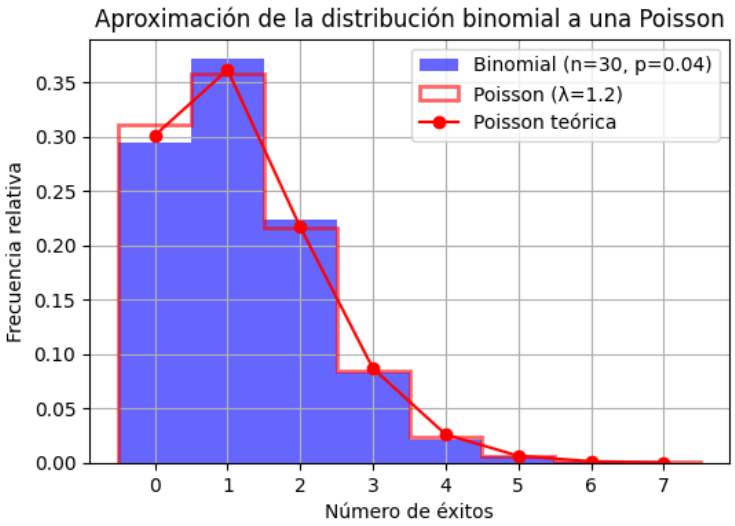

Figura 32: Aproximación de la distribución binomial a la Poisson

normal por definición, y represente dos gráficas, una en la que se observen varias normales con la misma media y distintas varianzas, y otro gráfico donde se visualicen distribuciones normales con distinta media y varianzas iguales.

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

def densidad_normal(x, media, desviacion):
    """Calcula la densidad de probabilidad de una distribución normal manualmente."""
    return (1 / (desviacion * np.sqrt(2 * np.pi))) * \
        np.exp(-((x - media) ** 2) / (2 * desviacion ** 2))

x = np.linspace(-10, 10, 400)

# Crear la figura y los subplots
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# 1. Variando la desviación típica (misma media)
med_fija = 0
desviaciones = [0.5, 1, 2, 3]

for sigma in desviaciones:
    axes[0].plot(x, densidad_normal(x, med_fija, sigma), label=f'$\sigma$={sigma}')

axes[0].set_title("Misma media, distintas desviaciones")
axes[0].set_xlabel("X")
axes[0].set_ylabel("Densidad de probabilidad")
axes[0].legend()
axes[0].grid()

# 2. Variando la media (misma desviación típica)
desviacion_fija = 1
medias = [-3, 0, 3, 6]

for mu in medias:
    axes[1].plot(x, densidad_normal(x, mu, desviacion_fija), label=f'$\mu$={mu}')

axes[1].set_title("Misma desviación, distintas medias")
axes[1].set_xlabel("X")
axes[1].set_ylabel("Densidad de probabilidad")
axes[1].legend()
axes[1].grid()
```

```
# Ajustar espacio y mostrar
plt.tight_layout()
plt.show()
```

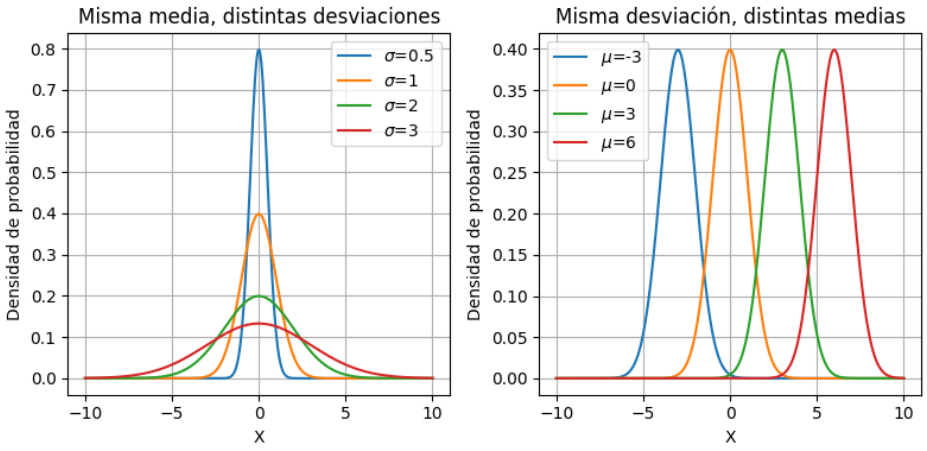

Figura 33: Distribuciones normales.

Ejercicio resuelto: El cálculo de probabilidades de una distribución normal de colas a la izquierda es una de las herramientas clave es el cálculo de probabilidades (determina la probabilidad de que una variable aleatoria tome un valor menor o igual a un valor dado). El diseño del cálculo de esta probabilidad se va a realizar desde dos enfoques: cálculo manual utilizando la fórmula del trapecio para la integración numérica y el cálculo usando la librearía SciPy. El programa debe recibir como entrada el valor numérico que representa el punto en el que se quiere calcular la probabilidad. La función de densidad de la curva normal $Z(0,1)$ es:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

NOTA: La función `stats.norm.cdf` forma parte del módulo `scipy.stats` en Python y se utiliza para calcular la función de distribución acumulativa (CDF, por sus siglas en inglés) de una distribución normal $P(X <= x)$. Su uso es de la siguiente manera:

- `stats.norm.cdf(x, loc=0, scale=1)`:
- `x`: El valor en el que se desea evaluar la CDF.
- `loc`: La media de la distribución normal (por defecto es 0).
- `scale`: La desviación estándar de la distribución normal (por defecto es 1).

```
import numpy as np

def probabilidad_cola_izquierda_manual(z):
    """
    Calcula la probabilidad con cola a la izquierda para una distribución
    normal estándar  utilizando una aproximación numérica.
    Argumentos:
        z: El valor en el que se quiere calcular la probabilidad.
```

```
    Retorno:
        La probabilidad con cola a la izquierda.
    """
    # Aproximación de la integral utilizando la regla del trapecio
    # 1. Definir el número de segmentos (trapecios) para la aproximación
    num_segmentos = 1000

    # 2. Crear los puntos de integración (x) en el intervalo [-5, z]
    # np.linspace crea un array de num_segmentos + 1 puntos espaciados
    #  uniformemente
    x = np.linspace(-5, z, num_segmentos + 1)  # Rango de integración

    # 3. Calcular el ancho de cada segmento (h en la fórmula del trapecio)
    dx = x[1] - x[0]

    # 4. Definir la función de densidad de probabilidad (f(x)) de la
    # distribución normal estándar
    def f(x):
        return (1 / np.sqrt(2 * np.pi)) * np.exp(-x**2 / 2)

    # 5. Calcular la integral usando la regla del trapecio:
    #    - f(x[0]) y f(x[-1]) son los valores de la función en los extremos
    #           del intervalo
    #    - f(x[1:-1]) son los valores de la función en los puntos intermedios
    #    - np.sum suma todos los valores en el array f(x[1:-1])
    #    - La fórmula completa se aplica: (h/2) * [f(a) + 2f(x1)
    #           + 2f(x2) + ... + 2f(xn-1) + f(b)]
    integral = 0.5 * (f(x[0]) + f(x[-1])) + np.sum(f(x[1:-1]))
    integral *= dx  # Multiplicar por el ancho del segmento (dx)

    return integral # Retornar el valor de la integral aproximada

def probabilidad_cola_izquierda_scipy(z):
    """
    Calcula la probabilidad con cola a la izquierda para una distribución
    normal estándar  utilizando la función cdf de SciPy.
    Argumentos:
        z: El valor en el que se quiere calcular la probabilidad.
    Retorno:
        La probabilidad con cola a la izquierda.
    """
    from scipy.stats import norm  # norm aquí para minimizar dependencias
    return norm.cdf(z)

# Ejemplo de uso
z = 1.96  # Valor de ejemplo

p_manual = probabilidad_cola_izquierda_manual(z)
p_scipy = probabilidad_cola_izquierda_scipy(z)

print(f"Probabilidad (Cálculo manual): {p_manual}")
print(f"Probabilidad (SciPy): {p_scipy}")
```

Ejercicio resuelto: **Calculadora de probabilidades de una distribución Normal.** Crear un programa que calcule la probabilidad de una variable aleatoria que sigue una distribución normal, dado su media, desviación estándar y el tipo de probabilidad que se desea calcular:

1. Solicitará al usuario la media y la desviación estándar de la distribución normal.
2. Presentará un menú con las siguientes opciones de cálculo de probabilidad:
   - $P(X <= a)$
   - $P(a <= X < b)$
   - $P(|X| <= a)$
3. Solicitará al usuario los valores de 'a' y 'b'.
   - Si el tipo de probabilidad es $P(a <= X <= b)$, se validará que 'b' sea mayor que 'a'.

Si no, se mostrará un mensaje de error y volver a solicitar el valor de 'b'.

4. Calculará la probabilidad utilizando la función `stats.norm.cdf` del módulo `scipy.stats`.
5. Mostrará el resultado de la probabilidad calculada al usuario.
6. Preguntará al usuario si desea realizar otro cálculo.

```
import scipy.stats as stats
import numpy as np

def calcular_probabilidad(media, desviacion_tipica, tipo_probabilidad, a, b=None):
    """Calcula la probabilidad de una distribución normal.
    Argumentos:
        media: La media de la distribución.
        desviacion_tipica: La desviación típica de la distribución.
        tipo_probabilidad: El tipo de probabilidad a calcular:
            1: P(X <= a)
            2: P(a <= X < b)
            3: P(|X| <= a)
        a: El límite inferior o el valor absoluto.
        b: El límite superior (solo para tipo_probabilidad 2).
    Retorno:
        La probabilidad calculada.    """

    if tipo_probabilidad == 1:                          # P(X <= a)
        probabilidad = stats.norm.cdf(a, loc=media, scale=desviacion_tipica)
    elif tipo_probabilidad == 2:  # P(a <= X <= b)
        probabilidad = stats.norm.cdf(b, loc=media, scale=desviacion_tipica) \
            - stats.norm.cdf(a, loc=media, scale=desviacion_tipica)
    elif tipo_probabilidad == 3:                        # P(|X| <= a)
        probabilidad = stats.norm.cdf(a, loc=media, scale=desviacion_tipica) \
            - stats.norm.cdf(-a, loc=media, scale=desviacion_tipica)
    else:
        raise ValueError("Tipo de probabilidad inválido.")
    return probabilidad

while True:
    # Solicitar datos al usuario
    media = float(input("Introduce la media de la distribución: "))
    desviacion_tipica = float(input("Introduce la desviación típica \
        de la distribución: "))

    # Mostrar menú de opciones
    print("\nTipos de probabilidad:")
    print("1. P(X <= a)")
    print("2. P(a <= X <= b)")
    print("3. P(|X| <= a)")
    tipo_probabilidad = int(input("Selecciona el tipo de  probabilidad (1, 2 o 3): "))

    # Solicitar límites según el tipo de probabilidad
    a = float(input("Introduce el valor de a: "))
    if tipo_probabilidad == 2:  # Solamente si 'b' es tipo_probabilidad 2
        while True:
            b = float(input("Introduce el valor de b (debe ser mayor que a): "))
            if b > a:
                break
            else:
                print("Error: b debe ser mayor que a. Inténtalo de nuevo.")

    # Calcular y mostrar la probabilidad
    if tipo_probabilidad == 2:
        probabilidad = calcular_probabilidad(media, desviacion_tipica, \
            tipo_probabilidad, a, b)
    else:
        probabilidad = calcular_probabilidad(media, desviacion_tipica, \
            tipo_probabilidad, a)
    print(f"\nLa probabilidad es: {probabilidad}")

    continuar = input("¿Deseas realizar otro cálculo? (s/n): ")
    if continuar.lower() != "s":
        break
```

Teorema Central del Límite

Crear un programa que permita comprobar empíricamente que si una variable aleatoria X con media μ y desviación estándar σ $[X \sim \mathcal{N}(\mu, \sigma)]$, la distribución de la media muestral $\overline{X}$ de muestras de tamaño n sigue una distribución normal: $\overline{X} \sim \mathcal{N}\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$.

Este resultado es fundamental en estadística, ya que permite aproximar la distribución de la media muestral a una normal, incluso cuando la distribución original de X no es normal, siempre que el tamaño de la muestra sea suficientemente grande.

Para la realización de este ejercicio se crea a partir de una población dada, todas las muestras de tamaño inferior a ésta, de manera que se irá calculando las medias y de las desviaciones típicas de la distribución de medias para cada uno de los tamaños de la muestra. Finalmente, se contrasta su valor con la media de la población y con la desviación típica para hallar la relación pedida.

```
import itertools
import numpy as np
import pandas as pd

def calcular_distribucion_de_medias(poblacion, tamanos_muestra):
    """
    Calcula la media y desviación típica de las muestras con reemplazamiento
    para diferentes tamaños de muestra de una población.
    Argumentos:
        poblacion: Lista que representa la población.
        tamanos_muestra: Lista de enteros que indica los tamaños de muestra a considerar.
    Retorno:
        Un diccionario donde las claves son los tamaños de muestra y los valores
        son DataFrames con las estadísticas de las medias muestrales."""

    print("Comprobación de que la distribución de medias muestras de una N(mu,sigma)")
    print("sigue una distribución N(mu, sigma/n^(1/2))")
    resultados = {}

    for n in tamanos_muestra:
        muestras = list(itertools.product(poblacion, repeat=n))
        medias = [np.mean(muestra) for muestra in muestras]
        desviaciones = [np.std(muestra, ddof=0) for muestra in muestras]
        media_de_medias = np.mean(medias)
        desviacion_tipica_de_medias = np.std(medias)
        data = {
            'Muestras': muestras,
            'Media': medias,
            'Desviación Típica': desviaciones,
        }
        df = pd.DataFrame(data)
        # Almacenar datos de la muestra en el diccionario con key tamaño de la muestra
        resultados[n] = df

        print("-" * 75)
        print(f"Resultados para muestras de tamaño {n}:")
        print(df.head())
        print(df.tail())
        print("\n\tMedia de la distribución de Medias:", media_de_medias)
        print("\tDesviación Típica de la distribución de Medias:", desviacion_tipica_de_medias)
        print(f"\n--- Comparación con la población ---")
        print(f"\tMedia de la población: {np.mean(poblacion):.2f}")
        print(f"\tDesv. típica población entre raiz tamaño muestra {n}: \
            {np.std(poblacion)/np.sqrt(n)}\n")
    return resultados

# Ejemplo de uso:
poblacion = [5, 6, 7, 8]
tamanos_muestra = [2, 3, 4]
resultados = calcular_distribucion_de_medias(poblacion, tamanos_muestra)
```

En este ejercicio se han combinado distintas estructuras de datos y librerías para conseguir comprobar empíricamente la relación. Este programa permite modificar la población así como los distintos tamaños muestrales. La salida del programa permite comprobar la relación indicada.

```
Comprobación de que la distribución de medias muestras de una N(mu,sigma)
sigue una distribución N(mu, sigma/n^(1/2))
------------------------------------------------------------------------------
Resultados para muestras de tamaño 2:
  Muestras  Media  Desviación Típica
0   (5, 5)    5.0                0.0
1   (5, 6)    5.5                0.5
2   (5, 7)    6.0                1.0
3   (5, 8)    6.5                1.5
4   (6, 5)    5.5                0.5
   Muestras  Media  Desviación Típica
11   (7, 8)    7.5                0.5
12   (8, 5)    6.5                1.5
13   (8, 6)    7.0                1.0
14   (8, 7)    7.5                0.5
15   (8, 8)    8.0                0.0

    Media de la distribución de Medias: 6.5
    Desviación Típica de la distribución de Medias: 0.7905694150420949

--- Comparación con la población ---
    Media de la población: 6.50
    Desviación típica de la población entre raiz tamaño muestra 2: 0.7905694150420948

------------------------------------------------------------------------------
Resultados para muestras de tamaño 3:
    Muestras     Media  Desviación Típica
0  (5, 5, 5)  5.000000           0.000000
1  (5, 5, 6)  5.333333           0.471405
2  (5, 5, 7)  5.666667           0.942809
3  (5, 5, 8)  6.000000           1.414214
4  (5, 6, 5)  5.333333           0.471405
     Muestras     Media  Desviación Típica
59  (8, 7, 8)  7.666667           0.471405
60  (8, 8, 5)  7.000000           1.414214
61  (8, 8, 6)  7.333333           0.942809
62  (8, 8, 7)  7.666667           0.471405
63  (8, 8, 8)  8.000000           0.000000

    Media de la distribución de Medias: 6.5
    Desviación Típica de la distribución de Medias: 0.6454972243679028

--- Comparación con la población ---
    Media de la población: 6.50
    Desviación típica de la población entre raiz tamaño muestra 3: 0.6454972243679029

------------------------------------------------------------------------------
Resultados para muestras de tamaño 4:
       Muestras  Media  Desviación Típica
0  (5, 5, 5, 5)   5.00           0.000000
1  (5, 5, 5, 6)   5.25           0.433013
2  (5, 5, 5, 7)   5.50           0.866025
3  (5, 5, 5, 8)   5.75           1.299038
4  (5, 5, 6, 5)   5.25           0.433013
         Muestras  Media  Desviación Típica
251  (8, 8, 7, 8)   7.75           0.433013
252  (8, 8, 8, 5)   7.25           1.299038
253  (8, 8, 8, 6)   7.50           0.866025
254  (8, 8, 8, 7)   7.75           0.433013
255  (8, 8, 8, 8)   8.00           0.000000

    Media de la distribución de Medias: 6.5
    Desviación Típica de la distribución de Medias: 0.5590169943749475

--- Comparación con la población ---
    Media de la población: 6.50
    Desviación típica de la población entre raiz tamaño muestra 4: 0.5590169943749475
```

Teorema de Moivre-Laplace

Una variable aleatoria X con distribución binomial con parámetros n (número de ensayos) y p (probabilidad de éxito). Entonces, para n suficientemente grande, la distribución de X se puede aproximar mediante una distribución normal $N\left(\mu = np, \sigma = \sqrt{npq}\right)$.

La variable estandarizada es:

$Z = \frac{X - np}{\sqrt{np(1-p)}}$

Entonces, la distribución de Z se aproxima a una distribución normal estándar (con media 0 y desviación estándar 1) cuando $n \to \infty$.

Comprueba que para aceptar esta aproximación algunos de los criterios son:

- $n \cdot p \geq 5 \quad y \quad n \cdot q \geq 5.$
- $p < 0{,}1 \quad y \quad n \cdot p > 5$
- $p > 0{,}1 \quad y \quad n \cdot p < 5$
- $p \cong 0{,}5 \quad y \quad n \cdot p > 3$

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

def validar_datos(n, p):
    """Valida los datos de entrada y devuelve los criterios de aproximación."""
    if not isinstance(n, int) or n <= 0:
        raise ValueError("n debe ser un entero positivo.")
    if not (0 < p < 1):
        raise ValueError("p debe estar entre 0 y 1.")

    q = 1 - p
    criterio1 = n * p >= 5 and n * q >= 5  # Condición clásica
    criterio2 = p < 0.1 and n * p > 5  # Pequeño p pero np suficientemente grande
    criterio3 = p > 0.1 and n * p < 5  # Probabilidad no demasiado pequeña pero np pequeño
    criterio4 = abs(p - 0.5) < 0.1 and n * p > 3  # p cercano a 0.5 con np moderado

    return criterio1, criterio2, criterio3, criterio4

def simular_aproximacion_normal(n, p, num_simulaciones=10000):
    """Simula la aproximación de la binomial a la normal y grafica la comparación."""

    # Validar datos y obtener criterios
    criterio1, criterio2, criterio3, criterio4 = validar_datos(n, p)

    # Parámetros de la distribución binomial
    mu = n * p
    sigma = np.sqrt(n * p * (1 - p))

    # Generar muestras de la distribución binomial
    muestras_binomial = np.random.binomial(n, p, size=num_simulaciones)

    # Crear el histograma de las muestras binomiales
    plt.figure(figsize=(10, 6))
    plt.hist(muestras_binomial, bins='auto', density=True, \
                alpha=0.6, color='b', label=f"Binomial (n={n}, p={p})")

    # Superponer la distribución normal teórica con corrección de continuidad
    x = np.linspace(muestras_binomial.min(), muestras_binomial.max(), 100)
    pdf_normal = (1 / (sigma * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - mu) / sigma) ** 2)
    plt.plot(x, pdf_normal, 'r-', linewidth=2, label="Normal teórica")

    # Mostrar criterios en el título
    titulo = "Aproximación Binomial a Normal\n"
```

```
    titulo += f"np >= 5 y nq >= 5: {'SI' if criterio1 else 'NO'}\n"
    titulo += f"p < 0.1 y np > 5: {'SI' if criterio2 else 'NO'}\n"
    titulo += f"p > 0.1 y np < 5: {'SI' if criterio3 else 'NO'}\n"
    titulo += f"p aproximadamente 0.5 y np > 3: {'SI' if criterio4 else 'NO'}"

    plt.xlabel("Número de éxitos")
    plt.ylabel("Densidad de probabilidad")
    plt.title(titulo)
    plt.legend()
    plt.grid(True)
    plt.show()

def main():
    """Función principal para ejecutar la simulación."""
    while True:
        try:
            n = int(input("Introduce el número de ensayos (n): "))
            p = float(input("Introduce la probabilidad de éxito (p): "))

            simular_aproximacion_normal(n, p)
            break  # Salir del bucle si los datos son válidos
        except ValueError as e:
            print(f"Error: {e}")

if __name__ == "__main__":
    main()
```

Queda al lector comprobar distintos casos de verificación de la aproximación. Una posible prueba o caso de validación es el siguiente:

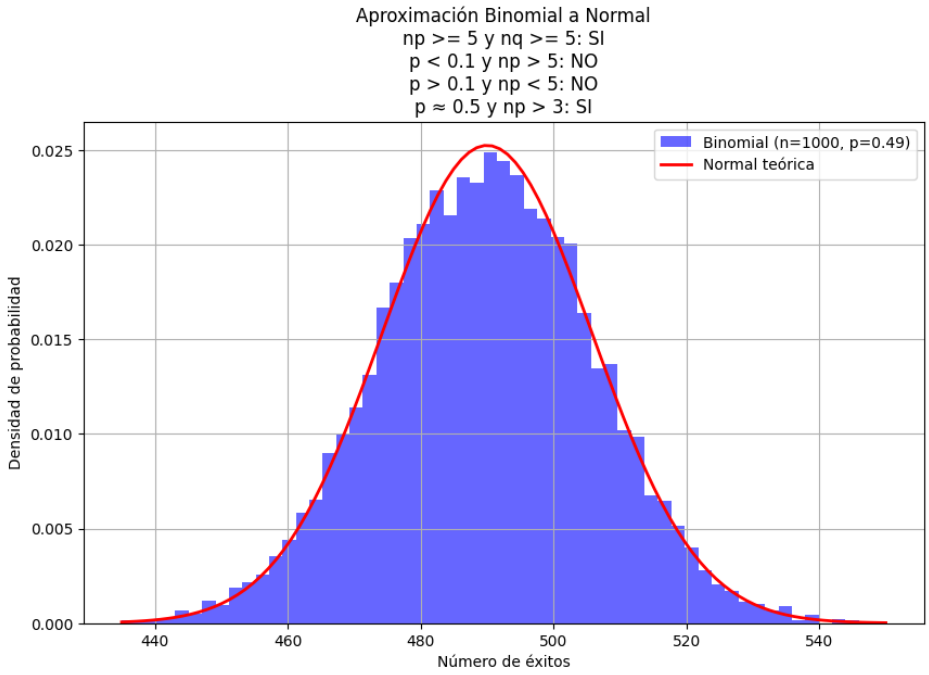

Figura 34: Aproximación de la distribución binomial a la normal

Intervalos de confianza y contraste de hipótesis

Ejercicio resuelto: Desarrollar un programa que calcule y visualice el intervalo de confianza para la media poblacional, asumiendo que la desviación estándar de la población es conocida.

IC para la media al nivel $(1-\alpha)$: $(\bar{x} - z_{\alpha/2}\frac{\sigma}{\sqrt{n}}, \bar{x} + z_{\alpha/2}\frac{\sigma}{\sqrt{n}})$

Los requisitos del programa son los siguientes:

1. El programa debe solicitar al usuario los siguientes datos:
   - Media muestral.
   - Desviación estándar de la población.
   - Tamaño de la muestra.
   - Nivel de confianza deseado (entre 0 y 1).
2. El programa debe calcular los límites inferior y superior del intervalo de confianza utilizando la fórmula apropiada.
3. El programa debe generar una gráfica que muestre:
   - La distribución normal estándar.
   - El intervalo de confianza, resaltado en la gráfica.
4. El programa debe mostrar en la consola el intervalo de confianza calculado.

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

def calcular_intervalo_confianza(media_muestral, desviacion_estandar, \
    tamaño_muestra, nivel_confianza):
    """
    Calcula el intervalo de confianza para la media poblacional
    con varianza conocida.
    Argumentos:
        media_muestral: La media de la muestra.
        desviacion_estandar: La desviación estándar de la población.
        tamaño_muestra: El tamaño de la muestra.
        nivel_confianza: El nivel de confianza deseado (entre 0 y 1).
    Retorno:
        Una tupla que contiene el límite inferior y el límite superior
        del intervalo de confianza.
    """

    # Calcular el valor crítico de Z
    alfa = 1 - nivel_confianza
    z_critico = stats.norm.ppf(1 - alfa / 2)

    # Calcular el margen de error
    margen_error = z_critico * (desviacion_estandar / np.sqrt(tamaño_muestra))

    # Calcular los límites del intervalo de confianza
    limite_inferior = media_muestral - margen_error
    limite_superior = media_muestral + margen_error

    return limite_inferior, limite_superior

def graficar_intervalo_confianza(media_muestral, desviacion_estandar, \
    limite_inferior, limite_superior):
    """
    Grafica la distribución normal y el intervalo de confianza.
    Argumentos:
        media_muestral: La media de la muestra.
        desviacion_estandar: La desviación estándar de la población.
        limite_inferior: El límite inferior del intervalo de confianza.
```

```
        limite_superior: El límite superior del intervalo de confianza.   """

    # Generar puntos para la distribución normal
    x = np.linspace(media_muestral - 4 * desviacion_estandar,
                    media_muestral + 4 * desviacion_estandar, 1000)
    y = stats.norm.pdf(x, loc=media_muestral, scale=desviacion_estandar)

    # Graficar la distribución normal
    plt.plot(x, y, label='Distribución Normal')

    # Rellenar el área del intervalo de confianza
    plt.fill_between(x, y, where=(x >= limite_inferior) & (x <= limite_superior),
                     color='lightblue', alpha=0.5, label='Intervalo de Confianza')

    # Agregar etiquetas y título
    plt.xlabel('Valores')
    plt.ylabel('Densidad de Probabilidad')
    plt.title('Intervalo de Confianza para la Media')
    plt.legend()
    plt.grid(True)
    plt.show()

# Solicitar datos al usuario
media_muestral = float(input("Introduzca la media muestral: "))
desviacion_estandar = float(input("Introduzca la desviación estándar de la población: "))
tamaño_muestra = int(input("Introduzca el tamaño de la muestra: "))
nivel_confianza = float(input("Introduzca el nivel de confianza (entre 0 y 1): "))

# Calcular el intervalo de confianza
limite_inferior, limite_superior = calcular_intervalo_confianza(
    media_muestral, desviacion_estandar, tamaño_muestra, nivel_confianza
)

# Mostrar el intervalo de confianza
print(f"Intervalo de confianza para la media: ({limite_inferior:.2f}, {limite_superior:.2f})")

# Graficar el intervalo de confianza
graficar_intervalo_confianza(media_muestral, desviacion_estandar, limite_inferior, limite_superior)
```

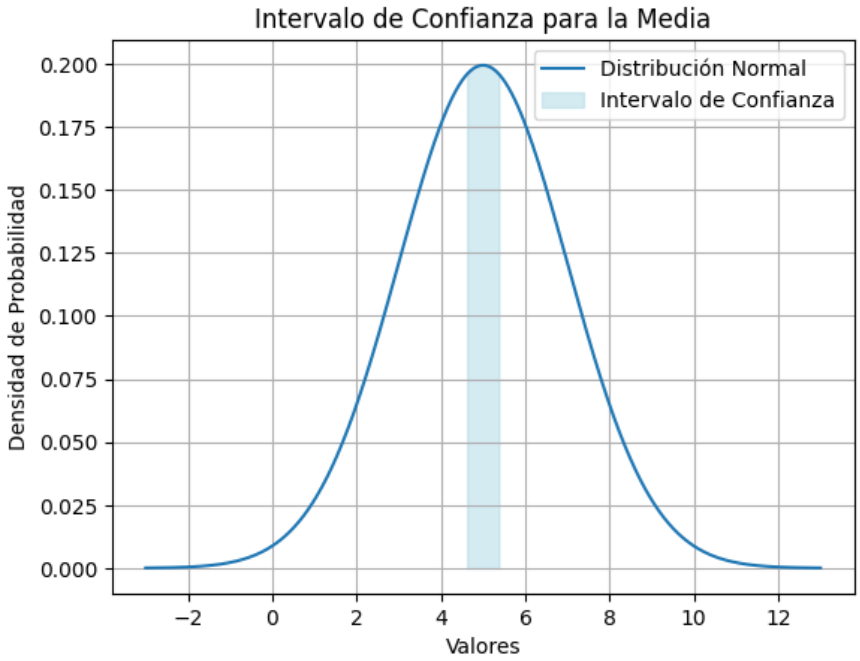

Figura 35: Intervalo de confianza

Ejercicio resuelto: Constraste de hipótesis para media con varianza conocida. Crear un programa en Python que realice un contraste de hipótesis para la media de una población con varianza conocida. El programa deberá:

1. Solicitar al usuario los siguientes datos:
   - Tamaño de la muestra (número entero).
   - Media muestral.
   - Desviación estándar de la población.
   - Media hipotética (μ_0) que se desea contrastar.
   - Nivel de significación ($alfa$) entre 0 y 1.
2. Permitir al usuario seleccionar el tipo de hipótesis alternativa:
   - Bilateral ($\mu \neq \mu_0$)
   - Unilateral derecha ($\mu > \mu_0$)
   - Unilateral izquierda ($\mu < \mu_0$)
3. Calcular el estadístico de prueba (Z).
4. Calcular el valor crítico y el p-valor según el tipo de hipótesis seleccionado.
5. Tomar la decisión de rechazar o aceptar la hipótesis nula (H0) basándose en la comparación del estadístico de prueba con el valor crítico, o del p-valor con el nivel de significación.
6. Mostrar los resultados del contraste de hipótesis, incluyendo:
   - Estadístico de prueba (Z).
   - Valor crítico.
   - P-valor.
   - Decisión (rechazar o aceptar H0).

Consideraciones:

- Utilizar las funciones `norm.ppf()` y `norm.cdf()` del módulo `scipy.stats` para calcular el valor crítico y el p-valor, respectivamente.
- Implementar un control de errores para asegurar que el tamaño de la muestra introducido por el usuario sea un número entero.
- Mostrar mensajes claros e informativos al usuario durante la ejecución del programa.
- Documentar el código con comentarios que expliquen la lógica del programa.

Caso de prueba:

Si el usuario introduce los siguientes datos:

- Tamaño de la muestra: 30
- Media muestral: 170
- Desviación estándar de la población: 10
- Media hipotética: 165
- Nivel de significación: 0.05
- Tipo de hipótesis alternativa: Unilateral derecha

El programa debería mostrar un resultado similar a este:

Resultados del contraste de hipótesis:

```
Estadístico de prueba (Z): 2.74
Valor crítico: 1.64
p-valor: 0.003
Rechaza la hipótesis nula (H0).
```

```
import numpy as np
import scipy.stats as stats

def contraste_hipotesis_media_varianza_conocida():
    """
    Realiza un contraste de hipótesis para la media de una población con varianza conocida.
    """

    # Solicitar datos al usuario
    while True:
        try:
            tam_muestra = int(input("Introduzca el tamaño de la muestra (debe ser un \
                número entero): "))
            break  # Sale del bucle si la entrada no es un entero
        except ValueError:
            print("Error: El tamaño de la muestra debe ser un número entero. Inténtalo de nuevo.")
    media_muestral = float(input("Introduzca la media muestral: "))
    desviacion_estandar = float(input("Introduzca la desviación estándar de la población: "))
    media_hipotetica = float(input("Introduzca la media hipotética que desea contrastar (mu_0): "))

    # Mostrar menú de opciones para el tipo de hipótesis alternativa
    print("\nTipos de hipótesis alternativa:")
    print("1. Bilateral (mu != mu0)")
    print("2. Unilateral derecha (mu > mu0)")
    print("3. Unilateral izquierda (mu < mu0)")
    tipo_hipotesis = int(input("Seleccione el tipo de hipótesis alternativa (1, 2 o 3): "))

    # Solicitar nivel de significacion
    nivel_significacion = float(input("Introduzca el nivel de significación entre 0 y 1 (alfa): "))

    # Calcular el estadístico de prueba
    estadistico_z=(media_muestral-media_hipotetica)/ (desviacion_estandar/np.sqrt(tam_muestra))

    # Calcular el valor crítico o p-valor según el tipo de hipótesis
    if tipo_hipotesis == 1:  # Bilateral
        valor_critico = stats.norm.ppf(1 - nivel_significacion / 2)
        p_valor = 2 * (1 - stats.norm.cdf(abs(estadistico_z)))
        # Se multiplica por 2 para considerar ambas colas
        decision = "Rechaza" if abs(estadistico_z) > valor_critico else "Acepta"
    elif tipo_hipotesis == 2:  # Unilateral derecha
        valor_critico = stats.norm.ppf(1 - nivel_significacion)
        p_valor = 1 - stats.norm.cdf(estadistico_z)
        decision = "Rechaza" if estadistico_z > valor_critico else "Acepta"
    elif tipo_hipotesis == 3:  # Unilateral izquierda
        valor_critico = stats.norm.ppf(nivel_significacion)
        p_valor = stats.norm.cdf(estadistico_z)
        decision = "Rechaza" if estadistico_z < valor_critico else "Acepta"
    else:
        raise ValueError("Tipo de hipótesis inválido.")

   # Mostrar resultados
    print("\nResultados del contraste de hipótesis:")
    print(f"Estadístico de prueba (Z): {estadistico_z:.2f}")
    print(f"Valor crítico: {valor_critico:.2f}")
    print(f"p-valor: {p_valor:.3f}")
    print(f"{decision} la hipótesis nula (H0).")

# Ejecutar la función
contraste_hipotesis_media_varianza_conocida()
```

Resolución con Pandas: Series y Dataframe

Ejercicio resuelto. A partir de los descriptivos anteriores, y utilizando los objetos Series y DataFrame de `pandas`, construye una función que devuelva el valor máximo y mínimo, junto con la tabla de distribución de frecuencias con las frecuencias absolutas, relativas y las correspondientes acumuladas para los siguientes datos: 1, 1, 1, 1, 2, 2, 2, 3, 3, 4.

```
import pandas as pd
s = pd.Series([1, 1, 1, 1, 2, 2, 2, 3, 3, 4])

def minmaxtabla(serie) :

    # Realizar el conteo de las frecuencias para la tabla y nombrar columnas (objeto Series)

    tabla_frec = serie.value_counts().sort_index().reset_index()
    tabla_frec.columns = ['Valor', 'Frecuencia']

    # Resumen de estadísticos
    print("\tResumen de estadísticos")
    print("================================================")
    print(f"El valor mínimo es {serie.min()} y el máximo {serie.max()}")
    print(f"\nLa media aritmética es {serie.mean()}, la cuasi-desviación típica {serie.std()}, \
          y la cuasi-varianza {serie.var()}")
    # Resumen descriptivo tamaño, media, desviación típica, mínimo, y cuartiles
    print(tabla_frec.describe())

    # Tabla de frecuencias
    print("\n\tTabla de frecuencias relativas")
    print("================================================")

    # Calcular frecuencias relativas y absolutas acumuladas como DataFrame
    tabla_frec['Ni'] = tabla_frec['Frecuencia'].cumsum()
    tabla_frec['fi'] = tabla_frec['Frecuencia'] / len(serie)
    tabla_frec['Fi'] = tabla_frec['fi'].cumsum()

    # Imprimir la tabla de frecuencias actualizada
    display(tabla_frec)
    return

minmaxtabla(s)
```

Ejercicio resuelto: Se desea analizar el rendimiento de diferentes variedades de tomate de una cooperativa agrícola. Se tienen los siguientes datos de rendimiento (en toneladas por hectárea) para cada variedad:

Variedad	Tm
cherry	103.5
kumato	94.2
pera	113.9
corazón	94.1
raf	84.3
muchamiel	103.8
roma	104.0
daniela	84.4
negro	94.1
marzano	84.2

Calcula:

- Rendimiento medio, desviación típica y varianza.
- Muestra las variedades que tienen un rendimiento superior al del promedio.
- Haz la representación de un gráfico de línea.

```
import pandas as pd
# Se declaran los datos como un diccionario
datos = {
    'cherry': 103.5,
    'kumato': 94.2,
    'pera': 113.9,
    'corazon': 94.1,
    'raf': 84.3,
    'muchamiel': 103.8,
    'roma': 104.0,
    'daniela': 84.4,
    'negro': 94.1,
    'marzano': 84.2
    }

#Convertir datos en objeto Series mediante el constructor
rendimiento = pd.Series(datos, name="Variedad")
N=rendimiento.size
print("Número de muestras:", N)

# Se obtienen distintos estadísticos del objeto Series rendimiento
promedio = rendimiento.mean()
print(f"Rendimiento promedio: {promedio:.2f}")
desv_estandar = rendimiento.std()
print(f"Cuasi-desviación estándar: {desv_estandar:.2f}")
print(f"Desviación estándar: {desv_estandar*(N-1)/N:.2f}")

# Selección de las variedades que tienen un rendimiento superior al promedio
superior_promedio = rendimiento[rendimiento > promedio]
print("Variedades con rendimiento superior al promedio: ")
print(superior_promedio)

Representación de los datos con un gráfico de línea  ``plot``:

import matplotlib.pyplot as plt
plt.figure(figsize=(10,2)) # cambiamos el tamaño de la figura

plt.plot(rendimiento.index,rendimiento.values, linestyle='--', marker='o',
         color='red')
plt.xlabel('Variedades')
plt.ylabel('Temperaturas')

#plt.yticks([3.5,3.8,4.1,4.4]) #valores a mostrar en eje Y
plt.show()
```

Ejercicio resuelto: Escribir una función que reciba un diccionario con las notas de los alumno de un curso y devuelva una serie con la nota mínima, la máxima, media y la desviación típica.

```
import pandas as pd

def estadistica_notas(notas):
    notas = pd.Series(notas)
    estadisticos = pd.Series([notas.min(), notas.max(), notas.mean(), \
                    notas.std()], index=['Min', 'Max', 'Media', 'Desviación típica'])
    return estadisticos

notas = {'Miguel':8.25, 'Ana':6.75, 'Daniel':6, 'Carmen': 8.5, 'Antonio': 5}
print(estadistica_notas(notas))

### Solución alterantiva

import pandas as pd

def estadistica_notas(notas):
```

```
    notas = pd.Series(notas)
    return notas.describe()

notas = {'Miguel':8.25, 'Ana':6.75, 'Daniel':6, 'Carmen': 8.5, 'Antonio': 5}
print(estadistica_notas(notas))
```

Ejercicio. Dada la distribución bidimensional de la tabla adjunta, determinar la recta de ajuste de Y en función de X y la bondad de dicho ajuste, asi como su representación.

X	1	2	3	4	5	6
Y	**2**	**5**	**9**	**13**	**17**	**21**

Importación de las librerías a utilizar:

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

En primer lugar, se construyen las listas como objetos Series de la librería `pandas` con los datos del enunciado, que representan a la **variable independiente** X y a la **dependiente** Y:

```
X = pd.Series([1, 2.5, 3.5, 4, 5, 6], name="X")
Y = pd.Series([2, 5.5, 9.5, 15, 16, 21],name="Y")
```

La **recta de regresión** de Y en función de X se calcula de la siguiente manera:

$$y = \frac{\sigma_{xy}}{\sigma_x^2}.(x - \overline{x}) + \overline{y}$$

Por tanto, se necesita calcular los coeficientes y para ello se calculan los estadísticos:

```
# Cálculos de estadísticos
n=X.size
mediaX = X.mean()
mediaY= Y.mean()
varX = X.var()*(n-1)/n
varY = Y.var()*(n-1)/n
DT_X=varX**(1/2)
DT_Y=varY**(1/2)
```

Para el cálculo de la **covarianza** hay que rectificar el valor que aporta `pandas` con el factor $\frac{n-1}{n}$. También se necesita convertir las series a un `DataFrame` para poder utilizar el método que calcula la covarianza.

```
# Unión de las dos series en un DataFrame
df = pd.concat([X,Y], axis=1)

# Matriz de covarianzas
df.cov()

# Se aplica factor de corrección para covarianza
covarXY=df.cov().loc["X","Y"]*(n-1)/n
covarXY

# Recta de regresión de Y/X
a=round(covarXY/varX,3)
b= round(mediaY - a*mediaX,3)
```

Sus **coeficientes** resultan ser:

```
print(f"Recta de regresión de Y/X: y = {a}.x + {b}")
```

La bondad del ajuste, calculada a través del **coeficiente de determinación** R^2:

```
R2 = round(covarXY**2/(varX*varY),4)
R2
```

Que indica que el ajuste es excelente.

Si se representa gráficamente el conjunto:

```
plt.title("Ajuste por mínimos cuadrados")
plt.xlabel("X")
plt.ylabel("Y")

plt.plot(X, a*X+b, color = 'lightblue', linewidth = 5) #Recta de regresión
plt.scatter(X, Y, color = 'orange')

plt.show()
```

Ejercicio resuelto. Suponga que se tienen los siguientes datos de una cosecha de trigo en diferentes regiones:

Región	Cosecha (toneladas)	Área sembrada (hectáreas)
A	300	150
B	500	250
C	400	200
D	450	225

Se quiere analizar estos datos utilizando DataFrames y gráficos.

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

datos = {
    "Región": ["A", "B", "C", "D"],
    "Cosecha (toneladas)": [300, 500, 400, 450],
    "Área sembrada (hectáreas)": [150, 250, 200, 225]
}

df = pd.DataFrame(datos)

# Ahora se pueden realizar diferentes análisis sobre estos datos.
# Por ejemplo, se calcula la cosecha promedio por hectárea:

df["Rendimiento (toneladas/hectárea)"] = df["Cosecha (toneladas)"] / df["Área sembrada (hectáreas)"]

# También se pueden visualizar los datos mediante gráficos.
# Por ejemplo, se puede hacer un gráfico de barras para comparar la cosecha en cada región:

plt.bar(df["Región"], df["Cosecha (toneladas)"])
plt.title("Cosecha por región")
plt.xlabel("Región")
plt.ylabel("Cosecha (toneladas)")
plt.show()
```

También se puede hacer un gráfico de dispersión para visualizar la relación entre la cosecha y el área sembrada:

```
plt.scatter(df["Área sembrada (hectáreas)"], df["Cosecha (toneladas)"])
plt.title("Relación entre área sembrada y cosecha")
plt.xlabel("Área sembrada (hectáreas)")
plt.ylabel("Cosecha (toneladas)")
plt.show()
```

Ejercicio resuelto. Se tienen los datos de la producción de maíz en kg/ha en diferentes regiones de un país durante los últimos 5 años. Se quiere realizar un análisis estadístico utilizando la librería Pandas de Python y visualizar los datos con matplotlib.

Año	2018	2019	2020	2021	2022
Región A	800	950	900	1000	1100
Región B	700	800	850	950	1000
Región C	600	700	750	800	900
Región D	650	750	800	900	950

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


datos = {
    "Región A": [800, 950, 900, 1000, 1100],
    "Región B": [700, 800, 850, 950, 1000],
    "Región C": [600, 700, 750, 800, 900],
    "Región D": [650, 750, 800, 900, 950]
}

df = pd.DataFrame(datos, index=[2016, 2017, 2018, 2019, 2020])
print(df,"\n")

# Resumen estadístico
print(df.describe())

Crear un diagrama de barras para comparar las producciones de cada región:

#regiones = ["Región A", "Región B", "Región C", "Región D"]
regiones = list(datos.keys())

# Calcular las producciones medias por región
producciones_medias = [df.mean()[region] for region in regiones]

# Imprimir las producciones medias
for i in range(len(regiones)):
    print(f"Producción media en {regiones[i]}: {producciones_medias[i]} kg/ha")


print(producciones_medias)

plt.bar(regiones, producciones_medias)
plt.grid()
plt.title("Producción media de maíz por región")
plt.xlabel("Regiones")
plt.ylabel("Producción (kg/ha)")
plt.show()

# Gráfico de líneas para la producción por región
plt.figure(figsize=(10, 6))
df.transpose().plot(marker='o')
plt.title('Producción de Maíz por Región (2018-2022)')
plt.xlabel('Año')
plt.ylabel('Producción (kg/ha)')
plt.legend(title='Región')
plt.show()
```

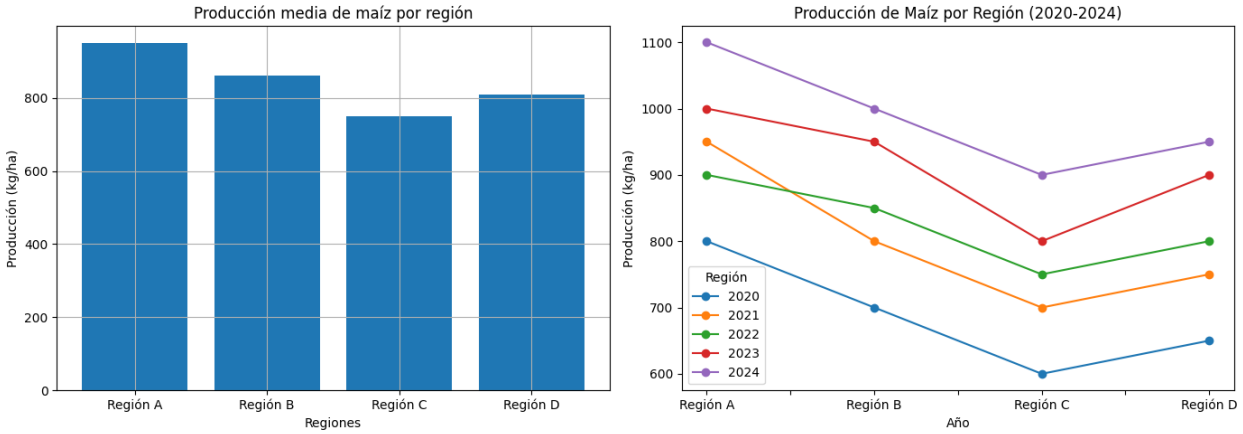

Figura 36: Gráficos de producción

Tratamiento de datos en un caso aplicado

La **ciencia de datos** es un campo interdisciplinar que combina principios de estadística, matemáticas, informática y conocimiento de dominios específicos para extraer información valiosa a partir de datos. Su objetivo es analizar, interpretar y utilizar grandes volúmenes de datos, estructurados y no estructurados, para generar conocimiento que ayude en la toma de decisiones, descubrimientos científicos, y soluciones a problemas complejos.

El ciclo de vida de los datos pasa por distintas etapas. Desde el momento de su creación o generación, los datos siguen un ciclo de vida hasta que se publican para su diseminación y también se integran en la implementación de los modelos creados. Las fases no son completamente lineales, y en muchos casos pueden superponerse o interactuar entre sí. Las principales etapas del ciclo de vida son las siguientes:

- **Generación**: los datos se generan como resultado de uno o más procesos, como el pago con una tarjeta de crédito o las interacciones entre los usuarios de una red social.
- **Captura**: en algunos casos, los datos generados ya se almacenan directamente, pero es necesario buscarlos y acceder a otros, ya sea mediante procedimientos establecidos (como una API- Application Programming Interface) o mediante el desarrollo de procedimientos de extracción (con herramientas de scraping).
- **Almacenamiento**: los datos se pueden almacenar en archivos planos, bases de datos relacionales, pero también en otros que explotan su naturaleza, como bases de datos orientadas a grafos, o columnas, etc.
- **PROCESAMIENTO**: normalmente antes de proceder con el análisis es necesario procesar los datos por diferentes razones, básicamente para seleccionar los que formarán parte del análisis y también calcular nuevas variables a partir de los originales.
- **ANÁLISIS**: se extrae conocimiento de los datos en forma de modelos que permitan explicar relaciones, detectar patrones y tendencias, hacer predicciones, etc.

- **VISUALIZACIÓN**: el objetivo de esta etapa es doble; por un lado, complementar el análisis mediante visualizaciones que proporcionen conocimiento sobre los datos, y por otro lado, la creación de visualizaciones que muestren las características más relevantes de los datos, y respondan a una pregunta concreta parte del análisis
- **Publicación**: los datos, normalmente ya procesados o el resultado de los modelos analíticos construidos pueden publicarse para que terceros puedan explotarlos.

Este caso de estudio se centrará en las etapas que aparecen con mayúscula: procesamiento, análisis y visualización utilizando diversas librerías, en concreto `Pandas` y algo de visualización con `Matplotlib`, pero a escala "casita de pájaros".

Lo habitual es trabajar con un conjunto de librerías que facilitan en tratamiento y análisis de los datos, que se denomina **ecosistema PyData**. Esta compuestos por las siguientes librerías:

- **`Pandas`** que incluye desde transformaciones de datos, como ordenar filas y tomar subconjuntos, sobre tablas de datos (DataFrames).
- **`NumPy`** para cálculo numérico, de hecho Pandas esta contruido sobre Numpy
- **`Matplotlib`**, **`Seaborn`**, **`Plotlyy`** otros paquetes de visualización de datos
- **`scikit-learn`** para aprendizaje automático

Entrada y salida de datos

En este caso simple se parte de tablas de datos organizadas en archivos, csv, texto o excel. Pero esta obtención y almacenamiento de datos puede ser diverso dependiendo del proyecto concreto debiendo necesitar bases de datos o scraping. El caso de estudio es un ejemplo de una tabla de datos sobre parcelas agrícolas y otros datos de carácter agronómico que se han simplificado y se resumen a continuación.

- parcelas – datos de las parcelas bajo la RAIF (Red de alerta e información fitosanitaria), cuyas columnas son:

```
CAMPAÑA; PROVINCIA; MUNICIPIO; CODPARCELA; Comarca; Paraje; Zona; U_H_C; Tipo_Explotación;
Sistema_Gestión; Superficie; Pendiente; Orientación; Tipo_suelo; Tipo_invernadero; Cubierta;
Fecha_plástico; Tipo_plástico; Sistema_nebulización; Doble_puerta; Ventilación; Malla;
Especie; Variedad; Cultivo_precedente; Incidencias_cultivo_anterior; Riego;
Procedencia_agua; Calidad_agua; Limpieza_exterior; Limpieza_interior; Cuba_independiente;
Cumplimiento_Lucha_biológica; Num_Campañas; Representa_U_H_C; Zona_Biológica_RAIF;
Secano; Codigo_invernadero; Instalaciones_pulverización; Coordenada_UTM;
Coordenada_Y_UTM; Altitud; Estación_climática; Otro_plástico; Calefacción; Sensor
```

- muestreoshort – una porción reducida en número de filas de las variables monitorizadas dentro del protocolo de producción integrada en agricultura, cuyas columnas son:

```
provincia; municipio; codparcela; cultivo; fecha; variable; valor
```

- producción – datos de producción en toneladas de hortícolas por provincias andaluzas en año 2014 para los meses enero, febrero y marzo

```
Producto; Año; mes; Provincia; Produccion.TM
```

- producción histórica. Datos de superficies y toneladas producidas del 2013 al 2024 en los cultivos de algodón y trigo en Andalucía.

```
producto; fecha; superficie.Ha; produccion.TM
```

Importar archivos CSV

Se debe utilizar `read_csv()` con la ruta al archivo para leer un archivo de valores separados por comas.

Importar archivos de texto

La lectura de archivos de texto es similar a la de archivos CSV. El único matiz es que se debe especificar un separador con el argumento `sep`. El argumento separador se refiere al símbolo utilizado para separar filas en un `DataFrame`.

Importar archivos Excel (una sola hoja)

Leer archivos excel (tanto XLS como XLSX) es tan fácil como la función `read_excel()`, con la ruta del archivo como entrada. También se pueden especificar otros argumentos, como header para especificar qué fila se convierte en la cabecera del DataFrame. Tiene un valor predeterminado de 0, que denota la primera fila como cabecera o nombre de columna. También se pueden especificar los nombres de las columnas como una lista en el argumento names. El argumento index_col (de forma predeterminada es None) puede utilizarse si el archivo contiene un índice de filas.

Importar archivos Excel (varias hojas)

Leer archivos Excel con varias hojas no es tan diferente. Solo hay que especificar un argumento adicional, sheet_name, en el que se puede pasar una cadena para el nombre de la hoja o un número entero para la posición de la hoja (ten en cuenta que Python utiliza la indexación 0, en la que se puede acceder a la primera hoja con `sheet_name = 0`).

```
df1 = pd.read_csv("parcelas.csv")
df2 = pd.read_csv("parcelas.txt", sep="\s")
df3 = pd.read_excel('parcelas.xlsx')
df4 = pd.read_excel('parcelas_total.xlsx', sheet_name=1)
```

La lectura de formatos específicos puede requerir de librerías adicionales, se recomienda el uso de `csv` indicando el separador que puede se la coma o el punto y coma. Además, se debe tener en cuenta donde está el archivo de datos. En el siguiente ejemplo se indica como gestionar esto de una forma simple, colocando la ruta del archivo de datos, para el uso en un codespace de Github.

```
import pandas as pd
import numpy as np
from pathlib import Path

def leer_csv(nombre_archivo):
    rutaActual = Path(__file__).parent.resolve()
    rutaDatos = rutaActual / nombre_archivo
    df = pd.read_csv(rutaDatos, sep=";")
    df.replace(0, np.nan, inplace=True)

    return df

if __name__ == "__main__":
```

```
    muestreos = leer_csv('muestreoshort.csv')
    parcelas = leer_csv('parcelas.csv')
    produccion = leer_csv('produccion.csv')
    historico = leer_csv('prodhistorico.csv')
```

Si se está usando el entorno de desarrollo Google Colaboratory (Colab), y suponiendo que los datos están en una carpeta llamada `datos_produccion` el código a utilizar es

```
import pandas as pd
import numpy as np
from pathlib import Path
import os

# Montar Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Ruta base a tu carpeta con archivos CSV en Drive
carpeta_drive = Path('/content/drive/MyDrive/datos_produccion')

def leer_csv(nombre_archivo):
    ruta = carpeta_drive / nombre_archivo

    # Verificar si el archivo existe
    if not ruta.exists():
        print(f"El archivo '{nombre_archivo}' no se encuentra en la carpeta especificada.")
        return None

    try:
        df = pd.read_csv(ruta, sep=";")
        df.replace(0, np.nan, inplace=True)
        print(f" Archivo '{nombre_archivo}' leído correctamente.")
        return df
    except Exception as e:
        print(f" Error al leer '{nombre_archivo}': {e}")
        return None

# Lectura segura de archivos
muestreos = leer_csv('muestreoshort.csv')
parcelas = leer_csv('parcelas.csv')
produccion = leer_csv('produccion.csv')
historico = leer_csv('prodhistorico.csv')
```

Al igual que Pandas puede importar datos de varios tipos de archivos, también permite exportar datos a varios formatos. Esto ocurre especialmente cuando los datos se transforman mediante Pandas y necesitan guardarse localmente en la máquina. A continuación se explica cómo convertir los DataFrames de pandas a varios formatos.

Un DataFrame de pandas (aquí estamos utilizando `df`) se guarda como un archivo CSV mediante el método `.to_csv()`. Los argumentos incluyen el nombre del archivo con la ruta, donde `index = True` implica escribir el índice del DataFrame.

```
df.to_csv("parcelastratadas.csv", index=False)
df.to_excel("parcelastratadas.xlsx", index=False)
```

Análisis de DataFrames con Pandas

Después de leer los datos tabulares como un DataFrame, será necesario echar un vistazo a los datos. Una porción del conjunto de datos o un resumen de los datos en forma de estadísticas resumidas.

Cómo ver los datos mediante .head() y .tail()

Se pueden especificar el número de filas mediante el argumento `n` (el valor predeterminado es 5) sobre las funciones para mostrar las primeras o las últimas filas del Dataframe.

```
df.head()
df.tail(n=10)
```

Revisar los datos con .describe()

El método `.describe()` imprime los estadísticos de resumen de todas las columnas numéricas, como el recuento, la media, la desviación típica, el rango y los cuartiles de las columnas numéricas.

```
df.describe()
```

También se pueden cambiar los cuartiles con el argumento percentiles. Aquí, por ejemplo, se muestran los percentiles 30 %, 50 % y 70 % de las columnas numéricas de `df`.

```
df.describe(percentiles=[0.3, 0.5, 0.7])
```

Se pueden aislar tipos de datos específicos en la salida resumida con el argumento `include`. Aquí, por ejemplo, solo se resumen las columnas con el tipo de datos entero (`int`).

```
df.describe(include=[int])
```

Estos resultados se pueden trasponer para facilitar la visualización

```
df.describe().T
```

Revisar los datos con .info()

El método `.info()` es una forma rápida de ver los tipos de datos. Con `show_counts=True`, se muestra la cantidad de valores no nulos en cada columna del DataFrame. Gracias a `memory_usage=True`, también se imprime el uso de memoria total que ocupa el DataFrame. Finalmente, con `verbose=True`, se asegura que se detallen todas las columnas, incluso si el DataFrame tiene muchas. Todo esto permite obtener un resumen detallado de la estructura, tipos de datos y tamaño en memoria del DataFrame `df`.

```
df.info(show_counts=True, memory_usage=True, verbose=True)
```

Revisar tus datos con .shape

El atributo `.shape` de un DataFrame permite identificar la cantidad de filas y columnas. Retorna una tupla (filas, columnas), de la cual se puede acceder individualmente a las filas o columnas mediante indexación.

```
df.shape # obtiene numero de filas y columnas
df.shape[0] # Obtiene el número de filas
df.shape[1] # obtiene le número de columnas
```

Obtener todas las columnas y nombres de columnas

La llamada al atributo `.columns` de un objeto DataFrame devuelve los nombres de las columnas en forma de objeto `Index`. Como recordatorio, un índice de pandas es la dirección/etiqueta de la fila o columna.

```
df.columns
list(df.columns)  # lo convierte en una lista
```

Comprobación de valores perdidos en pandas con .isnull()

El método `.copy()` hace una copia del DataFrame original. Esto se hace para garantizar que cualquier cambio en la copia no se refleje en el DataFrame original. Al utilizar `.loc`, se puede establecer las filas dos a cinco de la columna `Produccion` en valores `NaN`, que denotan valores omitidos.

```
df.isnull().head(7)
```

Sobre el caso, el ejemplo completo sería

```
from leer import leer_csv
import pandas

muestreos = leer_csv('muestreoshort.csv')
parcelas = leer_csv('parcelas.csv')
produccion = leer_csv('produccion.csv')

print(parcelas.head())
print('\n')
print(parcelas.tail(n=10))
print('\n')
print(produccion.describe)
print('\n')
print(produccion.describe(percentiles=[0.3, 0.5, 0.7]))
print('\n')
print(muestreos.describe(include=[float]))
print('\n')
print(muestreos.describe().T)
print('\n')
print( produccion.info(show_counts=True, memory_usage=True, verbose=True))
print('\n')
print(muestreos.shape)
print('\n')
print(muestreos.shape[0])
print('\n')
print(muestreos.shape[1])
print('\n')
print(list(parcelas.columns))  # lo convierte en una lista
print('\n')
print(list(produccion.columns))
print('\n')
print(produccion.isnull().head(8))
print('\n')
```

Extraer datos en Pandas

Pandas ofrece varias formas de subconjuntar, filtrar y aislar datos en tus DataFrames. Las formas más comunes son:

Aislar una columna con [ ]

Se puede aislar una sola columna utilizando un corchete `[ ]` con el nombre de la columna dentro. La salida es un objeto pandas Series. Una serie en pandas es una matriz unidimensional que contiene datos de cualquier tipo, incluidos enteros, flotantes, cadenas, booleanos, objetos Python, etc. Un DataFrame se compone de muchas series que actúan como columnas.

```
list(parcelas['Especie'])
```

Aislar una fila con [ ]

Se puede obtener una sola fila pasando una serie booleana con un valor `True`. En el ejemplo siguiente, se devuelve la segunda fila con `index = 1`. Aquí, `.index` devuelve las etiquetas de fila del DataFrame, y la comparación lo convierte en una matriz unidimensional booleana.

```
print(parcelas[parcelas.index == 4])
```

Aislar dos o más filas con [ ]

Del mismo modo, se pueden devolver dos o más filas utilizando el método `.isin()` en lugar de un operador `==`.

```
parcelas[parcelas.index.isin(range(2, 5))]
```

Corte condicional (que se ajusta a determinadas condiciones)

Pandas permite filtrar datos mediante condiciones sobre valores de fila/columna. Aquí, se aislan filas utilizando los corchetes `[ ]`. Sin embargo, en lugar de introducir índices de fila o nombres de columna, se está introduciendo una condición en la la columna, por ejemplo, seleccionar solo los muestreos de la fenología.

```
muestreos[muestreos.variable == "Fenologia"]
```

Aislar filas en función de una condición en pandas

El siguiente ejemplo obtiene todas las filas en las que `especie` es `'pimiento'`. Aquí `parcelas.Especie` selecciona esa columna, parcelas. `Especie == 'Pimiento'` devuelve una serie de valores booleanos determinando, a continuación `[]` toma un subconjunto de `df` donde esa serie booleana es `True`.

```
parcelas[parcelas.Especie == 'Pimiento']
```

Utilizando un operador `>` se establecen comparaciones. El código siguiente busca en las parcelas aquellas de más de una hectárea obteniendo la campaña, la provincia, el municipio y el código de parcela.

```
parcelas.loc[parcelas['Superficie'] > 10000, ['CAMPAÑA','PROVINCIA','MUNICIPIO','CODPARCELA']]
```

Limpieza de datos con Pandas

La limpieza de datos es una de las tareas más comunes en la ciencia de datos. Pandas permite preprocesar datos para cualquier uso, incluido, entre otros, el entrenamiento de modelos de machine learning y deep learning pero queda fuera del alcance de este caso.

Por ejemplo en producción hay muchos valores vacíos. Existen muchas técnicas para manejar los datos nulos, la versión más simple es eliminar estas filas puesto que en este caso es una sola variable.

```
produccionlimpia = produccion.dropna(subset=['Produccion.TM'])
```

Cálculos sobre datos en Pandas

La principal propuesta de valor de Pandas reside en su rápida funcionalidad de análisis de datos. En esta sección, se centrarña en un conjunto de técnicas de análisis que puedes utilizar en Pandas.

Operadores de resumen (media, moda, mediana)

Se puede obtener la media de cada valor de columna utilizando el método `.mean()`.

```
parcelas2023 = parcelas[parcelas.CAMPAÑA == 2023]
parcelas2023.mean(numeric_only=True)
```

La moda puede calcularse de forma similar utilizando el método `.mode()`.

```
print(produccion.mode())
```

Del mismo modo, la mediana de cada columna se calcula con el método `.median()`

```
produccion.median()
```

Crear nuevas columnas basadas en columnas existentes

Pandas proporciona un cálculo rápido y eficaz combinando dos o más columnas como variables escalares. Por ejemplo queremos calcular la oscilación termica.

```
historico['rendimiento'] = historico['producción.T']/historico['superficie.Ha']
```

Contar utilizando `.value_counts()`

A menudo se trabaja con valores categóricos, y se quiere contar el número de observaciones que tiene cada categoría en una columna. Los valores de las categorías pueden contarse utilizando los métodos `.value_counts()`.

Aquí, por ejemplo, contamos el número de observaciones por provincia.

```
print(parcelas['MUNICIPIO'].value_counts())
```

Si se añade el argumento `normalize`, obtendrás proporciones en lugar de recuentos absolutos.

```
print(parcelas['MUNICIPIO'].value_counts(normalize=True))
print('\n')
```

Uso de agrupaciones con groupby

El método `groupby()` en Pandas se utiliza para agrupar los datos de un DataFrame en función de los valores de una o más columnas. Una vez agrupados, puedes aplicar funciones de agregación como `mean()`, `sum()`, `count()`, entre otras, sobre cada grupo para obtener resúmenes estadísticos o realizar análisis específicos.

¿Cómo funciona `groupby()`?

- Agrupación de datos: Divide los datos en grupos según los valores de la(s) columna(s) que se indique.
- Aplicación de una función: Después de agrupar, se puede aplicar funciones de agregación (por ejemplo, calcular la media, suma, contar los elementos, etc.) sobre cada grupo.

- Combinación: El resultado de la función se devuelve en un nuevo DataFrame o Series.

```
# Filtrar solo los datos de "Calabacín protegido"
calabacin_protegido = produccion[produccion['Producto'] == 'Calabacín protegido']

# Agrupar por provincia y calcular la media de la producción
produccion_media = calabacin_protegido.groupby('Provincia')['Produccion.TM'].mean().reset_index()

# Renombrar la columna de producción
produccion_media.columns = ['Provincia', 'Produccion_Media_TM']
print(produccion_media)
```

El script ejemplo es:

```
from leer import leer_csv

muestreos = leer_csv('muestreoshort.csv')
parcelas = leer_csv('parcelas.csv')
produccion = leer_csv('produccion2014.csv')
historico = leer_csv('prodhistorico.csv')

print(list(set(parcelas['Especie'])))
print('\n')
print(parcelas[parcelas.index==4])
print('\n')
print(parcelas[parcelas.index.isin(range(2,5))])
print('\n')
print(muestreos[muestreos.variable == "M. blanca: % plantas con presencia"])
print('\n')
print(parcelas[parcelas.Especie == 'Pimiento'])
print('\n')
print(parcelas.loc[parcelas['Superficie'] > 10000, ['CAMPAÑA','PROVINCIA',\
    'MUNICIPIO','CODPARCELA']])
print('\n')
print(produccion)
print('\n')
produccionlimpia = produccion.dropna(subset=['Produccion.TM'])
print(produccionlimpia)
print('\n')
parcelas2023 = parcelas[parcelas.CAMPAÑA == 2023]
print(parcelas2023.mean(numeric_only=True))
print('\n')
print(produccionlimpia.mode())
print('\n')
print(produccionlimpia.median(numeric_only=True))
print('\n')
historico['rendimiento'] = historico['producción.T']/historico['superficie.Ha']
print(historico.head())
print('\n')
print(parcelas['MUNICIPIO'].value_counts())
print(parcelas['MUNICIPIO'].value_counts(normalize=True))
calabacin_protegido = produccion[produccion['Producto'] == 'Calabacín protegido']
produccion_media = calabacin_protegido.groupby('Provincia')['Produccion.TM'].mean().reset_index()
produccion_media.columns = ['Provincia', 'Produccion_Media_TM']
print(produccion_media)
```

Visualización de datos con Matplotlib

Para comenzar usando el módulo `pyplot` de **Matplotlib** en un programa **Python** la forma estándar es:

```
import numpy as np # no siempre es necesaria pero recomendable
import matplotlib.pyplot as plt
```

Función `plot()` y `show()`

Mediante la función `.plot()` suministramos una lista para su trazado y `.show()` muestra el

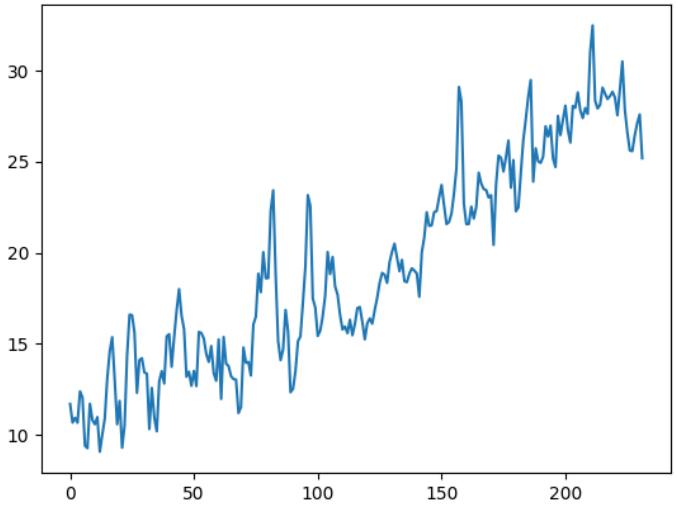

Figura 37: Temperaturas medias estación AL001

gráfico creado. Para la estación meteorológica "AL001".

```
# Figura 37
figura = plt.subplot()
figura.plot(clima_est[['T_Med', 'H_R_Med']])
```

o bien

```
# Figura 38
 plt.plot(clima_est['T_Med'], label='T_Med')  # primera columna
 plt.plot(clima_est['H_R_Med'], label='H_R_Med')
```

Gráfico nube de puntos o diagrama de dispersión

```
# Figura 39
temp = clima['T_Med']
hum = clima['H_R_Med']
plt.scatter(temp, hum)
plt.xlabel('Temperatura (°C)')
plt.ylabel('Humedad (%)')
plt.title('Temperatura vs Humedad Relativa')
plt.show()
```

Gráficos de barras

Mostrar un gráfico de barras por provincia agrupando los datos de las tres estaciones de medida.

```
# Figura 40
temperatura_media = clima.groupby('provincia')['T_Med'].mean().reset_index()

plt.bar(temperatura_media['provincia'], temperatura_media['T_Med'],\
      label='Temperatura Media (°C)')

plt.xlabel('Provincia')
plt.ylabel('Valor promedio')
plt.title('Temperatura')
```

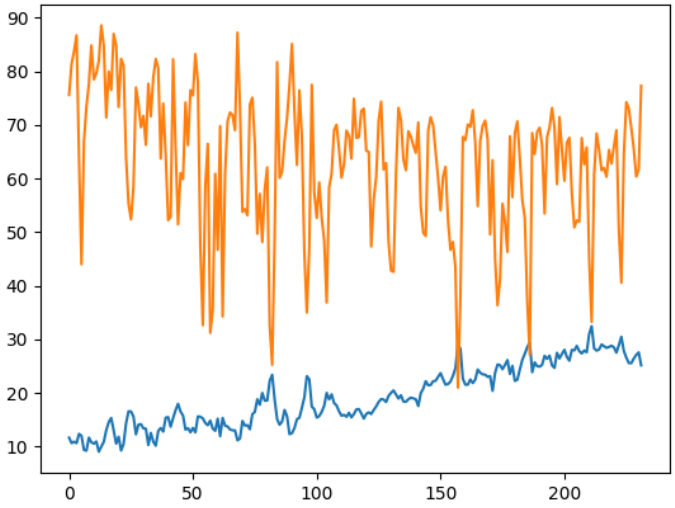

Figura 38: Temperatura y humedad media de la estación AL001

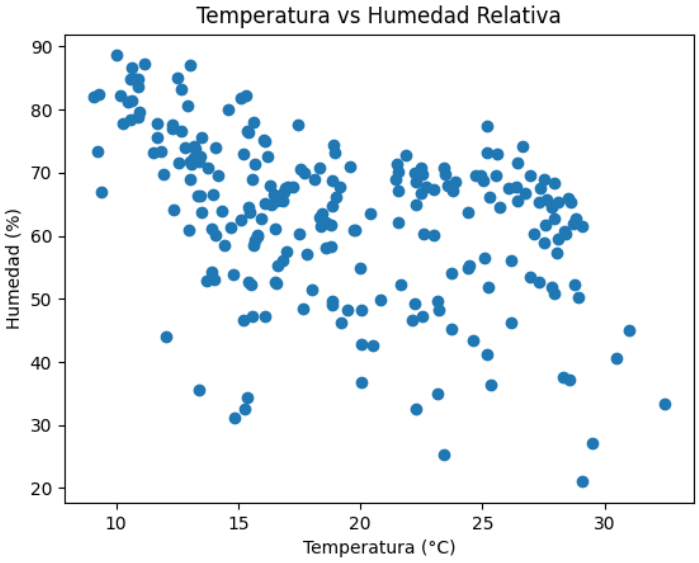

Figura 39: Temperatura media vs Humedad media para estación AL001

```
plt.savefig('mi_grafico_temperatura.png')
plt.close()
```

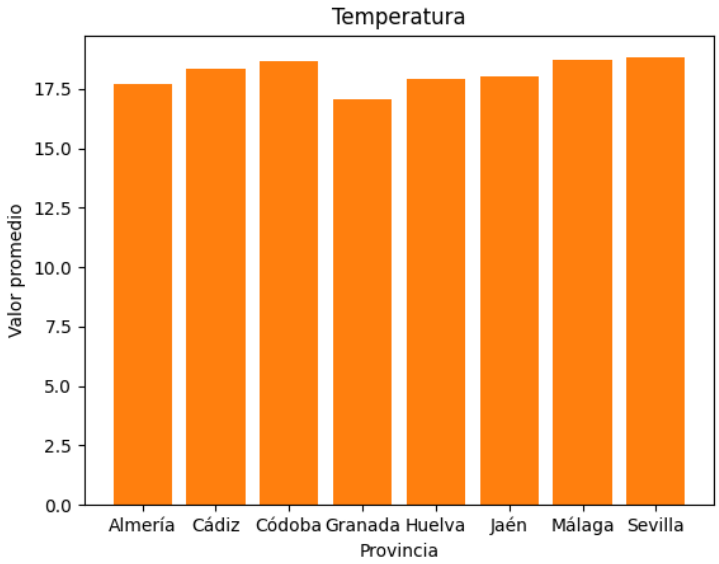

Figura 40: Promedio de temperaturas por provincias

Multiples barras coloreadas y otras opciones

Para representar la temperatura y la humedad por provincias

```
# Figura 41
temperatura_media = clima.groupby('provincia')['T_Med'].mean().reset_index()
humedad_media = clima.groupby('provincia')['H_R_Med'].mean().reset_index()

plt.figure(figsize=(12, 6))

bar_width = 0.35  # Ancho de las barras
x = np.arange(len(temperatura_media['provincia']))  # Posiciones para las barras
# Crear gráfico de barras agrupadas
plt.bar(x - bar_width/2, temperatura_media['T_Med'],\
     width=bar_width, color='skyblue', alpha=0.7, label='Temperatura Media (°C)')
plt.bar(x + bar_width/2, humedad_media['H_R_Med'], width=bar_width,\
     color='orange', alpha=0.7, label='Humedad Media (%)')

# Configurar el gráfico
plt.xlabel('Provincia', fontsize=12)
plt.ylabel('Valor promedio', fontsize=12)
plt.title('Temperatura y Humedad Media por Provincia', fontsize=16)
plt.xticks(x, temperatura_media['provincia'], rotation=45)  # Mostrar nombres de provincias
plt.legend()
plt.grid(axis='y')

plt.tight_layout()
plt.show()
```

Gráfico de sectores

Distribución de producciones medias:

```
# Figura 42
produccion = produccion.dropna(subset=['Produccion.TM'])
produccion_media = produccion.groupby(['Provincia', 'Producto'])\
```

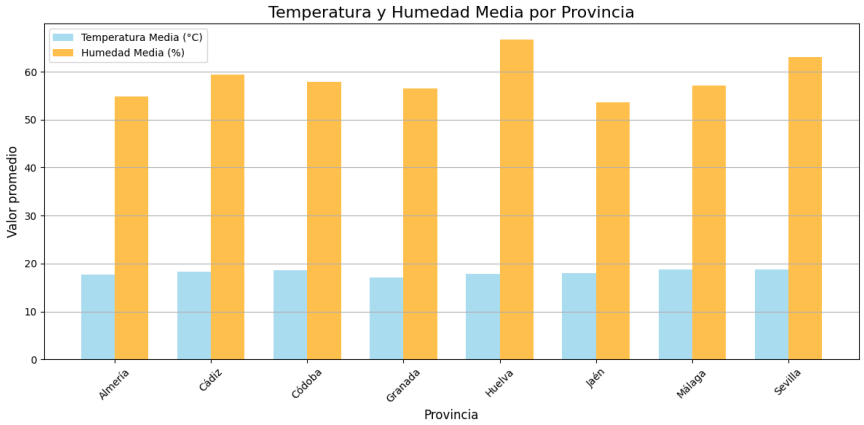

Figura 41: Temperatura y Humedad Media por Provincia

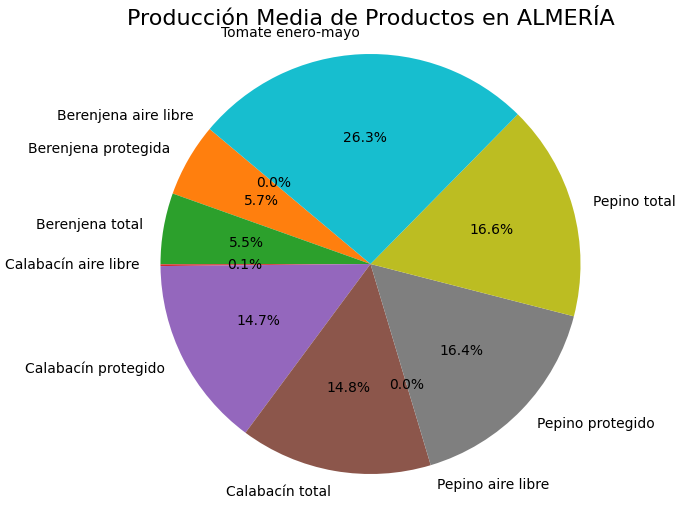

Figura 42: Producción Media de Productos en Almería

```
        ['Produccion.TM'].mean().reset_index()
produccion_media.columns = ['Provincia', 'Producto', 'Produccion_Media_TM']
print(produccion_media)
provincia_seleccionada = 'ALMERÍA'  # Cambia esto según la provincia que desees visualizar

datos_provincia = produccion_media[produccion_media['Provincia'] == provincia_seleccionada]

# Crear el gráfico de sectores
plt.figure(figsize=(10, 6))
plt.pie(datos_provincia['Produccion_Media_TM'],
        labels=datos_provincia['Producto'],
        autopct='%1.1f%%',
        startangle=140)
plt.title(f'Producción Media de Productos en {provincia_seleccionada}', fontsize=16)
plt.axis('equal')  # Igualar los ejes para que el gráfico sea un círculo
plt.show()
```

Histograma

Por ejemplo, sobre las temperaturas de una estación meteorológica dada:

```
# Filtrar los datos de la estación "AL001" en Almería
# Figura 43
almeria_al001 = clima[clima['cod_est'] == 'AL001']

# Asegurarse de que la columna 'T_Med' no contenga NaN
almeria_al001 = almeria_al001.dropna(subset=['T_Med'])

# Crear el histograma para las temperaturas medias de la estación "AL001"
plt.figure(figsize=(10, 6))
plt.hist(almeria_al001['T_Med'], bins=10, color='skyblue', edgecolor='black')
plt.title('Histograma de Temperatura Media en la Estación AL001 (Almería)', fontsize=16)
plt.xlabel('Temperatura Media (°C)', fontsize=12)
plt.ylabel('Frecuencia', fontsize=12)
plt.grid(axis='y', alpha=0.75)

plt.tight_layout()
plt.show()
```

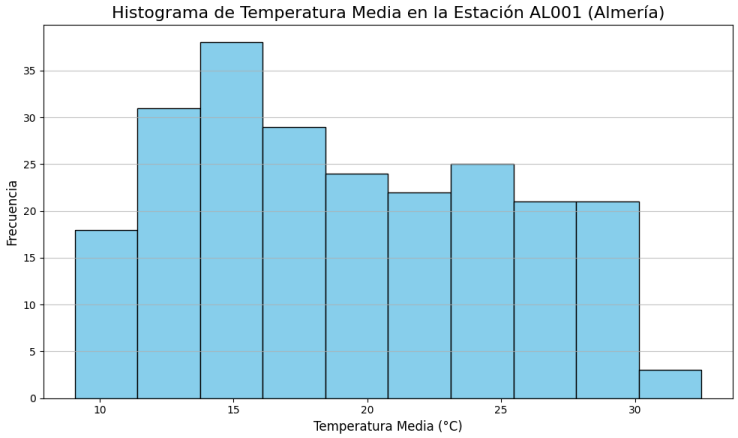

Figura 43: Histograma de Temperatura Media en la Estación AL001 (Almería)

Para ver otro ejemplo se consideran los muestreos de mosca blanca.

```
# Figura 44
spider = muestreos[muestreos['variable'] == 'Araña roja:% plantas con presencia']
spider_almeria = spider[spider['provincia'] == 'Almería']

# Asegurarse de que la columna 'valor' no contenga NaN
spider_almeria = spider_almeria.dropna(subset=['valor'])
```

```
plt.figure(figsize=(10, 6))
plt.hist(spider_almeria['valor'], bins=10, color='skyblue', edgecolor='black')
plt.title('Histograma de Araña Roja en Almería', fontsize=16)
plt.xlabel('Porcentaje de plantas con presencia de M. blanca (%)', fontsize=12)
plt.ylabel('Frecuencia', fontsize=12)
plt.grid(axis='y', alpha=0.75)
plt.show()
```

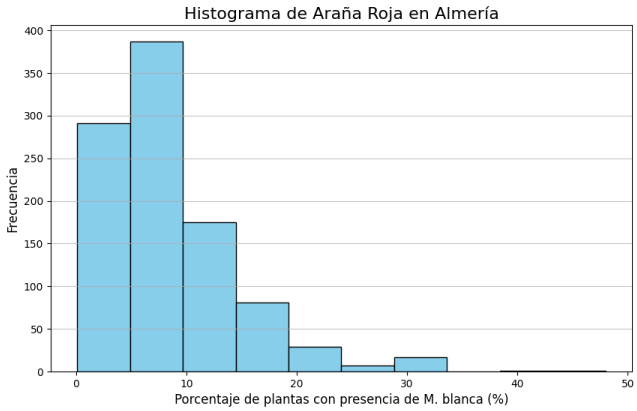

Figura 44: Histograma de Araña Roja en Almería

Diagrama de cajas y bigotes

```
import matplotlib.pyplot as plt

notas = [4, 8, 7.5, 6, 5.5, 5.2, 3.5, 7.7, 3.2, 9, 6.8, -2]

# Hemos forzado un outliner incluyendo un -2.  Figura 45

plt.boxplot(notas)
plt.title("Boxplot con Matplotlib")
plt.show()
```

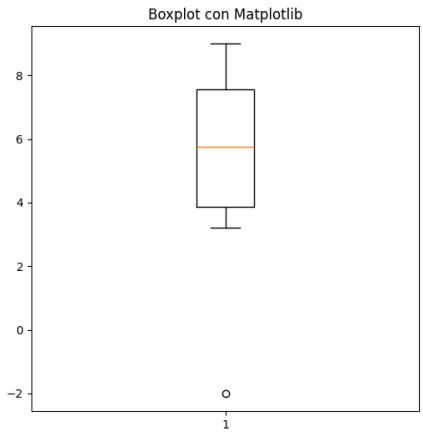

Figura 45: Diagrama de cajas y bigotes

Sobre las temperaturas podemos ver un ejemplo más detallado

```
# Figura 46
import matplotlib.pyplot as plt
from leer import leer_csv
import pandas as pd

clima = leer_csv('clima.csv')
clima_est = clima[clima['cod_est'] == 'AL001']

# Asegurarse de que la columna 'T_Med' sea numérica y convertir valores inválidos en NaN
clima_est['T_Med'] = pd.to_numeric(clima_est['T_Med'], errors='coerce')

# Eliminar valores NaN
clima_est = clima_est.dropna(subset=['T_Med'])

plt.figure(figsize=(8, 6))
plt.boxplot(clima_est['T_Med'], vert=False, showmeans=True)
plt.title("Boxplot de Temperatura Media en la Estación AL001")
plt.xlabel("Temperatura Media (°C)")
plt.show()
```

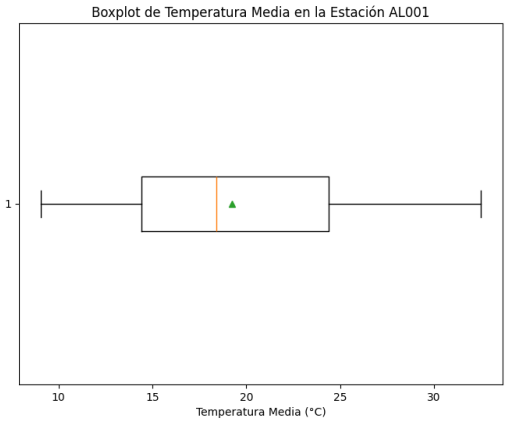

Figura 46: Boxplot de Temperatura Media en la Estación AL001

Elaboración de modelos

El modelo de regresión lineal trata de encontrar una línea o una fórmula que mejor describa cómo una variable (la que se conoce) afecta a la otra (la que se quiere predecir). Esta fórmula se basa en los datos que que existen y permite hacer predicciones sobre nuevos datos.

A continuación, se muestra el proceso para estudiar la producción de trigo y algodón considerando superficie cultivada y producción buscando detactar relaciones entre los valores.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os
from leer import leer_csv

# Función para mostrar los datos y la línea de regresión
def mostrar(df, X, Y, P, titulo):
    plt.scatter(df[X], df[Y], color='blue', label='Datos')
    plt.plot(df[X], df[P], color='red', label='Recta de regresión')
    plt.xlabel(X)
```

```
    plt.ylabel(Y)
    plt.legend()
    plt.title(titulo)
    plt.show()

# Función para leer los datos de trigo y algodón
def leer_datos():
    df = leer_csv('prodhistorico.csv')
    return df

# Función para mostrar los gráficos de regresión en un solo panel
def mostrar_en_panel(df1, X1, Y1, P1, titulo1, df2, X2, Y2, P2, titulo2):
    script_dir = os.path.dirname(os.path.abspath(__file__))
    fig, ax = plt.subplots(1, 2, figsize=(12, 6))  # dos gráficos en un panel horizontal

    # Gráfico para el primer dataset (trigo)
    ax[0].scatter(df1[X1], df1[Y1], color='blue', label='Datos')
    ax[0].plot(df1[X1], df1[P1], color='red', label='Recta de regresión')
    ax[0].set_xlabel(X1)
    ax[0].set_ylabel(Y1)
    ax[0].legend()
    ax[0].set_title(titulo1)

    # Gráfico para el segundo dataset (algodón)
    ax[1].scatter(df2[X2], df2[Y2], color='blue', label='Datos')
    ax[1].plot(df2[X2], df2[P2], color='red', label='Recta de regresión')
    ax[1].set_xlabel(X2)
    ax[1].set_ylabel(Y2)
    ax[1].legend()
    ax[1].set_title(titulo2)

    plt.tight_layout()  # Ajustar el espacio entre los gráficos

    plt.savefig(os.path.join(script_dir, 'algodon.png'))
    plt.show()

# Función para calcular la regresión y correlación
def correlacion(df, X, Y, predic):
    # Calcular la pendiente y la intersección de la línea de regresión
    m, i = np.polyfit(df[X], df[Y], 1)
    # Añadir columna de predicción al dataframe
    df[predic] = i + m * df[X]
    # Calcular R^2
    residual = ((df[Y] - df[predic]) ** 2).sum()
    total = ((df[Y] - df[Y].mean()) ** 2).sum()
    r2 = 1 - (residual / total)
    return r2, m, i

# Leer los datos
datos = leer_datos()

# Separar los datos por productos
trigo = datos[datos['producto'] == 'trigo']
algodon = datos[datos['producto'] == 'algodón']

# Calcular regresión para trigo
r2_trigo, pendiente_trigo, interseccion_trigo = correlacion(trigo, \
    'superficie.Ha', 'produccion.T', 'prediccion_trigo')

# Calcular regresión para algodón
r2_algodon, pendiente_algodon, interseccion_algodon = correlacion(algodon, \
     'superficie.Ha', 'produccion.T', 'prediccion_algodon')

# Mostrar los dos gráficos en un mismo panel, uno al lado del otro
mostrar_en_panel(trigo, 'superficie.Ha', 'produccion.T', 'prediccion_trigo', \
    'Regresión para Trigo', algodon, 'superficie.Ha', \
    'produccion.T', 'prediccion_algodon', 'Regresión para Algodón')

# Calcular la correlación de Pearson para ambos productos
pearson_trigo = trigo['superficie.Ha'].corr(trigo['produccion.T'])
pearson_algodon = algodon['superficie.Ha'].corr(algodon['produccion.T'])

# Mostrar coeficiente de correlación y detalles de la regresión para ambos productos
```

```
print(f'Trigo - Coeficiente de correlación de Pearson: {pearson_trigo:.2f}')
print(f'Trigo - Pendiente : {pendiente_trigo:.2f}')
print(f'Trigo - Intersección : {interseccion_trigo:.2f}')
print(f'Trigo - r^2: {r2_trigo:.2f}')

print(f'Algodón - Coeficiente de correlación de Pearson: {pearson_algodon:.2f}')
print(f'Algodón - Pendiente : {pendiente_algodon:.2f}')
print(f'Algodón - Intersección : {interseccion_algodon:.2f}')
print(f'Algodón - r^2: {r2_algodon:.2f}')
```

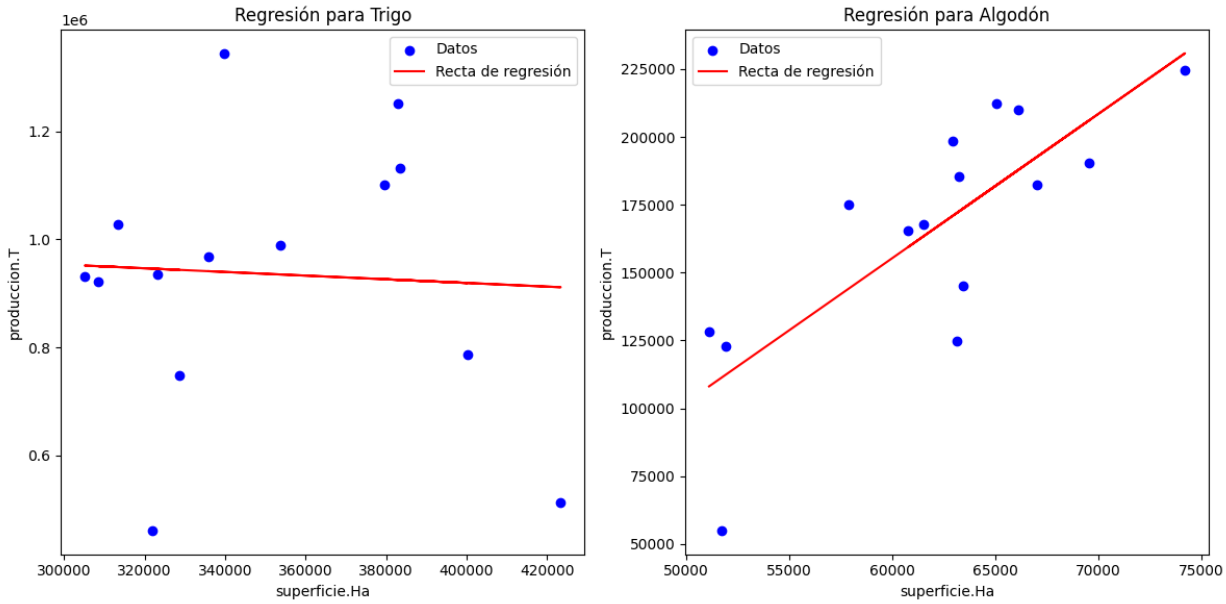

Figura 47: Modelo de regresión