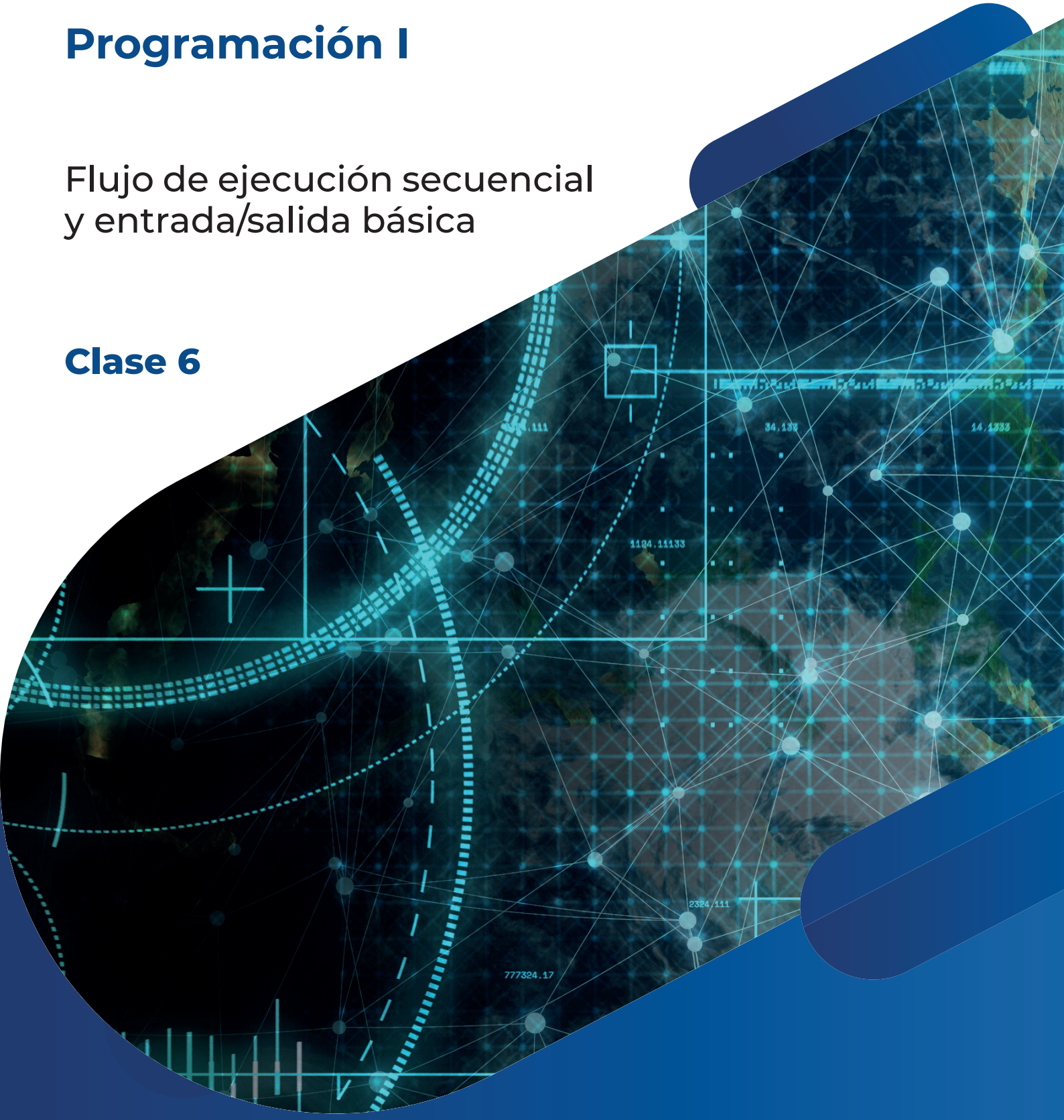
Programación I

Flujo de ejecución secuencial y entrada/salida básica

Clase 6



Ingeniería en ciberseguridad

La excelencia no se improvisa

1. INTRODUCCIÓN DE LA CLASE

¡Bienvenidos a la Clase 6! En la clase anterior, aprendimos la ejecución secuencial y la función input() para interactuar con el usuario. Hoy profundizaremos en dos aspectos esenciales: formateo de salida con print() y validación básica de datos. Imagina que tu programa ya lee valores, pero ahora queremos mostrar resultados de forma clara y profesional, al tiempo que verificamos que la información ingresada cumpla ciertos criterios. La robustez de una aplicación se ve reflejada en la forma en que maneja la presentación de la información y cómo responde a entradas inesperadas.

Así es, y descubrirás que la interacción con el usuario mejora sustancialmente cuando puedes controlar la apariencia de los mensajes (mediante print() y f-strings) y evitar bloqueos al validar los datos ingresados. Dominar el formateo y la validación es esencial para aplicaciones reales, donde la experiencia del usuario y la confiabilidad del sistema marcan la diferencia.

Clase 6: Flujo de ejecución secuencial y entrada/salida básica (Parte 2)

Resultado o resultados de aprendizaje que será abordado con el contenido de la clase.

Reconocer las estructuras básicas de un lenguaje de programación estructurado, su sintaxis y su utilidad en la solución de problemas de programación.

Reto # 2

Contenido de la Clase:

6. **Flujo de ejecución secuencial y entrada/salida básica (Parte 2)**

En esta segunda parte sobre el flujo de ejecución secuencial y la entrada/salida básica, profundizaremos en dos aspectos fundamentales que te permitirán crear programas no solo funcionales, sino también amigables y confiables para el usuario. Hasta ahora, hemos visto cómo organizar instrucciones en un orden lineal (de arriba hacia abajo), solicitar datos con la función input() y realizar operaciones sencillas. Sin embargo, la presentación de los resultados y la validación de la información ingresada son pasos decisivos para llevar tu código a un nivel más profesional y robusto.

- **Formateo de salida con print() (f-strings en Python):** Aprenderás a dar estilo a la información que presentas en pantalla, haciendo uso de f-strings, una de las características más potentes y sencillas para manejar cadenas en Python. Con ellas, podrás incrustar variables y expresiones dentro de una cadena de texto de forma directa, controlar la cantidad de decimales al mostrar números, alinear textos y números en columnas, e incluso rellenar espacios con caracteres específicos para lograr un diseño más ordenado. De esta manera, tus reportes, resultados de cálculos y mensajes al usuario lucirán claros y consistentes, brindando una mejor experiencia.

- **Validación básica de datos ingresados por el usuario:** Veremos cómo asegurarnos de que los valores capturados mediante input() cumplan con lo que tu programa espera (por ejemplo, que sean numéricos, que estén dentro de cierto rango o que sigan un patrón particular). Para lograrlo, exploraremos el manejo de excepciones (como ValueError) y la verificación de rangos. Con este conocimiento, tu programa podrá detectar entradas inválidas, mostrar mensajes de error claros y, si lo deseas, solicitar al usuario que ingrese la información de nuevo. Esto fortalece la solidez del software, evitando bloqueos y reduciendo la posibilidad de resultados inadecuados por datos mal introducidos.

Al concluir esta clase, estarás en capacidad de crear programas secuenciales que no solo soliciten información al usuario, sino que también formateen la salida con f-strings para presentarla de manera clara y estética, al tiempo que verifiquen la validez de los datos antes de proceder. Este conjunto de técnicas te permitirá construir aplicaciones más flexibles y agradables de usar, sentando las bases para que, en futuras sesiones, incorpores estructuras de control (condicionales, bucles) y mejores aún más la complejidad y adaptabilidad de tus proyectos. De esta forma, tu software no solo funcionará de manera lineal y ordenada, sino que interactuará con el usuario de forma más efectiva, manejando entradas y salidas de datos con profesionalismo y robustez.

6.1. Formateo de salida con print() (ej: f-strings en Python).

6.1.1. Repaso de print()

La función print() es, por defecto, la vía más sencilla y directa para mostrar información al usuario en Python. Aunque su uso más elemental consiste en colocar las variables y textos separados por comas, generando espacios entre ellos, este método puede limitar la legibilidad y la flexibilidad cuando se requieren resultados más elaborados o consistentes. A continuación, profundizamos en los aspectos básicos de print() para comprender cómo funciona antes de adentrarnos en métodos más avanzados de formateo.

Múltiples argumentos.

Por defecto, print() puede tomar múltiples argumentos, separados por comas, que se muestran con un espacio entre ellos:

```
nombre = “Carlos”

edad = 30

print(“Hola,”, nombre, “. Tienes”, edad, “años.”)
```

En este caso, la consola mostrará:

```
Hola, Carlos . Tienes 30 años.
```

Observa que se inserta un espacio adicional entre “Carlos” y el punto, y otro entre el punto y “Tienes”, lo cual puede resultar en un formato menos pulcro si tenemos varias variables y textos mezclados.

Esto es suficiente para programas sencillos, pero puede volverse engorroso al requerir operaciones más sofisticadas, como formateo decimal o alineaciones.

Separador Personalizado (sep).

De manera opcional, print() permite especificar un separador distinto al espacio por defecto. Por ejemplo:

```
print(“Hola,”, “Carlos”, sep=” - “)
```

Produciría la salida:

```
Hola, - Carlos
```

Este parámetro resulta útil cuando deseamos unir varios elementos con un carácter distinto, como comas, guiones o incluso saltos de línea.

Terminación de Línea (end).

Por defecto, print() finaliza su salida con un salto de línea (\n). Sin embargo, puedes modificarlo con el parámetro end. Por ejemplo:

```
print("Hola,", end=" ")

print("Carlos")
```

En este caso, ambos textos se muestran en la misma línea, separándolos con un espacio en lugar de un salto de línea, quedando:

```
Hola, Carlos
```

Usos Sencillos en Programas Pequeños

- Para scripts simples, la sintaxis básica de print() con argumentos separados por comas suele ser suficiente. Permite concatenar variables sin tener que convertirlos explícitamente a cadenas, y, en muchos casos, este método cumple con las necesidades de presentación.
- Sin embargo, cuando empiezas a requerir mayor control sobre la forma en que se muestran los datos (por ejemplo, un número de decimales específicos, alineación de columnas, o inserción de caracteres de relleno), esta forma elemental se queda corta y puede complicar el código.

Limitaciones del Enfoque Básico

- Cuando un programa comienza a crecer y necesita presentar varios valores o resultados, la repetición de print() con comas puede volverse engorrosa. Ajustar manualmente los espacios o insertar cadenas intermedias para formatear números no es tan intuitivo, provocando que el código sea más difícil de mantener.

- Si necesitamos mostrar, por ejemplo, precios con dos decimales o tablas con columnas alineadas, el método tradicional de print("Texto", variable, "Texto") no provee una forma fácil de lograrlo sin añadir una gran cantidad de concatenaciones y conversiones.

- En ocasiones, se requieren operaciones directas dentro de la cadena (por ejemplo, mostrar edad + 5). Con el método básico, esto implica generar variables temporales o concatenar resultados convertidos a cadena, lo cual puede volverse repetitivo y propenso a errores.

Evolución hacia Métodos de Formateo Avanzado:

- Ante las limitaciones del método básico, Python introdujo otras formas de formatear la salida. Antiguamente, se utilizaba la sintaxis de formateo con el operador % y, más tarde, el método str.format(). Sin embargo, a partir de Python 3.6, f-strings se convirtieron en la opción más clara y potente. En las siguientes secciones, veremos cómo los f-strings permiten incrustar variables, controlar decimales y alinear contenido sin sacrificar la legibilidad.

En definitiva, el uso elemental de print() con comas es una forma rápida de concatenar texto y variables para scripts pequeños o demostraciones simples. No obstante, a medida que un proyecto crece y requiere una salida más refinada, es esencial transicionar hacia métodos más avanzados.

Para complementar y profundizar, se sugieren acceder a este recurso:

- Título del enlace relacionado: Curso de Python 3.13.1 Función Print
- **Descripción del enlace relacionado:** El video explica el uso de la función print() en Python 3.13.1, explicando su sintaxis básica, parámetros clave.
- **Enlace:** https://www.youtube.com/watch?v=tKeSH5h1Itw

6.1.2. Introducción a los f-Strings

Los f-strings representan una de las formas más claras y poderosas para formatear cadenas en Python, al permitir incrustar variables y expresiones directamente dentro de llaves {}. Este enfoque resulta en un código más legible, al eliminar la necesidad de concatenaciones tediosas o el uso de placeholders complicados. En lugar de escribir:

```
nombre = “Carlos”

edad = 30

print(“Hola, “ + nombre + “. Tienes “ + str(edad) + “ años.”)
```

podemos simplemente usar:

```
nombre = “Carlos”

edad = 30

print(f”Hola, {nombre}. Tienes {edad} años.”)
```

Este método mejora la legibilidad y reduce la propensión a errores de concatenación. Además, admite expresiones dentro de {}, como:

```
print(f”El doble de tu edad es {edad * 2}.”)
```

A partir de Python 3.6, se introdujo este nuevo método de formateo, donde se antepone la letra f a la cadena de texto, habilitando la posibilidad de evaluar el contenido que va dentro de las llaves {}. Dicho contenido puede ser una variable (por ejemplo, {nombre}), una expresión ({edad * 2}) o incluso una llamada a función ({mi_funcion(param)}). Esto permite escribir código que sea tanto conciso como explícito, ya que el texto y las variables se mezclan de forma natural, evitando la complejidad de concatenar múltiples fragmentos de texto y funciones de conversión.

Ventajas de los f-Strings:

- El código se lee casi como una oración, integrando las variables en la misma cadena sin generar confusión ni interrupciones. Esto reduce los errores que surgen al olvidar convertir variables numéricas a cadenas o al insertar espacios adicionales.
- Los f-strings permiten formatear números (decimales, porcentajes, etc.), alinear texto, controlar el número de decimales y mucho más. Con una sintaxis como :.2f, por ejemplo, se puede mostrar un número flotante con exactamente dos decimales.
- Es posible colocar dentro de {} operaciones como {edad + 5}, funciones como {len(nombre)} o incluso condiciones simples, haciendo que el mensaje sea dinámico y se adapte a la situación sin necesidad de pasos intermedios..

Limitaciones y Consideraciones:

- Los f-strings solo están disponibles a partir de Python 3.6. En versiones anteriores, es necesario usar otros métodos de formateo como %-formatting o str.format().
- Aunque puedes incluir cálculos dentro de {}, se recomienda mantener estas expresiones cortas y claras. Colocar operaciones muy largas o llamadas anidadas puede dificultar la lectura y complicar la depuración.
- Si requieres realizar cálculos extensos, hazlos en líneas anteriores y solo inserta el resultado en la cadena, garantizando que el código mantenga su simplicidad y se comprenda de un vistazo.

Usar f-strings facilita la presentación de datos, otorgando un código más limpio y legible.

```
# Solicitar datos al usuario
nombre = input("Ingrese su nombre: ")
edad = int(input("Ingrese su edad: "))

# Realizar el cálculo: edad + 5
resultado = edad + 5

# Imprimir resultados utilizando f-strings
print(f"Nombre del usuario: {nombre:^20}")        # Alineado al centro en un campo de 20 cara
print(f"Edad + 5 = {resultado:>10}")               # Alineado a la derecha en un campo de 10
print(f"Resultado formateado: {resultado:.2f}")    # Formateado con 2 decimales
```

Imagen 1: Ejemplo de f-Strings con Variables y Expresiones

Damián Nicolalde Rodríguez. (2024). Ejemplo de f-Strings con Variables y Expresiones.

Ejemplos Prácticos de f-Strings

A continuación, se muestra cómo los f-strings pueden simplificar distintas tareas de formateo:

```
# Ejemplo 1: Inserción directa de variables
nombre = “Carlos”
edad = 30
print(f”Hola, {nombre}. Tienes {edad} años.”)

# Ejemplo 2: Expresiones dentro de {}
print(f”El doble de tu edad es {edad * 2}.”)

# Ejemplo 3: Formato decimal
pi = 3.14159
```

```
print(f"Pi con 2 decimales: {pi:.2f}")


# Ejemplo 4: Alineación de texto y relleno
codigo = 42
print(f"Código: {codigo:04d}")  # Rellena con ceros hasta 4 dígitos


# Ejemplo 5: Llamada a función dentro de {}
def saludar(nombre):
    return f"Hola, {nombre}"


print(f"Función saludar: {saludar('Ana')}")
```

Importancia en la Ejecución Secuencial

Al utilizar f-strings en un programa secuencial, la lógica del programa fluye de manera lineal: primero se definen o calculan las variables, luego se imprimen con la sintaxis de f-strings. Este enfoque facilita que, en programas más largos, cada paso sea transparente: se ve claramente cuándo se obtiene un valor y cómo se muestra. Cuando combinamos esto con validaciones de datos, la secuencialidad se vuelve aún más clara: primero se valida, luego se formatea la salida si todo está en orden.

6.1.3. Opciones Avanzadas de Formateo

Los f-strings no solo simplifican la inserción de variables y expresiones en cadenas, sino que también ofrecen una gran variedad de parámetros y códigos de formato que permiten personalizar la presentación de la información. A continuación, profundizamos en algunas de las opciones más útiles, que resultan esenciales para elaborar reportes, tablas o mensajes de salida con un nivel profesional de detalle y consistencia.

- **Limitar Decimales:**

Uno de los casos más frecuentes en la programación es controlar la cantidad de decimales que se muestran al imprimir números de tipo float. Con los f-strings, podemos especificar el número de decimales a través de la sintaxis :.Nf, donde N es la cantidad deseada:

```
pi = 3.1415926535

print(f"Pi con 4 decimales: {pi:.4f}")
```

Permite mostrar 3.1416 en lugar de muchos dígitos.

- **Alineación de Textos y Números:**

Con f-strings, es sencillo alinear el contenido de una variable dentro de un ancho fijo, lo cual resulta invaluable para la creación de tablas o reportes ordenados.

Por ejemplo, para alinear a la derecha en un campo de 5 caracteres:

```
numero = 42

print(f"|{numero:>5}|")  # Alineado a la derecha con 5 espacios
```

Esto produce algo como | 42|, donde el número queda a la derecha y se rellena el espacio restante con espacios en blanco. Esto resulta muy útil en la impresión de tablas o reportes.

Asimismo, se puede alinear a la izquierda (:<5) o centrar (:^5). Este control es sumamente útil cuando se tienen múltiples columnas (por ejemplo, nombre, precio, cantidad) y se desea que los valores se presenten de forma consistente en cada columna. De otro modo, la información puede verse desalineada y confusa, dificultando la comparación visual de datos.

- **Relleno con Caracteres:**

Además de los espacios, Python permite rellenar los huecos con otros caracteres para mejorar la legibilidad o destacar secciones de un reporte. Por ejemplo:

```
palabra = "Hi"

print(f"|{palabra:*<8}|")  # Alineado a la izquierda y rellenado con '*'
```

Aquí, * se utiliza para llenar el ancho restante de 8 caracteres, generando la salida |Hi******|. Esto se vuelve muy útil cuando queremos resaltar secciones, crear bordes o distinguir diferentes partes de un reporte.

Otros caracteres de relleno pueden ser 0 (cero), -, . o cualquier símbolo que ayude a la identificación de los datos.

Combinación de Formateo y Expresiones:

Las opciones anteriores pueden combinarse para lograr resultados más complejos. Por ejemplo, limitar decimales y alinear a la derecha en un espacio de 10 caracteres:

```
precio = 1599.5

print(f"Precio final: {precio:>10.2f}")
```

Esto imprime el valor en 10 espacios, alineado a la derecha, con dos decimales (1599.50). Dicha flexibilidad es esencial cuando se requiere un aspecto coherente y uniforme en las salidas, especialmente en entornos donde la presentación de datos debe ser clara (sistemas de facturación, reportes contables, tablas de estadísticas, etc.).

Tabla 1: Ejemplos de Formateo Avanzado con f-Strings

Funcionalidad	**Ejemplo de Código**	**Resultado**
Decimales limitados	valor = 2.7182818 print(f"E: {valor:.3f}")	E: 2.718
Alineación a la derecha (5 espacios)	num = 7 print(f"{num:>5}")	7 (cuatro espacios + 7)
Alineación al centro (10 espacios)	texto = "Python" print(f"{texto:^10}")	Python (centrado en 10 espacios)

Relleno con ceros	codigo = 42 print(f"{codigo:05d}")	00042
Expresión directa	print(f"La suma de 2 y 3 es {2+3}")	La suma de 2 y 3 es 5

Damián Nicolalde Rodríguez. (2024).

Como podemos ver, las opciones avanzadas de formateo en f-strings ofrecen un control muy detallado sobre cómo se muestran los datos. Desde la cantidad de decimales hasta la alineación y relleno, estas características aportan un nivel de profesionalismo y usabilidad que trasciende la simple concatenación de cadenas. Cuando se combina con la lógica secuencial, se garantiza que al final de un flujo de operaciones, la información aparezca en la pantalla de forma clara, ordenada y sencilla de interpretar.

```python
# Lista de productos y sus precios
productos = [
    ("Manzanas", 1.5),
    ("Naranjas", 2.0),
    ("Plátanos", 0.75),
    ("Fresas", 3.99)
]

# Imprimir encabezado de la tabla
print(f"{'Producto':>15} | {'Precio':>10}")
print("-" * 30)

# Recorrer la lista e imprimir cada producto y precio formateado
for producto, precio in productos:
    print(f"{producto:>15} | {precio:>10.2f}")
```

Imagen 2: Salida Formateada de una Tabla de Precios

Damián Nicolalde Rodríguez. (2024). Salida Formateada de una Tabla de Precios.

6.2. Validación básica de datos ingresados por el usuario.

La función input() permite que el usuario ingrese datos, pero no garantiza que estos sean correctos. Un programa robusto requiere verificar los valores y manejar posibles errores, evitando bloqueos o resultados incoherentes.

6.2.1. Manejo de Excepciones con try-except

Cuando intentamos convertir a int o float un texto no numérico, Python lanza un ValueError. Para evitar que el programa se bloquee, usamos bloques try-except:

```
try:
    edad = int(input("Ingrese su edad: "))
    print("Edad aceptada:", edad)
```

```
except ValueError:
    print(“Error: Debe ingresar un número entero.”)
```

Este método captura el error y muestra un mensaje claro, sin detener la ejecución abruptamente.

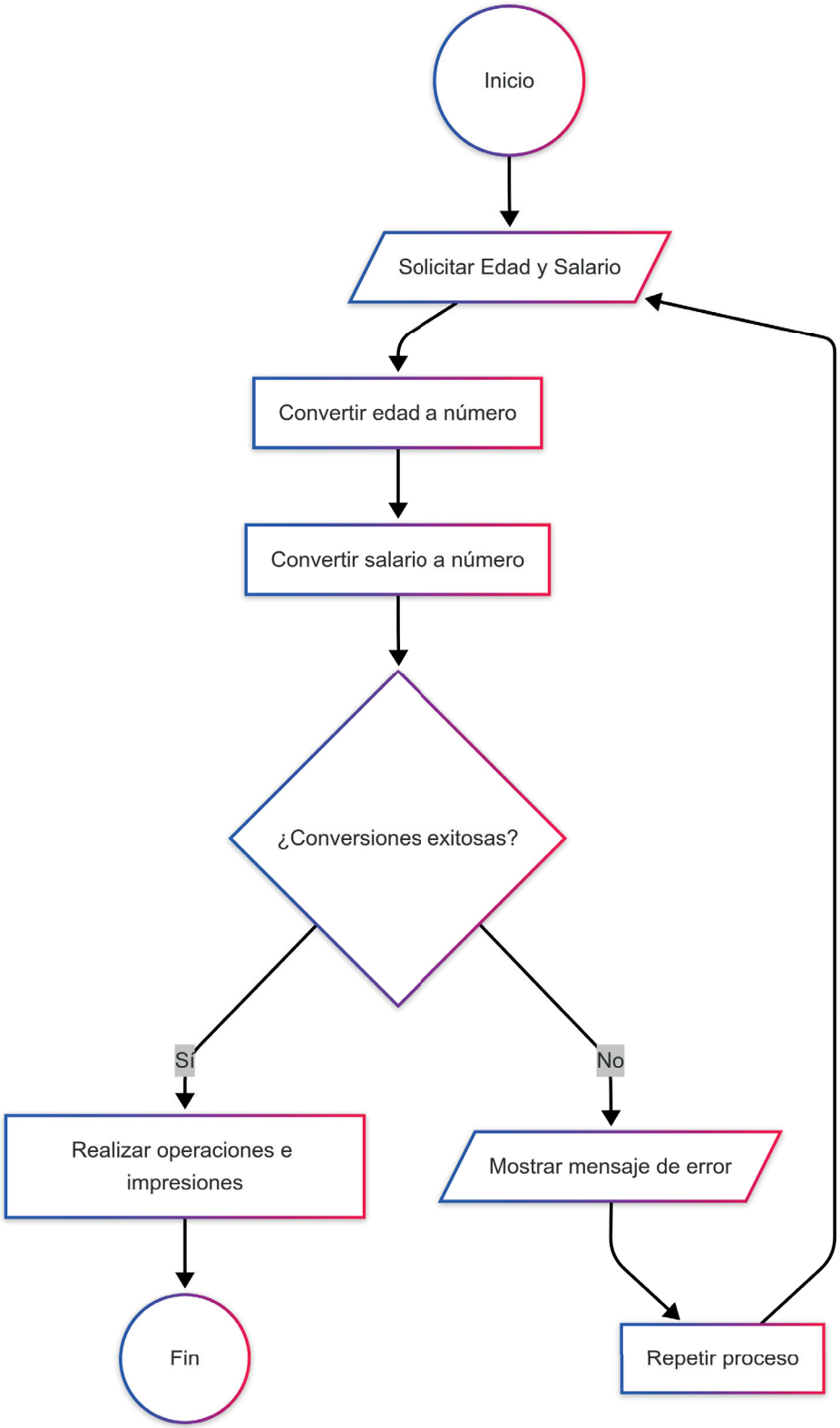


Imagen 3: Diagrama de Flujo para Validar Edad y Salario

Damián Nicolalde Rodríguez. (2024). Diagrama de Flujo para Validar Edad y Salario.

A menudo, al validar datos ingresados por el usuario, surgen situaciones o errores comunes que pueden bloquear el programa o confundir al usuario si no se manejan de forma adecuada. Estos errores van desde no capturar excepciones específicas (como ValueError) hasta presentar mensajes poco claros que no orientan correctamente a la persona que ingresa los datos. A continuación, se presenta una tabla que enumera los problemas más frecuentes y ofrece soluciones o recomendaciones para cada caso, ayudando a que la validación sea efectiva y el programa mantenga su robustez y legibilidad.

Tabla 2: Errores Comunes y Soluciones en Validación

Error Común	**Descripción**	**Solución o Recomendación**
No Manejar Excepciones (ValueError)	El programa se cierra al recibir texto en lugar de un número.	Usar try-except ValueError para capturar el error y pedir datos de nuevo.
Mensajes Confusos al Usuario	El usuario no sabe qué se espera, ingresando información inadecuada.	Mostrar mensajes descriptivos (ej. "Ingrese un número entero para su edad.").
Falta de Verificación de Rango	Se aceptan valores ilógicos (ej. edad = -10, edad = 999).	Comparar los datos con un rango razonable y mostrar un mensaje si se exceden.
Confundir la Lógica de Validación y Cálculo	Se mezclan validaciones con operaciones, dificultando la lectura del código.	Separar la validación en funciones o bloques distintos, manteniendo la claridad de cada paso del programa.

Damián Nicolalde Rodríguez. (2024).

6.2.2. **Verificación de Rango y Patrones**

Además de verificar que un valor sea numérico, podemos comprobar si está dentro de un rango lógico. Por ejemplo:

```
edad = int(input("Ingrese su edad: "))
if 0 <= edad <= 120:
    print("Edad válida.")
else:
    print("Edad fuera de rango.")
```

Esto evita datos absurdos y enriquece la confiabilidad del programa. Para cadenas, podemos comprobar longitud o patrones (por ejemplo, correos electrónicos) mediante expresiones regulares, aunque eso va más allá de la validación básica.

6.2.3. Ejemplos de Validación Combinada

Podemos encadenar validaciones. Primero, verificamos si es numérico; luego, si el rango es correcto:

```
try:
    salario = float(input("Ingrese su salario: "))
    if salario < 0:
        print("Error: El salario no puede ser negativo.")
    else:
```

```
        print(f"Salario aceptado: {salario:.2f}")
except ValueError:
    print("Debe ingresar un valor numérico (ej. 1500.00).")
```

De esta forma, cubrimos tanto la conversión de tipo como la coherencia del dato.

Validar entradas es esencial para la estabilidad y usabilidad de cualquier aplicación, evitando bloqueos y confusiones.

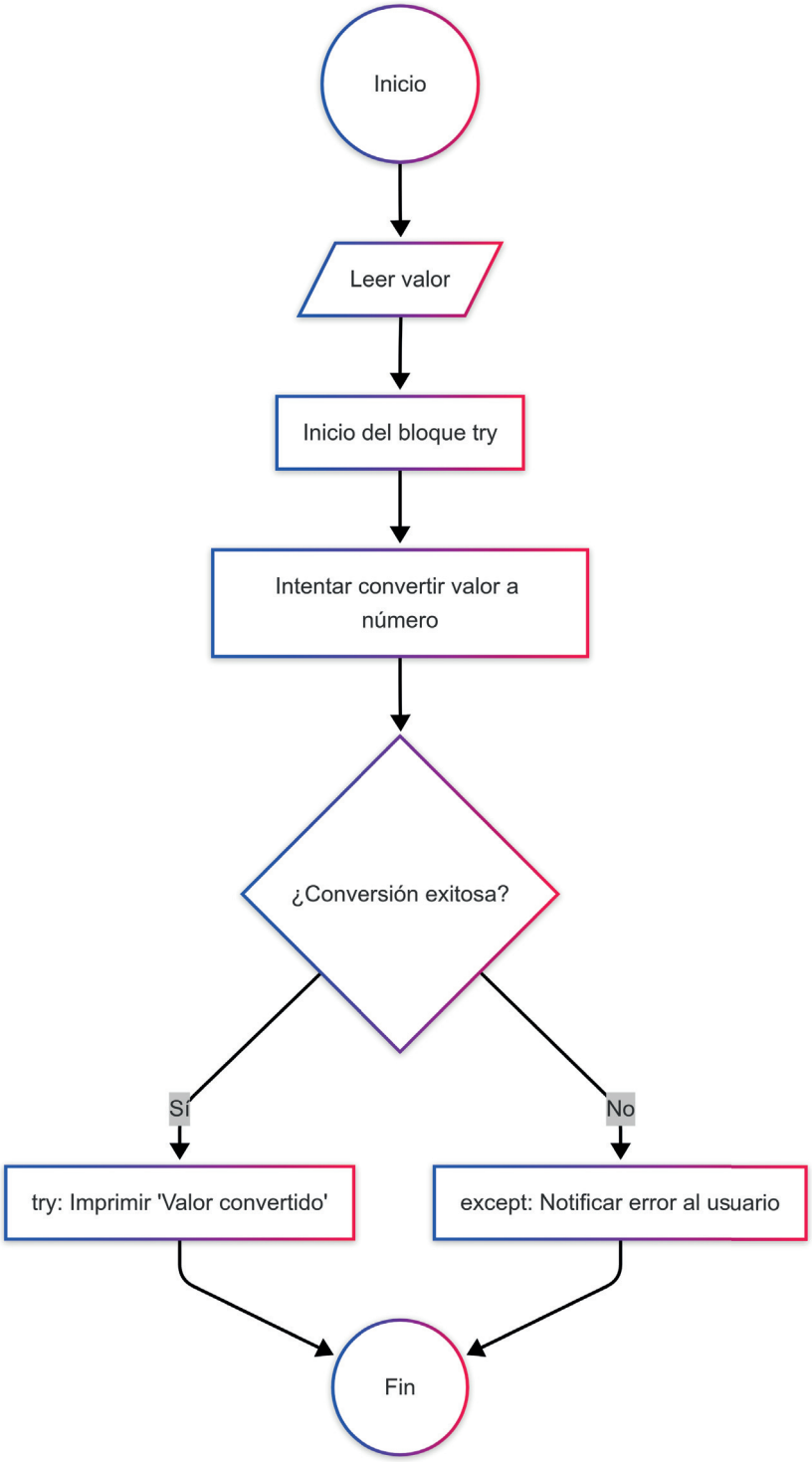


Imagen 4: Ilustración de Bloque try-except en Validación

Damián Nicolalde Rodríguez. (2024). Ilustración de Bloque try-except en Validación.

Para integrar el formateo con f-strings y la validación de datos, se presenta la siguiente tabla de ejemplos.

Tabla 3: Ejemplos Combinados de Formateo y Validación

Ejemplo	Código	Explicación
Formateo de Datos Validado	try: precio = float(input("Ingrese el precio: ")) print(f"Precio con 2 decimales: {precio:.2f}") except ValueError: print("Error: Debe ingresar un valor decimal.")	Si el usuario ingresa un número válido, se imprime con 2 decimales; en caso de error, se muestra un mensaje claro.

Verificación de Rango + f-Strings	nota = float(input("Nota: "))
	
	if 0 <= nota <= 10:
	
	print(f"Su nota es {nota:.1f}, valor válida.")
	
	else:
	
	print("Nota fuera de rango (0-10).")
		Se valida el rango de la nota (0 a 10) y se imprime con 1 decimal. En caso de exceder el rango, se informa al usuario.

Texto y Validación de Longitud	nombre = input("Nombre: ")
	
	if len(nombre.strip()) == 0:
	
	print("Error: Nombre vacío.")
	
	else:
	
	print(f"Bienvenido, {nombre}")
		Comprueba que el nombre no esté vacío y luego lo imprime de manera formateada, subrayando la utilidad de la validación secuencial.

Damián Nicolalde Rodríguez. (2024).

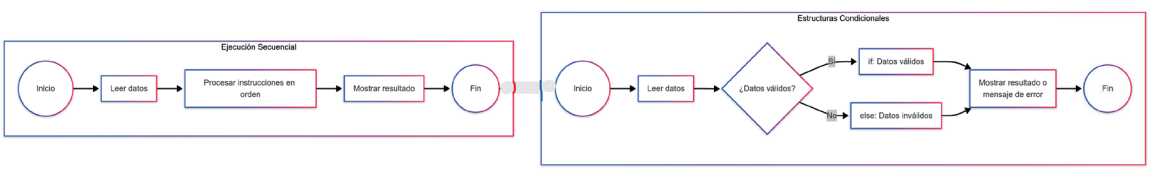


Imagen 5: Diferencias entre Ejecución Secuencial y Estructuras Condicionales

Damián Nicolalde Rodríguez. (2024). Diferencias entre Ejecución Secuencial y Estructuras Condicionales.

Para cerrar esta sección, se presenta una tabla que compara métodos de formateo (f-strings, str.format(), concatenación) y estrategias de validación (bloques try-except, comprobaciones con isdigit(), etc.):

Tabla 4: Comparativa de Métodos de Formateo y Validación

Aspecto	**f-Strings**	**str.format()**	**Concatenación Básica**	**Validación con try-except**	**Validación con Métodos (isdigit())**
Sintaxis	f”Hola, {nombre}”	“Hola, {}”.format(nombre)	“Hola, “ + nombre	try: ... except ValueError:	if entrada.isdigit(): ... else: ...
Legibilidad	Alta, variables y texto se integran en la misma cadena.	Moderada, requiere placeholders {} y métodos.	Baja, se tiende a sumar muchas cadenas y variables.	Alta, si se mantiene el código organizado.	Moderada, puede requerir checks adicionales.
Flexibilidad	Permite formatear números, fechas, expresiones directamente.	Permite formato pero la sintaxis es más extensa.	Muy limitada, no es fácil formatear números.	Captura múltiples tipos de errores en un solo bloque.	Solo chequea un patrón (por ejemplo, dígitos), no excepciones.
Uso Recomendado	A partir de Python 3.6, es la opción preferida.	Para versiones <3.6 o cuando se requiere una sintaxis distinta.	Programas muy simples o versiones muy antiguas.	Ideal para entradas numéricas y control de errores.	Verificación rápida de que el texto cumpla un criterio.
Desventajas	No disponible en Python <3.6.	Puede volverse confuso con muchos placeholders.	Difícil de mantener al crecer el programa.	Requiere bloque try-except para cada tipo de error.	No maneja rangos ni excepciones, solo checks básicos.

Damián Nicolalde Rodríguez. (2024).

Imagen 6: Representación Visual de Rango de Datos Válidos

Damián Nicolalde Rodríguez. (2024). Representación Visual de Rango de Datos Válidos.

Para complementar y profundizar, se sugieren acceder a este recurso:

- **Título del enlace relacionado**: La validación de datos en Python para principiantes
- **Descripción del enlace relacionado:** Video práctico que enseña métodos para validar datos en Python, manejo de excepciones (try-except) y validación personalizada con funciones. Incluye ejemplos útiles (formularios, entradas de usuario) y buenas prácticas para evitar errores comunes.
- **Enlace:** https://www.youtube.com/watch?v=hKOMUsqKQcY

Referencias citadas en la Clase 6.

- https://elibro.puce.elogim.com/es/ereader/puce/230298
- https://puce.odilo.us/info/facil-aprendizaje-estructuras-de-datos-algoritmos-c-aprenda-facilmente-estructuras-de-datos-graficamente-03127232
- https://puce.odilo.us/info/aprende-c-en-un-fin-de-semana-03105596

Definición de los términos citados en la Clase 6.

La interacción con el usuario: Proceso mediante el cual un programa solicita y recibe datos de la persona que lo utiliza (a través de funciones como input()) y muestra resultados o mensajes en pantalla (usando print() u otros métodos). Este intercambio influye directamente en la usabilidad y efectividad de la aplicación, ya que un programa con interacciones claras, mensajes descriptivos y validaciones adecuadas resulta más accesible y confiable para el usuario.

Programa robusto: Se refiere a un software capaz de manejar condiciones inesperadas, datos inválidos o escenarios anómalos sin colapsar ni producir resultados incoherentes. Un programa robusto no solo captura y gestiona los errores con mensajes claros, sino que también sigue funcionando de manera estable, ofreciendo seguridad y confiabilidad a los usuarios, incluso ante entradas equivocadas o problemas en tiempo de ejecución.

Profundización Clase 6.

Recurso_profundizacion_clase6.docx

PUCE Virtual

La excelencia no se improvisa

síguenos