



PROGRAMACIÓN BÁSICA

Osvaldo Habib González González

Universidad Autónoma de Nuevo León - Facultad de Ciencias Físico Matemáticas

Programación básica

Fecha de última modificación: 10 de abril de 2025

Fase I: Fundamentos de Programación	8
1. Conceptos básicos de programación	8
Definición de programación.	8
La historia de la programación: De los primeros lenguajes a Python	8
Lenguajes de programación: interpretados vs compilados.	8
Lenguajes de marcado y su rol en la programación.	9
Partes esenciales de la programación en una computadora	9
Tipos y clasificación de los lenguajes de programación	9
Conceptos actuales: Front-end, Back-end y Full Stack	9
La relación entre metodología de programación y desarrollo de código	10
2. Introducción a la línea de comandos	12
Navegación en la línea de comandos (comandos básicos).	12
Instrucciones para la instalación de Python en Windows	12
Problemas comunes y soluciones	14
Ejecución de programas en Python desde la línea de comandos.	14
Diferencias entre la línea de comandos y entornos gráficos.	15
3. Comenzando con Python	15
El REPL de Python	15
Programación utilizando el bloc de notas y la línea de comandos.	16
Generalidades de Python	17
Primer programa en Python: estructura y ejecución desde la línea de comandos.	18
4. Entrada y salida de datos en Python	18
Uso de la función input() para recibir datos del usuario.	19
Diferentes formas de utilizar la función print()	19
Concatenación de cadenas y visualización de datos en la salida.	22
5. Variables, constantes y operaciones básicas	24
Declaración de variables y constantes en Python.	25
Tipos de datos en Python: enteros, flotantes, cadenas.	25
Obtención del tipo de dato de una variable con type().	26
Conversión de tipos de datos (int(), float(), str()).	26
Operadores aritméticos, lógicos y de comparación.	26
6. Estructuras de control: Condicionales y ciclos	29
Uso de condicionales (if, else, elif) para tomar decisiones.	29
Estructuras de repetición: ciclos for y while.	30
Ciclos anidados y control de flujo avanzado (uso de break, continue).	31
Palabras reservadas en Python	32
7. Introducción a las palabras reservadas y su significado en Python.	33
Reglas de sintaxis y buenas prácticas en la escritura de código.	33
Introducción a listas en Python	34
Definición de listas, cómo crear y modificar listas.	34
Operaciones básicas con listas: agregar, eliminar, buscar elementos.	35

Fase II: Funciones, Estructuras de Datos, Algoritmos y Paradigmas de Programación	38
1. Funciones.....	38
Definición y creación de funciones en Python.	38
Paso de argumentos y retorno de valores.	39
Uso de funciones lambda para tareas simples.	40
Modularización del código a través de funciones.	42
2. Estructuras de datos.....	46
Clasificación de las Estructuras de Datos	46
Definición de tuplas y su uso en Python.	47
Operaciones básicas con conjuntos y su uso en manipulación de datos únicos.	48
Diccionarios: creación, acceso a pares clave-valor, y manipulación.	49
3. Estructuras de datos anidadas	51
Anidación de listas, diccionarios y otras estructuras de datos.	51
Tipos Comunes de Anidación	51
Manipulación y acceso a datos en estructuras anidadas.	52
4. Recursividad	55
5. Divide y vencerás	56
6. Algoritmos de ordenamiento.....	57
Ordenamiento por Burbuja (Bubble Sort)	57
Ordenamiento por Inserción (Insertion Sort)	59
Ordenamiento por mezcla o fusión (Merge sort).	60
Puntos importantes a tomar en cuenta:	60
Ordenamiento rápido (Quick sort).	61
7. Algoritmos de búsqueda.....	62
Búsqueda secuencial y su implementación.	62
Búsqueda binaria y su optimización.	63
8. Paradigmas de programación.....	64
Programación Imperativa	64
Programación Estructurada	64
Programación Modular	65
Programación Orientada a Objetos (POO)	66
Programación Multihilos	67
Multiparadigma:	69

Fase III: Aplicación de la Programación en Ciencia y Tecnología	71
1. Automatización de tareas con Python	71
Scripts para tareas web y uso de lenguajes de marcado (HTML, XML).	72
2. Manejo de archivos	72
Lectura y escritura de archivos en Python.	73
Modos de apertura comunes:	73
Ejemplo: Leer un archivo línea por línea	73
Manejo de errores y excepciones al manipular archivos.	74
3. Expresiones regulares y manejo de texto	74
Búsqueda y manipulación de cadenas utilizando expresiones regulares.	75
¿Qué son los patrones en expresiones regulares?	77
4. APIs y formato JSON	81
Introducción a las APIs y cómo interactuar con ellas utilizando Python.	81
Manejo de datos en formato JSON para intercambio de información.	81
Cargar y trabajar con JSON en Python	82
JSON y APIs: ¿cómo se conectan?	82
5. Matemáticas con Python	83
Cálculos con fracciones y números complejos.	83
Números complejos	83
Aplicación de métodos numéricos con statistics.	83
Operaciones simbólicas con SymPy.	84
6. Manejo de hojas de cálculo con Openpyxl	85
Lectura, manipulación y creación de archivos Excel con módulo Openpyxl.	85
7. Gráficas con Matplotlib y SymPy	86
Partes de una Gráfica según estándares internacionales	86
Creación de gráficos simples y avanzados con Matplotlib.	87
Graficación de fórmulas matemáticas utilizando SymPy.	88

Programación básica

¿Te imaginas tener una herramienta tan versátil que te permita no solo aprender a programar desde cero, sino también abordar problemas complejos en tus estudios y, eventualmente, en tu campo profesional? Python es exactamente eso: un lenguaje de programación poderoso y fácil de aprender, que te brindará las bases para desarrollar soluciones eficientes en áreas tan diversas como las matemáticas, la física, la ciberseguridad, la multimedia y muchas más.

¿Qué hace a Python tan especial?

Python ha ganado una enorme popularidad entre estudiantes y profesionales por varias razones clave:

Simplicidad y legibilidad: Su sintaxis es sencilla y directa, lo que lo convierte en una excelente opción para quienes están comenzando en el mundo de la programación. En Python, el código es casi como leer una frase en inglés. Esto facilita que te concentres en la lógica y no te pierdas en los detalles técnicos.

Versatilidad en aplicaciones: Ya sea que estés desarrollando un modelo matemático, simulaciones físicas, herramientas de análisis financiero, o incluso trabajando en seguridad cibernética, Python tiene bibliotecas y herramientas que permiten adaptar el lenguaje a prácticamente cualquier necesidad. ¡Lo que aprendas hoy te servirá mañana en el laboratorio, en la oficina o en proyectos personales!

Ampliamente utilizado en la industria: Compañías tecnológicas como Google, Netflix y Facebook usan Python para desarrollar aplicaciones y servicios. En el ámbito académico, es fundamental para la investigación en ciencia de datos, inteligencia artificial, física computacional y muchas otras disciplinas. Al aprender Python, no solo estarás desarrollando habilidades prácticas, sino también preparándote para un mundo laboral en el que este lenguaje es clave.

¿Quién creó Python y por qué?

La historia detrás de Python es tan interesante como el propio lenguaje. Fue creado a finales de los años 80 por Guido van Rossum, un programador holandés que, como un verdadero visionario, tenía el objetivo de crear un lenguaje que fuera fácil de aprender y usar, pero lo suficientemente potente para resolver problemas complejos. Lo curioso es que el nombre de Python no proviene de una serpiente, como muchos piensan, sino de un programa de comedia británico llamado Monty Python's Flying Circus, del cual Van Rossum era fanático. Este detalle refleja parte del humor y la accesibilidad que el creador quiso imprimir en el lenguaje: Python no es rígido ni complicado, es amigable y flexible.

Pero eso no es todo. Van Rossum no buscaba solamente crear otro lenguaje de programación. Quería ofrecer una herramienta que facilitara la colaboración entre diferentes disciplinas, algo que hoy vemos claramente con su uso en áreas tan variadas como la ciencia, la ingeniería, el análisis financiero y las artes visuales.

Python en tu carrera

Dependiendo de tu carrera, el impacto de aprender Python será inmediato:

- **Matemáticos y Físicos:** Python te permite desarrollar simulaciones y modelar fenómenos complejos con facilidad. ¿Necesitas resolver ecuaciones diferenciales o analizar grandes volúmenes de datos experimentales? Python es ideal para automatizar esos cálculos.
- **Actuaría:** Si te dedicas a la evaluación de riesgos y análisis financiero, Python se convertirá en tu aliado. Su capacidad para manejar grandes cantidades de datos y realizar análisis estadísticos te ayudará a diseñar modelos precisos y eficientes.
- **Ciencias Computacionales y Seguridad en TI:** Python es el lenguaje por excelencia para los profesionales de la tecnología. Es el pilar de muchas herramientas de ciberseguridad, pruebas de vulnerabilidades y desarrollo de software a gran escala. Además, su soporte para paradigmas de programación como el orientado a objetos o multiparadigma te permitirá construir soluciones escalables y seguras.
- **Multimedia y Animación Digital:** Si te interesa el desarrollo de efectos visuales, la animación o la creación de videojuegos, Python tiene aplicaciones en el diseño de herramientas personalizadas, automatización de procesos de producción y la integración con motores gráficos como Blender.

El primer paso hacia grandes logros

Python te abrirá las puertas a un universo de posibilidades. En este curso, aprenderás desde los fundamentos de la programación, sin depender de herramientas complejas, hasta cómo aplicar tu conocimiento para resolver problemas reales en tu disciplina. A lo largo del camino, descubrirás que Python no es solo un lenguaje, sino una mentalidad: te entrenará para pensar como un programador, analizar problemas con precisión y construir soluciones eficientes.

Y si alguna vez te preguntas cómo un lenguaje tan simple puede ser tan poderoso, recuerda el pensamiento visionario de su creador, Guido van Rossum, quien dijo que Python debía ser tan comprensible que los niños pudieran usarlo, pero tan potente que los profesionales más experimentados confiaran en él para sus proyectos más ambiciosos.

¡Este es tu momento! Con Python en tu caja de herramientas, estarás un paso más cerca de convertirte en un profesional que domina tanto la teoría como la práctica. ¿Listo para sumergirte en un lenguaje que, como su nombre lo indica, tiene un toque de humor, pero es capaz de resolver problemas tan grandes como los que enfrentarás en tu futuro?

Fase I: Fundamentos de Programación

Objetivo de la Fase: Que el estudiante comprenda los conceptos básicos de la programación, aprenda a utilizar la línea de comandos y desarrolle programas simples en Python utilizando el bloc de notas.

Bienvenidos al fascinante mundo de la programación. Al comenzar esta fase, nos adentraremos en los cimientos de la programación: comprenderemos desde lo que significa programar y su impacto en diversas disciplinas hasta los principios básicos para escribir y ejecutar nuestros propios programas en Python. La meta es que, al final de esta fase, cada estudiante pueda utilizar herramientas básicas como la línea de comandos, comprender el papel del bloc de notas en el desarrollo de código y escribir programas simples en Python.

1. Conceptos básicos de programación

Definición de programación.

La programación es el proceso mediante el cual damos instrucciones a una computadora para que realice tareas específicas. Imagina que la computadora es un "ejecutor" que no puede pensar por sí mismo; sin embargo, con instrucciones claras y bien estructuradas, podemos hacer que siga procesos y resuelva problemas de diversa índole. A través de la programación, podemos desarrollar desde aplicaciones móviles y páginas web hasta simulaciones científicas y modelos financieros.

La historia de la programación: De los primeros lenguajes a Python

El camino de la programación comenzó en los años 40, con las primeras computadoras y lenguajes de bajo nivel como el ensamblador, que permitían a los científicos e ingenieros comunicarse directamente con las máquinas usando instrucciones binarias. En los años 50 y 60, surgieron lenguajes de alto nivel como Fortran y COBOL, los cuales simplificaron esta interacción y se convirtieron en herramientas clave para científicos e ingenieros.

Con los años, la programación evolucionó rápidamente. Surgieron lenguajes como C, que aportó velocidad y versatilidad; posteriormente, lenguajes como Java, que introdujo el paradigma orientado a objetos, y Python, creado en 1989 por Guido van Rossum, que se caracteriza por su sencillez y legibilidad. Hoy en día, Python es fundamental en áreas como ciencia de datos, inteligencia artificial, desarrollo web y muchas otras disciplinas.

Lenguajes de programación: interpretados vs compilados.

Los lenguajes de programación se dividen en dos tipos principales:

- Lenguajes interpretados: Como Python, JavaScript y Ruby, son lenguajes que se ejecutan a través de un intérprete, línea por línea, durante la ejecución del programa. Esto permite una mayor flexibilidad y facilidad para corregir errores, pero puede ser más lento en comparación con los compilados.
- Lenguajes compilados: Como C, C++ y Java, son lenguajes que se traducen completamente en código máquina (binario) antes de su ejecución mediante un proceso llamado compilación. Esto hace que sean más rápidos y eficientes en ejecución, aunque la corrección de errores es menos inmediata.

Lenguajes de marcado y su rol en la programación.

Además de los lenguajes de programación, existen los lenguajes de marcado, como HTML y XML, que se utilizan para estructurar y organizar información dentro de un documento o una página web, en lugar de ejecutar operaciones o procesos. Aunque no son lenguajes de programación en sí, son fundamentales en el desarrollo web y en la comunicación entre sistemas.

Partes esenciales de la programación en una computadora

Para que un programa funcione, necesita estar respaldado por varios componentes esenciales:

- CPU (Unidad Central de Procesamiento): La "mente" de la computadora, que interpreta y ejecuta las instrucciones.
- Memoria: Donde se almacenan temporalmente los datos y las instrucciones mientras el programa está en ejecución.
- Almacenamiento y Sistema Operativo: Permiten guardar los programas y datos, y brindar un entorno donde el software se ejecute.

Tipos y clasificación de los lenguajes de programación

Los lenguajes de programación se pueden clasificar en varias categorías según sus características y enfoques:

Lenguajes de bajo nivel: Como ensamblador, están cerca del lenguaje de máquina, siendo rápidos y eficientes, pero complejos de programar.

Lenguajes de alto nivel: Como Python, Java y C++, que están diseñados para ser más accesibles para los programadores, abstrayendo detalles de la máquina.

Paradigmas de programación: Como el orientado a objetos (Python, Java), el funcional (Haskell), y el procedural (C), cada uno con su propio enfoque para resolver problemas.

Conceptos actuales: Front-end, Back-end y Full Stack

Hoy en día, la programación también se clasifica según el tipo de desarrollo que realiza el programador:

- Front-end: El desarrollo que interactúa directamente con los usuarios, principalmente mediante lenguajes como HTML, CSS y JavaScript, aplicados en el desarrollo de interfaces web y móviles.
- Back-end: Es la programación que se encarga de la "lógica" y el procesamiento en el servidor. Aquí se manejan bases de datos, cálculos y operaciones que el usuario no ve, empleando lenguajes como Python, Java y PHP.
- Full Stack: Los programadores "full stack" combinan tanto front-end como back-end, construyendo aplicaciones completas que interactúan entre el usuario y el servidor.

La relación entre metodología de programación y desarrollo de código

Para programar eficientemente, es importante tener un plan. Aquí es donde entran en juego los algoritmos y los diagramas de flujo. Cada uno de estos conceptos nos ayuda a planificar y organizar los pasos lógicos antes de escribir el código. Los algoritmos nos proporcionan una lista clara de pasos a seguir; y los diagramas de flujo son representaciones gráficas de estos pasos. Con estas herramientas, escribir el código en Python o cualquier otro lenguaje se vuelve mucho más sencillo y eficiente.

Ejemplo: Queremos desarrollar un programa en Python que determine si un número es par o impar. Para lograr esto de manera eficiente, utilizaremos un algoritmo y un diagrama de flujo antes de escribir el código final.

Nombre del Algoritmo: DETERMINAR_PARIDAD_NUMERO

Declaración de variables

NUMERO: Variable de tipo entero. Almacena el número ingresado.

1. Inicio
2. Leer NUMERO
3. Si $\text{NUMERO} \bmod 2 == 0$, entonces:
4. Mostrar "El número es par."
5. Si no
6. Mostrar "El número es impar."
7. Fin del algoritmo

Algoritmo

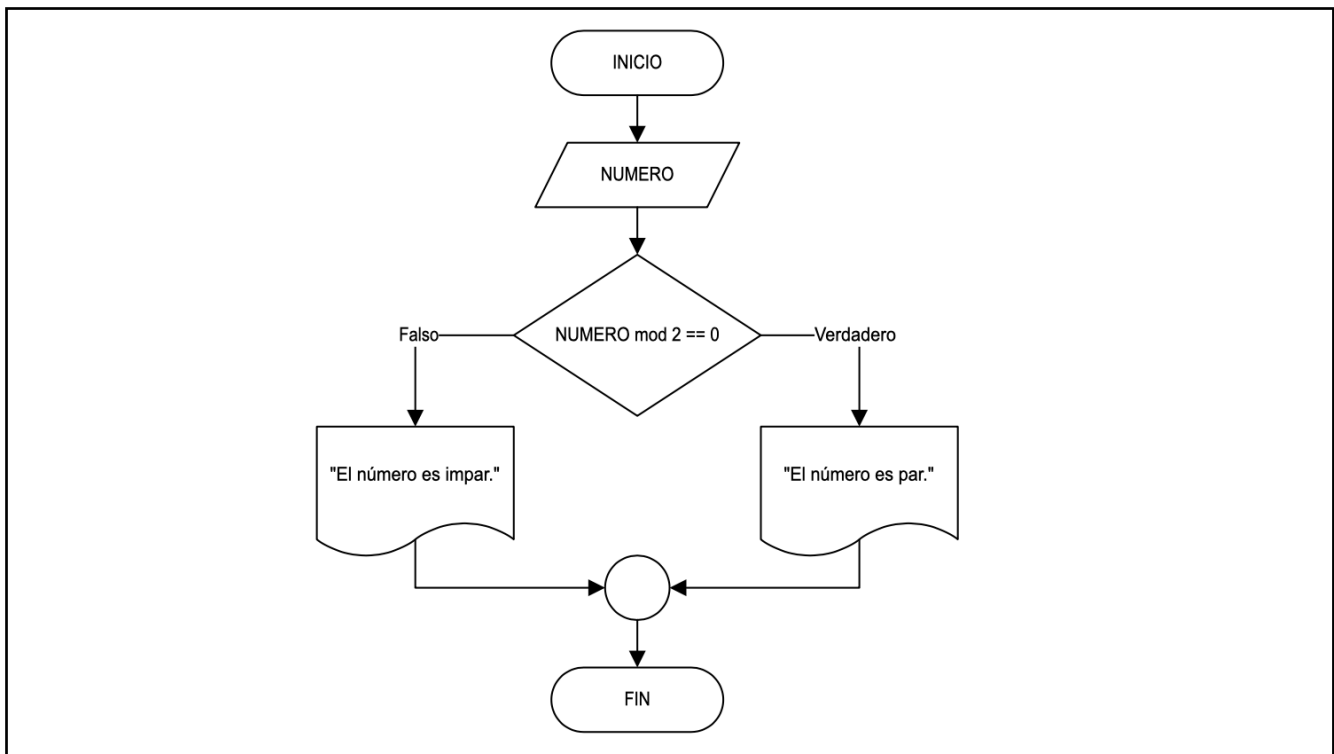


Diagrama de flujo

```
1 # DETERMINAR_PARIDAD_NUMERO
2 # Código para verificar si un número es par o impar
3
4 # Solicitar un número
5 numero = int(input("Ingresa un número: "))
6
7 # Verificar si el número es par o impar
8 if numero % 2 == 0:
9     print("El número es par")
10 else:
11     print("El número es impar")
```

Código escrito en python.

Gracias a esta planificación de desarrollar algoritmo y diagrama de flujo, sabemos exactamente qué hará cada línea del código, ya que lo hemos estructurado paso a paso. Este enfoque es fundamental para escribir código más claro, eficiente y fácil de depurar, ya que nos permite anticiparnos a posibles errores y verificar que la lógica es correcta antes de implementar el código en un lenguaje específico.

2. Introducción a la línea de comandos

En esta fase, aprenderemos a movernos con soltura en la línea de comandos para ejecutar programas en Python y conoceremos algunos comandos esenciales que nos facilitarán navegar por las carpetas de nuestro sistema y realizar acciones básicas. Esto les permitirá comprender cómo interactuar con una computadora en un nivel más técnico, usando únicamente texto, y verán la diferencia entre esta modalidad y el uso de entornos gráficos.

Navegación en la línea de comandos (comandos básicos).

La línea de comandos, también conocida como "símbolo del sistema" en Windows o "Terminal" en macOS y Linux, es una herramienta fundamental para cualquier programador. Esta interfaz de texto permite comunicarnos con el sistema operativo escribiendo comandos en lugar de utilizar un ratón o iconos gráficos. Aunque puede parecer intimidante al principio, aprender a manejarse en la línea de comandos les abrirá la puerta a un mayor control y velocidad en el desarrollo de software.

A continuación, exploraremos algunos de los comandos básicos que facilitarán su navegación y gestión de archivos en la línea de comandos:

- `cd` (change directory): Permite cambiar de directorio. Por ejemplo, `cd CarpetaEjemplo` nos lleva a la carpeta "CarpetaEjemplo". Para volver al directorio anterior, pueden usar `cd ...`.
- `dir` (directory): El comando `dir` muestra una lista de archivos y carpetas en el directorio actual. También se puede usar para mostrar archivos en un directorio específico.
- `mkdir` (make directory): Crea un nuevo directorio. Por ejemplo, `mkdir ProyectosPython` crea una carpeta llamada "ProyectosPython".
- `rm` (remove): Elimina archivos. Es importante tener cuidado con este comando, especialmente con el parámetro `-r`(recursive), que permite borrar carpetas completas. Ejemplo: `rm archivo.txt` o `rm -r CarpetaEjemplo`.

Estos comandos son básicos, pero dominar su uso les permitirá realizar una amplia variedad de tareas que facilitarán el manejo de sus archivos y carpetas de proyecto en Python.

Instrucciones para la instalación de Python en Windows

Sigue estos pasos para instalar Python en tu computadora con Windows y configurarlo adecuadamente.

1. Descargar Python

- i. Abre tu navegador web y visita la página oficial de Python: <https://www.python.org>.
- ii. Haz clic en el botón de descarga que aparece en la página principal. Por lo general, detectará automáticamente la versión adecuada para tu sistema operativo (Windows).
- iii. Descarga el instalador ejecutable (archivo .exe) para Windows.

2. Iniciar el instalador

- i. Ubica el archivo descargado (python-X.X.X.exe) y ábrelo para iniciar el proceso de instalación.
- ii. En la pantalla inicial, marca las siguientes opciones antes de continuar:
 - "Add Python to PATH" (Agregar Python a las variables de entorno).
 - "Install launcher for all users (recommended)" (Instalar el lanzador para todos los usuarios).

3. Personalizar la instalación

- i. Haz clic en "Customize installation" (Instalación personalizada).
- ii. En la ventana de personalización:
 - Documentation (Documentación).
 - pip (Gestor de paquetes de Python).
 - tcl/tk y IDLE (para usar interfaces gráficas y el editor básico de Python).
 - Python test suite (Opcional para pruebas avanzadas).
- iii. Haz clic en Next.

4. Configuración avanzada

- i. En la siguiente ventana, selecciona las siguientes opciones:
 - "Install for all users" (Instalar para todos los usuarios).
 - "Associate files with Python (.py)" (Asociar archivos con extensión .py).
 - "Create shortcuts for installed applications" (Crear accesos directos para las aplicaciones instaladas).
 - "Add Python to environment variables" (Agregar Python a las variables de entorno).
- ii. Selecciona la ubicación de instalación. Por defecto, se instalará en C:\Program Files\PythonXX.

5. Finalizar la instalación

- i. Haz clic en Install para iniciar la instalación.
- ii. Espera a que finalice el proceso. Esto puede tomar unos minutos.
- iii. Una vez terminado, verás un mensaje indicando que la instalación fue exitosa. Haz clic en Close.

6. Verificar la instalación

- i. Abre la terminal de comandos de Windows:
 - Presiona Win + R, escribe cmd y presiona Enter.
- ii. Escribe los siguientes comandos para verificar que Python y pip están instalados correctamente:
 - `python --version`
 - `pip --version`
- iii. Para verificar que el lanzador de Python (py launcher) está funcionando:
 - Escribe py en la terminal y presiona Enter.
 - Deberías entrar al entorno interactivo de Python (REPL), donde puedes escribir instrucciones como `print("Hola, Python")`.

Problemas comunes y soluciones

- Si el comando python no funciona, asegúrate de que "Add Python to PATH" estaba seleccionado durante la instalación.
- Si necesitas ajustar las variables de entorno manualmente:
 1. Ve a Configuración > Sistema > Acerca de > Configuración avanzada del sistema.
 2. En la pestaña Opciones avanzadas, haz clic en Variables de entorno.
 3. Busca la variable Path, editala y agrega la ruta de la instalación de Python (por ejemplo, C:\Program Files\PythonXXy C:\Program Files\PythonXX\Scripts).

Ejecución de programas en Python desde la línea de comandos.

Una vez que han creado su primer programa en Python (por ejemplo, un archivo llamado `mi_programa.py`), pueden ejecutarlo directamente desde la línea de comandos. Esto les permitirá ver cómo se ejecuta su código en tiempo real y verificar si todo funciona como esperaban.

Para ejecutar un programa en Python desde la línea de comandos, sigan estos pasos:

1. Naveguen hasta el directorio donde guardaron el archivo usando `cd`. Si el archivo está en una carpeta llamada "ProyectosPython", escriban `cd ProyectosPython`.
2. Escriban `python mi_programa.py` y presionen Enter. Esto ejecutará el archivo, y si el programa contiene instrucciones de salida como `print()`, verán los resultados en la línea de comandos.
3. Es importante notar que, en algunos sistemas, el comando para Python puede ser `python3` en lugar de `python`, dependiendo de la versión instalada en su computadora.

Diferencias entre la línea de comandos y entornos gráficos.

Al iniciar en la programación, muchos estudiantes están acostumbrados a los entornos gráficos (como el Explorador de Windows o Finder en macOS), donde pueden navegar por carpetas y archivos usando un ratón y clics. Sin embargo, la línea de comandos, aunque más simple visualmente, ofrece varias ventajas para los programadores:

- **Velocidad:** Con la línea de comandos, los programadores pueden realizar tareas de manera más rápida y con menos clics, lo que es especialmente útil cuando necesitan manejar múltiples archivos o ejecutar comandos repetitivos.
- **Precisión y control:** La línea de comandos ofrece un control directo sobre los archivos y el sistema, permitiendo opciones avanzadas que muchas veces no están disponibles en los entornos gráficos.
- **Automatización:** Muchos procesos pueden automatizarse fácilmente en la línea de comandos usando scripts, lo que es ideal para tareas repetitivas.
- **Versatilidad:** La línea de comandos es una herramienta universal entre diferentes sistemas operativos. Aunque cada sistema tiene ligeras variaciones en sus comandos, el conocimiento adquirido es aplicable en macOS, Linux y Windows, facilitando el trabajo en distintas plataformas.

Aunque en este curso utilizaremos el bloc de notas para editar el código y la línea de comandos para ejecutar los programas en Python, conocerán también algunas características de los entornos de desarrollo (o IDEs, como PyCharm, VS Code y Jupyter Notebook). Estos ofrecen herramientas gráficas avanzadas y son ideales para desarrolladores que buscan escribir, depurar y ejecutar código en un solo lugar. Sin embargo, aprender los fundamentos en la línea de comandos será de gran utilidad, ya que tendrán un control más profundo de su entorno de desarrollo, independientemente de la herramienta que elijan en el futuro.

3. Comenzando con Python

En esta etapa programaremos en un entorno básico usando el bloc de notas y la línea de comandos para escribir y ejecutar nuestros programas. Esta metodología les ayudará a conocer las bases de la programación sin necesidad de depender de herramientas avanzadas, entendiendo cada paso del proceso de codificación.

El REPL de Python

El REPL (Read-Eval-Print Loop, por sus siglas en inglés) es un entorno interactivo que Python ofrece para escribir y ejecutar código en tiempo real. Este entorno permite escribir una instrucción, evaluarla inmediatamente, mostrar el resultado y repetir el proceso. Es especialmente útil para pruebas rápidas, explorar conceptos de programación o realizar cálculos simples.

¿Para qué sirve el REPL de Python?

- Pruebas rápidas: Permite experimentar con fragmentos de código sin necesidad de escribir un programa completo.
- Exploración interactiva: Es ideal para probar cómo funcionan ciertas funciones o métodos.
- Depuración inicial: Ayuda a verificar rápidamente pequeños errores o probar soluciones inmediatas.
- Aprendizaje de conceptos básicos: Los principiantes pueden familiarizarse con la sintaxis de Python y obtener resultados inmediatos.

Aunque el REPL es una herramienta poderosa, tiene varias limitaciones que pueden afectar el aprendizaje estructurado de programación:

- Falta de persistencia del código: El código escrito en el REPL no se guarda automáticamente, lo que dificulta reutilizarlo o revisarlo posteriormente.
- Desorganización del código: El REPL no fomenta la creación de programas bien estructurados; los estudiantes pueden adquirir el hábito de escribir código desordenado.
- Dificultad para trabajar con proyectos complejos: No es adecuado para programas extensos o proyectos que requieran múltiples módulos y archivos.
- Escasa enseñanza de buenas prácticas: El REPL no promueve el desarrollo de habilidades fundamentales, como organizar el código en funciones, comentar adecuadamente o utilizar convenciones estándar.

En esta unidad, el objetivo es que los estudiantes no solo aprendan a programar, sino que también desarrollen una comprensión sólida de las bases de la programación estructurada. Usar el bloc de notas y la línea de comandos tiene varias ventajas:

- Fomenta una escritura más estructurada: Al escribir programas en el bloc de notas, los estudiantes aprenden a organizar el código, utilizar funciones y escribir comentarios claros.
- Experiencia con entornos reales: La línea de comandos es una herramienta esencial en el desarrollo de software profesional. Enseñar su uso desde el principio ayuda a los estudiantes a desarrollar habilidades prácticas que les serán útiles en su carrera.
- Facilita la transición a proyectos más complejos: Trabajar fuera del REPL permite comprender cómo se crean, guardan y ejecutan programas de manera independiente, un paso crucial antes de trabajar en entornos de desarrollo integrados (IDE).
- Fortalece la disciplina en la programación: Al no contar con ayudas visuales ni correcciones automáticas, los estudiantes desarrollan la habilidad de identificar errores manualmente y depurar su código de forma más efectiva.

Programación utilizando el bloc de notas y la línea de comandos.

Para nuestros primeros programas, usaremos una herramienta simple: el bloc de notas. Este editor de texto básico permite escribir y guardar código sin distracciones ni características avanzadas. Al utilizar el bloc de notas, ustedes aprenderán a enfocarse en la lógica y la estructura de su código.

Una vez que hayan escrito su código en el bloc de notas, lo guardarán con la extensión .py (por ejemplo, mi_programa.py). Este archivo contiene el código que Python interpretará cuando lo ejecuten en la línea de comandos.

La línea de comandos, también conocida como "símbolo del sistema" o "terminal", es un lugar donde pueden ejecutar instrucciones directamente. Para ejecutar su programa en Python desde la línea de comandos, navegarán a la ubicación del archivo con el comando cd (por ejemplo, cd Documentos\Python en Windows o cd ~/Documentos/Python en macOS y Linux). Una vez allí, escribirán python mi_programa.py (o python3 en algunas versiones) y el sistema ejecutará el código, mostrándoles el resultado en la misma ventana.

Generalidades de Python

¿Qué es un entorno de desarrollo integrado (IDE)? ¿Para qué sirven? ¿Qué ventajas tienen? Mención de los más utilizados (VSCode, PyCharm, Jupyter).

A medida que avancen, probablemente notarán que escribir y ejecutar código en el bloc de notas y la línea de comandos, aunque efectivo, puede volverse algo limitado. Aquí es donde entran los entornos de desarrollo integrados, o IDEs (por sus siglas en inglés). Un IDE es un programa que reúne todas las herramientas necesarias para el desarrollo de software en un solo lugar, facilitando la escritura, depuración y ejecución del código.

¿Cuáles son las ventajas de usar un IDE?

Los IDEs ofrecen numerosas ventajas, entre ellas:

- Comodidad y rapidez: Un IDE incluye características como autocompletado, resaltado de sintaxis, y gestión de archivos, lo que facilita la escritura del código.
- Depuración: Los IDEs permiten identificar y corregir errores de forma rápida con herramientas integradas de depuración.
- Ejecutar y probar: Permiten ejecutar y probar el código directamente dentro del mismo entorno, simplificando el flujo de trabajo.

Algunos de los IDEs más populares para Python incluyen:

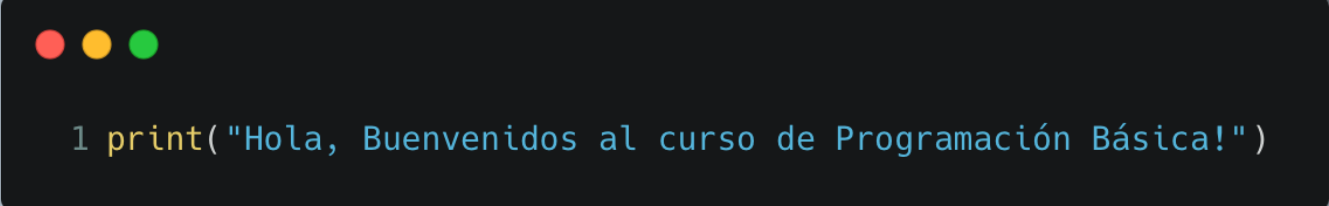
- Visual Studio Code (VSCode): Es un editor de código gratuito y personalizable, compatible con muchas extensiones para Python y otros lenguajes.
- PyCharm: Es un IDE específico para Python, con herramientas avanzadas para desarrolladores que trabajan en proyectos grandes.
- Jupyter Notebook: Muy utilizado en áreas como la ciencia de datos, permite escribir y ejecutar código en "celdas", lo que es ideal para proyectos de análisis de datos y visualización.

En este curso aprenderán a programar sin depender de un IDE, lo cual les dará una base sólida, aunque conocerán estos entornos y sus ventajas para el momento en que decidan usarlos.

Primer programa en Python: estructura y ejecución desde la línea de comandos.

Para comenzar, escribamos un programa sencillo que salude al usuario. Este es un buen primer paso para familiarizarse con la sintaxis básica de Python y la ejecución de programas desde la línea de comandos.

Abre el bloc de notas y escriban lo siguiente:



```
1 print("Hola, Bienvenidos al curso de Programación Básica!")
```

Código escrito en python.

Este comando `print()` le dice a Python que muestre el mensaje entre comillas en pantalla.

Guarden el archivo: Guarden el archivo con la extensión `.py`, por ejemplo, `primer_programa.py`. Recuerden elegir la ubicación donde lo guardarán para facilitar su ejecución posterior.

Ejecuten el programa desde la línea de comandos:

Abran la línea de comandos y usen el comando `cd` para navegar a la ubicación donde guardaron su archivo.

Una vez en la carpeta correcta, escriban `python primer_programa.py` y presionen Enter.

Si todo está correcto, verán en pantalla el mensaje ¡Hola, bienvenidos al curso de Programación en Python!. ¡Felicidades, han ejecutado su primer programa en Python!

A medida que avanzan, verán que programar en Python desde un entorno simple como el bloc de notas y la línea de comandos les permite comprender mejor los procesos de compilación y ejecución. Con el tiempo, podrán explorar IDEs que acelerarán su flujo de trabajo, pero este conocimiento básico les brindará las bases para moverse en cualquier entorno. Al familiarizarse con estos conceptos y herramientas iniciales, están construyendo una base sólida para seguir avanzando en su camino como programadores. ¡Adelante!

4. Entrada y salida de datos en Python

Conocerán conceptos básicos pero esenciales sobre cómo interactuar con un programa a través de la entrada y salida de datos. Este conocimiento les permitirá diseñar programas que soliciten información al usuario, la procesen y muestren resultados en pantalla. Al comprender cómo utilizar funciones como `input()` y `print()`, podrán crear programas más dinámicos y atractivos.

Uso de la función input() para recibir datos del usuario.

En Python, la función input() es la herramienta principal para recibir datos directamente del usuario. Cada vez que usamos input(), el programa se detiene y espera que el usuario ingrese algún dato. Por ejemplo:

```
1 nombre = input("Por favor, ingresa tu nombre:")
2 print("¡Hola, " + nombre + "!")
```

Código escrito en python.

En este ejemplo, el mensaje "Por favor, ingresa tu nombre: " aparece en pantalla y espera una respuesta. Cuando el usuario ingresa un valor, Python lo almacena en la variable nombre. Luego, al ejecutar la función print(), el programa muestra un saludo personalizado. Este tipo de interacción es muy útil para crear programas donde el usuario pueda proporcionar distintos valores de entrada.

Diferentes formas de utilizar la función print()

La función print() en Python se utiliza para mostrar información en pantalla. No solo permite imprimir texto, sino que también facilita el control del formato de salida con secuencias de escape, además de opciones avanzadas como especificar el número de decimales.

Secuencias de escape en print()

Las secuencias de escape son caracteres especiales que permiten modificar la apariencia de la salida en pantalla. A continuación, se presentan las secuencias de escape más comunes en Python:

- \n: Genera un salto de línea, dividiendo el texto en varias líneas.

```
1 print("Línea 1\nLínea 2\nLínea 3")
```

Código escrito en python.

```
Línea 1
Línea 2
Línea 3
```

Resultado de la ejecución del código de python en pantalla.

- \t: Añade una tabulación (espacio amplio), útil para organizar el texto en columnas.

```
1 print("Nombre\tEdad\tCiudad")
2 print("Dani\t21\tMonterrey")
```

Código escrito en python.

Nombre	Edad	Ciudad
Dani	21	Monterrey

Resultado de la ejecución del código de python en pantalla.

- `\\`: Permite imprimir un carácter de barra invertida `\`.

```
1 print("Este es un símbolo de barra invertida: \\")
```

Código escrito en python.

Este es un símbolo de barra invertida: \

Resultado de la ejecución del código de python en pantalla.

- `'` y `"`: Permiten incluir comillas simples y dobles en el texto sin cerrar la cadena de texto.



```
1 print("Ella dijo: \"Hola\" y él respondió: 'Hola'")
```


Código escrito en python.



```
Ella dijo: "Hola" y él respondió: 'Hola'
```

Resultado de la ejecución del código de python en pantalla.

- `\r`: Retorno de carro, que mueve el cursor al principio de la línea actual sin avanzar a la siguiente línea.



```
1 print("Hola\rMundo")
```

Código escrito en python.



```
Mundo
```

Resultado de la ejecución del código (solo visible en ciertas terminales):

- `\b`: Retroceso, elimina el carácter inmediatamente anterior.

```
1 print("Hola\b!")
```

Código escrito en python.

```
Hol!
```

Resultado de la ejecución del código de python en pantalla.

Estas secuencias son útiles para organizar y dar formato al texto de salida, haciendo que el resultado en pantalla sea más claro y fácil de leer.

Concatenación de cadenas y visualización de datos en la salida.

En Python, es común tener que mostrar números con un número específico de decimales, especialmente en campos como matemáticas y ciencias. Para lograr esto, pueden usar f-strings o el método `.format()`.

Usando f-strings para formatear decimales

Con f-strings, pueden especificar la cantidad de decimales que desean mostrando el valor dentro de las llaves `{}`. Agregar `:.2f` dentro de las llaves indica que se debe mostrar el número con dos decimales. Por ejemplo:

```
1 pi = 3.141592653589793
2 print(f"El valor de pi con dos decimales es: {pi:.2f}")
```

Código escrito en python.

```
El valor de pi con dos decimales es: 3.14
```

Resultado de la ejecución del código de python en pantalla.

Si desean mostrar el valor con un solo decimal, pueden cambiar el 2 que está dentro de la expresión "{pi:.2f}" por 1. Por ejemplo:

```
1 pi = 3.141592653589793
2 print(f"El valor de pi con dos decimales es: {pi:.1f}")
```

Código escrito en python.

```
El valor de pi con dos decimales es: 3.1
```

Resultado de la ejecución del código de python en pantalla.

Usando .format() para formatear decimales

Otra opción es el método .format(), que también permite especificar decimales. Por ejemplo:

```
1 pi = 3.141592653589793
2 print("Valor pi con dos decimales es: {:.2f}".format(pi))
```

Código escrito en python.

```
Valor pi con dos decimales es: 3.14
```

Resultado de la ejecución del código de python en pantalla.

Con estas técnicas, pueden combinar texto y variables para hacer el contenido más dinámico. La concatenación de cadenas se refiere a la unión de varios elementos de texto en una sola salida. Existen diferentes formas de hacerlo en Python:

- Uso del operador +:

```
1 nombre = "Ana"
2 print("Hola, " + nombre + "!")
```

Código escrito en python.

- f-strings:

```
1 edad = 20
2 print(f"¡Hola! Tienes {edad} años.")
```

Código escrito en python.

- .format():

```
1 nombre = "Dani"
2 print("Hola, {}. ¡Bienvenido!".format(nombre))
```

Código escrito en python.

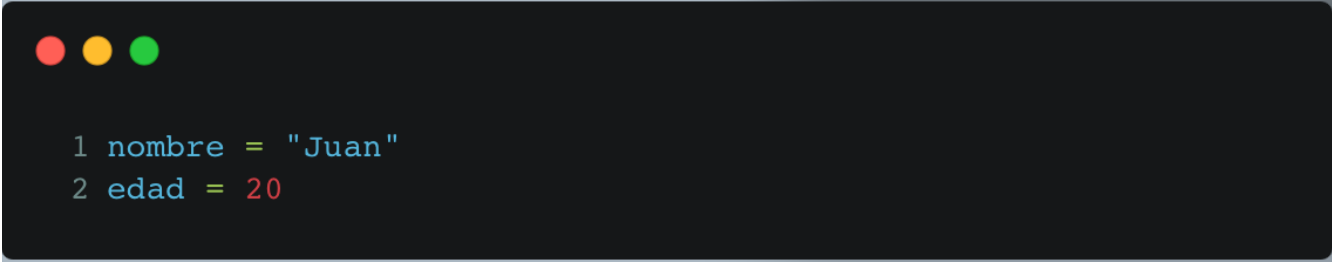
Al aprender estas técnicas, podrán crear programas que interactúen y muestren información de forma clara y organizada, lo que es esencial para desarrollar habilidades en programación. ¡Están en el camino correcto!

5. Variables, constantes y operaciones básicas

Ahora que comprenden cómo interactuar con un programa usando entradas y salidas de datos, es momento de explorar otro conjunto de conceptos fundamentales: variables, constantes y operaciones básicas. Estos elementos permiten almacenar, manipular y operar con datos de forma eficiente y estructurada, lo que constituye la base de cualquier programa en Python.

Declaración de variables y constantes en Python.

En Python, una variable es un espacio en la memoria de la computadora donde se almacena un valor que puede cambiar a lo largo de la ejecución del programa. Para declarar una variable, solo es necesario asignarle un valor usando el operador =. Por ejemplo:



```
1 nombre = "Juan"
2 edad = 20
```

Código escrito en python.

Aquí, nombre y edad son variables que contienen los valores "Juan" y 20, respectivamente. No es necesario especificar el tipo de variable en Python, ya que este es un lenguaje de tipado dinámico.

Por otro lado, una constante es un valor que, por convención, no debe cambiar durante la ejecución del programa. Python no tiene una manera oficial de declarar constantes, pero se acostumbra nombrarlas usando letras mayúsculas para distinguirlas de las variables regulares. Por ejemplo:



```
1 PI = 3.1416
```

Código escrito en python.

Aunque nada impide cambiar el valor de PI, el uso de mayúsculas indica que se considera constante.

Tipos de datos en Python: enteros, flotantes, cadenas.

Python cuenta con diversos tipos de datos que permiten trabajar con diferentes tipos de información. Entre los más comunes se encuentran:

- Enteros (int): Representan números enteros, ya sean positivos o negativos, como 5, -20 o 1000.
- Flotantes (float): Representan números con decimales, como 3.14, -0.5 o 2.718.
- Cadenas (str): Representan secuencias de caracteres, como "hola", "Python" o "1234". Se escriben entre comillas simples o dobles.

Obtención del tipo de dato de una variable con type().

Para saber el tipo de dato de una variable en Python, se usa la función type():

```
1 nombre = "Juan"
2 # Salida: <class 'str'>
3 print(type(nombre))
4
5 edad = 20
6 # Salida: <class 'int'>
7 print(type(edad))
```

Código escrito en python.

Conversión de tipos de datos (int(), float(), str()).

La conversión de tipos de datos es útil cuando se desea cambiar el tipo de una variable para realizar ciertas operaciones. Python ofrece funciones como int(), float() y str() para convertir entre tipos:

- int(): Convierte un valor a entero, eliminando los decimales si es un flotante.
- float(): Convierte un valor a flotante, agregando un punto decimal si era entero.
- str(): Convierte un valor a cadena.

Por ejemplo:

```
1 edad = "20"
2 # Convierte "20" a entero
3 edad_int = int(edad)
4 altura = 1.75
5 # Convierte 1.75 a cadena "1.75"
6 altura_str = str(altura)
```

Código escrito en python.

Es importante recordar que no se puede convertir cualquier valor; intentar convertir una cadena no numérica a un número causará un error.


Operadores aritméticos, lógicos y de comparación.

Los operadores permiten realizar operaciones matemáticas y lógicas con las variables. Python incluye tres categorías principales de operadores:

Operadores aritméticos: Permiten realizar cálculos matemáticos.

- `+`: Suma.
- `-`: Resta.
- `*`: Multiplicación.
- `/`: División (siempre devuelve un valor flotante).
- `//`: División entera (devuelve solo la parte entera de la división).
- `%`: Módulo (resto de la división).
- `**`: Exponenciación.

Ejemplo:




```
1 a = 10
2 b = 3
3 # Resultado: 13
4 suma = a + b
5 # Resultado: 3
6 division_entera = a // b
7 # Resultado: 1000
8 potencia = a ** b
```

Código escrito en python.

Operadores de comparación: Usados para comparar valores, devolviendo True o False.

- ==: Igualdad.
- !=: Desigualdad.
- >: Mayor que.
- <: Menor que.
- >=: Mayor o igual que.
- <=: Menor o igual que.

Ejemplo:




```
1 x = 5
2 y = 10
3 # Resultado: False
4 print(x > y)
5 # Resultado: True
6 print(x != y)
```

Código escrito en python.

Operadores lógicos: Permiten realizar combinaciones lógicas y son útiles en las condiciones.

- and: Devuelve True solo si ambas condiciones son verdaderas.
- or: Devuelve True si al menos una de las condiciones es verdadera.
- not: Invierte el resultado lógico.

Ejemplo:



```
1 es_mayor = x > 0
2 es_menor = y < 20
3 # Resultado: True
4 print(es_mayor and es_menor)
5 # Resultado: True
6 print(es_mayor or es_menor)
7 # Resultado: False
8 print(not es_mayor)
```

Código escrito en python.

El uso de variables, constantes y operadores les permite almacenar, manipular y comparar datos en sus programas. Esto es la base de operaciones más complejas, y les ayudará a desarrollar programas cada vez más poderosos. Recuerden que pueden usar `type()` para verificar los tipos de sus variables y convertir tipos cuando sea necesario. ¡Con estas herramientas y conceptos, están listos para avanzar en su aprendizaje de Python y en la creación de programas que resuelvan problemas reales!

6. Estructuras de control: Condicionales y ciclos

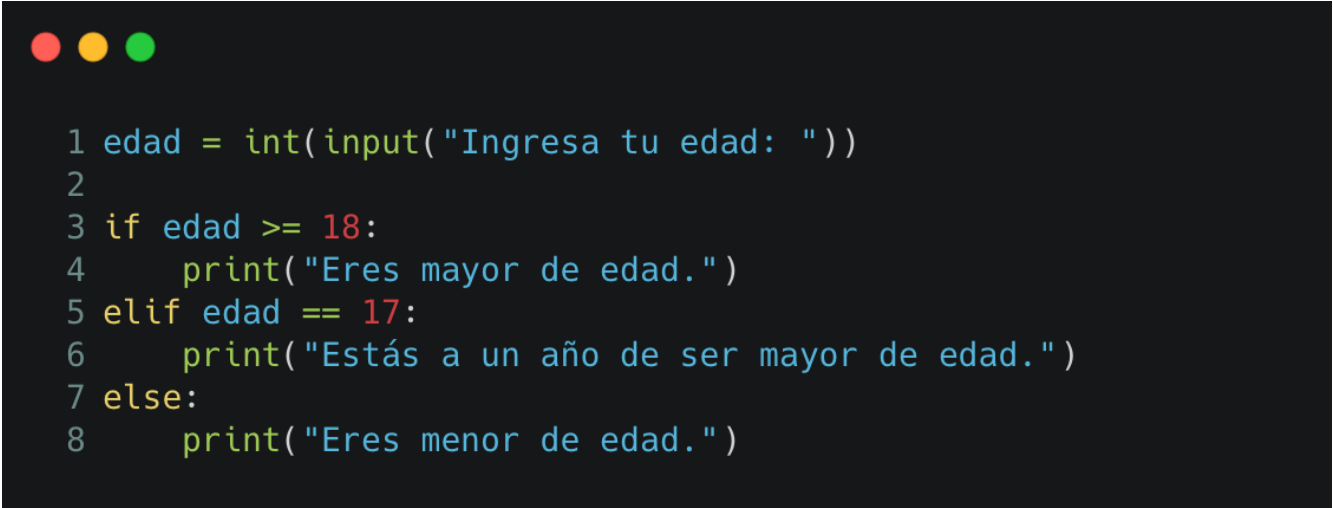
A medida que sus programas se vuelven más complejos, es importante que puedan tomar decisiones y repetir ciertas acciones de forma eficiente. Esto se logra mediante el uso de estructuras de control como las condicionales y los ciclos. En esta sección, exploraremos cómo usar condicionales para tomar decisiones y ciclos para repetir operaciones, dos herramientas que les ayudarán a crear programas dinámicos y útiles.

Uso de condicionales (`if`, `else`, `elif`) para tomar decisiones.

Las estructuras condicionales permiten que un programa tome decisiones basadas en condiciones específicas. En Python, se utilizan las palabras clave `if`, `elif` y `else` para ejecutar bloques de código bajo ciertas condiciones:

- `if`: Evalúa una condición y ejecuta el bloque de código asociado si la condición es verdadera.
- `elif`: Significa "else if" y permite evaluar otra condición si la anterior fue falsa.
- `else`: Se ejecuta solo si todas las condiciones anteriores fueron falsas.

Ejemplo de una estructura condicional:



```
1 edad = int(input("Ingresa tu edad: "))
2
3 if edad >= 18:
4     print("Eres mayor de edad.")
5 elif edad == 17:
6     print("Estás a un año de ser mayor de edad.")
7 else:
8     print("Eres menor de edad.")
```

Código escrito en python.

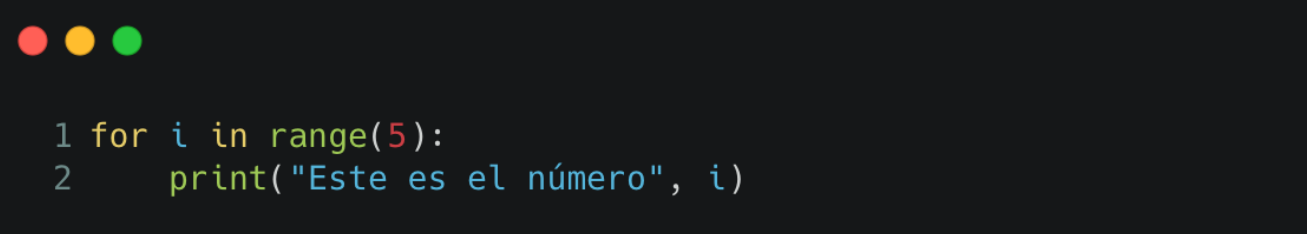
En este ejemplo, el programa evalúa si la variable `edad` cumple con las condiciones especificadas y ejecuta el mensaje adecuado.

Estructuras de repetición: ciclos for y while.

Los ciclos permiten que un bloque de código se ejecute varias veces, lo cual es útil cuando queremos repetir una acción hasta que se cumpla cierta condición o para iterar sobre una colección de elementos.

Ciclo for

El ciclo for se utiliza principalmente para recorrer una secuencia (como una lista o un rango de números). La palabra clave for se usa junto con in para iterar sobre cada elemento de la secuencia: Por ejemplo:

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Python and uses syntax highlighting: 'for' is blue, 'i' is green, 'in' is blue, 'range' is green, '5' is red, 'print' is blue, and the string is in quotes.


```
1 for i in range(5):  
2     print("Este es el número", i)
```

Código escrito en python.

Este código imprimirá los números del 0 al 4. La función range(5) genera una secuencia de números desde 0 hasta 4, y en cada iteración, el valor de i cambia.

Ciclo while

El ciclo while ejecuta un bloque de código mientras una condición sea verdadera. Es útil cuando no sabemos cuántas veces se repetirá la acción, pero sí conocemos la condición para finalizar. Por ejemplo:

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Python and uses syntax highlighting: 'while' is blue, 'contador' is green, '<' is blue, '5' is red, 'print' is blue, and the string is in quotes. The increment '+= 1' is also highlighted.

```
1 contador = 0  
2  
3 while contador < 5:  
4     print("Contador:", contador)  
5     contador += 1
```

Código escrito en python.

Este ciclo imprimirá los valores del contador de 0 a 4, incrementando su valor en 1 en cada iteración hasta que llegue a 5 y la condición contador < 5 sea falsa.

Ciclos anidados y control de flujo avanzado (uso de break, continue).

En Python, pueden anidar ciclos, es decir, tener un ciclo dentro de otro. Esta estructura permite recorrer matrices o listas de listas, y se usa en tareas más complejas.

Uso de break y continue

Dentro de los ciclos, podemos usar las palabras clave break y continue para controlar el flujo de ejecución.

- break: Detiene el ciclo de inmediato, independientemente de que la condición no haya sido falseada.
- continue: Salta a la siguiente iteración del ciclo sin ejecutar el resto del bloque.

Ejemplo:

```
1 for i in range(5):
2     if i == 3:
3         # Detiene el ciclo cuando i es igual a 3
4         break
5     print("Valor de i:", i)
6
7 for j in range(5):
8     if j == 2:
9         # Salta la iteración cuando j es igual a 2
10        continue
11    print("Valor de j:", j)
```

Código escrito en python.

```
Valor de i: 0
Valor de i: 1
Valor de i: 2
Valor de j: 0
Valor de j: 1
Valor de j: 3
Valor de j: 4
```


Resultado de la ejecución del código de python en pantalla.

Palabras reservadas en Python

Las palabras reservadas son términos que Python utiliza como comandos básicos. No pueden ser usados como nombres de variables, ya que tienen un significado especial dentro del lenguaje. Algunas de las palabras reservadas más importantes son:

- Estructuras de control: if, elif, else, for, while, break, continue
- Declaración de funciones y clases: def, class, return, yield
- Declaraciones de lógica: and, or, not, is, in
- Manejo de excepciones: try, except, finally, raise
- Otros: import, from, as, global, nonlocal

Un vistazo rápido a las palabras reservadas se puede hacer en Python con el módulo keyword:



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

Código escrito en python.

Este código imprimirá todas las palabras reservadas de Python. Conocer estas palabras y su uso les permitirá evitar errores y escribir código más claro y organizado.

Conclusión

El uso de condicionales, ciclos y control de flujo avanzado permite a sus programas realizar tareas dinámicas y repetitivas, lo cual es fundamental para cualquier tipo de software. Además, entender las palabras reservadas de Python facilita el uso correcto de las estructuras de programación y ayuda a evitar errores comunes. Al aprender y dominar estos conceptos, tendrán las bases necesarias para resolver problemas y desarrollar algoritmos cada vez más sofisticados en Python. ¡Continúen practicando y aplicando estos conceptos para convertirse en expertos programadores!

7. Introducción a las palabras reservadas y su significado en Python.

Es fundamental que comprendan las reglas de sintaxis y adquieran buenas prácticas de escritura de código. Esto no solo hará que sus programas sean más comprensibles para ustedes y otros programadores, sino que también evitará errores y facilitará la colaboración en proyectos futuros. En esta sección, también exploraremos las listas en Python, un tipo de dato poderoso que les permitirá almacenar y manipular colecciones de datos de forma eficiente.

Reglas de sintaxis y buenas prácticas en la escritura de código.

La sintaxis de Python es la estructura y reglas que definen cómo escribir código correctamente. A diferencia de otros lenguajes, Python es muy sensible a la indentación (espacios o tabulaciones al inicio de una línea), lo cual determina el agrupamiento de las instrucciones.

Reglas de Sintaxis Básicas en Python

- Indentación: Python requiere que cada bloque de código esté correctamente indentado. Por convención, se utilizan 4 espacios por nivel de indentación.

A screenshot of a code editor with a dark background. At the top left, there are three colored circles: red, yellow, and green. Below them, there is Python code. The first line is '1 if edad >= 18:' and the second line is '2 print("Eres mayor de edad")'. The second line is indented with four spaces.

```
1 if edad >= 18:
2     print("Eres mayor de edad")
```

Código escrito en python.

- Mayúsculas y minúsculas: Python distingue entre mayúsculas y minúsculas. Edad, edad y EDAD serían tres variables diferentes.
- Comentarios: Se utiliza el símbolo # para escribir comentarios en una línea, lo cual es útil para describir qué hace una sección del código.
- Buenas Prácticas en la Escritura de Código
- Nombrar las variables claramente: Los nombres de variables deben ser descriptivos, de modo que indiquen claramente su propósito. Por ejemplo, usar num_estudiantes en lugar de n.
- Comentarios claros y útiles: Los comentarios ayudan a entender el propósito de cada parte del código, y son esenciales cuando el programa se comparte o colabora con otros.
- PEP 8: Es la guía de estilo de Python y contiene recomendaciones sobre la estructura del código. Es una excelente referencia para escribir código limpio y legible.
- Evitar código redundante: Tratar de evitar operaciones o bloques de código innecesarios, buscando simplificar.

Al seguir estas prácticas, sus programas serán más legibles, fáciles de mantener y tendrán menos errores.

Introducción a listas en Python

Las listas son una estructura de datos en Python que permite almacenar varios elementos en un solo lugar. Son similares a los arreglos en otros lenguajes de programación, pero más flexibles, ya que pueden contener diferentes tipos de datos y variar en tamaño.

Definición de listas, cómo crear y modificar listas.

Una lista en Python se define usando corchetes `[]` y puede contener elementos de cualquier tipo, incluyendo números, cadenas o incluso otras listas. Aquí les mostramos cómo crear una lista:

```
1 # Lista vacía
2 mi_lista = []
3
4 # Lista con elementos
5 frutas = ["manzana", "plátano", "naranja"]
```

Código escrito en python.

Para acceder a un elemento de la lista, se utiliza su índice, que en Python comienza desde 0:

```
1 # Lista vacía
2 mi_lista = []
3
4 # Lista con elementos
5 frutas = ["manzana", "plátano", "naranja"]
6
7 # Imprime "manzana"
8 print(frutas[0])
```

Código escrito en python.

Modificar un elemento de la lista es tan fácil como asignar un nuevo valor al índice correspondiente:

```
1 # Lista vacía
2 mi_lista = []
3
4 # Lista con elementos
5 frutas = ["manzana", "plátano", "naranja"]
6
7 # Cambia "plátano" por "kiwi"
8 frutas[1] = "kiwi"
```

Código escrito en python.

Operaciones básicas con listas: agregar, eliminar, buscar elementos.

Python proporciona varios métodos útiles para trabajar con listas, facilitando la manipulación de los datos almacenados.

- Agregar elementos: Se puede agregar un nuevo elemento al final de la lista usando el método `append()`:

```
1 # Lista vacía
2 mi_lista = []
3
4 # Lista con elementos
5 frutas = ["manzana", "plátano", "naranja"]
6
7 #Agrega el elemento "uva" al final de la lista
8 frutas.append("uva")
```

Código escrito en python.

- También pueden insertar elementos en una posición específica con insert():

```
1 # Lista vacía
2 mi_lista = []
3
4 # Lista con elementos
5 frutas = ["manzana", "plátano", "naranja"]
6
7 # Inserta "mango" en la posición 1
8 frutas.insert(1, "mango")
```

Código escrito en python.

- Eliminar elementos: Se puede eliminar un elemento específico con remove() o eliminar un elemento por su índice usando pop().

```
1 # Lista vacía
2 mi_lista = []
3
4 # Lista con elementos
5 frutas = ["manzana", "plátano", "naranja"]
6
7 # Elimina "naranja" de la lista
8 frutas.remove("naranja")
9 # Elimina el elemento en la posición 2
10 frutas.pop(2)
```

Código escrito en python.

- Buscar elementos: Usen el método `index()` para encontrar la posición de un elemento en la lista:

```
1 # Lista vacía
2 mi_lista = []
3
4 # Lista con elementos
5 frutas = ["manzana", "plátano", "naranja"]
6
7 # Encuentra el índice de "manzana"
8 posicion = frutas.index("manzana")
```

Código escrito en python.

- Además, pueden verificar si un elemento está en la lista con el operador `in`:

```
1 # Lista vacía
2 mi_lista = []
3
4 # Lista con elementos
5 frutas = ["manzana", "plátano", "naranja"]
6
7 if "uva" in frutas:
8     print("La uva está en la lista")
```

Código escrito en python.

Conclusión

Las listas en Python son una de las estructuras de datos más versátiles y utilizadas, ya que permiten almacenar y manipular múltiples valores en un solo lugar. Al comprender las reglas de sintaxis, aplicar buenas prácticas y aprender a trabajar con listas, estarán adquiriendo las herramientas esenciales para estructurar y optimizar su código en Python. La capacidad de almacenar y gestionar colecciones de datos es fundamental en cualquier tipo de proyecto de programación, y aprender a hacerlo de manera eficiente les permitirá desarrollar programas más robustos y dinámicos. ¡Sigán practicando y experimentando con listas para mejorar su dominio de Python!

Fase II: Funciones, Estructuras de Datos, Algoritmos y Paradigmas de Programación

Objetivo de la Fase: Que el estudiante encapsule código utilizando funciones, manipule estructuras de datos avanzadas, aplique algoritmos de búsqueda, ordenamiento y recursividad para la resolución de problemas, e identifique distintos paradigmas de programación.

El siguiente paso en su aprendizaje de Python los lleva a profundizar en el uso de funciones, la creación de soluciones modulares y el manejo de estructuras avanzadas. En esta fase, aprenderán a encapsular código en funciones reutilizables, optimizando la organización y el mantenimiento de sus programas. Asimismo, explorarán herramientas como las funciones lambda, que permiten realizar tareas simples de manera compacta y eficiente.

1. Funciones

Definición y creación de funciones en Python.

Una función es un bloque de código que realiza una tarea específica y puede reutilizarse en diferentes partes del programa. Utilizar funciones no solo hace que su código sea más limpio y fácil de entender, sino que también reduce errores al evitar la repetición de código.

La estructura básica de una función en Python incluye:

1. La palabra clave `def`, que indica que están definiendo una función.
2. El nombre de la función.
3. Paréntesis para incluir los argumentos (si los hay).
4. Dos puntos (`:`) al final de la línea para comenzar el bloque de código.
5. El código indentado, que representa las instrucciones de la función.

Ejemplo: Crear una función simple



```
1 def saludar():  
2     print("¡Hola, bienvenido a Python!")
```

Código escrito en python.

Para ejecutar esta función, simplemente llamen su nombre:



```
1 saludar() # Imprime "¡Hola, bienvenido a Python!"
```

Código escrito en python.

Paso de argumentos y retorno de valores.

Las funciones pueden recibir argumentos (datos de entrada) y devolver valores como resultado. Esto las hace más dinámicas y útiles.

Paso de Argumentos

Los argumentos se definen dentro de los paréntesis de la función:



```
1 def saludar(nombre):  
2     print(f"¡Hola, {nombre}!")
```

Código escrito en python.

Cuando llaman a esta función, deben proporcionar un argumento:



```
1 saludar("Ana")  # Imprime "¡Hola, Ana!"
```

Código escrito en python.

Retorno de Valores

El valor que una función devuelve puede almacenarse en una variable o utilizarse directamente. Para devolver un valor, se usa la palabra clave return:



```
1 def suma(a, b):  
2     return a + b
```

Código escrito en python.



```
1 resultado = suma(3, 5)  # resultado será igual a 8  
2 print(resultado)
```

Código escrito en python.

Uso de funciones lambda para tareas simples.

Las funciones lambda, también conocidas como funciones anónimas, son funciones pequeñas y rápidas que se definen en una sola línea. Estas son útiles para tareas simples, como ordenar una lista o realizar operaciones matemáticas rápidas.

Sintaxis: lambda argumentos: expresión

Ejemplo: Sumar dos números



```
1 suma = lambda x, y: x + y
2 print(suma(4, 6)) # Imprime 10
```

Código escrito en python.

Otro ejemplo común es utilizar funciones lambda con herramientas como sorted() para ordenar listas:



```
1 nombres = ["Ana", "Luis", "Carlos"]
2 nombres_ordenados = sorted(nombres, key=lambda x: len(x))
3 print(nombres_ordenados) # Ordena por la longitud del nombre
```

Código escrito en python.

Funciones con Múltiples Argumentos Posicionales

Son funciones que reciben un número fijo de argumentos en un orden específico.

Ejemplo:



```
1 def sumar(a, b, c):
2     return a + b + c
3
4 print(sumar(2, 3, 4)) # Salida: 9
```

Código escrito en python.

Funciones con Argumentos por Defecto

Se pueden definir valores predeterminados para los parámetros, permitiendo que sean opcionales.

Ejemplo:



```
1 def saludar(nombre, mensaje="Hola"):
2     return f"{mensaje}, {nombre}!"
3
4 print(saludar("Carlos"))          # Salida: Hola, Carlos!
5 print(saludar("Ana", "Buenos días")) # Salida: Buenos días, Ana!
```

Código escrito en python.

Nota: Si no se proporciona el segundo argumento, se usa el valor por defecto.

Funciones con un Número Variable de Argumentos (*args)

Se usa *args para recibir una cantidad indefinida de argumentos posicionales. Se almacenan como una tupla dentro de la función.

Ejemplo:



```
1 def suma(*numeros):
2     return sum(numeros)
3
4 print(suma(1, 2, 3, 4, 5)) # Salida: 15
5 print(suma(10, 20))       # Salida: 30
```

Código escrito en python.

Nota: *args permite pasar cualquier cantidad de valores sin necesidad de especificar cuántos.


Modularización del código a través de funciones.

La modularización es una técnica que consiste en dividir un programa en pequeñas partes independientes (módulos), cada una de las cuales realiza una tarea específica. Esto facilita la organización, depuración y mantenimiento del código.

Beneficios de la modularización

- Reutilización: Pueden usar las mismas funciones en diferentes programas.
- Legibilidad: El código es más fácil de entender y seguir.
- Escalabilidad: Es más sencillo añadir nuevas funcionalidades a un programa modular.

Ejemplo de Programa Modular



```
1 # Función para obtener el área de un círculo
2 def area_circulo(radio):
3     return 3.1416 * radio**2
4
5 # Función para obtener el área de un rectángulo
6 def area_rectangulo(base, altura):
7     return base * altura
8
9 # Función principal
10 def main():
11     radio = float(input("Introduce el radio del círculo: "))
12     print(f"El área del círculo es: {area_circulo(radio):.2f}")
13
14     base = float(input("Introduce la base del rectángulo: "))
15     altura = float(input("Introduce la altura del rectángulo: "))
16     print(f"El área del rectángulo es: {area_rectangulo(base, altura):.2f}")
17
18 # Ejecutar la función principal
19 main()
```

Código escrito en python.

Las funciones son una de las herramientas más poderosas de Python, permitiéndoles estructurar su código de forma clara y eficiente. El uso de argumentos, valores de retorno y funciones lambda les da flexibilidad para resolver problemas de manera elegante. Finalmente, al adoptar la modularización, estarán siguiendo una práctica fundamental en el desarrollo de software profesional, facilitando la colaboración y el mantenimiento a largo plazo de sus proyectos. ¡Sigán practicando y experimentando para dominar estas habilidades!

Funciones de esenciales para trabajar con datos en Python

len()

La función len() se utiliza para obtener la cantidad de elementos de una secuencia o colección (listas, tuplas, cadenas, diccionarios, conjuntos, etc.).

Sintaxis: len(objeto)

- objeto: Puede ser una lista, tupla, string, diccionario, conjunto, etc.

Ejemplos:

Con una cadena (str)

Cuenta el número de caracteres en la cadena (incluyendo espacios y signos de puntuación).



```
1 texto = "Python"
2 print(len(texto)) # Salida: 6
```

Código escrito en python.

Con una lista (list)

Cuenta el número de elementos en la lista.



```
1 numeros = [10, 20, 30, 40]
2 print(len(numeros)) # Salida: 4
```

Código escrito en python.

Con un diccionario (dict)

Cuenta el número de claves en el diccionario.



```
1 datos = {"nombre": "Ana", "edad": 25, "ciudad": "Madrid"}
2 print(len(datos)) # Salida: 3
```

Código escrito en python.

sum()

La función sum() se usa para calcular la suma de los elementos en un iterable (lista, tupla, conjunto, etc.). Solo funciona con números o elementos que puedan sumarse.

Sintaxis: sum(iterable, inicio)

- iterable: Un objeto iterable (lista, tupla, conjunto, etc.) con valores numéricos.
- inicio (opcional): Un valor inicial que se sumará al resultado final. Su valor por defecto es 0.

Ejemplos:

Suma de una lista de números

La función suma $1 + 2 + 3 + 4 + 5 = 15$.



```
1 numeros = [1, 2, 3, 4, 5]
2 print(sum(numeros)) # Salida: 15
```

Código escrito en python.

Suma con un valor inicial

Se suma $10 + 1 + 2 + 3 = 16$.



```
1 numeros = [1, 2, 3]
2 print(sum(numeros, 10)) # Salida: 16
```

Código escrito en python.

Suma de una tupla

Funciona igual que con listas.



```
1 valores = (4.5, 3.2, 7.8)
2 print(sum(valores)) # Salida: 15.5
```

Código escrito en python.

Es importante tener en cuenta que sum() no funciona con cadenas de texto, ya que no puede sumarlas directamente.

sorted()

La función `sorted()` se usa para ordenar elementos de un iterable (listas, tuplas, cadenas, etc.) y devuelve una nueva lista ordenada, sin modificar la original.

Sintaxis: `sorted(iterable, key=None, reverse=False)`

- `iterable`: Cualquier colección ordenable (lista, tupla, string, etc.).
- `key` (opcional): Una función para personalizar el criterio de ordenación.
- `reverse` (opcional): Si es `True`, ordena en orden descendente.

Ejemplos:

Ordenación de una lista de números


Ordena los números en orden ascendente.



```
1 numeros = [3, 1, 4, 1, 5, 9, 2]
2 print(sorted(numeros)) # Salida: [1, 1, 2, 3, 4, 5, 9]
```

Código escrito en python.

Ordenación en orden descendente



```
1 numeros = [3, 1, 4, 1, 5, 9, 2]
2 print(sorted(numeros, reverse=True)) # Salida: [9, 5, 4, 3, 2, 1, 1]
```

Código escrito en python.

Ordenación de cadenas (alfabéticamente)




```
1 palabras = ["banana", "manzana", "pera", "uva"]
2 print(sorted(palabras)) # Salida: ['banana', 'manzana', 'pera', 'uva']
```

Código escrito en python.

Ordenación con una clave personalizada

Puedes usar el parámetro `key` para definir un criterio de ordenación.

Por ejemplo, ordenar palabras por su longitud:



```
1 palabras = ["banana", "manzana", "pera", "uva"]
2 print(sorted(palabras, key=len))
3 # Salida: ['uva', 'pera', 'banana', 'manzana']
```

Código escrito en python.

2. Estructuras de datos

Las estructuras de datos son fundamentales en la programación, ya que permiten almacenar, organizar y manipular información de manera eficiente. En el desarrollo de software, trabajar con datos de forma estructurada facilita la solución de problemas y la optimización del rendimiento de los programas.

En Python, las estructuras de datos juegan un papel clave en la administración de grandes volúmenes de información, permitiendo realizar operaciones como búsqueda, ordenamiento, inserción y eliminación de elementos de manera eficiente. La elección de una estructura de datos adecuada depende del tipo de problema a resolver y de los requisitos de tiempo y espacio de la aplicación.

Clasificación de las Estructuras de Datos

Las estructuras de datos se pueden clasificar en dos grandes grupos: primarias y secundarias.

1. Estructuras de Datos Primarias

Las estructuras primarias son las más simples y representan los datos básicos con los que trabaja un lenguaje de programación. En Python, estas estructuras incluyen:

- **Enteros (int):** Representan números enteros, como 10, -5 o 0.
- **Flotantes (float):** Números con decimales, como 3.14, -2.5 o 0.0.
- **Cadenas (str):** Texto o secuencias de caracteres, como "Hola, mundo".
- **Booleanos (bool):** Valores de verdad True o False.

Estas estructuras son la base para la construcción de estructuras más complejas.

2. Estructuras de Datos Secundarias

Las estructuras secundarias se construyen a partir de las primarias y permiten organizar mejor la información. Se dividen en estructuras lineales y estructuras no lineales.

1. Estructuras de Datos Lineales

Las estructuras de datos lineales organizan los elementos en una secuencia ordenada, en la que cada elemento tiene un único sucesor y un único predecesor (excepto el primero y el último).

- **Listas (list):** Son colecciones ordenadas y mutables que pueden contener diferentes tipos de datos.
- **Tuplas (tuple):** Similares a las listas, pero inmutables (no se pueden modificar después de su creación)
- **Pilas (stack):** Estructura que sigue el principio LIFO (Last In, First Out), donde el último elemento en ingresar es el primero en salir. Se puede implementar con listas.
- **Colas (queue):** Siguen el principio FIFO (First In, First Out), donde el primer elemento en ingresar es el primero en salir. Se puede implementar con `collections.deque`.

2. Estructuras de Datos No Lineales

A diferencia de las estructuras lineales, las no lineales no siguen un orden secuencial, lo que permite representar relaciones más complejas entre los datos.

- **Conjuntos (set):** Colecciones no ordenadas de elementos únicos
- **Diccionarios (dict):** Colecciones de pares clave-valor para almacenar información estructurada.
- **Grafos y Árboles:** Modelan estructuras jerárquicas y conexiones entre datos. Se utilizan en inteligencia artificial, redes y bases de datos.

Tuplas, Conjuntos y Diccionarios

En Python, existen diversas estructuras de datos que permiten organizar, almacenar y manipular información de manera eficiente. En esta sección exploraremos las tuplas, conjuntos y diccionarios, destacando su uso y sus características principales.

Definición de tuplas y su uso en Python.

Una tupla es una estructura de datos similar a una lista, pero con una diferencia importante: las tuplas son inmutables, es decir, no pueden modificarse después de ser creadas. Esto las hace ideales para almacenar datos que no deben cambiar, como coordenadas, configuraciones o constantes.

Creación de Tuplas

Las tuplas se definen utilizando paréntesis () o simplemente separando los elementos con comas:

```
mi_tupla = (1, 2, 3)
otra_tupla = "a", "b", "c"
```

Código escrito en python.

Acceso a Elementos de una Tupla

Se accede a los elementos mediante índices, igual que en las listas:

```
mi_tupla = (1, 2, 3)
print(mi_tupla[0]) # Imprime 1
```

Código escrito en python.

Usos Comunes de las Tuplas

Almacenamiento de Datos Inmutables: Garantizan que los datos no sean modificados accidentalmente.

Retorno de Múltiples Valores en Funciones:

```
def coordenadas():  
    return (10, 20)  
  
x, y = coordenadas()  
print(x, y) # Imprime 10 y 20
```

Código escrito en python.

Operaciones básicas con conjuntos y su uso en manipulación de datos únicos.

Un conjunto es una colección no ordenada de elementos únicos. Es útil para realizar operaciones matemáticas como uniones, intersecciones y diferencias, y para eliminar duplicados de una lista.

Creación de Conjuntos

Se utilizan llaves {} o la función set():

```
mi_conjunto = {1, 2, 3}  
otro_conjunto = set([3, 4, 5])
```

Código escrito en python.

Operaciones Básicas con Conjuntos

1. Unión: Combina los elementos de dos conjuntos.
2. Intersección: Obtiene los elementos comunes entre dos conjuntos.
3. Diferencia: Devuelve los elementos que están en un conjunto, pero no en el otro.

```
A = {1, 2, 3}  
B = {3, 4, 5}  
  
print(A | B) # Unión: {1, 2, 3, 4, 5}  
print(A & B) # Intersección: {3}  
print(A - B) # Diferencia: {1, 2}
```

Código escrito en python.

Eliminación de Duplicados

Los conjuntos son útiles para eliminar elementos duplicados de una lista:

```
lista = [1, 2, 2, 3, 4, 4]
conjunto = set(lista)

print(list(conjunto)) # Imprime [1, 2, 3, 4]
```

Código escrito en python.

Diccionarios: creación, acceso a pares clave-valor, y manipulación.

Los diccionarios son colecciones de pares clave-valor, donde cada clave está asociada a un valor. Son extremadamente útiles para almacenar y buscar datos de manera eficiente.

Creación de Diccionarios

Se utilizan llaves {} para definir un diccionario:

```
mi_diccionario = {
    "nombre": "Ana",
    "edad": 25,
    "ciudad": "Monterrey"
}
```

Código escrito en python.

Acceso a Valores

Los valores se obtienen utilizando sus claves:

```
mi_diccionario = {
    "nombre": "Ana",
    "edad": 25,
    "ciudad": "Monterrey"
}

print(mi_diccionario["nombre"]) # Imprime "Ana"
```

Código escrito en python.

Modificación de Diccionarios

Se pueden agregar, actualizar o eliminar elementos fácilmente:

```
# Agregar un nuevo par clave-valor
mi_diccionario["ocupación"] = "Estudiante"

# Actualizar un valor existente
mi_diccionario["edad"] = 26

# Eliminar un elemento
del mi_diccionario["ciudad"]
print(mi_diccionario)
```

Código escrito en python.

Métodos Útiles para Diccionarios

- **keys():** Devuelve todas las claves del diccionario.
- **values():** Devuelve todos los valores del diccionario.
- **items():** Devuelve una lista de pares clave-valor.

```
for clave, valor in mi_diccionario.items():
    print(f"{clave}: {valor}")
```

Código escrito en python.

Comparación de Estas Estructuras

Estructura	Características	Uso común
Tuplas	Inmutables, ordenadas	Datos constantes como coordenadas o configuraciones.
Conjuntos	No ordenados, únicos	Eliminación de duplicados, operaciones matemáticas.
Diccionarios	Pares clave-valor	Almacenamiento y búsqueda de datos relacionados.

Conclusión

Comprender las tuplas, conjuntos y diccionarios amplía significativamente su capacidad para manipular y organizar datos en Python. Cada una de estas estructuras tiene aplicaciones específicas que las hacen indispensables para resolver problemas de programación. A medida que avancen en esta fase, estas herramientas serán clave para desarrollar soluciones más robustas y eficientes.

3. Estructuras de datos anidadas

En programación, las estructuras de datos anidadas son una herramienta poderosa para organizar información compleja de forma jerárquica. Estas estructuras consisten en combinar listas, diccionarios, tuplas y conjuntos dentro de otras estructuras, creando una forma eficiente de manejar datos que tienen múltiples niveles de detalle.

Anidación de listas, diccionarios y otras estructuras de datos.

Una estructura de datos anidada es una colección que contiene otras estructuras de datos como elementos. Por ejemplo, puedes tener una lista de diccionarios, un diccionario que contenga listas, o incluso estructuras más complejas con múltiples niveles de anidación.

Ejemplo:

Imagina que necesitas almacenar información sobre varios estudiantes, incluyendo su nombre, matrícula y las calificaciones en distintas materias. Una lista de diccionarios sería una forma ideal de organizar estos datos:



Código escrito en python.

Tipos Comunes de Anidación

Listas dentro de listas

Se utilizan para representar datos en forma de matrices o tablas.

```
matriz = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]  
]
```

Código escrito en python.

Diccionarios dentro de listas

Útil para representar objetos o entidades con atributos múltiples.

```
inventario = [  
    {"producto": "Laptop", "precio": 15000, "cantidad": 10},  
    {"producto": "Teclado", "precio": 500, "cantidad": 50}  
]
```

Código escrito en python.

Listas dentro de diccionarios

Útil para agrupar elementos relacionados bajo una misma clave.

```
calificaciones = {  
    "Ana": [85, 90, 88],  
    "Luis": [78, 82, 80],  
    "Carlos": [92, 88, 95]  
}
```

Código escrito en python.

Estructuras más complejas

Se pueden combinar niveles múltiples de anidación según las necesidades del problema.

```
datos = {  
    "departamento": "Ventas",  
    "empleados": [  
        {"nombre": "Sofía", "ventas": [1000, 1200, 1100]},  
        {"nombre": "Pablo", "ventas": [900, 1150, 950]}  
    ]  
}
```

Código escrito en python.

Manipulación y acceso a datos en estructuras anidadas.

Acceder y manipular datos en estructuras anidadas requiere trabajar con índices y claves de forma jerárquica.

Ejemplo 1: Acceso a Listas Anidadas

Para acceder a un elemento específico en una lista dentro de otra lista:

```
matriz = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]  
]  
print(matriz[1][2]) # Imprime 6 (fila 2, columna 3)
```

Código escrito en python.

Ejemplo 2: Acceso a Diccionarios Dentro de una Lista

```
estudiantes = [  
    {"nombre": "Ana", "matrícula": "A001", "calificaciones": [85, 90, 88]},  
    {"nombre": "Luis", "matrícula": "A002", "calificaciones": [78, 82, 80]},  
    {"nombre": "Carlos", "matrícula": "A003", "calificaciones": [92, 88, 95]}  
]  
print(estudiantes[1]["nombre"]) # Imprime "Luis"
```

Ejemplo 3: Acceso a Listas Dentro de Diccionarios

```
estudiantes = [
    {"nombre": "Ana", "matrícula": "A001", "calificaciones": [85, 90, 88]},
    {"nombre": "Luis", "matrícula": "A002", "calificaciones": [78, 82, 80]},
    {"nombre": "Carlos", "matrícula": "A003", "calificaciones": [92, 88, 95]}
]
print(estudiantes[2]["calificaciones"][2]) # Imprime 95
```

Código escrito en python.

Ejemplo 4: Modificar Datos en Estructuras Anidadas

Puedes modificar datos accediendo directamente al nivel específico:

```
estudiantes = [
    {"nombre": "Ana", "matrícula": "A001", "calificaciones": [85, 90, 88]},
    {"nombre": "Luis", "matrícula": "A002", "calificaciones": [78, 82, 80]},
    {"nombre": "Carlos", "matrícula": "A003", "calificaciones": [92, 88, 95]}
]
estudiantes[0]["calificaciones"][1] = 95 # Cambia la segunda calificación de Ana a 95
```

Código escrito en python.

Ejemplo Práctico

Supongamos que necesitas calcular el promedio de las calificaciones de cada estudiante:

```
estudiantes = [
    {"nombre": "Ana", "matrícula": "A001", "calificaciones": [85, 90, 88]},
    {"nombre": "Luis", "matrícula": "A002", "calificaciones": [78, 82, 80]},
    {"nombre": "Carlos", "matrícula": "A003", "calificaciones": [92, 88, 95]}
]

for estudiante in estudiantes:
    promedio = sum(estudiante["calificaciones"]) / len(estudiante["calificaciones"])
    print(f'{estudiante["nombre"]} tiene un promedio de {promedio:.2f}')
```

"""

Resultado en pantalla:

Ana tiene un promedio de 89.33

Luis tiene un promedio de 80.00

Carlos tiene un promedio de 91.67

"""

Código escrito en python.

Buenas Prácticas al Trabajar con Estructuras Anidadas

- **Mantén la organización clara:** Usa nombres de claves y variables descriptivas para facilitar la comprensión.
- **Divide el trabajo en pasos pequeños:** Si trabajas con estructuras complejas, accede a los niveles uno a la vez.
- **Valida la existencia de elementos:** Usa métodos como `get()` para evitar errores al acceder a claves inexistentes.

```
datos = {  
    "departamento": "Ventas",  
    "empleados": [  
        {"nombre": "Sofía", "ventas": [1000, 1200, 1100]},  
        {"nombre": "Pablo", "ventas": [900, 1150, 950]}  
    ]  
}  
print(datos.get("departamento", "No especificado")) # Imprime "Ventas"
```

Código escrito en python.

Es importante mencionar que el método `.get()` en la estructura de datos diccionario de Python se utiliza para acceder a los valores asociados a una clave sin generar errores en caso de que la clave no exista.

```
valor = diccionario.get(clave, valor_por_defecto)
```

Código escrito en python.

Donde:

- **clave:** Es la clave cuyo valor queremos obtener.
- **valor_por_defecto (opcional):** Es el valor que se devuelve si la clave no está en el diccionario. Si no se especifica, devuelve `None` por defecto.

El método `.get()` se usa principalmente para:

1. Evitar errores al acceder a claves inexistentes

Si intentamos acceder a una clave inexistente con `diccionario['clave']`, Python lanza un error `KeyError`.

`.get()` devuelve `None` o un valor predeterminado si la clave no está presente, evitando la interrupción del código.

2. Proporcionar un valor predeterminado si la clave no existe

Útil cuando queremos devolver un valor por defecto en lugar de `None` o un error.

3. Consultar datos de forma segura sin necesidad de validaciones previas

Se evita el uso de `if` clave in diccionario, reduciendo líneas de código y mejorando la legibilidad.

Las estructuras de datos anidadas son herramientas fundamentales para manejar información estructurada y compleja. Su flexibilidad y capacidad de organizar datos jerárquicos hacen que sean esenciales en aplicaciones reales, desde la gestión de inventarios hasta el análisis de datos. Dominar su uso permitirá a los estudiantes abordar problemas más desafiantes de manera eficiente.

4. Recursividad

La recursividad es una técnica de programación en la que una función se llama a sí misma para resolver un problema. En términos simples, es como un "efecto espejo", donde una función se repite dentro de sí misma hasta alcanzar un caso base que detiene la ejecución.

Imagina que tienes una serie de muñecas rusas (matrioshkas). Para llegar a la más pequeña, primero debes abrir la más grande, luego la siguiente y así sucesivamente hasta que no haya más muñecas que abrir. La última muñeca es el caso base que detiene el proceso.

Una función recursiva tiene dos partes esenciales:

- **Caso base:**
La condición que detiene la recursión.
- **Caso recursivo:**
La parte donde la función se llama a sí misma con un problema reducido.

Ejemplo: Factorial de un número

El factorial de un número n (representado como $n!$) es el producto de todos los enteros positivos desde 1 hasta n : $n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$

```
def factorial(n):  
    if n == 0: # Caso base  
        return 1  
    else:  
        return n * factorial(n - 1) # Caso recursivo  
  
print(factorial(5)) # Salida: 120
```

Código escrito en python.

Explicación:

- `factorial(5)` llama a `factorial(4)`, que a su vez llama a `factorial(3)`, y así sucesivamente.
- Cuando `n == 0`, se devuelve 1 y se empieza a resolver de vuelta multiplicando los valores acumulados.

Es como una escalera en la que cada peldaño representa una llamada recursiva. Para bajar, sigues llamando la función hasta llegar al suelo (caso base). Luego, subes de nuevo multiplicando los valores acumulados.

La recursividad es una herramienta poderosa en la programación, permitiendo resolver problemas de manera elegante y estructurada. Sin embargo, debe usarse con precaución, ya que un mal diseño puede generar errores como recursión infinita o alto consumo de memoria. Para problemas donde se repiten muchas llamadas, es recomendable optimizar usando memorización (cache) o implementar una versión iterativa.

5. Divide y vencerás

"Divide y vencerás" es una técnica de resolución de problemas en la que un problema grande se divide en subproblemas más pequeños, se resuelven individualmente y luego se combinan para obtener la solución final.

Imagina que debes organizar una biblioteca desordenada. En lugar de revisar todos los libros a la vez, divides el trabajo en estantes más pequeños, organizas cada uno y luego los unes para tener una biblioteca ordenada.

Para aplicar este enfoque, un algoritmo debe seguir tres pasos:

1. **Dividir:** Se descompone el problema en subproblemas más pequeños.
2. **Resolver (Vencer):** Se resuelven los subproblemas de manera recursiva (o directamente si son simples).
3. **Combinar:** Se unen los resultados parciales para obtener la solución del problema original.

Ejemplo: Exponenciación Rápida (Potencia de un número)

En lugar de multiplicar el número manualmente varias veces, podemos usar Divide y Vencerás para reducir drásticamente la cantidad de multiplicaciones necesarias.

```
def potencia(base, exponente):  
    if exponente == 0: # Caso base  
        return 1  
    elif exponente % 2 == 0: # Si el exponente es par  
        mitad = potencia(base, exponente // 2)  
        return mitad * mitad  
    else: # Si el exponente es impar  
        return base * potencia(base, exponente - 1)  
  
# Prueba  
print(potencia(2, 10)) # Salida: 1024
```

Código escrito en python.

Es como doblar una hoja de papel: en lugar de sumarla muchas veces, la doblas por la mitad repetidamente hasta alcanzar el grosor deseado.

"Divide y vencerás" es una estrategia poderosa que permite resolver problemas de manera eficiente al dividirlos en partes más pequeñas. Se aplica en algoritmos de ordenamiento, búsqueda y cálculo matemático, optimizando el tiempo de ejecución y reduciendo la cantidad de operaciones necesarias.

Ventajas de "Divide y Vencerás"

- Reduce la complejidad computacional en comparación con enfoques iterativos.
- Es eficiente para problemas grandes.
- Se adapta bien a la recursión.

Desventajas

- Puede consumir más memoria debido a las llamadas recursivas.
- No siempre es la mejor opción para problemas pequeños donde un enfoque iterativo sería más simple.

Esta estrategia es una de las bases de la programación eficiente y es ampliamente utilizada en algoritmos modernos.

6. Algoritmos de ordenamiento

Los algoritmos de ordenamiento permiten organizar datos en una estructura (como una lista o arreglo) según un criterio específico, generalmente en orden ascendente o descendente.

Importancia del ordenamiento:

- Facilita la búsqueda de datos.
- Optimiza procesos como la búsqueda binaria.
- Mejora la eficiencia en el manejo de estructuras grandes.

Ordenamiento por Burbuja (Bubble Sort)

El algoritmo de burbuja compara elementos adyacentes y los intercambia si están en el orden incorrecto. Se repite el proceso hasta que toda la lista esté ordenada.

Imagina burbujas subiendo en un vaso de agua. Las burbujas más ligeras (valores más pequeños) suben a la superficie antes que las más pesadas (valores mayores).

Nombre del Algoritmo: ORDENAMIENTO_BURBUJA

Declaración de variables:

LISTA: Arreglo de enteros

I: Variable de tipo entero

J: Variable de tipo entero

AUX: Variable de tipo entero

1. Inicio
2. Repetir con I desde 0 hasta la longitud de LISTA - 1, hacer
3. Repetir con J desde 0 hasta la longitud de LISTA - 1 - I, hacer
4. Si $LISTA[J] > LISTA[J+1]$, entonces
5. $AUX \leftarrow LISTA[J]$
6. $LISTA[J] \leftarrow LISTA[J+1]$
7. $LISTA[J+1] \leftarrow AUX$
8. Fin de la condición del paso 4
9. Fin del ciclo del paso 3
10. Fin del ciclo del paso 2
11. Fin

Código escrito en python.

```
def ordenamiento_burbuja(lista):
    for i in range(len(lista) - 1):
        for j in range(len(lista) - 1 - i):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j] #Python permite el intercambio de variables

# Prueba del algoritmo
numeros = [64, 34, 25, 12, 22, 11, 90]
ordenamiento_burbuja(numeros)
print(numeros) # [11, 12, 22, 25, 34, 64, 90]
```

Código escrito en python.

Puntos importantes a tomar en cuenta:

- Complejidad: $O(n^2)$ en el peor caso.
- Uso: Sencillo pero ineficiente para listas grandes.

Ordenamiento por Inserción (Insertion Sort)

Este método construye una lista ordenada insertando elementos uno a uno en la posición correcta.

Es como ordenar cartas en la mano: se toma una carta y se inserta en su posición correcta en el grupo de cartas ya ordenadas.

Nombre del Algoritmo: ORDENAMIENTO_INSERTION

Declaración de variables:

LISTA: Arreglo de enteros

I: Variable de tipo entero

J: Variable de tipo entero

AUX: Variable de tipo entero

1. Inicio
2. Repetir con I desde 0 hasta la longitud de LISTA, hacer
3. AUX = LISTA[I]
4. J = I - 1
5. Mientras J >= 0 Y LISTA[J] > AUX, entonces
6. LISTA[J+1] = LISTA[J]
7. J = J - 1
8. Fin de la estructura repetitiva del paso 5
9. LISTA[J+1] = AUX
10. Fin de la estructura repetitiva del paso 2
11. Fin

Código escrito en python.

```
def ordenamiento_insercion(lista):
    for i in range(1, len(lista)):
        aux = lista[i]
        j = i - 1
        while j >= 0 and lista[j] > aux:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = aux

# Prueba del algoritmo
numeros = [64, 34, 25, 12, 22, 11, 90]
ordenamiento_insercion(numeros)
print(numeros) # [11, 12, 22, 25, 34, 64, 90]
```

Código escrito en python.

Puntos importantes a tomar en cuenta:

- Complejidad: $O(n^2)$ en el peor caso.
- Uso: Eficiente para listas pequeñas o casi ordenadas.

Ordenamiento por mezcla o fusión (Merge sort).

Uno de los algoritmos más conocidos que emplea "Divide y Vencerás" es el Merge Sort, que ordena una lista dividiéndola en mitades, ordenando cada una de ellas y combinando los resultados.

Es como ordenar fichas de dominó dividiéndolas en grupos pequeños, ordenando cada grupo y luego uniéndolos hasta que todas las fichas estén acomodadas.

Funcionamiento:

1. Se divide la lista en dos mitades.
2. Se aplica recursivamente el mismo procedimiento en cada mitad hasta que queden listas de un solo elemento.
3. Se combinan las listas ordenadas para obtener la lista final.

```
def merge_sort(lista):
    if len(lista) > 1:
        mitad = len(lista)//2
        izquierdo = lista[:mitad]
        derecho = lista[mitad:]
        merge_sort(izquierdo)
        merge_sort(derecho)

        d = 0
        i = 0
        l = 0

        while i < len(izquierdo) and d < len(derecho):
            if izquierdo[i] < derecho[d]:
                lista[l] = izquierdo[i]
                i+=1
            else:
                lista[k] = derecho[d]
                d+=1
            l+=1
        while i < len(izquierdo):
            lista[l] = izquierdo[i]
            i+=1
            k+=1
        while d < len(derecho):
            lista[l] = derecho[d]
```

Código escrito en python.

Puntos importantes a tomar en cuenta:

- Complejidad: $O(n \log n)$.
- Uso: Muy eficiente para listas grandes.

Ordenamiento rápido (Quick sort).

Usa Divide y Vencerás, eligiendo un pivote y separando los elementos menores a la izquierda y los mayores a la derecha.

Es como organizar estudiantes en una fila según altura, eligiendo un punto de referencia y moviendo los más bajos a la izquierda y los más altos a la derecha.

```
def ordenamiento_rapido(lista):
    pivote = lista[0]
    izquierdo = [ ]
    derecho = [ ]
    centro = [ ]

    for l in range(1, len(lista)):
        if lista[l] < pivote:
            izquierdo.append(lista[l])
        elif lista[l] > pivote:
            derecho.append(lista[l])
        else:
            centro.append(lista[l])

    return ordenamiento_rapido(izquierdo) + centro + ordenamiento_rapido(derecho)
```

Código escrito en python.

Puntos a tomar en cuenta:

- Complejidad: $O(n \log n)$ en el mejor caso, $O(n^2)$ en el peor.
- Uso: Es el más rápido en la práctica para listas desordenadas.

7. Algoritmos de búsqueda

Búsqueda secuencial y su implementación.

La búsqueda secuencial (o lineal) es el método más sencillo para encontrar un elemento dentro de una lista o estructura de datos. Consiste en recorrer uno por uno los elementos, comparando cada uno con el valor que se desea encontrar, hasta hallarlo o llegar al final sin éxito.

Imagina que tienes una lista de asistencia en papel, y buscas el nombre de un estudiante. Vas leyendo fila por fila hasta encontrarlo o llegar al final. Ese es el principio de la búsqueda secuencial.

```
def busqueda_secuencial(lista, objetivo):
    hallazgo = (False, None)
    for posicion in range(len(lista)):
        if objetivo == lista[posicion]:
            hallazgo = (True, posicion)
            break
    return hallazgo

lista = [3,6,8,1,4]
print(busqueda_secuencial(lista, 0))
```

Código escrito en python.

Ventajas

- Fácil de implementar.
- No requiere que los datos estén ordenados.

Desventajas

- Poco eficiente para listas grandes.
- En el peor de los casos, recorre toda la lista.

Búsqueda binaria y su optimización.

La búsqueda binaria es un algoritmo más eficiente, pero requiere que la lista esté previamente ordenada. Divide la lista en mitades y descarta sistemáticamente la mitad donde no puede estar el objetivo, hasta encontrarlo o agotar las posibilidades.

Es como buscar una palabra en un diccionario. No lo haces hoja por hoja, sino que abres por la mitad, ves si tu palabra está antes o después, y vuelves a dividir. Así encuentras el objetivo rápidamente.

```
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista)-1
    hallazgo = (False, None, None)

    while inicio <= fin and not hallazgo[0]:
        medio = (inicio + fin)//2
        print(objetivo, lista[medio])
        input()
        if objetivo == lista[medio]:
            hallazgo = (True, medio, objetivo)
        elif objetivo > lista[medio]:
            inicio = medio + 1
        else:
            fin = medio - 1

    return hallazgo

lista = [1,2,3,4,5,6,7,8,9]
print(busqueda_binaria(lista,1))
```

Código escrito en python.

Ventajas

- Mucho más rápida que la secuencial para listas grandes.
- Excelente rendimiento logarítmico ($O(\log n)$).

Desventajas

- Solo funciona si los datos están ordenados.

8. Paradigmas de programación

Programación Imperativa

La programación imperativa es un paradigma que se basa en dar instrucciones paso a paso para modificar el estado de un programa. El enfoque principal es "cómo" lograr un resultado, es decir, se le dice a la computadora cómo debe hacer las cosas.

Características clave:

- Se enfoca en secuencia de instrucciones.
- Utiliza variables, estructuras de control (if, while, for).
- Cambia el estado del programa a través de asignaciones y operaciones.
- Cercana al funcionamiento del hardware (modelo de máquina de Von Neumann).

Es como dar una receta de cocina paso a paso: primero prende la estufa, luego pon el sartén, después rompe los huevos, etc.

```
x = 5
y = 3
suma = x + y
print("El resultado es:", suma)
```

Código escrito en python.

Programación Estructurada

Es una extensión de la programación imperativa, que propone el uso de estructuras de control claras y organizadas, como secuencias, decisiones y bucles. Se desarrolló para evitar los problemas de los programas desorganizados (el famoso "código espagueti").

Características clave:

- Basada en estructuras de control: secuencia, selección (if), repetición (while, for).
- Rechaza el uso del goto.
- Favorece un código legible y mantenible.
- Usa funciones para evitar repeticiones.

Es como un manual de instrucciones con capítulos bien definidos y numerados, evitando saltos caóticos de página en página.

```
def calcular_area_triangulo(base, altura):
    return (base * altura) / 2

b = 10
h = 5
area = calcular_area_triangulo(b, h)
print("Área del triángulo:", area)
```

Código escrito en python.

Programación Modular

La programación modular lleva la estructuración un paso más allá. Divide un programa grande en módulos independientes, cada uno encargado de una tarea específica. Estos módulos pueden ser reutilizados, probados y mantenidos por separado.

Características clave:

- Un programa se divide en bloques funcionales.
- Cada módulo tiene una función clara y definida.
- Favorece la reutilización de código.
- Reduce la complejidad: "divide y vencerás".
- Facilita la colaboración en equipo.

Es como construir una casa con bloques LEGO: cada pieza tiene una forma y función específica, y se puede reemplazar sin destruir el resto de la estructura.

```
#Nombre del archivo: operaciones.py
def suma(a, b):
    return a + b

def resta(a, b):
    return a - b
```

Código escrito en python.

```
#Nombre del archivo: script_principal.py
import operaciones

x = 8
y = 3
print("Suma:", operaciones.suma(x, y))
print("Resta:", operaciones.resta(x, y))
```

Código escrito en python.

Programación Orientada a Objetos (POO)

En los primeros acercamientos al diseño de algoritmos, como se propone en Metodología de la Programación de Osvaldo Cairo, el análisis del problema parte de identificar con claridad tres elementos fundamentales:

- **Datos de entrada** → ¿Qué necesito para iniciar el proceso?
- **Procesos** → ¿Qué operaciones debo realizar?
- **Datos de salida** → ¿Qué quiero obtener?

Este enfoque es directo, lógico y adecuado para resolver problemas de forma secuencial y estructurada. Se trata de entender un sistema como una serie de pasos que se ejecutan para transformar entradas en salidas.

Cuando damos el salto a la Programación Orientada a Objetos, ya no analizamos solo el proceso como una secuencia de pasos, sino que empezamos a organizar el problema en torno a los entes que lo componen. Es decir, en lugar de preguntarnos solamente "¿qué hace el sistema?", ahora nos preguntamos:

- **¿De quién se habla?** → Esto nos lleva a identificar las clases y objetos (los protagonistas del problema).
- **¿Cómo es?** → Aquí definimos las propiedades o atributos que caracterizan al objeto.
- **¿Qué hace?** → Nos enfocamos en los métodos o comportamientos asociados al objeto.

La programación orientada a objetos es un paradigma que se basa en el concepto de "objetos", los cuales agrupan datos y comportamientos relacionados. Cada objeto es una instancia de una clase, que actúa como una plantilla o molde.

Características clave:

- Encapsulamiento: protege los datos dentro del objeto.
- Abstracción: oculta detalles internos y muestra solo lo necesario.
- Herencia: una clase puede heredar propiedades y métodos de otra.
- Polimorfismo: los métodos pueden comportarse de diferentes maneras según el objeto que los utilice.

Ejemplo aplicado

Supongamos que queremos resolver un problema sobre estudiantes y calificaciones.

- **Enfoque clásico:**
 - Entrada: nombre, calificación1, calificación2, calificación3.
 - Proceso: calcular promedio.
 - Salida: mostrar promedio.
- **Enfoque POO:**
 - ¿De quién se habla? → De un estudiante.
 - ¿Cómo es? → Tiene un nombre y tres calificaciones.
 - ¿Qué hace? → Puede calcular su promedio y mostrarlo.

```

class Estudiante:
    def __init__(self, nombre, cal1, cal2, cal3):
        self.nombre = nombre #propiedades o atributos
        self.cal1 = cal1
        self.cal2 = cal2
        self.cal3 = cal3

    def calcular_promedio(self): #método o comportamiento
        return (self.cal1 + self.cal2 + self.cal3) / 3

    def mostrar(self): #método o comportamiento
        print(f"{self.nombre} tiene un promedio de {self.calcular_promedio():.2f}")

estudiante1 = Estudiante("Andi",70,80,90) #Construcción del objeto
estudiante2 = Estudiante("Joz",75,85,95)

estudiante1.calcular_promedio() #Llamando al método del objeto estudiante 1 "Andi"
estudiante1.mostrar()

estudiante2.calcular_promedio() #Llamando al método del objeto estudiante 2 "Joz"
estudiante2.mostrar()

```

Código escrito en python.

Con la programación orientada a objetos, buscamos imitar cómo las cosas funcionan en el mundo real, construyendo un modelo que sea más natural, más reutilizable y más fácil de mantener. Esto no sustituye al pensamiento lógico de los algoritmos clásicos, sino que lo complementa y lo organiza desde una nueva perspectiva.

Programación Multihilos

Es una técnica que permite que varias tareas se ejecuten simultáneamente dentro de un mismo programa, utilizando hilos (threads). Cada hilo representa una línea de ejecución independiente, lo cual es útil para programas que requieren eficiencia y respuesta en tiempo real.

Imagina una marioneta tradicional:

- Un solo titiritero tiene que mover una cuerda a la vez.
- Si quiere que la marioneta camine, primero mueve un pie, luego el otro, luego los brazos, uno por uno.
- Esto representa un programa monohilo: ejecuta las instrucciones de forma secuencial, paso a paso.

Ahora, imagina que cada cuerda está conectada a una mano diferente. Cada mano puede mover una parte de la marioneta al mismo tiempo: una levanta un pie, otra mueve la cabeza, otra agita un brazo.

¡La marioneta cobra vida de forma mucho más fluida y rápida!

Eso es la programación multihilos: varios hilos de ejecución trabajando al mismo tiempo dentro de un mismo programa.

Características clave:

- Aumenta la eficiencia en sistemas con múltiples núcleos.
- Permite mantener una interfaz activa mientras se procesan tareas.
- Requiere manejar con cuidado el acceso a recursos compartidos (sincronización).

Puntos a tomar en cuenta:

- Aumenta la eficiencia en sistemas con múltiples núcleos.
- Permite mantener una interfaz activa mientras se procesan tareas.
- Requiere manejar con cuidado el acceso a recursos compartidos (sincronización).
- Python usa el GIL (Global Interpreter Lock), lo cual significa que los hilos no pueden ejecutarse verdaderamente en paralelo en todos los casos (particularmente cuando todos los hilos usan CPU intensamente).
- Los hilos comparten memoria, por lo que hay riesgo de errores por interferencia si no se manejan bien (por ejemplo, dos hilos intentando modificar la misma variable al mismo tiempo).
- No todos los problemas se benefician del uso de múltiples hilos.

```
import threading
import time

def saludar():
    for i in range(3):
        print("¡Hola!")
        time.sleep(1)

def despedir():
    for i in range(3):
        print("¡Adiós!")
        time.sleep(1)

# Crear dos hilos
hilo1 = threading.Thread(target=saludar)
hilo2 = threading.Thread(target=despedir)

# Iniciar los hilos
hilo1.start()
hilo2.start()

# Esperar que terminen
hilo1.join()
hilo2.join()
```

Código escrito en python.

Multiparadigma:

Es un enfoque de programación que combina varios paradigmas en un mismo lenguaje o programa. En lugar de limitarse a una sola forma de pensar (por ejemplo, solo imperativa o solo orientada a objetos), permite elegir la más adecuada según el problema.

Python es un lenguaje multiparadigma

Es como una navaja suiza: tiene herramientas para cortar, abrir, atornillar... Tú eliges cuál usar dependiendo de la situación. La programación multiparadigma te da la libertad de mezclar herramientas, mientras mantengas coherencia.

El conocimiento de estos paradigmas permite que el programador:

- Elija la estrategia adecuada según el problema.
- Escriba código más flexible, escalable y reutilizable.
- Mejore su pensamiento algorítmico y sus habilidades de diseño.

Una pausa para mirar atras

Antes de continuar, es importante detenerse un momento y reconocer todo lo que has logrado hasta ahora. Al comenzar este curso, venías con las herramientas de la Metodología de la Programación, entendiendo cómo se resuelven problemas desde lo lógico: entradas, procesos y salidas. Eso sentó una base sólida.

Pero el paso que diste después fue aún más grande: aprender a programar en un lenguaje real, Python, y hacerlo desde lo más básico, sin depender de entornos sofisticados. Usaste el bloc de notas y la línea de comandos para enfocar tu atención en lo verdaderamente importante: la lógica y la estructura del código.

Durante este recorrido, ya no solo sabes qué es una variable o una lista. Has aprendido:

- A usar funciones como bloques reutilizables.
- A ordenar datos, tomar decisiones, repetir procesos.
- A entender paradigmas de programación que antes ni sabías que existían.
- A resolver problemas usando técnicas como recursividad y "divide y vencerás".
- A leer estructuras complejas como listas anidadas o diccionarios dentro de diccionarios.
- Incluso a comprender conceptos como programación orientada a objetos y programación concurrente.

Eso ya no es básico. Es intermedio. Y lo lograste paso a paso.

A veces, mientras aprendemos, no notamos cuánto hemos crecido. Por eso este momento es importante: para mirar atrás y darte cuenta de que ya piensas como un programador. No solo sabes "escribir código", sabes resolver problemas y estructurar soluciones.

Lo que sigue no será más fácil, pero ahora estás mejor preparado.

Pronto hablaremos de:

- Lectura y escritura de archivos,
- Expresiones regulares para manipular texto,
- APIs y estructuras JSON,
- Manipulación de hojas de cálculo,
- Y generación de gráficos que comuniquen información.

Lo harás con la misma lógica y disciplina que has desarrollado hasta ahora.

Porque ya no estás empezando.

Ya formas parte del mundo de la programación.

Fase III: Aplicación de la Programación en Ciencia y Tecnología

Objetivo de la Fase: El objetivo de esta fase es que el estudiante aplique Python en la resolución de problemas del mundo real, especialmente en áreas científicas, tecnológicas y académicas. Para ello, se introducen herramientas y bibliotecas que permiten el manejo de archivos, automatización de tareas, acceso a información estructurada y visualización de datos. Se trata de un salto de la programación puramente estructurada a la programación aplicada, donde Python se convierte en una herramienta poderosa para procesar información, extraer conclusiones y generar soluciones automáticas.

1. Automatización de tareas con Python

¿Qué es la automatización?

Automatizar tareas significa crear scripts o programas que realicen por nosotros actividades repetitivas o que, de otra forma, requerirían intervención humana constante. Esto permite ahorrar tiempo, evitar errores manuales y liberar al usuario para enfocarse en tareas más importantes o creativas.

Es como programar un robot que hace tu cama todas las mañanas: una vez que aprende cómo hacerlo, no necesitas decirle nada más. Solo presionas un botón (o ni eso), y el trabajo se realiza.

Casos comunes de automatización con Python:

- Renombrar archivos en masa.
- Mover o copiar archivos entre carpetas automáticamente.
- Leer hojas de cálculo y enviar correos con reportes automáticos.
- Obtener datos de páginas web de forma automática.
- Generar gráficos con estadísticas actualizadas.

Herramientas clave:

- `os` y `shutil` para gestionar archivos y directorios.
- `time` para programar tareas en intervalos específicos.
- `schedule` para programar tareas periódicas.
- `pyautogui` para simular movimientos del mouse o teclado.
- `smtpplib` para enviar correos electrónicos desde scripts.

Uso de scripts para la automatización de procesos repetitivos

Un script en Python es un archivo `.py` que contiene un conjunto de instrucciones. Ejecutarlo equivale a darle la orden al sistema de que realice cada paso escrito, desde lo más simple como abrir un archivo, hasta lo más complejo como conectarse a internet, extraer datos y generar un gráfico.

Ejemplo de proceso repetitivo:

Imagina que cada semana debes descargar un archivo, abrirlo, copiar ciertos datos y generar un gráfico en Excel. Esto puede automatizarse con Python de la siguiente forma:

1. Usar `urllib.request` o `requests` para descargar el archivo.
2. Leerlo con `pandas`.
3. Hacer cálculos con `numpy`.
4. Visualizarlo con `matplotlib` o `seaborn`.
5. Guardar el resultado como imagen o Excel con `openpyxl`.

Cada paso puede escribirse en un script que se ejecuta automáticamente con un doble clic o programado con una tarea del sistema operativo.

Scripts para tareas web y uso de lenguajes de marcado (HTML, XML).

¿Qué es un lenguaje de marcado?

Los lenguajes de marcado como HTML y XML son formas de representar datos estructurados. HTML define cómo se ve y organiza una página web; XML describe datos en un formato que puede ser leído por personas y máquinas.

Relación con Python

Python no solo es capaz de leer este tipo de documentos, sino que también puede extraer información de ellos, modificarlos o incluso automatizar su generación.

Herramientas clave:

- BeautifulSoup para analizar contenido HTML.
- lxml para analizar contenido XML.
- requests para acceder a sitios web.
- json para manejar datos estructurados de forma moderna.
- re para buscar patrones específicos en textos web.

Esta etapa marca el inicio del uso de Python como herramienta de trabajo profesional. El estudiante ya no solo conoce la sintaxis del lenguaje, sino que ahora puede usarlo para resolver problemas reales, automatizar procesos tediosos y extraer conocimiento útil de datos estructurados o no estructurados. Esta fase establece las bases para el trabajo interdisciplinario, donde Python se convierte en un aliado del matemático, físico, ingeniero, científico de datos o tecnólogo.

2. Manejo de archivos

En la programación real, gran parte de la información que se procesa no se encuentra escrita dentro del código, sino que proviene de archivos externos. Estos archivos pueden contener datos numéricos, texto, registros, configuraciones, etc. Python permite acceder a estos archivos, leer su contenido, modificarlo o crear nuevos archivos, todo mediante instrucciones sencillas y potentes.

Manipular archivos con Python es como abrir un cuaderno: puedes leer lo que está escrito, agregar nuevas anotaciones o arrancar páginas enteras para volver a escribirlas.

Lectura y escritura de archivos en Python.

Apertura de archivos con open()

La función open() se utiliza para abrir un archivo y obtener un objeto que lo representa. Tiene la siguiente estructura:

```
archivo = open('ruta/del/archivo.txt', 'modo')
```

Código escrito en python.

Modos de apertura comunes:

Modo	Significado
r'	Lectura (read)
w'	Escritura (write, sobrescribe)
a'	Agregar al final (append)
b'	Modo binario (ej. imágenes)
+'	Lectura y escritura

Ejemplo: Leer un archivo línea por línea

```
archivo = open('datos.txt', 'r')
for linea in archivo:
    print(linea.strip()) # .strip() elimina saltos de línea extra
archivo.close()
```

Código escrito en python.

Siempre recuerda cerrar el archivo al finalizar su uso con close().

Ejemplo: Escribir en un archivo

```
archivo = open('salida.txt', 'w')
archivo.write('Hola, mundo.\n')
archivo.write('Primera línea de salida.')
archivo.close()
```

Código escrito en python.

Si el archivo no existe, se creará. Si ya existe, se sobrescribirá completamente.

Forma segura y moderna: with open()

Usar with asegura que el archivo se cierre automáticamente, incluso si ocurre un error:

```
with open('datos.txt', 'r') as archivo:
    for linea in archivo:
        print(linea.strip())
```

Código escrito en python.

Y para escribir:

```
with open('salida.txt', 'a') as archivo:  
    archivo.write('Nueva línea\n')
```

Código escrito en python.

Manejo de errores y excepciones al manipular archivos.

No siempre los archivos estarán disponibles o en el formato esperado. Por eso, Python nos permite capturar errores mediante bloques try, except.

```
try:  
    with open('datos.txt', 'r') as archivo:  
        contenido = archivo.read()  
        print(contenido)  
except FileNotFoundError:  
    print('El archivo no fue encontrado.')  
except IOError:  
    print('Ocurrió un error al acceder al archivo.')
```

Código escrito en python.

Esta estructura evita que el programa se detenga abruptamente por errores comunes como un archivo inexistente o sin permisos de lectura.

Dominar el manejo de archivos es esencial en cualquier proyecto realista de programación, ya que gran parte de los datos que procesamos están almacenados externamente. Python ofrece una sintaxis clara, pero también requiere que el programador actúe con cuidado y orden, especialmente al manejar errores o manipular archivos grandes o críticos.

3. Expresiones regulares y manejo de texto

Cuando se trabaja con texto en Python, muchas veces no basta con usar métodos básicos como `.find()` o `.replace()`. En escenarios más complejos, como validar correos electrónicos, extraer fechas, identificar nombres o analizar archivos con formatos particulares, necesitamos una herramienta más poderosa y flexible: las expresiones regulares (regular expressions o regex).

Las expresiones regulares son como detectores de metales que nos ayudan a encontrar patrones específicos dentro de montañas de texto. No buscan palabras exactas, sino estructuras.

Python incluye el módulo `re` que nos permite trabajar con expresiones regulares.

¿Qué es una expresión regular?

Una expresión regular es una secuencia de caracteres que define un patrón de búsqueda. Este patrón puede utilizarse para verificar, buscar, extraer, dividir o reemplazar partes del texto.

Ejemplo de un patrón: `\d{3}-\d{2}-\d{4}`

Este patrón buscaría una cadena con el formato de un número como: 123-45-6789.

Sintaxis básica de expresiones regulares

Patrón	Significado	Ejemplo coincide con
.	Cualquier carácter excepto salto de línea	a.c → abc, axc, a#c
\d	Un dígito (0-9)	\d\d\d → 123
\w	Caracter alfanumérico o guion bajo	\w+ → palabra, abc123
\s	Espacio en blanco	\s → espacio, tab, salto
^	Inicio de línea	^Hola → línea que inicia con "Hola"
\$	Fin de línea	mundo\$ → termina con "mundo"
*	0 o más repeticiones	a* → "", a, aaa
+	1 o más repeticiones	a+ → a, aaa
{n}	Exactamente n repeticiones	\d{2} → 23, 45
[]	Conjunto de caracteres	[aeiou] → vocales

Búsqueda y manipulación de cadenas utilizando expresiones regulares.

Buscar un patrón con search()

Busca una sola coincidencia del patrón dentro del texto. Si encuentra la primera coincidencia, detiene la búsqueda y la devuelve como un objeto especial llamado Match.

Imagina que tienes un documento y le pides a alguien que encuentre el primer párrafo donde aparezca una palabra clave. Cuando la encuentra, deja de buscar.

Sintaxis: re.search(patron, texto)

¿Qué devuelve?

Un objeto Match, que puedes examinar:

- match.group() → el texto que coincidió.
- match.start() → índice donde empieza.
- match.end() → índice donde termina.

Ejemplo:

```
import re

texto = "Mi número es 123-45-6789"
resultado = re.search(r"\d{3}-\d{2}-\d{4}", texto)

if resultado:
    print("¡Coincidencia encontrada!", resultado.group())
```

Código escrito en python.

Buscar todas las coincidencias con findall()

Busca todas las coincidencias del patrón dentro del texto y devuelve una lista con todas las ocurrencias encontradas.

Si `search()` era como un vigilante que te avisa cuando ve la primera coincidencia, `findall()` es como un escáner que detecta todas las apariciones del patrón y te las entrega en una lista.

Sintaxis: `re.findall(patron, texto)`

```
import re

texto = "Correos: ana@mail.com, juan@gmail.com"
patron = r"\w+@\w+\.\w+"
emails = re.findall(patron, texto)
print(emails)
```

Código escrito en python.

Característica	<code>search()</code>	<code>findall()</code>
¿Cuántas coincidencias busca?	Solo la primera	Todas las coincidencias
¿Qué devuelve?	Objeto Match	Lista de cadenas
Uso típico	Verificar si un patrón aparece	Extraer múltiples ocurrencias

¿Cuándo usar uno u otro?

- Usa `search()` si solo te interesa saber si el patrón aparece o dónde aparece por primera vez.
- Usa `findall()` si necesitas todas las ocurrencias de ese patrón, por ejemplo, todas las fechas, correos, números, etc.

Aplicaciones comunes en ciencia y tecnología

- Validación de datos (correos, contraseñas, nombres).
- Limpieza de datos en ciencia de datos.
- Extracción de información de páginas web.
- Procesamiento de archivos de texto masivos.

Ejemplo real: Extraer números de un archivo de sensores donde las líneas contienen etiquetas y datos mezclados.

Las expresiones regulares son una herramienta muy poderosa en programación, especialmente cuando se necesita identificar, validar o transformar patrones de texto en grandes volúmenes de información. Aunque pueden parecer difíciles al principio, su aprendizaje representa un gran paso hacia una programación más profesional y eficiente.

¿Qué son los patrones en expresiones regulares?

Un patrón es una regla escrita con símbolos especiales que nos permite identificar coincidencias dentro de una cadena de texto. Estos patrones pueden ser generales o muy específicos, dependiendo de lo que se quiere encontrar.

Símbolo	Significado	Equivalente en conjunto	Descripción
\d	Un dígito (0-9)	[0-9]	Cualquier número del 0 al 9
\D	Cualquier no dígito	[^0-9]	Todo menos números
\w	Un carácter alfanumérico o guion bajo	[a-zA-Z0-9_]	Letras, números y el guion bajo _
\W	Cualquier no alfanumérico	[^a-zA-Z0-9_]	Todo menos letras, números y _
\s	Cualquier espacio en blanco	[\t\n\r\f\v]	Espacio, tabulación, salto de línea...
\S	Cualquier no espacio en blanco	[^\t\n\r\f\v]	Cualquier carácter visible

Nota: Los conjuntos entre corchetes [] son más personalizables, pero más largos de escribir. Los símbolos como \d son más concisos y se usan con más frecuencia cuando buscamos tipos de caracteres comunes.

Ejemplos prácticos

Buscar un número de teléfono simple (3 dígitos - 3 dígitos - 4 dígitos)

```
import re

texto = "Mi número es 123-456-7890"
patron1 = r"\d{3}-\d{3}-\d{4}" # usando \d
patron2 = r"[0-9]{3}-[0-9]{3}-[0-9]{4}" # usando conjuntos

print(re.search(patron1, texto).group())
print(re.search(patron2, texto).group())
```

Código escrito en python.

Ambos patrones hacen exactamente lo mismo, pero el primero es más legible y fácil de mantener.

Validar nombres de usuario (solo letras, números y guion bajo)

```
import re

usuario = "ana_123"
patron1 = r"^\w+$"          # usando \w
patron2 = r"^[a-zA-Z0-9_]+$" # usando conjuntos

print(bool(re.match(patron1, usuario)))
print(bool(re.match(patron2, usuario)))
```

Código escrito en python.

\w incluye el guion bajo _, lo cual lo hace ideal para nombres de usuario tipo redes sociales o identificadores en código.

¿Cuándo usar cuál?

Usar \d, \w, \s...	Usar conjuntos []
Cuando el patrón es común y estándar	Cuando necesitas control más específico
Para hacer código más legible y corto	Para incluir/excluir caracteres muy concretos
En combinaciones rápidas de validación (emails, contraseñas, códigos)	Cuando trabajas con alfabetos personalizados o reglas complejas

También es válido combinar \d, \w, \s conjuntamente con conjuntos y otras estructuras:

```
patron = r"\w+@\w+\.(com|mx|org)"
```

Código escrito en python.

Este patrón acepta correos con dominios .com, .mx, o .org.

Asterisco * → “Cero o más veces”

¿Qué significa?

El símbolo * indica que el carácter o grupo anterior puede repetirse cero, una o muchas veces.

```
import re

texto = "Laaaaana"
patron = r"La*na" # "a" puede estar 0 o más veces

coincidencia = re.search(patron, texto)
print(coincidencia.group()) # Salida: Laaaaana
```

Código escrito en python.

- Coincidiría también con: Lana, Lna, Laana, etc.
- No coincidiría con: Lina, Lona (porque no tienen una "a" directamente después de la "L").

Signo de más + → “Una o más veces”

¿Qué significa?

El símbolo + exige que el carácter o grupo anterior aparezca al menos una vez, pero puede repetirse muchas veces.

```
import re

texto = "Laaaaana"
patron = r"La+na" # "a" debe aparecer al menos una vez

coincidencia = re.search(patron, texto)
print(coincidencia.group()) # Salida: Laaaaana
```

Código escrito en python.

- Coincidiría con: Lana, Laana, Laaaaana
- No coincidiría con: Lna (porque falta al menos una "a")

Barra vertical | → “O (alternativa lógica)”

¿Qué significa?

El símbolo | actúa como un “o lógico”. Permite buscar una coincidencia entre varias alternativas.

```
import re

texto = "Hoy es lunes"
patron = r"lunes|martes|miércoles"

coincidencia = re.search(patron, texto)
print(coincidencia.group()) # Salida: lunes
```

Código escrito en python.

- Coincide con el primer valor que se cumpla.
- Puedes usarlo dentro de un grupo también: (abc|def|ghi)

También puedes combinarlo todo

```
import re

texto = "abc123"
patron = r"(abc|xyz)+\d*"

# abc o xyz deben aparecer al menos una vez, seguidos de cero o más dígitos
coincidencia = re.match(patron, texto)
print(coincidencia.group()) # Salida: abc123
```

Código escrito en python.

Comparación rápida

Símbolo	Significado	Ejemplo	Coincide con
*	Cero o más repeticiones	ho*la	hla, hola, hoolaa, etc.
+	Una o más repeticiones	ho+la	hola, hoolaa (pero no hla)
		Alternativa lógica (“o”)	`perro

A manera de repaso

- **Asterisco** : * es como decir: "puede o no puede estar, no me importa cuántas veces."
- **Signo de más**: + es como decir: "mínimo una vez, pero más veces también está bien."
- **Barra vertical**: | es como decir: "acepto esta opción o esta otra."

4. APIs y formato JSON

Introducción a las APIs y cómo interactuar con ellas utilizando Python.

Una API (Application Programming Interface) es un conjunto de reglas y protocolos que permiten que dos aplicaciones se comuniquen entre sí. En términos simples, una API es como un mesero en un restaurante:

- Tú (el cliente) haces un pedido (una solicitud).
- El mesero (la API) lleva tu pedido a la cocina (el servidor).
- Luego te devuelve lo que pediste (la respuesta).

Así funcionan muchas aplicaciones que usamos a diario: cuando ves el clima en tu teléfono, cuando haces una compra en línea, o cuando ves un mapa. Detrás de todo eso, hay APIs comunicándose y compartiendo información entre servicios.

¿Cómo interactuar con una API en Python?

En Python, usamos módulos como requests para enviar solicitudes HTTP a una API. Las solicitudes más comunes son:

- GET: Pedir información.
- POST: Enviar información.

```
import requests

url = "https://jsonplaceholder.typicode.com/todos/1"
respuesta = requests.get(url)

print(respuesta.status_code) # Verifica si fue exitosa (200)
print(respuesta.json())      # Imprime el contenido en formato JSON
```

Código escrito en python.

Este código hace una solicitud a una API falsa de prueba y muestra los datos que devuelve.

Manejo de datos en formato JSON para intercambio de información.

¿Qué es JSON?

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos. Se usa comúnmente en APIs porque es fácil de leer por humanos y fácil de procesar por las máquinas.

JSON en Python se representa con diccionarios:

```
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false #Cuidado! En python se escribe False. ¿Ves que se parece mucho?
}
```

JSON

Cargar y trabajar con JSON en Python

Leer JSON desde un string o archivo:

```
import json

# Desde una cadena
json_texto = '{"nombre": "Ana", "edad": 25}'
datos = json.loads(json_texto) # Convierte string JSON a diccionario
print(datos["nombre"]) # Ana

# Desde un archivo
with open('archivo.json', 'r') as archivo:
    contenido = json.load(archivo)
```

Código escrito en python.

Escribir JSON a un archivo:

```
import json

datos = {"curso": "Python", "nivel": "intermedio"}
with open('salida.json', 'w') as archivo:
    json.dump(datos, archivo, indent=4)
```

Código escrito en python.

JSON y APIs: ¿cómo se conectan?

Las respuestas de las APIs suelen estar en formato JSON. Python convierte automáticamente estas respuestas en diccionarios que puedes manipular como cualquier otro dato en tu programa.

Ejemplo

```
import requests

respuesta = requests.get("https://jsonplaceholder.typicode.com/users/1")
usuario = respuesta.json()

print(usuario["name"]) # Imprime el nombre del usuario
print(usuario["email"]) # Imprime el correo electrónico
```

Código escrito en python.

Entender APIs y JSON es como abrir la puerta a un mundo interconectado. Es el primer paso para trabajar con servicios externos, integrar datos del clima, enviar mensajes en redes sociales, automatizar tareas, o incluso construir tus propias aplicaciones web y móviles.

5. Matemáticas con Python

Python es un lenguaje ideal para trabajar con matemáticas, no solo por su sintaxis clara, sino porque cuenta con bibliotecas que amplían sus capacidades para hacer cálculos numéricos, simbólicos y estadísticos. En esta sección abordaremos operaciones con fracciones, números complejos, estadística básica y cálculo simbólico.

Cálculos con fracciones y números complejos.

Para trabajar con fracciones exactas, usamos la clase Fraction del módulo fractions.

```
from fractions import Fraction

a = Fraction(3, 4)
b = Fraction(2, 5)

suma = a + b
print(suma) # 23/20
```

Código escrito en python.

Esto evita los errores de redondeo comunes con números decimales.

Números complejos

En Python, un número complejo se representa como $a + bj$, donde j es la unidad imaginaria.

```
z1 = 2 + 3j
z2 = 1 - 4j

producto = z1 * z2
print(producto) # (14-5j)
```

Código escrito en python.

Python puede realizar operaciones como suma, resta, multiplicación, división y obtener el conjugado o el módulo de un número complejo.

Aplicación de métodos numéricos con statistics.

Para cálculos estadísticos usamos el módulo statistics.

```
import statistics

datos = [2, 4, 4, 4, 5, 5, 7, 9]

print("Media:", statistics.mean(datos))
print("Mediana:", statistics.median(datos))
print("Moda:", statistics.mode(datos))
```

Código escrito en python.

Podemos usar un Counter de collections para contar la frecuencia de elementos:

```
from collections import Counter

frecuencias = Counter(datos)
print(frecuencias)
```

Código escrito en python.

Esto devuelve un diccionario con la cantidad de veces que aparece cada número. Es útil para construir histogramas y análisis de distribuciones.

Operaciones simbólicas con SymPy.

SymPy es una biblioteca para matemáticas simbólicas, es decir, trabaja con expresiones como si fueran álgebra pura. A diferencia de los cálculos numéricos, aquí se opera con símbolos.

```
from sympy import symbols, simplify, expand

x, y = symbols('x y')
expresion = (x + y)**2

print("Expandido:", expand(expresion))
print("Simplificado:", simplify(x**2 + 2*x + 1 - (x+1)**2))
```

Código escrito en python.

Límites

```
from sympy import limit

x = symbols('x')
f = (x**2 - 1) / (x - 1)

print(limit(f, x, 1)) # Resultado: 2
```

Código escrito en python.

Derivadas

```
from sympy import diff

f = x**3 + 2*x**2 - x + 5
derivada = diff(f, x)

print("Derivada:", derivada)
```

Código escrito en python.

Integrales

```
from sympy import integrate

f = x**2
integral = integrate(f, x)

print("Integral indefinida:", integral)

# Integral definida de 0 a 2
definida = integrate(f, (x, 0, 2))
print("Integral definida de 0 a 2:", definida)
```

Código escrito en python.

Podemos pensar en SymPy como una especie de "cuaderno mágico de matemáticas", donde no solo escribimos fórmulas, sino que podemos pedirle que las desarrolle, derive, integre y resuelva por nosotros. Para los estudiantes de ciencia y tecnología, dominar este tipo de herramientas puede significar ahorrar tiempo y evitar errores humanos en cálculos complejos.

6. Manejo de hojas de cálculo con Openpyxl

openpyxl es una biblioteca de Python que permite leer, escribir y modificar archivos de Excel (.xlsx). A diferencia del formato .csv, que es plano, Excel permite hojas múltiples, celdas con formato, fórmulas y más.

Lectura, manipulación y creación de archivos Excel con módulo Openpyxl.

Aplicación práctica

Imagina que debes enviar reportes semanales con información de ventas, asistencia o resultados de laboratorio. Con openpyxl, puedes automatizar este proceso:

- Leer archivos anteriores.
- Agregar nuevas filas.
- Calcular promedios o totales.
- Guardar y renombrar archivos por fecha.
- Enviar por correo electrónico con otro script.

Podemos pensar en openpyxl como un robot oficinista: sabe abrir tu archivo de Excel, buscar celdas, escribir nombres o cantidades, actualizar valores antiguos y guardar el documento limpio y listo para ser compartido. Tú das las órdenes con Python, y él ejecuta el trabajo mecánico.

7. Gráficas con Matplotlib y SymPy

Una gráfica puede revelar relaciones que no son evidentes en una tabla o en cálculos. La visualización es una herramienta poderosa en ciencia, ingeniería, economía, educación y más.

Cuando se generan gráficas para documentos de investigación científica o técnica, es fundamental seguir ciertas convenciones que aseguren claridad, legibilidad y profesionalismo. Estas convenciones están alineadas con normas internacionales como las del IEEE, APA, ISO y recomendaciones generales para publicaciones académicas (como Elsevier, Springer, Nature, etc.).

A continuación, te explico las partes clave que debe contener una gráfica para cumplir con estos estándares:

Partes de una Gráfica según estándares internacionales

1. Título claro y descriptivo

- Debe estar centrado arriba de la gráfica.
- Resume de forma breve qué muestra la gráfica.
- Evita títulos genéricos como “Gráfica 1”.

Ejemplo:

- Relación entre la temperatura y el tiempo de ebullición del agua a diferentes altitudes

2. Ejes etiquetados correctamente

- Ambos ejes (X e Y) deben incluir:
- Nombre de la variable
- Unidad de medida entre paréntesis o al lado (si aplica)
- Usa notación científica cuando sea necesario.

Ejemplo:

- Eje X: Tiempo (s)
- Eje Y: Temperatura (°C)

3. Escala proporcional y legible

- Las escalas deben ser uniformes y permitir una lectura intuitiva.
- Evita sobrecargar con demasiados ticks o números decimales innecesarios.

4. Leyenda (si hay múltiples elementos)

- Indica qué representa cada curva, línea o barra.
- Usa colores o estilos claramente distinguibles (líneas sólidas, punteadas, etc.).
- Coloca la leyenda dentro de la gráfica sin cubrir los datos, o en una posición externa clara.

5. Fuente de los datos (opcional, pero recomendada)

- Si los datos provienen de una base externa o experimento, se puede incluir una pequeña cita debajo o como nota.

Ejemplo:

- Fuente: Elaboración propia con base en datos del INEGI (2024)

6. Notas aclaratorias (si aplica)

- Incluir abreviaturas, condiciones del experimento o suposiciones importantes.

Ejemplo:

- Nota: La presión atmosférica se mantuvo constante en 1 atm.

7. Figura numerada y pie de figura (en el cuerpo del texto)

- En documentos académicos, cada gráfica se inserta como una “Figura X” y se acompaña de un pie de figura con una explicación.
- El pie de figura va debajo y debe ser preciso y autosuficiente.

Ejemplo:

- Figura 2. Variación del pH durante el crecimiento bacteriano en un medio ácido a 37 °C.

Recomendaciones adicionales

- Usa tipografías legibles y tamaños adecuados (entre 9 y 12 pt en etiquetas).
- Colores con buen contraste (evita combinaciones confusas como rojo-verde si no son accesibles para todos).
- Evita decoración innecesaria como efectos 3D o sombras.
- Siempre usa un formato vectorial (SVG, PDF, EPS) para publicaciones, y PNG de alta resolución si es imagen.

Creación de gráficos simples y avanzados con Matplotlib.

Instalación: `pip install matplotlib`

Gráfico de línea (el más común)

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.plot(x, y)
plt.title("Gráfico de línea")
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.grid(True)
plt.show()
```

Código escrito en python.

Gráfico de barras

```
import matplotlib.pyplot as plt

etiquetas = ['A', 'B', 'C']
valores = [10, 15, 7]

plt.bar(etiquetas, valores, color='orange')
plt.title("Gráfico de barras")
plt.show()
```

Código escrito en python.

Gráfico de pastel (pie)

```
import matplotlib.pyplot as plt

porcentajes = [40, 35, 25]
categorías = ['Python', 'C++', 'Java']

plt.pie(porcentajes, labels=categorías, autopct='%1.1f%%')
plt.title("Lenguajes más usados")
plt.axis('equal')
plt.show()
```

Código escrito en python.

Personalización de gráficas

- Colores: 'red', 'blue', 'green', '#hex'
- Estilos de línea: '--' (guiones), ':' (puntos)
- Marcadores: 'o', 's', '^'

Graficación de fórmulas matemáticas utilizando SymPy.

SymPy es una biblioteca para matemáticas simbólicas. Nos permite graficar funciones exactas, sin necesidad de evaluarlas manualmente.

Instalación: `pip install sympy`

Ejemplo básico

```
import sympy as sp
from sympy.plotting import plot

x = sp.Symbol('x')
función = x**2 + 2*x + 1

plot(función, (x, -10, 10), title="Gráfica de una parábola")
```

Código escrito en python

Funciones trigonométricas, exponenciales, etc.

```
f = sp.sin(x) * sp.exp(-x/3)
plot(f, (x, 0, 20), title="Gráfica de una función senoidal amortiguada")
```

Código escrito en python

Combinación con Matplotlib

```
from sympy import lambdify
import numpy as np

f = sp.cos(x) / x
f_numérica = lambdify(x, f, "numpy")

x_vals = np.linspace(0.1, 10, 100)
y_vals = f_numérica(x_vals)

plt.plot(x_vals, y_vals)
plt.title("Gráfica de cos(x)/x")
plt.grid(True)
plt.show()
```

Código escrito en python

Piensa que Matplotlib es como un lienzo y tú eres el pintor: puedes usarlo para dibujar líneas, barras, pasteles o cualquier combinación. Por otro lado, SymPy es como un matemático que te da funciones perfectas, que puedes visualizar tal como se escriben en un libro, sin aproximaciones numéricas.

El manejo de gráficas con Python no es solamente una habilidad técnica, sino una herramienta poderosa para traducir datos en conocimiento visualmente comprensible. A través de bibliotecas como Matplotlib y SymPy, los estudiantes y profesionales pueden transformar números abstractos en representaciones gráficas claras, precisas y estéticamente profesionales.

En un mundo dominado por la ciencia de datos, la ingeniería y la investigación, saber cómo construir gráficos efectivos permite descubrir patrones, comunicar hallazgos, respaldar argumentos y tomar decisiones informadas. Desde una simple gráfica de barras hasta funciones matemáticas complejas, Python ofrece la flexibilidad y el control para que cada gráfica cumpla un propósito concreto, siguiendo estándares internacionales de publicación.

Dominar esta herramienta es dar un paso hacia adelante en el pensamiento analítico, en la comunicación de resultados, y en la práctica profesional del estudiante. Porque, al final, una gráfica bien diseñada vale más que mil líneas de datos.

De los fundamentos a la frontera de la programación

Este libro ha sido más que un compendio de técnicas, estructuras y sintaxis; ha sido un recorrido formativo por los pilares del pensamiento algorítmico, la lógica computacional y el poder expresivo de Python como lenguaje de programación.

Iniciamos con los fundamentos: comprendiendo qué es programar, cómo se comunica una máquina con instrucciones humanas, y cómo se estructura un algoritmo de forma ordenada y precisa. Aprendimos a escribir código desde cero, entendiendo las estructuras de control, las funciones, la recursividad, y cómo dividir problemas complejos en soluciones más pequeñas, tal como lo sugiere la estrategia de "divide y vencerás".

Exploramos paradigmas de programación: imperativa, estructurada, modular, orientada a objetos, multihilos y multiparadigma. Cada uno aportó una nueva forma de pensar, de modelar la realidad y de resolver problemas de forma eficiente y escalable. Con analogías claras y comparaciones didácticas, comprendimos que la programación no es solo escribir código, sino estructurar ideas y procesos con claridad y propósito.

Nos adentramos en algoritmos de búsqueda y ordenamiento, aplicando lógica pura y refinada. Luego, cruzamos la frontera hacia la aplicación real: manejo de archivos, expresiones regulares, APIs, JSON, hojas de cálculo, análisis de datos y graficación. Herramientas que hoy son parte esencial del entorno profesional en sectores como la ingeniería, la ciencia, la economía, la inteligencia artificial, la ciberseguridad y más.

Cada biblioteca que exploramos —os, re, json, openpyxl, matplotlib, sympy, entre otras— nos abrió una ventana a problemas reales y soluciones concretas. Aprendimos que Python no es solo un lenguaje para principiantes, sino una poderosa herramienta utilizada por gigantes de la industria como Google, NASA, Netflix o Spotify, y por investigadores, analistas y desarrolladores de todo el mundo.

Este camino que iniciaste te ha transformado. Ahora puedes ver un problema y pensar cómo traducirlo a código; puedes automatizar procesos, analizar información, visualizar datos, y construir soluciones inteligentes. Has pasado de la teoría a la práctica, del concepto a la ejecución.

Pero este no es el final. Es apenas el comienzo de una trayectoria profesional llena de retos y oportunidades. El conocimiento adquirido aquí es la base sólida para especializarte en ciencia de datos, inteligencia artificial, desarrollo web, automatización industrial, análisis financiero o cualquier campo donde la programación sea clave.

Lleva contigo la lógica, la curiosidad y el deseo de seguir aprendiendo. Porque el verdadero programador no se define por el número de líneas que escribe, sino por la capacidad de pensar, abstraer, resolver y mejorar continuamente.

¡Felicidades! Ahora estás listo para construir soluciones reales para un mundo que cada vez necesita más mentes como la tuya.