

Capítulo 3

Lenguaje de Programación

Resultado de aprendizaje

Desarrolla aplicaciones utilizando el lenguaje de programación.

Indicador de logro

Identifica la sintaxis y semántica en un lenguaje de programación.
Diseña e implementa programas en un lenguaje de programación.

Un lenguaje de Programación es un conjunto de símbolos y códigos usados para orientar el flujo de una solución con base en las estructuras de control propias, durante el desarrollo o implementación de algoritmos. Conocer cómo funciona un lenguaje de Programación en el computador es importante para mejorar procesos reales que hagan una diferenciación competitiva en todo ámbito. En los últimos años, los lenguajes de programación han permitido mejores desarrollos (aplicaciones de escritorio, aplicativos en la web, aplicación en dispositivos), aumentado su potencia y flexibilidad para llevar a cabo tareas complejas que la innovación exige. El índice de la Comunidad de Programación TIOBE¹ es un indicador de la popularidad de los lenguajes de programación, ver Fig. 3.1.

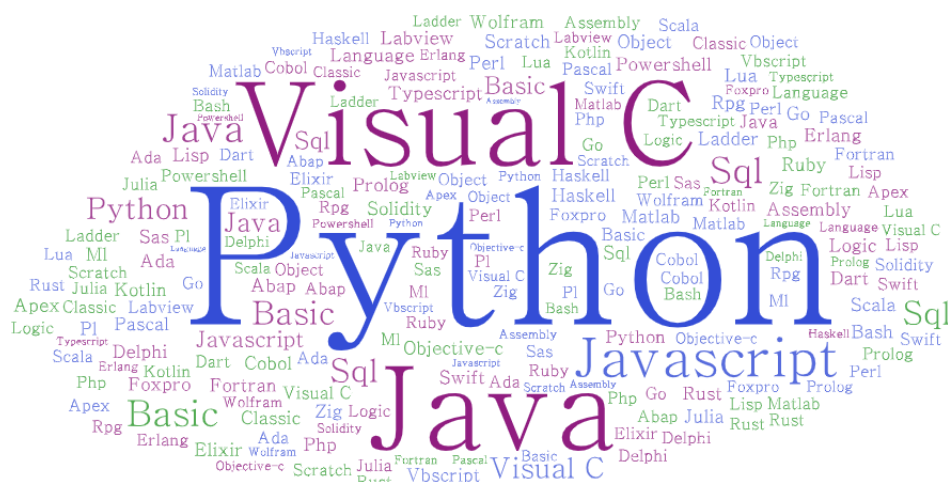


Figura 3.1: Los lenguajes de programación del índice TIOBE expresados en una nube de palabras.

¹TIOBE Quality Indicator Quality Model

3.1. Fundamentos a los lenguajes de desarrollo

Hoy en día muchos dispositivos y máquinas requieren de un lenguaje de programación para cumplir con sus funciones. Si se conoce como poder utilizarlos adecuadamente, estos permitirán alcanzar los objetivos en menos tiempo y poder alcanzar mayor productividad. Así, un lenguaje de programación, es el conjunto de instrucciones a través del cual los humanos programan e interactúan con las computadoras. Un lenguaje de programación nos permite establecer una forma de comunicarnos con las máquinas a través de instrucciones escritas en una sintaxis que representa el algoritmo y la computadora lo entiende e interpreta en lenguaje de máquina.

Los programadores utilizan un lenguaje de programación para implementar una idea a partir de un algoritmo, la pragmática. El lenguaje de programación permite al computador procesar de forma rápida y eficientemente grandes y complejas cantidades de información, la semántica. Por ejemplo, si a una persona se le da una lista de números aleatorios que van de uno a diez mil y se le pide que los coloque en orden ascendente, es probable que tome una cantidad considerable de tiempo e incluya algunos errores, mientras que si le das la misma instrucción a una computadora utilizando un lenguaje de programación, podrás obtener la respuesta en unos cuantos segundos y sin errores.

El uso de un lenguaje de programación, requiere entonces de la comprensión desde tres perspectivas. Desde la sintaxis: que es el conjunto de símbolos y reglas para formar sentencias y así su escritura. Desde la semántica: que son las reglas que permiten transformar las sentencias en instrucciones lógicas. Desde la pragmática: que son aquellas construcciones particulares del lenguaje que van estableciendo un estilo de programar. Así como en nuestro lenguaje natural, las letras forman palabras y estas forman oraciones, luego un estilo de escribir y redactar. En los lenguajes de programación, los caracteres forman sentencias que en su conjunto forman las instrucciones, luego un programador va desarrollando un pensamiento algorítmico para establecer buenas prácticas al programar, ver Fig. 3.2.

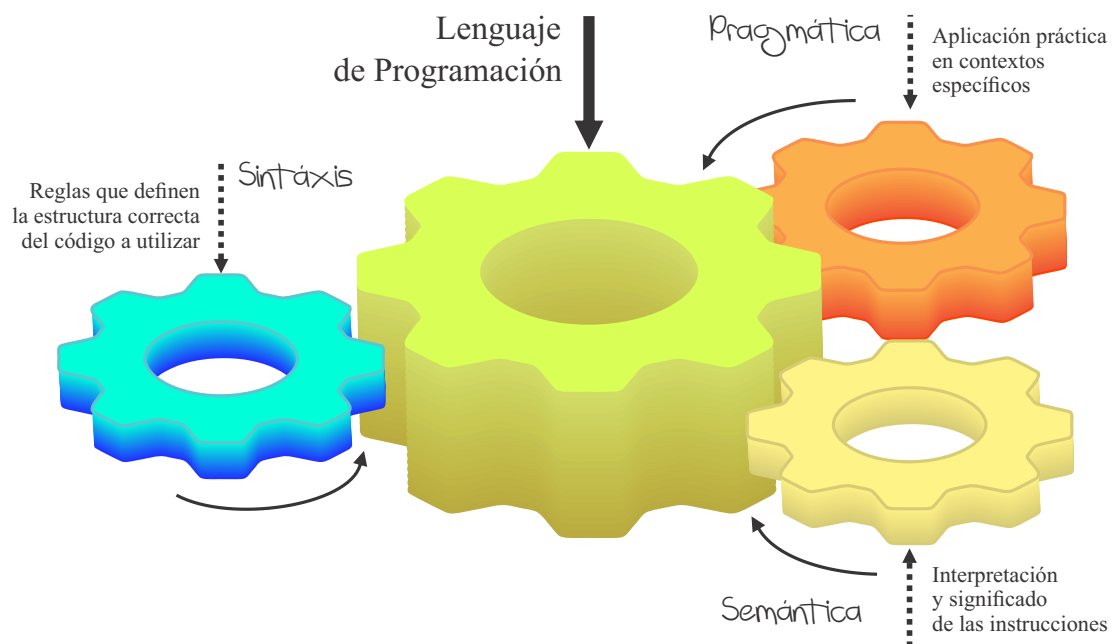


Figura 3.2: Características de un lenguaje de programación.

3.1.1. Introducción a los lenguajes de desarrollo

Los lenguajes de desarrollo, también conocidos como lenguajes de programación, son herramientas necesarias en el mundo de la computación, de los sistemas tecnológicos, de la informática en general. Estos lenguajes se basan en un conjunto de conceptos que definen la utilidad de elementos como las variables y sus estructuras (simples y compuestas), los operadores (matemáticos o aritméticos, lógicos y relacionales), las estructuras de control (secuenciales, selectivas o condicionales y, repetitivas o bucles) y funciones (subrutinas), que forman la base de cualquier programa, aplicativo o desarrollo.

Definición general

Un lenguaje de programación se define como un conjunto de instrucciones y reglas sintácticas que permiten a los humanos comunicarse con las computadoras para resolver problemas, tareas o cálculos, y automatizarlas.

Sin estos elementos característicos, sería complejo crear software correcto y eficiente. Por ejemplo, las variables compuestas permiten almacenar datos, mientras que los operadores aritméticos y lógicos realizan cálculos matemáticos, razonables y coherentes respectivamente, lo que facilita la manipulación de datos e información en tiempo real, ver Fig. 3.3.

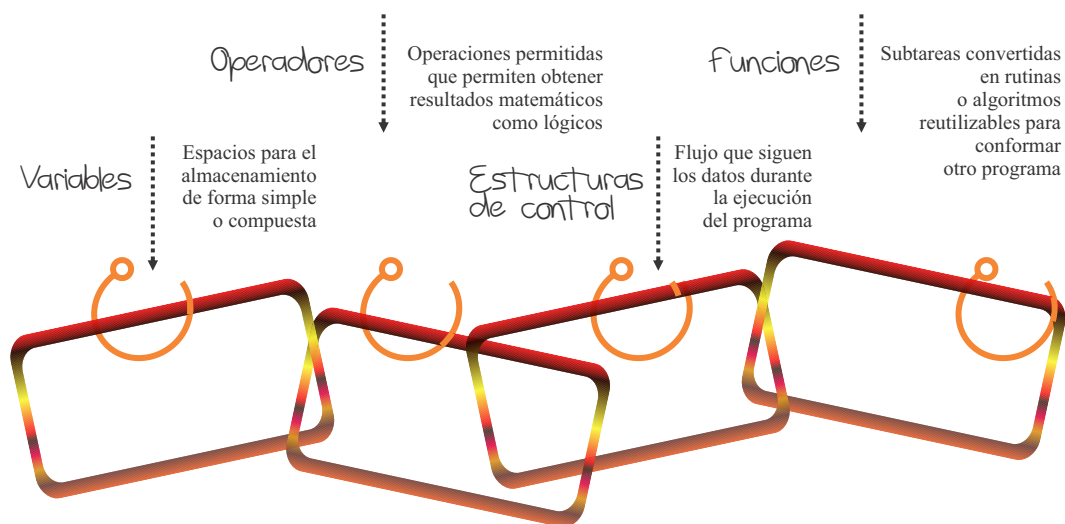


Figura 3.3: Elementos fundamentales de un lenguaje de programación.

Otro aspecto importante es la comprensión de la lógica de programación, que implica descomponer problemas complejos en otros más simples, así como realizar pasos simples y secuenciales, es decir “divide y vencerás”. Esto se relaciona con el paradigma de la programación estructurada, de Edsger Dijkstra [Dijkstra, 1974], quien resalta la importancia de evitar código desorganizado [Randell, 2003]. Además, hay que tener en cuenta conceptos como la depuración (identificación y corrección de errores) y la optimización de código, que aseguran que la solución algorítmica sea funcional y eficiente en términos de respuesta (tiempo) y elementos computacionales (recursos del sistema) [First et al., 2025]. Estos aspectos aplican a todos los lenguajes, independientemente de su complejidad o propósito, formando la base para cualquier profesional, desarrollador, principiante o experto [Jäger et al., 2025].

Es así como los lenguajes de desarrollo, surgieron en la década de 1950 con la llegada de las primeras computadoras digitales [Baker, 1972]. El lenguaje FORTRAN, desarrollado por IBM en 1957 [Van Snyder, 2007; Backus, 1978], fue uno de los pioneros, diseñado específicamente para

cálculos científicos y matemáticos. Posteriormente, lenguajes como COBOL (para aplicaciones empresariales) [Sammet, 1978] y BASIC (para educación) ampliaron su disponibilidad [Newell et al., 1957]. Hoy en día, el uso de un lenguaje de programación es indispensable en campos como el desarrollo web, la inteligencia artificial, la investigación espacial y los videojuegos [Carver et al., 2025]. Su evolución ha sido impulsada por la necesidad de mayor abstracción, permitiendo a los programadores enfocarse en la lógica en lugar de en los detalles del hardware [Von Neumann & Kurzweil, 2012].

3.1.2. Clasificación de los lenguajes de desarrollo

Los lenguajes de desarrollo se clasifican dependiendo de diferentes criterios como el nivel de abstracción, el paradigma de programación o el propósito específico. En cuanto al nivel de abstracción, se dividen en lenguajes de bajo nivel (como el ensamblador, cercanos al código máquina) y de alto nivel (como Python, C o Java, más cercanos al lenguaje humano). Los de bajo nivel ofrecen mayor control sobre el hardware, pero son más difíciles de aprender, mientras que los de alto nivel priorizan la productividad y la legibilidad, y su nivel de aprendizaje es adecuado, ver Fig. 3.4.

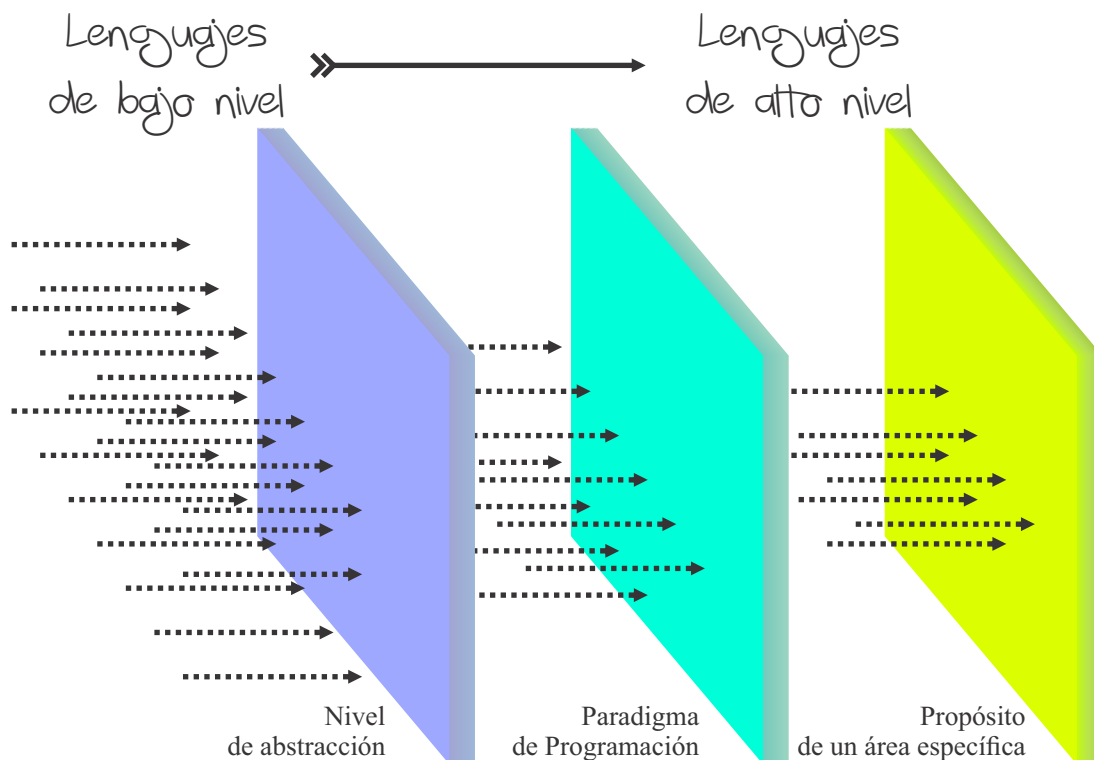


Figura 3.4: Clasificación de los lenguajes de programación.

Cuando se habla de la clasificación de los lenguajes de programación por paradigmas, se hace referencia a los lenguajes con características procedimentales (como el lenguaje C, enfocados en procedimientos y funciones), los lenguajes orientados a objetos - POO - (como Java, que usan clases y objetos para modelar el mundo real), funcionales (como Haskell, basados en funciones matemáticas puras) y declarativos (como SQL - *Structured Query Language* - o lenguaje de consulta estructurado, donde se describe el resultado deseado en lugar de los pasos).

Además, existen lenguajes multiparadigma, como Python, que combinan varios enfoques. Gracias a esta diversidad es necesario saber elegir el lenguaje adecuado según el proyecto a desarrollar, desde aplicaciones móviles hasta sistemas embebidos.

Los paradigmas de programación son enfoques o estilos fundamentales para estructurar y resolver problemas en el código. Cada uno enfatiza diferentes conceptos, como procedimientos, objetos, funciones puras o reglas lógicas. A continuación, se presentan los principales paradigmas con ejemplos de lenguajes representativos. Muchos lenguajes modernos son multiparadigma, lo que significa que soportan varios enfoques simultáneamente.

El paradigma *Imperativo o Procedimental*, se centra en describir cómo realizar tareas mediante secuencias de comandos que cambian el estado del programa. Ejemplos clásicos incluyen C, Pascal, Fortran y BASIC. Estos lenguajes usan procedimientos o funciones para organizar el código.

El paradigma *Orientado a Objetos - OOP*, modela el mundo real mediante objetos que encapsulan datos y comportamientos. Los ejemplos más destacados son Java, C++, Python, C# y Ruby. Conceptos clave como herencia, polimorfismo y encapsulamiento son centrales aquí.

El paradigma *Funcional* trata la computación como evaluación de funciones matemáticas, evitando estados mutables y efectos secundarios para favorecer su naturaleza. Ejemplos puros incluyen Haskell y Lisp (en su variante Scheme). Otros con soporte fuerte son Scala, Erlang, F# y Clojure. Lenguajes como Python y JavaScript también incorporan elementos funcionales.

El paradigma *Declarativo* se enfoca en describir qué se quiere lograr, no cómo hacerlo paso a paso. Incluye subparadigmas como el lógico (basado en hechos y reglas) con ejemplos como Prolog; y el de bases de datos con SQL. Otros declarativos son HTML (HyperText Markup Language) o CSS (Cascading Style Sheets), para estructurar y estilar web, o lenguajes como Mercury.

Otros paradigmas, como el paradigma *concurrente o paralelo* destaca en lenguajes como Go (goroutines) o Rust (propiedad y borrowing para seguridad). Asimismo, el paradigma *basado en eventos* es común en JavaScript (para web asíncrona). Lenguajes multiparadigma como Python, JavaScript, Scala o Kotlin combinan varios de estos enfoques, ofreciendo flexibilidad para diferentes tipos de proyectos, ver Fig. 3.5.

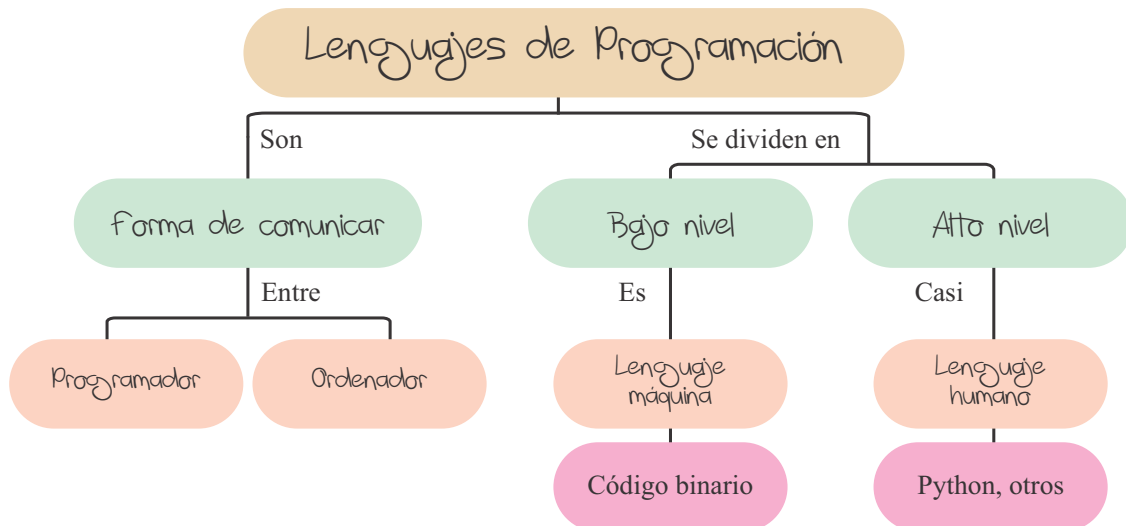


Figura 3.5: Clasificación de los lenguajes de programación.

3.2. Introducción a Python

Con el paso del tiempo, se han visto en crecimiento un gran número de lenguajes de desarrollo y con ello una secuencia de herramientas para los desarrolladores. Pocos lenguajes han logrado alcanzar dos ámbitos a nivel mundial, ser una herramienta de estudio para estudiantes y educadores y, ser el motor principal del desarrollo tecnológico para las corporaciones.

Python, creado a finales de la década de 1980 por el programador holandés Guido van Rossum, representa una singularidad particular, un lenguaje que prioriza la elegancia y la simplicidad humana por encima de la eficiencia ordinaria de la máquina.

El inicio de Python se sitúa en las vacaciones de Navidad de 1989 (según registros en Amazon²). Lo que comenzó como un proyecto personal de “pasatiempo” para dar continuidad al lenguaje ABC en el Centrum Wiskunde & Informatica (CWI), terminó por redefinir los estándares de la industria. Desde su publicación oficial en 1991, Python introdujo conceptos que hoy damos por hecho, como el manejo de excepciones y las clases con herencia, pero lo hizo con una sintaxis tan limpia que parecía pseudocódigo en inglés.

El crecimiento del lenguaje no fue repentino, sino organizado. Su primera gran transformación ocurrió con Python 2.0 en el año 2000, cuando se introdujeron características como la recolección de basura y el soporte para Unicode. Sin embargo, el verdadero punto de cambio de dirección fue su filosofía de código abierto. Al permitir que una comunidad global contribuyera a sus bibliotecas, Python dejó de ser una herramienta de una comunidad especializada para sistemas operativos y se convirtió en un ecosistema modular.

Uno de los acontecimientos más críticos en su historia fue el lanzamiento de Python 3.0 en 2008. Esta versión “limpió” el lenguaje de redundancias, pero no era compatible con las versiones anteriores (Python 2.0). Durante una década, la comunidad estuvo dividida, pero la apuesta por la modernización resultó ser la correcta. Hoy, gracias a esa evolución, Python es la columna vertebral de la revolución de la Inteligencia Artificial. Bibliotecas como NumPy, Pandas y TensorFlow han encontrado en Python el “adhesivo” perfecto para unir algoritmos complejos con interfaces fáciles de usar.

3.2.1. Tipos de datos y variables

En el centro de todo lenguaje de programación reside la capacidad de representar, tratar y procesar información. En Python, esta capacidad es una extensión de su filosofía de diseño. A través de un sistema de variables flexibles y una jerarquía de tipos de datos rica y coherente, Python permite a los desarrolladores pasar de la abstracción de una idea a la concreción del código con una fricción mínima.

En Python, las variables son nombres que se vinculan a objetos mediante el operador de asignación igual a (=). No es necesario declarar el tipo de una variable de antemano, ya que Python es un lenguaje de tipado dinámico (el tipo se determina en tiempo de ejecución según el objeto asignado). Por ejemplo, al escribir `x = 5`, la variable `x` se refiere a un objeto de tipo entero (declarativamente como `int`), y si luego asignamos `x = “hola”`, ahora se refiere a una cadena de caracteres (declarativamente como `str`). Esta flexibilidad permite cambiar el tipo de una variable en cualquier momento. Las asignaciones múltiples, como `a, b = 0, 1`, son comunes y evalúan el lado derecho primero antes de vincular las variables.

Los tipos de datos básicos en Python incluyen números y texto. Entre los numéricos destacan los enteros (`int`), que representan números sin decimales y tienen precisión arbitraria (por ejemplo, `50` o `2**100` o `2100`, para potencias grandes); los flotantes (declarativamente como `float`), para

²[Historia de Python en Amazon](#)

números con decimales (como 5.0 o 1.6); y operaciones como la división (/) que siempre devuelve un número real (float), o la división entera (//).

Python maneja automáticamente conversiones en operaciones mixtas, convirtiendo enteros a flotantes cuando es necesario. Las cadenas de texto (str) son secuencias inmutables de caracteres, encerradas entre comillas simples o dobles, y soportan concatenación con +, repetición con *, indexación y rebanado (slicing). Otro aspecto fundamental es la inmutabilidad de ciertos tipos. Las cadenas, números y tuplas son inmutables: una vez creados, no se pueden modificar directamente. Intentar cambiar un elemento de una cadena, como `s[0] = "x"`, genera un error.

En cambio, operaciones como la concatenación crean un nuevo objeto. Esto contrasta con tipos mutables, donde el objeto puede cambiar in situ. Entre los tipos de datos estructurados más importantes se encuentran las listas (list), que son secuencias mutables y ordenadas de elementos heterogéneos. Se definen con corchetes [`1, "dos", 3.0`] y permiten métodos como `append()`, `extend()`, `pop()` o `sort()` para modificar su contenido. Las listas son ideales para colecciones que cambian durante la ejecución, como pilas o colas.

Las tuplas (tuple) son similares a las listas, pero inmutables, definidas con paréntesis (`1, 2, 3`). Se usan para datos fijos o como claves en diccionarios. Los conjuntos (set), creados con llaves `1, 2, 3` o la función `set()`, almacenan elementos únicos sin orden, y son útiles para operaciones matemáticas como unión o intersección. Finalmente, los diccionarios (dict) son colecciones mutables de pares clave-valor, con llaves nombre: "Ana", edad: 25, donde las claves deben ser inmutables.

Python trata todo como objetos, lo que implica que las variables son referencias a estos objetos. Al asignar una variable a otra se crea un alias (referencias compartidas), por lo que modificar un objeto mutable afecta a todas las referencias. Esta característica, combinada con la diversidad de tipos built-in, hace que Python sea potente y expresivo para manejar datos de manera eficiente.

3.2.2. Operadores

Los operadores en Python actúan como un puente entre la información almacenada y los resultados deseados. Son los componentes o símbolos que permiten que el lenguaje deje de ser una simple lista de valores para convertirse en un sistema lógico capaz de tomar decisiones, realizar cálculos científicos y manipular estructuras de datos masivas.

Los operadores en Python son símbolos especiales que realizan operaciones aritméticas y lógicas, sobre variables y valores, permitiendo manipular datos de manera eficiente. Python ofrece una amplia variedad de operadores clasificados en categorías como aritméticos, de comparación, lógicos, de asignación, bitwise, de identidad y de membresía. Estos operadores son fundamentales para escribir expresiones que forman la base de cualquier programa, desde cálculos simples hasta lógica compleja. Su uso correcto mejora la legibilidad y el rendimiento del código, y Python prioriza la claridad con una precedencia definida (por ejemplo, los aritméticos tienen mayor prioridad que los lógicos).

Los operadores aritméticos son los más básicos y se utilizan para realizar operaciones matemáticas. Incluyen suma (+), resta (-), multiplicación (*), división (/ que siempre devuelve un real, float), división entera (//), módulo (% para el resto) y potenciación (**). Por ejemplo, `5 ** 2`, `25`, resulta en 25, y `10 % 3` da 1. Estos operadores funcionan con números (int y float) y, en algunos casos, con otros tipos como cadenas (para concatenación con + o repetición con *).

Los operadores de comparación evalúan relaciones entre valores y devuelven un booleano (True o False). Son: igualdad (==), desigualdad (!=), mayor (>), menor (<), mayor o igual (>=) y menor o igual (<=). Se usan comúnmente en estructuras de control como (si) en inglés if. Python permite encadenar comparaciones, como `a < b < c`, lo que equivale a `a < b` and `b < c`, facilitando expresiones más naturales y legibles.

Los operadores lógicos combinan expresiones booleanas: and (verdadero si ambos son verda-

deros), or (verdadero si al menos uno es verdadero) y not (invierte el valor). Estos son esenciales para decisiones complejas. Además, los operadores bitwise operan a nivel de bits: `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (desplazamiento izquierdo) y `>>` (desplazamiento derecho). Son útiles en optimizaciones de bajo nivel, como manipulación de flags o compresión de datos.

Otros operadores importantes incluyen los de asignación, como `=` para asignar valores, y versiones aumentadas como `+=`, `-=`, `*=`, que combinan operación y asignación (por ejemplo, `x += 5` es equivalente a `x = x + 5`). Los operadores de identidad (`is` y `is not`) verifican si dos variables referencian el mismo objeto en memoria, diferente de `==` que compara valores. Finalmente, los de membresía (`in` y `not in`) comprueban si un elemento está en una secuencia como listas o diccionarios. La combinación de estos operadores, junto con la precedencia y asociatividad definida en Python, permite crear expresiones fuertes y concisas. Dominar el uso de estas expresiones es importante para avanzar en la programación, desde *scripts* (códigos precompiladores o funcionales) simples hasta aplicaciones avanzadas en datos o *machine learning*.

3.2.3. Estructuras secuenciales

La estructura secuencial se basa en el principio de “paso a paso”. En Python, esto se traduce en la ejecución de líneas de código que el intérprete lee de manera descendente. Aunque parezca un concepto trivial, es donde se establecen las reglas de dependencia de un programa. Para procesar una variable, primero debe ser declarada; para mostrar un resultado, primero debe ser calculado. Esta dependencia jerárquica es lo que permite que una computadora, una máquina de estados finitos, transforme electricidad en información útil.

El diseño de una secuencia eficaz requiere claridad de pensamiento. El programador debe actuar como un coreógrafo que decide qué acción precede a la otra para evitar errores de referencia o lógica. Por ejemplo, en Python, la interacción con el usuario a través de `input()` y la entrega de resultados mediante `print()` son los “sujetalibros” de una secuencia. Entre ellos ocurre la magia del procesamiento, donde los operadores y las variables que hemos analizado previamente se combinan para transformar la realidad cruda de los datos en conocimiento estructurado, ver Fig. 3.6.

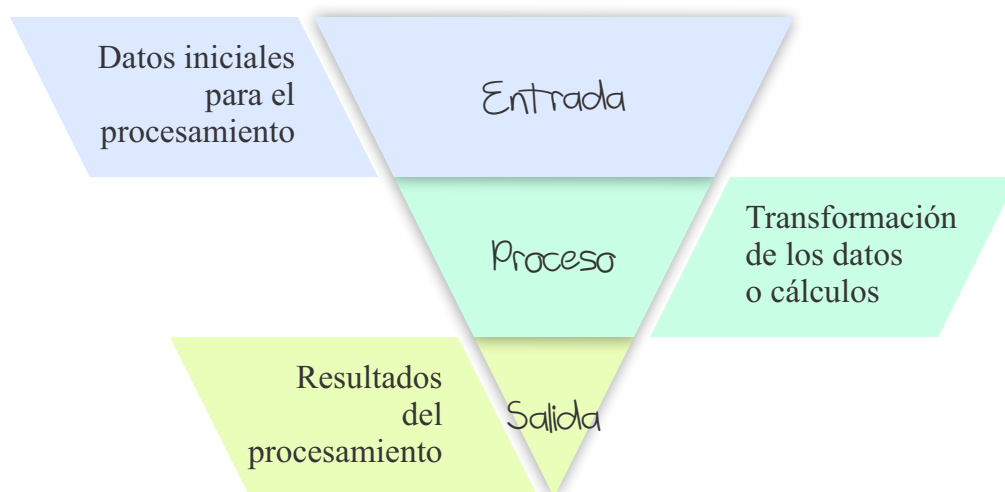


Figura 3.6: Programación secuencial.

Además, la linealidad proporciona una ventaja crítica: la facilidad de depuración (debugging). Al no haber saltos lógicos ni ramificaciones, el flujo de datos es transparente. Esta simplicidad es la que permite a los principiantes construir sus primeras herramientas —como convertidores de

moneda o calculadoras de área— y a los expertos construir los cimientos de sistemas críticos que requieren una ejecución predecible y segura.

Las estructuras secuenciales representan el fundamento más básico del control de flujo en la programación. Consisten en la ejecución lineal y ordenada de instrucciones, una tras otra, sin desviaciones ni repeticiones condicionales. En este paradigma, el programa sigue un camino único desde el inicio hasta el fin, procesando cada sentencia en el orden en que aparece en el código. Esta estructura es ideal para tareas simples como cálculos directos, entrada y salida de datos, o procesos que no requieren decisiones o bucles. En términos conceptuales, la estructura secuencial se basa en tres pasos principales: entrada de datos (obtener información del usuario o fuentes externas), procesamiento (realizar operaciones con esos datos) y salida (mostrar resultados). Por ejemplo, un programa que calcula el área de un rectángulo recibe la base y la altura, multiplica ambos valores y muestra el resultado. Esta simplicidad facilita la comprensión y depuración del código, especialmente para principiantes. En Python, las estructuras secuenciales se implementan de forma natural, ya que el intérprete ejecuta las líneas de código de arriba hacia abajo.

Algoritmo 3.1: Ejemplo básico de Entrada y Salida en Python

```

1 # 0. Comentar, autor: Joe Llerena, fecha: 30-12-2025
2 # 1. Solicitar el nombre (input siempre devuelve una cadena de texto)
3 nombre = input("Ingrese su nombre: ")
4 # 2. Solicitar la edad y convertirla a un valor entero (int)
5 edad = int(input("Ingrese su edad: "))
6 # 3. Mostrar el saludo usando una "f-string"
7 print(f"Hola {nombre}, tienes {edad} años.")
8 # 3. Mostrar otra alternativa
9 print("Hola", nombre, "tienes", edad, "años.")
10 # 4. Mensaje final de bienvenida
11 print("Bienvenido al programa.")

```

El resultado alcanzado, del programa anterior, se muestra en la imagen siguiente donde se presentan las líneas de código, ver Fig. 3.7.

```

Ingrese su nombre: Joe
Ingrese su edad: 12
Hola Joe, tienes 12 años.
Hola Joe tienes 12 años.

```

Figura 3.7: Programación secuencial.

3.2.4. Estructuras de control condicionales

Las estructuras condicionales se representan visualmente mediante diagramas de flujo, donde un rombo simboliza la decisión (condición), con ramas “Sí” y “No” que llevan a procesos rectangulares. Esto facilita el diseño algorítmico antes de codificar.

En Python, la estructura condicional se rige por el principio de la elegancia. El uso de la palabra clave *if* seguida de una expresión lógica es casi idéntico al razonamiento humano: “Si ocurre esto, haz aquello”. Sin embargo, la sofisticación de Python brilla en su manejo de la ambigüedad mediante el *elif* y el *else*. Estas herramientas permiten al programador cubrir todo el espectro de posibilidades, asegurando que el programa nunca se encuentre en un estado de incertidumbre.

Un aspecto revolucionario en la sintaxis de Python es la obligatoriedad de la indentación. En otros lenguajes, un programador podría escribir condicionales desordenados que funcionan, pero son imposibles de leer. En Python, la estructura visual del código refleja directamente su estructura

lógica. Esto reduce drásticamente los errores de “lógica anidada”, donde un desarrollador podría perderse en múltiples niveles de decisiones.

Además, las condicionales modernas en Python han evolucionado para manejar la complejidad con mayor eficacia y simplicidad. Con la introducción de características como las expresiones condicionales (el operador ternario) y, más recientemente, el *Structural Pattern Matching* (match-case), el lenguaje demuestra que la toma de decisiones puede ser tan potente como una base de datos y tan clara como un esquema. Estas estructuras no solo controlan el flujo; definen la robustez del sistema, permitiendo que el software maneje excepciones y casos de borde que, en una estructura secuencial pura, causarían el colapso del programa, ver Fig. 3.8.

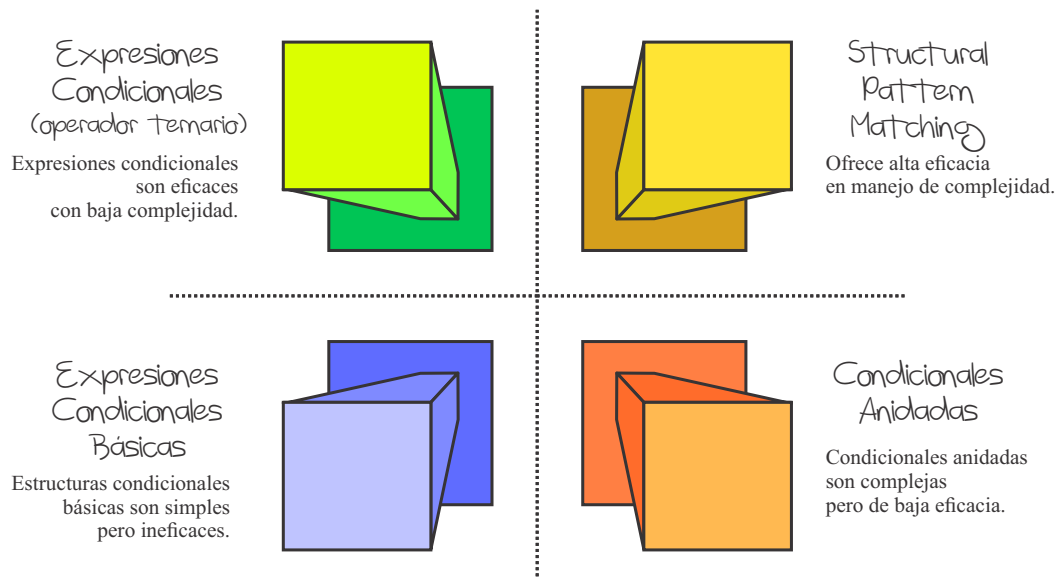


Figura 3.8: Estructuras de control.

3.2.5. Estructuras de control repetitivas

Las estructuras de control repetitivas, también conocidas como bucles o ciclos, permiten ejecutar un bloque de código múltiples veces mientras se cumpla una condición o para un número determinado de iteraciones. Estas estructuras son el tercer pilar fundamental del control de flujo, junto con las secuenciales y condicionales, y son indispensables para automatizar tareas repetitivas, procesar colecciones de datos o implementar algoritmos eficientes. Sin ellas, tareas como sumar elementos de una lista o validar entradas hasta que sean correctas requerirían código redundante y poco mantenible.

Existen tres tipos principales de bucles: el bucle *while* (*mientras*), que evalúa la condición antes de ejecutar el bloque y repite mientras sea verdadera; el bucle *for* (*para*), ideal para iterar sobre secuencias conocidas como rangos o listas; y el bucle *do-while* (*hacer-mientras*), que ejecuta el bloque al menos una vez y luego verifica la condición (aunque no está nativo en Python, se puede emular con un *while*).

La iteración en Python destaca por su elegancia sintáctica y su versatilidad. El bucle *while* representa la persistencia; es la herramienta del control continuo, fundamental en sistemas que deben mantenerse activos hasta que ocurre un evento específico, como un servidor esperando peticiones o un videojuego procesando la física en cada fotograma. Su riesgo es el “bucle infinito”, un error lógico donde la condición nunca se vuelve falsa, pero bajo un diseño cuidadoso, es la base de la estabilidad de muchos sistemas operativos.

Por otro lado, el bucle *for* en Python redefine la manera de interactuar con los datos. Al adoptar un enfoque basado en iteradores, Python permite al programador decir: “Para cada elemento en esta colección, haz esto”. Esta abstracción elimina la necesidad de gestionar manualmente contadores o índices complejos (comunes en lenguajes como C o Java), reduciendo así la probabilidad de errores de tipo “fuera de rango”. La combinación de *for* con la función `range()` o con estructuras como listas, convierte a Python en el lenguaje predilecto para la ciencia de datos, donde el procesamiento de filas y columnas es constante.

Finalmente, la capacidad de estas estructuras para anidarse —un bucle dentro de otro— permite resolver problemas multidimensionales, como procesar imágenes (píxel por píxel) o matrices matemáticas. Aunque la potencia es inmensa, la filosofía de Python nos recuerda que “lo simple es mejor que lo complejo”, instando a los desarrolladores a utilizar estas herramientas de forma eficiente para evitar el desperdicio de recursos computacionales.

3.2.6. Funciones

Las funciones son uno de los conceptos más importantes en la programación, ya que permiten modularizar el código, reutilizarlo y organizar programas complejos en bloques manejables. Una función es un bloque de código reutilizable que realiza una tarea específica, recibe entradas (parámetros), procesa datos y opcionalmente devuelve un resultado.

Este enfoque sigue el principio *DRY* (*Don't Repeat Yourself*), reduciendo la duplicación y facilitando el mantenimiento. Además, las funciones mejoran la legibilidad, permiten depuración independiente y promueven el diseño estructurado o modular. En Python, las funciones se definen con la palabra clave `def`, seguida del nombre, paréntesis con parámetros opcionales y dos puntos. El bloque de código se indenta, y se usa `return` para devolver valores.

Definir una función en Python mediante la palabra clave `def` es un acto de creación de herramientas personalizadas. Una función bien diseñada actúa como una “caja negra”: recibe entradas, opera sobre ellas y devuelve un resultado. Esta capacidad de aislamiento es fundamental; permite que equipos de desarrolladores trabajen en diferentes partes de un sistema sin interferir entre sí. Al segmentar un problema complejo en pequeñas funciones manejables, la tarea de programar se vuelve una labor de ensamblaje de piezas lógicas.

La elegancia de Python se manifiesta especialmente en cómo trata a las funciones: son “objetos de primera clase”. Esto significa que pueden ser pasadas como argumentos a otras funciones, asignadas a variables o devueltas como resultados. Esta característica permite paradigmas avanzados como la programación funcional, facilitando la creación de código más limpio y expresivo. Además, el uso de parámetros con valores predeterminados permite que las funciones sean flexibles, adaptándose a casos simples y complejos sin necesidad de duplicar el código.

Otro aspecto vital es la documentación intrínseca. A través de los docstrings (cadenas de texto descriptivas), Python fomenta que la función explique su propósito por sí misma. Esto, sumado a la legibilidad natural del lenguaje, convierte a las funciones en la mejor forma de comunicación entre programadores. Un código compuesto por funciones bien nombradas se lee como un manual de instrucciones, claro, reduciendo el tiempo de mantenimiento y facilitando la detección de errores.

3.2.7. Arreglos o vectores

El concepto de arreglo nace de la necesidad de orden y velocidad. En un arreglo tradicional, los datos se almacenan “hombro con hombro” en la memoria del sistema. Esta disposición no es trivial; permite que la CPU prediga y acceda a los datos con una latencia mínima. En Python, aunque las listas ofrecen una flexibilidad asombrosa, es a través de herramientas como NumPy donde el concepto de vector alcanza su máximo potencial. Un vector no es solo una lista de números;

es una entidad matemática sobre la que se pueden aplicar transformaciones complejas en un solo paso.

La importancia de los vectores reside en la “vectorización”. En un programa convencional, para sumar dos listas de mil números, un programador usaría un bucle que recorre cada posición, una por una. Con vectores, la instrucción se envía al procesador como una única operación masiva. Esta distinción es la que permite que aplicaciones modernas, desde el reconocimiento de rostros hasta las predicciones climáticas, funcionen en tiempo real. Los arreglos transforman el procesamiento de datos de una tarea secuencial agotadora en un flujo paralelo de alta velocidad.

Además, los arreglos multidimensionales (matrices) expanden esta lógica a planos y volúmenes, permitiendo representar desde imágenes digitales (donde cada píxel es un elemento de un arreglo) hasta modelos físicos del universo. La rigidez de los arreglos —el hecho de que normalmente todos sus elementos deben ser del mismo tipo— es, paradójicamente, su mayor fortaleza, ya que elimina la ambigüedad y permite una optimización que las estructuras de datos más flexibles no pueden alcanzar.

Los arreglos (arrays) son estructuras de datos fundamentales que almacenan una colección fija de elementos del mismo tipo en posiciones contiguas de memoria. Esto permite acceso directo a cualquier elemento mediante su índice, con una complejidad de $O(1)$. Los arreglos son estáticos: su tamaño se define al crearse y no puede cambiarse fácilmente. Se clasifican en unidimensionales (como vectores), bidimensionales (matrices) y multidimensionales. Su ventaja principal es la eficiencia en acceso aleatorio, pero insertar o eliminar elementos en medio requiere desplazar otros, lo que es costoso ($O(n)$). En muchos lenguajes como C o Java, los arreglos son de tamaño fijo. Por ejemplo, en Python las listas (*list*) actúan como arreglos dinámicos: pueden crecer o reducirse, pero internamente usan un mecanismo de sobre asignación (over-allocation) para amortizar el costo de redimensionamiento. Los vectores, en contextos de estructuras de datos, suelen referirse a arreglos unidimensionales o a implementaciones dinámicas como `std::vector` en C++, que crecen automáticamente duplicando su capacidad cuando se llenan. En matemáticas y física, un vector es una entidad con magnitud y dirección, representada frecuentemente como un arreglo de componentes, ver Fig. 3.9.

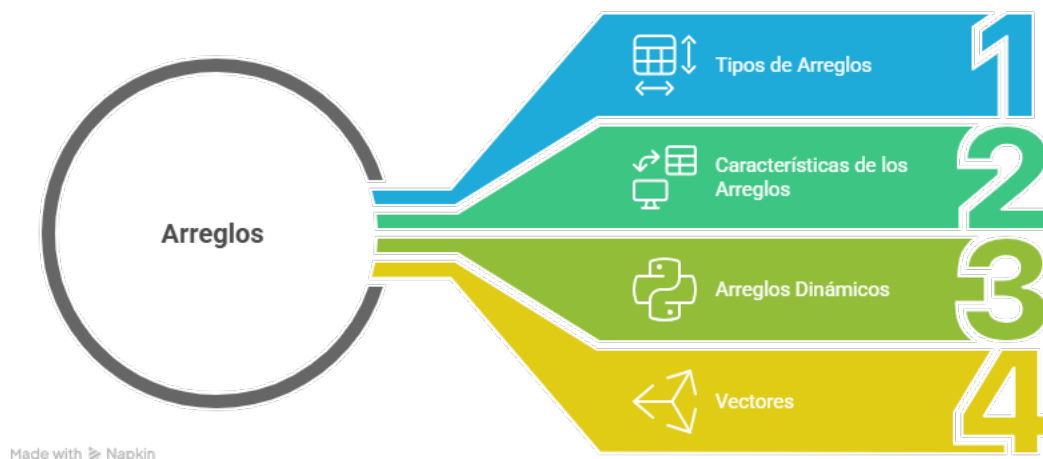


Figura 3.9: Imagen creada por la inteligencia artificial Napkin.ai, en la imagen se muestran los elementos importantes de los arreglos.

En programación científica, bibliotecas como NumPy en Python usan arreglos (ndarray) optimizados para operaciones vectorizadas, evitando bucles explícitos y aprovechando código compilado para mayor velocidad. En Python, las listas son heterogéneas y flexibles, pero menos eficientes para cálculos numéricos. NumPy introduce arreglos homogéneos (todos del mismo tipo),

con soporte para broadcasting y funciones universales (ufuncs), lo que los hace ideales para datos científicos, machine learning y procesamiento de imágenes.

3.2.8. Matrices

Una matriz es, en esencia, una colección de vectores organizados armónicamente. En Python, esta organización permite pasar del pensamiento lineal al pensamiento espacial. La belleza de trabajar con matrices radica en la capacidad de manipular grandes conjuntos de datos como una sola entidad. Por ejemplo, al aplicar un filtro a una fotografía, no estamos modificando cada píxel individualmente mediante instrucciones aisladas; estamos aplicando una operación matemática a una matriz completa. Esta capacidad de síntesis es lo que permite que el software moderno procese video en alta definición o simule fluidos en un entorno virtual. En el ámbito del aprendizaje automático (*Machine Learning*), las matrices son el lenguaje de las neuronas artificiales, ver Fig. 3.10.

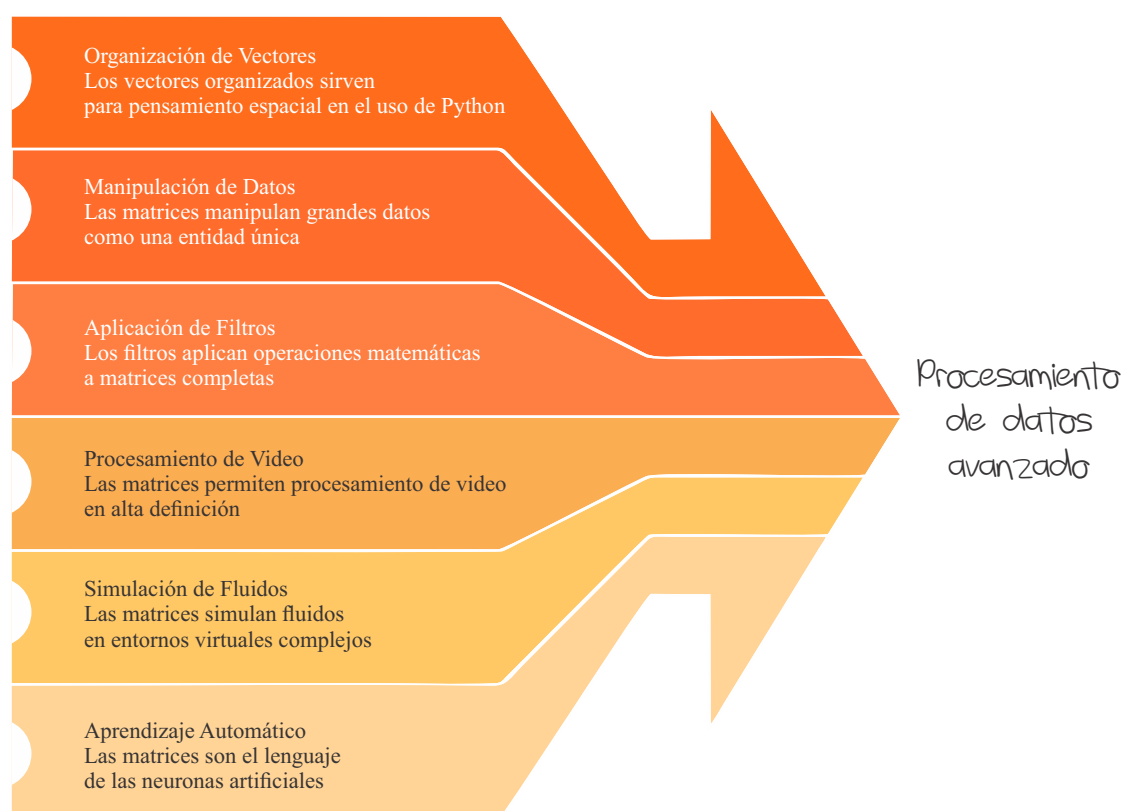


Figura 3.10: Matrices en Python.

Los pesos de una red neuronal se almacenan y procesan en forma de matrices, y el aprendizaje no es otra cosa que el ajuste constante de los valores dentro de esas grillas. Python, gracias a su sintaxis limpia, permite que estas operaciones —que matemáticamente pueden ser abrumadoras— se expresen de forma casi poética, permitiendo que el desarrollador se concentre en la arquitectura del modelo y no en la aritmética subyacente. Sin embargo, el manejo de matrices también impone un desafío de eficiencia. Debido a que el tamaño de los datos crece de forma cuadrática (una matriz de 1000×1000 ya contiene un millón de elementos), el programador de Python debe aprender a utilizar la memoria de forma inteligente. Aquí es donde la historia de Python y su evolución se unen: el lenguaje ha desarrollado interfaces que permiten que estas estructuras residan en la memoria de manera optimizada, permitiendo cálculos que serían imposibles de gestionar manualmente.

Las matrices son estructuras de datos bidimensionales que extienden el concepto de arreglos unidimensionales (vectores) para organizar elementos en filas y columnas, formando una rejilla rectangular. Cada elemento se accede mediante dos índices: uno para la fila y otro para la columna (por ejemplo, `matriz[i][j]`). Las matrices son homogéneas (generalmente del mismo tipo de dato) y de tamaño fijo en muchos lenguajes, aunque en Python se pueden implementar de forma flexible. Son fundamentales en problemas que involucran datos tabulares, como gráficos, imágenes digitales, ecuaciones lineales o algoritmos de procesamiento. En términos de representación en memoria, las matrices se almacenan de forma contigua, usualmente en orden row-major (fila por fila, común en C y Python) o column-major (columna por columna, como en Fortran). Esto afecta la eficiencia de acceso: recorrer por filas es más rápido en row-major debido a la localidad de caché. Por ejemplo, una matriz 3x3 ocupa 9 posiciones consecutivas en memoria, permitiendo cálculos rápidos de direcciones.

3.2.9. Gráficas en Python

Las gráficas o visualizaciones de datos en Python son herramientas esenciales para explorar, analizar y comunicar información de manera efectiva. Convertir datos numéricos en representaciones visuales como líneas, barras o dispersión facilita la identificación de patrones, tendencias y anomalías que serían difíciles de detectar en tablas crudas. Python destaca en este ámbito gracias a su ecosistema rico en bibliotecas, desde las básicas hasta las avanzadas para visualizaciones interactivas y estadísticas. La biblioteca más fundamental es Matplotlib, que sirve como base para muchas otras. Es altamente personalizable y permite crear gráficos estáticos de calidad pública.

La potencia de Python en la visualización radica en su versatilidad. El proceso comienza con la librería Matplotlib, que estableció el estándar de “lo que puedes imaginar, lo puedes programar”. Sin embargo, la evolución hacia librerías como Seaborn refleja un cambio de paradigma: ya no solo buscamos representar datos, sino entender sus correlaciones. Una gráfica de dispersión (*scatter plot*) puede revelar en segundos una relación entre dos variables que a una computadora le tomaría miles de ciclos procesar lógicamente.

Además, la visualización en Python ha trascendido el papel estático. Con la llegada de herramientas interactivas, las gráficas se han convertido en lienzos dinámicos. El usuario ya no es un espectador pasivo; puede hacer zoom, filtrar categorías y explorar dimensiones temporales en tiempo real. Esta interactividad es lo que permite que el “análisis exploratorio de datos” (EDA) sea una conversación entre el humano y la máquina, donde la gráfica responde preguntas a medida que surgen.

Por último, el aspecto estético en Python tiene una función pragmática. El uso correcto del color (mapas de calor o *heatmaps*) y la jerarquía visual no son meros adornos; son herramientas para evitar el sesgo y la mala interpretación. Una gráfica bien construida en Python respeta la “integridad de los datos”, asegurando que la escala y la proporción cuenten la verdad matemática sin distorsiones, facilitando la toma de decisiones basada en evidencia.



Laboratorio 7

Capítulo 3

3.3. Ejercicios del Laboratorio 7

Efectuar la solución del problema usando el lenguaje Python y utilice como variables sus nombres y apellidos, ejemplo si usted se llama Joe Frand Llerena Izquierdo, use esa información en las variables como “area”, “perimetro” o “suma” por “Joe”, “Frand” o “Llerena”.

* Ejercicio 1

Escriba un programa que muestre la siguiente figura:

```

\ | /
 @ @
  *
 \"/ /

```

Algoritmo 3.2: Programa para mostrar figura usando nombres como variables

```

1 # Definicion de la figura usando nombres y apellidos como variables
2 # Se usa doble backslash (\\) para que Python
3 # lo interprete como un caracter literal
4 Joe = " \\ | /"
5 Frand = " @ @ "
6 Llerena = " * "
7 Izquierdo = " \"/ /"
8
9 # Imprimir contenido de las variables para formar la figura
10 print (Joe)
11 print (Frand)
12 print (Llerena)
13 print (Izquierdo)

```

* Ejercicio 2

Escriba un programa que siendo a=“Hola”, b=“mundo”, c=87 y d=2.33145, armar y mostrar las siguientes cadenas: “Hola mundo” (usando a y b), “-Hola-mundo-” (usando a y b), “El resultado es: 87” (usando c), “El resultado es: 87 min (5220 seg)” (usando c ambas veces), “La temperatura es: 2.3”

Algoritmo 3.3: Manipulación de cadenas y formateo en Python

```

1 # Asignar valores a variables con nombres y apellidos
2 Joe = "Hola" # Corresponde a 'a'
3 Frand = "mundo" # Corresponde a 'b'
4 Llerena = 87 # Corresponde a 'c'
5 Izquierdo = 2.33145 # Corresponde a 'd'
6
7 # 1. "Hola mundo"
8 print (f"{Joe} {Frand}")
9 # 2. "-Hola-mundo-"
10 print (f"-{Joe}-{Frand}-")
11 # 3. "El resultado es: 87"
12 print (f"El resultado es: {Llerena}")
13 # 4. "El resultado es: 87 min (5220 seg)"
14 # Realizamos el calculo de segundos (87 * 60) dentro de la misma llave
15 print (f"El resultado es: {Llerena} min ({Llerena * 60} seg)")
16 # 5. "La temperatura es: 2.3"
17 # Utilizamos :.1f para formatear el valor a un solo decimal
18 print (f"La temperatura es: {Izquierdo:.1f}")

```

*** Ejercicio 3**

Hacer un programa que le pida una cadena al usuario (usando la función “input”) y la imprima a pantalla al revés. Por ejemplo, para el texto “Esto es así”, debería mostrar: “ísa se otsE”.

Algoritmo 3.4: Invertir una cadena de texto en Python

```

1 # 1. Solicitar la cadena al usuario
2 # usando el primer nombre como variable
3 Joe = input("Ingrese el texto que desea invertir: ")
4 # 2. Invertir la cadena usando slicing [::-1]
5 # y guardarla en el segundo nombre
6 # El formato [inicio:fin:paso] con paso -1 recorre la cadena
7 # del final hacia adelante
8 Frand = Joe[::-1]
9 # 3. Mostrar el resultado
10 print(f"El texto invertido es: {Frand}")

```

*** Ejercicio 4**

Hacer un programa que le pida un número al usuario (usando la función “input”) y muestre ese número menos dos, más dos, multiplicado por dos, dividido por dos, dividido por dos de forma entera, y elevado a la potencia de dos. Por ejemplo, para el número 7, debería mostrar: 5 9 14 3 3.5 49

Algoritmo 3.5: Operadores aritmeticos básicos en Python

```

1 # 1. Solicitar el valor al usuario
2 # (usando Joe como variable de entrada)
3 Joe = float(input("Ingrese un valor: "))
4
5 # 2. Realizar las operaciones solicitadas
6 # usando los nombres como variables
7 Frand = Joe - 2 # Resta
8 Llerena = Joe + 2 # Suma
9 Izquierdo = Joe * 2 # Producto
10 Joe_Frand = Joe // 2 # Cociente entero (Ej: 7 // 2 = 3)
11 Llerena_Izquierdo = Joe / 2 # Cociente real (Ej: 7 / 2 = 3.5)
12 Joe_Llerena = Joe ** 2 # Potencia al cuadrado (Ej: 7 ** 2 = 49)
13
14 # 3. Mostrar los resultados siguiendo el orden del ejemplo
15 print(Frand)
16 print(Llerena)
17 print(Izquierdo)
18 print(Joe_Frand)
19 print(Llerena_Izquierdo)
20 print(Joe_Llerena)

```

*** Ejercicio 5**

Pedirle un número al usuario, elevarlo al cuadrado, y mostrar los dígitos al revés y separados por espacio. Por ejemplo, si el usuario ingresa 17, la salida tiene que ser “9 8 2”.

Algoritmo 3.6: Elevar al cuadrado e invertir dígitos con espacios

```

1 # 1. Pedir valor y convertirlo a entero
2 Joe = int(input("Ingrese un valor: "))
3
4 # 2. Elevar al cuadrado
5 Frand = Joe ** 2
6

```

```

7 # 3. Convertir el resultado a texto y darle la vuelta (slicing [::-1])
8 # Ejemplo: 289 -> "982"
9 Llerena = str(Frand) [::-1]
10
11 # 4. Separar los digitos con un espacio usando .join()
12 # Izquierdo tomara cada caracter de Llerena
13 # y le pone un espacio en medio
14 Izquierdo = " ".join(Llerena)
15
16 # 5. Mostrar el resultado final
17 print(Izquierdo)

```

* Ejercicio 6

Pedirle tres números (que pueden ser con decimales) al usuario, base y altura de un rectángulo, y radio de un círculo. Calcular perímetro y área de cada figura.

Algoritmo 3.7: Cálculo de Área y Perímetro para Rectángulo y Círculo

```

1 import math
2
3 # 1. Solicitar datos al usuario (convertidos a float para decimales)
4 Joe = float(input("Ingrese la base del rectangulo: "))
5 Frand = float(input("Ingrese la altura del rectangulo: "))
6 Llerena = float(input("Ingrese el radio del circulo: "))
7
8 # 2. Calculos para el RECTANGULO
9 # Izquierdo almacenara el area, Joe_Frand el perimetro
10 Izquierdo = Joe * Frand
11 Joe_Frand = 2 * (Joe + Frand)
12
13 # 3. Calculos para el CIRCULO
14 # Llerena_Izquierdo almacenara el area, Joe_Llerena
15 # el perimetro (longitud)
16 Llerena_Izquierdo = math.pi * (Llerena ** 2)
17 Joe_Llerena = 2 * math.pi * Llerena
18
19 # 4. Mostrar resultados
20 print("\n--- RESULTADOS DEL RECTANGULO ---")
21 print(f"Area: {Izquierdo:.2f}")
22 print(f"Perimetro: {Joe_Frand:.2f}")
23
24 print("\n--- RESULTADOS DEL CIRCULO ---")
25 print(f"Area: {Llerena_Izquierdo:.2f}")
26 print(f"Perimetro (Circunferencia): {Joe_Llerena:.2f}")

```

* Ejercicio 7

Pedirle un número con muchos decimales al usuario y mostrarlo redondeado a 3 dígitos decimales, y como entero. ¿Qué tipo de redondeo usó? ¿Por qué?

Algoritmo 3.8: Redondeo y Truncamiento de valores en Python

```

1 # 1. Solicitar el valor con muchos decimales
2 Joe = float(input("Un valor con muchos decimales (ej: 15.67891): "))
3
4 # 2. Redondear a 3 decimales usando la funcion round()
5 Frand = round(Joe, 3)
6

```

```

7 # 3. Convertir a entero (esto provoca truncar)
8 Llerena = int(Joe)
9
10 # 4. Mostrar los resultados
11 print(f"Valor original: {Joe}")
12 print(f"Redondeado a 3 decimales: {Frاند}")
13 print(f"Como entero (usando int): {Llerena}")

```

Al ejecutar este código, Python aplica dos comportamientos distintos, el redondeo si se usó para los decimales. Y el redondeo si se usó para el entero.

Para la variable Frاند (decimales), usamos la función round(valor, 3). El tipo de redondeo es al más cercano. Python utiliza un estándar llamado Banker's Rounding (Redondeo de banquero) o redondeo al par más cercano en casos de empate exacto (cuando termina en .5). Para la mayoría de los casos cotidianos, si el cuarto decimal es 5 o superior, sube el tercero; si es menor a 5, lo mantiene. Esto se hace para reducir el sesgo acumulativo en cálculos estadísticos.

Para la variable Llerena (entero), al usar int(), no se realiza un redondeo matemático. El tipo de redondeo es truncamiento (Hacia cero). La función int() simplemente “corta” o elimina toda la parte decimal sin importar si el número era .1 o .99. Por ejemplo, int(15.999) resultará en 15. Es la forma más rápida de obtener la parte entera de un número en programación.

A veces, debido a cómo las computadoras almacenan números de punto flotante en binario, podrías ver resultados inesperados (como 2.675 redondeándose a 2.67 en lugar de 2.68). Si necesitas precisión financiera exacta, en el futuro podrías usar la librería decimal.

* Ejercicio 8

Sumar 0.3 diez veces. Por otro lado, multiplicar 0.3 por diez. ¿El resultado es el mismo? ¿Por qué? ¿Qué tipo de dato debe usar?

Algoritmo 3.9: Precisión de punto flotante en sumas y multiplicaciones

```

1 # 1. Sumar 0.3 diez veces usando un bucle
2 Joe = 0.0
3 for i in range(10):
4     Joe = Joe + 0.3
5
6 # 2. Multiplicar 0.3 por diez
7 Frاند = 0.3 * 10
8
9 # 3. Mostrar resultados con alta exactitud
10 print(f"Resultado de la suma (10 veces 0.3): {Joe:.20f}")
11 print(f"Resultado de la multiplicacion (0.3 * 10): {Frاند:.20f}")
12
13 # 4. Comparar directamente
14 Llerena = (Joe == Frاند)
15 print(f"Son exactamente iguales?: {Llerena}")

```

A la pregunta si ¿el resultado es el mismo?, se responde No exactamente. Si observamos la salida con muchos decimales, se nota que la suma suele dar algo como 2.9999999999999996 o similar, mientras que la multiplicación tiende a ser más cercana a 3.0 (aunque también podría variar según el procesador). Para Python, en una comparación estricta (==), el resultado será False. Esto sucede por la representación binaria de los números de punto flotante (estándar IEEE 754). Los computadores no guardan los números como base 10 (decimal), sino como base 2 (binario). El número 0.3 es un número periódico infinito en binario (algo así como intentar escribir 1/3 en decimal: 0.3333...). Al truncar ese número infinito para que quepa en la memoria, se introduce un error minúsculo. Al sumar el número diez veces, ese error se acumula, mientras que al multiplicar,

el error se procesa de forma distinta en el hardware. Para cálculos científicos o generales donde un error de 10^{-16} no importa, se usa float (el estándar por defecto). Si necesitas precisión exacta (como en aplicaciones financieras o contables donde cada centavo cuenta), debes usar el tipo de dato Decimal de la librería decimal. Para una exactitud usamos:

```
1 from decimal import Decimal
2 Joe = Decimal('0.3') + Decimal('0.3') + ... # Esto si daría 3.0 exacto
```

* Ejercicio 9

Pedirle una frase al usuario e imprima, en orden, qué consonantes se utilizaron. Por ejemplo, si se ingresó “la rana rene”, mostrar “lnr”.

Algoritmo 3.10: Extraer y ordenar consonantes únicas de una frase

```
1 # 1. Pedir la frase al usuario
2 Joe = input("Ingrese una frase: ").lower()
3
4 # 2. Definir que NO es una consonante (vocales, espacios y signos)
5 Frand = "aeiou "
6
7 # 3. Filtrar las consonantes unicas
8 # Llerena sera un conjunto (set) para evitar duplicados automaticamente
9 Llerena = set()
10
11 for caracter in Joe:
12     # Verificamos si es una letra y si no esta en nuestra lista
13     # de "no consonantes"
14     if caracter.isalpha() and caracter not in Frand:
15         Llerena.add(caracter)
16
17 # 4. Ordenar las consonantes alfabeticamente para que coincida
18 # con el ejemplo
19 # Izquierdo sera la lista ordenada unida en una sola cadena
20 Izquierdo = "".join(sorted(Llerena))
21
22 # 5. Mostrar el resultado
23 print(f"Consonantes utilizadas: {Izquierdo}")
```

* Ejercicio 10

Mostrar los números potencia de 2 menores a 10000 de la siguiente manera: 0001 0002 0004 0008 ... 8192

Algoritmo 3.11: Potencias de 2 menores a 10000 con formato de 4 dígitos

```
1 # 1. Inicializar la variable con la primera potencia (2^0 = 1)
2 Joe = 1
3
4 # 2. Definir el limite maximo
5 Frand = 10000
6
7 # 3. Ciclo para generar y mostrar las potencias
8 while Joe < Frand:
9     # El formato :04d indica:
10     # 0 -> rellenar con ceros
11     # 4 -> ancho total de 4 caracteres
12     # d -> valor entero (decimal)
```

```

13     print(f"{Joe:04d}")
14
15     # 4. Calcular la siguiente potencia de 2
16     Joe = Joe * 2

```

* Ejercicio 11

Pedir una frase al usuario y mostrar en mayúscula la inicial de cada palabra. Ejemplo, si el usuario ingresa “No somos nada”, mostrar NSN.

Algoritmo 3.12: Extraer e imprimir iniciales en mayúscula (Acrónimo)

```

1  # 1. Pedir la frase al usuario
2  Joe = input("Ingrese una frase: ")
3
4  # 2. Dividir la frase en una lista de palabras
5  # (usando el espacio como separador)
6  Frand = Joe.split()
7
8  # 3. Extraer la primera letra de cada palabra,
9  # pasarla a mayuscula y unirlas
10 # Recorremos cada 'palabra' en la lista 'Frand'
11 Llerena = "".join([palabra[0].upper() for palabra in Frand])
12
13 # 4. Mostrar el resultado final
14 print(f"Resultado: {Llerena}")

```

* Ejercicio 12

Hacer un programa que le pida una cadena al usuario, y arme un triángulo creciente y decreciente con ese texto. Por ejemplo, para el texto “klop”, el resultado sería:

```

      k
     kl
    klo
   klop
  klo
 kl
 k

```

Algoritmo 3.13: Triángulo creciente y decreciente con texto

```

1  # 1. Pedir la cadena al usuario
2  Joe = input("Ingrese una palabra o frase: ")
3
4  # 2. Obtener la longitud de la cadena
5  Frand = len(Joe)
6
7  # 3. Parte Creciente: recorremos desde 1 hasta la longitud total
8  print("--- Resultado ---")
9  for Llerena in range(1, Frand + 1):
10     # Mostramos desde el inicio hasta la posicion Llerena
11     print(Joe[:Llerena])
12
13 # 4. Parte Decreciente: recorremos desde la longitud-1 hasta 1
14 for Izquierdo in range(Frand - 1, 0, -1):

```

```

15     # Mostramos desde el inicio hasta la posicion Izquierdo
16     print (Joe[:Izquierdo])

```

* Ejercicio 13

Escribir un programa que le pregunte un número al usuario. Si el número es 5, que muestre ¡“Suerte!”; si el número es mayor a 10, que muestre “Grande!”; para los otros casos que muestre “Sin suerte, :(”.

Algoritmo 3.14: Uso de condicionales anidados en Python

```

1  # 1. Solicitar el valor al usuario
2  # Usamos float para que funcione tanto con enteros como con decimales
3  Joe = float(input("Ingrese un valor: "))
4
5  # 2. Estructura de control para evaluar las condiciones
6  if Joe == 5:
7      # Condicion de igualdad exacta
8      Frand = "Suerte!"
9  elif Joe > 10:
10     # Condicion de mayor que
11     Frand = "Grande!"
12 else:
13     # Caso por defecto (cualquier otro valor)
14     Frand = "Sin suerte, :("
15
16 # 3. Mostrar el resultado almacenado en la variable
17 print (Frand)

```

* Ejercicio 14

Escribir un programa que le pregunte un número al usuario. Si el número es 5, que muestre ¡“Suerte!”; si el número es mayor a 10, que le vuelva a pedir el número al usuario; para los otros casos que muestre “Sin suerte, :(”.

Algoritmo 3.15: Uso de bucles y condicionales combinados

```

1  # 1. Inicializamos la variable de control
2  Joe = float(input("Ingrese un valor: "))
3
4  # 2. Bucle: Mientras el valor sea mayor a 10, se volvera a pedir
5  while Joe > 10:
6      print("Grande! (Intentalo de nuevo)")
7      Joe = float(input("Ingrese un valor: "))
8
9  # 3. Una vez que el valor NO es mayor a 10, evaluamos los casos finales
10 if Joe == 5:
11     Frand = "Suerte!"
12 else:
13     Frand = "Sin suerte, :("
14
15 # 4. Mostrar el resultado final
16 print (Frand)

```

* Ejercicio 15

Hacer una función que recibe un número y contesta “par” o “impar” en función de si el número es par o no.

Algoritmo 3.16: Función para determinar si un número es par o impar

```

1 # 1. Definir la funcion usando nombres como variables
2 def Joe(Frand):
3     # El operador % calcula el residuo de la division entre 2
4     if Frand % 2 == 0:
5         return "par"
6     else:
7         return "impar"
8
9 # 2. Ejemplo de uso (Pedir valor al usuario)
10 Llerena = int(input("Ingrese un valor para verificar: "))
11
12 # 3. Llamada a la funcion y almacenamiento del resultado
13 Izquierdo = Joe(Llerena)
14
15 # 4. Mostrar el resultado
16 print(f"El valor {Llerena} es {Izquierdo}.")

```

*** Ejercicio 16**

Hacer una función que recibe dos números y devuelve “mayor” (si el primer número es mayor que el segundo), “menor”, o “iguales”.

Algoritmo 3.17: Función para comparar dos números

```

1 # 1. Definicion de la funcion con nombres y apellidos
2 def Joe(Frand, Llerena):
3     if Frand > Llerena:
4         return "mayor"
5     elif Frand < Llerena:
6         return "menor"
7     else:
8         return "iguales"
9
10 # 2. Solicitar datos al usuario para probar la funcion
11 # Usamos Izquierdo y Joe_Frand como variables de entrada
12 Izquierdo = float(input("Ingrese el primer valor: "))
13 Joe_Frand = float(input("Ingrese el segundo valor: "))
14
15 # 3. Llamada a la funcion y almacenamiento del resultado
16 resultado = Joe(Izquierdo, Joe_Frand)
17
18 # 4. Mostrar el veredicto
19 print(f"El primer valor es {resultado} que el segundo.")

```

*** Ejercicio 17**

Hacer un programa que genere un número entero al azar (módulo random) entre 0 y 1000, y le vaya pidiendo al usuario que ingrese números enteros para adivinarlo. Si el usuario ingresa un número menor que el objetivo, muestra “¡Es más alto!”; si el usuario ingresa uno mayor, muestra “¡Es más bajo!”; si el usuario acierta, muestra “¡Viva Python!”, y termina. Si el usuario no acertó en 7 intentos, muestre “¡Alpiste perdiste! Booo” y termine.

Algoritmo 3.18: Juego de adivinar número con límite de intentos

```

1 import random
2

```

```

3 # 1. Generar valor al azar y configurar intentos
4 # Joe sera el valor secreto, Frand el contador de intentos
5 Joe = random.randint(0, 1000)
6 Frand = 0
7 Llerena = False # Variable de control para saber si acerto
8 print("Piensa un valor entre 0 y 1000.", end=" ")
9 print("Tienes 7 intentos para adivinarlo.")
10 # 2. Bucle principal (maximo 7 intentos)
11 while Frand < 7:
12     # Izquierdo es el valor que ingresa el usuario
13     Izquierdo = int(input(f"Intento {Frand + 1} - Ingresa tu valor: "))
14     Frand += 1 # Aumentamos el contador de intentos
15     if Izquierdo == Joe:
16         print("Viva Python!")
17         Llerena = True
18         break # Sale del bucle inmediatamente
19     elif Izquierdo < Joe:
20         print("Es mas alto!")
21     else:
22         print("Es mas bajo!")
23 # 3. Mensaje final si agoto los intentos y no acerto
24 if not Llerena:
25     print(f"Alpiste perdiste! Booo. El valor era {Joe}")

```

* Ejercicio 18

Hacer una función que reciba un texto y devuelva el mismo texto, pero con cada palabra invertida. Por ejemplo, llamándola con “Esto es una prueba”, debe devolver “otsE se anu abeurp”.

Algoritmo 3.19: Función para invertir cada palabra de un texto

```

1 # 1. Definir la funcion con nombres y apellidos
2 def Joe(Frand):
3     # Separamos el texto en una lista de palabras
4     Llerena = Frand.split()
5     # Invertimos cada palabra individualmente
6     # y las guardamos en una lista nueva
7     # Usamos una "list comprehension" para mayor eficiencia
8     Izquierdo = [palabra[::-1] for palabra in Llerena]
9     # Unimos las palabras invertidas con un espacio de por medio
10    return " ".join(Izquierdo)
11 # 2. Ejemplo de uso del programa
12 texto_usuario = "Esto es una prueba"
13 resultado = Joe(texto_usuario)
14 # 3. Mostrar el resultado final
15 print(f"Original: {texto_usuario}")
16 print(f"Resultado: {resultado}")

```

* Ejercicio 19

Hacer una función que reciba dos palabras y que imprima línea por línea las primeras, segundas, etc. letras de ambas palabras. Por ejemplo, llamándola con "Holaz "mundo", el resultado sería:

H m

o u

l n
a d
o

Algoritmo 3.20: Imprimir letras de dos palabras fila por fila

```
1 from itertools import zip_longest
2
3 # 1. Definicion de la funcion
4 def Joe(Frand, Llerena):
5     # zip_longest empareja las letras indice por indice.
6     # fillvalue=' ' asegura que si una palabra es mas corta,
7     # se use un espacio.
8     for letra1, letra2 in zip_longest(Frand, Llerena, fillvalue=' '):
9         print(f"{letra1} {letra2}")
10 # 2. Ejemplo de uso con las palabras del enunciado
11 Izquierdo = "Hola"
12 Joe_Frand = "mundo"
13 # 3. Llamada a la funcion
14 Joe(Izquierdo, Joe_Frand)
```

* Ejercicio 20

Armar una función que reciba una tupla y devuelva si la tupla está ordenada (de menor a mayor) o no.

Algoritmo 3.21: Función para verificar si una tupla esta ordenada

```
1 # 1. Definicion de la funcion
2 def Joe(Frand):
3     # sorted(Frand) crea una lista ordenada de menor a mayor
4     # tuple(...) la convierte de nuevo a tupla para comparar
5     Llerena = tuple(sorted(Frand))
6
7     # Comparamos si la original es igual a la version ordenada
8     if Frand == Llerena:
9         return True
10    else:
11        return False
12 # 2. Ejemplos de uso
13 Izquierdo = (1, 5, 8, 10)
14 Joe_Frand = (10, 5, 8, 1)
15
16 print(f"Esta ordenada {Izquierdo}?: {Joe(Izquierdo)}")
17 print(f"Esta ordenada {Joe_Frand}?: {Joe(Joe_Frand)}")
```

Si la tupla es muy grande, existe una forma de hacerlo sin crear una copia ordenada, comparando cada elemento con el siguiente:

```
1 def Joe_eficiente(Frand):
2     # Verifica si cada elemento es menor o igual al siguiente
3     return all(Frand[i] <= Frand[i+1] for i in range(len(Frand)-1))
```

* Ejercicio 21

Escriba un algoritmo que permita hallar la suma de los N primeros números enteros.

Algoritmo 3.22: Suma de los N primeros números enteros

```

1 # 1. Pedir el valor de N al usuario
2 Joe = int(input("Ingrese la cantidad de valores a sumar (N): "))
3 # --- METODO 1: Usando un bucle (Iterativo) ---
4 Frand = 0
5 for i in range(1, Joe + 1):
6     Frand += i
7 # --- METODO 2: Usando la formula de Gauss (Matematico) ---
8 # Formula: S = (n * (n + 1)) / 2
9 Llerena = (Joe * (Joe + 1)) // 2
10 # 3. Mostrar resultados
11 print(f"Suma calculada con bucle: {Frand}")
12 print(f"Suma calculada con formula: {Llerena}")

```

Otra versión es:

Algoritmo 3.23: Suma de los N primeros números enteros en otra versión

```

1 N = input("Ingresar el valor: ")
2 suma = N
3 i = 0
4 print("Suma")
5 for i in range(N):
6     suma = suma + i;
7 print("La suma es:", suma)

```

*** Ejercicio 22**

Escriba un algoritmo que permita hallar cualquier tabla de multiplicar.

Algoritmo 3.24: Generador de tablas de multiplicar

```

1 # 1. Pedir que tabla desea y hasta donde
2 Joe = int(input("Tabla de multiplicar desea ver?: "))
3 Frand = int(input("Hasta que valor desea multiplicar?: "))
4 print(f"\n--- Tabla del {Joe} ---")
5 # 2. Bucle para generar la tabla
6 # Llerena sera el multiplicador que cambia en cada vuelta
7 for Llerena in range(1, Frand + 1):
8     # Izquierdo almacenara el resultado de la operacion
9     Izquierdo = Joe * Llerena
10    print(f"{Joe} x {Llerena} = {Izquierdo}")

```

Otra versión es:

Algoritmo 3.25: Generador de tablas de multiplicar en otra versión

```

1 N = input("Ingresar el valor de la tabla de multiplicar: ")
2 for i in range(1, 13):
3     print(" %d * %d= %d" % (N, i, N*i))

```

*** Ejercicio 23**

Escriba un algoritmo que permita hallar si es primo o no.

Algoritmo 3.26: Algoritmo para verificar números primos

```

1 # 1. Solicitar el valor al usuario
2 Joe = int(input("Ingrese un valor para verificar si es primo: "))
3 # 2. Variable de control (bandera)

```

```

4 # Llerena empezara siendo True y cambiara si encontramos un divisor
5 Llerena = True
6 # 3. Validacion inicial (valores menores a 2 no son primos)
7 if Joe < 2:
8     Llerena = False
9 else:
10    # 4. Probar divisores desde 2 hasta la raiz cuadrada
11    # del valor (mas eficiente)
12    # Frand sera nuestro divisor de prueba
13    for Frand in range(2, int(Joe**0.5) + 1):
14        if Joe % Frand == 0:
15            Llerena = False # Encontramos un divisor, no es primo
16            break # Salimos del bucle para ahorrar tiempo
17 # 5. Mostrar el resultado final usando Izquierdo
18 if Llerena:
19     Izquierdo = "es primo"
20 else:
21     Izquierdo = "no es primo"
22 print(f"El valor {Joe} {Izquierdo}.")

```

Otra versión es:

Algoritmo 3.27: Algoritmo para verificar números primos en otra versión

```

1 a=0
2 n=int(input("Ingrese valor: "))
3 for i in range(1,n+1):
4     if(n % i==0):
5         a=a+1
6 if(a!=2):
7     print("No es primo")
8 else:
9     print("si es primo")

```

* Ejercicio 24

Se tiene un arreglo de diez números, solicite al usuario un número a encontrar, el programa debe buscar el valor e indicar si lo encontró o no lo encontró y finaliza mostrando el número (si lo encontró) en la posición y ubicación en el arreglo (vector).

Algoritmo 3.28: Búsqueda secuencial en un arreglo de 10 números

```

1 # 1. Definir el arreglo de diez valores (pueden ser fijos o aleatorios)
2 # Joe sera nuestro vector/arreglo
3 Joe = [15, 22, 45, 8, 99, 54, 31, 7, 12, 60]
4 # 2. Solicitar al usuario el valor a buscar
5 Frand = int(input("Ingrese el valor a encontrar en el arreglo: "))
6 # 3. Variables para el control de la busqueda
7 Llerena = False # Bandera para saber si se encontro
8 Izquierdo = -1 # Variable para guardar la posicion (indice)
9 # 4. Proceso de buscar
10 for i in range(len(Joe)):
11     if Joe[i] == Frand:
12         Llerena = True
13         Izquierdo = i
14         break # Detener la busqueda al encontrar la primera coincidencia
15 # 5. Mostrar resultados finales
16 if Llerena:

```

```

17     # Mostramos la posicion (indice)
18     # y la ubicacion humana (posicion + 1)
19     print(f"Lo encuentro! El valor {Frاند}")
20     print(f"ubicado en la posicion {Izquierdo}")
21     print("Lugar real en el arreglo:")
22     print(f"elemento ubicado en {Izquierdo + 1}")
23 else:
24     print("No lo encuentro.")

```

* Ejercicio 25

Realice la sumatoria de los números del 1 al 10, visualizando los números de dos formas, ascendente y descendente (de forma horizontal), al final de la línea de números muestre la suma total.

Algoritmo 3.29: Sumatoria con visualizacion ascendente y descendente

```

1 # 1. Inicializar la variable para la suma total
2 Joe = 0
3 # 2. Visualizacion Ascendente (1 al 10)
4 print("Orden Ascendente:", end=" ")
5 for Frاند in range(1, 11):
6     print(Frand, end=" ")
7     Joe += Frاند # Acumulamos la suma en cada paso
8 print(f"| Suma Total: {Joe}")
9 # 3. Visualizacion Descendente (10 al 1)
10 # Usamos range(inicio, fin, paso) con paso -1 para ir hacia atras
11 print("Orden Descendente:", end=" ")
12 for Llerena in range(10, 0, -1):
13     print(Llerena, end=" ")
14 # Izquierdo almacenara el mensaje final
15 Izquierdo = f"| Suma Total: {Joe}"
16 print(Izquierdo)

```

Otra versión es:

Algoritmo 3.30: Sumatoria dinámica con visualización dual

```

1 # 1. Pedir el limite al usuario
2 Joe = int(input("Valor que desea realizar la sumatoria: "))
3 Frاند = 0
4 # 2. Generar secuencia ascendente y calcular suma
5 print(f"\nAscendente (1 a {Joe}):", end=" ")
6 for Llerena in range(1, Joe + 1):
7     print(Llerena, end=" ")
8     Frاند += Llerena
9 print(f"| Total: {Frاند}")
10 # 3. Generar secuencia descendente
11 print(f"Descendente ({Joe} a 1):", end=" ")
12 for Izquierdo in range(Joe, 0, -1):
13     print(Izquierdo, end=" ")
14 print(f"| Total: {Frاند}")

```



Laboratorio 8

Capítulo 3

3.4. Ejercicios del Laboratorio 8

* Ejercicio 1

Elabore un programa que muestre los números del 1 al 10 de forma ascendente.

Algoritmo 3.31: Contador ascendente del 1 al 10 con nuevas variables

```

1 # 1. Definir el limite superior (queremos llegar al 10)
2 Natalia = 11
3
4 # 2. Bucle para recorrer la secuencia
5 # Giselle toma los valores cada vez que itera
6 print("Valores del 1 al 10:")
7 for Giselle in range(1, Natalia):
8     # Llerena recibe el valor actual y Ayala se encarga de mostrarlo
9     Llerena = Giselle
10    Ayala = Llerena
11    print(Ayala)

```

* Ejercicio 2

Elabore un programa que muestre los números del 1 al 10 de forma descendente.

Algoritmo 3.32: Contador descendente del 10 al 1

```

1 # 1. Definir los puntos de inicio y fin
2 Natalia = 10 # Inicio de la cuenta
3 Giselle = 0 # Fin de la cuenta (se detiene en el 1)
4
5 # 2. Bucle para recorrer la secuencia hacia atras
6 print("Valores del 10 al 1:")
7 for Llerena in range(Natalia, Giselle, -1):
8     # Ayala toma el valor actual para imprimirlo
9     Ayala = Llerena
10    print(Ayala)

```

Usando una estructura **While** se tiene:

Algoritmo 3.33: Cuenta regresiva del 10 al 1 usando bucle while

```

1 # 1. Inicializamos la variable con el valor de partida
2 Natalia = 10
3
4 # 2. El bucle se ejecuta mientras Natalia sea mayor o igual a 1
5 print("Cuenta regresiva con while:")
6 while Natalia >= 1:
7     # Giselle guarda el valor actual
8     Giselle = Natalia
9
10    # Llerena y Ayala se encargan de la visualizacion
11    Llerena = Giselle
12    Ayala = Llerena
13    print(Ayala)
14
15    # 3. IMPORTANTE: Restamos 1 a Natalia en cada vuelta
16    # Si no hacemos esto, el bucle seria infinito
17    Natalia = Natalia - 1

```

Usando una estructura **Do - While** se tiene:

Algoritmo 3.34: Simulación de un bucle Hacer-Mientras del 10 al 1

```

1 # 1. Inicializamos la variable inicial
2 Natalia = 10
3
4 print("Simulando Hacer-Mientras (descendente):")
5
6 # 2. Iniciamos un bucle infinito
7 while True:
8     # --- CUERPO DEL BUCLE (Se ejecuta al menos una vez) ---
9     Giselle = Natalia
10    Llerena = Giselle
11    Ayala = Llerena
12    print(Ayala)
13
14    # Restamos para la siguiente vuelta
15    # Natalia = Natalia - 1 , o
16    Natalia -= 1
17
18    # --- CONDICION DE SALIDA (Al final del bloque) ---
19    # Si ya bajamos de 1, rompemos el bucle
20    if Natalia < 1:
21        break

```

*** Ejercicio 3**

Desarrolle un programa que dado un número determine la sumatoria del 1 hasta ese número.

Algoritmo 3.35: Sumatoria acumulada desde 1 hasta N

```

1 # 1. Natalia es el valor del final ingresado por el usuario
2 Natalia = int(input("Ingrese un valor para calcular la sumatoria: "))
3
4 # 2. Giselle procede como el acumulador (donde guardamos la suma)
5 Giselle = 0
6
7 # 3. Llerena es nuestra variable de control en el bucle
8 # Usamos Natalia + 1 para que incluya el valor ingresado
9 for Llerena in range(1, Natalia + 1):
10     Giselle = Giselle + Llerena
11
12 # 4. Ayala guarda el mensaje del resultado final
13 Ayala = f"La sumatoria de 1 hasta {Natalia} es: {Giselle}"
14 print(Ayala)

```

*** Ejercicio 4**

Determine la secuencia de Ulam para un número n determinado o ingresado por teclado.

Algoritmo 3.36: Generación de la secuencia de Ulam o Collatz

```

1 # 1. Natalia es el valor inicial ingresado por el usuario
2 Natalia = int(input("Ingrese un valor para la secuencia de Ulam: "))
3
4 print(f"Secuencia para {Natalia}:")
5
6 # 2. El proceso continua mientras el valor sea distinto de 1
7 while Natalia != 1:
8     # Mostramos el valor actual seguido de una flecha
9     print(Natalia, end=" -> ")
10

```

```

11 # 3. Aplicamos la logica de Ulam
12 # Giselle nos ayuda a verificar si es par o impar
13 Giselle = Natalia % 2
14
15 if Giselle == 0:
16     # Si es par, se divide por 2
17     Llerena = Natalia // 2
18 else:
19     # Si es impar, se multiplica por 3 y suma 1
20     Llerena = (Natalia * 3) + 1
21
22 # Actualizamos Natalia con el nuevo valor calculado en Llerena
23 Natalia = Llerena
24
25 # 4. Ahora, Ayala muestra el valor final (que siempre sera 1)
26 Ayala = Natalia
27 print(Ayala)

```

* Ejercicio 5

Determine la secuencia de Fibonacci para un número n determinado o ingresado por teclado.

Algoritmo 3.37: Generador de la serie de Fibonacci

```

1 # 1. Natalia define la cantidad de la secuencia queremos ver
2 Natalia = int(input("Ingrese la cantidad de terminos de Fibonacci: "))
3
4 # 2. Inicializamos los dos primeros valores
5 Giselle = 0
6 Llerena = 1
7
8 print(f"Secuencia de Fibonacci ({Natalia} terminos):")
9
10 # 3. Bucle para calcular y mostrar la serie
11 for i in range(Natalia):
12     # Ayala muestra el valor actual de la secuencia
13     print(Giselle, end=" ")
14
15     # Calculamos el siguiente valor sumando los dos anteriores
16     Ayala = Giselle + Llerena
17
18     # Desplazamos los valores para la siguiente vuelta
19     Giselle = Llerena
20     Llerena = Ayala

```

* Ejercicio 6

Determine el número factorial para un número n determinado o ingresado por teclado.

Algoritmo 3.38: Cálculo del factorial de un número N

```

1 # 1. Natalia es el valor que ingresa el usuario
2 Natalia = int(input("Ingrese un valor para calcular su factorial: "))
3
4 # 2. Giselle se inicializa en 1 (no en 0, porque es un producto)
5 Giselle = 1
6
7 # Validar: el factorial de 0 es 1 y no existe para negativos
8 if Natalia < 0:
9     print("El factorial no existe para valores negativos.")

```

```

10 elif Natalia == 0:
11     print("El factorial de 0 es 1.")
12 else:
13     # 3. Multiplicamos desde 1 hasta el valor Natalia
14     for Llerena in range(1, Natalia + 1):
15         Giselle = Giselle * Llerena
16
17     # 4. Ayala almacena el mensaje final
18     Ayala = f"El factorial de {Natalia} es: {Giselle}"
19     print(Ayala)

```

* Ejercicio 7

Determine la potencia para un número n determinado o ingresado por teclado.

Algoritmo 3.39: Cálculo de potencia mediante bucles

```

1 # 1. Solicitar base y exponente
2 Natalia = int(input("Ingrese la base: "))
3 Giselle = int(input("Ingrese el exponente: "))
4
5 # 2. Llerena es nuestro acumulador.
6 # Inicializamos en 1 porque es una multiplicacion.
7 Llerena = 1
8
9 # 3. El bucle se repite tantas veces
10 # como indique el exponente (Giselle)
11 for i in range(Giselle):
12     Llerena = Llerena * Natalia
13
14 # 4. Ayala guarda el resultado final
15 Ayala = f"El resultado de {Natalia} elevado"
16 print(Ayala)
17 print(f"a la {Giselle} es: {Llerena}")

```

* Ejercicio 8

Elabore un menú para los cuatro anteriores ejercicios.

Algoritmo 3.40: Menú interactivo para ejercicios matemáticos

```

1 # Natalia procede como la variable de opcion del menu
2 Natalia = -1
3
4 while Natalia != 0:
5     print("\n--- MENU DE EJERCICIOS ---")
6     print("1. Secuencia de Ulam")
7     print("2. Secuencia de Fibonacci")
8     print("3. Factorial de un valor")
9     print("4. Potencia de un valor")
10    print("0. Salir")
11
12    Natalia = int(input("\nSeleccione una alternativa: "))
13
14    if Natalia == 1:
15        # --- EJERCICIO: ULAM ---
16        Giselle = int(input("Ingrese valor para Ulam: "))
17        while Giselle != 1:
18            print(Giselle, end=" -> ")
19            if Giselle % 2 == 0:

```

```

20         Giselle //= 2
21     else:
22         Giselle = (Giselle * 3) + 1
23     print(1)
24
25     elif Natalia == 2:
26         # --- EJERCICIO: FIBONACCI ---
27         Llerena = int(input("Cantidad de valores que desea?: "))
28         Giselle, Ayala = 0, 1
29         for i in range(Llerena):
30             print(Giselle, end=" ")
31             Giselle, Ayala = Ayala, Giselle + Ayala
32         print()
33
34     elif Natalia == 3:
35         # --- EJERCICIO: FACTORIAL ---
36         Llerena = int(input("Ingrese un valor para factorial: "))
37         Giselle = 1
38         for i in range(1, Llerena + 1):
39             Giselle *= i
40         print(f"Resultado: {Giselle}")
41
42     elif Natalia == 4:
43         # --- EJERCICIO: POTENCIA ---
44         Llerena = int(input("Ingrese la base: "))
45         Ayala = int(input("Ingrese el exponente: "))
46         Giselle = 1
47         for i in range(Ayala):
48             Giselle *= Llerena
49         print(f"Resultado: {Giselle}")
50
51     elif Natalia == 0:
52         print("Saliendo del programa...")
53     else:
54         print("Alternativa no valida, intente de nuevo.")

```

* Ejercicio 9

Elabore un programa que encuentre el resultado de las Series de Taylor: aproximación de e^x .

Algoritmo 3.41: Aproximación de e^x usando Series de Taylor

```

1  import math
2
3  # 1. Natalia es el valor de x (exponente)
4  Natalia = float(input("Ingrese el valor de x para e^x: "))
5  # 2. Giselle es la cantidad de valores (a mayor valor, mas preciso)
6  Giselle = int(input("Ingrese el valor de cantidad para la serie: "))
7
8  # 3. Llerena acumula la suma de la serie
9  Llerena = 0.0
10
11 for n in range(Giselle):
12     # Ayala calcula el factorial de n de forma sencilla
13     Ayala = math.factorial(n)
14
15     # Aplicamos la formula del termino n: (x^n) / n!
16     termino = (Natalia**n) / Ayala

```

```

17     Llerena += termino
18
19     print(f"\nResultado aproximado con Taylor: {Llerena}")
20     print(f"Resultado real (math.exp): {math.exp(Natalia)}")

```

Ahora elabore un programa que encuentre el resultado de las Series de Taylor: aproximación de Seno de x.

Algoritmo 3.42: Aproximación de Seno(x) usando Taylor

```

1  import math
2
3  # 1. Natalia es el angulo en grados (luego lo pasamos a radianes)
4  Ayala = float(input("Ingrese el valor en grados: "))
5  Natalia = math.radians(Ayala) # Taylor requiere radianes
6
7  # 2. Giselle define la precision (cantidad de terminos)
8  Giselle = int(input("Ingrese la cantidad de valores: "))
9
10 # 3. Llerena acumula el valor del Seno
11 Llerena = 0.0
12
13 for n in range(Giselle):
14     # Calculamos el exponente y factorial impar: (2n + 1)
15     expo = 2 * n + 1
16
17     # Calculamos el termino con signo alternado
18     termino = ((-1)**n) * (Natalia**expo) / math.factorial(expo)
19
20     Llerena += termino
21
22 print(f"\nResultado con Taylor: {Llerena}")
23 print(f"Resultado real (math.sin): {math.sin(Natalia)}")

```

Ahora elabore un programa que encuentre el resultado de las Series de Taylor: aproximación de Coseno de x.

Algoritmo 3.43: Aproximacion de Coseno(x) usando Taylor

```

1  import math
2
3  # 1. Ayala recibe el angulo y Natalia lo convierte a radianes
4  Ayala = float(input("Ingrese el angulo en grados: "))
5  Natalia = math.radians(Ayala)
6
7  # 2. Giselle define la cantidad de iteraciones para la precision
8  Giselle = int(input("Ingrese el valor de terminos (ej. 10): "))
9
10 # 3. Llerena es el acumulador del resultado
11 Llerena = 0.0
12
13 for n in range(Giselle):
14     # En el coseno usamos exponentes pares: 2 * n
15     expo_par = 2 * n
16
17     # Calculamos el termino actual con signo alternado
18     # El primer termino siempre es 1 ya que x^0 / 0! = 1
19     termino = ((-1)**n) * (Natalia**expo_par) / math.factorial(expo_par)

```

```

20
21     Llerena += termino
22
23 print(f"\nResultado aproximado del Coseno: {Llerena}")
24 print(f"Resultado real (math.cos): {math.cos(Natalia)}")

```

* Ejercicio 10

El método de multiplicación rusa consiste en multiplicar sucesivamente por 2 el multiplicando y dividir por 2 el multiplicador hasta que el multiplicador tome el valor 1. Luego, se suman todos los multiplicandos correspondientes a los multiplicadores impares. Dicha suma es el producto de los dos números. La siguiente tabla muestra el cálculo realizado para multiplicar 37 por 12, cuyo resultado final es $12 + 48 + 384 = 444$. Desarrolle un programa que reciba como entrada el multiplicador y el multiplicando, y entregue como resultado el producto de ambos, calculado mediante el método de multiplicación rusa.

Algoritmo 3.44: Método de Multiplicación Rusa

```

1  # 1. Solicitar los datos de entrada
2  # Natalia sera el multiplicador (el que dividimos)
3  Natalia = int(input("Ingrese el multiplicador: "))
4  # Giselle sera el multiplicando (el que duplicamos)
5  Giselle = int(input("Ingrese el multiplicando: "))
6
7  # 2. Llerena guardara la suma de los resultados parciales
8  Llerena = 0
9
10 print(f"\nProceso de multiplicacion para {Natalia} x {Giselle}:")
11 print("-" * 30)
12 print("Multiplicador | Multiplicando | Es Impar?")
13
14 # 3. El ciclo continua hasta que el multiplicador (Natalia) llegue a 1
15 while Natalia >= 1:
16     # Verificamos si el multiplicador actual es impar
17     # Ayala nos sirve para evaluar la paridad
18     Ayala = Natalia % 2
19
20     if Ayala != 0:
21         # Si es impar, sumamos el multiplicando actual a la suma total
22         Llerena = Llerena + Giselle
23         es_impar = "SI (Sumar)"
24     else:
25         es_impar = "NO"
26
27     # Mostramos la fila actual de la tabla
28     print(f"{Natalia:<13} | {Giselle:<13} | {es_impar}")
29
30     # 4. Actualizamos los valores:
31     # Natalia se divide por 2 (division entera)
32     Natalia = Natalia // 2
33     # Giselle se duplica
34     Giselle = Giselle * 2
35
36 # 5. Resultado final
37 print("-" * 30)
38 print(f"El producto total es: {Llerena}")

```



Laboratorio 9

Capítulo 3

3.5. Ejercicios del Laboratorio 9

* Ejercicio 1

Elabore un programa que determine si un número es primo o no.

Algoritmo 3.45: Determinación de números primos

```

1 # 1. Anthony es el valor que ingresa el usuario
2 Anthony = int(input("Ingrese un valor para verificar si es primo: "))
3
4 # 2. Santiago es como una bandera (booleano)
5 # Asumimos que es primo hasta que se demuestre lo contrario
6 Santiago = True
7
8 # 3. Llerena es nuestro contador para probar divisores
9 if Anthony < 2:
10     Santiago = False # Los valores menores a 2 no son primos
11 else:
12     for Llerena in range(2, int(Anthony**0.5) + 1):
13         # Ayala comprueba el residuo de la division
14         Ayala = Anthony % Llerena
15
16         if Ayala == 0:
17             Santiago = False
18             break # Si encontramos un divisor, dejamos de buscar
19
20 # 4. Mostrar el resultado final
21 if Santiago:
22     print(f"El valor {Anthony} es PRIMO.")
23 else:
24     print(f"El valor {Anthony} NO ES PRIMO.")

```

* Ejercicio 2

Elabore un programa que muestre los números del 100 al 200 en bloques de cinco en cinco, utilice las estructuras repetitivas for, while y do-while para demostrar que las tres realizan el mismo resultado. Usando el bucle for:

Algoritmo 3.46: Secuencia con bucle for

```

1 print("--- Bucle FOR (100 al 200 de 5 en 5) ---")
2 # Anthony define el inicio, Santiago el fin, Llerena el paso
3 Anthony = 100
4 Santiago = 201 # Usamos 201 para incluir el 200
5 Llerena = 5
6 for Ayala in range(Anthony, Santiago, Llerena):
7     print(Ayala, end=" ")
8 print("\n")

```

Usando el bucle while:

Algoritmo 3.47: Secuencia con bucle while

```

1 print("--- Bucle WHILE ---")
2 Anthony = 100 # Inicializacion
3 while Anthony <= 200: # Condicion
4     Santiago = Anthony
5     print(Santiago, end=" ")
6     Anthony += 5 # Incremento manual de 5 en 5

```

```
7 print ("\n")
```

Usando el bucle do-while:

Algoritmo 3.48: Secuencia simulando do-while

```
1 print ("--- Bucle DO-WHILE (Simulado) ---")
2 Anthony = 100
3 while True:
4     Llerena = Anthony
5     print(Llerena, end=" ")
6
7     Anthony += 5
8
9     # Ayala evalua la condicion de salida al final
10    if Anthony > 200:
11        break
12 print ("\n")
```

Ahora con la secuencia de números correlativos de cinco en cinco:

Algoritmo 3.49: Números correlativos con saltos de línea cada 5 unidades

```
1 # --- VERSION CON BUCLE FOR ---
2 print("Valores del 100 al 200 (Bloques de 5):")
3 # Anthony define el inicio y Santiago el fin
4 Anthony = 100
5 Santiago = 200
6 # Llerena sirve para contar cuantos valores
7 # llevamos impresos en la linea actual
8 Llerena = 0
9 for Ayala in range(Anthony, Santiago + 1):
10    print(Ayala, end=" ")
11    Llerena += 1
12
13    # Si Llerena llega a 5, imprimimos
14    # un salto de linea y reiniciamos el contador
15    if Llerena == 5:
16        print() # Salto de linea
17        Llerena = 0
18
19 # --- VERSION CON BUCLE WHILE ---
20 print("\nRepitiendo con WHILE:")
21 Anthony = 100
22 Llerena = 0
23 while Anthony <= 200:
24    print(Anthony, end=" ")
25    Llerena += 1
26
27    if Llerena == 5:
28        print()
29        Llerena = 0
30
31    Anthony += 1
```

*** Ejercicio 3**

Elabore un programa que genere un número aleatorio.

Algoritmo 3.50: Generación de números aleatorios

```

1 import random
2
3 # 1. Anthony definirá el rango inferior
4 Anthony = 1
5 # 2. Santiago definirá el rango superior
6 Santiago = 100
7 # 3. Llerena genera el valor aleatorio entero entre Anthony y Santiago
8 Llerena = random.randint(Anthony, Santiago)
9 # 4. Ayala guarda el mensaje para mostrar el resultado
10 Ayala = f"El valor aleatorio generado es: {Llerena}"
11
12 print(Ayala)

```

*** Ejercicio 4**

Elabore un programa que redondee “hacia arriba” y “hacia abajo”.

Algoritmo 3.51: Redondeo hacia arriba y hacia abajo

```

1 import math
2
3 # 1. Anthony recibe el valor decimal por teclado
4 Anthony = float(input("Ingrese un valor decimal (ejemplo 7.4): "))
5 # 2. Santiago realiza el redondeo hacia arriba (Techo)
6 # El resultado será el entero más cercano mayor o igual al valor
7 Santiago = math.ceil(Anthony)
8 # 3. Llerena realiza el redondeo hacia abajo (Piso)
9 # El resultado será el entero más cercano menor o igual al valor
10 Llerena = math.floor(Anthony)
11 # 4. Ayala muestra los resultados finales
12 Ayala = f"Valor original: {Anthony}\nRedondeo arriba: {Santiago}"
13 print(Ayala)
14 print(f"\nRedondeo abajo: {Llerena}")

```

*** Ejercicio 5**

Elabore un programa que muestre los números pares comprendidos entre 10 y 20 inclusive.

Algoritmo 3.52: Números pares entre 10 y 20 inclusive

```

1 # 1. Anthony y Santiago definen el rango de búsqueda
2 Anthony = 10
3 Santiago = 20
4 print(f"Valores pares entre {Anthony} y {Santiago}:")
5 # 2. Llerena recorre los valores en el rango
6 # Usamos Santiago + 1 para que el 20 sea incluido
7 for Llerena in range(Anthony, Santiago + 1):
8
9     # 3. Ayala verifica si el valor es par
10    # Un valor es par si el residuo de dividirlo por 2 es 0
11    Ayala = Llerena % 2
12
13    if Ayala == 0:
14        print(Llerena)

```

*** Ejercicio 6**

Elabore un programa que permita ingresar mi nombre.

Algoritmo 3.53: Programa para captura de nombre

```

1 # 1. Anthony se encarga de recibir el dato desde el teclado
2 Anthony = input("Por favor, ingrese su nombre: ")
3 # 2. Santiago puede usarse para transformar el texto (ej. ponerlo en mayusculas)
4 Santiago = Anthony.upper()
5 # 3. Llerena prepara el mensaje de bienvenida
6 # personalizado
7 Llerena = "Bienvenido al sistema,"
8 # 4. Ayala une todo y muestra el resultado final en pantalla
9 Ayala = f"{Llerena} {Santiago}."
10 print(Ayala)

```

*** Ejercicio 7**

Elabore un programa que cuente la cantidad de vocales de una palabra.

Algoritmo 3.54: Contador de vocales en una palabra

```

1 # 1. Anthony recibe la palabra ingresada por el usuario
2 Anthony = input("Ingrese una palabra o frase: ")
3 # 2. Llerena define cuales son las letras que queremos buscar
4 Llerena = "aeiouAEIOU"
5 # 3. Santiago funciona como nuestro acumulador (empieza en 0)
6 Santiago = 0
7 # 4. El bucle recorre cada letra de la palabra (Ayala)
8 for Ayala in Anthony:
9     if Ayala in Llerena:
10         Santiago += 1
11
12 # Mostrar el resultado final
13 print(f"La palabra '{Anthony}' tiene {Santiago} vocales.")

```

*** Ejercicio 8**

Elabore un programa que cuente la cantidad de espacios vacíos de una frase.

Algoritmo 3.55: Contador de espacios en blanco

```

1 # 1. Anthony captura la frase completa
2 Anthony = input("Ingrese una frase: ")
3 # 2. Santiago actuara como el contador de espacios
4 Santiago = 0
5 # 3. El bucle recorre cada caracter de la frase
6 for Ayala in Anthony:
7     # 4. Llerena verifica si el caracter actual es un espacio
8     if Ayala == " ":
9         Santiago += 1
10
11 print(f"La frase ingresada tiene {Santiago} espacios vacios.")

```

*** Ejercicio 9**

Elabore un programa que valide el número de cédula ecuatoriana.

Algoritmo 3.56: Validador de cedula ecuatoriana

```

1 # 1. Anthony recibe el valor de cedula
2 Anthony = input("Ingrese su valor de cedula (10 digitos): ")
3 # Validacion basica de longitud
4 if len(Anthony) == 10 and Anthony.isdigit():
5     # 2. El decimo digito es el verificador
6     verificador_real = int(Anthony[9])

```

```

7     Santiago = 0
8
9     # 3. Iteramos por los primeros 9 digitos
10    for i in range(9):
11        Ayala = int(Anthony[i])
12
13        # Multiplicamos posiciones impares por 2 y pares por 1
14        if (i + 1) % 2 != 0:
15            Ayala = Ayala * 2
16            # Si el resultado es mayor a 9, restamos 9
17            if Ayala > 9:
18                Ayala -= 9
19
20        Santiago += Ayala
21
22    # 4. Calculamos el dígito verificador esperado (Llerena)
23    decena_superior = ((Santiago // 10) + 1) * 10
24    if Santiago % 10 == 0:
25        Llerena = 0
26    else:
27        Llerena = decena_superior - Santiago
28
29    # Comprobación final
30    if Llerena == verificador_real:
31        print("La cédula es VALIDA.")
32    else:
33        print("La cédula es INCORRECTA.")
34 else:
35    print("Entrada no válida. Debe tener 10 dígitos numéricos.")

```

* Ejercicio 10

Implemente el siguiente juego: el programa seleccionará un número aleatorio entre 0 y 100 y el jugador debe acertarlo. En cada intento el jugador propondrá una solución y se le informará si el número a acertar es menor o mayor que el propuesto. El juego termina cuando se acierte la cifra o haya realizado un máximo de 12 intentos en cuyo caso se le mostrará al jugador la calificación obtenida según la siguiente tabla: Número de intentos por categoría, 1-3 Suertudo, 4-6 Genio, 7 No está mal, 8 Se puede mejorar, ≥ 9 Qué pasa amigo.

Algoritmo 3.57: Juego de Adivina el valor con calificación

```

1  import random
2
3  # 1. Anthony genera el valor aleatorio secreto entre 0 y 100
4  Anthony = random.randint(0, 100)
5  # 2. Llerena será nuestro contador de intentos
6  Llerena = 0
7  # 3. Santiago guardará el intento del usuario (inicializado fuera del rango)
8  Santiago = -1
9  print("Bienvenido al juego! Tienes un máximo de 12 intentos.")
10 # El juego sigue mientras no acierte y no supere los 12 intentos
11 while Santiago != Anthony and Llerena < 12:
12     Llerena += 1
13     Santiago = int(input(f"Intento {Llerena} - Ingrese un valor: "))
14
15     if Santiago < Anthony:
16         print("El valor secreto es MAYOR.")

```

```

17 elif Santiago > Anthony:
18     print("El valor secreto es MENOR.")
19 else:
20     print(f"FELICIDADES! Acertaste en el intento {Llerena}.")
21 # 4. Al terminar el juego, Ayala determina la categoria segun Llerena
22 if Santiago == Anthony:
23     if 1 <= Llerena <= 3:
24         Ayala = "Suertudo"
25     elif 4 <= Llerena <= 6:
26         Ayala = "Genio"
27     elif Llerena == 7:
28         Ayala = "No esta mal"
29     elif Llerena == 8:
30         Ayala = "Se puede mejorar"
31     else:
32         Ayala = "Que pasa amigo"
33
34     print(f"Tu calificacion es: {Ayala}")
35 else:
36     print(f"Lo siento, agotaste tus 12 intentos. El valor era {Anthony}.")
37     print("Calificacion: Que pasa amigo")

```

* Ejercicio 11

Elabora un programa para el juego adivine el número. El usuario ingresa un número y si el valor ingresado es mayor, el programa debe enviar un mensaje indicando al usuario: “Ingrese un valor más bajo”.

Algoritmo 3.58: Juego de Adivina el valor con mensajes de guía

```

1 import random
2
3 # 1. Anthony genera el valor aleatorio entre 0 y 100
4 Anthony = random.randint(0, 100)
5 Llerena = 0
6 Santiago = -1
7 print("Bienvenido! Intenta adivinar el valor secreto entre 0 y 100.")
8 # El juego dura maximo 12 intentos (Llerena)
9 while Santiago != Anthony and Llerena < 12:
10     Llerena += 1
11     Santiago = int(input(f"\nIntento {Llerena}: Ingrese su valor: "))
12     # 2. Logica de guía para el usuario
13     if Santiago > Anthony:
14         print("Ingrese un valor mas bajo.")
15     elif Santiago < Anthony:
16         print("Ingrese un valor mas alto.")
17     else:
18         print(f"Excelente! Acertaste en {Llerena} intentos.")
19 # 3. Asignacion de categoria en Ayala tras terminar el bucle
20 if Santiago == Anthony:
21     if 1 <= Llerena <= 3:
22         Ayala = "Suertudo"
23     elif 4 <= Llerena <= 6:
24         Ayala = "Genio"
25     elif Llerena == 7:
26         Ayala = "No esta mal"
27     elif Llerena == 8:
28         Ayala = "Se puede mejorar"

```

```

29     else:
30         Ayala = "Que pasa amigo"
31
32     print(f"Tu calificacion es: {Ayala}")
33 else:
34     print(f"\nSe agotaron los intentos. El valor era {Anthony}.")
35     print("Calificacion: Que pasa amigo")

```

* Ejercicio 12

Elabora un programa para generar una tabla de bingo para elegir entre 75 números y lo muestra en pantalla una tabla, con las columnas de B, I, N G y O.

Algoritmo 3.59: Generador de Tabla de Bingo 75 valores

```

1  import random
2
3  # 1. Anthony almacena los valores de nuestra tabla por columnas
4  Anthony = {
5      'B': range(1, 16),
6      'I': range(16, 31),
7      'N': range(31, 46),
8      'G': range(46, 61),
9      'O': range(61, 76)
10 }
11 # 2. Creamos la matriz de la tabla seleccionando
12 # 5 valores al azar por letra
13 tabla = {}
14 for columna, Santiago in Anthony.items():
15     # Llerena selecciona 5 valores sin repetir de cada rango
16     Llerena = random.sample(Santiago, 5)
17     tabla[columna] = Llerena
18 # 3. Ponemos el espacio libre en el centro de la columna 'N'
19 tabla['N'][2] = "FREE"
20 # 4. Ayala se encarga de mostrar la tabla formateada en pantalla
21 print(" B I N G O")
22 print("-" * 30)
23
24 for i in range(5):
25     linea = ""
26     for columna in ['B', 'I', 'N', 'G', 'O']:
27         valor = tabla[columna][i]
28         # Formateamos para que cada columna ocupe 6 espacios
29         linea += f"{str(valor):<6}"
30     print(linea)

```



Laboratorio 10

Capítulo 3

3.6. Ejercicios del Laboratorio 10

* Ejercicio 1

Ingrese una cantidad que represente dólares, muestre a continuación el valor correspondiente en euros. (1 dólar estadounidense = 0,897070169 euros).

Algoritmo 3.60: Conversor de Dólares a Euros

```

1 # 1. Raquel captura la cantidad en dolares (tipo flotante)
2 Raquel = float(input("Ingrese un valor en dolares (USD): "))
3 # 2. Ayala define la tasa de cambio oficial segun el ejercicio
4 Ayala = 0.897070169
5 # 3. Carabajo realiza el calculo de la conversion
6 Carabajo = Raquel * Ayala
7 # 4. Mostramos el resultado formateado a dos decimales
8 print(f"\n--- Resultado de la conversion ---")
9 print(f"{Raquel} dolares equivalen a {Carabajo:.2f} euros.")

```

* Ejercicio 2

Ingrese tres valores de calificación sobre diez (notas de una materia), luego calcule su promedio.

Algoritmo 3.61: Cálculo de promedio de calificaciones

```

1 # 1. Raquel y Ayala capturan las primeras dos notas
2 Raquel = float(input("Ingrese la primera nota (sobre 10): "))
3 Ayala = float(input("Ingrese la segunda nota (sobre 10): "))
4 # 2. Reutilizamos Carabajo para la tercera nota temporalmente
5 nota_tres = float(input("Ingrese la tercera nota (sobre 10): "))
6 # 3. Carabajo calcula el promedio final
7 # Sumamos las tres notas y dividimos para 3
8 Carabajo = (Raquel + Ayala + nota_tres) / 3
9 # 4. Mostramos el resultado
10 print(f"\nEl promedio final de la materia es: {Carabajo:.2f}")
11 # Verificacion de estado
12 if Carabajo >= 7:
13     print("Estado: Aprobado")
14 else:
15     print("Estado: Reprobado")

```

* Ejercicio 3

Ingrese dos valores enteros, correspondientes a edades de personas, luego muestre en pantalla cuál de las dos es mayor.

Algoritmo 3.62: Comparación de dos edades

```

1 # 1. Raquel y Ayala capturan las edades
2 # (usamos int para valores enteros)
3 Raquel = int(input("Ingrese la edad de la primera persona: "))
4 Ayala = int(input("Ingrese la edad de la segunda persona: "))
5 # 2. Carabajo determinara el resultado mediante comparaciones logicas
6 if Raquel > Ayala:
7     Carabajo = f"La primera persona ({Raquel} anos)"
8     Carabajo1 = " es mayor que la segunda."
9 elif Ayala > Raquel:
10    Carabajo = f"La segunda persona ({Ayala} anos)"
11    Carabajo1 = " es mayor que la primera."
12 else:

```

```

13     Carabajo = "Ambas personas tienen la misma edad."
14 # 3. Mostrar el resultado final almacenado en Carabajo
15 print(f"\nResultado: {Carabajo}{Carabajo1}")

```

* Ejercicio 4

Ingrese un número del uno al siete, luego muestra su correspondiente día.

Algoritmo 3.63: Conversor de número a día de la semana

```

1 # 1. Raquel recibe el numero del 1 al 7
2 Raquel = int(input("Ingrese un numero del 1 al 7: "))
3 # 2. Ayala contiene la coleccion de dias (usamos una lista)
4 # Nota: La posicion 0 es lunes, la 1 es martes, etc.
5 Ayala = ["Lunes", "Martes", "Miercoles",
6          "Jueves", "Viernes", "Sabado", "Domingo"]
7 # 3. Carabajo valida si el numero esta en el rango correcto
8 if 1 <= Raquel <= 7:
9     # Restamos 1 a Raquel porque las listas en Python empiezan desde 0
10    Carabajo = f"El dia correspondiente es: {Ayala[Raquel - 1]}"
11 else:
12    Carabajo = "Error: El valor debe estar entre 1 y 7."
13 # 4. Mostrar el resultado
14 print(Carabajo)

```

* Ejercicio 5

Muestra los números del 25 al 58 utilizando el lazo repetitivo “mientras” o “while”.

Algoritmo 3.64: Bucle While del 25 al 58

```

1 # 1. Raquel define el punto de partida
2 Raquel = 25
3 # 2. Ayala define el punto de parada
4 Ayala = 58
5 print(f"Iniciando conteo desde {Raquel} hasta {Ayala}:")
6 # 3. El lazo "mientras" se ejecuta mientras Raquel
7 # sea menor o igual a Ayala
8 while Raquel <= Ayala:
9     # 4. Carabajo muestra el valor actual
10    Carabajo = Raquel
11    print(Carabajo, end=" ")
12    # IMPORTANTE: Incrementamos Raquel en 1 para evitar
13    # un bucle infinito
14    Raquel += 1

```

* Ejercicio 6

Muestra los números del 74 al 96 utilizando el lazo repetitivo “hacer mientras o hasta que” o “do while o repeat until”.

Algoritmo 3.65: Emulación de Do-While del 74 al 96

```

1 # 1. Raquel define el inicio del conteo
2 Raquel = 74
3 # 2. Ayala define el limite final
4 Ayala = 96
5 print(f"Conteo (estilo Do-While) desde {Raquel} hasta {Ayala}:")
6 # 3. Iniciamos el bucle infinito controlado
7 while True:
8     # Carabajo toma el valor actual e imprimimos
9     Carabajo = Raquel

```

```

10 print(Carabajo, end=" ")
11 # Incrementamos el contador
12 Raquel += 1
13 # 4. Condicion de salida (el "Hasta que" o "Until")
14 # Si Raquel supera el limite, rompemos el ciclo
15 if Raquel > Ayala:
16     break

```

* Ejercicio 7

Muestra los números del 11 al 99 en bloques de diez en diez utilizando el lazo repetitivo “para” o “for”.

Algoritmo 3.66: Números del 11 al 99 en bloques de 10

```

1 # 1. Raquel define el rango del 11 al 99
2 # (usamos 100 para incluir el 99)
3 Raquel = range(11, 100)
4 # 2. Carabajo sirve para contar cuantos numeros van en la fila actual
5 Carabajo = 0
6 print("Valores en bloques de diez:")
7 # 3. Ayala recorre cada valor dentro de Raquel
8 for Ayala in Raquel:
9     print(Ayala, end=" ")
10    Carabajo += 1
11    # 4. Cuando Carabajo llega a 10, hacemos un salto de linea
12    if Carabajo == 10:
13        print() # Salto de linea
14        Carabajo = 0

```

* Ejercicio 8

Ingrese una cantidad de x números del uno (1) al noventa y nueve (99), su programa finaliza cuando el usuario ingresa -888. A continuación, se mostrará en pantalla cuántos números son: impares, pares y primos.

Algoritmo 3.67: Clasificador de números

```

1 # 1. Ayala almacena los contadores en una lista pares, impares, primos
2 Ayala = [0, 0, 0]
3 print("Ingrese valores del 1 al 99 (Use -888 para salir):")
4 while True:
5     # 2. Raquel recibe la entrada del usuario
6     Raquel = int(input("Ingrese un numero: "))
7     # Condicion de salida
8     if Raquel == -888:
9         break
10    # Validacion de rango
11    if 1 <= Raquel <= 99:
12        # A. Clasificacion Par/Impar
13        if Raquel % 2 == 0:
14            Ayala[0] += 1 # Es par
15        else:
16            Ayala[1] += 1 # Es impar
17        # B. Clasificacion de Primos (Carabajo)
18        if Raquel > 1:
19            es_primo = True
20            for Carabajo in range(2, int(Raquel**0.5) + 1):
21                if Raquel % Carabajo == 0:
22                    es_primo = False

```

```

23         break
24     if es_primo:
25         Ayala[2] += 1
26     else:
27         print("Numero fuera de rango (1-99). Intente de nuevo.")
28 # 3. Mostrar resultados finales
29 print("\n--- RESUMEN DE CONTEO ---")
30 print(f"Numeros Pares: {Ayala[0]}")
31 print(f"Numeros Impares: {Ayala[1]}")
32 print(f"Numeros Primos: {Ayala[2]}")

```

* Ejercicio 9

Realizar la sumatoria de los números del 20 al 30.

Algoritmo 3.68: Sumatoria de números del 20 al 30

```

1 # 1. Definimos el rango de inicio y fin
2 Raquel = 20
3 Ayala = 30
4 # 2. Carabajo sirve para guardar la suma acumulada (empieza en 0)
5 Carabajo = 0
6 # 3. Utilizamos un bucle for para recorrer el rango
7 # Usamos Ayala + 1 para que el numero 30 sea incluido
8 for i in range(Raquel, Ayala + 1):
9     Carabajo += i
10 # 4. Mostramos el resultado de la sumatoria
11 print(f"La sumatoria de los numeros del {Raquel} al {Ayala}", end="")
12 print(f" es: {Carabajo}")

```

* Ejercicio 10

Realiza la simulación del lanzamiento de dos dados. Recuerda que son aleatorios el valor a obtener y solo puede generarse los números del 1 al 6.

Algoritmo 3.69: Simulación de lanzamiento de dos dados

```

1 import random
2 # 1. Raquel simula el lanzamiento del primer dado (rango 1 a 6)
3 Raquel = random.randint(1, 6)
4 # 2. Ayala simula el lanzamiento del segundo dado
5 Ayala = random.randint(1, 6)
6 # 3. Carabajo calcula la suma de ambos resultados
7 Carabajo = Raquel + Ayala
8 # 4. Mostrar los resultados en pantalla
9 print("--- Resultado del Lanzamiento ---")
10 print(f"Dado 1: {Raquel}")
11 print(f"Dado 2: {Ayala}")
12 print(f"Suma total: {Carabajo}")
13 # Verificacion especial (si ambos dados son iguales)
14 if Raquel == Ayala:
15     print("Genial! Sacaste un par.")

```

* Ejercicio 11

Emula un juego del lanzamiento de dos dados, si suman 7 tres veces gana el jugador.

Algoritmo 3.70: Juego: Tres veces Siete

```

1 import random
2 # 1. Carabajo funciona como nuestro contador de victorias (inicia en 0)
3 Carabajo = 0

```

```

4 intentos = 0
5 print("Iniciando el juego... El objetivo es sumar 7 tres veces.")
6 # 2. El bucle continua hasta que Carabajo llegue a 3
7 while Carabajo < 3:
8     intentos += 1
9     # Simulamos el lanzamiento de los dos dados
10    Raquel = random.randint(1, 6)
11    Ayala = random.randint(1, 6)
12    suma = Raquel + Ayala
13    print(f"Lanzamiento {intentos}: [{Raquel}] + [{Ayala}] = {suma}")
14    # 3. Verificamos si la suma es 7
15    if suma == 7:
16        Carabajo += 1
17        print(f"--> Llevas {Carabajo} vez/veces sumando siete!")
18 # 4. Al salir del bucle, el jugador ha ganado
19 print(f"\nFELICIDADES! Ganaste el juego en {intentos} lanzamientos.")

```

* Ejercicio 12

Una versión mejorada del juego anterior, para saber cuántas veces debe jugarse el juego hasta ganar 3 veces.

Algoritmo 3.71: Simulador de intentos hasta ganar

```

1 import random
2 # 1. Carabajo cuenta cuantas veces hemos sumado 7 (necesitamos 3)
3 Carabajo = 0
4 # 2. Variable para contar el total de jugadas realizadas
5 total_jugadas = 0
6 print("Contando lanzamientos hasta obtener tres '7'...")
7 # El bucle se repite hasta que Carabajo sea igual a 3
8 while Carabajo < 3:
9     total_jugadas += 1
10    # 3. Raquel y Ayala simulan los dados en cada jugada
11    Raquel = random.randint(1, 6)
12    Ayala = random.randint(1, 6)
13    # Verificamos si la suma de esta jugada es 7
14    if (Raquel + Ayala) == 7:
15        Carabajo += 1
16        print(f"Jugada {total_jugadas}: [{Raquel}]+[{Ayala}] ", end="")
17        print(f"= 7 (Victoria {Carabajo}/3)")
18 # 4. Resultado final de la simulacion
19 print("-" * 40)
20 print(f"EL JUEGO HA TERMINADO.")
21 print(f"Un total de {total_jugadas} jugadas para ganar 3 veces.")

```



Laboratorio 11

Capítulo 3

3.7. Ejercicios del Laboratorio 11

Ejercicio 1

A continuación, tienes bloques de códigos que representan la secuencia de creación de archivos. Tu misión, si decides aceptar, es hacer funcionar todo el código.

Archivo: `ejercicios.py`

Algoritmo 3.72: Proyecto de misión parte 1

```

1 from math import *
2 print (" / ")
3 print (" / ")
4 print (" / ")
5 print ("/ ")
6 #####
7 character_name = "Joe"
8 character_age = "35"
9 age = 45
10 isMale = True
11 is_male = False
12 print (" Hola soy "+character_name+" Llerena")
13 print (" y tengo "+character_age)
14 print ("Su alumno es "+str(is_male))
15 print ("Su hermano tiene "+str(age)+" años")
16 #####
17 # Usando strings
18 print (" Universidad \n Politecnica \n Salesiana")
19 frase = "Esto es un string o cadena de caracteres "
20 print (frase)
21 print (frase.index("d"))
22 print (frase.replace("es", "mira a"))
23 #####
24 # visualizar numeros
25 my_num = -4
26 print (str(my_num))
27 print (abs(my_num))
28 print (str(abs(my_num)) + " es mi numero preferido")
29 print (pow(3, 2))
30 my_num = 4
31 print (round(3.4))
32 print (floor(my_num))
33 print (sqrt(my_num))
34 # get input de users
35 name = input("Ingrese su nombre ")
36 age = input("Ingrese su edad ")
37 print ("Su nombre es " + name + " y su edad es "+ age)
38 num1 = input("Ingrese un primer numero ")
39 num2 = input("Ingrese un segundo numero ")
40 result = num1 + num2 # no es correcto
41 result1 = int(num1) + int(num2) # es correcto
42 print (result)
43 print (result1)

```

Archivo: principal1.py

Algoritmo 3.73: Proyecto de misión parte 2

```

1 # forma 1
2 # import funciones
3 # funciones.menu()
4 # forma 2
5 from funciones import menu, metodol, factorial
6 menu()

```

Archivo: funciones.py

Algoritmo 3.74: Proyecto de misión parte 2

```

1 def menu():
2     while True:
3         opcion = input("Ingrese una opcion: ")
4         if opcion==1:
5             print ("metodo 1")
6             metodol()
7         elif opcion==2:
8             print ("metodo 2")
9             n = raw_input("Ingrese numero: ")
10            r = factorial(n)
11            print ("resultado es "+str(r))
12        elif opcion==3:
13            print ("metodo 1")
14        elif opcion==4:
15            print ("metodo 1")
16        elif opcion==5:
17            print ("metodo 1")
18        elif opcion==6:
19            print ("metodo 1")
20        elif opcion==7:
21            print ("Salir")
22            break
23 def metodol():
24     nombre = raw_input("Ingrese su nombre: ")
25     veces = input("Ingrese veces a repetir: ")
26     for i in range (veces):
27         print (str(i+1) + " " + nombre)
28 def factorial(a):
29     fac = 1
30     for i in range(1, int(a)+1):
31         fac = fac * (i)
32     return fac

```

 Ejercicio 2

El código que se presenta a continuación, debe ser complementado con las líneas de código adecuadas además de corregir aquellas líneas que no corresponden a la secuencia lógica. Agrega e importa aquellas librerías necesarias para que el algoritmos se ejecute sin error.

Algoritmo 3.75: Nueva Misión a completar el siguiente código

```

1 def rellenar_matriz(filas, columnas, matriz):
2     for f in range(filas):
3         matriz.append([])
4         for c in range(columnas):
5             print("digite el valor de la posicion fila: ", (f+1), end="")
6             print("columna: ", (c+1))
7             matriz[f].append(int(input()))
8     print (matriz)
9 def sumar_matriz (matriz, filasu, columnas, filas):
10    sumafila=0
11    for x in range (filas):
12        sumafila =sumafila+ matriz [filasu][x]
13        print("la suma es: " , sumafila)
14    opciones=["1", "2", "3", "4", "5", "6"]
15    while True:
16        print('')
17        Seleccione una opcion(1, 2, 3, 4, 5, 6):
18        1. Rellenar matriz
19        2. Suma Fila
20        3. Suma columna
21        4. Suma Diagonal principal
22        5. Suma Diagonal Inversa
23        6. Salir
24        '')
25    opcion=input("Introduce una alternativa: ")
26    if not (opcion in opciones):
27        print("No selecciono ninguna opcion valida")
28        input ("Pulse para continuar")
29        continue
30    if (opcion == "1"):
31        try:
32            filas=int(input("ingrese la cantidad de filas"))
33            columnas=int(input("ingrese la cantidad de columnas"))
34            matriz=[]
35            rellenar_matriz(filas, columnas, matriz)
36            rellenado= True
37        except:
38            input = ("Pulse para escoger")
39            continue
40    if (opcion == "2"):
41        try:
42            if rellenado:
43                filasu=int(input("Digite la fila a Sumar"))
44                if (filasu >= 0 and filasu <=filas):
45                    sumar_matriz(matriz, filasu, columnas, filas)
46            print("Ya se lleno la matriz")
47        else:
48            print ("Digite una Fila en el valor correcto")
49        except:
50            print("falta no se rellena")

```



Laboratorio 12

Capítulo 3

3.8. Ejercicios del Laboratorio 12

* Ejercicio 1

Lea una matriz cuadrada. Descomponga la matriz en tres matrices: submatriz debajo de la diagonal, submatriz diagonal, y submatriz sobre la diagonal. Verifique que la suma de las tres matrices coincide con la matriz original. Muestre las matrices obtenidas.

Algoritmo 3.76: Descomposición de Matriz Cuadrada

```

1 import numpy as np
2 # 1. Definir el tamaño de la matriz (Llerena)
3 Llerena = int(input("Ingrese el tamaño de la matriz cuadrada (n): "))
4 # 2. Leer la matriz original (Raquel)
5 print(f"Ingrese los elementos de la matriz {Llerena}x{Llerena}:")
6 Raquel = []
7 for i in range(Llerena):
8     fila = [float(x) for x in input(f"Fila {i+1}: ").split()]
9     Raquel.append(fila)
10 Raquel = np.array(Raquel)
11 # 3. Inicializar las tres matrices con ceros
12 Joe = np.zeros((Llerena, Llerena)) # Debajo
13 Anthony = np.zeros((Llerena, Llerena)) # Diagonal
14 Natalia = np.zeros((Llerena, Llerena)) # Sobre
15 # 4. Descomposición lógica
16 for i in range(Llerena):
17     for j in range(Llerena):
18         if i > j:
19             Joe[i][j] = Raquel[i][j] # i > j: Debajo de la diagonal
20         elif i == j:
21             Anthony[i][j] = Raquel[i][j] # i == j: Diagonal principal
22         else:
23             Natalia[i][j] = Raquel[i][j] # i < j: Sobre la diagonal
24 # 5. Verificación de la suma (Ayala)
25 Ayala = Joe + Anthony + Natalia
26 # 6. Mostrar resultados
27 print("\n--- MATRIZ ORIGINAL (Raquel) ---")
28 print(Raquel)
29 print("\n--- SUBMATRIZ DEBAJO DE LA DIAGONAL (Joe) ---")
30 print(Joe)
31 print("\n--- SUBMATRIZ DIAGONAL (Anthony) ---")
32 print(Anthony)
33 print("\n--- SUBMATRIZ SOBRE LA DIAGONAL (Natalia) ---")
34 print(Natalia)
35 if np.array_equal(Ayala, Raquel):
36     print("\nVERIFICACION: La suma coincide perfectamente")
37     print("\ncon la matriz original.")

```

* Ejercicio 2

Lea una matriz $n \times m$. Para cada fila, muestre el producto de los elementos cuyo valor es un número par.

Algoritmo 3.77: Producto de elementos pares por fila

```

1 import numpy as np
2 # 1. Joe y Anthony definen las dimensiones de la matriz
3 Joe = int(input("Ingrese el número de filas (n): "))
4 Anthony = int(input("Ingrese el número de columnas (m): "))

```

```

5 # 2. Raquel almacena la matriz cargada por el usuario
6 print(f"Ingrese los elementos de la matriz {Joe}x{Anthony}:")
7 Raquel = []
8 for i in range(Joe):
9     fila = [int(x) for x in input(f"Fila {i+1}: ").split()]
10    Raquel.append(fila)
11 # 3. Procesamiento por fila
12 print("\n--- Resultados del producto de pares por fila ---")
13 for i in range(Joe):
14     Natalia = 1 # Reiniciamos el producto para la nueva fila
15     Ayala = False # Bandera para saber si encontramos al menos un par
16     for j in range(Anthony):
17         Llerena = Raquel[i][j]
18
19         # Verificamos si Llerena es un numero par
20         if Llerena % 2 == 0:
21             Natalia *= Llerena
22             Ayala = True
23 # 4. Mostrar el resultado de la fila actual
24 if Ayala:
25     print(f"Fila {i+1}: El producto de los pares es {Natalia}")
26 else:
27     print(f"Fila {i+1}: No se encontraron")
28     print("numeros pares (Producto: 0)")

```

* Ejercicio 3

Lea una matriz cuadrada. Muestre la suma de los elementos que no están en las dos diagonales principales.

Algoritmo 3.78: Suma de elementos fuera de las diagonales

```

1 import numpy as np
2 # 1. Joe define el tamaño de la matriz cuadrada
3 Joe = int(input("Ingrese el tamaño de la matriz cuadrada (n): "))
4 # 2. Raquel almacena la matriz
5 print(f"Ingrese los elementos de la matriz {Joe}x{Joe}:")
6 Raquel = []
7 for i in range(Joe):
8     fila = [float(x) for x in input(f"Fila {i+1}: ").split()]
9     Raquel.append(fila)
10 # 3. Anthony inicializa la suma en 0
11 Anthony = 0
12 print("\nElementos que no están en las diagonales:")
13 # 4. Recorremos la matriz con Natalia (i) y Llerena (j)
14 for Natalia in range(Joe):
15     for Llerena in range(Joe):
16         Ayala = Raquel[Natalia][Llerena]
17         # Condición para DIAGONAL PRINCIPAL: Natalia == Llerena
18         # Condición para DIAGONAL SECUNDARIA:
19         # Natalia + Llerena == Joe - 1
20         if not (Natalia == Llerena or Natalia + Llerena == Joe - 1):
21             Anthony += Ayala
22             print(f"[{Ayala}]", end=" ")
23 # 5. Resultado final
24 print("\n\nLa suma de los elementos")
25 print(f"\n\nfuera de las diagonales es: {Anthony}")

```

* Ejercicio 4

Un cuadrado semi-mágico es una matriz cuadrada conteniendo números tales que la suma, de las dos diagonales principales, producen el mismo resultado. Ejemplo. Un cuadrado semi-mágico de 4 filas y 4 columnas:

8	1	6	7
6	5	7	3
4	3	2	1
2	8	9	4

Escriba un programa que coloque números enteros aleatorios de una cifra en una matriz de 4 filas y 4 columnas. Repita el ciclo hasta que la matriz sea un cuadrado semi-mágico. El programa debe mostrar la matriz resultante y la cantidad de intentos que realizó el programa hasta llenar la matriz con éxito.

Algoritmo 3.79: Simulador del Cuadrado Semi-Magico 4x4

```

1 import random
2 # 1. Joe contara todos los intentos realizados
3 Joe = 0
4 Ayala = False
5 print("Buscando un cuadrado semi-magico...")
6 print("... Esto puede tardar unos segundos.")
7 # 2. El ciclo se repite hasta que Ayala sea True
8 while not Ayala:
9     Joe += 1
10    # Generamos una matriz 4x4 con numeros aleatorios de una cifra (0-9)
11    Raquel = []
12    for i in range(4):
13        fila = [random.randint(0, 9) for _ in range(4)]
14        Raquel.append(fila)
15    # 3. Calcular suma de la diagonal principal (Anthony)
16    Anthony = 0
17    for i in range(4):
18        Anthony += Raquel[i][i]
19    # 4. Calcular suma de la diagonal secundaria (Natalia)
20    Natalia = 0
21    for i in range(4):
22        Natalia += Raquel[i][3 - i]
23    # 5. Verificar condicion de cuadrado semi-magico
24    if Anthony == Natalia:
25        Ayala = True
26 # 6. Mostrar resultados finales
27 print("\n--- CUADRADO ENCONTRADO! ---")
28 for fila in Raquel:
29     print(fila)
30 print(f"\nSuma de las diagonales: {Anthony}")
31 print(f"Cantidad de intentos realizados: {Joe}")

```

* Ejercicio 5

Escriba un programa que coloque números aleatorios de una cifra en los cuatro bordes de una matriz. Después, rellene los elementos del interior de la matriz con números aleatorios de una cifra, tales que cada uno sea menor o igual al promedio de todos los elementos en los bordes. Ejemplo. Matriz inicial de 4 x 4

```
8 1 6 7
6 0 0 3
4 0 0 1
2 8 9 4
```

El promedio es 4.91

Ejemplo. Matriz rellena

```
8 1 6 7
6 3 4 3
4 2 3 1
2 8 9 4
```

Al inicio no interesan los valores que se asignan a los elementos interiores, pues serán sustituidos. El programa debe mostrar la matriz resultante y la cantidad de intentos que realizó el programa hasta llenar la matriz con éxito.

Algoritmo 3.80: Relleno de matriz basado en promedio de bordes

```
1 import random
2 import math
3 # 1. Crear matriz vacia 4x4
4 Raquel = [[0 for _ in range(4)] for _ in range(4)]
5 Joe = 0
6 Natalia = 0
7 Llerena = 0 # Contador de intentos totales
8 # 2. Llenar los cuatro bordes con numeros aleatorios de una cifra (0-9)
9 for i in range(4):
10     for j in range(4):
11         # Condicion de borde: primera/ultima
12         # fila o primera/ultima columna
13         if i == 0 or i == 3 or j == 0 or j == 3:
14             Raquel[i][j] = random.randint(0, 9)
15             Joe += Raquel[i][j]
16             Natalia += 1
17 # 3. Calcular el promedio de los bordes (Anthony)
18 Anthony = Joe / Natalia
19 # 4. Rellenar el interior (posiciones [1,1], [1,2], [2,1], [2,2])
20 # Repetimos intentos hasta que cada numero cumpla la condicion
21 for i in range(1, 3):
22     for j in range(1, 3):
23         exito = False
24         while not exito:
25             Llerena += 1 # Cada generacion cuenta como un intento
26             Ayala = random.randint(0, 9)
27             if Ayala <= Anthony:
28                 Raquel[i][j] = Ayala
29                 exito = True
30 # 5. Mostrar resultados
```

```

31 print(f"Promedio de los bordes: {Anthony:.2f}")
32 print(f"Intentos totales realizados: {Llerena}")
33 print("\n--- Matriz Resultante ---")
34 for fila in Raquel:
35     print(fila)

```

Ejercicio 6

Escriba una función `b = diagonales(a)` que reciba una matriz $n \times n$ e intercambie los elementos de la diagonal principal con los elementos de la otra diagonal. Ejemplo. Matriz inicial de 4×4

```

3 2 7 9
6 5 3 7
8 8 1 6
3 5 9 2

```

Ejemplo. Matriz modificada

```

9 2 7 3
6 3 5 7
8 1 8 6
2 5 9 3

```

En la ventana interactiva, genere una matriz cuadrada con números aleatorios de una cifra. Llame a la función y verifique si el resultado es correcto.

Algoritmo 3.81: Intercambio de diagonales en matriz cuadrada

```

1 import random
2 def diagonales(Raquel):
3     # Obtenemos el tamaño de la matriz (Anthony)
4     Anthony = len(Raquel)
5     # Copiamos la matriz para no modificar la original directamente
6     Joe = [fila[:] for fila in Raquel]
7     # Recorremos la matriz con Natalia (i)
8     for Natalia in range(Anthony):
9         # La diagonal principal es [Natalia][Natalia]
10        # La diagonal secundaria es [Natalia][Anthony - 1 - Natalia]
11        # Intercambio de valores
12        Joe[Natalia][Natalia], Joe[Natalia][Anthony - 1 - Natalia] = \
13        Joe[Natalia][Anthony - 1 - Natalia], Joe[Natalia][Natalia]
14    return Joe
15
16 # --- Ventana Interactiva ---
17 # 1. Definir tamaño y generar matriz aleatoria (Llerena)
18 Anthony = 4
19 Llerena = [[random.randint(0, 9) for _ in range(Anthony)]
20             for _ in range(Anthony)]
21 print("--- Matriz Inicial (Llerena) ---")
22 for fila in Llerena:
23     print(fila)
24 # 2. Llamar a la función y guardar en Ayala
25 Ayala = diagonales(Llerena)
26 print("\n--- Matriz Modificada (Ayala) ---")
27 for fila in Ayala:
28     print(fila)

```

* Ejercicio 7

Escriba una función que reciba una matriz. La función debe entregar un vector con la cantidad de elementos pares que contiene cada columna de la matriz: Ejemplo. Entra la siguiente información:

```

3 4 5
6 1 8
8 6 3
7 8 7

```

sale [2, 3, 1]

Algoritmo 3.82: Conteo de pares por columna

```

1 def contar_pares_columnas(Raquel):
2     # 1. Obtener dimensiones: Anthony (filas) y Natalia (columnas)
3     Anthony = len(Raquel)
4     Natalia = len(Raquel[0])
5     # 2. Joe sera el vector donde guardaremos los resultados
6     Joe = []
7     # 3. Recorremos primero las columnas (Natalia)
8     for j in range(Natalia):
9         Llerena = 0 # Reiniciamos el contador de pares para cada columna
10        # 4. Recorremos las filas para la columna actual
11        for i in range(Anthony):
12            Ayala = Raquel[i][j]
13            # Verificamos si Ayala es par
14            if Ayala % 2 == 0:
15                Llerena += 1
16            # Agregamos el resultado de la columna al vector Joe
17            Joe.append(Llerena)
18    return Joe
19 # --- Ejemplo de uso ---
20 matriz_ejemplo = [
21     [2, 4, 1],
22     [6, 8, 3],
23     [1, 2, 5],
24     [4, 6, 7]
25 ]
26 resultado = contar_pares_columnas(matriz_ejemplo)
27 print(f"Vector de pares por columna: {resultado}")

```

* Ejercicio 8

Escriba un programa que lea una matriz, llame a la función creada y determine cuál es la columna con la mayor cantidad de números pares.

Algoritmo 3.83: Identificar columna con más pares

```

1 def contar_pares_columnas(matriz):
2     filas = len(matriz)
3     columnas = len(matriz[0])
4     vector_pares = []
5     for j in range(columnas):
6         conteo = 0
7         for i in range(filas):
8             if matriz[i][j] % 2 == 0:
9                 conteo += 1

```

```

10     vector_pares.append(conteo)
11     return vector_pares
12 # --- Programa Principal ---
13 # 1. Leer dimensiones de la matriz
14 Anthony = int(input("Ingrese numero de filas (n): "))
15 Natalia = int(input("Ingrese numero de columnas (m): "))
16 # 2. Cargar la matriz Raquel
17 Raquel = []
18 print(f"Ingrese los elementos de la matriz {Anthony}x{Natalia}:")
19 for i in range(Anthony):
20     fila = [int(x) for x in input(f"Fila {i+1}: ").split()]
21     Raquel.append(fila)
22 # 3. Llamar a la funcion y obtener el vector Joe
23 Joe = contar_pares_columnas(Raquel)
24 # 4. Determinar la columna con mayor cantidad de pares
25 Llerena = max(Joe) # Cantidad maxima de pares
26 Ayala = Joe.index(Llerena)
27 # Indice de la primera columna que tiene ese maximo
28 # 5. Mostrar resultados
29 print(f"\nConteo de pares por columna: {Joe}")
30 print(f"La columna con mas pares es la columna {Ayala + 1}")
31 print(f"(con {Llerena} pares).")

```

Ejercicio 9

Escriba una función con nombre suma(n) que entregue la suma de las cifras de un número dado n. Con esta función escriba un programa que genere 10 números aleatorios entre 1 y 100 y encuentre cuál de ellos tiene la mayor suma de sus cifras.

Algoritmo 3.84: Suma de cifras y búsqueda de máximo

```

1 import random
2 def suma(n):
3     # La funcion recibe un numero n y suma sus digitos
4     # Ayala actua como el acumulador de la suma
5     Ayala = 0
6     for cifra in str(abs(n)):
7         Ayala += int(cifra)
8     return Ayala
9 # --- Programa Principal ---
10 # 1. Generar 10 numeros aleatorios entre 1 y 100 en la lista Joe
11 Joe = [random.randint(1, 100) for _ in range(10)]
12 # 2. Inicializar Anthony (el numero) y Natalia (su suma)
13 # con el primer elemento
14 Anthony = Joe[0]
15 Natalia = suma(Joe[0])
16 print("Listado de numeros y sus sumas:")
17 # 3. Recorrer cada Raquel en la lista Joe
18 for Raquel in Joe:
19     Llerena = suma(Raquel)
20     print(f"Numero: {Raquel:3} | Suma de cifras: {Llerena}")
21     # 4. Determinar si es la mayor suma encontrada hasta ahora
22     if Llerena > Natalia:
23         Natalia = Llerena
24         Anthony = Raquel
25 # 5. Mostrar el resultado final
26 print("-" * 35)
27 print(f"El numero con mayor suma de cifras es: {Anthony}")

```

```
28 print(f"La suma obtenida fue: {Natalia}")
```

* Ejercicio 10

Escriba una función llamado `cuadrado(n)` que determine si el cuadrado de un número natural n dado, es igual a la suma de los primeros n números impares. Ej. $62 = 1+3+5+7+9+11$ Escriba un programa de prueba que ingrese un dato desde el teclado, use la función y muestre el resultado en la pantalla.

Algoritmo 3.85: Verificación de propiedad de números impares

```
1 def cuadrado(n):
2     # 1. Joe calcula el cuadrado del numero
3     Joe = n ** 2
4     # 2. Anthony acumulara la suma de los primeros n impares
5     Anthony = 0
6     # Llerena sera el primer numero impar
7     Llerena = 1
8     # 3. Sumamos los primeros n impares usando un bucle Natalia
9     for Natalia in range(n):
10        Anthony += Llerena
11        # Avanzamos al siguiente numero impar
12        Llerena += 2
13    # 4. Ayala determina si la propiedad se cumple
14    Ayala = (Joe == Anthony)
15    # Devolvemos el resultado y el valor de la suma para informar
16    return Ayala, Joe, Anthony
17 # --- Programa de Prueba ---
18 # Raquel recibe el dato desde el teclado
19 Raquel = int(input("Ingrese un numero natural (n): "))
20 if Raquel > 0:
21     # Llamada a la funcion
22     es_igual, valor_cuadrado, valor_suma = cuadrado(Raquel)
23     # Mostrar resultados
24     print(f"\nAnálisis para n = {Raquel}:")
25     print(f"El cuadrado (n^2) es: {valor_cuadrado}")
26     print(f"La suma de los primeros {Raquel} impares es: {valor_suma}")
27     if es_igual:
28         print("Resultado: La propiedad se cumple!")
29     else:
30         print("Resultado: La propiedad no se cumple.")
31 else:
32     print("Por favor, ingrese un numero natural mayor a 0.")
```

* Ejercicio 11

Escriba una función `secuencial(n)` que entregue el n -ésimo término de la siguiente secuencia, en la cual cada término, a partir del tercero, se obtiene sumando los dos anteriores: 1, 1, 2, 3, 5, 8, 13, 21, ... Escriba un programa de prueba que ingrese un dato desde el teclado, use la función y muestre el resultado en la pantalla.

Algoritmo 3.86: Cálculo del n -ésimo término de Fibonacci

```
1 def secuencial(n):
2     # Casos base: si n es 1 o 2, el resultado es 1
3     if n <= 2:
4         return 1
5     # Inicializamos los dos primeros terminos
6     Natalia = 1 # n-2
```

```

7     Anthony = 1 # n-1
8     Joe = 0
9     # Calculamos desde el tercer termino hasta n
10    for Llerena in range(3, n + 1):
11        Joe = Anthony + Natalia
12        # Desplazamos los valores para la siguiente iteracion
13        Natalia = Anthony
14        Anthony = Joe
15    return Joe
16 # --- Programa de Prueba ---
17 # Raquel recibe la posicion deseada desde el teclado
18 Raquel = int(input("Ingrese la posicion (n) de la secuencia: "))
19 if Raquel > 0:
20     # Ayala guarda el resultado de llamar a la funcion
21     Ayala = secuencial(Raquel)
22
23     print(f"\nEl termino en la posicion {Raquel} es: {Ayala}")
24     # Ejemplo visual de la secuencia para n=8
25     if Raquel == 8:
26         print("Secuencia: 1, 1, 2, 3, 5, 8, 13, 21")
27 else:
28     print("Por favor, ingrese un numero entero positivo.")

```

Ejercicio 12

Escriba una función `secuencia2(n)` que entregue el n -ésimo término de la siguiente secuencia, en la cual cada término, a partir del cuarto, se obtiene sumando los tres anteriores: 1, 1, 1, 3, 5, 9, 17, 31, 57... Escriba un programa de prueba que ingrese un dato desde el teclado, use la función y muestre el resultado en la pantalla.

Algoritmo 3.87: Calculo del n -ésimo termino de Tribonacci

```

1 def secuencia2(n):
2     # Casos base: si n es 1, 2 o 3, el resultado es 1
3     if n <= 3:
4         return 1
5     # Inicializamos los tres primeros terminos
6     Llerena = 1 # n-3
7     Natalia = 1 # n-2
8     Anthony = 1 # n-1
9     Joe = 0
10    # Calculamos desde el cuarto termino hasta n
11    for i in range(4, n + 1):
12        Joe = Anthony + Natalia + Llerena
13        # Desplazamos los valores para la siguiente iteracion
14        Llerena = Natalia
15        Natalia = Anthony
16        Anthony = Joe
17    return Joe
18
19 # --- Programa de Prueba ---
20 # Raquel recibe la posicion deseada desde el teclado
21 Raquel = int(input("Ingrese (n) para la secuencia Tribonacci: "))
22 if Raquel > 0:
23     # Ayala guarda el resultado de llamar a la funcion
24     Ayala = secuencia2(Raquel)
25
26     print(f"\nEl termino en la posicion {Raquel} es: {Ayala}")

```

```

27 else:
28     print("Por favor, ingrese un numero entero positivo.")

```

* Ejercicio 13

Escriba una función `simetrico(x)` que reciba un entero y determine si es simétrico, es decir, si los dígitos opuestos alrededor del centro son iguales. Escriba un programa de prueba que genere números aleatorios entre 1 y 10000 hasta obtener un número que sea simétrico.

Algoritmo 3.88: Búsqueda aleatoria de números simétricos

```

1 import random
2 def simetrico(x):
3     # Convertimos el numero a texto para comparar extremos
4     # Anthony guardara la cadena de texto
5     Anthony = str(x)
6     # Comparamos la cadena con su version invertida
7     return Anthony == Anthony[::-1]
8
9 # --- Programa de Prueba ---
10 # Natalia contara cuantos numeros generamos hasta tener exito
11 Natalia = 0
12 Joe = False
13 print("Buscando un numero simetrico entre 1 y 10000...")
14 while not Joe:
15     Natalia += 1
16     # Generamos un numero aleatorio (Raquel)
17     Raquel = random.randint(1, 10000)
18     # Verificamos si es simetrico llamando a la funcion
19     Joe = simetrico(Raquel)
20     # Opcional: imprimir intentos para ver el progreso
21     if Natalia % 10 == 0 or Joe:
22         print(f"Intento {Natalia}: {Raquel} ->")
23         print(f"{'Simetrico!' if Joe else 'No'}")
24 # --- Resultado Final ---
25 print("-" * 40)
26 print(f"Bien! El numero simetrico encontrado es: {Raquel}")
27 print(f"Se realizaron {Natalia} intentos.")

```

Esta siguiente versión utiliza aritmética pura (divisiones y residuos) para “voltear” el número sin depender de herramientas de texto. Es la forma clásica en la que se enseña lógica de programación y matemáticas computacionales.

Algoritmo 3.89: Simetría numérica mediante lógica matemática

```

1 import random
2 def simetrico(x):
3     # Caso para numeros de una sola cifra
4     if x < 10:
5         return True
6     Joe = x
7     Anthony = 0
8     # 1. Construimos el numero invertido matematicamente
9     while Joe > 0:
10        # Extraemos el ultimo digito (Llerena)
11        Llerena = Joe % 10
12        # Lo agregamos a Anthony desplazando los anteriores una posicion
13        Anthony = (Anthony * 10) + Llerena
14        # Eliminamos el ultimo digito de Joe

```

```

15     Joe = Joe // 10
16     # 2. Si el original es igual al invertido, es simetrico
17     return x == Anthony
18
19 # --- Programa de Prueba ---
20 Natalia = 0
21 Ayala = False
22 print("Buscando numero simetrico (Version Matematica)...")
23 while not Ayala:
24     Natalia += 1
25     Raquel = random.randint(1, 10000)
26     # Verificamos la simetria
27     Ayala = simetrico(Raquel)
28     if Ayala:
29         print(f"\nENCONTRADO!")
30         print(f"Numero: {Raquel}")
31         print(f"Intentos: {Natalia}")

```

* Ejercicio 14

Escriba una función final(n) que entregue como resultado la cantidad de veces que debe lanzarse un dado hasta que salga un número n dado como parámetro. Escriba un programa de prueba que ingrese un dato desde el teclado, use la función y muestre el resultado en la pantalla.

Algoritmo 3.90: Simulación de lanzamientos hasta obtener n

```

1 import random
2 def final(Raquel):
3     # 1. Joe contara cuantas veces lanzamos el dado
4     Joe = 0
5     Ayala = False
6     # 2. El bucle sigue hasta que el dado sea igual a Raquel
7     while not Ayala:
8         Joe += 1
9         # Anthony simula el lanzamiento del dado (1 a 6)
10        Anthony = random.randint(1, 6)
11        # Verificamos si logramos el objetivo
12        if Anthony == Raquel:
13            Ayala = True
14    return Joe
15
16 # --- Programa de Prueba ---
17 # Natalia recibe el numero deseado por el usuario
18 Natalia = int(input("Ingrese el numero que desea obtener (1-6): "))
19 # Validacion de entrada
20 if 1 <= Natalia <= 6:
21     # Llerena recibe la cantidad de intentos desde la funcion
22     Llerena = final(Natalia)
23     print("-" * 45)
24     print(f"Para obtener el numero {Natalia}, el dado")
25     print(f"tuvo que lanzarse {Llerena} veces.")
26     print("-" * 45)
27 else:
28     print("Error: El numero debe estar entre 1 y 6.")

```

* Ejercicio 15

Escriba una función conteo(x) que determine la cantidad de términos que deben sumarse de la serie: $1*2*3 + 2*3*4 + 3*4*5 + 4*5*6 + \dots$ hasta que la suma exceda a un valor x dado. Escriba

un programa de prueba que genere un número aleatorio para x entre 1 y 1000, use la función y muestre el resultado en la pantalla.

Algoritmo 3.91: Cálculo de términos necesarios para exceder x

```

1 import random
2 def conteo(Raquel):
3     # 1. Inicializamos variables
4     Joe = 0 # Suma acumulada
5     Natalia = 0 # Cantidad de terminos
6     Llerena = 1 # Base del termino (i)
7     # 2. Bucle hasta que la suma Joe exceda el limite Raquel
8     while Joe <= Raquel:
9         # Calculamos el termino Anthony: i * (i+1) * (i+2)
10        Anthony = Llerena * (Llerena + 1) * (Llerena + 2)
11        # Sumamos al acumulador y aumentamos contadores
12        Joe += Anthony
13        Natalia += 1
14        Llerena += 1
15    return Natalia, Joe
16
17 # --- Programa de Prueba ---
18 # 3. Ayala genera un numero aleatorio entre 1 y 1000
19 Ayala = random.randint(1, 1000)
20 # 4. Llamamos a la funcion
21 total_terminos, suma_final = conteo(Ayala)
22 # 5. Mostrar resultados
23 print(f"Valor limite generado (x): {Ayala}")
24 print("-" * 45)
25 print(f"Cantidad de terminos necesarios: {total_terminos}")
26 print(f"Suma final alcanzada: {suma_final}")
27 print("-" * 45)

```

Ejercicio 16

Realiza un programa que almacene en una variable a la lista obtenida con `range(1,4)` y, a continuación, la modifique para que cada componente sea igual al cuadrado del componente original. El programa mostrará la lista resultante por pantalla.

Algoritmo 3.92: Modificación de elementos de una lista al cuadrado

```

1 # 1. Almacenamos en Raquel la lista obtenida con range(1, 4)
2 # Nota: range(1, 4) genera [1, 2, 3]
3 Raquel = list(range(1, 4))
4 print(f"Lista original: {Raquel}")
5 # 2. Modificamos cada componente por su cuadrado
6 # Usamos Joe como indice para recorrer la posicion 0, 1 y 2
7 for Joe in range(len(Raquel)):
8     # Anthony guarda el valor original de la posicion actual
9     Anthony = Raquel[Joe]
10    # Natalia calcula el cuadrado
11    Natalia = Anthony ** 2
12    # Actualizamos la componente original en Raquel
13    Raquel[Joe] = Natalia
14 # 3. Mostramos la lista resultante
15 print(f"Lista modificada (cuadrados): {Raquel}")

```

En esta versión siguiente, la técnica de List Comprehension (comprensión de listas) es una de las características más potentes y elegantes de Python. Permite crear o transformar listas en una

sola línea de código, siendo más eficiente que un bucle for tradicional.

Algoritmo 3.93: Transformación de lista usando List Comprehension

```

1 # 1. Creamos la lista inicial en Raquel
2 Raquel = list(range(1, 4))
3 # 2. Aplicamos List Comprehension para calcular los cuadrados
4 # Anthony = [expresion for elemento in lista]
5 Anthony = [Joe ** 2 for Joe in Raquel]
6 # 3. Mostrar resultados
7 print(f"Lista original (Raquel): {Raquel}")
8 print(f"Lista resultante (Anthony): {Anthony}")

```

* Ejercicio 17

Realiza un programa que almacene en una lista obtenida con `range(1, n)`, donde `n` es un entero que se pide al usuario y modifique dicha lista para que cada componente sea igual al cuadrado del componente original. El programa mostrará la lista resultante por pantalla.

Algoritmo 3.94: Cuadrados de una lista de tamaño `n`

```

1 # 1. Raquel recibe el limite superior n desde el teclado
2 Raquel = int(input("Ingrese el valor de n para crear la lista: "))
3 # 2. Joe almacena la lista inicial.
4 # Nota: range(1, Raquel) genera numeros desde 1 hasta n-1
5 Joe = list(range(1, Raquel))
6 # 3. Anthony usa List Comprehension para calcular los cuadrados
7 Anthony = [Natalia ** 2 for Natalia in Joe]
8 # 4. Mostrar los resultados
9 print(f"\nLista original (Joe): {Joe}")
10 print(f"Lista de cuadrados (Anthony): {Anthony}")

```

* Ejercicio 18

Realiza un programa que, dada una lista a cualquiera, sustituya cualquier elemento negativo por cero.

Algoritmo 3.95: Sustitución de negativos por cero

```

1 # 1. Definimos una lista Raquel con valores variados
2 Raquel = [10, -5, 3, -1, 0, 8, -22, 15]
3 # 2. Usamos List Comprehension con una condicion (if-else)
4 # Sintaxis: [valor_si_true if condicion else valor_si_false
5 # for elemento in lista]
6 Joe = [Anthony if Anthony >= 0 else 0 for Anthony in Raquel]
7 # 3. Mostrar resultados
8 print(f"Lista original (Raquel): {Raquel}")
9 print(f"Lista modificada (Joe): {Joe}")

```

* Ejercicio 19

Diseña un programa que construya una lista con los `n` primeros números primos (ojo: no los primos entre 1 y `n`, sino los `n` primeros números primos). ¿Necesitas usar `append`? ¿Puedes reservar en primer lugar un vector con `n` celdas nulas y asignarle a cada una de ellas uno de los números primos?

Algoritmo 3.96: Lista de los `n` primeros primos sin usar `append`

```

1 def es_primo(num):
2     if num < 2: return False
3     for i in range(2, int(num**0.5) + 1):
4         if num % i == 0:

```

```

5         return False
6     return True
7
8     # --- Programa Principal ---
9     # 1. Raquel es la cantidad de primos solicitada
10    Raquel = int(input("Cuántos números primos deseas obtener?: "))
11    # 2. Reservamos un vector con n celdas nulas (None o 0)
12    # Esto evita el uso de .append()
13    Joe = [None] * Raquel
14    Anthony = 0 # Contador de primos encontrados
15    Natalia = 2 # Primer número a evaluar
16    # 3. Bucle hasta llenar las celdas de Joe
17    while Anthony < Raquel:
18        # Ayala verifica si Natalia es primo
19        Ayala = es_primo(Natalia)
20        if Ayala:
21            # Asignamos directamente a la celda usando el índice Llerena
22            Llerena = Anthony
23            Joe[Llerena] = Natalia
24            Anthony += 1
25        # Pasamos al siguiente número para evaluar
26        Natalia += 1
27    # 4. Mostrar el resultado
28    print(f"\nLos {Raquel} primeros números primos son:")
29    print(Joe)

```

Ejercicio 20

Diseña un programa que elimine de una lista todos los elementos de índice par y muestre por pantalla el resultado. (Ejemplo: si trabaja con la lista [1, 2, 1, 5, 0, 3], esta pasará a ser [2, 5, 3]).

Algoritmo 3.97: Eliminación de índices pares

```

1 # 1. Definimos la lista original Raquel
2 Raquel = [1, 2, 1, 5, 0, 3]
3 Llerena = len(Raquel)
4 # METODO 1: Usando Slicing (Rebanado)
5 # Sintaxis: lista[inicio:fin:paso]
6 # Empezamos en 1, hasta el final, saltando de 2 en 2
7 Joe = Raquel[1::2]
8 print(f"Lista original: {Raquel}")
9 print(f"Resultado (solo índices impares): {Joe}")
10 # METODO 2: Usando List Comprehension (para entender la lógica)
11 # Solo incluimos a Natalia si su índice Anthony es impar
12 Joe_v2 = [Raquel[Anthony] for Anthony in range(Llerena)
13           if Anthony % 2 != 0]
14 # Comprobación: [2, 5, 3]

```

Nota adicional: El operador `is` en Python es muy especial porque no compara si dos valores son iguales (como lo hace `==`), sino si dos objetos son exactamente el mismo en la memoria (es decir, si comparten la misma dirección de memoria).

Una versión que utiliza `is` para identificar qué elementos de la lista original no son los mismos que los elementos de índice par que queremos descartar.

Algoritmo 3.98: Uso de `is` para filtrar elementos

```

1 # 1. Lista original Raquel
2 Raquel = [1, 2, 1, 5, 0, 3]
3 # 2. Creamos una lista con los elementos que queremos ELIMINAR (índices pares)

```

```

4 # Estos son: Raquel[0], Raquel[2], Raquel[4] -> [1, 1, 0]
5 eliminados = Raquel[0::2]
6 # 3. Usamos List Comprehension con 'is not'
7 # Nota: Esto solo funciona de forma segura si los objetos son unicos
8 # o si comparamos las posiciones de memoria.
9 Joe = []
10 for Anthony in range(len(Raquel)):
11     Natalia = Raquel[Anthony]
12     # Si el indice Anthony es impar, lo agregamos a Joe
13     if not (Anthony % 2 == 0):
14         Joe.append(Natalia)
15 print(f"Lista resultante: {Joe}")
16 # --- Ejemplo tecnico de 'is' ---
17 # Si quisieras saber si dos listas son la misma:
18 a = [1, 2, 3]
19 b = a
20 c = [1, 2, 3]
21 print(f"a is b: {a is b}") # True (mismo objeto)
22 print(f"a is c: {a is c}") # False (mismo valor, distinto objeto)

```

Se usa == (Igualdad) si es para comprobar si dos variables tienen el mismo valor (ej. si ambos son 5). Y se usa is (Identidad) para comprobar si dos variables son el mismo “recipiente” en la memoria del computador. Es decir, se usa is principalmente para comparar con None (ej. if x is None) o para verificar si dos variables apuntan a la misma lista u objeto complejo.

Caso 1: En el ejercicio donde reservamos un vector con n celdas nulas (Joe = [None] * Raquel), el uso de is es la mejor práctica de la industria.

Algoritmo 3.99: Uso recomendado de is con None

```

1 # En el ejercicio de los n primeros primos:
2 Joe = [None] * Raquel
3 # Para verificar si una celda sigue vacia:
4 if Joe[0] is None:
5     print("La primera posicion aun no tiene un primo")

```

None es un objeto único (singleton) en Python. Solo existe un None en toda la memoria del programa, por lo que is None es más rápido y seguro que == None.

Caso 2: Para verificar si dos listas son la misma (Identidad).

En los ejercicios de modificar listas, is es útil si necesitas saber si estás trabajando sobre la lista original o sobre una copia.

Algoritmo 3.100: Verificar si es el mismo objeto lista

```

1 Raquel = [1, 2, 3]
2 Joe = Raquel # Joe es un alias (misma direccion)
3 Anthony = Raquel[:] # Anthony es una copia (distinta direccion)
4 print(Joe is Raquel) # True: Son el mismo objeto
5 print(Anthony is Raquel)
6 # False: Tienen los mismos valores pero son distintos objetos

```

Casos donde NO es recomendable el uso de is (Es un peligro) De los ejercicios anteriores, en aquellos, donde el uso de is podría fallar son en el de Suma de cifras / Simetría: Si comparas Anthony is Raquel para ver si los números son iguales, podrías obtener resultados inconsistentes. Python “pre-crea” los números del -5 al 256 en memoria, pero para números más grandes (como 10,000), cada uno es un objeto nuevo. El otro ejercicio es en Índices pares: Como discutimos antes, si comparas valores numéricos con is, podrías terminar eliminando elementos que no querías solo porque “se parecen” en memoria.



Guido van Rossum, creador de Python

¡Muy buen trabajo!

Finalización del capítulo 3