



UTPL

La Universidad Católica de Loja

Vicerrectorado de Modalidad Abierta y a Distancia

Fundamentos de Programacion

Guía didáctica





Facultad Ingenierías y Arquitectura

Fundamentos de Programacion

Guía didáctica

Carrera	PAO Nivel
Redes y Analítica de Datos	I

Autor:

René Rolando Elizalde Solano



Diagramación y diseño digital

Ediloja Cía. Ltda.

Marcelino Champagnat s/n y París

edilocialtda@ediloja.com.ec

www.ediloja.com.ec

ISBN digital -978-9942-47-358-5

Año de edición: abril, 2025

Edición: primera edición

El autor de esta obra ha utilizado la inteligencia artificial como una herramienta complementaria. La creatividad, el criterio y la visión del autor se han mantenido intactos a lo largo de todo el proceso.

Loja-Ecuador



Los contenidos de este trabajo están sujetos a una licencia internacional Creative Commons **Reconocimiento-NoComercial-CompartirIgual 4.0** (CC BY-NC-SA 4.0). Usted es libre de **Compartir** — copiar y redistribuir el material en cualquier medio o formato. *Adaptar* — remezclar, transformar y construir a partir del material citando la fuente, bajo los siguientes términos: *Reconocimiento*- debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciante. *No Comercial*-no puede hacer uso del material con propósitos comerciales. *Compartir igual*-Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original. No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia. <https://creativecommons.org/licenses/by-nc-sa/4.0>



Índice

1. Datos de información	9
1.1 Presentación de la asignatura.....	9
1.2 Competencias genéricas de la UTPL.....	9
1.3 Competencias del perfil profesional.....	9
1.4 Problemática que aborda la asignatura	9
2. Metodología de aprendizaje	11
3. Orientaciones didácticas por resultados de aprendizaje.....	12
Primer bimestre	12
Resultado de aprendizaje 1:	12
Contenidos, recursos y actividades de aprendizaje recomendadas.....	12
Semana 1	12
Unidad 1. Introducción a la programación y solución de problemas	13
1.1. Metodología de solución de problemas	13
1.2. Lógica de la Programación	14
1.3. Algoritmos	15
1.4. Miniespecificaciones	18
1.5. Diagramas de Flujo	18
Actividades de aprendizaje recomendadas	21
Contenidos, recursos y actividades de aprendizaje recomendadas.....	23
Semana 2.....	23
Unidad 1. Introducción a la programación y solución de problemas	23
1.6. Lenguajes de programación	23
1.7. Lenguajes de alto y bajo nivel	25
1.8. Lenguajes de programación compilados e interpretados.....	26
1.9. Paradigmas de programación.....	27
1.10. Entornos de programación.....	28
1.11. Gestión de código mediante software de control de versiones.	32
Actividades de aprendizaje recomendadas	35



Autoevaluación 1	36
Contenidos, recursos y actividades de aprendizaje recomendadas.....	39
Semana 3	39
Unidad 2. Fundamentos de programación	39
2.1. Elementos básicos de la programación	39
2.2. Manejo de variables y tipos de datos	40
2.3. Manejo de operadores y expresiones.....	44
2.4. Uso de expresiones y asignaciones para crear programas en un lenguaje de programación	48
Actividades de aprendizaje recomendadas	60
Contenidos, recursos y actividades de aprendizaje recomendadas.....	61
Semana 4	61
Unidad 2. Fundamentos de programación	61
2.5. Estructuras de decisión simple – if	61
2.6. Estructuras de decisión doble - if/else	65
2.7. Estructuras de decisión múltiple.....	68
Actividades de aprendizaje recomendadas	73
Autoevaluación 2.....	75
Contenidos, recursos y actividades de aprendizaje recomendadas.....	79
Semana 5	79
Unidad 2. Fundamentos de programación	79
2.8. Ciclo repetitivo while.....	79
2.9. Ciclo repetitivo for	84
Actividades de aprendizaje recomendadas	88
Contenidos, recursos y actividades de aprendizaje recomendadas.....	89
Semana 6	89
Unidad 2. Fundamentos de programación	90
2.10. Arreglos unidimensionales.....	90
Actividades de aprendizaje recomendadas	98



Contenidos, recursos y actividades de aprendizaje recomendadas..... 99

Semana 7 99

 Unidad 2. Fundamentos de programación 100

 2.11. Arreglos bidimensionales 100

 Actividades de aprendizaje recomendadas 106

 Autoevaluación 3..... 108

Contenidos, recursos y actividades de aprendizaje recomendadas..... 112

Semana 8 112

 Actividades finales del bimestre 112

Segundo bimestre..... 114

Resultado de aprendizaje 2: 114

Contenidos, recursos y actividades de aprendizaje recomendadas..... 114

Semana 9 115

 Unidad 3. Manejo avanzado de datos y estructuras 115

 3.1. Concepto de función..... 115

 3.2. Funciones que no regresan valor 117

 3.3. Funciones que regresan valor 123

 Actividades de aprendizaje recomendadas 128

Contenidos, recursos y actividades de aprendizaje recomendadas..... 132

Semana 10 132

 Unidad 3. Manejo avanzado de datos y estructuras 132

 3.4. Paso de parámetros en funciones 132

 3.5. Variables locales y globales 139

 Actividades de aprendizaje recomendadas 144

 Autoevaluación 4..... 150

Contenidos, recursos y actividades de aprendizaje recomendadas..... 155

Semana 11 155

 Unidad 3. Manejo avanzado de datos y estructuras 155

 3.6. Conceptos básicos de recursividad 155



3.7. Ejemplos de recursividad 157

Actividades de aprendizaje recomendadas 162

Contenidos, recursos y actividades de aprendizaje recomendadas..... 165

Semana 12..... 165

Unidad 3. Manejo avanzado de datos y estructuras 165

3.8. Creación y uso de módulos y paquetes..... 165

3.9. Uso de funciones y módulos predefinidos 176

Actividades de aprendizaje recomendadas 181

Autoevaluación 5..... 185

Contenidos, recursos y actividades de aprendizaje recomendadas..... 188

Semana 13..... 188

Unidad 4. Técnicas avanzadas y operaciones I/O 189

4.1. Introducción al manejo de excepciones 189

4.2. Uso de excepciones en soluciones..... 190

Actividades de aprendizaje recomendadas 193

Contenidos, recursos y actividades de aprendizaje recomendadas..... 193

Semana 14..... 193

Unidad 4. Técnicas avanzadas y operaciones I/O 194

4.3. Conceptos sobre archivos..... 194

4.4. Lectura de archivos..... 195

4.5. Escritura de archivos..... 202

Actividades de aprendizaje recomendadas 206

Contenidos, recursos y actividades de aprendizaje recomendadas..... 210

Semana 15..... 210

Unidad 4. Técnicas avanzadas y operaciones I/O 211

4.6. Proceso de instalación de librerías externas 211

4.7. Uso de librerías externas 213

Actividades de aprendizaje recomendadas 217

Autoevaluación 6..... 219



Contenidos, recursos y actividades de aprendizaje recomendadas..... 222

Semana 16..... 222

 Actividades finales del bimestre 222

4. Autoevaluaciones 224

5. Referencias bibliográficas 231





1. Datos de información

1.1 Presentación de la asignatura



1.2 Competencias genéricas de la UTPL

Comportamiento Ético.

1.3 Competencias del perfil profesional

Modelar y analizar datos a través de técnicas apropiadas para identificar patrones y tendencias para mejorar la eficiencia de redes de telecomunicaciones públicas y privadas con responsabilidad social y códigos deontológicos

1.4 Problemática que aborda la asignatura

La asignatura Fundamentos de Programación en la carrera de Redes y Analítica de Datos permite desarrollar habilidades esenciales en la construcción de algoritmos, el diseño de miniespecificaciones y la programación en lenguajes de alto nivel. El uso de estructuras de control y la manipulación de datos son aspectos fundamentales para la automatización de procesos y el análisis estructurado de la información.

El estudio de la programación estructurada, el manejo de archivos, el uso de funciones y la gestión de excepciones proporciona a los estudiantes una base sólida para integrar soluciones tecnológicas que optimicen el procesamiento de datos y la toma de decisiones en entornos digitales.





2. Metodología de aprendizaje

Para el desarrollo de los contenidos en la asignatura Fundamentos de Programación, se utiliza la metodología de aprendizaje [Blended Learning](#), adaptada a un entorno de enseñanza en línea. Este enfoque combina actividades autónomas con sesiones síncronas y asíncronas, permitiendo que los estudiantes organicen su tiempo de estudio y fortalezcan tanto los aspectos teóricos como prácticos de la programación.

A través de actividades autónomas, los estudiantes exploran conceptos clave como algoritmos, miniespecificaciones, estructuras de control, funciones y manejo de archivos, mediante recursos interactivos, lecturas y ejercicios prácticos recomendados. Las sesiones síncronas brindan acompañamiento docente para aclarar dudas, realizar demostraciones, fomentando la participación activa. Adicionalmente, las actividades asíncronas permiten reforzar el aprendizaje con autoevaluación, evaluaciones en línea, foros de discusión y actividades práctico-experimentales.

Este modelo garantiza un aprendizaje flexible y efectivo, combinando la autonomía del estudiante con el acompañamiento docente, lo que facilita la adquisición de habilidades en programación y resolución de problemas computacionales.





3. Orientaciones didácticas por resultados de aprendizaje



Primer bimestre

Resultado de aprendizaje 1:

Desarrolla algoritmos y programas utilizando técnicas de resolución de problemas para crear soluciones eficientes y funcionales en sistemas de análisis de datos.

Para lograr este resultado de aprendizaje, la asignatura Fundamentos de Programación proporcionará una base sólida en el diseño e implementación de algoritmos, enfocándose en la resolución de problemas a través de metodologías estructuradas. A lo largo de la asignatura, los estudiantes explorarán la lógica de programación, la construcción de algoritmos eficientes y su aplicación en lenguajes de alto nivel, resaltando el uso de estructuras de control. El proceso de aprendizaje se desarrollará mediante lecturas y actividades recomendadas, donde los estudiantes diseñarán soluciones utilizando miniespecificaciones, diagramas de flujo y codificación en Python.

Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



Semana 1

Durante la Semana 1, los estudiantes podrán estudiar los conceptos fundamentales de la programación, iniciando con la metodología de solución de problemas y el desarrollo del pensamiento lógico. Se revisarán los algoritmos, su estructura y aplicación, junto con la construcción de



miniespecificaciones para diseñar soluciones eficientes. Finalmente, se abordarán los diagramas de flujo, como herramienta visual esencial para representar soluciones.

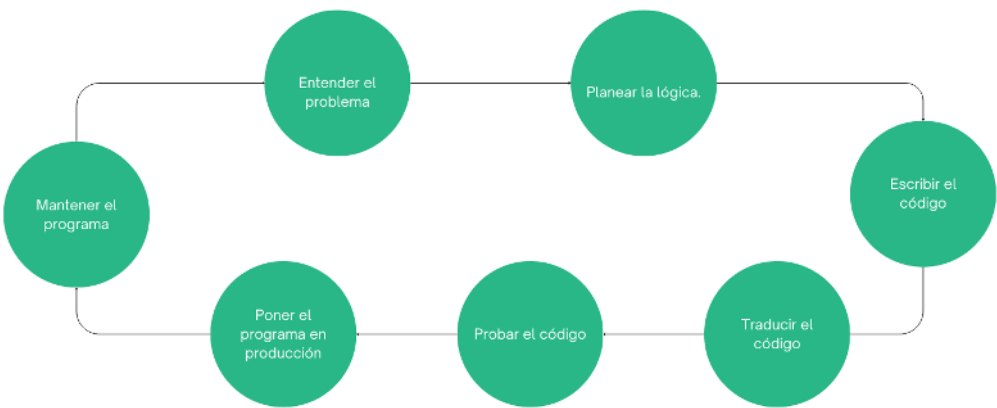
Unidad 1. Introducción a la programación y solución de problemas

1.1. Metodología de solución de problemas

El desarrollo de programas informáticos no se basa únicamente en la escritura de código, sino que requiere un enfoque estructurado que garantice la eficiencia y efectividad de la solución. Farrell, J. (2013) explica en su obra que un programador no solo ingresa instrucciones en un teclado, sino que sigue un ciclo de desarrollo bien definido, compuesto por una serie de pasos esenciales. Este ciclo metodológico permite abordar problemas de manera sistemática y reducir errores en la implementación. A continuación, se detalla un conjunto de fases, ver **Figura 1**, que siempre se sugiere seguir para resolver un problema de computación. Algunos autores pueden variar al momento de nombrar las fases, pero siempre en su conjunto llevan al desarrollador a trabajar bajo lineamientos establecidos.

Figura 1

Fases para el desarrollo de un programa



Nota. Elizalde, R., 2025.

La figura anterior muestra el conjunto de fases que se siguen para el ciclo de desarrollo básico de un programa.

Ahora le invito a revisar el siguiente video, sobre los pasos del ciclo de desarrollo de un problema:

[Fases del desarrollo de un problema](#)

1.2. Lógica de la Programación

Para llegar al concepto de Lógica de la Programación, es necesario conocer la definición de Lógica; en tal sentido, se toma la idea Herrera (2015)

“La lógica es una rama de la filosofía que estudia de manera formal las deducciones válidas que se derivan de un sistema de razonamiento fundamentado en un conjunto de reglas. Si el sistema de razonamiento mencionado se expresa en un lenguaje matemático, recibe el nombre de “lógica matemática”; en el caso de que el sistema de razonamiento utilice un lenguaje simbólico y un conjunto de reglas de inferencia recibe el nombre de “lógica simbólica”; y es precisamente esta última la que a través de los algoritmos conformados por estructuras lógicas ha permitido el desarrollo de la informática.” (p. 2)

Con base al concepto relacionando el ámbito informático, la lógica de programación es una habilidad importante para los estudiantes que se inician en el estudio de la programación, ya que permite estructurar el pensamiento y resolver problemas de manera eficiente. Se usa la lógica para aprender a descomponer problemas complejos en pasos más simples y secuenciales, facilitando su resolución mediante algoritmos o miniespecificaciones. Además, este proceso de razonamiento estructurado permite generar bases para la toma de decisiones en diversas disciplinas de la ingeniería. Finalmente, el desarrollo del pensamiento a través de la lógica de programación fomenta la capacidad de análisis y síntesis, habilidades críticas para cualquier profesional en formación.



1.3. Algoritmos

En la vida diaria se usan mentalmente algoritmos en las actividades que se realizan. Algunos autores indican las siguientes definiciones.

Fritelli, V., Guzman, A., & Tymoschuk, J. (2020), resalta que:

"El concepto de algoritmo es fundamental en el proceso de programación de una computadora, pero si nos tomamos el trabajo de observar detenidamente a nuestro alrededor, descubriremos que hay algoritmos en todas partes: nos están dando un algoritmo cuando nos indican la forma de llegar a una dirección dada, o cuando nos dan una receta de cocina. También encontramos algoritmos en las cartillas de instrucciones para usar un aparato cualquiera. Seguimos algoritmos cuando conducimos un automóvil o cualquier tipo de vehículo. Todos los procesos de cálculo matemático que normalmente realiza una persona en sus tareas cotidianas, como sumar, restar, multiplicar o dividir, están basados en algoritmos que fueron aprendidos en la escuela primaria. Como se ve, la ejecución de algoritmos forma parte indiscutiblemente de la vida moderna." (p. 9).

El autor Trejos Buriticá, O. I. (2017), expresa la siguiente idea

"Es un conjunto de pasos secuenciales y ordenados que permiten lograr un objetivo. Que sean pasos secuenciales significa que deben ser ejecutados uno después de otro y que sean pasos ordenados quiere decir que deben llevar un orden que, en algunos casos, podría ser obligatorio. Como puede notar, el ALGORITMO permite lograr un OBJETIVO, o sea, que este es el camino que necesitamos para lograrlo." (p. 29).



Los autores mencionados resaltan la importancia de entender, generar y usar algoritmos; en consecuencia, un algoritmo es una secuencia finita y ordenada de pasos diseñados para resolver un problema o llevar a cabo una tarea específica de manera lógica y estructurada. Las características esenciales son:

- Que debe estar desarrollado en lenguaje natural, lo que permite su comprensión por cualquier persona sin necesidad de conocimientos informáticos avanzados.
- Debe ser claro y preciso para garantizar que el usuario pueda interpretarlo sin dificultad.
- Todo algoritmo cuenta con un inicio y un fin bien definidos, lo que implica que su ejecución siempre debe tener un propósito y un resultado concreto.
- La estructura de un algoritmo debe presentar los pasos de manera ordenada y numerada, facilitando así su seguimiento y aplicación.
- La validación de un algoritmo requiere pruebas de simulación, para lo cual se implementan pruebas de escritorio que permiten evaluar su funcionalidad antes de su implementación definitiva.



Estas características hacen que los algoritmos sean herramientas fundamentales en la resolución de problemas, permitiendo la comunicación efectiva entre los seres humanos en la búsqueda de soluciones lógicas y estructuradas.

Ramírez Marín, J. H. (2019), indica que un algoritmo puede tener tres partes: entradas, procesos y salidas.

- Las entradas son los datos que se solicita al usuario para poder realizar los cálculos. Ya en la implementación tanto de miniespecificación o lenguaje de programación el conjunto de entradas se asocia a las directrices existan en cada implementación.
- Los procesos representan el modelo matemático que se necesita para obtener dichas salidas; aquí se efectúan, luego de análisis previo, las operaciones y cálculos necesarios para lograr llegar a la solución planteada, usando el conjunto de entradas.



- Las salidas son los resultados que se esperar del programa, dando una respuesta óptima a los requerimientos del problema. Es recomendable generar salidas con mensajes explicativos, con el objetivo que los usuarios que usen el programa a futuro sin complicaciones.

Ejemplo 1: Calcular el costo de un terreno, a partir de sus dimensiones y el costo por metro cuadrado.

1. Inicio
2. Se pide el largo del terreno
3. Se pide el ancho del terreno
4. Se pide el costo del terreno por m2.
5. Se calcula el área del terreno multiplicando el largo por el ancho del terreno
6. Se calcula el costo total del terreno multiplicando el area_terreno por el costo de terreno por metro cuadrado
7. Se muestra el costo total del terreno
8. Fin

Se recomienda, al momento de empezar una solución a un problema computaciones que involucre desarrollo, que se tome un tiempo prudencial para entender la problemática a través de la generación de un algoritmo.

Ejemplo 2: Determinar el Índice de Masa Corporal (IMC) de una persona a partir de su peso y altura.

1. Inicio
2. Se pide el peso de la persona en kilogramos.
3. Se pide la altura de la persona en metros.
4. Se calcula el Índice de Masa Corporal utilizando la fórmula: $IMC = \text{peso} / (\text{altura} * \text{altura})$
5. Se muestra el resultado del IMC.
6. Se clasifica el IMC en bajo peso, normal, sobrepeso u obesidad según los valores estándar.
7. Se muestra la clasificación del IMC en pantalla
8. Fin



1.4. Miniespecificaciones

El siguiente paso luego de realizar un algoritmo, es generar una miniespecificación o pseudocódigo. El autor Ramírez F. (2007), cuando se refiere a una miniespecificación resalta que, dentro del ciclo de desarrollo de sistema, luego de pasar por la etapa de análisis y la primera parte de diseño, se tiene la capacidad de desarrollar soluciones en pseudocódigo, considerándolo como el proceso de detallar al máximo posible las operaciones que se realizan con información de las problemáticas y llegar a una solución. La miniespecificación se convierte en un instrumento clave en la resolución de problemáticas computacionales.

No existen reglas obligatorias para generar una miniespecificación, depende mucho del autor; con base a ello, a continuación, se definen algunas características que se usaran en la presente guía:

1. Tener inicio y fin.
2. Numerar las líneas de la solución.
3. Las miniespecificaciones no depende del lenguaje de programación.
4. Si no existe una simbología dada para un proceso, el mismo debe ser descrito de forma textual (Ejemplo, acceso a un base de datos)

1.5. Diagramas de Flujo

Los Diagramas de Flujo representan una herramienta fundamental en la resolución de problemas computacionales, ya que permiten visualizar de manera clara y estructurada la secuencia de pasos necesarios para alcanzar una solución. A lo largo de su formación académica, los estudiantes se encontrarán con diversos tipos de diagramas, cada uno con aplicaciones específicas en distintas áreas del desarrollo de software. Sin embargo, en esta etapa inicial, es fundamental comprender la importancia de los Diagramas de Flujo. Estos diagramas según Ramírez F. (2007), son una representación visual de la resolución de un problema, donde se muestran las operaciones realizadas de forma secuencial y se lo usa en fases iniciales del ciclo de vida de desarrollo de un programa: Análisis y Diseño.



Un diagrama de flujo básicamente está conformado por símbolos y reglas. Existe correspondencia entre estos símbolos y las sentencias validas de un programa, por lo que se puede generar un diagrama de flujo a partir de una miniespecificación o hacerlo sin ella.

Los diagramas de flujo como herramienta esencial en la representación gráfica de algoritmos proporcionan una visión clara y estructurada del proceso a seguir para resolver un problema. A diferencia de las miniespecificaciones, que pueden estar asociadas a determinado lenguaje de programación en particular, los diagramas de flujo son completamente independientes del código y permiten visualizar de manera más intuitiva la secuencia de pasos a seguir. Gracias a su notación visual, facilitan la comprensión de la solución mediante el uso de símbolos gráficos estandarizados, lo que permite a los desarrolladores y analistas de sistemas identificar posibles errores y optimizar el diseño del algoritmo antes de su implementación en un lenguaje de programación específico.

Con relación a los símbolos usados, revisar **Figura 2**, para generación de estos diagramas, en la presente guía se usará la siguiente clasificación:

- De datos (entrada / salida)

Este tipo de símbolo se usa para los procesos de un ingreso de datos (GET) o una presentación de datos (PUT). En la práctica muchos programadores que se enfocan en la solución de un problema consideran que no es necesario colocar mensajes de entrada de datos, puesto que esos son detalles de implementación, en tal virtud podría ser suficiente utilizar PUT para mostrar resultados y GET para obtener valores en cuyo caso se colocara únicamente el nombre de las variables en las que se almacenaran los datos.

- De proceso

- Asignaciones, permiten realizar preparación de datos, a través de la asignación o cambios de estado de las diversas variables que se usen en el programa.


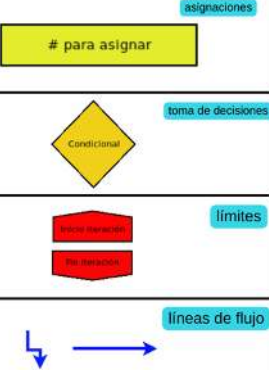
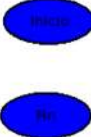


- Toma de decisiones, permite representar estructuras de decisión simples o compuestas.
- Límites y condiciones de ejecuciones iterativos
- Las líneas de flujo se usan para unir símbolos y nos ayudan a establecer el orden de ejecución de las instrucciones establecidas por los símbolos, estas líneas deben terminar en una punta de flecha para indicar a dónde se dirige el flujo

• Especiales

- Tienen este nombre porque si bien no representan ningún proceso, ayudan a identificar el inicio o el final del diagrama de flujo.

Figura 2
Símbolos usados en la creación de Diagramas de Flujo

De datos (entrada / salida)	De proceso	Especiales
		

Nota. Elizalde, R., 2025.

La figura anterior indica los diversos símbolos usados en la generación de diagramas de flujo.

Ejemplo 1: Calcular el costo de un terreno, a partir de sus dimensiones y el costo por metro cuadrado.

La solución la puede revisar en la **Figura 3**.

Figura 3

Ejemplo de Diagrama de Flujo

Ejemplo Diagrama de Flujo

Algoritmo:

1. Inicio
2. Se pide el largo del terreno
3. Se pide el ancho del terreno
4. Se pide el costo del terreno por m2.
5. Se calcula el área del terreno multiplicando el largo por el ancho del terreno
6. Se calcula el costo total del terreno multiplicando el área terreno por el costo de terreno por metro cuadrado
7. Se muestra el costo total del terreno
8. Fin



Nota. Elizalde, R., 2025.

La figura anterior indica la estructura de un diagrama de flujo, en función de un algoritmo, previo realizado.



Actividades de aprendizaje recomendadas

Es hora de reforzar los conocimientos adquiridos resolviendo las siguientes actividades:

1. Realizar una lectura detenida del [capítulo uno del texto “Lógica de Programación con PSeInt: Enfoque Práctico”](#), donde se abordan los conceptos de algoritmos, análisis del problema y desarrollo del algoritmo. Esta actividad permitirá al estudiante comprender la importancia de estructurar correctamente un algoritmo antes de su implementación, facilitando la resolución de problemas mediante un enfoque lógico y secuencial. Se recomienda resaltar las ideas más importantes y relacionarlas con los conceptos ya estudiados en la

primera parte de la unidad, estableciendo conexiones entre los conceptos de algoritmos y su aplicación en la programación. Finalmente, reflexionar sobre la importancia de un buen análisis previo permitirá fortalecer las bases para la programación estructurada y el desarrollo eficiente de software.

2. Estudiar y analizar el contenido del capítulo tres, [“Algoritmos”, del texto “Fundamentos Básicos de Programación: Aplicación Práctica con Scratch y Python”](#), donde se establece la relación entre el concepto de algoritmo y su representación mediante diagramas de flujo. Esta actividad permitirá al estudiante visualizar de manera estructurada la secuencia de instrucciones necesarias para resolver un problema, facilitando la comprensión de la lógica algorítmica antes de su implementación en código. Se recomienda realizar una lectura detallada del material, identificando las convenciones y símbolos utilizados en los diagramas de flujo. Además, se sugiere representar algunos algoritmos básicos mediante diagramas de flujo. Reflexionar sobre la importancia de este enfoque gráfico en la planificación y diseño de soluciones computacionales que contribuirá a mejorar la estructuración y claridad en la resolución de problemas de programación.
3. En esta actividad, usted deberá revisar y analizar las soluciones expresadas en [algoritmos](#) y [diagramas de flujo](#) (se está usando el programa DIA-UML para la solución a nivel de diagramas de flujo, usted lo puede instalar en su computador) disponibles en el repositorio proporcionado. Posteriormente, deberá diseñar su propia solución para cada [problemática](#), generando:
 - Un algoritmo, siguiendo una secuencia lógica clara.
 - Un diagrama de flujo, utilizando de [UML](#) u otro software de su preferencia que permita representar gráficamente la lógica del programa.

Se recomienda que, antes de generar su propia solución, analice detalladamente los problemas planteados, identifique por cada uno de ellos, las entradas, procesos y salidas.





Semana 2

En la Semana 2, los estudiantes empezarán el estudio de los lenguajes de programación, distinguiendo entre lenguajes de alto y bajo nivel, así como entre lenguajes compilados e interpretados. También estudiarán los paradigmas de programación, comprendiendo sus enfoques y aplicaciones. Además, explorarán los entornos de programación relacionados con el lenguaje Python y la gestión de código con software de control de versiones, herramientas esenciales para el desarrollo y mantenimiento de proyectos.

Unidad 1. Introducción a la programación y solución de problemas

1.6. Lenguajes de programación

Existen varias definiciones sobre los lenguajes de programación; a continuación, se detallan algunos.

Rodríguez-Losada González et al. (2022), lo describe como:

“Un lenguaje artificial que permite escribir un programa (conjunto de instrucciones que interpreta y ejecuta un ordenador). Por ejemplo, un sistema operativo, un navegador de Internet o un procesador de textos son programas. Para que un ordenador pueda 'entender' y hacer lo que dice un programa, sus instrucciones tienen que estar codificadas en lenguaje binario, o lenguaje máquina, compuesto únicamente por dos símbolos: 0 y 1 (p. 9).

También López, L. (2013), indica que:

“El lenguaje de programación es el medio a través del cual le comunicamos a la computadora el programa o el conjunto de instrucciones que debe ejecutar para llevar a cabo actividades o solucionar problemas. Ejemplos de lenguajes de programación son: Java, C#, C++, C, Pascal, Visual Basic, FORTRAN, COBOL, etcétera. Todo



lenguaje permite el manejo de los tres elementos que componen un programa, a saber: estructuras de datos, operaciones primitivas elementales y estructuras de control” (p. 7).

Como lo indica López, L (2013), para lograr la comunicación con la computadora, se requiere entender algunas características propias de los lenguajes de programación de en general: Alfabeto o conjunto de caracteres, Vocabulario o léxico y Gramática.

- El alfabeto o conjunto de caracteres de los lenguajes de programación están compuestos por un conjunto de caracteres que permiten la construcción de instrucciones para que un computador pueda interpretarlas y ejecutarlas. Este conjunto de caracteres, conocido como alfabeto del lenguaje, incluye tres tipos principales: caracteres alfabéticos, que pueden ser letras en minúscula o mayúscula; caracteres numéricos, representados por los dígitos del 0 al 9, y caracteres especiales, que incluyen símbolos como el punto (.), los dos puntos (:), el punto y coma (;), el signo de dólar (\$), el numeral (#) y la barra inclinada (/), entre otros.
- El vocabulario o léxico de un lenguaje de programación está compuesto por un conjunto de palabras reservadas y válidas que tienen un significado específico dentro del lenguaje. Estas palabras, como do, while, for, if, else, int, float, etc., no pueden ser utilizadas para nombrar variables u otros identificadores, ya que están destinadas a funciones específicas dentro de la sintaxis del lenguaje. Cada lenguaje de programación posee su propio conjunto de palabras reservadas, lo que define su estructura y reglas de uso. Por ejemplo, las palabras reservadas de Python son:

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

- La gramática en los lenguajes de programación establece las reglas sintácticas que deben seguirse para construir instrucciones correctamente estructuradas. Estas reglas garantizan que el código sea comprensible



tanto para el programador como para el intérprete o compilador del lenguaje. En el caso de Python, la sintaxis es clara y directa, evitando el uso excesivo de caracteres especiales como llaves {} o punto y coma; comunes en otros lenguajes como Java. Por ejemplo, en Python, la lectura de datos desde la entrada estándar se realiza mediante la función `input()`, que captura datos en formato de cadena (str), a diferencia de Java, donde se usa la clase Scanner.

1.7. Lenguajes de alto y bajo nivel

Para esta sección se solicita realizar una lectura comprensiva de la sección denominada Lenguajes de Programación del texto [Python 3: curso práctico](#). En el recurso se detalla la diferencia entre los lenguajes de alto nivel y bajo nivel.

Los lenguajes de alto nivel están diseñados para ser más comprensibles y cercanos al lenguaje humano, utilizando palabras clave y estructuras sintácticas más intuitivas. Son independientes del hardware, estos lenguajes permiten a los programadores centrarse en la lógica del problema sin preocuparse por la gestión de la memoria o la arquitectura del procesador. Entre los lenguajes de alto nivel más representativos se tiene: Python, Java, C#, JavaScript, Ruby, entre otros.

Por otro lado, los lenguajes de bajo nivel están más cercanos al hardware de la computadora; permiten un control sobre los recursos del sistema. Se dividen en dos categorías:

- Lenguaje ensamblador, es un lenguaje que traduce directamente instrucciones a código máquina. Se usan abreviaturas en inglés para representar las operaciones elementales.
- El Lenguaje de Máquina, se compone únicamente de instrucciones en código binario (0s y 1s), lo que lo hace de difícil entendimiento para los humanos, pero esencial para el funcionamiento interno del hardware. Son dependientes de cada máquina.





Los lenguajes de alto nivel facilitan la escritura de código de manera más intuitiva para los humanos, pero es importante entender que, al final del proceso, cualquier código debe traducirse a lenguaje máquina (binario) para que el hardware del computador pueda ejecutarlo. Los lenguajes de alto nivel necesitan un proceso de traducción para que sean ejecutados por el computador. Este proceso se puede realizar a través de la compilación o interpretación.

1.8. Lenguajes de programación compilados e interpretados

Hinojosa Gutiérrez, Á. (2015), indica que un lenguaje compilado:

“Es el que, tras escribir el programa, debe ser pasado por un programa especial (llamado compilador) que lo lee y crea a partir de él una versión en código máquina que es comprensible por el procesador. Al código escrito en el lenguaje de programación se le llama código fuente y a la versión compilada se le llama normalmente binario. Si hacemos algún cambio en el código fuente, es necesario volver a compilarlo para obtener un nuevo programa” (p, 17).

Mientras que el mismo autor define así al lenguaje interpretado:

“Es el que se puede ejecutar sin necesidad de ser compilado. Para ello, en lugar de un compilador, tenemos lo que se llama un intérprete, que lee el código y ejecuta las instrucciones en él contenidas. Al no haber compilación, no existe un binario, y los programas escritos en lenguajes interpretados se suelen llamar scripts” (p, 18).

Comprender la diferencia entre los lenguajes compilados e interpretados es fundamental para los desarrolladores, pues, puede influir en el rendimiento, la portabilidad y el desarrollo de software. Los lenguajes compilados, como C, C++, Rust y Go, Java, requieren un proceso de compilación que traduce el código fuente a código máquina antes de su ejecución, lo que los hace más eficientes



en términos de velocidad. En cambio, los lenguajes interpretados, como Python, JavaScript, Ruby y PHP, son ejecutados línea por línea por un intérprete, lo que facilita la depuración y el desarrollo ágil.

1.9. Paradigmas de programación

Los paradigmas de programación son considerados como enfoques o estilos que brindan directrices para estructurar y organizar instrucciones de un lenguaje de programación. Existen algunas clasificaciones para referirse a los paradigmas de programación, aquí se indicará la clasificación que realiza Hinojosa Gutiérrez, Á. (2015). Dicha clasificación habla de:

- Programación imperativa.

Es considerado como el paradigma base para el resto; se caracteriza por construir programas que tienen una secuencia de instrucciones que llegan a modificar el estado. Fundamentalmente, se usan operaciones, variables, asignaciones y estructuras como condicionales y ciclos repetitivos. En la presente guía se realiza el estudio de las bases de programación a través de este paradigma.

- Programación orientada a objetos

En la actualidad, es considerado como el paradigma que más se usa. López, L (2013) indica que este enfoque surge a finales de los años 80 e inicios de los 90 como la forma que facilita la organización y reutilización del código a través de la creación de objetos. Se basa en conceptos como clases, objetos, encapsulación, herencia y polimorfismo. Lenguajes como C++, Java y Python adoptan este paradigma, donde cada objeto contiene datos (atributos) y métodos (funciones) que definen su comportamiento. Este paradigma será estudiado en los siguientes ciclos de su carrera.

- Programación funcional



Este paradigma permite trabajar sobre datos por intermedio de funciones puras, con una semejanza a las funciones matemáticas. Se caracteriza por evitar el uso de estados compartidos y variables globales. Ejemplos de funciones son: map, filter, reduce.

1.10. Entornos de programación

En la asignatura se abordan conceptos básicos de programación relacionados con los algoritmos, miniespecificaciones y lenguajes de alto nivel.

Python es el lenguaje de programación de alto nivel que se usará en la asignatura para la puesta en marcha de las soluciones.

¿Por qué Python? En la página oficial del lenguaje se pueden revisar algunas características: Python es un lenguaje de programación ampliamente reconocido por su simplicidad y potencia, lo que lo convierte en una opción básica para estudiantes que están empezando en el mundo de la programación. Su sintaxis clara y legible facilita el aprendizaje y la escritura de código, reduciendo la complejidad en comparación con otros lenguajes. Algunas ventajas de Python:

- Ofrece una tipificación dinámica, lo que significa que los desarrolladores pueden escribir código sin necesidad de declarar explícitamente los tipos de datos, permitiendo una mayor flexibilidad en el desarrollo.
- Posee una gran biblioteca estándar, que proporciona herramientas y módulos para abordar una gran variedad de problemas.
- Tiene librerías que ayudan en diferentes áreas, desde manipulación de datos y redes hasta desarrollo web e inteligencia artificial.
- El enfoque multiplataforma, permite la ejecución de aplicaciones en distintos sistemas operativos sin necesidad de modificaciones significativas en el código.
- Siempre sobresale su gran comunidad de usuarios y su ecosistema de paquetes externos, accesibles a través de gestores como pip (Python Software Foundation, 2024).



Es momento de analizar y configurar los entornos de programación donde se podrán desarrollar soluciones en Python.

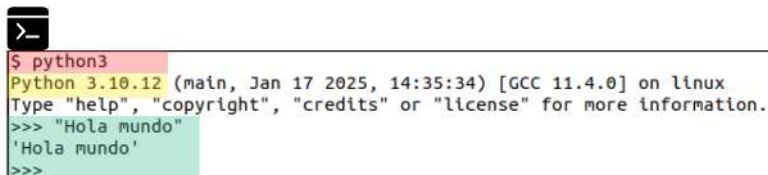
- Proceso de instalación de Python.

Python viene instalado por defecto en sistemas operativos [Linux](#) y [MacOS](#). Para comprobar que esté instalado, se deberá ingresar a la consola o terminal del sistema operativo y digitar la palabra “python” o “python3”. Se observa una imagen similar a la **Figura 4**.

Figura 4

Interprete interactivo de Python

Acceso al intérprete interactivo de Python en un terminal o consola



```
>-  
$ python3  
Python 3.10.12 (main, Jan 17 2025, 14:35:34) [GCC 11.4.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> "Hola mundo"  
'Hola mundo'  
>>>
```

Nota. Elizalde, R., 2025.

La figura describe la salida que se tiene al momento de escribir el comando python3 en un terminal o CMD. La salida puede variar según la versión de Python. Además, se indica que se puede interactuar, agregando código Python.

Cuando se escribe el comando python o python3 en la terminal de un sistema operativo compatible, como Linux o macOS, se accede al intérprete interactivo de Python, también conocido como la consola de Python o REPL

(Read-Eval-Print Loop). Este entorno permite ejecutar instrucciones de Python de manera inmediata, lo que resulta útil para pruebas rápidas de código sin necesidad de crear un archivo independiente. Si luego de ingresar el comando, existe un mensaje de error, se debe revisar o verificar si el lenguaje está instalado en el sistema operativo.

Por otro lado, en los sistemas operativos [Windows](#) se debe realizar el proceso de instalación desde cero. Para ello se recomienda seguir las indicaciones de la página oficial del [lenguaje Python](#). Los pasos que se indican son:

- Paso 1: Ingresar a la [página de descargas](#), y dirigirse a las opciones descargas para el Sistema Operativo Windows y descargar el instalador
- Paso 2: Ejecutar el archivo de instalación de Python para Windows, seguir las indicaciones y verificar que se seleccione la opción “Add Python 3.x to Path”, con el objetivo que se pueda ingresar a un terminal o consola de Windows y poder acceder al interprete.
- Paso 3: Verificar que se pueda ingresar al interprete.

Ver **Figura 5**, con los pasos indicados se tiene instalado Python en el Sistema Operativo.



Figura 5

Pasos para la instalación de Python en Windows



Nota. Elizalde, R., 2025.

Nota. En la figura se describe los pasos que se deben seguir para realizar el proceso de instalación de lenguaje de programación Python en sistema operativo Windows.

Luego de la instalación se debe ejecutar el Entorno Integrado de Desarrollo (IDE) que usará. Moreno Pérez, J. C. (2015) define un IDE como "una herramienta la cual poder desarrollar y probar proyectos en un lenguaje determinado" (p. 12).

Es necesario aclarar que un programa de Python se lo puede realizar en editores de texto sencillos como Bloc de Notas / Gedit / VIM. Si se sigue esta opción en la creación de programas, se debe usar la consola o terminal para ejecutar los scripts, usando la forma: **python nombre-archivo.py**

Recuerde que la extensión de los archivos o scripts de Python es .py

Pero también se puede elegir entre la variedad de opciones que existen actualmente como IDE's formales y aprovechar las ventajas. Para la asignatura se sugiere usar opciones como:

- **PyCharm:** es un Entorno de Desarrollo Integrado (IDE) especializado en Python, desarrollado por [JetBrains](#). Ofrece herramientas avanzadas como autocompletado inteligente, depuración integrada, gestión de entornos virtuales y soporte para frameworks de Python. Su interfaz permite un desarrollo eficiente de aplicaciones de básicas de programación, ciencia de datos, inteligencia artificial y desarrollo web. PyCharm tiene una versión gratuita (Community) y una versión de pago (Professional / Académica) con funcionalidades adicionales para el desarrollo. Se solicita revisar una guía simplificada para [acceder e instalar PyCharm](#) y la puesta en marcha en un computador local.
- **Visual Studio Code:** es un editor de código fuente ligero y multiplataforma, desarrollado por Microsoft. Su flexibilidad permite personalizarlo con extensiones, incluyendo soporte para Python, JavaScript, C++, entre otras tecnologías.

1.11. Gestión de código mediante software de control de versiones.

En el mundo de la programación es necesario tener a la mano herramientas y tecnologías que permitan realizar de manera ágil el proceso de desarrollo. En la actualidad es necesario conocer sobre cómo manejar el versionamiento de código generado mediante un software de control de versiones.

Para empezar esta sección se invita a realizar una lectura comprensiva de libro [Pro Git](#), en lo relacionado al capítulo 1, llamado Inicio - Sobre el Control de Versiones.

El recurso define a un control de versiones como una herramienta que facilita el seguimiento y almacenamiento de las modificaciones realizadas en uno o varios archivos dentro de un proyecto. Permite retroceder a versiones



anteriores cuando sea necesario y proporciona un historial detallado de ediciones, incluyendo información sobre los autores de los cambios y el momento en que fueron realizados.

Además, indica que Git es un sistema de control de versiones creado por Linus Torvalds, diseñado para gestionar cambios en proyectos de software de manera eficiente. Es utilizado en desarrollos de gran escala como el kernel de Linux, Android, y plataformas de empresas como Google, Facebook, Netflix, etc. Su uso es beneficioso tanto para el trabajo individual, permitiendo el seguimiento y control de modificaciones, como en entornos colaborativos, donde facilita la coordinación entre los miembros de un equipo de desarrollo.

A lo largo de la asignatura se va a trabajar con algunos repositorios de código, por ello se solicita lo siguiente:

- Paso 1, instalar GIT en las máquinas personales, seguir los pasos indicados en la [página oficial de instalación](#).
- Paso 2: luego de la instalación debe configurar GIT a través de Git-Bash en Windows; o un terminal en Linux o MacOS.
 - Existen dos comandos necesarios para configurar de forma global Git, estos comandos se lo debe ingresar en Git-Bash
 - `git config --global user.name "Nombre del Usuario"`
 - `git config --global user.email correos-del-usuario@dominio.com`
- Paso 3: familiarizarse con comando básicos de GIT como:
 - `git clone`

Este comando se usa para descargar un proyecto completo desde un servidor remoto a la computadora local. Es como hacer una copia del repositorio en la máquina local para poder trabajar en él. La dirección del repositorio se obtiene generalmente desde plataformas como GitHub o GitLab.

- `git add`



Con este comando, se agrega todos los archivos nuevos o modificados al área de preparación de Git. Es un paso necesario antes de confirmar los cambios en el historial del proyecto. Si solo se necesita agregar un archivo específico, se debe usar `git add nombre_del_archivo`.

- `git commit -a -m "mensaje"`

Aquí es donde se guarda formalmente los cambios en el historial del proyecto dentro de la computadora local. El `-m "mensaje"` permite escribir una descripción breve de los cambios realizados, lo cual es útil para llevar un registro organizado del trabajo.

- `git push`

Después de hacer cambios y guardarlos con `commit`, este comando envía los cambios al repositorio en línea (por ejemplo, en GitHub o GitLab). Es como actualizar la versión del proyecto en la nube para que otros puedan verlo o colaborar con nosotros.

- `git status`

Sirve para verificar qué archivos han sido modificados, cuáles están listos para ser guardados y cuáles aún no han sido agregados al control de versiones.

- `git pull`

Este comando se usa para actualizar el proyecto local con los cambios más recientes que otras personas hayan subido al servidor. Es similar a sincronizar el trabajo con la versión más actualizada del repositorio en línea.

Lo anterior permite tener un instalado en la máquina personal, pero ahora es necesario entender que existen plataformas que ayudan a mantener el código versionado, pero en servidores en Internet. Existen algunas alternativas para ello, pero en la guía se usará GitHub.



[GitHub](#) es una plataforma diseñada para el desarrollo colaborativo de software, permitiendo almacenar y gestionar proyectos a través del sistema de control de versiones Git. Facilita la organización del código, el seguimiento de cambios y la cooperación entre múltiples desarrolladores en un entorno centralizado.

Para el uso de GitHub y su posterior sincronización con los repositorios locales administrados por Git, es necesario realizar lo siguiente:

- Crear una [cuenta personal en GitHub](#)
- Revisar la guía para [crear repositorios en GitHub](#)
- Revisar el proceso de [fork a un repositorio de GitHub](#)
- Revisar el proceso de [sincronización de un proyecto de GitHub con Git](#) en un entorno local.



Actividades de aprendizaje recomendadas

Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. Analizar detenidamente el recurso que explica el proceso de instalación de Python en diversos sistemas operativos, ubicado en el [capítulo 3 del texto “Python paso a paso”](#), con el objetivo de comprender sus conceptos clave. A través de esta actividad, el estudiante podrá identificar las particularidades de la instalación en diferentes entornos, reconocer los requisitos previos y familiarizarse con los comandos necesarios para llevar a cabo una instalación exitosa. Se recomienda considerar las diferencias entre los sistemas operativos y practicar el proceso en su propio equipo para reforzar el aprendizaje.
2. Leer e interpretar el recurso [“INTRODUCCIÓN A GIT Y GITHUB - DÍA 1”](#), identificando los puntos clave sobre el proceso de instalación y los comandos básicos para el manejo de versiones de código. Esta actividad permitirá al estudiante comprender la importancia del control de versiones desde las primeras etapas de su formación, facilitando la



organización y seguimiento del desarrollo de proyectos. Se recomienda tomar notas sobre los comandos esenciales y realizar pruebas en un entorno local para afianzar el uso de Git y GitHub en la gestión de código. Como ejercicio práctico, se recomienda buscar repositorios en GitHub y realizar el proceso de fork y posterior clonación en los entornos locales.

3. Estimado estudiante, ha llegado el momento de evaluar su comprensión sobre los temas de la Unidad 1: Introducción a la programación y solución de problemas. Esta autoevaluación le permitirá identificar qué conceptos comprende de mejor manera y en cuáles debería reforzar su aprendizaje.

Antes de realizar la autoevaluación, se recomienda seguir estas estrategias de estudio:

- Revisar cuidadosamente el contenido de cada temática estudiada.
- Consultar los recursos educativos (documentos, códigos) recomendados para reforzar sus conocimientos.
- Elaborar esquemas o mapas conceptuales para visualizar mejor la relación entre los temas.

Recuerde que cada pregunta tiene una única respuesta correcta. Lea con atención cada enunciado y seleccione la opción que usted crea, y responda la interrogante. ¡Mucho éxito en esta autoevaluación!



Autoevaluación 1

Seleccione la opción correcta.

1. **¿Cuál de los siguientes elementos no es una característica esencial de un algoritmo?**

Respuestas:

- a. Debe ser claro y preciso.
- b. Debe ser ambiguo.



c. Debe ser finito.

2. Autocompletar:

En la fase de planear la lógica de un algoritmo, se recomienda usar un _____ para visualizar la secuencia de pasos de manera clara y estructurada.

3. Autocompletar:

_____ es una estructura que permite describir de manera detallada los pasos de un algoritmo sin depender de un lenguaje de programación específico.

4. ¿Cuál de las siguientes afirmaciones sobre los lenguajes de alto y bajo nivel es correcta?

Respuestas:

- a. Los lenguajes de bajo nivel son más fáciles de entender que los de alto nivel.
- b. Los lenguajes de alto nivel son más cercanos al lenguaje humano, mientras que los de bajo nivel están más cerca del lenguaje máquina.
- c. Los lenguajes de bajo nivel son más eficientes que los de alto nivel en todos los casos.

5. ¿Cuál es una característica clave de los lenguajes de programación interpretados?

Respuestas:

- a. Son más rápidos en tiempo de ejecución que los compilados.
- b. Generan código de máquina optimizado antes de su ejecución.
- c. Se ejecutan directamente mediante un intérprete sin necesidad de compilación previa.

6. Seleccione la opción correcta para la siguiente idea:



Git es una herramienta utilizada para controlar los cambios en el código cuando se trabaja en equipo.

Respuestas:

- a. Verdadero. ()
- b. Falso. ()

7. ¿Qué función cumple un Entorno de Desarrollo Integrado (IDE) en programación?

Respuestas:

- a. Es una herramienta que permite desarrollar y comprobar proyectos en un lenguaje de alto nivel.
- b. Ejecutar programas sin necesidad de código fuente.
- c. Compilar automáticamente todos los programas en ejecutables.

8. En la fase de planear la lógica, un analista desarrollador utiliza una técnica para simular manualmente la ejecución de un algoritmo antes de escribir código. ¿Cuál es esta técnica?

Respuestas:

- a. Depuración de código.
- b. Pruebas de escritorio.
- c. Traducción del código.

9. ¿Cuál de los siguientes lenguajes de programación es de bajo nivel?

Respuestas:

- a. Python.
- b. JavaScript.
- c. Ensamblador.



10. **Un algoritmo está compuesto por tres partes fundamentales. ¿Cuál de las siguientes opciones describe correctamente la función de los procesos dentro de un algoritmo?**

Respuestas:

- a. Son los datos que el usuario ingresa antes de realizar los cálculos.
- b. Representan un posible modelo matemático necesario para obtener las salidas del algoritmo.
- c. Son instrucciones específicas que solo pueden escribirse en un lenguaje de programación.

[Ir al solucionario](#)

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 3

En la Semana 3, los estudiantes iniciarán el estudio de los fundamentos de programación, comprendiendo los elementos básicos necesarios para escribir código en un lenguaje de programación. También aprenderán sobre el manejo de variables y tipos de datos, así como la correcta utilización de operadores y expresiones en distintos contextos. Además, practicarán el uso de expresiones y asignaciones para desarrollar programas simples, aplicando los conceptos estudiados en ejercicios prácticos.

Unidad 2. Fundamentos de programación

2.1. Elementos básicos de la programación

En los primeros temas se generan explicaciones de los conceptos, asociando miniespecificaciones y lenguaje de programación.



La idea es crear una estructura base que se pueda seguir e implementar al momento de generar una respuesta. Al momento de plantear la solución, se solicita en la medida de lo posible crear dos bloques: bloque de declaraciones y bloque de instrucciones.

- Bloque de declaraciones.

En este bloque se deben especificar las variables que luego son necesarias para los procesos y solución. Para la generación de las variables se deben seguir reglas que se conocerán en la siguiente temática.

- Bloque de instrucciones.

Este bloque se constituye por una serie de acciones u operaciones necesarias para obtener los resultados esperados. Se pueden identificar tres tipos principales de operaciones:

- Entrada de datos: Comprende todas aquellas instrucciones que permiten capturar información desde el teclado o archivos.
- Procesamiento de datos: Incluye las instrucciones encargadas de modificar y transformar los datos obtenidos en la etapa de entrada, pasando a la solución deseada.
- Salida de resultados: Consiste en el conjunto de instrucciones que toman los datos procesados y los presentan a través de dispositivos de salida, como la pantalla, archivos, permitiendo que el usuario visualice la información final generada.

2.2. Manejo de variables y tipos de datos

Un concepto importante es la creación de variables. Se presentan dos definiciones que serán de ayuda para comprender el concepto.

Arteaga Martínez, M. M. (2023) lo define como “un ente que identifica a una característica o propiedad específica de un determinado elemento dentro de un problema. Su valor puede variar en el transcurso de la ejecución del algoritmo” (p. 22).



Muñoz Guerrero, L. E. (II.) y Trejos Buriticá, O. I. (2021) señalan que es “un espacio de memoria en donde se puede guardar UN dato. En una variable se puede guardar un número (como 4 o 3,8) o una cadena de caracteres (como “Omar” o “UTP”)” (p. 18).

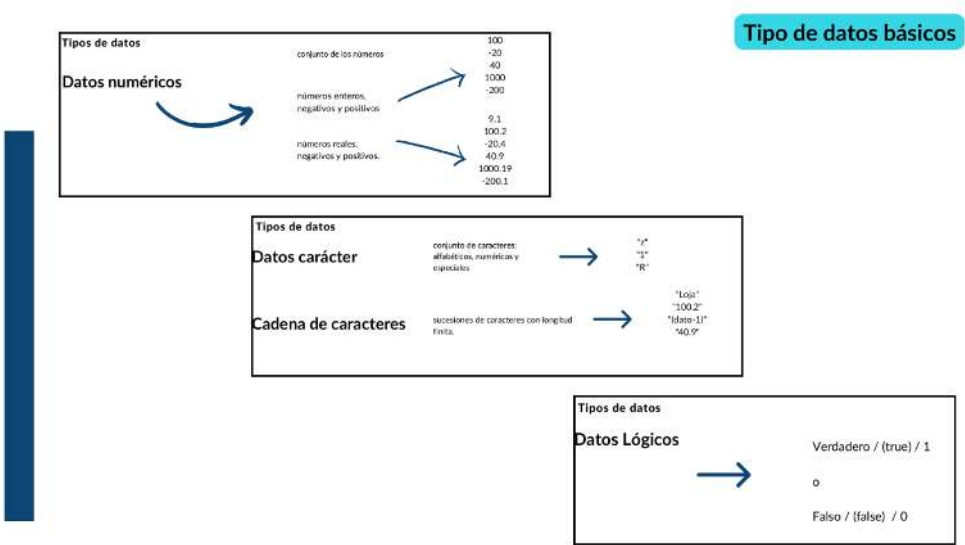
Al referirse al concepto de variable, Herrera, A. M., Gómez, R. E., & Portilla, J. C. (2016) resalta que es un espacio en la memoria compuesto por un conjunto de bytes, identificado mediante un nombre, cuya función es almacenar un valor asociado a un dato específico; además brinda algunas pautas para definir las variables:

- Tener un nombre.
- Tipo de dato (en la guía esto será obligatorio solo para las miniespecificaciones).
- Siempre iniciar con 1 letra.
- Debe contener letras y dígitos.
- No incluir espacios en blanco ni caracteres especiales.
- El nombre tiene relación con la problemática.
- Se la debe inicializar, previo al proceso de usarla.

Con relación a los tipos de datos, en los ejercicios a estudiar se usarán los tipos básicos, que comprenden los tipos de datos numéricos, de cadena y lógicos. En la **Figura 6**, se indican ejemplos de valores según el tipo.



Figura 6
Ejemplo de tipos de datos básicos



Nota. Elizalde, R., 2025.

En la figura anterior se describe algunos ejemplos de valores de tipos de datos básicos.

La nomenclatura de las miniespecificaciones se basa en las estructuras que recomienda Ramírez, F. (2007).

Al referirse a las miniespecificaciones se usará el siguiente contexto para los tipos de datos, es recomendable seguir estos lineamientos, pues, se los aplicará en los ejemplos detallados en las siguientes temáticas:

- l: La letra "l" representa un tipo de dato booleano.
- x(n): Representa un tipo de dato cadena; el valor "n" indica el tamaño que se desea asignar a la cadena.
- i: La letra "i" representa un tipo de dato entero.
- d: La letra "d" representa un tipo de dato decimal.

Además, como un proceso de entendimiento, a cada variable en miniespecificaciones se le asignará un dominio, se insiste que solo será para las miniespecificaciones.

En la representación del dominio de los tipos de datos en las miniespecificaciones, se utilizan diferentes notaciones para establecer restricciones y definir conjuntos de valores permitidos para cada tipo de dato. Se explican algunos casos para comprender su uso:

- Dominio con rango continuo:
 - Se usa el guion - para definir un intervalo.
 - Ejemplo: `d[10-100]` indica un dominio para valores decimales entre 10 y 100.
- Dominio con valores específicos:
 - Se usa la coma, para separar valores puntuales.
 - Ejemplo: `i[19, 20, 21]` define un dominio que solo acepta esos tres valores enteros.
- Dominio con valor máximo indefinido:
 - Se usa `n` para representar el límite superior desconocido.
 - Ejemplo: `d[1000-n]` permite valores desde 1000 hasta el máximo permitido para el tipo de dato.
- Dominio con patrón predecible:
 - Se usa `...` para definir secuencias con una regla clara.
 - Ejemplo: `i[10,12,14,...,30]` representa solo los valores pares entre 10 y 30.
- Dominio con subconjuntos específicos:
 - Se usa `{}` para definir conjuntos de caracteres o valores permitidos.
 - Ejemplo: `x(20)[{a-z}, {A,E,I,O,U}, {BS}]` restringe el uso a letras minúsculas, vocales mayúsculas y espacios en blanco.



- Dominio con valores excluidos:

- Se usa ! para indicar valores prohibidos en el dominio.
- Ejemplo: `i[{10,12,14,...,30}, {!26}]` permite los valores pares entre 10 y 30, excepto el 26.

En Python los tipos de datos básicos con lo que se va a trabajar en la asignatura son:

- int, representar los números enteros
- float, representa números decimales
- str, representa el almacenamiento de caracteres
- bool, tiene dos valores posibles True o False

Es importante en este punto resaltar una particularidad de Python como lenguaje, que se constituye en una característica fundamental; Hinojosa Gutiérrez, Á. (2015) destaca el tipado dinámico así “en Python no es necesario declarar las variables explícitamente antes de usarlas, sino que se crean automáticamente y adoptan un tipo de contenido ... al asignarles un valor por primera vez.” (p. 50)

2.3. Manejo de operadores y expresiones.

En programación, un operador es un símbolo o conjunto de símbolos que indican una operación específica que se debe realizar sobre uno o más valores, llamados operandos. Veamos la siguiente presentación interactiva:

[Operadores y expresiones](#)

Estimado estudiante, se deja el [repositorio de GitHub](#) para que se pueda revisar y ejecuta el código en su entorno local. Existen dos formas de ejecutar el código, a través de la terminal de forma tradicional o copiar el código en el intérprete interactivo de Python o consola de Python.

- Ejemplos de uso de operadores aritméticos





```
a = 10
b = 3
print("Suma:", a + b) # 13
print("Resta:", a - b) # 7
print("Multiplicación:", a * b) # 30
print("División real:", a / b) # 3.3333...
print("División entera:", a // b) # 3
print("Residuo:", a % b) # 1
print("Potenciación:", a ** b) # 1000 (103)
```

- Ejemplos de uso de operadores relacionales

```
a = 5
b = 10

print(a == b) # False (5 no es igual a 10)
print(a < b) # True (5 es menor que 10)
print(a > b) # False (5 no es mayor que 10)
print(a <= b) # True (5 es menor o igual a 10)
print(a >= b) # False (5 no es mayor o igual a 10)
print(a != b) # True (5 es diferente de 10)
```

- Ejemplos de uso de operadores lógicos

```
print("NOT")
print(not True) # False (Negación de True es False)

print("AND")
print(True and True) # True (Ambos valores son True, resultado True)
print(True and False) # False (Uno de los valores es False, resultado False)
print(False and True) # False (Uno de los valores es False, resultado False)
print(False and False) # False (Ambos valores son False, resultado False)

print("OR")
print(True or True) # True (Al menos uno de los valores es True, resultado True)
```

```
print(True or False) # True (Al menos uno de los valores es True, resultado True)
print(False or True) # True (Al menos uno de los valores es True, resultado True)
print(False or False) # False (Ambos valores son False, resultado False)
```

Es momento de revisar las reglas de precedencia, mismas que evalúan las expresiones en función de un orden de prioridad. La Tabla 1, muestra la prioridad y su descripción. Las reglas se las aplica para las miniespecificaciones y lenguaje Python.

Tabla 1
Reglas de Precedencia

Prioridad	Operador	Descripción
1 – Mayor	()	Paréntesis (Comenzando por los más internos)
2	**	Potenciación
3	*, /, //, %	Multiplicación, división, división entera y residuo de la división entera / módulo
4	+, -	Suma y resta
5	<, >, <=, >=, ==, !=	Comparaciones
6	Not	Negación lógica
7	And	Operador lógico AND - Producto lógico
8	Or	Operador lógico OR – Suma lógica

Nota. Elizalde, R., 2025.

Ejemplo 1: Resolver la siguiente expresión: $(3 + 2) * 5 > 20$ and not $(10 \% 3 == 0)$

"""

Ejemplo de uso de reglas de precedencia



La expresión es

$(3 + 2) * 5 > 20$ and not $(10 \% 3 == 0)$

Paso a paso con precedencia aplicada (así se lo debe hacer en algo similar a una miniespecificación)

1. Paréntesis internos primero

$(3 + 2)$ es igual a 5

$(10 \% 3)$ es igual a 1 (El módulo da el residuo de la división $10 \div 3$, que es 1)

$1 == 0$ es igual a False (Comparación relacional: ¿1 es igual a 0? No.)

La expresión queda:

$5 * 5 > 20$ and not False

2. Multiplicación

$5 * 5 \rightarrow 25$

Ahora se tiene:

$25 > 20$ and not False

3. Comparación relacional del símbolo ($>$)

$25 > 20$ es igual a True

La expresión ahora es:

True and not False

4. Negación (not) tiene prioridad sobre and

not False \rightarrow True

Ahora se tiene:

True and True

5. Evaluación de and

True and True es igual a True

6. Resultado final es True



"""

```
# El lenguaje lo hace de forma directa
# Se debe tener la habilidad de armar bien la expresión
#
resultado = (3 + 2) * 5 > 20 and not (10 % 3 == 0)
print("El resultado de la expresión: (3 + 2) * 5 > 20 and not (10 % 3 == 0) :",
      resultado)
# la salida por pantalla es True
```

Estimado estudiante, se deja el [repositorio de GitHub](#) para que se pueda revisar y ejecuta el código en su entorno local. Existen dos formas de ejecutar el código, a través de la terminal de forma tradicional o copiar el código en el intérprete interactivo de Python o consola de Python.

A medida que avanzamos, el siguiente tema le permitirá conectar conceptos importantes y aplicarlos en diferentes contextos

2.4. Uso de expresiones y asignaciones para crear programas en un lenguaje de programación

Luego de revisar los tipos de datos y los diversos operadores, es momento de crear las primeras soluciones. En los ejercicios se trata de presentar procesos a través de miniespecificaciones y lenguaje Python.

Cuando se empieza el proceso de solución, en el bloque de declaración formalmente se debe declarar las variables. La declaración de una variable es el proceso mediante el cual se crea y se define con un tipo de dato dentro de un programa, permitiendo su uso en la solución de un problema. En el desarrollo de miniespecificaciones, es esencial declarar todas las variables antes de utilizarlas en el bloque de instrucciones. Si una variable no ha sido declarada y se intenta emplear, el programa generará un error, ya que no reconocerá su existencia. Por otro lado, la inicialización consiste en asignarle un valor inicial de acuerdo con el tipo de dato a la variable para su posterior manipulación en el código.



Importante resaltar y considerar que, en Python debido a su tipado dinámico, la declaración e inicialización de una variable ocurren al mismo tiempo. A diferencia de otros lenguajes de programación como Java o C, donde es obligatorio declarar una variable con un tipo específico antes de asignarle un valor, en Python es suficiente con asignar un valor para que la variable exista y el lenguaje determine su tipo de dato de forma automáticamente. De igual forma que en las miniespecificaciones si en Python se quiere usar una variable si haberla inicializada, generará un error.

La forma general para declarar e inicializar un variable en miniespecificaciones es la siguiente:

(Nombre variable, Símbolo Tipo de dato [Dominio])

Ejemplo:

(edad, i[1-n])

- edad: es el nombre de la variable
- i: tipo de dato, asociado a la variable, en este caso es un tipo entero.
- [1-n]: dominio de datos permitidos, para el ejemplo, todos los números enteros, mayores a 1.

Luego, se puede realizar la asignación, usando en miniespecificaciones es símbolo "`<--`", del valor con una expresión así:

```
edad <-- 20
```

Aquí, se le asigna el valor entero 20 a la variable edad.

Es posible realizar la declaración y asignación en un mismo paso:

```
(edad, i[1-n]) <-- 20
```

Donde,

- edad es el nombre de la variable.
- i, indica que la variable es un número entero.



- [1-n], dominio de datos permitidos, para el ejemplo, todos los números enteros, mayores a 1.
- <-- 20 ; el operador <-- significa asignación, lo que indica que edad recibirá el valor 20.

La forma para iniciar una variable en Python, es como sigue, usando su tipado dinámico.

```
edad = 20
```

Donde,

- edad, es el nombre de la variable que la almacena en memoria
- =, es el operador que se usa para asignar el valor
- 20, es el dato numérico que se le da a la variable, Python lo interpreta forma automática como un tipo int. Se puede usar la línea de código `print(type(nombre de la variable))`, en un script o la consola de Python para saber el tipo de dato de una variable.

Se observa un proceso más sencillo que la miniespecificación, pero es muy importante que se entienda las dos formas en nuestra formación académica.

Ahora es momento de entender un proceso que se requiere en las miniespecificaciones y los lenguajes de programación para la entrada y salida de los datos. Cuando se inicia en el mundo de la programación en cualquier lenguaje se recomienda estudiar estos dos procesos.

En miniespecificaciones se usará las siguientes formas, se asume lo indicado por Ramirez, F (2007) como estructura base:

- Lectura de datos (Petición): se utiliza para solicitar al usuario ingresar datos por teclado o cualquier dispositivo de entrada y luego sean asignados a las variables correspondiente. Los símbolos que se puede usar son:
 - >>
 - LEER
 - LEA



Ejemplo,

```
>> nombre
```

Se lo interpreta como el proceso de solicitar al usuario que ingrese una cadena de texto que será almacenada, en una variable que se llama nombre, se asume que la variable debe estar al menos declarada.

- Salida de datos (imprimir en pantalla): Se usa para presentar al usuario, los datos que se tengan almacenados en las variables, por lo general, se usan para mostrar (imprimir por pantalla) los datos de los procesos realizados, en el desarrollo de las soluciones. Los símbolos que se puede usar son:

- <<
- ESCRIBIR
- ESCRIBE

Ejemplo:

```
nombre <-- "Ciudad de Loja"  
<< nombre
```

Lo anterior, se lo entiende como el proceso de presentar en pantalla o interfaz, el valor que tenga en ese determinado momento la variable nombre.

En Python, por otro lado, se trabajará así:

- Lectura de datos (Petición), en Python la entrada de datos se realiza con la función `input()`, que permite al usuario ingresar datos desde el teclado. Dicha función siempre devuelve una cadena (str). Luego, si necesita dicho valor con otro tipo de dato, se lo debe transformar (hacer un casting) mediante funciones propias de Python como: `int()`, `float()` principalmente. Además, `input` como función, permite incluir una cadena explicativa para la solicitud de información; este proceso en miniespecificaciones y otros lenguajes, se lo realiza en dos pasos.





Ejemplo:

```
nombre = input("Ingrese su nombre: ") # El usuario ingresa un texto y se  
almacena como str
```

Donde,

- nombre es la variable que se va a crear o actualizar el valor.
 - A través de input, se espera el ingreso por teclado de un valor, mismo que será almacenado en la variable nombre, se recuerda que el tipo de dato, por defecto es str (cadena)
- Salida de datos (imprimir en pantalla), la función print() en Python es la principal vía para mostrar información en la salida la pantalla. Permite imprimir texto, variables y resultados de operaciones.

Ejemplo:

```
print("Hola desde Python") # presenta en pantalla la frase: Hola desde  
Python
```



Los lenguajes de programación tienen sus propias convenciones para poder realizar un formateo de cadenas, que permite presentar información de manera limpia y ordenada.

Entre sus características se tiene:

- Mejora la legibilidad del código generado
- Evita errores al momento de usar diferentes tipos de datos en las salidas
- Mejora de forma eficiente los textos, reduciendo el consumo de memoria.

En lenguaje Python existen algunas técnicas para usar el formateo de cadenas:

- Uso de operador % (porcentaje)

Una forma que la usan lenguajes con C, Java, entre otros. Se puede especificar información de acuerdo a las siguientes reglas:

- %s para reemplazar cadenas
- %d para números enteros
- %f para números decimales

Ejemplo:

```
nombre_ciudad = "Loja"
print("Hola desde %s" % (nombre_ciudad))
# Salida: Hola desde Loja
```

- Uso de la función format

Esta forma está presente desde Python 3, su característica es la flexibilidad en la interpolación de variables.

Ejemplo:

```
nombre_ciudad = "Loja"
print("Hola desde {}".format(nombre_ciudad))
# Salida: Hola desde Loja
```

- Uso de f-string

Forma usada desde la versión de Python 3.6, se considera más fácil y eficiente de trabajar. Se caracteriza porque se antepone la letra f antes de empezar la comillas (simples o dobles), lo que indica que dentro de la cadena se puede usar expresiones entre llaves { } para ser evaluadas y presentadas.

Ejemplo:

```
nombre_ciudad = "Loja"
pais = "Ecuador"
print(f"Hola desde {nombre_ciudad} - {pais}")
# Salida: Hola desde Loja – Ecuador
```



Ejemplo 1: Calcular el costo de un terreno, a partir de sus dimensiones y el costo por metro cuadrado

Solución en miniespecificación:

1. Inicio
2. // Declaración de variables
3. largo_terreno, d[0-n] // Almacena el largo del terreno
4. ancho_terreno, d[0-n] // Almacena el ancho del terreno
5. costo_metro, d[0-n] // Almacena el costo por metro cuadrado
6. area_terreno, d[0-n] // Almacena el área total del terreno
7. costo_total, d[0-n] // Almacena el costo total del terreno
- 8.
9. // Solicitar entrada de datos
10. Escribir "Ingrese el largo del terreno (en metros):"
11. Leer largo_terreno
12. Escribir "Ingrese el ancho del terreno (en metros):"
13. Leer ancho_terreno
14. Escribir "Ingrese el costo por metro cuadrado:"
15. Leer costo_metro
- 16.
17. // Cálculo del área del terreno
18. $area_terreno \leftarrow largo_terreno * ancho_terreno$
- 19.
20. // Cálculo del costo total del terreno
21. $costo_total \leftarrow area_terreno * costo_metro$
- 22.
23. // Mostrar los resultados
24. Escribir "El área del terreno es: " + area_terreno + " metros cuadrados"
25. Escribir "El costo total del terreno es: \$" + costo_total
26. Fin

Solución en lenguaje Python:



Inicio

```
# Declaración de variables (Python no requiere especificar el tipo de dato
# previamente)
# Se capturan los valores mediante entrada de usuario con input()
# input() devuelve un valor de tipo "cadena de texto" (str), por lo que es necesario
# convertirlo al formato que se necesita

largo_terreno = input("Ingrese el largo del terreno (en metros): ") # Captura como
texto
largo_terreno = float(largo_terreno) # Convierte el texto a número decimal

ancho_terreno = input("Ingrese el ancho del terreno (en metros): ") # Captura como
texto
ancho_terreno = float(ancho_terreno) # Convierte el texto a número decimal

costo_metro = input("Ingrese el costo por metro cuadrado: ") # Captura como texto
costo_metro = float(costo_metro) # Convierte el texto a número decimal

# Cálculo del área del terreno
# Se multiplica el largo por el ancho para obtener el área total
area_terreno = largo_terreno * ancho_terreno

# Cálculo del costo total del terreno
# Se multiplica el área obtenida por el costo del metro cuadrado
costo_total = area_terreno * costo_metro

# Mostrar los resultados en pantalla
# Se usa print() con f-strings para formatear la salida con dos decimales (.2f)
print(f"El área del terreno es: {area_terreno:.2f} metros cuadrados")
print(f"El costo total del terreno es: ${costo_total:.2f}")
```

En la Figura 7, se puede observar cómo se ejecutaría este ejemplo de Python, desde un terminal o línea de comandos, bajo la forma o comando:

```
python solucion_python.py
```



Figura 7

Ejemplo básico de Python, con entrada de valores por teclado

Ejemplo básico de Python

```
$ python solucion_python.py
Ingrese el largo del terreno (en metros): 100
Ingrese el ancho del terreno (en metros): 40
Ingrese el costo por metro cuadrado: 20
El área del terreno es: 4000.00 metros cuadrados
El costo total del terreno es: $80000.00
```

Nota. Elizalde, R., 2025.

La figura anterior describe la ejecución de un programa en Python, desde un terminal, es la forma más tradicional.

Estimado estudiante, el ejemplo completo se lo puede descargar del repositorio de Github y realizar los cambios para su estudio en su entorno local.

Ejemplo 2:

Generar una solución que permita calcular y mostrar el valor de la planilla de teléfono de una casa. Se debe ingresar el costo por minutos, el número de minutos consumidos en el mes, la dirección del domicilio, el nombre completo del dueño de la línea telefónica. Finalmente presentar el siguiente reporte

Reporte:

Nombres completos: Luis Alberto Carvajal Ludeña

Dirección: Calle primera entre segunda y décima



Costo por minuto: \$1.5

Número de minutos consumidos: 50

Valor a cancelar: \$75

Solución en miniespecificación:

1. Inicio
2. // Declaración de variables
3. nombre_completo, x(100)[{a-z}, {A-Z}, BS] // Nombre completo del dueño
4. direccion, x(100)[{a-z}, {A-Z}, BS] // Dirección del domicilio
5. costo_por_minuto, d[0-n] // Costo por minuto de llamada
6. minutos_consumidos, i[0-n] // Número de minutos consumidos
7. valor_cancelar, d[0-n] // Costo total a pagar
- 8.
9. // Solicitar entrada de datos
10. Escribir "Ingrese los nombres completos del dueño de la línea telefónica:"
11. Leer nombre_completo
12. Escribir "Ingrese la dirección del domicilio:"
13. Leer direccion
14. Escribir "Ingrese el costo por minuto:"
15. Leer costo_por_minuto
16. Escribir "Ingrese el número de minutos consumidos en el mes:"
17. Leer minutos_consumidos
- 18.
19. // Cálculo del valor a cancelar
20. valor_cancelar <- costo_por_minuto * minutos_consumidos
- 21.
22. // Mostrar el reporte de la planilla
23. Escribir "Reporte:"
24. Escribir "Nombres completos: " + nombre_completo
25. Escribir "Dirección: " + direccion
26. Escribir "Costo por minuto: \$" + costo_por_minuto
27. Escribir "Número de minutos consumidos: " + minutos_consumidos
28. Escribir "Valor a cancelar: \$" + valor_cancelar
29. Fin



Solución en lenguaje Python:

```
# Declaración de variables en Python (no se especifica el tipo de dato
explícitamente)
# Se capturan los valores mediante entrada de usuario
nombre_completo = input("Ingrese los nombres completos del dueño de la línea
telefónica: ")
direccion = input("Ingrese la dirección del domicilio: ")

# Captura de valores numéricos
costo_por_minuto = input("Ingrese el costo por minuto: ")
costo_por_minuto = float(costo_por_minuto) # Conversión a decimal (float) para
permitir valores con decimales

minutos_consumidos = input("Ingrese el número de minutos consumidos en el
mes: ")
minutos_consumidos = int(minutos_consumidos) # Conversión a entero (int), ya
que son minutos enteros

# Cálculo del valor a cancelar
valor_cancelar = costo_por_minuto * minutos_consumidos # Se multiplica el costo
por los minutos consumidos

# Mostrar el reporte de la planilla

# Se utiliza una variable de tipo cadena (`cadena_final`) para acumular toda la
información del reporte.
# Esto permite organizar los datos en una única estructura antes de imprimirlos.
# Se usa `f-strings` para formatear los valores dentro de la cadena de texto.

# Se usan **tres comillas triples (`"""` o `"""`)** para definir la cadena de texto
multilínea de manera legible.
# Esto evita la necesidad de usar múltiples `\n` o concatenaciones con `+`.
# Las cadenas entre comillas triples pueden mantener el formato original con
espacios y saltos de línea.

cadena_final = f"""
Reporte:
Nombres completos: {nombre_completo}
Dirección: {direccion}
Costo por minuto: ${costo_por_minuto:.2f}
Número de minutos consumidos: {minutos_consumidos}
Valor a cancelar: ${valor_cancelar:.2f}
"""
```



```
# Finalmente, se imprime el contenido de `cadena_final`  
print(cadena_final)
```

En la Figura 8, se puede observar cómo sería el proceso de ejecución y visualización del reporte

Figura 8

Salida de un ejemplo básico de Python, con variable str acumuladora

Ejemplo básico de Python con variables str acumuladora

```
$ python ejemplo.py  
Ingrese los nombres completos del dueño de la línea telefónica: Luis Alberto Carvajal Ludeña  
Ingrese la dirección del domicilio: Calle primera entre segunda y décima  
Ingrese el costo por minuto: 1.5  
Ingrese el número de minutos consumidos en el mes: 50  
  
Reporte:  
Nombres completos: Luis Alberto Carvajal Ludeña  
Dirección: Calle primera entre segunda y décima  
Costo por minuto: $1.50  
Número de minutos consumidos: 50  
Valor a cancelar: $75.00
```

Nota. Elizalde, R., 2025.

En la figura anterior se describe la salida por pantalla de un código en Python, haciendo uso de una variable acumuladora.

Estimado estudiante, el ejemplo completo se lo puede descargar del repositorio de Github y realizar los cambios para su estudio en su entorno local.





Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Realizar una lectura detallada del [capítulo cinco del texto “Python paso a paso”](#), enfocándose en los conceptos relacionados con el manejo de variables y datos en Python. Esta actividad le permitirá comprender la declaración, uso y tipos de datos en Python, elementos fundamentales para la construcción de programas. Se recomienda tomar notas sobre los principales tipos de datos, realizar ejemplos prácticos. Además, se sugiere analizar cómo la conversión de tipos influye en la manipulación de la información dentro de un programa. Finalmente, aplicar los conocimientos adquiridos en ejercicios prácticos contribuirá a fortalecer la comprensión y el uso adecuado de las variables en Python para los futuros temas.
2. Estudiar y sintetizar la información presentada en el recurso [“Introducción a la programación con Python 3”, específicamente en el capítulo dos, sección “Los operadores aritméticos”](#), para comprender las reglas de precedencia al evaluar expresiones en Python. A través de esta actividad, el estudiante podrá identificar el orden en que se ejecutan las operaciones matemáticas dentro de una expresión y cómo influye en los resultados obtenidos. Se recomienda elaborar un cuadro resumen con la jerarquía de operadores, realizar ejercicios prácticos aplicando diferentes combinaciones de operaciones y comparar los resultados con los obtenidos manualmente. Asimismo, puede practicar el uso de paréntesis para modificar la precedencia en expresiones complejas y reflexionar sobre su impacto en la lógica de los programas.
3. Es importante reforzar los conceptos de manejo de variables, ingreso de datos y formateo de cadena en Python; para ello, se solicita revisar el [ejemplo desarrollado](#). Luego, debe tener su entorno de programación listo para resolver los ejercicios dados en el repositorio de GitHub.



Recuerde las buenas prácticas de programación usadas hasta el momento.

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 4

En la Semana 4, se continúa con el estudio de los fundamentos de programación, enfocándose en las estructuras de decisión, esenciales para controlar el flujo de un programa. Aprenderán a utilizar la estructura de decisión simple (if), la estructura de decisión doble (if/else) y la estructura de decisión múltiple, permitiendo que los programas tomen decisiones según distintas condiciones. Además, realizarán ejercicios prácticos para aplicar estos conceptos en la resolución de problemas computacionales.

Unidad 2. Fundamentos de programación

2.5. Estructuras de decisión simple – if

Es momento de empezar el estudio de un conjunto de estructuras que brindan las bases en el proceso de solución de problemas computacionales. Se recomienda realizar una lectura comprensiva del texto [Fundamentos iniciales de lógica de programación 1. Algoritmos en PseInt y Python](#), en lo relacionado con la unidad de estructuras de decisión y selección.

En el texto se indica que las estructuras de decisión y selección son utilizadas en problemáticas que necesitan evaluar ciertas condiciones o decidir entre múltiples opciones para alcanzar una solución más adecuada. En el transcurso de su estudio, es posible que se las pueda nombrar como estructuras de control selectivas o alternativas.



Cuando se construye una solución a través de estructuras de decisión, se debe analizar de forma detallada el número de posibles condiciones para poder escoger el tipo de condicional que se va a usar. Los tipos de estructuras de selección que existen y se usarán en las temáticas siguientes son: simples, compuestas y múltiples.

En este apartado, se revisan las condicionales simples, estructura que permite ejecutar un conjunto de instrucciones/miniespecificaciones/líneas de código siempre que la condición establecida sea verdadera. En este tipo de estructura, solo se indica el bloque o conjunto de código que debe ejecutarse cuando la expresión lógica de la condición se cumple.

En miniespecificación o pseudocódigo se pueden usar las siguientes directrices para el uso de un condicional simple, con las palabras reservadas: Sí, entonces, FinSi.

Si expresión-lógica entonces

 Instrucciones a ejecutar si la expresión-lógica es True (verdadera)

Fin Si

En lenguaje Python, se usa la palabra reservada `if`, con la siguiente estructura:

`if` expresión-lógica:

 // instrucciones

 // instrucciones

Al finalizar el encabezado del condicional, luego de la condición, se deben escribir de forma obligatoria dos puntos (:), que significa que está iniciando la estructura.

Importante resaltar en los lenguajes Python la indentación del código en función de quién lo contiene y que permite definir la estructura y el flujo de información de cada solución; en otros lenguajes de programación se usan llaves `{}` o palabras como `begin/end`. En Python, la indentación es obligatoria para que el código sea apto para el proceso de interpretación. Para hacer uso de la indentación se puede trabajar con espacios o tabulaciones (es un



estándar usar 4 espacios o un tabulador). Se considera fundamental trabajar con indentación en condicionales, ciclos repetitivos, funciones, clases, entre otras estructuras de Python.

Ejemplo 1:

- Se desea generar una solución que permita determinar si la edad de una persona es mayor de edad en Ecuador, considerando que las personas que son mayores a 18 años y son las que cumplen con la condición. La edad será ingresada por teclado.

Para la solución de la problemática, siempre se sugiere realizar el siguiente proceso: crear el algoritmo, luego la miniespecificación y finalmente la implementación en lenguaje de alto nivel, en este caso Python.

Solución en miniespecificación:

1. Inicio
2. // se declara la variable entera
3. edad, i[0-n]
4. // se presenta un mensaje en pantalla
5. Escribir "Ingrese la edad de la persona:"
6. // se lee por teclado el valor que será asumido por edad
7. Leer edad
8. // Se inicia el condicional
9. Si edad \geq 18 entonces
10. Escribir "La persona es mayor de edad en Ecuador."
11. Fin Si
12. Fin

Solución en lenguaje Python

```
# variable edad, asume el valor ingresado por teclado
edad = input("Ingrese la edad de la persona: ")
```

```
# la variable edad, asume un valor de tipo cadena
# por defecto con la función input
```



```
# Se debe hacer un cambio de tipo de dato para que
# sea considerado con entero, a través de int

edad = int(edad)

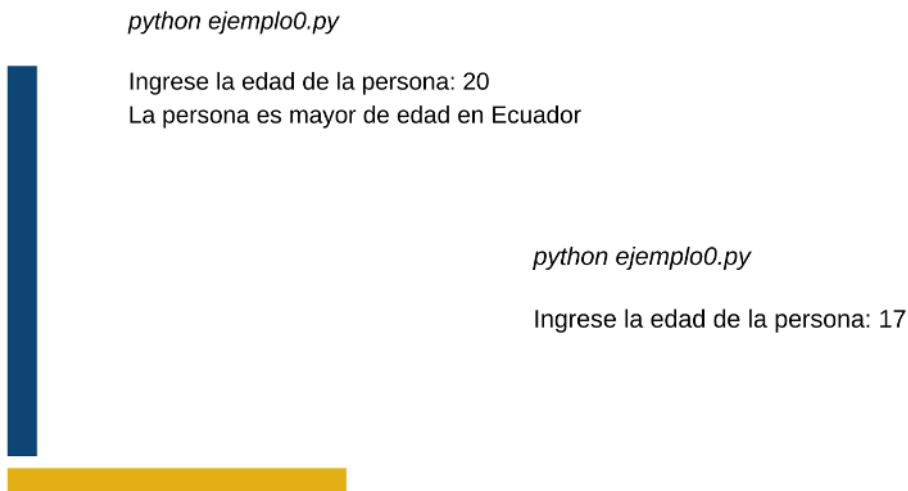
# desde aquí la variable edad es considerada con tipo d
# de dato entero

if edad >= 18:
    print("La persona es mayor de edad en Ecuador")
```

En la Figura 9, se puede observar la salida que se genera mediante el lenguaje Python. Cuando el usuario ingresa por teclado un valor mayor o igual a 18, se presenta el mensaje que está en el cuerpo del condicional. Si el usuario ejecuta el script e ingresa un valor menor a 18, no se presenta ningún mensaje, pues, no existen líneas de código para tal efecto.

Figura 9

Ejemplo de ejecución del script / archivo de Python usando condicionales simples



Nota. Elizalde, R., 2025.

En la figura anterior se describe la forma de ejecutar el código en Python, que hace uso de condicionales simples.



Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.

2.6. Estructuras de decisión doble - if/else

Otro tipo de condicional son las estructuras de decisión doble o compuesta. Aquí la ejecución del código varía según el resultado de la expresión lógica. Si la condición se cumple, se ejecutan únicamente las instrucciones definidas para el caso verdadero. En cambio, si la condición no se cumple, solo se ejecutan las estructuras de control diseñadas para la alternativa falsa.

En miniespecificación o pseudocódigo se puede usar las siguientes directrices para el uso de una estructura de decisión compuesta, con las palabras reservadas: Si, entonces, De lo contrario, FinSi

Si expresión-lógica entonces

 Instrucciones a ejecutar si la expresión-lógica es True (verdadera)

De lo contrario

 Instrucciones a ejecutar si la expresión lógica es False (falsa)

Fin Si

En lenguaje Python, se usa las palabras reservadas if / else, con la siguiente estructura:

if expresión-lógica:

 // instrucciones

 // instrucciones

else:

 // instrucciones

 // instrucciones



Cómo se puede observar, en Python, el condicional compuesto agrega la otra palabra reservada else, que es el indicativo de inicio para el conjunto de instrucciones, en caso de que la condición de if, se falso. Es importante mencionar que luego de la palabra reservada else, es obligatorio el uso de los dos puntos (:)

Ejemplo 1:

- Se desea generar una solución que permite determinar si la edad de una persona es mayor de edad en Ecuador, considerando que las personas que son mayores a 18 años son las que cumplen con la condición. La edad será ingresada por teclado. Si la persona es mayor de edad se debe presenta un mensaje que diga "Usted es mayor de edad en Ecuador", caso contrario se presenta un mensaje como "Usted es menor de edad en Ecuador"

Solución en miniespecificación:

1. Inicio
2. // Declaración de la variable para almacenar la edad
3. edad, i[0-n]
- 4.
5. // Solicitar al usuario que ingrese la edad
6. Escribir "Ingrese la edad de la persona:"
7. Leer edad
- 8.
9. // Verificar si la persona es mayor de edad
10. Si edad \geq 18 entonces
11. // Si la condición es verdadera, la persona es mayor de edad
12. Escribir "Usted es mayor de edad en Ecuador."
13. De lo contrario
14. // Si la condición es falsa, la persona es menor de edad
15. Escribir "Usted es menor de edad en Ecuador."
16. Fin Si
- 17.
18. // Fin del programa
19. Fin

Solución en lenguaje Python:



```
# variable edad, asume el valor ingresado por teclado
edad = input("Ingrese la edad de la persona: ")
```

```
# la variable edad, asume un valor de tipo cadena
# por defecto con la función input
# Se debe hacer un cambio de tipo de dato para que
# sea considerado con entero, a través de int
```

```
edad = int(edad)
```

```
# desde aquí la variable edad es considerada con tipo
# de dato entero
```

```
# Se realiza la comparación en el condicional
# en función del valor ingresado por teclado
```

```
if edad >= 18:
```

```
    # si la condición verdadera, se ejecutan las líneas de
    # código que pertenece al if
    print("Usted es mayor de edad en Ecuador.")
```

```
else:
```

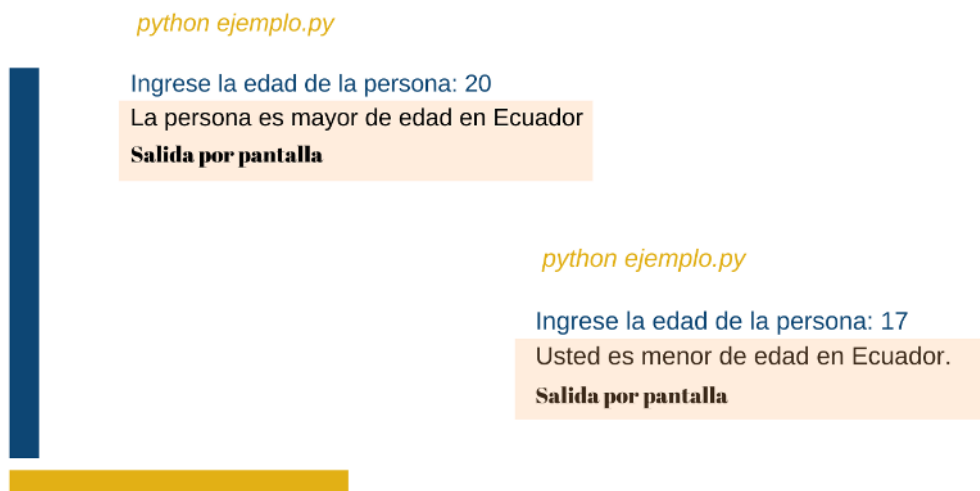
```
    # si la condición es falsa en el if, se ejecutan las líneas
    # de código que pertenecen al else
    print("Usted es menor de edad en Ecuador.")
```

En la Figura 10 se observa el proceso de ejecución y las salidas correspondientes, en función del valor que sea ingresado por teclado.



Figura 10

Uso de Python con estructuras de decisión compuestas



Nota. Elizalde, R., 2025.

En la figura anterior se describen los pasos y la salida de código Python, que hace uso de condicionales compuestos.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.

2.7. Estructuras de decisión múltiple

El tercer tipo de estructura de decisión que se puede usar en la solución de problemáticas es la múltiple. El recurso recomendado, indica que esta estructura es utilizada cuando una miniespecificación o solución en lenguaje de programación de alto nivel, necesita ejecutar más de dos acciones basadas en el resultado de una condición evaluada. La sintaxis se construye a partir de la estructura usadas en las estructuras de decisión simples y compuestas; se



inicia con una condición inicial, seguida de múltiples instrucciones De lo contrario / Si para cada alternativa posible, salvo la última, que en muchos casos puede omitirse porque su resultado es evidente.

Es importante considerar que el número de condiciones Si en una estructura de decisión debe coincidir con el número de FinSi. Además, cada bloque de código debe estar correctamente indentado para mantener una estructura ordenada y clara, asegurando la correspondencia entre Si, Caso Contrario y FinSi.

La estructura en miniespecificación es:

Si expresión-lógica entonces

 Instrucciones a ejecutar si la expresión-lógica es True (verdadera)

De lo contrario

 Si expresión-lógica entonces

 Instrucciones a ejecutar

 De lo contrario

 Instrucciones a ejecutar

 Fin Si

Fin Si

La estructura en lenguaje Python:

```
if expresión-lógica:
    // instrucciones
    // instrucciones
else:
    if expresión-lógica
        // instrucciones
        // instrucciones
    else:
        // instrucciones
        // instrucciones
```

Como ya se indicado, es importante mantener la indentación en cada una de las líneas de código que se van a generar con esta estructura, principalmente en lenguaje Python



Ejemplo 1:

- Genere una miniespecificación que pida por teclado la calificación de un estudiante. Con base en ella, indique la calificación cualitativa que tendría la calificación ingresada. Para obtener la calificación cualitativa, debe usar las siguientes reglas:
 - Mayor o igual a 0 y menor o igual a 2 equivale a Muy mala
 - Mayor a 2 y menor o igual a 3 equivale a Mala
 - Mayor a 3 y menor o igual a 5 equivale a Deficiente
 - Mayor a 5 y menor o igual a 7 equivale a Buena
 - Mayor a 7 y menor o igual a 8 equivale a Buena
 - Mayor a 8 y menor o igual a 9 equivale a Muy buena
 - Mayor a 9 y menor o igual a 10 equivale a Excelente

Solución en miniespecificación

1. Inicio
2. // Declaración de variables
3. calificacion, d[0-10]
4. resultado, x(20)[{a-z}, {A-Z}, {BS}]
- 5.
6. // Solicitar al usuario que ingrese la calificación
7. Escribir "Ingrese la calificación del estudiante:"
8. Leer calificacion
- 9.
10. // Determinar la calificación cualitativa usando condicionales múltiples
11. Si calificacion ≥ 0 y calificacion ≤ 2 entonces
12. resultado \leftarrow "Muy mala"
13. De lo contrario
14. Si calificacion > 2 y calificacion ≤ 3 entonces
15. resultado \leftarrow "Mala"
16. De lo contrario
17. Si calificacion > 3 y calificacion ≤ 5 entonces
18. resultado \leftarrow "Deficiente"



```

19. De lo contrario
20. Si calificacion > 5 y calificacion <= 7 entonces
21.     resultado <-- "Buena"
22. De lo contrario
23. Si calificacion > 7 y calificacion <= 8 entonces
24.     resultado <-- "Buena"
25. De lo contrario
26. Si calificacion > 8 y calificacion <= 9 entonces
27.     resultado <-- "Muy buena"
28. De lo contrario
29. Si calificacion > 9 y calificacion <= 10 entonces
30.     resultado <-- "Excelente"
31. Fin Si
32. Fin Si
33. Fin Si
34. Fin Si
35. Fin Si
36. Fin Si
37. Fin Si
38.
39. // Mostrar el resultado al usuario
40. Escribir "La calificación cualitativa es: " + resultado
41. Fin

```

Solución en lenguaje Python

```

# Declaración de variables
# Variable para almacenar la calificación cualitativa
resultado = "" # variable de tipo cadena

# Variable para almacenar la calificación del estudiante
# Solicitar al usuario que ingrese la calificación
calificacion = input("Ingrese la calificación del estudiante: ")
# Para asignar el valor a calificación, se pide el valor por teclado
# a través de input, luego se hace la transformación del valor de tipo
# cadena a decimal, a través de la función float
calificacion = float(calificacion)

```



```

# Determinar la calificación cualitativa usando solo if / else
if calificacion >= 0 and calificacion <= 2:
    resultado = "Muy mala"
else:
    if calificacion > 2 and calificacion <= 3:
        resultado = "Mala"
    else:
        if calificacion > 3 and calificacion <= 5:
            resultado = "Deficiente"
        else:
            if calificacion > 5 and calificacion <= 7:
                resultado = "Buena"
            else:
                if calificacion > 7 and calificacion <= 8:
                    resultado = "Buena"
                else:
                    if calificacion > 8 and calificacion <= 9:
                        resultado = "Muy buena"
                    else:
                        if calificacion > 9 and calificacion <= 10:
                            resultado = "Excelente"
                        else:
                            resultado = "Calificación fuera de rango"

# Mostrar el resultado al usuario
# opción 1
print("La calificación cualitativa es %s" % (resultado))

# opción 2
# print(f"La calificación cualitativa es {resultado}")

```

La ejecución del script de Python del ejemplo anterior, lo puede revisar en la Figura **¡Error! No se encuentra el origen de la referencia..**



Figura 11

Uso de la estructura de decisión múltiple a través de Python.

```
(envpy310-ia-2) $ python ejemplo.py
Ingrese la calificación del estudiante: 10
La calificación cualitativa es Excelente
(envpy310-ia-2) $ python ejemplo.py
Ingrese la calificación del estudiante: 4
La calificación cualitativa es Deficiente
(envpy310-ia-2) $ python ejemplo.py
Ingrese la calificación del estudiante: 11
La calificación cualitativa es Calificación fuera de rango
(envpy310-ia-2) $ python ejemplo.py
Ingrese la calificación del estudiante: -1
La calificación cualitativa es Calificación fuera de rango
```

Nota. Elizalde, R., 2025.

En la figura anterior se hace referencia a la ejecución de un script de Python en varias oportunidades, ingresando valores diferentes para obtener salidas diversas. El código en Python, hace uso de la estructura de decisión múltiple.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Realizar el taller de repaso de la [lección cuatro sobre “Sentencias condicionales” del texto “Introducción a la Programación con Python”](#) para fortalecer conocimientos sobre estructuras de decisión. Esta actividad permitirá al estudiante reconocer la importancia de las condiciones y la toma de decisiones en la programación, facilitando la construcción de programas más dinámicos y adaptables. Además, es



importante comparar y relacionar los ejemplos estudiados en la unidad y se sugiere modificar los ejercicios propuestos para observar cómo pequeños cambios en las condiciones pueden alterar el flujo del programa y su comportamiento.

2. Estudiar y sintetizar la información presentada en el [capítulo cuatro del texto “Introducción a la Programación con Python 3”](#), subsección centrada en las estructuras de control, con el propósito de reforzar los conceptos sobre estructuras de decisión. Esta actividad permitirá al estudiante comprender la lógica detrás de las condiciones if y if-else, fundamentales para la toma de decisiones en los programas. Se recomienda revisar detenidamente el **contenido del recurso y replicar los ejemplos detallados**, analizando su funcionamiento y modificándolos para observar cómo los cambios en las condiciones afectan la ejecución del programa. Además, se sugiere diseñar ejercicios propios adicionales que involucren estructuras de decisión.
3. Estimado estudiante, ha llegado el momento de evaluar su comprensión sobre los temas abordados en la Unidad 2: Fundamentos de Programación, los cuales incluyen elementos básicos de la programación, manejo de variables y tipos de datos, operadores y expresiones, estructuras de decisión y asignaciones en miniespecificaciones y lenguaje de programación Python. Esta autoevaluación le permitirá identificar qué conceptos domina y en cuáles debería reforzar su aprendizaje.

Antes de realizar la autoevaluación, se recomienda seguir estas estrategias de estudio:

- Consultar los recursos educativos (documentos, ejemplos de código y bibliografía recomendada) para reforzar sus conocimientos. Revisar cuidadosamente el contenido de cada tema, prestando especial atención a la estructura de las expresiones y asignaciones. Practicar con ejercicios sobre operadores aritméticos, relacionales y lógicos, comprendiendo su funcionamiento y precedencia en las expresiones. Analizar la implementación de estructuras de decisión (if, if-else,



estructuras múltiples), identificando su aplicación en problemas computacionales.

Familiarizarse con el uso de variables y tipos de datos en Python, reconociendo la importancia del tipado dinámico y la asignación de valores.

Recuerde que cada pregunta tiene una única respuesta correcta. Lea con atención cada enunciado y seleccione la opción que usted considere adecuada. ¡Mucho éxito en esta autoevaluación!



Autoevaluación 2

1. ¿Cuál es la función del bloque de declaraciones en la estructura de un programa?

Respuestas:

- a. Escribir las instrucciones principales del programa.
- b. Ejecutar las operaciones matemáticas del programa.
- c. Definir variables y sus características antes de su uso.

2. Autocompletar:

En programación, una variable es un _____ que almacena un valor y puede cambiar durante la ejecución del programa.

3. En Python, ¿cómo se declara e inicializa una variable de tipo entero correctamente?

Respuestas:

- a. `int edad = 25.`
- b. `edad = 25.`
- c. `edad <- 25.`

4. ¿Cuál de las siguientes opciones no es una regla para la creación de variables?



Respuestas:

- a. Debe comenzar con una letra.
- b. No puede contener caracteres especiales.
- c. Puede contener espacios en blanco.

5. En el lenguaje de programación Python, las variables adoptan automáticamente el tipo de dato según su contenido.

Respuestas:

- a. Verdadero. ()
- b. Falso. ()

6. El resultado que se presentará en pantalla de la siguiente línea de código en Python `print(10 % 3)`, es: 0.

Respuestas:

- a. Verdadero. ()
- b. Falso. ()

7. ¿Cuál es la salida del siguiente código en Python si el usuario ingresa 3?

```
num = int(input("Ingrese un número: "))  
  
if num % 2 == 0 or num > 10:  
    print("Cumple una de las condiciones")  
  
else:  
    print("No cumple ninguna condición")
```

Respuestas:

- a. No cumple ninguna condición.
- b. Cumple una de las condiciones.



c. Error de sintaxis.

8. **¿Cuál es la salida del siguiente código en Python si el usuario ingresa 8?**

```
nota = int(input("Ingrese su nota: "))
```

```
if nota >= 7 and nota <= 10:
```

```
    print("Aprobado")
```

```
else:
```

```
    print("Reprobado")
```

Respuestas:

a. Aprobado/Reprobado.

b. Reprobado.

c. Aprobado.

9. **¿Cuál será la salida si el usuario ingresa 85?**

```
puntuacion = int(input("Ingrese la puntuación del jugador: "))
```

```
if puntuacion < 50:
```

```
    print("Nivel principiante")
```

```
else:
```

```
    if puntuacion >= 50 and puntuacion < 80:
```

```
        print("Nivel intermedio")
```

```
    else:
```

```
        if puntuacion >= 80 and puntuacion <= 100:
```

```
            print("Nivel avanzado")
```



```
else:
```

```
    print("Puntuación fuera de rango")
```

Respuestas:

- a. Puntuación fuera de rango.
- b. Nivel avanzado.
- c. Nivel principiante.

10. **¿Cuál será la salida si el usuario ingresa 15?**

```
edad = int(input("Ingrese su edad: "))
```

```
if edad >= 18 and edad <= 30:
```

```
    print("Joven adulto")
```

```
else:
```

```
    if edad > 30 and edad <= 50:
```

```
        print("Adulto")
```

```
    else:
```

```
        if edad > 50:
```

```
            print("Adulto mayor")
```

```
        else:
```

```
            print("Menor de edad")
```

Respuestas:

- a. Joven adulto.
- b. Menor de edad.
- c. Adulto mayor.



Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 5

En la Semana 5, se avanzará en el estudio de los fundamentos de programación, enfocándose en las estructuras de repetición. Aprenderán a utilizar el ciclo while, usado principalmente para ejecutar instrucciones mientras se cumpla una condición, y el ciclo for, útil para iteraciones controladas sobre secuencias de datos. Además, realizarán ejercicios prácticos que les permitirán aplicar estos conceptos en la automatización de tareas repetitivas dentro de un programa.

Unidad 2. Fundamentos de programación

2.8. Ciclo repetitivo while

Continuando con el estudio de las bases de programación, el siguiente punto a considerar es entender el uso de ciclos repetitivos o bucles. Para ello se solicita la lectura de la sección 16, denominada Bucles, del texto [Python práctico: Herramientas, conceptos y técnicas](#).

El texto indica que, en el ámbito de la programación, los bucles son estructuras fundamentales que permiten la ejecución repetitiva de un conjunto de instrucciones. Cada vez que se repite este conjunto de instrucciones, se denomina iteración. Gracias a los bucles, es posible automatizar procesos que de otro modo requerirían una gran cantidad de código repetitivo, haciendo que los algoritmos sean más eficientes y dinámicos.

Existen distintos tipos de bucles, cada uno diseñado para ser utilizado en contextos específicos. La elección de un tipo de bucle depende de la condición de parada, es decir, la forma en que se determina cuándo el bucle debe



detenerse. En algunos casos, el número de iteraciones está predefinido, mientras que en otros la repetición continúa hasta que se cumpla cierta condición lógica.

La condición de parada es el criterio que determina cuándo un bucle debe detenerse. Es un elemento fundamental en cualquier estructura repetitiva, ya que, sin una condición de parada adecuada, un bucle podría ejecutarse indefinidamente, generando lo que se conoce como un bucle infinito.

Independientemente del tipo de bucle que se utilice, todos comparten ciertos elementos esenciales:

- Punto de inicio: Indica el momento en que comienza la ejecución del bucle.
- Punto de fin: Define la condición bajo la cual el bucle dejará de ejecutarse.
- Número de iteraciones: Puede ser fijo o depender de una condición lógica.
- Bloque de instrucciones a ejecutar: Contiene las acciones que se repetirán en cada iteración.

En esta sección, se estudiarán los diferentes tipos de bucles y cómo utilizarlos de manera eficiente en la programación, asegurando la correcta aplicación según las necesidades de cada problema.

Antes de conocer los conceptos relacionados con los dos tipos de bucles que se comprenderán, se debe conocer y entender conceptos como: contadores y acumuladores.

Una variable de tipo contador se emplea para registrar la cantidad de veces que ocurre un determinado evento dentro de un programa. Generalmente, el conteo comienza en cero y se incrementa de manera secuencial, generalmente de uno en uno. Sin embargo, el valor del incremento puede variar según las necesidades del problema, permitiendo aumentos de dos, tres o más unidades en cada iteración. Para utilizar correctamente un contador, primero se debe inicializar y luego incrementar; ejemplo.




```
contador <-- 0 // Inicialización del contador, con valor cero
// Incremento de 1 en 1, al contador se le asigna el valor de contador más uno
contador <-- contador + 1
contador <-- contador + 1
```

Por otro lado, se tienen los acumuladores, que se considera una variable diseñada para almacenar el resultado de la acumulación progresiva de valores mediante una misma operación. Se utiliza con frecuencia en cálculos como sumas o productos. Al igual que los contadores, es necesario inicializarlo antes de comenzar a acumular valores, aplicando posteriormente la operación correspondiente en cada iteración. Ejemplo:

```
suma <-- 0
suma <-- suma + 10
suma <-- suma + 20
suma <-- suma + 2
suma <-- suma + 20
```

Es momento de analizar el ciclo repetitivo while (mientras); la lógica de este ciclo permite la realización de un conjunto de sentencias de código, las cuales podrán ser ejecutadas cuando la condición de la expresión lógica del while sea verdadera.

El recurso recomendado indica que el bucle while es ideal para situaciones en las que no se conoce con precisión la cantidad de iteraciones a ejecutar, pero sí se tiene claro que la repetición debe continuar hasta que una determinada condición deje de cumplirse. Además, el ciclo while, se lo considera un ciclo de comparación al inicio, es decir, que, si la condición inicial de comparación es falsa, el ciclo no realizará ninguna iteración.

La forma general del ciclo en miniespecificación es:

Mientras expresión-lógica Entonces

 Instrucciones a ejecutar si la expresión-lógica es True (verdadera)

Fin Mientras



// Donde las palabras o expresiones reservadas que se usan en la representación del

// bucle de comparación al inicio son: Mientras, Entonces, Fin Mientras

La forma general del ciclo en Python es:

while condición-lógica:

 // instrucción

 // instrucción

 // instrucción

 // instrucción

En Python, la palabra reservada while es la que inicia el bucle, indicando que el bloque de código dentro de él se ejecutará mientras la condición lógica sea verdadera. Los dos puntos (:) son obligatorios al final de la línea donde se define el while, ya que señalan el inicio del bloque de instrucciones. Además, la indentación es fundamental en Python, ya que delimita el cuerpo del bucle; todas las instrucciones dentro del while deben estar correctamente indentadas.

Ejemplo 1: Genere una solución que permita presentar la suma y el promedio de los números del 1 al 10.

Solución en miniespecificación.

1. Inicio

2. // Declaración de variables

3. numero, i[1-10] <-- 1 // Contador: controla el número hasta donde debe ejecutarse el bucle

4. suma, i[0-n] <-- 0 // Acumulador: almacena la suma total de los números

5. promedio, d[0-n] // Variable para calcular el promedio

6.

7. // Proceso para calcular la suma usando un ciclo Mientras

8. Mientras numero <= 10 entonces // El bucle se ejecuta mientras número sea menor o igual a 10



```

9. suma <- suma + numero    // Se acumula el valor actual en la variable
    suma, con el valor de número
10. numero <- numero + 1    // Contador: incrementa en 1 para avanzar al
    siguiente número
11. Fin Mientras            // Termina el bucle cuando número > 10
12.
13. // Calcular el promedio
14. promedio <- suma / 10    // Se divide la suma total entre 10
15.
16. // Presentar los resultados
17. Escribir "La suma de los números del 1 al 10 es: " + suma
18. Escribir "El promedio de los números del 1 al 10 es: " + promedio
19. Fin

```

Solución en lenguaje Python

```

# Declaración de variables
numero = 1 # Contador: controla el número hasta donde debe ejecutarse el bucle
suma = 0   # Acumulador: almacena la suma total de los números
promedio = 0 # Variable para calcular el promedio

# Proceso para calcular la suma usando un ciclo while
while numero <= 10: # El bucle se ejecuta mientras número sea menor o igual a 10
    suma = suma + numero # Se acumula el valor actual en la variable suma
    numero = numero + 1 # Se incrementa el contador en 1 para avanzar al
    siguiente número

# Calcular el promedio
promedio = suma / 10 # Se divide la suma total entre 10

# Presentar los resultados
print("La suma de los números del 1 al 10 es: %d" % suma)
print("El promedio de los números del 1 al 10 es: %.2f" % promedio)

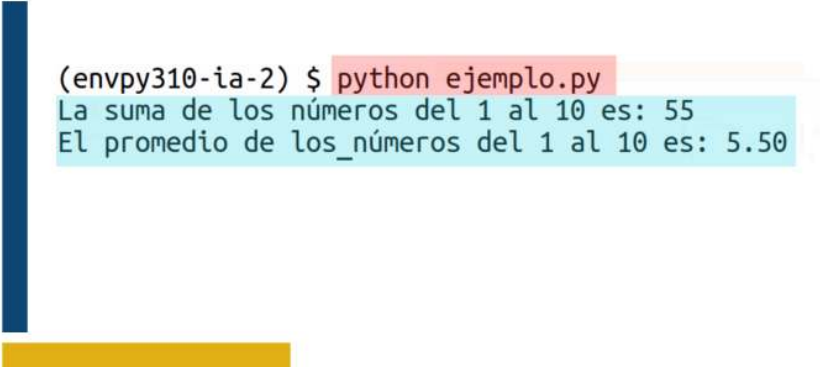
```

La ejecución del ejemplo en lenguaje Python se puede ver en la Figura 12.



Figura 12

Ciclo while en Python, usando contador y acumulador

A terminal window with a dark blue vertical bar on the left and a yellow horizontal bar at the bottom. The terminal text is as follows:

```
(envpy310-ia-2) $ python ejemplo.py
La suma de los números del 1 al 10 es: 55
El promedio de los números del 1 al 10 es: 5.50
```

The first line of the command prompt is highlighted in red, and the two lines of output are highlighted in light blue.

Nota. Elizalde, R., 2025.

La figura anterior muestra la salida de un ejemplo hecho en Python, que hace uso del ciclo repetitivo while, además de contadores y acumuladores

Estimado estudiante, el ejemplo completo se puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.

2.9. Ciclo repetitivo for

Es momento de revisar y estudiar otro ciclo repetitivo. La estructura for (Para en miniespecificación), permite, al igual que el ciclo while, la ejecución repetitiva de un conjunto de instrucciones dentro del cuerpo del ciclo. En este tipo de bucle, se define un valor inicial, una condición y un incremento, los cuales determinan su funcionamiento. A diferencia de otros ciclos, el Para tiene la particularidad de que estos tres elementos se establecen directamente en la cabecera del bucle, facilitando su control y lectura.



Es importante destacar que la comprensión del ciclo Para en miniespecificación se presenta de manera más estructurada y explícita, mientras que su implementación en Python presenta ciertas particularidades y diferencias que requieren una adaptación en la forma de escribir y entender la estructura del bucle.



Según lo indicado en el recurso recomendado, explica que el bucle for es ideal para situaciones en las que se conoce de antemano la cantidad exacta de repeticiones que se deben realizar. Su propósito es ejecutar un conjunto de instrucciones de manera iterativa hasta alcanzar el número establecido de ejecuciones. En Python, los bucles for operan sobre elementos iterables, como listas, tuplas, cadenas de texto o diccionarios. La cantidad de veces que se ejecutará el ciclo dependerá del número de elementos presentes en el iterable. Indica que, los elementos iterables, serán estudiados más adelante.

La forma general del ciclo Para en miniespecificación es:

// Inicio del bucle Para

Para i = 1 hasta n con incrementos j

// Se inicializa la variable i con un valor inicial (en este caso, 1)

// El bucle se ejecuta mientras i sea menor o igual a n;

// n, puede reemplazada por la condición

// En cada iteración, i se incrementa en j unidades

Instrucciones a ejecutar hasta que i sea menor o igual a n

Fin_Para // Fin del bucle, cuando i supera el valor de n o su condición lógica

En la expresión <Para i = 1 hasta n con incrementos j> ; se quiere resaltar lo siguiente:

- Define el valor inicial de i (generalmente 1); pero puede ser el valor que se considere de acuerdo a las problemáticas.
- Establece el límite (n), hasta el cual se ejecutará el bucle; se puede reemplazar por una condición lógica, ejemplo, $10 < \text{limite}$.
- Determina el incremento (j), es decir, cuánto se suma o resta a i en cada iteración. Puede ser incremento o decremento en la estructura:
 - 1; significa de 1 en 1
 - 2; significa de 2 en 2
 - -1; significa de -1 en -1
 - -2; significa de -2 en -2

La forma general del ciclo for en Python es:



```
// Inicio del ciclo
```

```
for x in <colección-iterable>:
```

```
    // bloque de instrucción
```

```
    instrucción 1
```

```
    instrucción 2
```

Algunas consideraciones del ciclo:

- **for**, es la palabra reservada que indica el inicio del bucle.
- **x**, es la variable que almacena cada elemento de la colección iterable en cada iteración; lo hace de forma temporal para cada iteración
- **in**, es un operador que indica que la variable recorrerá los elementos de la colección iterable.
coleccion_iterable, es el conjunto de elementos sobre los que se ejecutará el bucle, puede ser una lista, una tupla, un diccionario o una cadena de texto. Se insiste la colección iterable puede partir de muchas formas, de acuerdo a la variedad de funciones o estructuras propias de Python. Un ejemplo común es el uso de la función `range`, que permite generar una secuencia de valores (lista), con base a los parámetros que se le envíe.
- **Bloque de instrucciones**, es el conjunto de sentencias que se ejecutarán en cada iteración del bucle; estas líneas de código deben estar indentadas de forma obligatoria en Python.

Ejemplo 1:

- Genere una solución que permita presentar la suma y el promedio de los números del 1 al 10.

Solución en miniespecificación

1. Inicio

2. // Declaración de variables

3. `numero, i[1-10]` // Contador: almacena los valores del 1 al 10 en el bucle

4. `suma, i[0-n] <-- 0` // Acumulador: almacena la suma total de los números

5. `promedio, d[0-n]` // Variable para calcular el promedio



- 6.
7. // Proceso para calcular la suma usando un ciclo Para
8. Para (numero <- 1, numero <= 10, 1) entonces // El bucle recorre del 1 al 10
9. suma <- suma + numero // Se acumula el valor actual en la variable suma
10. Fin Para // Fin del bucle después de 10 iteraciones
- 11.
12. // Calcular el promedio
13. promedio <- suma / 10.0 // Se divide la suma total entre 10
- 14.
15. // Mostrar los resultados
16. Escribir "La suma de los números del 1 al 10 es: " + suma
17. Escribir "El promedio de los números del 1 al 10 es: " + promedio
18. Fin

Solución en lenguaje Python

```
print("Ejemplo de uso del ciclo for en Python\n") # es un mensaje en pantalla
# Declaración de variables
numero = 1 # Contador: controla el número hasta donde debe ejecutarse el bucle
suma = 0 # Acumulador: almacena la suma total de los números
promedio = 0 # Variable para calcular el promedio


# Proceso para calcular la suma usando un ciclo For
for numero in range(1, 11): # El bucle recorre los valores del 1 al 10
    # (incremento por defecto es 1)
    # la función range genera una lista con los siguientes
    # valores [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]; donde
    # range es la función propia de python
    # 1 es el valor de inicial para los valores de range
    # 11 es el límite superior, no se incluye, en este
    # caso el máximo valor es 10
    suma = suma + numero # Se acumula el valor actual en la variable suma
# Calcular el promedio, se lo hace fuera del ciclo
promedio = suma / 10 # Se divide la suma total entre 10
# Presentar los resultados
print("La suma de los números del 1 al 10 es: %d" % suma)
print("El promedio de los números del 1 al 10 es: %.2f" % promedio)
```



Se puede observar en la Figura 13 la salida por pantalla que se generará al ejecutar el script de Python

Figura 13

Ejemplo de uso de contador y acumulador a través del ciclo for en Python



```
(envpy3-programacion) $ python ejemplo.py
Ejemplo de uso del ciclo for en Python

La suma de los números del 1 al 10 es: 55
El promedio de los números del 1 al 10 es: 5.50
```

Nota. Elizalde, R., 2025.

En la figura anterior se muestra la salida al ejecutar un código en Python, que hace uso del ciclo for, además de contadores y acumuladores.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Estudiar y sintetizar la información presentada en el [capítulo cuatro del texto “Introducción a la Programación con Python 3”](#), subsección centrada en las estructuras de iterativas, con el propósito de reforzar los conceptos sobre ciclos repetitivos. Esta actividad permitirá al estudiante comprender la lógica detrás de los ciclos for y while. Se



recomienda revisar detenidamente el contenido del recurso y replicar los ejemplos detallados, analizando su funcionamiento y modificándolos para observar cómo los cambios en las condiciones afectan la ejecución del programa. Además, se sugiere diseñar ejercicios propios adicionales que involucren ciclos repetitivos.

2. Revisar los ejemplos planteados sobre uso de las sentencias:

- [break](#).
- [continue](#).

Ejecutar los ejemplos en sus entornos locales (máquinas personales) y revisar la explicación de [las diferencias entre las dos sentencias](#), que son usadas en soluciones computacionales a través del lenguaje de programación Python. Finalmente, se puede revisar mayores detalles en los ejercicios sobre el uso de break y continue en la temática del texto [“Flujo de Ejecución con Bucles”, dentro del capítulo tres del texto “Python a Fondo”](#).

3. Revisar la problemática presentada en el [repositorio](#). La idea es crear una solución haciendo uso de ciclos repetitivos. Se genera una propuesta de solución a través del ciclo [while](#), revísela y compruebe su funcionamiento en un entorno local. Finalmente, usted debe crear una propuesta mediante el ciclo for.

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 6

En la Semana 6, se explora el uso de los arreglos unidimensionales en miniespecificaciones y Python, utilizando listas, la estructura de datos que permite almacenar y manipular múltiples valores dentro de una sola variable. Aprenderán a crear listas, acceder a sus elementos mediante índices y aplicar operaciones como modificación y recorrido. Además, realizarán ejercicios



prácticos para comprender cómo las listas pueden utilizarse en la resolución de problemas computacionales, optimizando la organización y gestión de datos en programas escritos en Python.

Unidad 2. Fundamentos de programación

2.10. Arreglos unidimensionales

En la presente semana se inicia el estudio de estructuras de datos básicas, que serán de mucha ayuda en los procesos de construcción de soluciones. Según Moreno Pérez (2014), indica algunas ideas respecto a las estructuras estáticas, resaltando que pueden existir dos tipos, de vectores y matrices; se caracterizan por ser estructuras que, al ser creadas, ya se conoce el número de elementos que van a tener. Los vectores o arrays están compuestos por un conjunto de posiciones consecutivas de memoria.

Trejos Buriticá (2017) resalta que al estudiar sobre arreglos se debe entender el concepto y uso de los índices; variable que permite acceder a una determinada posición de la estructura.

En este apartado se abordarán los arreglos unidimensionales a través de la creación de miniespecificaciones e implementación en lenguaje Python.

Cuando se requiera implementar una solución usando arreglos unidimensionales en miniespecificación, se deben seguir las siguientes pautas:

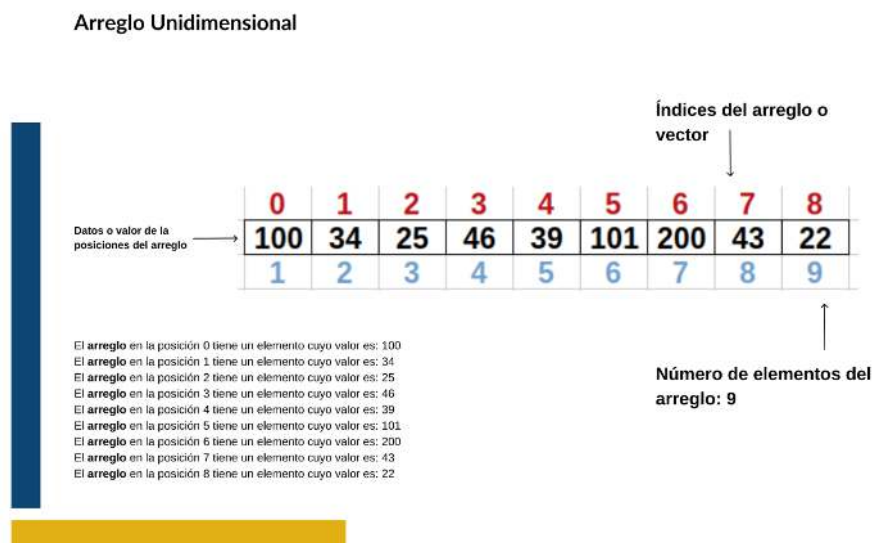
1. Los arreglos tienen características como: nombre, tipo, contenido, tamaño (número de elementos del arreglo), índice (permite acceder a una posición particular del arreglo).
2. Los arreglos poseen elementos o datos que son del mismo tipo (enteros, decimales, cadenas, booleanos, etc.). Esta afirmación varía cuando se estudien los arreglos en lenguaje Python.
3. El arreglo sigue la convención estándar en programación, donde el primer elemento se ubica en la posición 0, el segundo en la posición 1, y así sucesivamente hasta el último índice. En la Figura 14, se indican algunas



características al momento de crear un arreglo; en arreglos, el número de elementos siempre es 1, más que el índice más alto posible ($n = \text{max_index} + 1$).

Figura 14

Características de un arreglo unidimensional



Nota. Elizalde, R., 2025.

La figura anterior explica algunas características de un arreglo unidimensional, como el uso de índices para referenciar a un valor.

4. La representación simbólica en miniespecificación de un arreglo es la siguiente: (Nombre-del-Arreglo(número-elementos-arreglo),tipo-de-dato[dominio]). Ejemplo

◦ (notasEstudiantes(7),i[1-n]) ; donde:

- Nombre-del-Arreglo: notasEstudiantes
- Número elementos-arreglo: 7
- tipo-de-dato del arreglo: Entero
- dominio: 1-n

5. Considerando que al crear una miniespecificación se debe indicar el número de elementos, es importante considerar que los valores por defecto que se le asigna a un arreglo de tipo entero es 0 en cada posición, los valores por defecto que se le asigna a un arreglo de tipo cadena es null en cada posición y se asigna Falso en los valores por defecto de los arreglos de tipo booleano.

6. La asignación se la realiza así:

- `notas-del-arreglo[indice] <-- valor`
- Ejemplo: `notasEstudiantes[0] <-- 9`

A continuación, se deja un ejemplo sencillo en miniespecificación

1. Inicio
2. // Declaración del arreglo de notas con tamaño 2
3. `notasEstudiantes(2)`, índice [0-1]
4. // Definición del índice
5. índice, `i[0-1]`
6. // Asignación de valores al arreglo
7. `notasEstudiantes[0] <-- 9`
8. `notasEstudiantes[1] <-- 10`
9. // Impresión del primer valor almacenado en el arreglo
10. Escribir `notasEstudiantes[0]`
11. // Impresión del segundo valor almacenado en el arreglo
12. Escribir `notasEstudiantes[1]`
13. Fin

En lenguaje Python, no existen arreglos nativos como en otros lenguajes como Java o C, pero se puede usar las estructuras denominadas listas, mismas que poseen un comportamiento similar, pero con ventajas adicionales. Para este apartado del manejo de arreglos unidimensionales con Python, se solicita realizar una lectura comprensiva del recurso [Python práctico: Herramientas, conceptos y técnicas](#) en la sección 13: Listas, Tuplas y Diccionarios.



El recurso indica que las listas en Python, permiten almacenar un conjunto de elementos ordenados que pueden de cualquier tipo, marcando una diferencia de lo revisado en la miniespecificación. Además, al manejar una lista, se sale del concepto estricto de estructura estática, pues, una lista si puede ser expandida en su longitud agregando más elementos. Algunas pautas para crear y usar listas en Python son:

1. Los elementos de una lista deben estar entre []
2. La primera posición de una lista empieza en cero (0), tal cual se lo estudió en la miniespecificación
3. Los elementos deben ser separados por comas (,)
4. Formalmente el tipo de dato en Python de una lista es **list**
5. El número de elemento de una lista se la obtiene a través de la función de Python len, ejemplo, **len(lista)**

A continuación, se dejan algunos ejemplos básicos de uso de una lista en Python

```
# Declaración de una lista sin valores
mi_lista = []
# Declaración de una lista con valores dados
mi_lista_01 = [1, 2, 3, 10, 8, 9]
# Declaración de una lista con valores por defecto (0), de acuerdo
# a un tamaño conocido
# Crea una lista con 10 elementos, todos inicializados en 0
mi_lista = [0]*10
# Presentación en pantalla de una lista
print(mi_lista_01) # Salida: [1, 2, 3, 10, 8, 9]
# Presentación en pantalla de una posición determinada de la lista
print(mi_lista_01[0]) # Salida: 1 (Primer elemento)
print(mi_lista_01[4]) # Salida: 8 (Quinto elemento, índice 4)
# Obtener el número de elementos de un arreglo, usando la función len
numero_elementos = len(mi_lista_01)
print("Número de elementos de la lista: %d" % (numero_elementos))
# Salida: Número de elementos de la lista: 6
# agregar elementos a un arreglo
mi_lista_01.append(100)
# Eliminar un elemento de una posición determinada, siempre y cuando el índice
exista
del mi_lista_01[3]
print(mi_lista_01)
```



```
# Creación de una lista con diferentes tipos de datos en cada posición
mi_lista_02 = ["1", "Loja", 13, 10.2, True, 9]
# Presentación de la lista con tipos datos diversos
print(mi_lista_02)
# Salida: ['1', 'Loja', 13, 10.2, True, 9]
```

Otra estructura, que explica el recurso y que se usa en Python de forma habitual son los diccionarios. Un diccionario en Python almacena pares clave-valor, permitiendo acceder a los valores de manera rápida a través de sus claves. A diferencia de las listas, que almacenan elementos ordenados por índice numérico, los diccionarios permiten acceder a la información mediante nombres descriptivos (claves). Algunas características:

- Las claves no se pueden repetir en el mismo diccionario
- Los valores que se le asignan a las claves pueden tener diversos tipos de datos.
- La forma de iniciar un diccionario vacío es con llaves vacías.

Se deja un ejemplo para explicar de mejor manera:

```
# Creación de un diccionario con información de una persona
# Un diccionario almacena datos en formato "clave: valor"
persona = {
    "nombre": "Carlos",
    "edad": 30,
    "ciudad": "Quito",
    "telefono": "0987654321"
}
# Imprimir el diccionario completo
print(persona)
# Salida: {'nombre': 'Carlos', 'edad': 30, 'ciudad': 'Quito', 'telefono': '0987654321'}
# Obtener y mostrar todas las claves del diccionario
print(persona.keys())
# Salida: dict_keys(['nombre', 'edad', 'ciudad', 'telefono'])
# Acceder al valor de una clave específica
print(persona['ciudad']) # Salida: Quito
# Recorrer el diccionario e imprimir los valores asociados a cada clave
for llave in persona.keys(): # Itera sobre cada clave en el diccionario
    # Accede al valor correspondiente a cada clave y lo imprime
    print(persona[llave])
# La salida del bucle for será:
# Carlos
```



30
Quito
0987654321

Siguiendo con el tema de arreglos unidimensionales, se deja el siguiente ejemplo:

Ejemplo 1:

- Genere una solución que permita ingresar por teclado valores de tipo entero a un arreglo. El tamaño del arreglo debe ser ingresado por teclado. Los valores permitidos son aquellos que están comprendidos entre el número 20 y los menores o iguales a 40; si se ingresa un valor fuera del rango, se debe asumir un valor de 100 por defecto. Finalmente presentar los valores ingresados, haciendo uso de una variable de tipo cadena acumuladora.

Solución en miniespecificación

1. Inicio
2. // Declaración de variables
3. tamaño, i[1-n] // Variable para almacenar el tamaño del arreglo
4. Escribir "Ingrese el tamaño del arreglo:"
5. Leer tamaño // Se captura el tamaño del arreglo desde el teclado
- 6.
7. (numeros(tamaño), i[0-n]) // Declaración del arreglo con el tamaño ingresado
8. indice, i[0-n] // Contador para recorrer el arreglo
9. valoresIngresados, x(100)[{a-z}, {A-Z}, {0-9}, { }, {BS}] // Acumulador de cadena
- 10.
11. // Ingresar valores en el arreglo con validación
12. Para (indice <= 0, indice < tamaño, 1) entonces
13. Escribir "Ingrese un número para la posición " + (indice + 1) + ":"
14. Leer numeros[indice] // Captura el valor ingresado por el usuario
- 15.
16. // Verificación del rango permitido



```

17. Si numeros[indice] >= 20 and numeros[indice] <= 40 entonces
18.     // El número está dentro del rango permitido, no se hace nada
19. De lo contrario
20.     numeros[indice] <-- 100 // Asignación de valor por defecto
21. Fin Si
22. Fin Para
23.
24. // Agregar los valores a la cadena acumuladora
25. valoresIngresados <-- "Valores ingresados: \n"
26. Para (indice <-- 0, indice < tamaño, 1) entonces
27.     // Se agrega valores a la cadena, manteniendo lo que ya existía
28.     valoresIngresados <-- valoresIngresados + numeros[indice] + "\n"
29. Fin Para
30.
31. // Mostrar los resultados
32. Escribir valoresIngresados
33. Fin

```



Solución en lenguaje Python

```

# Inicio
# Solicitar el tamaño del arreglo
tamaño = int(input("Ingrese el tamaño del arreglo: ")) # Captura el tamaño
ingresado por el usuario
# Declaración del arreglo con el tamaño ingresado
numeros = [0] * tamaño # Se crea una lista de tamaño 'tamaño' inicializada en 0
# Acumulador de cadena para almacenar los valores ingresados
valoresIngresados = "Valores ingresados:\n"
# Ingresar valores en el arreglo con validación
for indice in range(tamaño):
    numero = int(input(f"Ingrese un número para la posición {indice + 1}: ")) #
Captura del usuario
    # Verificación del rango permitido (20-40)
    if 20 <= numero <= 40:
        numeros[indice] = numero # Se almacena el número ingresado
    else:
        numeros[indice] = 100 # Se asigna 100 si el número no está en el rango
permitido
# Agregar los valores a la cadena acumuladora
for i in numeros: # se hace uso de la secuencia con los valores previos ingresados

```



```
valoresIngresados = f"{valoresIngresados}{i}\n" # Concatenar valores a la cadena
# Mostrar los resultados
print(valoresIngresados)
# Fin
```

En función de la solución de Python, es importante recalcar los dos enfoques se usa para recorrer una secuencia:

1. Enfoque uno: for indice in range(tamaño):

- En este enfoque se usa la iteración mediante índices, range(tamaño) – genera una secuencia de números desde 0 hasta un valor menor del ingresado para tamaño; a través de la forma numeros[indice] se le asigna un valor a una determinada posición de la estructura.
- Considerar esta forma cuando se necesita modificar valores específicos a una posición del arreglo

2. Enfoque dos: for i in numeros:

- Se denomina a este enfoque iteración directa sobre una secuencia de información que posee datos, para el ejemplo, los valores que fueron ingresados para la lista números.
- i toma los valores almacenados en números, sin acceder a los índices.
- Esta forma es útil cuando solo necesita leer los valores, no modificarlos; para el ejemplo, i solo almacena copia de los valores de la lista, pero no permite alterarlos dentro del bucle.

Estos dos enfoques serán usando en diversas problemáticas, es importante que se lo recuerde. En la Figura 15 se puede observar la ejecución en Python.



Figura 15

Ejemplo de uso de lista en Python

Arreglo Unidimensional

```
(envpy3-programacion) $ python ejemplo.py
Ingrese el tamaño del arreglo: 7
Ingrese un número para la posición 1: 21
Ingrese un número para la posición 2: 25
Ingrese un número para la posición 3: 29
Ingrese un número para la posición 4: 1
Ingrese un número para la posición 5: 10
Ingrese un número para la posición 6: 39
Ingrese un número para la posición 7: 90
Valores ingresados:
21
25
29
100
100
39
100
```

Nota. Elizalde, R., 2025.

En la figura anterior se describe la salida por pantalla de un código hecho en Python que hace uso de la estructura list.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Revisar con atención el material del texto [“Introducción a la Programación con Python 3”, específicamente en el capítulo cinco “Tipos Estructurados – Secuencias”](#), en la sección Listas, para profundizar en el uso de la estructura list en Python. Esta actividad permitirá al estudiante comprender cómo manipular listas, acceder a sus elementos, modificar su contenido y aplicar métodos incorporados

para su gestión eficiente. Se recomienda analizar los ejemplos detallados en el recurso y replicarlos en su propio entorno de programación para observar su comportamiento. Además, se sugiere experimentar con las funciones `append()` y `sort()`, aplicándolas en distintos escenarios. Finalmente, realizar ejercicios donde se combinen listas con estructuras de control permitirá fortalecer la comprensión sobre su aplicación en la resolución de problemas computacionales.

2. Recrear la [problemática planteada](#) en su entorno local, [revisar la solución](#) y verificar que el código esté adecuado en función de lo que se desea solucionar. Luego, usted debe modificar el problema, agregando una nueva lista en el proceso de solución. Este ejemplo reafirma conceptos fundamentales en Python, como el uso de listas y su indexación, permitiendo acceder a elementos mediante su posición numérica dentro de la estructura de datos. También refuerza la importancia de la iteración con `for` y `range(len(...))`, facilitando el recorrido de listas de manera controlada cuando es necesario acceder a múltiples listas relacionadas simultáneamente.

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 7

En la Semana 7, los estudiantes estudiarán los arreglos bidimensionales en miniespecificaciones y Python, utilizando listas anidadas, una estructura que permite organizar datos en filas y columnas, similar a una matriz. Aprenderán a declarar, inicializar y recorrer listas bidimensionales con ciclos anidados, además de aplicar operaciones como acceso, modificación y búsqueda de elementos. Asimismo, realizarán ejercicios prácticos en los que implementarán matrices para representar y manipular información estructurada, principalmente en programas desarrollados en Python.



Unidad 2. Fundamentos de programación

2.11. Arreglos bidimensionales

Luego de estudiar y entender cómo crear arreglos en miniespecificación y manejo de listas en Python, es momento de abordar el tema de arreglos bidimensionales o matrices.

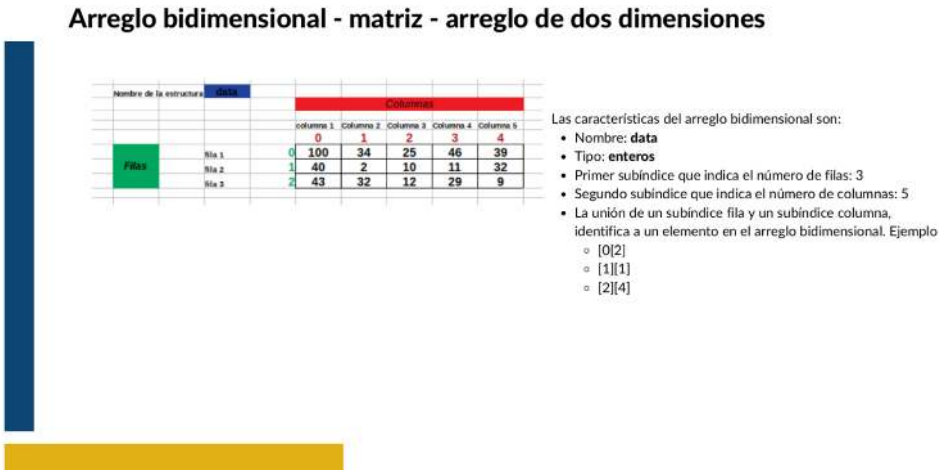
Al momento de referirse a las matrices, Trejos Buriticá (2017) indica que, son consideradas como un conjunto de datos que están organizados a través de filas y columnas. Se requiere indicar tanto la fila como la columna para asociar un valor o para obtenerlo.

Como en el manejo de arreglos unidimensionales, las matrices son estructuras que están formadas con tipos de datos de la misma especie, contenidos en un conjunto de vectores filas o líneas, los cuales se cruzan con otro conjunto de vectores columna (filas, columna). Los arreglos bidimensionales tienen características como: Nombre; Tipo; Primer valor que indica el número de filas o líneas; Segundo valor que indica el número de columnas. La unión de un subíndice fila y un subíndice columna, identifica a un elemento en el arreglo bidimensional. Los subíndices empiezan en cero 0; y el contenido.

En la Figura 16, se indica un arreglo bidimensional o matriz, donde los datos están organizados en filas y columnas. La estructura se denomina data y contiene valores enteros, con 3 filas y 5 columnas. Cada elemento dentro de la matriz se identifica mediante un par de índices [fila][columna], lo que permite acceder a valores específicos. Por ejemplo, data[0][2] corresponde al valor 25, data[1][1] al 40 y data[2][4] al 9. La figura también resalta que la unión de un subíndice de fila con un subíndice de columna permite localizar cualquier elemento dentro de la matriz.



Figura 16
Arreglo bidimensional con datos en cada posición



Nota. Elizalde, R., 2025.

La figura anterior resalta las características a considerar al momento de iniciar o declarar un arreglo bidimensional, principalmente en miniespecificaciones.

La representación simbólica en miniespecificación de un arreglo bidimensional es la siguiente:

• (Nombre-del-Arreglo(númeroFilas)(númeroColumnas),tipo-de-dato[dominio]); ejemplo, (notasEstudiantes(2)(3),i[1-n])

Donde, nombre-del-arreglo es notasEstudiantes, número filas del arreglo es 2, número columnas del arreglo es 3, tipo-de-dato del arreglo es entero y el dominio de datos es desde 1 a n(valor máximo posible)

Al igual que en los arreglos unidimensionales en las matrices en miniespecificación se debe indicar el número de elementos de filas y columnas, es importante considerar que los valores por defecto que se le asigna a un arreglo de tipo entero es 0 en cada posición, los valores por



defecto que se le asigna a un arreglo de tipo cadena es null en cada posición y se asigna Falso en los valores por defecto de los arreglos de tipo booleano. La asignación se la realiza así:

- `notas-del-arreglo[fila][columna] <-- valor`
 - Ejemplo: `notasEstudiantes[0][2] <-- 9`

A continuación, se deja un ejemplo sencillo en miniespecificación

```
// Ejemplo de uso de arreglos bidimensionales
//
```

1. Inicio
2. // Declaración de un arreglo bidimensional con 2 filas y 3 columnas
3. (`notasEstudiantes(2)(3)`, `i[1-n]`)
4. // Asignación de valores en la matriz (`posición [fila][columna]`)
5. `notasEstudiantes[0][0] <-- 10` // Posición `[0][0]`
6. `notasEstudiantes[0][1] <-- 7` // Posición `[0][1]`
7. `notasEstudiantes[0][2] <-- 8` // Posición `[0][2]`
8. `notasEstudiantes[1][0] <-- 9` // Posición `[1][0]`
9. `notasEstudiantes[1][1] <-- 5` // Posición `[1][1]`
10. `notasEstudiantes[1][2] <-- 7` // Posición `[1][2]`
11. // Mostrar valores específicos del arreglo bidimensional
12. Escribir `notasEstudiantes[0][2]` // Muestra el valor de la posición `[0][2]`
13. Escribir `notasEstudiantes[1][1]` // Muestra el valor de la posición `[1][1]`
14. Escribir `notasEstudiantes[1][2]` // Muestra el valor de la posición `[1][2]`
15. Fin

En relación a la implementación de matrices en Python, es importante considerar que, a diferencia de lenguajes como Java o C, Python no cuenta con una estructura nativa para matrices. En estos otros lenguajes, las matrices se manejan mediante arreglos bidimensionales con un tamaño fijo y un acceso directo en memoria. Sin embargo, en Python, se utilizan listas anidadas, donde cada fila de la matriz se representa como una lista dentro de otra lista más grande.



Este enfoque ofrece una mayor flexibilidad, permitiendo matrices de tamaños dinámicos en lugar de estructuras de tamaño fijo como en C o Java. Para recorrer o manipular una matriz en Python, se emplean bucles for anidados, donde el primer bucle recorre las filas y el segundo las columnas. También se pueden usar [comprensiones de listas](#) para generar matrices dinámicamente.

Un ejemplo sencillo en Python

```
# Declaración de una matriz en Python (Lista Anidada Vacía)
# En este caso, se crean dos listas vacías dentro de otra lista.
mi_matriz = [], []
print(mi_matriz) # Salida: [], [] (Matriz con dos filas vacías)
# Declaración de una matriz en Python con valores predefinidos
# Aquí cada sublista representa una fila en la matriz
mi_matriz_2 = [
    [1, 2], # Fila 0
    [3, 10], # Fila 1
    [8, 9] # Fila 2
]
print(mi_matriz_2)
# Salida: [[1, 2], [3, 10], [8, 9]] (Matriz de 3 filas x 2 columnas)
# Declaración de una matriz con valores por defecto (0), según un tamaño conocido
# Se crea una matriz con 2 filas, cada una con 10 elementos inicializados en 0
mi_matriz_3 = [[0] * 10, [0] * 10]
print(mi_matriz_3)
# Salida: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Ejemplo 1:

- Generar una solución donde, dadas las notas de tres (3) estudiantes; generar una solución que permita encontrar el promedio de las calificaciones por cada conjunto de notas de los estudiantes. La problemática sugiere el uso de estructuras de datos. Las notas están en un arreglo bidimensional llamado notas; notas tiene 3 filas y 4 columnas; los datos deben ser ingresados por teclado.

Solución en miniespecificación

1. Inicio
2. // Declaración de variables



```

3. (notas(3)(4), d[0-n]) // Matriz de 3 filas (estudiantes) y 4 columnas (notas)
4. (promedios(3), d[0-n]) // Arreglo unidimensional para almacenar los
   promedios
5. estudiante, i[0-2] // Contador para recorrer filas (estudiantes)
6. nota, i[0-3] // Contador para recorrer columnas (notas)
7. suma, d[0-n] // Acumulador para la suma de notas de un estudiante
8.
9. // Ingreso de notas por teclado
10. Para (estudiante <-- 0, estudiante < 3, 1) entonces
11.   Para (nota <-- 0, nota < 4, 1) entonces
12.     // Se usa (nota + 1) y (estudiante + 1) porque los índices inician en 0
13.     Escribir "Ingrese la nota " + (nota + 1) + " del estudiante " + (estudiante
        + 1) + ":"
14.     Leer notas[estudiante][nota] // Captura las notas en la matriz
15.   Fin Para
16. Fin Para
17.
18. // Cálculo del promedio por estudiante y almacenamiento en el arreglo
   promedios
19. Para (estudiante <-- 0, estudiante < 3, 1) entonces
20.   suma <-- 0 // Reinicia el acumulador de suma por cada estudiante
21.   Para (nota <-- 0, nota < 4, 1) entonces
22.     suma <-- suma + notas[estudiante][nota] // Suma todas las notas del
        estudiante
23.   Fin Para
24.   promedios[estudiante] <-- suma / 4 // Guarda el promedio en el arreglo
25. Fin Para
26.
27. // Mostrar los promedios almacenados en el arreglo promedios
28. Escribir "Promedios de los estudiantes:"
29. Para (estudiante <-- 0, estudiante < 3, 1) entonces
30.   Escribir "El promedio del estudiante " + (estudiante + 1) + " es: " +
        promedios[estudiante]
31. Fin Para

```



32. Fin

Solución en Python

```
# Inicio

# Declaración de la matriz para almacenar notas (3 filas - estudiantes, 4 columnas -
# notas)
notas = [[0] * 4,
         [0] * 4,
         [0] * 4] # Matriz 3x4 correctamente inicializada en 0
# Declaración del arreglo unidimensional para almacenar los promedios de los
# estudiantes
promedios = [0] * 3 # Lista de 3 elementos inicializados en 0 (un promedio por
# estudiante)
# Ingreso de notas por teclado
for estudiante in range(3): # Recorre las filas (estudiantes)
    for nota in range(4): # Recorre las columnas (notas)
        # Se usa (nota + 1) y (estudiante + 1) porque los índices inician en 0
        nota_ingresada = input(f"Ingrese la nota {nota + 1} del estudiante {estudiante +
1}: ")
        notas[estudiante][nota] = float(nota_ingresada) # Conversión a float para
# manejar decimales
# Cálculo del promedio por estudiante y almacenamiento en el arreglo promedios
for estudiante in range(3): # Recorre las filas (estudiantes)
    suma = 0 # Reinicia el acumulador de suma por cada estudiante
    for nota in range(4): # Recorre las notas del estudiante
        suma = suma + notas[estudiante][nota] # Acumulación correcta de notas
    promedios[estudiante] = suma / 4 # Cálculo correcto del promedio
# Mostrar los promedios almacenados en el arreglo promedios
print("\nPromedios de los estudiantes:")
contador = 1
for promedio in promedios:
    print(f"El promedio del estudiante {contador} es: {promedio:.2f}") # Se imprime
# con dos decimales
    contador = contador + 1
# Fin
```

En la Figura 17, se observa la ejecución del solución e Python



Figura 17

Uso de Python con estructuras de tipo list anidadas

Ejemplo de uso de Python con listas anidadas

```
(envpy3-programacion) $ python ejemplo.py
Ingrese la nota 1 del estudiante 1: 10
Ingrese la nota 2 del estudiante 1: 9
Ingrese la nota 3 del estudiante 1: 7.5
Ingrese la nota 4 del estudiante 1: 5
Ingrese la nota 1 del estudiante 2: 10
Ingrese la nota 2 del estudiante 2: 10
Ingrese la nota 3 del estudiante 2: 10
Ingrese la nota 4 del estudiante 2: 9
Ingrese la nota 1 del estudiante 3: 5
Ingrese la nota 2 del estudiante 3: 6.7
Ingrese la nota 3 del estudiante 3: 3.3
Ingrese la nota 4 del estudiante 3: 10
```

Nota. Elizalde, R., 2025.

En la figura anterior se muestra la salida por pantalla, al momento de ejecutar un script de Python, que hace uso de listas (tipo list) en Python.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Revisar con atención el material del texto [“Introducción a la Programación con Python 3”, específicamente en el capítulo cinco, sección “Tipos Estructurados – Secuencias”, apartado Matrices](#), para profundizar en el uso de listas combinadas en Python. Esta actividad permitirá al estudiante comprender cómo representar y manipular matrices mediante listas anidadas, accediendo a sus elementos y



aplicando métodos para su modificación y recorrido. Se recomienda analizar los ejemplos propuestos en el recurso y replicarlos en un entorno de programación para observar su funcionamiento. Además, se sugiere diseñar ejercicios que incluyan la iteración sobre matrices con bucles anidados (for), aplicando operaciones como suma de filas o columnas, búsqueda de valores y ordenamiento de datos.

2. Revisar el ejemplo planteado en el [repositorio](#) y la solución realizada en [lenguaje Python](#), recuerde que es muy importante que este ejercicio sea ejecutado en su entorno local. Además, lo planteado es importante para seguir en el proceso de entendimiento sobre el manejo de estructuras de datos en Python, específicamente el uso de listas, la manipulación de cadenas de acumulación, y el control de flujo con bucles y condiciones. Se resalta el uso de listas que permiten almacenar y recuperar datos de manera flexible. Finalmente, se recomienda recrear el ejemplo usando otro tipo de ciclo repetitivo.
3. Realizar las actividades de la [sección quince denominada Listas y Matrices en el capítulo tres del texto Fundamentos básicos de programación: aplicación práctica con Scratch y Python](#), enfocándose en los conceptos de listas y listas combinadas en lenguaje Python. Esta actividad permitirá al estudiante ejercitarse sobre la estructura y manipulación de listas simples y anidadas, su aplicación en la organización de datos.
4. Estimado estudiante, ha llegado el momento de evaluar su comprensión sobre los temas abordados en la segunda parte de la Unidad 2: Fundamentos de programación, en la que se profundiza en el uso de ciclos repetitivos (while y for), contadores, acumuladores, arreglos unidimensionales y bidimensionales en miniespecificaciones y lenguaje de programación Python. Esta autoevaluación le permitirá identificar qué conceptos domina y en cuáles debería reforzar su aprendizaje.



Antes de realizar la autoevaluación, se recomienda seguir estas estrategias de estudio:

- Consultar los recursos educativos (documentos, ejemplos de código y bibliografía recomendada) para reforzar sus conocimientos.
- Revisar cuidadosamente el contenido de cada tema, comprendiendo el funcionamiento y las diferencias entre los ciclos while y for.
- Practicar con ejercicios sobre contadores y acumuladores, comprendiendo cómo se implementan y cuándo utilizarlos.
- Analizar la estructura de los arreglos unidimensionales y bidimensionales, comprendiendo la manipulación de datos mediante índices y su implementación en Python.
- Familiarizarse con el uso de listas y diccionarios en Python, aplicándolos en la solución de problemas.

Recuerde que cada pregunta tiene una única respuesta correcta. Lea con atención cada enunciado y seleccione la opción que usted considere adecuada. ¡Mucho éxito en esta autoevaluación!



Autoevaluación 3

1. Autocompletar:

El bucle más adecuado, cuando no se conoce con anterioridad cuántas veces se ejecutará la iteración, es _____.

2. ¿Cuál es la diferencia principal entre un contador y un acumulador en programación?

Respuestas:

- a. Un contador almacena la cantidad de veces que ocurre un evento, mientras que un acumulador guarda la suma de valores, que pueden ser diversos.
- b. Un contador no necesita ser inicializado antes de su uso.



c. Un contador solo almacena números impares, mientras que un acumulador almacena números pares.

3. **¿Cuál de las siguientes opciones es una ventaja de los arreglos unidimensionales en Python (listas)?**

Respuestas:

- a. Solo pueden contener datos numéricos.
- b. Permiten almacenar múltiples elementos en una sola variable.
- c. No pueden ser modificados después de su creación.

4. **Autocompletar:**

La estructura de datos en Python que permite almacenar valores en formato clave-valor es _____.

5. **¿Cuál será la salida de este código?**

```
contador = 0

while contador < 5:

    contador = contador + 1

    if contador == 4:

        print("Número encontrado")
```

Respuestas:

- a. Error en la ejecución.
- b. El programa no imprime nada.
- c. Número encontrado.

6. **Considere el siguiente código**

```
suma = 0
```



```
for i in range(1, 6):
```

```
    suma = suma + i
```

```
print(suma)
```

Se afirma lo siguiente: La salida por pantalla al ejecutar es 15.

Respuestas:

a. Verdadero ()

b. Falso ()

7. **¿Cuál será la salida si el usuario ingresa 10?**

```
n = int(input("Ingrese un número: "))
```

```
contador = 0
```

```
while n > 1:
```

```
    n = n / 2
```

```
    contador = contador + 1
```

```
print(contador)
```

Respuestas:

a. 2.

b. 3.

c. 4.

8. **¿Cuál es la salida del siguiente código?**

```
numeros = [10, 20, 30, 40]
```

```
for num in numeros:
```

```
    if num > 25:
```



```
print(num)
```

Respuestas:

- a. 30 y 40.
- b. 30.
- c. 70.

9. **¿Cuál es la salida del siguiente código?**

```
matriz = [[5, 10], [15, 20]]
```

```
contador = 0
```

```
for fila in matriz:
```

```
    for elemento in fila:
```

```
        if elemento % 5 == 0:
```

```
            contador = contador + 1
```

```
print(contador)
```

Respuestas:

- a. 2.
- b. 3.
- c. 4.

10. **¿Qué valor imprimirá el siguiente código?**

```
matriz = [[4, 8, 12], [3, 6, 9]]
```

```
suma_pares = 0
```

```
for fila in matriz:
```

```
    for elemento in fila:
```



```
if elemento % 2 == 0:

    suma_pares = suma_pares + elemento

print(suma_pares)
```

Respuestas:

- a. 24.
- b. 30.
- c. 12.

[Ir al solucionario](#)

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 8

Actividades finales del bimestre

La semana 8 es un período importante para que usted, estimado estudiante, pueda volver a revisar los temas estudiados en el primer bimestre y reforzar aquellos conceptos que requieran mayor comprensión. Este tiempo está destinado a consolidar su aprendizaje y prepararse para la evaluación del bimestre.

En esta semana, se propone revisar las siguientes unidades:

- Unidad 1: Introducción a la programación y solución de problemas.
- Unidad 2: Fundamentos de programación.

Las estrategias sugeridas para aprovechar este período de repaso incluyen

- Elaborar resúmenes, cuadros sinópticos o mapas conceptuales sobre los temas estudiados.



- Revisar las actividades realizadas en semanas anteriores y, cuando sea necesario, recrear los ejercicios desde cero para afianzar el manejo de herramientas y conceptos.
- Consultar en los recursos de aprendizaje recomendados en la planificación de la asignatura.
- Practicar con [ejercicios similares a los presentados](#) en la guía didáctica para reforzar la resolución de problemas.
- Repasar las preguntas de autoevaluación para evaluar su comprensión de los contenidos.
- Revisar los cuestionarios de repaso disponibles en la plataforma virtual de aprendizaje.

¿Listo para rendir la evaluación bimestral?





Segundo bimestre

Resultado de aprendizaje 2:

Evalúa técnicas avanzadas de manipulación de datos en el manejo de grandes volúmenes de información, el busca de optimizar el procesamiento y la gestión de datos en sistemas complejos.

Para lograr este resultado de aprendizaje, la asignatura Fundamentos de Programación proporcionará una base sólida en el diseño e implementación de algoritmos, con un enfoque en la manipulación avanzada de datos y la optimización del procesamiento. A través del estudio de funciones, recursividad, modularización y estructuras de datos avanzadas, los estudiantes desarrollarán habilidades clave para estructurar programas eficientes y escalables.

El proceso de aprendizaje combinará actividades autónomas, donde los estudiantes explorarán el uso de funciones que retornan y no retornan valores, paso de parámetros, variables locales y globales, así como la organización del código mediante módulos y paquetes. Además, se abordará el manejo de excepciones y operaciones de entrada/salida (I/O) para garantizar el procesamiento de información. Finalmente, la integración de librerías externas permitirá conocer sobre herramientas que optimizan la manipulación de datos en entornos reales.

Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.





Semana 9

En la Semana 9, los estudiantes comenzarán el estudio del manejo avanzado de datos y estructuras, enfocándose en el uso de funciones en Python. Aprenderán el concepto de función y su importancia en la modularización del código, diferenciando entre funciones que no retornan valores y funciones que sí lo hacen. Además, realizarán ejercicios prácticos para implementar funciones, mejorando la organización y reutilización del código en sus programas.

Unidad 3. Manejo avanzado de datos y estructuras

3.1. Concepto de función

En la presente semana se estudiará una estructura importante en la formación de las bases de programación. A continuación, se listan conceptos respecto a la función; la idea es que se tenga muy claro la importancia, el proceso de implementación y uso en la formulación de soluciones.

Según Trejos Buriticá (2017):

“Una función es un conjunto de órdenes que tiene las siguientes características:

- Permite lograr un objetivo.
- Tiene un único nombre identificativo.
- Puede necesitar parámetros para lograr dicho objetivo.
- Nos puede retornar un resultado que deberá concordar con el objetivo propuesto.
- Puede ser manejado como una sola unidad”. (p. 364).

Según Muñoz Guerrero y Trejos Buriticá (2021):

“Una función es un conjunto breve de instrucciones que permiten alcanzar fácilmente un pequeño objetivo. Las funciones son la base para resolver los tres grandes problemas de la programación: a) la



reutilización del código, b) la simplificación del objetivo y c) la facilidad para realizar las pruebas en frío. Con las funciones se hace efectiva la estrategia “Divide y Vencerás”, que es una manera de hacer que los programas, por complejos que parezcan, sean más fáciles de entender y, sobre todo, más fáciles de corregir”. (p. 44).

El agregar el concepto de función en las primeras etapas de la formación de fundamentos de programación, ayuda a la comprensión más significativa de la lógica de la programación y un inicio más sencillo en la comprensión de nuevos paradigmas de programación en el futuro, por ejemplo, la Programación Orientada a Objetos.

En los temas del primer bimestre, se destacó la importancia de seguir un proceso estructurado para la resolución de problemas, iniciando con la generación de algoritmos, su miniespecificación y, finalmente, la implementación de soluciones en lenguajes de alto nivel. Esta metodología sigue siendo fundamental, pero en las temáticas de este nuevo bimestre, se profundizará en nuevos conceptos enfocándose en el lenguaje de programación Python.

En el estudio de funciones, es fundamental comprender dos conceptos clave: procedimientos y funciones. Ambos comparten características similares en cuanto a su estructura y propósito dentro de un programa, ya que permiten modularizar el código, mejorar su legibilidad y facilitar la reutilización. Sin embargo, existe una diferencia fundamental entre ellos: mientras que un procedimiento ejecuta una serie de instrucciones sin devolver un valor, una función realiza un cálculo o proceso y retorna un resultado que puede ser almacenado en una variable o utilizado en expresiones dentro del código. Dado que en Python no existe una distinción explícita entre procedimientos y funciones como en otros lenguajes, siempre se utilizará el término función para referirse a esta estructura.



3.2. Funciones que no regresan valor

Las funciones que no regresan o devuelven valor, son consideradas como procedimientos, que ejecutan una serie de acciones, pero no retornan un valor al finalizar su ejecución. Se usa, principalmente, para realizar tareas dentro de una solución, como mostrar mensajes en pantalla o modificar variables globales, etc.

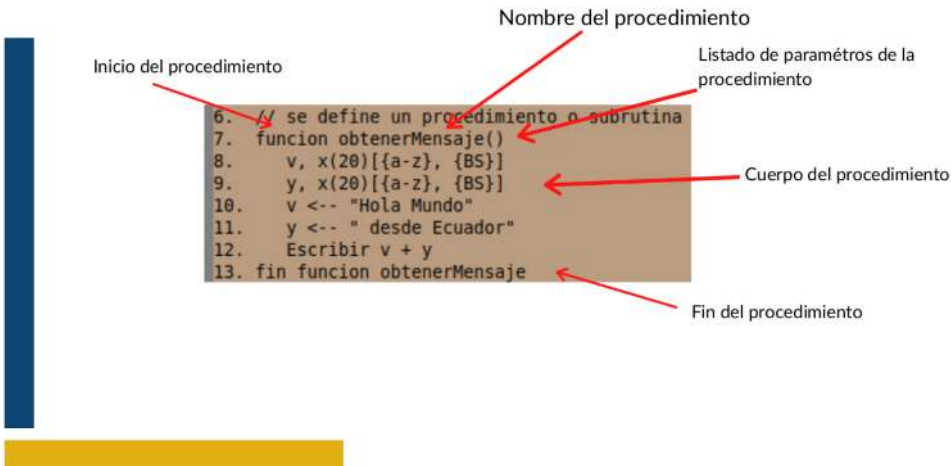
En la Figura 18, muestra la estructura base de un procedimiento, destacando sus principales componentes y su funcionamiento dentro de un programa.

1. Inicio del procedimiento, se define con la palabra clave **funcion**, seguida del nombre del procedimiento (obtenerMensaje); en este caso, el procedimiento no recibe parámetros, ya que los paréntesis () están vacíos.
2. Cuerpo del procedimiento: Es la parte donde se incluyen las instrucciones que se ejecutarán cuando el procedimiento sea llamado. En este caso, se asignan valores a las variables *v* / *y* con los textos "Hola Mundo" y "desde Ecuador", respectivamente. Luego, se utiliza la instrucción Escribir *v* + *y* para concatenar y mostrar el mensaje "Hola Mundo desde Ecuador" .
3. Fin del procedimiento, se marca con fin de funcion obtenerMensaje, lo que indica que el bloque de código del procedimiento ha finalizado.



Figura 18
Ejemplo de Procedimiento en Miniespecificación

Procedimiento en Miniespecificación



Nota. Elizalde, R., 2025.

La figura anterior describe las características que se consideran para crear un procedimiento en miniespecificación.

Seguimos, es el momento de abordar funciones que no retornan valor desde lenguaje Python. Antes de ello, se quiere explicar el uso de una sentencia de código que permite discriminar el uso de un archivo de Python, según su forma de ejecución:

A través del uso de:

```
if __name__ == '__main__'
```

Se pueden generar las siguientes situaciones, pues en Python, cuando un archivo .py se ejecuta, el intérprete asigna automáticamente un valor especial a la variable `__name__`. Este valor depende de cómo se está ejecutando el script:

1. Si el script se ejecuta directamente, es decir, cuando lo ejecutamos como un programa principal, por ejemplo, desde un terminal como: `python nombre_archivo.py`; `__name__` toma el valor `'__main__'`; con lo que el flujo de información empieza bajo las líneas de código del condicional; es importante que se recuerde lo anterior.
2. Si el script se importa como un módulo en otro archivo, `__name__` toma el nombre del archivo sin la extensión .py.

Lo anterior, permite indicar que `if __name__ == '__main__'` puede hacer funcionar a un archivo de Python, como un programa independiente o como un módulo reutilizable en otros archivos.

Para entender el proceso de implementación de funciones en Python, se recomienda realizar una lectura comprensiva de la sección 17 denominada Funciones del texto llamado [Python práctico: Herramientas, conceptos y técnicas](#).

En el texto, se indica la forma estándar de crear una función en Python de la siguiente manera:

```
def obtener_mensaje():  
    // instrucciones  
    // instrucciones  
    pass
```

Algunas consideraciones importantes:

1. Se utiliza la palabra reservada `def` para definir una función en Python.
2. En este ejemplo se usa `obtener_mensaje` como el nombre de la función, el cual debe seguir las reglas de nombres en Python (sin espacios, sin caracteres especiales, descriptivo).



3. Los paréntesis () indican que la función puede recibir parámetros, aunque en este caso está vacía. En las siguientes temáticas se explicará de forma detallada el uso de parámetros.
4. Los dos puntos : indican que el bloque de código de la función comienza en la siguiente línea, al igual que en temáticas de Python estudiadas, se requiere indentación de forma obligatoria.
5. Cuerpo de la función, dentro de la función se colocan las instrucciones que se ejecutarán cuando la función sea llamada; aquí, se pueden incluir operaciones, cálculos, condicionales, bucles, impresión de mensajes, etc.
6. pass, es una palabra reservada en Python que se usa cuando una función no tiene una implementación aún, se utiliza como expresión para evitar errores de sintaxis si la función está vacía. Luego, esta expresión debe ser reemplazada por código que implemente la solución a la problemática.
 - Ejemplo 1: Genera una solución que permita ingresar por teclado datos de empleados como: nombres, apellidos, años de nacimiento; el usuario decide hasta cuando ingresar valores, cuando se termine el ciclo, debe presentarse el listado de empleados, donde se indique la edad del empleado con base al año de nacimiento.

```
# Importa el módulo datetime para manejar fechas y obtener el año actual
import datetime
```

```
# Define un procedimiento (función sin retorno) en Python
```

```
def generar_reporte():
```

```
    """
```

```
    Procedimiento que recopila datos de empleados y calcula su edad
    en función del año de nacimiento ingresado.
```

```
    """
```

```
    mensaje_final = "" # Almacena los datos ingresados por el usuario
```

```
    bandera = True # Controla la ejecución del bucle while para permitir
    múltiples ingresos de datos
```




```

# Inicia un bucle para ingresar información de empleados
while bandera:
    # Captura el nombre y apellido del empleado
    nombre = input("Ingrese nombre de empleado: ")
    apellido = input("Ingrese apellido de empleado: ")

    # Captura y convierte la entrada del año de nacimiento a entero
    anio_nacimiento = input("Ingrese año de nacimiento de empleado: ")
    anio_nacimiento = int(anio_nacimiento) # Convierte la entrada a entero

    # Calcula la edad utilizando datetime.datetime.now().year
    # Obtiene el año actual y se resta el año de nacimiento
    edad = datetime.datetime.now().year - anio_nacimiento

    # Acumula los datos en un string formateado
    mensaje_final = f"{mensaje_final}Nombre: {nombre}\n" \
        f"Apellido: {apellido}\n" \
        f"Edad: {edad}\n" \
        f"-----|||-----\n"

    # Solicita al usuario si desea continuar o salir del ciclo
    opcion = input("Ingrese 1 si desea salir del ciclo: ")
    opcion = int(opcion) # Convierte la entrada a entero

    # Si el usuario ingresa "1", la bandera se cambia a False y el ciclo termina
    if opcion == 1:
        bandera = False

# Muestra en pantalla el mensaje final con todos los datos ingresados
print(mensaje_final)

# Este procedimiento no utiliza "return", ya que solo ejecuta una tarea
# sin devolver un valor explícito. En Python, esto significa que retorna None
por defecto

```



```
# Define el punto de entrada principal del script
# Garantiza que el código dentro de este bloque solo se ejecute si el script
# es ejecutado directamente y no si se importa como módulo en otro archivo
if __name__ == "__main__":
    #
    # Llama al procedimiento (que en Python se define como una función sin
    retorno)
    # No es necesario recibir su valor en una variable, ya que no devuelve datos
    generar_reporte()
```

En la Figura 19, se puede revisar la ejecución del script de Python, haciendo uso de procedimientos

Figura 19

Uso de lenguaje Python, a través de procedimientos

Procedimiento en Python

```
(envpy3-programacion) $ python ejemplo.py
Ingrese nombre de empleado: José
Ingrese apellido de empleado: Pineda
Ingrese año de nacimiento de empleado: 1983
Ingrese 1 si desea seguir en el ciclo: 2
Ingrese nombre de empleado: María
Ingrese apellido de empleado: Suárez
Ingrese año de nacimiento de empleado: 1985
Ingrese 1 si desea seguir en el ciclo: 1
Nombre: José
Apellido: Pineda
Edad: 42
-----|||-----Nombre: María
Apellido: Suárez
Edad: 40
-----|||-----
```

Nota. Elizalde, R., 2025.

La figura anterior describe la ejecución de un script de Python, que hace uso de funciones (procedimientos) que no retornan ningún valor.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.

3.3. Funciones que regresan valor

Las funciones que retornan valores se caracterizan por devolver un valor en función de las necesidades que se requieran, el tipo de dato a devolver puede variar entre tipos de datos básicos, estructuras de datos o tipos de datos propios.

En la Figura 20, se describe una estructura base de una función en miniespecificación. Donde:

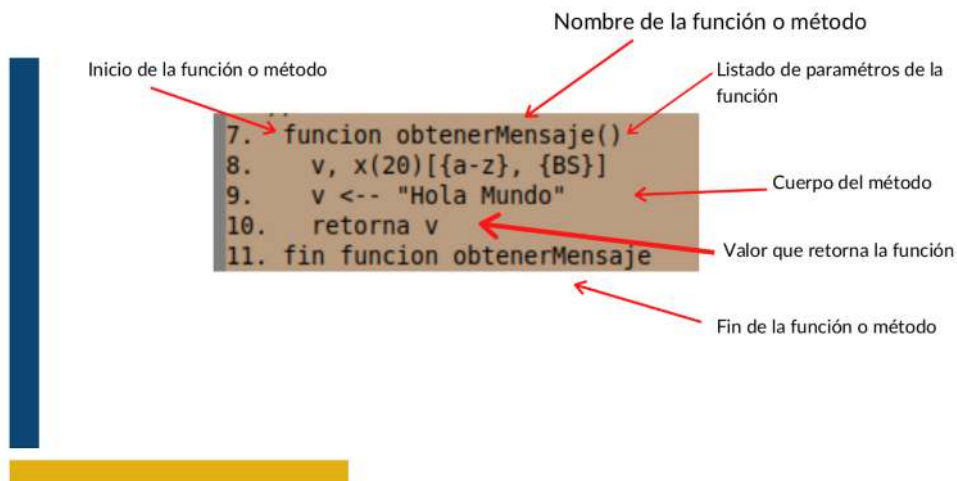
1. En el inicio de la función se utiliza la palabra clave **funcion** para definirla.
2. Se indica su nombre (obtenerMensaje), lo que permite llamarla dentro del programa
3. Los paréntesis () sugieren que la función podría recibir parámetros, aunque en este caso están vacíos.
4. Cuerpo de la función, contiene el bloque de instrucciones que se ejecutarán cuando la función sea invocada. Se observa que se asigna un valor a la variable v, que almacena la cadena "Hola Mundo".
5. Valor que retorna la función, se utiliza la palabra clave retorna para indicar que la función devuelve un valor. En este caso, retorna v, lo que significa que cuando se llame a obtenerMensaje(), la función devolverá el contenido de v;
6. Fin de la función;
7. Se marca con fin funcion obtenerMensaje, indicando el cierre del bloque de código de la función.



Figura 20

Ejemplo de función en miniespecificación

Función en Miniespecificación



Nota. Elizalde, R., 2025.

En la figura anterior se describen las características a considerar al momento de implementar una función en miniespecificaciones.

La forma estándar de crear una función que retorna un valor en Python de la siguiente manera:

```
def obtener_mensaje():  
    // instrucciones  
    // instrucciones  
    return "Saludo"
```

Se aplican los mismos principios analizados en el punto anterior, agregando la palabra clave `return` para especificar el valor que será retornado. Es fundamental asegurarse de que, al invocar la función, el valor devuelto se almacene en una variable cuyo tipo de dato sea compatible con el que la función retorna.

- Ejemplo 1: Genera una solución que permita ingresar por teclado datos de empleados como: nombres, apellidos, años de nacimiento; el usuario decide hasta cuando ingresar valores, cuando se termine el ciclo, debe presentarse el listado de empleados, donde se indique la edad del empleado con base al año de nacimiento. Se debe crear una función que devuelva un listado con los valores ingresados.

```
# Importa el módulo datetime para manejar fechas y obtener el año actual
import datetime
```

```
# Define una función en Python que devuelve un valor
def generar_reporte():
```

```
    """
    Función que recopila datos de empleados, calcula su edad
    en función del año de nacimiento ingresado y retorna una lista con los datos
    formateados.
    """
```

```
    lista_final = [] # Lista que almacenará los datos ingresados por el usuario
    bandera = True # Controla la ejecución del bucle while para permitir
    múltiples ingresos de datos
```

```
# Inicia un bucle para ingresar información de empleados
while bandera:
```

```
    # Captura el nombre y apellido del empleado
    nombre = input("Ingrese nombre de empleado: ")
    apellido = input("Ingrese apellido de empleado: ")
```

```
    # Captura y convierte la entrada del año de nacimiento a entero
    anio_nacimiento = input("Ingrese año de nacimiento de empleado: ")
    anio_nacimiento = int(anio_nacimiento) # Convierte la entrada a entero
```



```
# Calcula la edad utilizando datetime.datetime.now().year
```

```
# Obtiene el año actual y se resta el año de nacimiento
```

```
edad = datetime.datetime.now().year - anio_nacimiento
```

```
# Formatea los datos del empleado en un string y lo agrega a la lista
```

```
datos_empleado = f"Nombre: {nombre}\n" \
```

```
    f"Apellido: {apellido}\n" \
```

```
    f"Edad: {edad}\n" \
```

```
    f"-----|||-----\n"
```

```
lista_final.append(datos_empleado)
```

```
# Solicita al usuario si desea continuar o salir del ciclo
```

```
opcion = input("Ingrese 1 si desea salir del ciclo: ")
```

```
opcion = int(opcion) # Convierte la entrada a entero
```

```
# Si el usuario ingresa "1", la bandera se cambia a False y el ciclo termina
```

```
if opcion == 1:
```

```
    bandera = False
```

```
# Retorna la lista con todos los datos ingresados
```

```
return lista_final
```

Esta función utiliza "return", lo que la convierte en una función que retorna un valor.

En este caso, retorna una lista con la información de los empleados.

Al ser una función con retorno, su resultado debe ser almacenado en una variable al momento de llamarla.

```
# Define el punto de entrada principal del script
```

```
# Garantiza que el código dentro de este bloque solo se ejecute si el script
```

```
# es ejecutado directamente y no si se importa como módulo en otro archivo
```

```
if __name__ == "__main__":
```

```
    #
```

```
    # Llamado a la función "generar_reporte"
```



La función devuelve una lista, por lo que es necesario almacenarla en una variable.

```
lista = generar_reporte()
```

Se recorre la lista usando un ciclo for para imprimir cada elemento.

Cada elemento de la lista corresponde a los datos de un empleado formateados.

```
for i in lista:
```

```
    print(i)
```

La ejecución del código se lo puede observar en la Figura 21

Figura 21

Uso de Python con manejo de funciones

Función en Python

```
(envpy3-programacion) $ python ejemplo.py
Ingrese nombre de empleado: Andrés
Ingrese apellido de empleado: Jara
Ingrese año de nacimiento de empleado: 1970
Ingrese 1 si desea salir del ciclo: 2
Ingrese nombre de empleado: Ana
Ingrese apellido de empleado: Bernuncio
Ingrese año de nacimiento de empleado: 1980
Ingrese 1 si desea salir del ciclo: 1
Nombre: Andrés
Apellido: Jara
Edad: 55
-----|||-----
Nombre: Ana
Apellido: Bernuncio
Edad: 45
-----|||-----
```

Nota. Elizalde, R., 2025.

En la figura anterior se muestra la salida de un script de Python, que hace uso de funciones.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Revisar con atención los conceptos de funciones y procedimientos, dentro del capítulo tres, [“Programación Modular”, del texto “Lógica de programación con PSeInt: enfoque práctico”](#). Aunque este recurso no presenta implementaciones específicas, su explicación didáctica facilita la comprensión de la modularización en la programación, permitiendo a los estudiantes diferenciar el uso de funciones y procedimientos en el diseño de algoritmos estructurados. Se recomienda tomar notas sobre la diferencia entre procedimientos (que no retornan valores) y funciones (que sí devuelven un resultado), así como reflexionar sobre su utilidad en la optimización y organización del código.
2. Revisar la solución a la siguiente problemática y ejecutarla en su entorno local.

Un programa que tiene como objetivo permitir el registro de información sobre estudiantes y docentes de una institución educativa. Para ello, se implementan dos funciones: `obtener_estudiantes()` y `obtener_docentes()`, las cuales permiten al usuario ingresar los datos de cada grupo de manera iterativa. En el caso de los estudiantes, se solicita su nombre, apellidos, correo electrónico y edad, mientras que para los docentes se registra su nombre, apellidos y ciudad de residencia. La cantidad de registros a ingresar es determinada por el usuario, quien deberá indicar cuántos estudiantes o docentes desea agregar. El programa emplea listas para almacenar la información ingresada, lo que permite organizar y procesar los datos de manera estructurada. Una vez finalizado el ingreso, se genera un reporte formateado que muestra la lista de estudiantes o docentes junto con sus respectivas características. Además, el sistema usa un menú interactivo, en el cual el usuario debe seleccionar si desea registrar estudiantes o docentes.




```

def obtener_estudiantes():
    # Inicialización de variables
    reporte_estudiantes = "\nListado de Estudiantes\n" # Cadena para
    almacenar la lista de jugadores
    contador = 0 # Contador para numeración de jugadores
    bandera = True # Variable de control para el ciclo while

    elementos = input("Ingrese el número de estudiantes a generar: ") #
    Se solicita la edad del jugador
    elementos = int(elementos) # Conversión a entero
    contador = 1
    # Uso de listas para almacenar datos de los jugadores y sus edades
    lista_estudiantes = [] # Lista para almacenar la información de los
    jugadores (nombre, posición, edad, estatura)

    # Bucle para ingreso de datos
    while contador <= elementos:
        print("\nIngrese la información del estudiante:")
        nombre = input("Nombre del estudiante: ") # Se solicita el nombre
        del jugador
        apellido = input("Apellidos del estudiante: ") # Se solicita la
        posición del jugador
        correo = input("Correo del estudiante: ") # Se solicita la posición
        del jugador
        edad = input("Edad del estudiante: ") # Se solicita la edad del
        jugador
        edad = int(edad) # Conversión a entero

        # Uso de listas para almacenar la información de cada jugador
        lista = [nombre, apellido, correo, edad] # Se almacena la
        información en una lista
        lista_estudiantes.append(lista) # Se agrega la lista del jugador a la
        lista principal

```



```
    contador = contador + 1 # Se incrementa el contador para
numerar a los jugadores
```

```
    # Construcción del reporte de jugadores a partir de la lista de
jugadores
```

```
    for i in lista_estudiantes:
        reporte_estudiantes = f"{reporte_estudiantes}{i[0]} {i[1]} -correo:
{i[2]}-, edad:{i[3]}\n"
```

```
    # Impresión del reporte final
```

```
    print(reporte_estudiantes) # Se imprime el listado de jugadores
```

```
def obtener_docentes():
```

```
    # Inicialización de variables
```

```
    reporte = "\nListado de Docentes\n" # Cadena para almacenar la
lista de jugadores
```

```
    contador = 0 # Contador para numeración de jugadores
```

```
    bandera = True # Variable de control para el ciclo while
```

```
    elementos = input("Ingrese el número de docentes a generar: ") # Se
solicita la edad del jugador
```

```
    elementos = int(elementos) # Conversión a entero
```

```
    contador = 1
```

```
    # Uso de listas para almacenar datos de los jugadores y sus edades
```

```
    lista_docentes = [] # Lista para almacenar la información de los
jugadores (nombre, posición, edad, estatura)
```

```
    # Bucle para ingreso de datos
```

```
    while contador <= elementos:
```

```
        print("\nIngrese la información del docente:")
```

```
        nombre = input("Nombre del docente: ") # Se solicita el nombre
```



```

del jugador
    apellido = input("Apellidos del docente: ") # Se solicita la posición
del jugador
    ciudad = input("Ciudad de residencia: ") # Se solicita la edad del
jugador

# Uso de listas para almacenar la información de cada jugador
lista = [nombre, apellido, ciudad] # Se almacena la información en
una lista
lista_docentes.append(lista) # Se agrega la lista del jugador a la
lista principal

    contador = contador + 1 # Se incrementa el contador para
numerar a los jugadores

# Construcción del reporte de jugadores a partir de la lista de
jugadores
for i in lista_docentes:
    reporte = f"{reporte}{i[0]} {i[1]} -ciudad de residencia:{i[2]}\n"

# Impresión del reporte final
print(reporte) # Se imprime el listado de jugadores

if __name__ == "__main__":

    mensaje = "Menú\nIngrese 1 para generar estudiantes; 2 para
generar docentes: "
    opcion = input(mensaje)
    opcion = int(opcion)
    if opcion == 1:
        obtener_estudiantes()
    else:
        if opcion == 2:

```



```
    obtener_docentes()
else:
    print("Opción correcta")
```

El código completo se lo puede obtener en el [repositorio de GitHub](#).

Este ejercicio refuerza el uso de bucles while, estructuras de control, manipulación de listas y construcción de cadenas de acumulación para la presentación de datos. Su implementación permite a los estudiantes comprender cómo gestionar información en programas que requieren el ingreso y procesamiento de múltiples registros de manera flexible.

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 10

En la Semana 10, se continúa con el estudio del manejo avanzado de datos y estructuras, centrándose en el paso de parámetros en funciones para enviar y procesar información dentro de un programa. Aprenderán la diferencia entre variables locales y globales, comprendiendo cómo influyen en el alcance y comportamiento de las funciones. Se realizarán ejercicios prácticos donde aplicarán estos conceptos para estructurar mejor sus programas y mejorar la reutilización del código en Python.

Unidad 3. Manejo avanzado de datos y estructuras

3.4. Paso de parámetros en funciones

En esta semana se continúa el estudio del uso de funciones en las soluciones de problemas computacionales. Para ello, se solicita realizar una lectura comprensiva del capítulo 7 del texto [Python paso a paso](#).

El texto resalta que, una vez comprendido el funcionamiento y las diferencias entre funciones que devuelven un valor y aquellas que no lo hacen, el siguiente paso importante en el estudio de la programación es analizar la capacidad de



las funciones para recibir parámetros o argumentos. Esta característica es fundamental, ya que permite que las funciones sean más versátiles, reutilizables y adaptables a diferentes situaciones dentro de un programa.

Los parámetros son el conjunto de variables que están expresadas en la definición/firma/cabecera de la función, van entre paréntesis; las variables se las considera como internas o locales para la función.

Ejemplo:

```
def presentar_reporte(nombre): # "nombre" es un parámetro
    print(f"Hola, {nombre}!")
```

Los argumentos se usan al momento de llamar o invocar a una función, que debe tener correspondencia con la función a la que se quiere hacer referencia.

Ejemplo:

```
presentar_reporte("José Luis") # "José Luis" es un argumento
```

Una función puede definirse sin parámetros o incluir uno o varios de ellos. En caso de contar con múltiples parámetros, estos deben escribirse dentro de los paréntesis y separados por comas. Al momento de llamar a la función, es necesario proporcionar los argumentos respetando el mismo orden en el que fueron declarados para que sean utilizados correctamente.

- Ejemplo 1: construir una solución que permita a través de una función generar un reporte de notas a partir de la información de un estudiante, incluyendo nombre, apellido, edad y una lista de notas; calcula el promedio de las notas y muestra el informe formateado.

```
# Define la función (procedimiento) con 4 parámetros
def generar_reporte(nombre, apellido, edad, lista_notas):
```

```
    """
```

```
    Función que genera un reporte de notas a partir de la información
```





de un estudiante, incluyendo nombre, apellido, edad y una lista de notas.
Calcula el promedio de las notas y muestra el informe formateado.

"""

```
# Calcula la suma total de las notas contenidas en lista_notas
suma_notas = sum(lista_notas)
```

```
# Calcula el promedio dividiendo la suma entre la cantidad de notas
promedio_notas = suma_notas / len(lista_notas)
```

```
# Se genera el reporte en función de las variables recibidas como
parámetros
```

```
mensaje_final = "Reporte de Notas\n"
```

```
mensaje_final = f"{mensaje_final}Nombres y Apellidos: {nombre} {apellido}
\n" \
```

```
    f"Edad: {edad}\n\n" \
```

```
    f"Notas:\n"
```

```
# Se recorre la lista de notas para agregarlas al reporte, usando un bucle for
for i in lista_notas:
```

```
    mensaje_final = f"{mensaje_final}{i}\n"
```

```
# Se agrega el promedio final al reporte, formateado con 2 decimales
mensaje_final = f"{mensaje_final}Promedio: {promedio_notas:.2f}"
```

```
# Se muestra el mensaje final con los datos del estudiante
print(mensaje_final)
```

```
# Define el punto de entrada principal del script
```

```
# Garantiza que el código dentro de este bloque solo se ejecute si el script
```

```
# es ejecutado directamente y no si se importa como módulo en otro archivo
```

```
if __name__ == "__main__":
```

```
    #
```

```
    # Se crea una lista con las notas del estudiante
```

```
mi_lista_notas = [10, 9, 8.5]
```

```
# Se invoca la función generar_reporte enviando argumentos específicos:
```

```
# 1. "María Elizabeth" -> Nombre del estudiante
```

```
# 2. "García López" -> Apellido del estudiante
```

```
# 3. 25 -> Edad del estudiante
```

```
# 4. mi_lista_notas -> Lista con las notas obtenidas
```

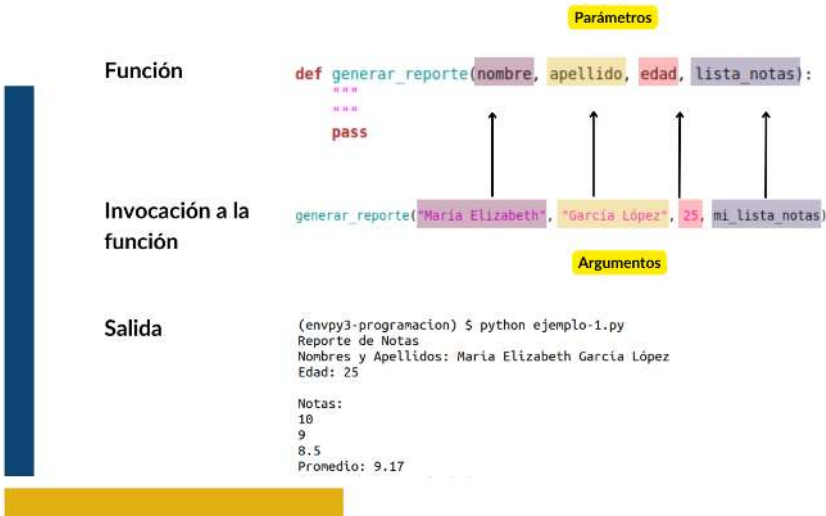
```
generar_reporte("María Elizabeth", "García López", 25, mi_lista_notas)
```

En la Figura 22, se puede revisar en la primera sección, se muestra la definición de la función `generar_reporte`, donde se destacan los parámetros (nombre, apellido, edad, `lista_notas`). Estos actúan como variables dentro de la función, esperando recibir valores cuando la función sea invocada. En la segunda sección, se observa la invocación de la función, donde se pasan argumentos reales ("María Elizabeth", "García López", 25, `mi_lista_notas`). Estos argumentos reemplazan a los parámetros cuando la función es llamada, asegurando que la función trabaje con los valores proporcionados. Y finalmente, en la tercera sección, se muestra la salida del programa, donde se genera un reporte con los datos proporcionados a la función.



Figura 22

Uso de funciones en Python con parámetros / argumentos



Nota. Elizalde, R., 2025.

La figura anterior muestra el proceso de invocar a una función en Python, diferenciado entre parámetros y argumentos.

En Python, los parámetros de una función pueden recibir valores por defecto al momento de ser declarados en la cabecera. Esto se logra mediante el uso del operador "=", permitiendo que dichos parámetros asuman un valor predefinido si no se les proporciona un argumento durante la invocación de la función. Cuando se llama a una función que contiene parámetros con valores por defecto, el usuario tiene la opción de enviar o no un argumento. Si se proporciona un argumento, este reemplaza el valor por defecto; de lo contrario, la función utiliza el valor establecido en la definición. Además, facilita la creación de funciones con comportamientos adaptativos, donde se pueden personalizar solo los parámetros necesarios sin afectar la estructura general de la función.

- Ejemplo 2: Se realiza una solución a la problemática anterior, con el uso de parámetros con valores por defecto


```

# Define la función (procedimiento) con 4 parámetros
# Se asigna un valor por defecto al parámetro "lista_notas"
def generar_reporte(nombre, apellido, edad, lista_notas=[10, 10, 5]):
    """
    Función que genera un reporte de notas a partir de la información
    de un estudiante, incluyendo nombre, apellido, edad y una lista de notas.
    Calcula el promedio de las notas y muestra el informe formateado.
    """

    # Calcula la suma total de las notas contenidas en lista_notas
    suma_notas = sum(lista_notas)

    # Calcula el promedio dividiendo la suma entre la cantidad de notas
    promedio_notas = suma_notas / len(lista_notas)

    # Se genera el reporte en función de las variables recibidas como
    parámetros
    mensaje_final = "Reporte de Notas\n"
    mensaje_final = f"{mensaje_final}Nombres y Apellidos: {nombre} {apellido}
\n" \
        f"Edad: {edad}\n\n" \
        f"Notas:\n"

    # Se recorre la lista de notas para agregarlas al reporte, usando un bucle for
    for i in lista_notas:
        mensaje_final = f"{mensaje_final}{i}\n"

    # Se agrega el promedio final al reporte, formateado con 2 decimales
    mensaje_final = f"{mensaje_final}Promedio: {promedio_notas:.2f}"

    # Se muestra el mensaje final con los datos del estudiante
    print(mensaje_final)

```



```

# Define el punto de entrada principal del script
# Garantiza que el código dentro de este bloque solo se ejecute si el script
# es ejecutado directamente y no si se importa como módulo en otro archivo
if __name__ == "__main__":
    #
    # Se crea una lista con las notas del estudiante
    mi_lista_notas = [10, 9, 8.5]

    # Se invoca la función "generar_reporte" enviando 4 argumentos explícitos
    # En este caso, el parámetro "lista_notas" recibe un argumento
    # personalizado
    generar_reporte("María Elizabeth", "García López", 25, mi_lista_notas)

    print("-----")

    # Se invoca la función sin enviar un argumento para "lista_notas"
    # Como el parámetro tiene un valor por defecto ([10, 10, 5]),
    # la función utilizará esa lista de manera automática
    generar_reporte("Paul José", "Benítez Salazar", 20)

```

En la Figura 23, se puede revisar la salida del ejemplo descrito.



Figura 23

Uso de Python a través de funciones con parámetros y valores por defecto

```
Uso de funciones en Python con parámetros con valores por defecto

(envpy3-programacion) $ python ejemplo-2.py
Reporte de Notas
Nombres y Apellidos: María Elizabeth García López
Edad: 25

Notas:
10
9
8.5
Promedio: 9.17
-----
Reporte de Notas
Nombres y Apellidos: Paul José Benítez Salazar
Edad: 20

Notas:
10
10
5
Promedio: 8.33
```

Nota. Elizalde, R., 2025.

La figura anterior muestra la salida de un script de Python, donde se hace uso de funciones con parámetros y valores por defecto

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.

3.5. Variables locales y globales

Previo al estudio de variables de ámbito local y global, es importante realizar una lectura comprensiva del [capítulo 17 en lo relacionado al Ámbito de las Variables](#). El recurso indica que el ámbito de la variable es la parte del código desde donde está accesible para lectura o escritura.

Las variables que han sido creadas fuera de las líneas de código de cualquiera de las funciones, toman el nombre de globales; y las variables que están dentro de las estructuras, se denominan locales.

Se presentan los siguientes casos:

Caso 1:

```
# Variable global declarada fuera de cualquier función
frase = "Ecuador país de Sudamérica" # Variable global

def presentar_mensaje():
    """
    Función que define una variable local con el mismo nombre que la global.
    Esta variable local solo existe dentro de la función.
    """
    # Se define una variable local con el mismo nombre que la global
    frase = "Ecuador país con cuatro regiones naturales" # Variable local
    print(frase) # Imprime la variable local dentro de la función

# Punto de entrada principal del script
if __name__ == "__main__":

    # Se imprime la variable global
    print(frase) # Salida: "Ecuador país de Sudamérica"

    # Se llama a la función que tiene una variable local con el mismo nombre
    presentar_mensaje() # Salida: "Ecuador país con cuatro regiones naturales"

    # Se vuelve a imprimir la variable global, su valor no ha cambiado
    print(frase) # Salida: "Ecuador país de Sudamérica"
```

En el código anterior, se define una variable global

```
frase = "Ecuador país de Sudamérica"
```

Antes de cualquier función, lo que permite su acceso en todo el archivo/script.

Dentro de la función `presentar_mensaje()`, se declara una variable local con el mismo nombre (`frase = "Ecuador país con cuatro regiones naturales"`), pero esta solo existe dentro del ámbito de la función.



Al ejecutar el archivo, se imprime la variable frase antes de llamar a la función, se muestra el valor de la variable global; cuando la función `presentar_mensaje()` se ejecuta, imprime la variable local, sin afectar la global; y finalmente, al imprimir nuevamente frase fuera de la función, se sigue obteniendo el valor global original, lo que demuestra que las variables locales no modifican las variables globales, aunque tengan el mismo nombre.

Caso 2:

```
# Variable global declarada fuera de cualquier función
frase = "Ecuador país de Sudamérica" # Variable global

def presentar_mensaje():
    """
    Función que imprime el valor de la variable global 'frase'.
    Como no se define una variable local con el mismo nombre dentro de la
    función,
    Python automáticamente usa la variable global.
    """

    print(frase) # Se accede y muestra el valor de la variable global

# Punto de entrada principal del script
if __name__ == "__main__":

    # Se imprime la variable global antes de llamar a la función
    print(frase) # Salida: "Ecuador país de Sudamérica"

    # Se invoca la función, que imprimirá la variable global
    presentar_mensaje() # Salida: "Ecuador país de Sudamérica"

    # Se vuelve a imprimir la variable global, sin haber sido modificada
    print(frase) # Salida: "Ecuador país de Sudamérica"
```

Este caso demuestra cómo una variable global puede ser accedida dentro de una función sin necesidad de volver a declararla. La variable frase se declara a nivel global y, como no se define una variable local con el mismo nombre



dentro de `presentar_mensaje()`, Python automáticamente busca y utiliza la variable global. Antes de ejecutar la función, `print(frase)` muestra el valor global, luego la función imprime ese mismo valor y, finalmente, `print(frase)` fuera de la función sigue mostrando el mismo contenido, ya que la variable no ha sido modificada dentro de la función.

Caso 3:

Uso de la palabra reservada `global`.

```
# Variable global declarada fuera de cualquier función
frase = "Ecuador país de Sudamérica" # Variable global
```

```
def presentar_mensaje():
```

```
    """
```

```
    Función que modifica la variable global 'frase'.
```

```
    Para cambiar su valor dentro de la función, se usa la palabra clave 'global',
```

```
    lo que permite modificar la variable global en lugar de crear una nueva
variable local.
```

```
    """
```

```
    global frase # Se indica que se trabajará con la variable global
```

```
    frase = "Ecuador país donde el mundo se une en un solo paisaje" # Se
modifica la variable global
```

```
    print(frase) # Se imprime el nuevo valor de la variable global
```

```
# Punto de entrada principal del script
```

```
if __name__ == "__main__":
```

```
    # Se imprime la variable global antes de llamar a la función
```

```
    print(frase) # Salida: "Ecuador país de Sudamérica"
```

```
    # Se invoca la función, que modificará la variable global
```

```
    presentar_mensaje() # Salida: "Ecuador país donde el mundo se une en un
solo paisaje"
```



```
# Se vuelve a imprimir la variable global, que ahora ha cambiado su valor
print(frase) # Salida: "Ecuador país donde el mundo se une en un solo
paisaje"
```

En esta ejemplificación se indica cómo una variable global puede ser modificada dentro de una función utilizando la palabra clave **global**. La variable `frase` se declara a nivel global con el valor "Ecuador país de Sudamérica". Dentro de la función `presentar_mensaje()`, se usa `global frase`, lo que permite modificar la variable global en lugar de crear una nueva variable local. Antes de ejecutar la función, `print(frase)` muestra el valor original de la variable global. Luego, cuando se llama a `presentar_mensaje()`, la variable global se modifica y se imprime su nuevo valor. Finalmente, al imprimir `frase` nuevamente fuera de la función, se observa que su valor ha cambiado permanentemente.

En la Figura 24, se puede observar la salida de los tres casos explicativos del uso de variables globales y locales en Python; resaltar que el caso tres, es el menos recomendado, porque puede llegar a genera código que no sea plenamente mantenible.

Figura 24

Uso de funciones en Python con variables globales y locales

Uso de funciones en Python con variables globales y locales

```
(envpy3-programacion) $ python ejemplo-1.py
Ecuador país de Sudamérica
Ecuador país con cuatro regiones naturales
Ecuador país de Sudamérica
(envpy3-programacion) $ python ejemplo-2.py
Ecuador país de Sudamérica
Ecuador país de Sudamérica
Ecuador país de Sudamérica
(envpy3-programacion) $ python ejemplo-3.py
Ecuador país de Sudamérica
Ecuador país donde el mundo se une en un solo paisaje
Ecuador país donde el mundo se une en un solo paisaje
(envpy3-programacion) $
```

Nota. Elizalde, R., 2025.

La figura anterior muestra la salida de un código en Python, que está usando variables globales y locales.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Revisar con atención los conceptos de [paso de parámetros, variables globales y locales, dentro del capítulo tres, “Programación Modular”, del texto “Lógica de Programación con PSeInt: enfoque práctico”](#). Aunque los ejemplos en el recurso están desarrollados en PSeInt, los conceptos explicados son completamente aplicables a cualquier lenguaje de programación, incluyendo Python. Se recomienda analizar la diferencia entre paso de parámetros por valor y por referencia, así como el alcance de variables globales y locales dentro de un programa. Además, se sugiere replicar los ejemplos en Python, adaptándolos a la sintaxis del lenguaje para fortalecer la comprensión. Finalmente, analizar sobre buenas prácticas en el uso de variables globales y su impacto en la estructura del código ayudará a desarrollar programas más organizados y eficientes.
2. Estudiar y sintetizar la información presentada en la sección [Funciones del capítulo tres del texto Introducción a la programación](#). Esta actividad permitirá al estudiante comprender la importancia de las funciones para modularizar el código, reutilizar instrucciones y mejorar la organización en el desarrollo de software. Se recomienda analizar la estructura de una función, identificar los elementos clave en su declaración y explorar ejemplos sobre cómo llamarlas dentro de un programa y paso de argumentos. Además, se sugiere replicar y modificar los ejercicios del recurso en un entorno de programación. Finalmente, reflexionar sobre la utilidad de las funciones en la



resolución de problemas y su aplicación en programas más complejos contribuirá a consolidar estos conocimientos.

3. Revisar la solución de la siguiente problemática.

Un programa en Python que permita gestionar la información de estudiantes y docentes de una institución educativa. Para ello, el programa deberá solicitar al usuario que ingrese un número determinado de estudiantes y docentes, capturando su información personal y almacenándola para su posterior presentación en un reporte estructurado. El sistema deberá permitir el registro de estudiantes, solicitando su nombre, apellidos, correo electrónico y edad. Una vez ingresados los datos, estos deberán almacenarse en una lista para su procesamiento y visualización en un reporte detallado. De manera similar, se deberá permitir el registro de docentes, solicitando su nombre, apellidos y ciudad de residencia, almacenando la información en otra lista para su posterior presentación.

Para mejorar la interacción del usuario, el programa deberá contar con un menú interactivo que le permita elegir entre tres opciones principales: registrar estudiantes, registrar docentes o registrar automáticamente un estudiante y un docente. El usuario podrá indicar cuántos estudiantes o docentes desea registrar en cada ejecución del programa. Además, el menú deberá mantenerse en ejecución mediante un bucle while, permitiendo realizar múltiples registros hasta que el usuario decida salir del sistema.

El diseño del programa deberá seguir una estructura donde se usen funciones para manejar el registro de estudiantes y docentes. Asimismo, se deberá hacer uso de listas para almacenar y manipular los datos ingresados. La presentación de la información deberá realizarse mediante un reporte formateado, utilizando cadenas de acumulación y el método f-string para una salida clara y organizada.



```

def obtener_estudiantes(numero=1):
    # Inicialización de variables
    reporte_estudiantes = "\nListado de Estudiantes\n" # Cadena para
    almacenar la lista de estudiantes
    contador = 0 # Contador para numeración de estudiantes
    bandera = True # Variable de control para el ciclo while

    contador = 1 # Se inicializa el contador en 1

    # Uso de lista para almacenar la información de los estudiantes
    lista_estudiantes = [] # Lista donde cada estudiante se almacenará
    como una sublista

    # Bucle para ingreso de datos de los estudiantes
    while contador <= numero: # Se ejecuta hasta que se ingresen la
    cantidad de estudiantes solicitados
        print("\nIngrese la información del estudiante:")
        nombre = input("Nombre del estudiante: ") # Se solicita el nombre
        del estudiante
        apellido = input("Apellidos del estudiante: ") # Se solicita el
        apellido del estudiante
        correo = input("Correo del estudiante: ") # Se solicita el correo
        electrónico del estudiante
        edad = input("Edad del estudiante: ") # Se solicita la edad del
        estudiante
        edad = int(edad) # Conversión de la edad a entero

        # Se almacena la información en una lista temporal
        lista = [nombre, apellido, correo, edad]
        lista_estudiantes.append(lista) # Se agrega la lista del estudiante
        a la lista principal

        contador = contador + 1 # Se incrementa el contador para
        controlar el número de registros

```



```
# Construcción del reporte de estudiantes a partir de la lista de
estudiantes
```

```
for i in lista_estudiantes:
    reporte_estudiantes = f"{reporte_estudiantes}{i[0]} {i[1]} -correo:
{i[2]}-, edad:{i[3]}\n"
```

```
# Impresión del reporte final de estudiantes
```

```
print(reporte_estudiantes) # Se muestra el listado de estudiantes
```

```
def obtener_docentes(numero=1):
```

```
    # Inicialización de variables
```

```
    reporte = "\nListado de Docentes\n" # Cadena para almacenar la
lista de docentes
```

```
    contador = 0 # Contador para numeración de docentes
```

```
    bandera = True # Variable de control para el ciclo while
```

```
    contador = 1 # Se inicializa el contador en 1
```

```
    # Uso de lista para almacenar la información de los docentes
```

```
    lista_docentes = [] # Lista donde cada docente se almacenará como
una sublista
```

```
    # Bucle para ingreso de datos de los docentes
```

```
    while contador <= numero: # Se ejecuta hasta que se ingresen la
cantidad de docentes solicitados
```

```
        print("\nIngresa la información del docente:")
```

```
        nombre = input("Nombre del docente: ") # Se solicita el nombre
del docente
```

```
        apellido = input("Apellidos del docente: ") # Se solicita el apellido
del docente
```

```
        ciudad = input("Ciudad de residencia: ") # Se solicita la ciudad de
residencia del docente
```



```
# Se almacena la información en una lista temporal
lista = [nombre, apellido, ciudad]
lista_docentes.append(lista) # Se agrega la lista del docente a la
lista principal
```

```
contador = contador + 1 # Se incrementa el contador para
controlar el número de registros
```

```
# Construcción del reporte de docentes a partir de la lista de
docentes
```

```
for i in lista_docentes:
```

```
    reporte = f"{reporte}{i[0]} {i[1]} -ciudad de residencia:{i[2]}\n"
```

```
# Impresión del reporte final de docentes
```

```
print(reporte) # Se muestra el listado de docentes
```

```
# Menú principal del programa
```

```
if __name__ == "__main__":
```

```
    bandera = True # Variable de control para mantener el programa en
ejecución
```

```
    while bandera: # Se ejecuta hasta que el usuario decida salir
```

```
        mensaje = "Menú\nIngrese 1 para generar estudiantes; 2 para
generar docentes; 3 para generar un estudiante y un docente: "
```

```
        opcion = input(mensaje) # Se solicita al usuario que elija una
opción del menú
```

```
        opcion = int(opcion) # Conversión a entero para procesar la
elección
```

```
    if opcion == 1:
```

```
        # Solicitar el número de estudiantes a registrar
```

```
        elementos = input("Ingrese el número de estudiantes a generar:
")
```

```
        elementos = int(elementos) # Conversión a entero
```



```

    obtener_estudiantes(elementos) # Se llama a la función para
    registrar estudiantes
else:
    if opcion == 2:
        # Solicitar el número de docentes a registrar
        elementos = input("Ingrese el número de docentes a generar:
")
        elementos = int(elementos) # Conversión a entero
        obtener_docentes(elementos) # Se llama a la función para
        registrar docentes
    else:
        # Se ejecuta la opción por defecto, generando un estudiante y
        un docente
        # pues en cada función existe un parámetro con un valor por
        defecto
        obtener_estudiantes()
        obtener_docentes()

    # Se solicita al usuario si desea salir del programa
    salida = input("Ingrese 'si' para salir del proceso, caso contrario use
    cualquier valor para seguir: ")

    if salida.lower() == "si": # Si el usuario ingresa "si", se termina el
    bucle
    bandera = False # Se cambia la variable de control para salir del
    programa

```

La solución para ejecutar en su entorno local, la puede obtener en el siguiente [repositorio](#).

Este ejercicio permitirá a los estudiantes reforzar conceptos clave en programación, como el uso de funciones con parámetros predeterminados, la manipulación de listas, el manejo de estructuras de control como while y if-else, y la generación de reportes dinámicos.



4. Estimado estudiante, ha llegado el momento de evaluar su comprensión sobre los temas abordados en la Unidad 3: Manejo avanzado de datos y estructuras, en la que se profundiza en el uso de funciones en Python, desde su definición hasta su aplicación en la solución de problemas. Esta autoevaluación le permitirá identificar qué conceptos domina y en cuáles debería reforzar su aprendizaje.

Antes de realizar la autoevaluación, se recomienda seguir estas estrategias de estudio:

- Revisar cuidadosamente el contenido de cada tema, comprendiendo cómo las funciones estructuran el código de manera modular.
- Consultar los recursos educativos (documentos, ejemplos de código y bibliografía recomendada) para reforzar sus conocimientos.
- Practicar la implementación de funciones, tanto aquellas que devuelven valores como las que no.
- Analizar el uso del paso de parámetros en funciones, comprendiendo cómo afectan el flujo de información en un programa.
- Diferenciar el alcance de las variables locales y globales, reconociendo las mejores prácticas para su uso en la programación estructurada.

Recuerde que cada pregunta tiene una única respuesta correcta. Lea con atención cada enunciado y seleccione la opción que usted considere adecuada. ¡Mucho éxito en esta autoevaluación!



Autoevaluación 4

1. Responda verdadero o falso a la siguiente afirmación:

Un procedimiento no retorna valores, mientras que una función sí.

Respuestas:

- a. Verdadero. ()
- b. Falso. ()



2. ¿Cuál de las siguientes afirmaciones sobre parámetros en funciones de Python es correcta?

Respuestas:

- a. Todos los parámetros deben tener un valor por defecto.
- b. Se pueden definir funciones sin parámetros.
- c. Solo se permite un máximo de dos parámetros por función.

3. ¿Cuál de las siguientes opciones representa un parámetro con valor por defecto en Python?

Respuestas:

- a. `def saludo(nombre="Invitado"):`

`print(f"Hola, {nombre}")`

- b. `def saludo(nombre):`

`print(f"Hola, {nombre}")`

- c. `saludo("Carlos")`

4. ¿Cuál de las siguientes afirmaciones sobre el uso de global en Python es correcta?

Respuestas:

- a. Convierte todas las variables locales en globales automáticamente.
- b. Se usa solo en funciones recursivas.
- c. Permite modificar una variable global dentro de una función.

5. Analice el siguiente código:

`def funcion():`

`print("Hola, mundo")`



funcion()

Responda si la siguiente afirmación es verdadera o falsa:

Al momento de ejecutar el código, la salida es None.

Respuestas:

- a. Verdadero. ()
- b. Falso. ()

6. **¿Qué imprimirá el siguiente código en Python?**

```
def suma(a, b):
```

```
    return a + b
```

```
def doble_suma(x, y):
```

```
    return suma(x, y) * 2
```

```
print(doble_suma(3, 4))
```

Respuestas:

- a. 7.
- b. 14.
- c. 21.

7. **¿Qué imprimirá el siguiente código?**

```
def saludo(nombre, ciudad="Quito"):
```

```
    return f"Hola {nombre} desde {ciudad}"
```

```
print(saludo("Carlos", "Loja"))
```

```
print(saludo("Ana"))
```



Respuestas:

a. Hola, Carlos, desde Loja.

Hola, Ana, desde Quito.

b. Hola, Carlos, desde Quito.

Hola, Ana, desde Quito.

c. Hola, Carlos, desde Loja.

Hola, Ana.

8. **¿Cuál será la salida del siguiente código?**

```
def cuadrado(x):
```

```
    return x * x
```

```
def suma_cuadrados(a, b):
```

```
    return cuadrado(a) + cuadrado(b)
```

```
print(suma_cuadrados(2, 3))
```

Respuestas:

a. 13.

b. 10.

c. 12.

9. **¿Cuál será la salida de este código?**

```
def mostrar_mensaje(mensaje="Hola"):
```

```
    print(mensaje)
```

```
mostrar_mensaje("Buenos días")
```



`mostrar_mensaje()`

Respuestas:

a. Hola.

Buenos días.

b. Buenos días.

Hola.

c. Hola.

Hola.

10. **¿Cuál es la salida por pantalla del siguiente código en Python?**

```
def dividir(a, b):
```

```
    return a / b
```

```
def multiplicar(a, b):
```

```
    return a * b
```

```
def calcular(a, b, c):
```

```
    return dividir(multiplicar(a, b), c)
```

```
print(calcular(2, 3, 2))
```

Respuestas:

a. 6.

b. 3.

c. 2.

[Ir al solucionario](#)





Semana 11

La Semana 11, permitirá que los estudiantes exploren los conceptos básicos de la recursividad, una técnica en la que una función se llama a sí misma para resolver problemas de manera estructurada. Aprenderán cómo funciona la recursión en Python, identificando el caso base y el caso recursivo para evitar llamadas infinitas. Realizarán ejercicios prácticos con ejemplos de recursividad aplicada a problemas comunes, reforzando su comprensión sobre esta técnica y su utilidad en la programación.

Unidad 3. Manejo avanzado de datos y estructuras

3.6. Conceptos básicos de recursividad

En la presente semana, se estudiará un concepto que puede ayudar en la resolución de problemáticas. Para ello es importante recordar cómo aplicar el uso de funciones. Algunos autores definen la recursividad así:

Según Martín Villalba, Urquía Moraleda y Rubio González (2021):

“Una función f es recursiva si el cuerpo de la función contiene directa o indirectamente una invocación a la propia función f . La invocación puede aparecer directamente en el cuerpo de la función, o bien en el cuerpo de la función f puede invocarse a otras funciones que, a su vez, llamen a la función f .

En la función recursiva debe existir una condición de término o caso base, que establece cuándo el método debe dejar de invocarse recursivamente. También, debe producirse la convergencia a la condición de término, en las sucesivas invocaciones a la función recursiva (p. 346).

También, Marzal Varó, García Sevilla y Gracia Luengo (2016) indican que:



“Una función que se llama a sí misma, directa o indirectamente, es una función recursiva. La recursión es un potente concepto con el que se pueden expresar ciertos procedimientos de cálculo muy elegantemente. No obstante, al principio cuesta un poco entender las funciones recursivas... y un poco más diseñar nuestras propias funciones recursivas (p. 304).

Se solicita realizar una lectura analítica del capítulo 18 denominado Recursividad del texto [Python práctico: Herramientas, conceptos y técnicas](#), donde se explica con ejemplos el concepto de recursividad en lenguaje Python. Al igual que los autores anteriores, en el recurso manifiesta que la recursividad se la usa para dividir tareas en subtareas de menor complejidad con el objetivo de tratar de mejor forma y buscar una solución. Además, existen algunas características que deben cumplirse al momento de implementar la recursividad:

- Una función recursiva debe contener un caso base.
- Caso base, paso que permite terminar la función recursiva en determinado momento.
- Una función recursiva debe modificar su estado y dirigirse al caso base.
- Una función recursiva tiene la obligación de llamarse a sí mismo, de forma recursiva; se la denomina como caso recursivo.

Si no se tiene la precaución debida de tener un caso base, es posible que se convierta en un proceso infinito. En esencia, la recursión es una técnica usada en programación que permite que una función se llame a sí misma para resolver un problema en partes más pequeñas. Mientras la función siga dividiendo el problema en subproblemas más simples, continuará llamándose a sí misma en un proceso llamado paso recursivo. Este proceso sigue en activo hasta que se alcanza una condición especial llamada caso base, que es el punto donde la recursión se detiene. En ese momento, el caso base devuelve un valor a la función anterior en la cadena de llamadas, iniciando una secuencia de retornos hacia atrás hasta que el resultado final regresa a la función original que inició el proceso.



3.7. Ejemplos de recursividad

Ejemplo 1: Generar una solución que permite calcular el factorial de un número. Realizar dos soluciones, usando iteraciones y otra aplicando recursividad.

Solución 1:

```
def obtener_factorial(numero):
```

```
    """
```

Función que tiene un proceso de iteración para calcular el factorial de un número.

Parámetro:

- numero (int): Número para calcular su factorial.

Retorna:

- int: El factorial del número ingresado.

```
    """
```

```
    # Inicializar la variable del factorial en 1
```

```
    valor_factorial = 1
```

```
    # Bucle para calcular el factorial en orden descendente
```

```
    for i in range(numero, 0, -1):
```

```
        valor_factorial = valor_factorial * i # Multiplicar el valor actual del factorial
    por i
```

```
    return valor_factorial
```

```
if __name__ == '__main__':
```



```

resultado = obtener_factorial(5)
print(f"Valor Factorial (-- usando ciclos --): {resultado}")
"""

```

Salida: Valor Factorial (-- usando ciclos --): 120

Solución 2:

```

def obtener_factorial(numero):
    """

```

Función recursiva para calcular el factorial de un número.

Parámetro:

- numero (int): Número para calcular su factorial.

Retorna:

- int: El factorial del número ingresado.

```

if numero <= 1:

```

Caso base: Cuando el número es 1 o menor, se detiene la recursión

y retorna 1, ya que $1! = 1$ y $0! = 1$.

```

    return 1

```

```

else:

```

Paso recursivo: Se multiplica el número actual por el factorial

del número anterior (numero - 1), llamando nuevamente a la función.

```

    return numero * obtener_factorial(numero - 1)

```

```

# Punto de entrada principal del script

```

```

if __name__ == '__main__':

```

Llamada a la función recursiva con el número 5

```

    resultado = obtener_factorial(5)

```

```

# Se imprime el resultado del cálculo factorial

```



```
print(f"Valor Factorial (-- usando recursividad --): {resultado}")  
"""
```

Salida: Valor Factorial (-- usando ciclos --): 120

"""

Donde,

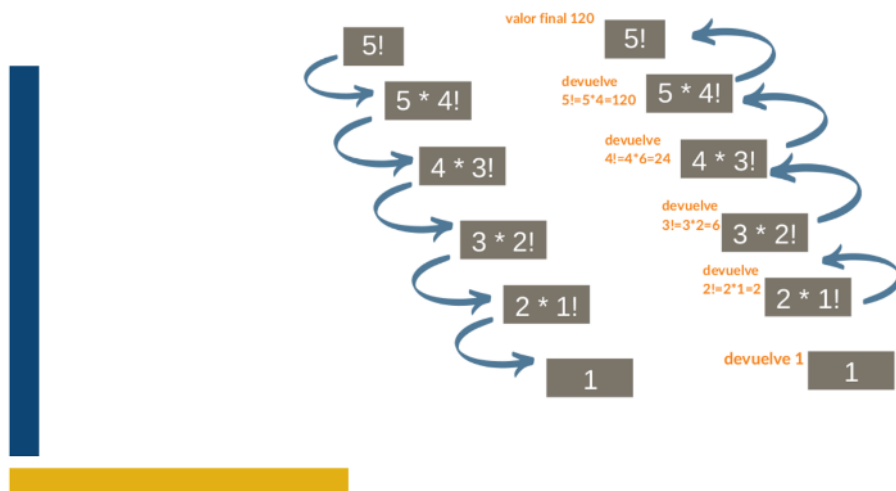
- El caso base (if numero <= 1:), establece la condición de parada o terminación para la recursión. ¡Cuando el número es 1 o 0, se retorna 1, ya que matemáticamente el 0! ¡Es igual a 1 y el 1! Es igual a 1.
- El paso recursivo (return numero * obtener_factorial(numero - 1)), se multiplica el número actual por el resultado de la función aplicada al número anterior (numero - 1). Esto genera múltiples llamadas recursivas hasta llegar al caso base.
- Hacer uso de la función obtener_factorial(5), aquí empieza el proceso de llamada:
 - 5 * obtener_factorial(4)
 - 4 * obtener_factorial(3)
 - 3 * obtener_factorial(2)
 - 2 * obtener_factorial(1) # En este punto se alcanza el caso base, retorna 1.
- ¡Finalmente , las funciones van devolviendo valores y multiplicando en orden inverso hasta obtener $5! = 120$, ver Figura 25.
- Salida: Valor Factorial (-- usando recursividad --): 120.



Figura 25

Proceso Recursivo.

Ejemplo de Recursividad



Nota. Adaptado de Deitel, H. M. (2016). Java: como programar

En la figura anterior se muestra el proceso recursivo, que se lo puede implementar en cualquier lenguaje de alto nivel.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.

Ejemplo 2:

- Generar una solución Python que permita sumar los elementos de una lista, haciendo uso de un proceso recursivo.

Solución:

```
# función recursiva
def suma_recursiva(lista_valores, numero_elementos):
    """
```

Parámetro:

- lista_valores: Lista que contiene los números para sumarlos.

- numero_elementos: el tamaño de la lista, que se irá reduciendo en cada llamada recursiva.

Retorna:

- int: la suma de los elementos del arreglo o lista.

"""

```
if numero_elementos == 1:
```

```
    # caso base
```

```
    # El caso base es la condición que detiene la recursión.
```

```
    # Cuando se llega al último elemento de la lista, se retorna ese valor.
```

```
    # Sin el caso base, la función seguiría llamándose a sí misma infinitamente.
```

```
    return lista_valores[0]
```

```
else:
```

```
    # proceso/paso recursivo
```

```
    # La función se llama a sí misma con el mismo parámetro de lista_valores,
```

```
    # pero con numero_elementos reducido en 1. Esto reduce el tamaño del
```

problema

```
    # en cada llamada hasta llegar al caso base.
```

```
    # En cada paso, se agrega el valor actual de lista_valores[numero_elementos -
```

1]

```
    # al resultado de la llamada recursiva que suma los elementos anteriores.
```

```
    return lista_valores[numero_elementos - 1] + \
```

```
        suma_recursiva(lista_valores, numero_elementos - 1)
```

```
def valores_cadena(lista_valores):
```

"""

Parámetro:

- lista_valores: lista de elementos para pasarlos a una sola cadena.

Retorna:

- str: una cadena que contiene todos los elementos de la lista.

"""

```
cadena = ""
```

```
# El ciclo for recorre cada valor de la lista y lo convierte en una cadena.
```

```
for valor in lista_valores:
```

```
    # En cada iteración, concatenamos el valor con un salto de línea.
```

```
    # De esta manera, generamos una cadena de texto donde cada valor de la lista
```

está

```
    # separado por una nueva línea.
```

```
    cadena = f"{cadena}{valor}\n"
```

```
return cadena
```

```
if __name__ == '__main__':
```

```
    valores = [10, 20, 100, 3000, 40]
```

```
    # Se convierte la lista de valores a una cadena con formato
```

```
    valores_en_cadena = valores_cadena(valores)
```



```
# Se calcula la suma de los valores en la lista utilizando recursividad
suma = suma_recursiva(valores, len(valores))
```

```
# Se muestra la cadena con los valores y la suma final calculada recursivamente
print(f"La suma de los valores del arreglo {valores_en_cadena}es: {suma}")
```

```
"""
La salida es:
```

```
La suma de los valores del arreglo 10
```

```
20
```

```
100
```

```
3000
```

```
40
```

```
es: 3170
"""
```

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Realizar una lectura detallada del material descrito en la [sección "Recursividad" del capítulo tres del texto "Introducción a la Programación"](#), donde se presenta una explicación ampliada del concepto con el apoyo de diagramas y un lenguaje de alto nivel, completamente adaptable al entorno de programación utilizado en la guía. Esta actividad permitirá al estudiante comprender cómo funciona la recursividad, su aplicación en la resolución de problemas y la importancia de definir correctamente un caso base para evitar llamadas infinitas. Se recomienda analizar los ejemplos presentados, replicarlos en el lenguaje de programación utilizado en la guía y modificar las condiciones de salida para observar su impacto en la ejecución del código. Finalmente, se sugiere realizar una comparación entre soluciones recursivas e iterativas, analizar sobre sus ventajas y desventajas en distintos escenarios computacionales.



2. Analizar detenidamente el recurso titulado [“Introducción a la programación con Python 3”, en la sección “Recursión” del capítulo seis, “Funciones”](#), donde se explican los fundamentos del concepto de recursividad en Python. Esta actividad permitirá al estudiante comprender cómo una función puede llamarse a sí misma para resolver problemas, dividiéndolos en subproblemas más pequeños, facilitando soluciones de forma eficiente. Se recomienda revisar los ejemplos proporcionados en el recurso, replicarlos en un entorno de programación y analizar el impacto del caso base en la ejecución del código. Además, se sugiere modificar los ejercicios para explorar distintos escenarios y comparar la solución recursiva con una iterativa. Reflexionar sobre cuándo es más conveniente utilizar la recursividad contribuirá a una mejor comprensión y aplicación del concepto en problemas computacionales.
3. Revisar el recurso [tipo vídeo](#) que describe el proceso de la recursividad mediante un ejemplo donde se realiza una comparación con los procesos iterativos. Es necesario que los ejemplos planteados sean recreados y ejecutados en los entornos locales.
4. Revisar el siguiente ejemplo donde aplica el concepto de recursividad para invertir una lista de cadenas de texto.

El ejercicio tiene como objetivo desarrollar una función recursiva en Python que permita invertir una cadena de texto sin utilizar estructuras iterativas como bucles for o while. Para ello, el programa debe tener una función que reciba una cadena y devolver su versión invertida, utilizando llamadas recursivas para descomponer el problema en subproblemas más pequeños. La solución debe hacer uso de estructuras de control adecuadas, como el caso base, para detener la recursión y la concatenación progresiva de caracteres para formar la cadena invertida. Finalmente, el programa debe mostrar la salida en un formato claro y estructurado, permitiendo al usuario visualizar tanto la cadena original como su versión invertida.

```
def invertir_cadena(cadena):  
    """
```



Invierte una cadena de texto utilizando recursión.

Parámetro:

- cadena: Cadena de texto a invertir.

Retorna:

- str: La cadena invertida.

```
"""
if len(cadena) == 0:
    # Caso base: Si la cadena está vacía, retorna una cadena vacía.
    # Esto detiene la recursión evitando una llamada infinita.
    return ""
else:
    # Paso recursivo:
    # - Se toma el último carácter de la cadena: cadena[-1]
    # - Se llama recursivamente a la función con el resto de la cadena (sin
    el último carácter): cadena[:-1]
    return cadena[-1] + invertir_cadena(cadena[:-1])

if __name__ == '__main__':
    # Lista de frases a invertir
    lista_frases = [
        "Ecuador, un país megadiverso.",
        "Ecuador tiene cuatro mundos en un solo país.",
        "Del Oriente al mar en un solo día.",
        "La mitad del mundo está aquí.",
    ]

    # Se recorre la lista de frases
    for i in lista_frases:
        # Se llama a la función recursiva para invertir la cadena
        resultado = invertir_cadena(i)
        # Se imprime la cadena original junto con su versión invertida
        print(f"La cadena invertida de '{i}' es '{resultado}'")
```

Se puede ver el código en el [repositorio de GitHub](#).

Este ejercicio refuerza conceptos clave como recursividad y manipulación de cadenas, ayudando a los estudiantes a desarrollar un pensamiento algorítmico.





Semana 12

En la Semana 12, se estudiará la creación y uso de módulos y paquetes en Python, conceptos básicos para organizar y reutilizar código en programas más complejos. Aprenderán cómo estructurar sus propios módulos y paquetes, facilitando la modularización y mantenimiento del código. Realizarán ejercicios prácticos donde implementarán módulos personalizados y explorarán el uso de funciones y módulos predefinidos.

Unidad 3. Manejo avanzado de datos y estructuras

3.8. Creación y uso de módulos y paquetes

En la presente semana se estudiará lo referente a la creación de módulo y paquetes en Python; para ello, se solicita una lectura de la lección 14 denominada Módulos y paquetes del texto [Introducción a la programación con Python](#).

Al referirse a los módulos, en el recurso se establece que cualquier archivo con la extensión .py es reconocido como un módulo en Python, siempre que contenga declaraciones de variables, funciones o clases, las cuales pueden ser accedidas y utilizadas desde otros archivos dentro del mismo proyecto. Este enfoque modular facilita la organización del código, permitiendo a los programadores dividir aplicaciones complejas en componentes más manejables. Una de las principales ventajas de los módulos es que simplifican el desarrollo, ya que permiten separar la posible solución en archivos independientes, lo que favorece la claridad y el mantenimiento del software. Además, la capacidad de realizar pruebas de manera más eficiente es otro beneficio importante, pues los módulos pueden probarse individualmente sin afectar otras partes del código. Pero, el aspecto más destacado es la reutilización del código, un principio fundamental en la optimización del desarrollo de software. Al utilizar módulos, los desarrolladores pueden evitar la repetición innecesaria de código y reducir errores.



En Python, las directivas que se usan para realizar importaciones de un módulo son: `from` e `import`. La forma a seguir es:

- `from nombre-módulo import clases, funciones y variables`
 - Se importan las clases, funciones y variables que se deseen con relación a la solución.
- `from nombre-modulo import *`
 - Se importan todas las clases, funciones y variables del módulo correspondiente.

Para crear un ejemplo del uso de funciones, se sugiere los siguientes pasos:

1. Crear un directorio llamado `ejemplo`.
2. Crear un archivo llamado `mis_funciones.py`, dentro de la carpeta `ejemplo`.
3. Crear un archivo llamado `principal`, dentro de la carpeta `ejemplo`.
4. La estructura debe estar de la siguiente forma:

```
|— ejemplo
|   |— mis_funciones.py
|   |— principal.py
```

5. Estructura de los archivos.

Los archivos tienen la siguiente estructura

```
# mis_funciones.py
# Definición de la función para imprimir un reporte con los datos ingresados
def imprimir_reporte(nombre, apellido, edad, ciudad, correo):
```

```
    """
```

```
        Función que genera un reporte de "matrícula" a partir de los datos
        ingresados,
```

```
        aplicando transformaciones (pasarlos a mayúsculas, con la función
```



upper)

a los valores de nombre, apellido y ciudad.

"""

Se construye la cadena formateada con la información del estudiante

cadena = f"Reporte de Matrícula\n"

f"Nombre: {nombre.upper()}\n"

f"Apellido: {apellido.upper()}\n"

f"Correo: {correo}\n"

f"Edad: {edad}\n"

f"Ciudad: {ciudad.upper()}"

Se imprime el reporte en pantalla

print(cadena)

Definición de la función que genera un correo institucional

def generar_correo(nombre, apellido):

"""

Función que genera un correo institucional en función del nombre y apellido

pasados como parámetros.

Los datos se convierten a minúsculas para mantener un formato estándar.

"""

Se convierten los valores de nombre y apellido a minúsculas, con la

función lower()

nombre = nombre.lower()

apellido = apellido.lower()

Se construye la dirección de correo electrónico

correo = f"{nombre}_{apellido}@utpl.edu.ec"

Se retorna el correo generado

return correo



```

# principal.py
# Se importa las funciones desde el módulo 'mis_funciones'
from mis_funciones import imprimir_reporte, generar_correo
# Alternativamente, se podría importar todo el módulo con: from
mis_funciones import *

# Definición de una función que solicita datos al usuario por teclado
def ingresar_datos():
    """
    Función que solicita datos de un usuario por teclado y genera un reporte,
    llamando a funciones que están en otro módulo
    """

    # Ingreso de datos por teclado
    nombre_ingresado = input("Ingresar nombre: ")
    apellido_ingresado = input("Ingresar apellido: ")

    edad_ingresada = input("Ingresar edad: ")
    edad_ingresada = int(edad_ingresada) # Conversión a entero

    ciudad_ingresada = input("Ingresar ciudad: ")

    # Se hace uso de la función generar_correo del módulo importado
    # Se genera el correo con base en los datos ingresados
    correo_generado = generar_correo(nombre_ingresado,
    apellido_ingresado)

    # Se llama a la función imprimir_reporte (proveniente del módulo
    importado)
    # Se envían los argumentos correspondientes para generar la salida en
    pantalla
    imprimir_reporte(nombre_ingresado, apellido_ingresado, edad_ingresada,
    ciudad_ingresada, correo_generado)

```




```

if __name__ == "__main__":
    #
    # Se llama a la función ingresar_datos para iniciar el ingreso de
    información
    ingresar_datos()

```

Explicación de lo realizado:

- Se crea un módulo llamado `mis_funciones`, que contiene dos funciones. La primera, `imprimir_reporte()`, genera y muestra un reporte con información personal, aplicando algunos formatos a los datos recibidos. La segunda, `generar_correo()`, genera un correo institucional a partir del nombre y apellido ingresados.
- En el archivo principal, se utiliza ***from mis_funciones import imprimir_reporte, generar_correo*** para importar solo las funciones necesarias, permitiendo su uso. Esto optimiza la organización del código y facilita su reutilización.
- En el script principal, se define una función encargada de solicitar datos al usuario, convertirlos a los tipos de datos necesarios y almacenarlos en variables. Posteriormente, se utiliza `generar_correo(nombre, apellido)`, importada desde el módulo, para generar automáticamente un correo electrónico con los datos ingresados.
- Luego de obtener los datos del usuario y generar el correo, se llama/invoca a la función `imprimir_reporte(nombre, apellido, edad, ciudad, correo)`, la cual proviene del módulo `mis_funciones`. Esta función recibe los datos como argumentos y muestra un reporte formateado con la información del usuario.

En la Figura 26, se puede observar el funcionamiento, al momento de ejecutar desde un terminal el archivo principal.py.



Figura 26

Uso de funciones de un módulo en Python

Uso de funciones de un módulo en Python

```
(envpy3-programacion) $ python principal.py
```

```
Ingresar nombre: Felix
```

```
Ingresar apellido: Santos
```

```
Ingresar edad: 30
```

```
Ingresar ciudad: Quito
```

```
Reporte de Matrícula
```

```
Nombre: FELIX
```

```
Apellido: SANTOS
```

```
Correo: felix_santos@utpl.edu.ec
```

```
Edad: 30
```

```
Ciudad: QUITO
```

Nota. Elizalde, R., 2025.

En la figura anterior se muestra la salida por pantalla de un código hecho en Python que hace uso funciones de un módulo dado.

El ejemplo está disponible en el [repositorio de GitHub](#), para el uso en los entornos locales.

Es momento de estudiar e implementar paquetes; cuando se habla de un paquete en Python, estrictamente es un directorio o carpeta; donde residen uno o varios archivos considerados módulos. Un paquete se considera otra forma de organizar módulos de forma óptima.

Como se mencionó un paquete es un directorio, para que Python lo considere formalmente como un paquete, es necesario que dentro de la carpeta se cree un archivo cuyo nombre de forma obligatoria es `__init__.py`; dicho archivo puede estar vacío.



En Python las directivas que se usan para realizar importaciones de un paquete son: `from` e `import`. La forma a seguir es:

- `from nombre-paquete.nombre-modulo import clases, funciones y variables`
 - Se importa las clases, funciones y variables que se deseen del módulo que pertenece a un paquete
- `from nombre-paquete.nombre-modulo import *`
 - Se importa todas las clases, funciones y variables del módulo que pertenece a un paquete

Para crear un ejemplo del uso de funciones de un paquete se sugiere los siguientes pasos:

1. Crear un directorio llamado ejemplo
2. Crear un subdirectorio llamado `paquete01`, este será el paquete a usar.
Dentro del `paquete01` un archivo `__init__.py` vacío
3. Crear un archivo llamado `mis_funciones01.py`, dentro de la carpeta `paquete01` y un archivo llamado `mis_funciones02.py`, dentro de la carpeta `paquete01`
4. Crear un archivo llamado `principal`, dentro de la carpeta `ejemplo`
5. La estructura debe estar de la siguiente forma:

```
|— paquete01
|   |— mis_funciones01.py
|   |— mis_funciones02.py
|— principal.py
```

6. Estructura de los archivos

```
# paquete01/mis_funciones01.py
# Definición de la función para obtener los datos personales
def obtener_datos_personales():
    """
    Función que solicita al usuario ingresar su nombre y apellido.
```



Se genera automáticamente un correo utilizando la función `generar_correo`.

```
"""
```

```
# Ingreso de datos por teclado
```

```
nombre_ingresado = input("Ingresar nombre: ")
```

```
apellido_ingresado = input("Ingresar apellido: ")
```

```
# Se llama a la función generar_correo del mismo módulo
```

```
correo_generado = generar_correo(nombre_ingresado,
apellido_ingresado)
```

```
# Se retorna un conjunto de valores para este caso
```

```
# formalmente está retornando una tupla de valores:
```

```
return nombre_ingresado, apellido_ingresado, correo_generado
```

```
# Definición de la función que obtiene las notas ingresadas por el usuario
```

```
def obtener_notas():
```

```
"""
```

Función que solicita al usuario ingresar 5 notas y las almacena en una lista.

```
"""
```

```
lista_nota = []
```

```
for i in range(0, 5):
```

```
    nota = input("Ingrese nota :")
```

```
    nota = float(nota)
```

```
    lista_nota.append(nota)
```

```
return lista_nota
```

```
# Definición de la función que calcula el promedio de las notas
```

```
def obtener_promedio(lista_notas):
```

```
"""
```

Función que calcula el promedio de una lista de notas proporcionada como argumento.

```
"""
```



```

suma = 0
for i in lista_notas:
    suma = suma + i
promedio = suma / len(lista_notas)

```

```

return promedio

```

Definición de la función que genera un correo institucional

```

def generar_correo(nombre, apellido):

```

```

    """

```

Función que genera un correo institucional en función del nombre y apellido

pasados como parámetros.

Los datos se convierten a minúsculas para mantener un formato estándar.

```

    """

```

Se convierten los valores de nombre y apellido a minúsculas, con la

función lower()

```

nombre = nombre.lower()

```

```

apellido = apellido.lower()

```

Se construye la dirección de correo electrónico

```

correo = f"{nombre}_{apellido}@utpl.edu.ec"

```

Se retorna el correo generado

```

return correo

```

```

# paquete01/mis_funciones02.py

```

Definición de la función para imprimir un reporte con los datos ingresados

```

def imprimir_reporte(nombre, apellido, correo, notas, promedio):

```

```

    """

```

Función que recibe los datos y los muestra en formato de reporte.

```

    """

```



Se construye la cadena formateada con la información del estudiante

```
cadena = f"Reporte de Matrícula\n"\n        f"Nombre: {nombre.upper()}\n"\n        f"Apellido: {apellido.upper()}\n"\n        f"Correo: {correo}\n"\n        f"Notas\n"
```

for l in notas:

```
    cadena = f"{cadena}{l}\n"
```

```
cadena = f"{cadena}Promedio: {promedio:.2f}"
```

Se imprime el reporte en pantalla

```
print(cadena)
```

principal.py

Importación de módulos (mis_funciones01 y mis_funciones02) desde el paquete01

```
from mis_funciones01 import obtener_datos_personales, obtener_notas,\n    obtener_promedio, generar_correo\nfrom mis_funciones02 import imprimir_reporte
```

Definición de la función principal con el nombre 'inicio'

```
def inicio():
```

```
    """
```

Función principal del programa que permite la obtención de datos y la generación del reporte.

```
    """
```

Se solicitan los datos personales del usuario

```
nombre, apellido, correo = obtener_datos_personales()
```

Se solicitan las notas del usuario

```
notas = obtener_notas()
```

Se calcula el promedio de las notas ingresadas

```
promedio = obtener_promedio(notas)
```



```
# Se imprime el reporte con toda la información recopilada
imprimir_reporte(nombre, apellido, correo, notas, promedio)
```

```
if __name__ == "__main__":
    inicio() # Se llama a la función principal 'inicio'
```

En el ejemplo, se ha implementado un paquete en Python, el cual organiza múltiples módulos (`mis_funciones01.py` y `mis_funciones02.py`) para facilitar la reutilización del código. Se crea un archivo `__init__.py` que da vida al paquete formalmente.

En este caso, el módulo `mis_funciones01.py` contiene funciones relacionadas con la recopilación de datos y el cálculo de valores para el promedio, mientras que `mis_funciones02.py` se encarga de la presentación de información mediante la función `imprimir_reporte`.

El archivo `principal.py` utiliza `from paquete01.mis_funciones01 import obtener_datos_personales, obtener_notas, obtener_promedio, generar_correo` y `from paquete01.mis_funciones02 import imprimir_reporte` para importar solo las funciones necesarias de cada módulo del paquete.

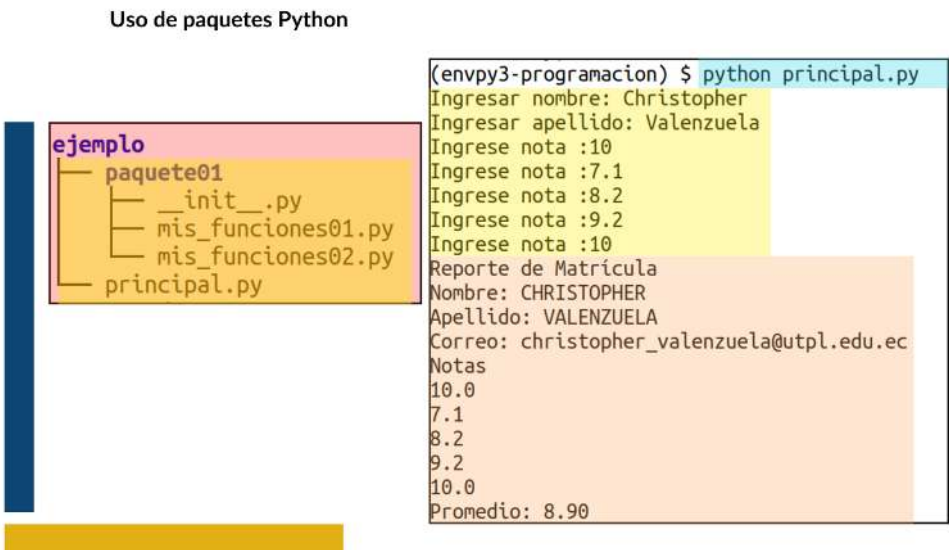
Finalmente la dinámica del problema empieza cuando se llama a la función `inicio()`, bajo el condicional `if __name__ == "__main__"`.

En la Figura 27, se denota la ejecución del archivo `principal.py`, quien desencadena acciones a través de sus propias funciones, además de las que se importa de los módulos del paquete.



Figura 27

Uso de paquetes en Python



Nota. Elizalde, R., 2025.

La figura anterior detalla un directorio considerado con Python. Y la salida por pantalla del código, que hace uso de funciones que están en módulo de un paquete.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.

Ahora es momento de dar un paso más en nuestro aprendizaje y revisar el próximo tema, que enriquecerá su comprensión

3.9. Uso de funciones y módulos predefinidos

En Python existen un conjunto de funciones y módulos predefinidos que los usuarios pueden utilizar para trabajar en cada solución generada.

En relación a las funciones predefinidas, son todas aquellas funciones que son parte del módulo `builtins`, mismo que se carga de forma automática al momento de ejecutar Python, por cualquier forma posible. Dichas funciones, no necesitan una importación para poder ser usadas. Para listar este tipo de funciones, se lo puede realizar a través de una consola de Python, con las siguientes sentencias:

```
import builtins
print(dir(builtins))
```

la salida de las líneas anteriores es:

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',
'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning',
'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__IPYTHON__',
'__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__',
'__package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'bool',
'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate',
'eval', 'exec', 'filter', 'float', 'format', 'frozenset', 'get_ipython', 'getattr', 'globals',
'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len',
```



'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

En la presente guía, ya se ha hecho uso de algunas funciones predefinidas; a continuación, expresamos las características de algunas de ellas:

- función `abs`, que permite obtener el valor absoluto de un número

```
valor = abs(-10)
print(valor) # la salida es 10
```

- función `max`, que permite obtener el valor máximo de una secuencia o lista de valores numéricos

```
valor = [10, 9, 6, 100, 99]
print(max(valor)) # la salida es 100
```

- función `pow`, que permite obtener la potencia de un número; se envían dos argumentos, la base y la potencia deseada.

```
valor = 2
potencia = 10
print(pow(2, 10)) # la salida es 1024
```

- función `sorted`, que permite ordenar una secuencia de información, por defecto de forma ascendente.

```
print(sorted([10, 1, 2, 4, 3]))
# salida [1, 2, 3, 4, 10]
```

Además, Python tiene la posibilidad de agregar en el trabajo módulos predefinidos, quienes poseen un conjunto de funciones. Dichos módulos deben importarse para poder trabajar con sus características.

Para ejemplificar se usará los módulos `math` y `random`.



```

# ejemplo1.py
# Importa el módulo math, el cual contiene funciones matemáticas avanzadas
import math

# Uso de la función predefinida math.sqrt()
print("Uso de math.sqrt")
# math.sqrt(x) calcula la raíz cuadrada del número dado como argumento
print(math.sqrt(9)) # Salida esperada: 3.0

print("-----")

# Uso de la función predefinida math.pow()
print("Uso de math.pow")

# Se definen valores para la base y la potencia
base = 3
potencia = 4
# math.pow(x, y) eleva x a la potencia y (equivalente a x**y)
print(math.pow(base, potencia)) # Salida esperada: 81.0

print("-----")

# Uso de la función predefinida math.floor()
print("Uso de math.floor")

# Se define un valor decimal
valor = 10.8
# math.floor(x) redondea el número x hacia abajo al entero más cercano
print(math.floor(valor)) # Salida esperada: 10

# ejemplo2.py
# Importa el módulo random, el cual permite generar valores aleatorios
import random

# Ejemplo 1: Generar un número entero aleatorio
print("Uso de random.randint()")

```



```
# random.randint(a, b) genera un número entero aleatorio en el rango [a, b]
numero_aleatorio = random.randint(1, 100)
print(f"Número aleatorio entre 1 y 100: {numero_aleatorio}")
# Salida esperada: Un número aleatorio entre 1 y 100, por ejemplo: 57
```

```
print("-----")
```

```
# Ejemplo 2: Seleccionar un elemento aleatorio de una lista
print("Uso de random.choice()")
```

```
# Se define una lista con diferentes provincias de Ecuador
provincias = ["Loja", "Azuay", "Pichincha", "Guayas", "Esmeraldas"]
# random.choice(lista) selecciona un elemento aleatorio de la lista
provincia = random.choice(provincias)
print(f"Provincia seleccionada: {provincia}")
# Salida esperada: Una provincia aleatoria de la lista, por ejemplo: Loja

print("-----")
```

```
# Ejemplo 3: Desordenar aleatoriamente los elementos de una lista
print("Uso de random.shuffle()")
```

```
# Se define una lista con números del 1 al 5
numeros = [1, 2, 3, 4, 5]
print(f"Lista original: {numeros}")
# random.shuffle(lista) desordena los elementos de la lista
random.shuffle(numeros)
print(f"Lista desordenada: {numeros}")
# Salida esperada: La lista con los mismos elementos, pero en un orden
diferente,
# por ejemplo: [3, 1, 5, 2, 4]
```

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.





Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Comprender y analizar el contenido del [recurso Python paso a paso en el capítulo nueve llamado Módulos](#) para fortalecer conocimientos sobre funciones predefinidas, módulos y paquetes. Esta actividad permitirá al estudiante identificar cómo organizar el código en módulos reutilizables, importar funciones de manera eficiente y estructurar proyectos de mayor complejidad utilizando paquetes. Se recomienda revisar los ejemplos del recurso y replicarlos en un entorno de programación para observar su funcionamiento. Además, se sugiere explorar la forma de crear y utilizar paquetes personalizados. Finalmente, practicar la importación de módulos utilizando diferentes métodos y analizar sobre su impacto en la organización y mantenimiento del código contribuirá a consolidar estos conocimientos.
2. Realizar una lectura detallada sobre la [sección llamada Módulos del texto Introducción a la programación con Python 3 en el capítulo seis](#). Se hace una introducción conceptual sobre la creación de módulos propios y el uso de módulos predefinidos en Python. Esta actividad permitirá al estudiante comprender cómo modularizar su código, reutilizar funciones en diferentes programas y mejorar la organización de proyectos mediante la importación de módulos. Se recomienda analizar los ejemplos presentados en el recurso y replicarlos en un entorno de programación. Además, se sugiere crear módulos personalizados con funciones específicas y probar su integración en distintos programas. Reflexionar sobre las ventajas de modularizar el código y su impacto en la legibilidad, mantenimiento y escalabilidad de los proyectos contribuirá a fortalecer estos conocimientos.
3. Revisar la solución de la siguiente problemática y recrear en su entorno local.



Un programa en Python que permita gestionar la información de estudiantes y docentes de una institución educativa. Para ello, el programa deberá solicitar al usuario que ingrese un número determinado de estudiantes y docentes, capturando su información personal y almacenándola para su posterior presentación en un reporte estructurado. El sistema deberá permitir el registro de estudiantes, solicitando su nombre, apellidos, correo electrónico y edad. Una vez ingresados los datos, estos deberán almacenarse en una lista para su procesamiento y visualización en un reporte detallado. De manera similar, se deberá permitir el registro de docentes, solicitando su nombre, apellidos y ciudad de residencia, almacenando la información en otra lista para su posterior presentación.

Para mejorar la interacción del usuario, el programa deberá contar con un menú interactivo que le permita elegir entre tres opciones principales: registrar estudiantes, registrar docentes o registrar automáticamente un estudiante y un docente. El usuario podrá indicar cuántos estudiantes o docentes desea registrar en cada ejecución del programa. Además, el menú deberá mantenerse en ejecución mediante un bucle while, permitiendo realizar múltiples registros hasta que el usuario decida salir del sistema.

El diseño del programa deberá seguir una estructura donde se use paquete, módulos y funciones para manejar el registro de estudiantes y docentes. Asimismo, se deberá hacer uso de listas para almacenar y manipular los datos ingresados. La presentación de la información deberá realizarse mediante un reporte formateado, utilizando cadenas de acumulación y el método f-string para una salida clara y organizada.

Para dar solución, la estructura del proyecto tiene la siguiente forma:

```
|— paquete
|   |— __init__.py
|   |— modulo1.py
|   |— modulo2.py
```



```
|   └─ modulo3.py
|   └─ README.md
|   └─ run.py
```

El archivo run.py tiene la siguiente estructura:

```
from paquete.modulo1 import obtener_estudiantes
from paquete.modulo2 import obtener_docentes

# Menú principal del programa
if __name__ == "__main__":
    bandera = True # Variable de control para mantener el programa en
    ejecución

    while bandera: # Se ejecuta hasta que el usuario decida salir
        mensaje = "Menú\nIngrese 1 para generar estudiantes; 2 para
        generar docentes; 3 para generar un estudiante y un docente: "
        opcion = input(mensaje) # Se solicita al usuario que elija una
        opción del menú
        opcion = int(opcion) # Conversión a entero para procesar la
        elección

        if opcion == 1:
            # Solicitar el número de estudiantes a registrar
            elementos = input("Ingrese el número de estudiantes a generar:
            ")
            elementos = int(elementos) # Conversión a entero
            obtener_estudiantes(elementos) # Se llama a la función para
            registrar estudiantes
        else:
            if opcion == 2:
                # Solicitar el número de docentes a registrar
                elementos = input("Ingrese el número de docentes a generar:
                ")
                elementos = int(elementos) # Conversión a entero
```



```
    obtener_docentes(elementos) # Se llama a la función para
    registrar docentes
```

```
    else:
```

```
        # Se ejecuta la opción por defecto, generando un estudiante y
        un docente
```

```
        # pues en cada función existe un parámetro con un valor por
        defecto
```

```
        obtener_estudiantes()
```

```
        obtener_docentes()
```

```
    # Se solicita al usuario si desea salir del programa
```

```
    salida = input("Ingrese 'si' para salir del proceso, caso contrario use
    cualquier valor para seguir: ")
```

```
    if salida.lower() == "si": # Si el usuario ingresa "si", se termina el
    bucle
```

```
        bandera = False # Se cambia la variable de control para salir del
        programa
```

Se puede revisar el ejemplo completo en el [repositorio de GitHub](#).

4. Estimado estudiante, ha llegado el momento de evaluar su comprensión sobre los temas abordados en la segunda parte de la Unidad 3: Manejo avanzado de datos y estructuras, los cuales incluyen recursividad, creación y uso de módulos y paquetes, y funciones predefinidas en Python. Esta autoevaluación le permitirá identificar qué conceptos domina y en cuáles debería reforzar su aprendizaje.

Antes de realizar la autoevaluación, se recomienda seguir estas estrategias de estudio:

- Revisar cuidadosamente el contenido de cada tema, prestando atención a la estructura de las funciones recursivas y los casos base.
- Consultar los recursos educativos (documentos, ejemplos de código y bibliografía recomendada) para reforzar sus conocimientos.



- Practicar con ejercicios de recursividad y modularización, comprendiendo cómo dividir problemas en subproblemas más pequeños.
- Explorar el uso de funciones y módulos predefinidos, identificando cómo pueden optimizar el desarrollo de programas en Python.

Recuerde que cada pregunta tiene una única respuesta correcta. Lea con atención cada enunciado y seleccione



Autoevaluación 5

1. ¿Cuál de las siguientes opciones describe correctamente la recursividad en programación?

Respuestas:

- Un método exclusivo de la Programación Orientada a Objetos.
- Un tipo especial de bucle que se ejecuta indefinidamente.
- Un proceso en el cual una función se llama a sí misma para resolver un problema.

2. Autocompletar:

Una función recursiva debe tener un caso base y debe _____ en algún punto del código. Sin embargo, no es obligatorio que utilice una variable global.

3. ¿Cuál es el propósito de un caso base en una función recursiva?

Respuestas:

- Garantizar que la función no se llame a sí misma infinitamente.
- Hacer que la función siempre retorne None.
- Evitar que la función devuelva un valor incorrecto.

4. ¿Cuál de las siguientes afirmaciones sobre módulos en Python es correcta?



Respuestas:

- a. Solo se pueden importar módulos que vienen predefinidos en Python.
- b. Un módulo es cualquier archivo Python con extensión .py.
- c. Un módulo solo puede contener una función.

5. Responda verdadero o falso a la siguiente idea:

La manera correcta de importar solo la función sumar() desde un módulo llamado operaciones.py es from operaciones import *

Respuestas:

- a. Verdadero. ()
- b. Falso. ()

6. ¿Cuál es el propósito del archivo __init__.py dentro de un paquete en Python?

Respuestas:

- a. Hacer que todas las funciones sean públicas.
- b. Convertir un directorio en un paquete Python válido.
- c. Convertir automáticamente los archivos en funciones.

7. ¿Cuál será la salida si el usuario ingresa 6?

```
def es_par(n):  
  
    if n == 0:  
  
        return True  
  
    elif n == 1:  
  
        return False  
  
    else:
```



```
return es_par(n - 2)
```

```
print(es_par(6))
```

Respuestas:

- a. None.
- b. False.
- c. True.

8. **¿Cuál de las siguientes estructuras representa correctamente un paquete en Python?**

Respuestas:

- a. /paquete

```
__init__.py
```

```
modulo1.py
```

```
modulo2.py
```

- b. /paquete

```
modulo1.py
```

```
modulo2.py
```

- c. /paquete

```
archivo.txt
```

```
modulo.py
```

9. **¿Cuál será la salida si el usuario ingresa 4?**

```
def suma_recursiva(n):
```



```
if n == 0:

    return 0

else:

    return n + suma_recursiva(n - 1)

print(suma_recursiva(4))
```

Respuestas:

- a. 15.
- b. 4.
- c. 10.

10. **¿Cuál de las siguientes funciones NO es una función predefinida en Python?**

Respuestas:

- a. max().
- b. sum().
- c. factorial().

[Ir al solucionario](#)

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 13

La Semana 13, permite a los estudiantes el estudio en el manejo de excepciones en Python, una técnica fundamental para controlar errores y mejorar la estabilidad de los programas. Aprenderán cómo utilizar bloques try y except para gestionar situaciones inesperadas y evitar fallos en la ejecución



del código. Realizarán ejercicios prácticos donde aplicarán excepciones en la resolución de problemas, asegurando que los programas manejen adecuadamente posibles errores durante la ejecución.

Unidad 4. Técnicas avanzadas y operaciones I/O

4.1. Introducción al manejo de excepciones

Un concepto importante para el desarrollo de aplicaciones de software es el manejo de excepciones. En la presente semana se estudiará la implementación en el lenguaje Python.

Según Moreno Muñoz (2017), se define a una excepción como:

“Un error lógico que ocurre mientras se ejecuta el programa y que provoca su detención. La detección puede ser controlada y no producirse si se realiza un control de excepciones correcto en el código fuente” (p.70).

López & Rojas (2021), manifiesta:

“El manejo de excepciones es una característica de la mayoría de los lenguajes de programación modernos que nos permite lidiar con situaciones en las que es posible que ocurran errores y, en lugar de terminar la ejecución del programa completamente, podemos hacer un desvío y tratar de solucionar el problema (p. 78).



Los autores resaltan la importancia del manejo de las excepciones para lograr que la ejecución de las soluciones no tenga o sufra un corte inesperado, generando una experiencia negativa en el usuario final que usa el programa.

Se solicita realizar una lectura comprensiva de la Lección 8 – [Excepciones del texto Introducción a la programación con Python](#), el recurso indica que las excepciones permiten que un programa siga su proceso de ejecución a través del control de los posibles errores que puedan suceder en escenarios,



logrando que se pueda llegar al final. La aplicación de excepciones en una solución en lenguajes de alto nivel depende de la característica de cada programa.

4.2. Uso de excepciones en soluciones

En el lenguaje Python se utilizan las palabras reservadas try y except en el manejo de excepciones, permitiendo que se capturen y se administren los posibles errores que se presentan en la ejecución de un programa.

Para conocer más sobre estas excepciones, le invito a revisar el siguiente módulo didáctico:

[TRY Y EXCEPT EN PYTHON](#)

Estimado estudiante, el ejemplo completo se puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.

Ejemplo 2:

- Generar un programa en Python que solicite al usuario el ingreso de dos valores numéricos: un numerador y un denominador. Para garantizar la validez de los datos, el programa usa validaciones mediante el manejo de excepciones. Primero, se verifica que el numerador ingresado sea mayor o igual a 20; caso contrario, se lanza una excepción personalizada utilizando raise Exception. Luego, se captura el error de ZeroDivisionError que se genera cuando se ingresa un valor de 0 en el denominador. Además, si el usuario introduce valores no numéricos. Para el numerador o denominador, se captura un ValueError, mostrando un mensaje de error. Una vez que se tienen valores válidos, el programa realiza la operación de división y muestra el resultado con dos decimales de precisión. Todo el código está estructurado dentro de un bloque try-except, asegurando que el usuario pueda interactuar de manera continua hasta proporcionar datos válidos. Finalmente, el programa imprime el resultado de la operación y muestra un mensaje de finalización, indicando el término exitoso del proceso.



```

# Se establece una bandera, variable de tipo booleana
# para controlar la ejecución del bucle
# Se inicializa en True (verdadero)
bandera = True

# Se inicia un bucle que continuará hasta que el usuario ingrese datos válidos,
# Caso contrario, seguirá ejecutándose el ciclo
while bandera:
    # Se inicia un bloque try para capturar posibles errores durante la ejecución
    try:
        # Se solicita al usuario un valor para el numerador y se convierte a float
        numerador = input("Ingrese un valor numérico como numerador: ")
        numerador = float(numerador)

        # Se solicita al usuario un valor para el denominador y se convierte a float
        denominador = input("Ingrese un valor numérico como denominador: ")
        denominador = float(denominador)

        # Validación del numerador: debe ser mayor o igual a 20
        # Si no cumple, se lanza una excepción personalizada usando raise
        Exception

        if numerador < 20:
            raise Exception("Error: El numerador debe ser mayor o igual a 20.")

        # Se realiza la operación de división
        division = numerador / denominador

        # Se imprime el resultado de la división, formateado con 2 decimales
        print(f"La división entre {numerador}/{denominador} es igual a {division:.2f}")

    # Se cambia la bandera a False para salir del bucle
    bandera = False

```



Se captura el error ValueError cuando el usuario ingresa un valor no numérico

```
except ValueError as v:
```

```
    print("Error: Se debe ingresar un valor numérico válido.")
```

```
    print(v) # Se muestra el mensaje técnico del error
```

Se captura el error ZeroDivisionError cuando el usuario intenta dividir por cero

```
except ZeroDivisionError as z:
```

```
    print("Error: No se puede dividir por cero.")
```

```
    print(z) # Se muestra el mensaje técnico del error
```

Se captura cualquier otra excepción, incluyendo la generada por raise Exception

```
except Exception as e:
```

```
    print(e) # Se muestra el mensaje de error personalizado
```

Mensaje final del programa, se ejecuta independientemente de si hubo error o no

```
print("Fin del programa")
```

Del código anterior se resalta lo siguiente:

- ValueError: Se captura cuando el usuario ingresa un valor que no es numérico, como una letra o un símbolo.
- ZeroDivisionError: Se captura si el usuario ingresa 0 como denominador, lo que genera una división no válida.
- Exception: Se usa para capturar cualquier otro error, incluyendo la excepción personalizada generada con raise Exception cuando el numerador es menor a 20; raise Exception se usa para lanzar una excepción manualmente en el código. El bloque except Exception as e: captura cualquier otro error que no haya sido manejado por los except anteriores (ValueError y ZeroDivisionError). Exception es la clase base de todas las excepciones en Python. Si se coloca primero, evitaría que cualquier otra excepción específica se maneje correctamente; por eso se recomienda usarla al final.



Estimado estudiante, el ejemplo completo se puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Revisar los recursos de [tipo vídeo](#) donde se explican ejemplos de excepciones en Python; es importante que se recreen los ejemplos en los entornos locales, para lograr una mayor comprensión de los conceptos.
2. Leer e interpretar la sección de excepciones del [capítulo tres relacionado con Fundamentos del Lenguaje en el texto Python a fondo](#). Dicho recurso explica el uso de excepciones en Python a través de try/except y raise. Esta actividad permitirá al estudiante comprender cómo manejar errores en tiempo de ejecución, evitando que los programas se detengan de forma inesperada y mejorando la robustez del código. Se recomienda analizar los ejemplos del recurso y replicarlos en un entorno de programación para observar cómo se capturan diferentes tipos de excepciones. Además, se sugiere experimentar con la generación de errores personalizados utilizando raise, aplicándolo en validaciones de entrada y control de flujo del programa. Finalmente, reflexionar sobre la importancia del manejo de excepciones en aplicaciones reales contribuirá a desarrollar programas más seguros y eficientes.

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 14

En la Semana 14, los estudiantes entenderán los conceptos sobre el manejo de archivos en Python, comprendiendo su importancia en la persistencia de datos. Aprenderán lo esencial sobre archivos, incluyendo su estructura y formas de acceso. Realizarán ejercicios prácticos en los que implementarán la



lectura y escritura de archivos, utilizando diferentes modos de apertura para manipular información almacenada en texto, aplicando buenas prácticas para evitar errores en el procesamiento de datos.

Unidad 4. Técnicas avanzadas y operaciones I/O

4.3. Conceptos sobre archivos

Hasta ahora, los datos utilizados en los programas que se han desarrollado solo existen en la memoria volátil (RAM) mientras el programa se ejecuta. Esto significa que, una vez que el programa finaliza o el computador se apaga, toda la información procesada desaparece, lo que limita o impide la capacidad de almacenamiento y persistencia de datos. Para solucionar este problema, los lenguajes de programación ofrecen herramientas para leer y escribir archivos, permitiendo que la información se conserve incluso después de que el programa deje de ejecutarse.

Con relación al manejo de archivos, Ayala San Martín (2020) indica que:

“Un archivo es una estructura de datos en la cual el almacenamiento se hace en un dispositivo de memoria que permite que permanezcan independientemente de si nuestro programa está ejecutándose o no. Los procesos típicos en el uso de un archivo en conjunto con un arreglo son los siguientes:

- Cargar datos de un archivo para almacenarlos en un arreglo.
- Procesar los datos utilizando el arreglo.
- Grabar los datos de un arreglo en el archivo”(p. 116).

Algunas características adicionales sobre el manejo de archivos para los desarrolladores, las destaca Fritelli, Guzman & Tymoschuk (2020).

“El uso de archivos permite el manejo de grandes volúmenes de datos sin tener que volverlos a cargar desde teclado cada vez que se desee procesarlos. Por otra parte, una vez que un archivo fue creado y grabado en un disco, ese archivo puede ser utilizado por cualquier otro programa,



ya sea para obtener datos del archivo y/o para modificar los datos del mismo. En otras palabras: no solo el archivo puede ser usado por el programa que lo creó, sino también por cualquier otro programa que realice las definiciones adecuadas (p. 207).

4.4. Lectura de archivos

Con relación a la lectura y escritura de archivos en Python, se solicita realizar una revisión analítica de la [Lección 15 del texto Introducción a la programación con Python](#), en el mismo se explican los procesos básicos para el manejo de archivos.

Los archivos pueden clasificarse en estructurados y no estructurados, dependiendo de cómo se almacena y organiza la información en ellos. Los archivos estructurados siguen un formato predefinido que facilita la manipulación y análisis de los datos. Estos suelen estar organizados en registros y campos delimitados por separadores como comas, puntos y comas o tabulaciones, permitiendo un acceso eficiente a cada dato dentro del archivo. Por otro lado, los archivos no estructurados no siguen un formato específico ni poseen delimitadores que segmenten los datos de manera clara. Para la lectura de archivos, es importante comprender cómo están organizados los datos, cuál es el delimitador usado y cómo se estructuran los registros para poder procesarlos correctamente en los programas. Se puede ver un ejemplo de los tipos de archivos en la Figura.



Figura 27

Archivos estructurados y no estructurados

Archivos

Archivo estructurado

```
1 ID,Nombre,Apellido,Edad,Departamento,Salario
2 1,Juan,Pérez,30,Contabilidad,2500
3 2,María,López,27,Recursos Humanos,2800
4 3,Carlos,García,35,IT,3200
5 4,Ana,Rodríguez,29,Marketing,2700
```

Archivo no estructurado

```
1 Hoy tuvimos una reunión con el equipo de desarrollo.
2 Discutimos sobre los avances en el nuevo sistema de gestión de clientes.
3 María mencionó que hay algunos problemas en la integración de la base de datos.
4 Carlos sugirió hacer pruebas antes de continuar con la implementación.
5 El próximo encuentro será el lunes.
```

Nota. Elizalde, R., 2025.

En la figura anterior se muestran ejemplos típicos de archivos con datos estructurados y no estructurados

Para la lectura de archivos se usará la función open, de la siguiente forma:

```
archivo = open("ruta/nombre-archivo.extensión-del-archivo", "r")
```

Donde,

- Se declara una variable llamada archivo (se puede usar otro nombre) que almacenará el objeto de archivo devuelto por open(). Esta variable se usará para realizar operaciones sobre el archivo.
- open() es una función predefinida en Python que permite abrir archivos en diferentes modos (lectura, escritura, etc.).



- "ruta/nombre-archivo.extensión-del-archivo" es una cadena de texto que representa la ubicación del archivo en el sistema.
 - "ruta/" ; especifica la ruta del archivo en el sistema (puede ser absoluta o relativa).
 - "nombre-archivo" es el nombre del archivo.
 - ".extensión-del-archivo" es el formato del archivo (por ejemplo, .txt, .csv, .json, etc.).
- "r"; representa el modo de apertura, "r" significa modo de lectura. En este modo, el archivo se abre solo para leer datos. Si el archivo no existe, se genera una excepción del tipo FileNotFoundError.

Ejemplo 1: Generar una solución en Python que permita leer y presentar por pantalla la información de un archivo que tiene la siguiente información:

ID,Nombre,Apellido,Edad,Departamento,Salario

1,Juan,Pérez,30,Contabilidad,2500

2,María,López,27,Recursos Humanos,2800

3,Carlos,García,35,IT,3200

4,Ana,Rodríguez,29,Marketing,2700

Solución

```
# 1. Crear la variable que manejará el archivo, a través de open
#
archivo = open("data/informacion-01.txt", "r")
# donde:
# data es la carpeta donde está el archivo
# mis-datos.txt, es el nombre del archivo que se desea leer
# "r" es el modo lectura que deseamos usar para manejar el archivo

# se usa el método/función readlines, que ubica cada línea del archivo en una
# posición de una lista
textos = archivo.readlines()
# Cerrar el archivo después de la lectura para liberar recursos del sistema
archivo.close()
# se imprime texto a esta altura se tiene
# ['ID,Nombre,Apellido,Edad,Departamento,Salario\n',
# '1,Juan,Pérez,30,Contabilidad,2500\n',
```



```
# '2,María,López,27,Recursos Humanos,2800\n',
# '3,Carlos,García,35,IT,3200\n',
# '4,Ana,Rodríguez,29,Marketing,2700\n']
#
# Donde se observa que cada posición es una cadena, el método usado
# agrega un salto de línea, que se lo debería eliminar para evitar
# problemas con los datos
```

```
# Se usa una lista compresada de python - list comprehension
# que permite aplicar la función strip (para eliminar espacios y
# saltos de línea de una cadena) a cada iteración (de tipo cadena)
# de la lista.
textos = [l.strip() for l in textos]
```

```
# se presenta cada posición de la lista en pantalla
for t in textos:
    print(t)
```

```
print("-----")
# la salida es:
#
# ID,Nombre,Apellido,Edad,Departamento,Salario
# 1,Juan,Pérez,30,Contabilidad,2500
# 2,María,López,27,Recursos Humanos,2800
# 3,Carlos,García,35,IT,3200
# 4,Ana,Rodríguez,29,Marketing,2700
```

```
# Al considerar que el archivo tiene información estructurada
# Se puede hacer lo siguiente por cada iteración
```

```
for t in textos:
    # se almacena en una variable mi_cadena, el valor de iteración
    # iteración, a la cual se le aplica la función split,
    # que permite separar la cadena mediante el carácter
    # coma (,) en una lista
    mi_cadena = t.split(",") # mi_cadena es una lista ahora
    for v in mi_cadena:
        print(v)
    print("#####")
```

```
print("-----")
"""
```

```
La salida es:
ID
Nombre
Apellido
Edad
```



```

Departamento
Salario
#####
1
Juan
Pérez
30
Contabilidad
2500
#####
2
María
López
27
Recursos Humanos
2800
#####
3
Carlos
García
35
IT
3200
#####
4
Ana
Rodríguez
29
Marketing
2700
#####
-----
"""

```



Modificar la solución anterior, para que permita sumar los valores de los salarios de cada línea de archivo, considerar que no debe usar la primera línea, pues son los encabezados.

- # 1. Abrir el archivo de texto en modo lectura ("r")
`archivo = open("data/informacion-01.txt", "r")`
- # 2. Leer todas las líneas del archivo y almacenarlas en una lista
`textos = archivo.readlines()`
- # 3. Cerrar el archivo después de la lectura para liberar recursos del sistema

```
archivo.close()
```

```
# 4. Eliminar los saltos de línea "\n" en cada línea del archivo usando lista por comprensión
```

```
textos = [l.strip() for l in textos]
```

```
# 5. Eliminar la primera línea (encabezados) del archivo
```

```
textos = textos[1:]
```

```
# 6. Inicializar una variable para almacenar la suma de los salarios
```

```
suma_salarios = 0
```

```
# 7. Recorrer cada línea del archivo procesado
```

```
for t in textos:
```

```
    # 7.1. Separar la línea en una lista utilizando la coma como delimitador
```

```
    mi_cadena = t.split(",")
```

```
    # Ejemplo de la lista generada en cada iteración:
```

```
    # ['1', 'Juan', 'Pérez', '30', 'Contabilidad', '2500']
```

```
    # 7.2. Extraer el valor de la columna "Salario" (posición 5 en la lista)
```

```
    valor = mi_cadena[5]
```

```
    # 7.3. Convertir el valor de salario de string a float
```

```
    valor = float(valor)
```

```
    # 7.4. Acumular el salario en la variable "suma_salarios"
```

```
    suma_salarios = suma_salarios + valor
```

```
# 8. Mostrar el resultado de la suma de todos los salarios
```

```
print(f"La suma de salarios es: {suma_salarios}")
```

```
"""
```

```
Salida esperada:
```

```
La suma de salarios es: 11200.0
```

```
"""
```

En los ejemplos previos, se empleó la instrucción `archivo.close()` con el propósito de cerrar el archivo una vez finalizada su lectura, evitando así el uso innecesario de recursos del sistema. Sin embargo, Python proporciona una alternativa más eficiente mediante la estructura `with`, la cual gestiona automáticamente el cierre del archivo, eliminando la necesidad de hacerlo de forma manual y reduciendo el riesgo de posibles errores a futuro.



La nueva versión para el ejemplo donde se suma los salarios de cada línea del archivo es:

```
# 1. Utilizar la estructura "with" para abrir el archivo en modo lectura ("r")
# El archivo se abrirá y cerrará automáticamente al salir del bloque with
with open("data/informacion-01.txt", "r") as archivo:
    # 2. Leer todas las líneas del archivo y almacenarlas en una lista
    textos = archivo.readlines()

# 3. El archivo ya está cerrado en este punto, por lo que se puede manipular la
variable "textos" sin estar dentro de with

# 4. Eliminar los saltos de línea "\n" en cada línea del archivo usando lista por
comprensión
textos = [l.strip() for l in textos]

# 5. Eliminar la primera línea (encabezados) del archivo
textos = textos[1:]

# 6. Inicializar una variable para almacenar la suma de los salarios
suma_salarios = 0

# 7. Recorrer cada línea del archivo procesado
for t in textos:
    # 7.1. Separar la línea en una lista utilizando la coma como delimitador
    mi_cadena = t.split(",")
    # Ejemplo de la lista generada en cada iteración:
    # ['1', 'Juan', 'Pérez', '30', 'Contabilidad', '2500']

    # 7.2. Extraer el valor de la columna "Salario" (posición 5 en la lista)
    valor = mi_cadena[5]

    # 7.3. Convertir el valor de salario de string a float
    valor = float(valor)

    # 7.4. Acumular el salario en la variable "suma_salarios"
    suma_salarios = suma_salarios + valor

# 8. Mostrar el resultado de la suma de todos los salarios
print(f"La suma de salarios es: {suma_salarios}")

"""
Salida esperada:
La suma de salarios es: 11200.0
"""
```



Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.

Ahora es momento de dar un paso más en nuestro aprendizaje y revisar el próximo tema, que enriquecerá su comprensión

4.5. Escritura de archivos

Para la escritura de un archivo a través de lenguaje Python se usará la siguiente forma:

```
open("ruta/nombre-archivo.extensión-del-archivo", "w") as archivo:  
    # Se escribe la información en el archivo usando write()  
    archivo.write(cadena-para-guardar) # Se guarda la cadena en el archivo
```

Donde:

- `open("ruta/nombre-archivo.extensión-del-archivo", "w") as archivo`
 - `open`, integrada de Python que abre un archivo en un modo específico.
 - `"ruta/nombre-archivo.extensión-del-archivo"`, especifica la ubicación y el nombre del archivo que se quiere escribir o sobrescribir
 - `"ruta/"`, es el directorio donde se encuentra o se creará el archivo.
 - `"nombre-archivo"`, es el nombre del archivo que se desea usar.
 - `".extensión-del-archivo"`, los tipos de archivo a usar (.txt, .csv, .json, etc.).
 - `"w"`, se refiere al modo de apertura "write" (escritura). Si el archivo ya existe, se sobrescribe su contenido. Si el archivo no existe, Python lo crea automáticamente.
 - `as archivo`, crea una variable (archivo) que representa el archivo abierto, permitiendo manipularlo.
- `archivo.write(cadena-para-guardar)`
 - `write()`, es el método que escribe contenido en el archivo.



- Cadena-para-guardar, es el contenido que se va a almacenar en el archivo.

Ejercicio 1: Generar una solución que permita ingresar datos de parroquias a través del teclado; el usuario decide cuando ya no quiera ingresar más datos. Finalmente guardar los datos ingresados en un archivo llamado informacion-escritura-03.txt.

Definir una función para guardar información en un archivo de texto
def guardar_informacion(cadena_final):

"""

Función que guarda la información recibida en un archivo de texto.

Parámetros:

cadena_final (str): Cadena de texto que contiene la información
que se almacenará en el archivo.

"""

El archivo se abrirá y cerrará automáticamente al salir del bloque with

Se usa el modo "w" para escritura, lo que sobrescribe el contenido del
archivo si ya existe.

with open("data/informacion-escritura-03.txt", "w") as archivo:

Se escribe la información en el archivo usando write()

archivo.write(f"{cadena_final}") # Se guarda la cadena en el archivo

Al salir del bloque 'with', el archivo se cierra automáticamente

Definir una función para ingresar datos desde el usuario
def ingresar_informacion():

"""

Función que permite ingresar información sobre parroquias,
almacenando datos como el nombre, número de mujeres, número de
hombres
y total de población.

Retorna:





```
str: Una cadena con la información ingresada, separada por "|".
"""

bandera = True # Controla el bucle while para permitir múltiples ingresos de
datos
cadena_parroquias = "" # Almacena la información ingresada en formato de
cadena

while bandera:
    # Solicitar el nombre de la parroquia
    nombre = input("Ingrese el nombre de la parroquia: ")

    # Capturar y convertir a entero la cantidad de mujeres
    poblacion_mujeres = input("Ingrese número de mujeres de la parroquia: ")
    poblacion_mujeres = int(poblacion_mujeres)

    # Capturar y convertir a entero la cantidad de hombres
    poblacion_hombres = input("Ingrese número de hombres de la parroquia:
")
    poblacion_hombres = int(poblacion_hombres)

    # Calcular el total de la población
    total_poblacion = poblacion_mujeres + poblacion_hombres

    # Formatear los datos ingresados y agregarlos a la cadena final
    # Cada línea contiene los datos separados por "|"
    cadena_parroquias = f"{cadena_parroquias}{nombre}|{poblacion_mujeres}|
{poblacion_hombres}|{total_poblacion}\n"

    # Solicitar al usuario si desea continuar o finalizar el ingreso de datos
    valor = input("Ingrese el cero (0) para salir del ciclo: ")

    try:
        valor = int(valor) # Convertir el valor ingresado a entero
        if valor == 0: # Si el usuario ingresa 0, finaliza el bucle
            bandera = False
```

```

except Exception as e:
    pass # Se ignoran errores si el usuario ingresa un valor no numérico

return cadena_parroquias # Se retorna la cadena con la información
ingresada

# Punto de entrada principal del script
if __name__ == "__main__":
    # Se solicita información al usuario
    mis_parroquias = ingresar_informacion()

    # Se guarda la información ingresada en un archivo de texto
    guardar_informacion(mis_parroquias)

    # Mensaje de finalización
    print("Fin")

```

En la Figura 29, se observa el proceso de ingreso de datos y la generación del archivo, con la información previa ingresada por teclado, al ejecutar el código anterior



Figura 29

Escritura de archivos en Python



Nota. Elizalde, R., 2025.

La figura anterior muestra la forma como se ingresan valores, el destino o directorio donde almacenará la información y como se visualizará la información.

Estimado estudiante, el ejemplo completo se lo puede descargar del [repositorio de Github](#) y realizar los cambios para su estudio en su entorno local.



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Estudiar y analizar la [sección "Ficheros" del capítulo siete, "Ficheros y Excepciones", del texto "Python 3: Curso Práctico"](#), donde se explica detalladamente el manejo de archivos en Python mediante ejemplos documentados. Esta actividad permitirá al estudiante comprender los tipos de archivos, los modos de apertura (r, w), y las funciones para

lectura y escritura de datos. Se recomienda replicar en un entorno local los ejemplos presentados en el recurso para observar su comportamiento, experimentando con la apertura, escritura y lectura de archivos de texto. Además, se sugiere realizar modificaciones en los ejemplos para explorar el uso de `with open()`, evitar errores comunes y garantizar el correcto cierre de archivos.

2. Revisar la solución de la siguiente problemática y recrear en su entorno local.

Un programa en Python que permita gestionar la información de estudiantes y docentes de una institución educativa. Para ello, el programa deberá solicitar al usuario que ingrese un número determinado de estudiantes y docentes, capturando su información personal y almacenándola para su posterior presentación en un reporte estructurado. El sistema deberá permitir el registro de estudiantes, solicitando su nombre, apellidos, correo electrónico y edad. Una vez ingresados los datos, estos deberán almacenarse en una lista para su procesamiento y visualización en un reporte detallado. De manera similar, se deberá permitir el registro de docentes, solicitando su nombre, apellidos y ciudad de residencia, almacenando la información en otra lista para su posterior presentación.

Para mejorar la interacción del usuario, el programa deberá contar con un menú interactivo que le permita elegir entre tres opciones principales: registrar estudiantes, registrar docentes o registrar automáticamente un estudiante y un docente. El usuario podrá indicar cuántos estudiantes o docentes desea registrar en cada ejecución del programa. Además, el menú deberá mantenerse en ejecución mediante un bucle `while`, permitiendo realizar múltiples registros hasta que el usuario decida salir del sistema.



El diseño del programa deberá seguir una estructura donde se use paquete, módulos y funciones para manejar el registro de estudiantes y docentes. Asimismo, se deberá hacer uso de listas para almacenar y manipular los datos ingresados. La presentación de la información deberá guardarse en archivos.

Para dar solución, la estructura del proyecto tiene la siguiente forma:

```
|— data
|   |— informacion-docentes.txt
|   |— informacion-estudiantes.txt
|— paquete
|   |— __init__.py
|   |— modulo1.py
|   |— modulo2.py
|   |— modulo3.py
|— README.md
|— run.py
```

El archivo modulo3.py tiene la siguiente estructura:

```
def obtener_reporte(tipo, lista):

    reporte = ""

    if tipo == 1:
        reporte = "\nListado de Estudiantes\n" # Cadena para almacenar
        la lista de docentes

        # Construcción del reporte de estudiantes a partir de la lista de
        estudiantes
        for i in lista:
            reporte = f"{reporte}{i[0]} {i[1]} -correo:{i[2]}-, edad:{i[3]}\n"
            guardar_informacion(reporte, 1)
    else:
```




```

if tipo == 2:
    reporte = "\nListado de Docentes\n" # Cadena para almacenar

    # Construcción del reporte de docentes a partir de la lista de
    docentes
    for i in lista:
        reporte = f"{reporte}{i[0]} {i[1]} -ciudad de residencia:{i[2]}\n"

    guardar_informacion(reporte, 2)
    return reporte

```

```

def guardar_informacion(cadena_final, tipo):

```

```

    """

```

Función que guarda la información recibida en un archivo de texto.

Parámetros:

`cadena_final (str)`: Cadena de texto que contiene la información
que se almacenará en el archivo.

```

    """

```

El archivo se abrirá y cerrará automáticamente al salir del bloque
with

Se usa el modo "w" para escritura, lo que sobrescribe el contenido
del archivo si ya existe.

```

if tipo == 1:
    cadena = "data/informacion-estudiantes.txt"
else:
    if tipo == 2:
        cadena = "data/informacion-docentes.txt"

```

with open(cadena, "a") as archivo:

```

    # Se escribe la información en el archivo usando write()
    archivo.write(f"{cadena_final}") # Se guarda la cadena en el

```



archivo

Al salir del bloque 'with', el archivo se cierra automáticamente

```
def presentar_informacion():
```

```
    # 1. Utilizar la estructura "with" para abrir el archivo en modo lectura ("r")
```

```
    # El archivo se abrirá y cerrará automáticamente al salir del bloque with
```

```
    with open("data/informacion-estudiantes.txt", "r") as archivo:
```

```
        # 2. Leer todas las líneas del archivo y almacenarlas en una lista
```

```
        textos = archivo.readlines()
```

```
        for t in textos:
```

```
            print(t)
```

```
    with open("data/informacion-docentes.txt", "r") as archivo:
```

```
        # 2. Leer todas las líneas del archivo y almacenarlas en una lista
```

```
        textos = archivo.readlines()
```

```
        for t in textos:
```

```
            print(t)
```

El ejemplo completo se lo puede descargar del [repositorio de GitHub](#)

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 15

En la Semana 15, los estudiantes aprenderán a instalar y utilizar librerías externas en Python, herramientas que amplían las funcionalidades del lenguaje y facilitan el desarrollo de proyectos. Estudiarán el proceso de instalación de paquetes mediante pip, entendiendo cómo gestionar dependencias en un entorno de desarrollo. Realizarán ejercicios prácticos en los que aplicarán librerías externas conocidas para resolver problemas específicos, comprendiendo su importancia en la solución de problemáticas.



Unidad 4. Técnicas avanzadas y operaciones I/O

4.6. Proceso de instalación de librerías externas

Los lenguajes de programación cuentan con un conjunto de módulos predefinidos que pueden utilizarse directamente sin necesidad de importación, ya que forman parte de la funcionalidad base del lenguaje. Además, existe una amplia gama de módulos adicionales que se encuentran dentro de lo que se conoce como la librería estándar, los cuales pueden ser incorporados en los programas mediante la instrucción `import`. Estos módulos proporcionan herramientas para diversas tareas, como manipulación de archivos, gestión de fechas y tiempos, entre otras funcionalidades principales, como se indicó en temáticas anteriores.

Sin embargo, las necesidades que se pueden presentar en el mundo de la programación van más allá de lo que la librería estándar de un lenguaje puede ofrecer. Es por ello que las comunidades de desarrolladores, y en especial del software libre, han impulsado la creación de librerías externas que amplían las capacidades de los lenguajes de programación. Estas librerías permiten trabajar de forma más acertada y optimizada en problemáticas específicas en áreas como inteligencia artificial, análisis de datos, desarrollo web, automatización de procesos, etc. Dichas temáticas serán estudiadas en profundidad a lo largo de su carrera.

Python tiene algunos repositorios oficiales de librerías, uno de ellos es [Python Package Index \(PyPI\)](#), que es un repositorio donde los desarrolladores pueden subir y distribuir un conjunto de librerías para la comunidad. Dichas librerías se pueden instalar en las máquinas personales o servidores; y se puede hacer a través de la herramienta de gestión de paquetes denominada `pip`. `Pip` viene instalado por defecto en el proceso de instalación de Python desde la versión 3.4.



Se solicita realizar una lectura comprensiva de la sección Instalar nuevos módulos y paquetes del capítulo 9 del texto [Python paso a paso](#); en el recurso indica que pip permite la instalación, actualización y desinstalación de paquetes. En contextos más complejos, se recomienda trabajar las librerías con pip y con entornos virtuales en Python, a través de [venv](#) o [virtualenv](#).

Algunos de los comandos más importantes cuando se use pip, son los siguientes:

- pip --version; permite saber la versión del manejador de paquetes.
- pip install nombre_paquete; permite la instalación de paquetes. En nombre_paquete debe reemplazarse con el nombre del paquete, según la información oficial que se disponga. Ejemplo:
 - pip install pandas
- pip install nombre_paquete==versión; este comando brinda la opción de instalar una librería con su versión exacta. Ejemplo:
 - pip install pandas==2.2.3
- pip install --upgrade nombre_paquete / pip install -U nombre_paquete; permite la actualización de un paquete previamente instalado.
- pip install -r requirements.txt ; este comando instalar un conjunto de paquetes, que deben estar listados en el archivo requeriments.txt
- pip uninstall nombre_paquete ; permite la quitar o desinstalar un paquete. Ejemplo:
 - pip uninstall pandas
- pip list ; a través del comando de lista, los paquetes con sus versiones exactas que están instaladas.

Se recomienda revisar el listado y descripción detallada de los [comandos posibles de pip](#).



4.7. Uso de librerías externas

A continuación, se realizará el proceso de instalación y ejemplificación de algunas librerías, considerando que se profundizarán algunas de ellas en el futuro; el objetivo en esta sección es indicar el proceso para tener operativa una librería externa en su computador personal.

- Numpy, es una librería que permite realizar operaciones matemáticas eficientes y manipulación de arreglos multidimensionales. Las áreas donde se puede usar es ciencia de datos, machine learning y simulaciones científicas debido a su rapidez.

- Comando de instalación: `pip install numpy`.
- Página oficial de la librería: [NumPy](https://numpy.org/).
- Ejemplo.

```
# Se importa el módulo y se usa por intermedio de as
# un alias para ser usado en el script, en este caso np
import numpy as np
# Crear un array de 2x2
matriz = np.array([[1, 2], [3, 4]])

# Calcular la media de la matriz
media = np.mean(matriz)

print("Matriz:\n", matriz)
print("Media de la matriz:", media)
```

"""

La salida es:

Matriz:

```
[[1 2]
```

```
[3 4]]
```

Media de la matriz: 2.5

"""



- Pandas, es una librería que ayuda en procesos para análisis y manipulación de datos. Agrega conceptos para trabajar con estructuras como DataFrames y Series; las librerías permiten la carga, transformación y exportación de datos en distintos formatos como CSV, Excel y JSON.

- Instalación: `pip install pandas`.
- Página oficial: [pandas - pydata](https://pandas.pydata.org).
- Ejemplo.

Se importa el módulo y se le asigna un alias

pd

`import pandas as pd`

Crear un DataFrame con datos de estudiantes

Se crea un diccionario de datos.

`data = {`

`"Nombre": ["Ana", "Juan", "Pedro"],`

`"Edad": [20, 22, 21],`

`"Calificacion": [90, 85, 88]`

`}`

Se crea la estructura DataFrame con el valor de data

`df = pd.DataFrame(data)`

Mostrar el DataFrame en pantalla

`print(df)`

Calcular la media de calificaciones, haciendo uso de la

función `mean` a la columna de calificaciones

#

`print(f'Media de calificaciones: {df["Calificacion"].mean():.2f}')`

`"""`

Salida:

Nombre Edad Calificacion

0 Ana 20 90

1 Juan 22 85



2 Pedro 21 88

Media de calificaciones: 87.67

=====

- Sympy, librería que permite realizar cálculos de derivadas, integrales y ecuaciones.

- Instalación: `pip install sympy`
- Página oficial: [sympy](https://sympy.org/)
- Ejemplo

Se importa el módulo y se le asigna un alias

sp

`import sympy as sp`

Definir la variable simbólica x

`x = sp.Symbol('x')`

Definir la ecuación $2x + 3 = 7$

`ecuacion = 2*x + 3 - 7`

Resolver la ecuación

`solucion = sp.solve(ecuacion, x)`

`print(f"La solución de la ecuación $2x + 3 = 7$ es: $x = \{solucion[0]\}")$`

=====

La solución de la ecuación $2x + 3 = 7$ es: $x = 2$

=====

- Matplotlib, librería muy usada para la creación y visualización en Python; genera gráficos de líneas, barras, dispersión, histogramas, etc.
- Página oficial: Matplotlib
 - Instalación: `pip install matplotlib`
 - Ejemplo





```
# Se importa el módulo y se le asigna un alias
# plt
import matplotlib
matplotlib.use("TkAgg") # Forzar el uso de un backend interactivo
import matplotlib.pyplot as plt

# Datos
x = ["A", "B", "C", "D"]
y = [5, 8, 3, 6]

# Crear gráfico de barras
plt.bar(x, y, color='blue')

# Etiquetas
plt.xlabel("Categorías")
plt.ylabel("Valores")
plt.title("Ejemplo de Gráfico de Barras")

# Mostrar el gráfico
# plt.show()

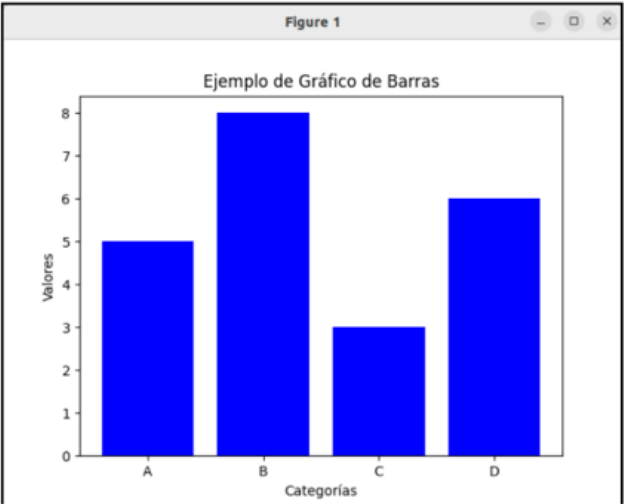
plt.show(block=True) # Bloquea la ejecución hasta que cierres la ventana
```

La salida se la puede observar en la Figura 30

Figura 30

Uso de matplotlib

Uso de matplotlib



Nota. Elizalde, R., 2025.

La figura anterior muestra un Ejemplo de salida de una gráfica, hecha en Python a través de la librería externa matplotlib



Actividades de aprendizaje recomendadas

Es momento de aplicar su conocimiento a través de las actividades que se han planteado a continuación:

1. Revisar el siguiente recurso que contiene una [lista de vídeos relacionados](#) con el uso de pip. La idea es entender cómo trabajar con pip para el proceso de instalación y actualización de librerías externas. Se recomienda usar los comandos en su entorno local.
2. Estudiar y reflexionar el contenido del [capítulo cinco del texto "Python: Aplicaciones Prácticas"](#), donde se detalla el proceso de instalación y uso de librerías externas como Scipy, Numpy y Matplotlib. Este recurso permitirá al estudiante comprender cómo estas herramientas amplían las capacidades de Python en distintas áreas, gracias a la contribución



de su comunidad y la diversidad de módulos disponibles. Se recomienda seguir los pasos de instalación descritos en el texto y replicar los ejemplos proporcionados para familiarizarse con el uso de cada librería. Además, se sugiere explorar casos prácticos en los que Numpy optimiza cálculos numéricos, Scipy facilita análisis científicos y Matplotlib permite generar visualizaciones de datos. Finalmente, reflexionar sobre la importancia de las librerías externas en la resolución de problemas avanzados ayudará a consolidar su aplicación en proyectos de programación y análisis de datos.

3. Estimado estudiante, ha llegado el momento de evaluar su comprensión sobre los temas abordados en la Unidad 3 y Unidad 4, relacionados con el manejo avanzado de estructuras de datos, excepciones en Python, manipulación de archivos y el uso de librerías externas. Esta autoevaluación le permitirá identificar qué conceptos domina y en cuáles debería reforzar su aprendizaje.

Antes de realizar la autoevaluación, se recomienda seguir estas estrategias de estudio:

- Revisar cuidadosamente el contenido de cada tema tratado, prestando especial atención a los conceptos clave.
- Consultar los recursos educativos (documentos, códigos de ejemplo y bibliografía recomendada) para reforzar sus conocimientos.
- Practicar con ejercicios de código, tanto en la gestión de excepciones como en la lectura y escritura de archivos.
- Investigar sobre el uso de librerías externas, comprendiendo su instalación y aplicación en distintos contextos de desarrollo.

Recuerde que cada pregunta tiene una única respuesta correcta. Lea con atención cada enunciado y seleccione la opción que usted considere adecuada. ¡Mucho éxito en esta autoevaluación!





Autoevaluación 6

1. **¿Cuál de las siguientes opciones describe correctamente una excepción en programación?**

Respuestas:

- a. Un mensaje que muestra información al usuario.
- b. Una advertencia que aparece antes de ejecutar un código.
- c. Un error lógico que ocurre mientras se ejecuta un programa.

2. **Responda verdadero o falso:**

El propósito de raise Exception("Mensaje de error") en Python es definir una función nueva.

Respuestas:

- a. Verdadero. ()
- b. Falso. ()

3. **¿Cuál de las siguientes afirmaciones sobre el uso de open("archivo.txt", "r") es correcta?**

Respuestas:

- a. Abre el archivo solo para escritura.
- b. Abre el archivo en modo lectura y genera un error si el archivo no existe.
- c. Borra el contenido del archivo al abrirlo.

4. **Responda verdadero o falso:**

En Python, el método readlines() permite leer todas las líneas de un archivo y almacenarlas en una lista.

5. **¿Cuál es el comando para instalar una librería externa en Python con pip?**



Respuestas:

- a. `install pip nombre_libreria`
- b. `pip install nombre_libreria`
- c. `python pip install nombre_libreria`

6. **¿Cuál será la salida si el usuario ingresa "abc" por teclado para la variable valor?**

try:

```
valor = int(input("Ingrese un número: "))
```

```
print(valor * 2)
```

except ValueError:

```
print("Entrada inválida. Debe ingresar un número.")
```

```
print("Fin")
```

Respuestas:

- a. Fin.
- b. Entrada inválida. Debe ingresar un número.

Fin.

- c. Entrada inválida. Debe ingresar un número.

7. **¿Cuál es la forma correcta de escribir "¡Hola, Python!" en un archivo llamado "mensaje.txt"?**

Respuestas:

- a. `with open("mensaje.txt", "w") as archivo:`

```
    archivo.write("Hola, Python!")
```

- b. `archivo = open("mensaje.txt", "r")`



```
archivo.write("Hola, Python!")
```

```
c. open("mensaje.txt", "w")
```

```
archivo.write("Hola, Python!")
```

8. ¿Cuál de las siguientes opciones es un error de tipo TypeError?

Respuestas:

- a. 10 / 0.
- b. "Loja" + 10.
- c. int("abc").

9. ¿Qué comando en pip permite ver todas las librerías instaladas?

Respuestas:

- a. pip list installs.
- b. pip list.
- c. list apps.

10. ¿Cuál es la salida del siguiente código si el usuario ingresa -5?

try:

```
num = int(input("Ingrese un número: "))
```

```
if num < 0:
```

```
    raise Exception("No se permiten números negativos.")
```

```
print(f"El número ingresado es {num}")
```

```
except Exception as e:
```

```
    print(f"Error: {e}")
```



Respuestas:

a. El número ingresado es -5.

Error: No se permiten números negativos.

b. El número ingresado es -5.

c. Error: No se permiten números negativos.

[Ir al solucionario](#)

Contenidos, recursos y actividades de aprendizaje recomendadas



Semana 16

Actividades finales del bimestre

La semana 16 es un período para que usted, estimado estudiante, tenga la oportunidad de volver a revisar los temas estudiados en el segundo bimestre y reforzar aquellos conceptos que requieran mayor comprensión. Este tiempo está destinado a consolidar su aprendizaje y prepararse para la evaluación del bimestre.

En esta semana, se propone revisar las siguientes unidades:

- Unidad 3: Unidad 3. Manejo avanzado de datos y estructuras.
- Unidad 4: Técnicas avanzadas y operaciones I/O.

Las estrategias sugeridas para aprovechar este período de repaso incluyen:

- Elaborar resúmenes, cuadros sinópticos o mapas conceptuales sobre los temas estudiados.
- Revisar las actividades realizadas en semanas anteriores y, cuando sea necesario, recrear los ejercicios desde cero para afianzar el manejo de herramientas y conceptos.
- Consultar en los recursos de aprendizaje recomendados en la planificación de la asignatura.



- Practicar con [ejercicios similares a los presentados](#) en la guía didáctica para reforzar la resolución de problemas.
- Repasar las preguntas de autoevaluación para evaluar su comprensión de los contenidos.
- Revisar los cuestionarios de repaso disponibles en la plataforma virtual de aprendizaje.

¿Listo para rendir la evaluación bimestral?





4. Autoevaluaciones

Autoevaluación 1

Pregunta	Respuesta	Retroalimentación
1	B	Un algoritmo no debe ser ambiguo, ya que su propósito es proporcionar una secuencia clara y precisa de pasos para resolver un problema. Si un algoritmo es ambiguo, puede generar resultados erróneos.
2	Diagrama de flujo.	Un diagrama de flujo permite visualizar los pasos a seguir de un algoritmo.
3	Las miniespecificaciones.	Las miniespecificaciones permiten escribir los pasos de un algoritmo en lenguaje natural sin depender de la sintaxis de un lenguaje de programación.
4	B	Los lenguajes de alto nivel como Python o Java son más comprensibles para los humanos, mientras que los de bajo nivel como ensamblador se acercan más a la estructura del hardware.
5	C	Los lenguajes interpretados, como Python, ejecutan su código línea por línea sin necesidad de un proceso de compilación previo, facilitando la depuración pero con menor rendimiento.
6	Verdadero.	Git es un sistema de control de versiones que permite gestionar cambios en proyectos de software, facilitando la colaboración y el seguimiento de modificaciones.
7	A	Un IDE facilita la programación al incluir herramientas como un editor de código, depuración y ejecución en un mismo entorno, mejorando la productividad del desarrollador.
8	B	Las pruebas de escritorio permiten verificar si la lógica de un algoritmo es correcta simulando su ejecución con diferentes datos de entrada, lo que ayuda a detectar errores antes de programar.



Pregunta	Respuesta	Retroalimentación
9	C	Los lenguajes de bajo nivel, como el ensamblador, están más cerca del hardware y requieren instrucciones detalladas para interactuar con la arquitectura del procesador.
10	B	Los procesos en un algoritmo corresponden a la transformación de las entradas mediante cálculos, operaciones y análisis previos para llegar a una solución. Estos definen la lógica que permite obtener los resultados esperados.
Ir a la autoevaluación		



Autoevaluación 2

Pregunta	Respuesta	Retroalimentación
1	C	En el bloque de declaraciones se especifican las variables necesarias para el programa. Esto garantiza que los datos estén bien definidos antes de ser usados en los cálculos o procesos.
2	Espacio en memoria.	Una variable es un espacio en memoria con un nombre asignado que permite almacenar y modificar datos durante la ejecución del programa.
3	B	En Python, el tipado dinámico permite declarar una variable simplemente asignándole un valor, sin necesidad de especificar el tipo de dato.
4	C	En la mayoría de los lenguajes de programación, las variables no pueden contener espacios en blanco, ya que esto genera errores de sintaxis.
5	Verdadero.	En Python, las variables se crean automáticamente al asignarles un valor y adoptan el tipo de dato correspondiente.
6	Falso.	El operador % devuelve el residuo de la división entera en Python. Por lo tanto, en este caso, $10 \% 3 = 1$ porque $10 \div 3$ tiene un residuo de 1.
7	A	3 no es par ($3 \% 2 \neq 0$) y no es mayor que 10, por lo que ninguna condición se cumple, ejecutando el bloque else.
8	C	La nota 8 está en el rango 7-10, por lo que se cumple la condición y se ejecuta "Aprobado".
9	B	Considerando que el número 85 está dentro del rango 80-100, se ejecuta el bloque correspondiente a "Nivel avanzado".
10	B	15 no está en ninguno de los rangos de adulto, por lo que se ejecuta la última opción "Menor de edad".

[Ir a la autoevaluación](#)



Autoevaluación 3

Pregunta	Respuesta	Retroalimentación
1	while.	El bucle while se usa cuando la cantidad de iteraciones no es fija, sino que depende de una condición lógica.
2	A	Un contador registra cuántas veces sucede un evento, mientras que un acumulador almacena la suma de valores en cada iteración.
3	B	Las listas en Python permiten almacenar y manipular varios elementos dentro de una misma variable, lo que facilita la gestión de datos.
4	Diccionario.	Los diccionarios en Python almacenan información en pares clave-valor, permitiendo un acceso eficiente a los datos.
5	C	El bucle while se ejecuta mientras contador < 5. Cuando contador llega a 4, se imprime " Número encontrado" .
6	Verdadero.	La variable suma acumula los valores del 1 al 5, por lo que el resultado es 15 (1+2+3+4+5).
7	C	El bucle divide n entre 2 hasta que sea menor que 1, contando cuántas veces ocurre. Para n=10, las iteraciones son: 10 5 2 1, lo que da 4.
8	A	Solo los números mayores a 25 (30 y 40) se imprimen.
9	C	Todos los elementos en la matriz son múltiplos de 5, por lo que contador cuenta 4 valores.
10	B	Solo los números pares (4, 8, 12, 6) se suman: $4 + 8 + 12 + 6 = 30$.

[Ir a la autoevaluación](#)



Autoevaluación 4

Pregunta	Respuesta	Retroalimentación
1	Verdadero.	Un procedimiento ejecuta instrucciones sin retornar valores, mientras que una función devuelve un resultado.
2	B	En Python, se pueden definir funciones sin parámetros o con parámetros opcionales.
3	A	Un parámetro con valor por defecto permite que la función sea llamada sin proporcionar un argumento. En este caso, a través del operador "=" se le asigna el valor a nombre en el encabezado de la función.
4	C	La palabra clave global permite modificar una variable global dentro de una función, pero debe usarse con precaución para evitar efectos no deseados.
5	Falso.	Pues, al momento de llamar la función funcion() se ejecuta e imprime "Hola, mundo".
6	B	La función doble_suma() llama a suma() y multiplica el resultado por 2, por lo que $3+4 = 7$, luego $7*2 = 14$.
7	A	En el primer print, la función recibe valores para los parámetros, guardando la correspondencia; se imprime Hola Carlos desde Loja, y en el segundo print, el parámetro ciudad usa su valor por defecto "Quito", ya que no se pasa un argumento, se imprime Hola, Ana, desde Quito.
8	A	suma_cuadrados(2, 3) llama a cuadrado(2) = 4 y cuadrado(3) = 9, sumando $4 + 9 = 13$.
9	B	La función mostrar_mensaje imprime primero "Buenos días" y luego "Hola", porque el segundo llamado no proporciona un argumento, usando el valor por defecto.
10	B	Cuando se realiza la llamada a calcular(2, 3, 2), luego se invoca a la función multiplicar(2, 3), que devuelve 6. Finalmente, se llama a la función dividir(6, 2), que devuelve 3.0.

[Ir a la autoevaluación](#)



Autoevaluación 5

Pregunta	Respuesta	Retroalimentación
1	C	La recursividad permite que una función se llame a sí misma para resolver problemas, dividiéndolos en partes más pequeñas.
2	Llamarse a sí misma	Una función recursiva tiene la característica de llamarse a sí misma.
3	A	El caso base es la condición que detiene la recursión para evitar que la función entre en un bucle infinito.
4	B	Un módulo en Python es cualquier archivo .py que contiene funciones, clases o variables reutilizables.
5	Falso.	Para importar solo una función específica, se usa <code>from nombre_modulo import nombre_funcion</code> ; con <code>*</code> se llaman a todas las clases, funciones y variables.
6	B	<code>__init__.py</code> es un archivo especial que convierte un directorio en un paquete en Python.
7	C	<code>es_par(6)</code> se llama recursivamente restando 2 hasta que <code>n=0</code> , devolviendo <code>True</code> .
8	A	Un paquete en Python es un directorio que contiene un archivo <code>__init__.py</code> y otros módulos .py.
9	C	La función <code>suma_recursiva(4)</code> se llama a sí misma sumando $4+3+2+1+0 = 10$; el proceso para cuando se hace uso del caso base <code>n==0</code> .
10	C	<code>Factorial()</code> pertenece al módulo <code>math</code> , no es una función predefinida en Python.

[Ir a la autoevaluación](#)



Autoevaluación 6

Pregunta	Respuesta	Retroalimentación
1	C	Una excepción es un error que ocurre en tiempo de ejecución y que puede ser manejado para evitar que el programa se detenga de forma inesperada.
2	Falso.	Raise Exception(“ Mensaje”) permite lanzar errores personalizados, lo que es útil para validar datos o definir reglas específicas.
3	B	El modo “r” solo permite leer archivos existentes y genera un FileNotFoundError si el archivo no se encuentra.
4	Verdadero.	Readlines() lee todas las líneas de un archivo y las almacena en una lista de Python.
5	B	Para instalar una librería externa, se usa pip install nombre_libreria.
6	B	ValueError se activa cuando se intenta convertir un texto no numérico a int e imprime “Entrada inválida. Debe ingresar un número”. Al salir de la estructura, imprime Fin.
7	A	“ w” abre el archivo en modo escritura, sobrescribiendo su contenido si existe.
8	B	TypeError ocurre cuando se intenta realizar una operación entre tipos incompatibles, como sumar un número y una cadena.
9	B	P ip list muestra todas las librerías instaladas junto con sus versiones.
10	C	Raise Exception(“ No se permiten números negativos) genera una excepción personalizada, capturada por la except.

[Ir a la autoevaluación](#)





5. Referencias bibliográficas

- Ramírez Marín, J. H. (2019). *Fundamentos iniciales de lógica de programación I. Algoritmos en PseInt y Python*: (1 ed.). D - Institución Universitaria de Envigado. <https://elibro.net/es/ereader/bibliotecautpl/226488>
- Moreno Muñoz, A. & Córcoles Córcoles, S. (2019). *Python práctico: Herramientas, conceptos y técnicas*: (1 ed.). RA-MA Editorial. <https://elibro.net/es/ereader/bibliotecautpl/222728>
- Moreno Pérez, J. C. (2014). *Programación en lenguajes estructurados*: (ed.). RA-MA Editorial. <https://elibro.net/es/ereader/bibliotecautpl/106445>
- Trejos Buriticá, O. I. (2017). *Lógica de programación*: (ed.). Ediciones de la U. <https://elibro.net/es/ereader/bibliotecautpl/70315>
- Muñoz Guerrero, L. E. (II.) & Trejos Buriticá, O. I. (2021). *Introducción a la programación con Python*: (1 ed.). RA-MA Editorial. <https://elibro.net/es/ereader/bibliotecautpl/230298>
- Hinojosa Gutiérrez, Á. (2015). *Python paso a paso*: (ed.). RA-MA Editorial. <https://elibro.net/es/ereader/bibliotecautpl/107213>
- López, A. & Rojas, A. L. (2021). *Introducción a Python para Estudiantes de Ciencias*: (1 ed.). Editorial UPTC. <https://elibro.net/es/ereader/bibliotecautpl/219225>
- Ayala San Martín, G. (2020). *Algoritmos y programación: mejores prácticas*: (ed.). Fundación Universidad de las Américas Puebla (UDLAP). <https://elibro.net/es/ereader/bibliotecautpl/180290>



- Fritelli, V. Guzman, A. & Tymoschuk, J. (2020). *Algoritmos y estructuras de datos*: (2 ed.). Jorge Sarmiento Editor - Universitas. <https://elibro.net/es/ereader/bibliotecautpl/175249>
- Hinojosa Gutiérrez, Á. (2015). *Python paso a paso*: (ed.). RA-MA Editorial. <https://elibro.net/es/ereader/bibliotecautpl/107213>
- Martín Villalba, C., Urquía Moraleda, A., & Rubio González, M. Á. (2021). *Lenguajes de programación*. UNED - Universidad Nacional de Educación a Distancia. <https://elibro.net/es/ereader/bibliotecautpl/184827>
- Marzal Varó, A. García Sevilla, P. & Gracia Luengo, I. (2016). *Introducción a la programación con Python 3*: (ed.). D - Universitat Jaume I. Servei de Comunicació i Publicacions. <https://elibro.net/es/ereader/bibliotecautpl/51760>
- Ramírez, F. (2007). *Introducción a la programación: Algoritmos y su implementación en vb. net, c#, java y c++*. Alpha Editorial.
- Farrell, J. (2013). *Introducción a la programación lógica y diseño* (7.^a). Cengage Learning Editores.
- Herrera, A. M. (2015). *Diseño y construcción de algoritmos*. Ediciones de la U.
- Rodríguez-Losada González, D. Muñoz Cano, J. & García Cena, C. (2022). *Introducción a la programación en C*: (1 ed.). UPM Press. <https://elibro.net/es/ereader/bibliotecautpl/256906>
- López, L. (2013). *Metodología de la programación orientada a objetos*. Alpha Editorial.
- Cuevas Álvarez, A. (2016). *Python 3: curso práctico*: (ed.). RA-MA Editorial. <https://elibro.net/es/ereader/bibliotecautpl/106404>



Moreno Pérez, J. C. (2015). *Programación*: (ed.). RA-MA Editorial. <https://elibro.net/es/ereader/bibliotecautpl/62476>

Arteaga Martínez, M. M. (2023). *Lógica de programación con Pseint: enfoque práctico*: (1 ed.). Corporación Universitaria Remington. <https://elibro.net/es/ereader/bibliotecautpl/250650>

(2024). ChatGPT (Versión 4.0) [Modelo de lenguaje de IA]. OpenAI. <https://openai.com>.

Parte de este contenido fue desarrollado con la asistencia de un modelo de inteligencia artificial (ChatGPT-4.0) y revisado/modificado por René Rolando Elizalde Solano.

