Módulo 2. Conceptos básicos de programación

Ya conocemos lo que es un algoritmo, pero ¿cómo se relaciona esto con la generación de programas computacionales?

Antes de dar este paso, es necesario estandarizar la forma en que representamos y expresamos el algoritmo, para poder evitar ambigüedades al momento de ejecutar.

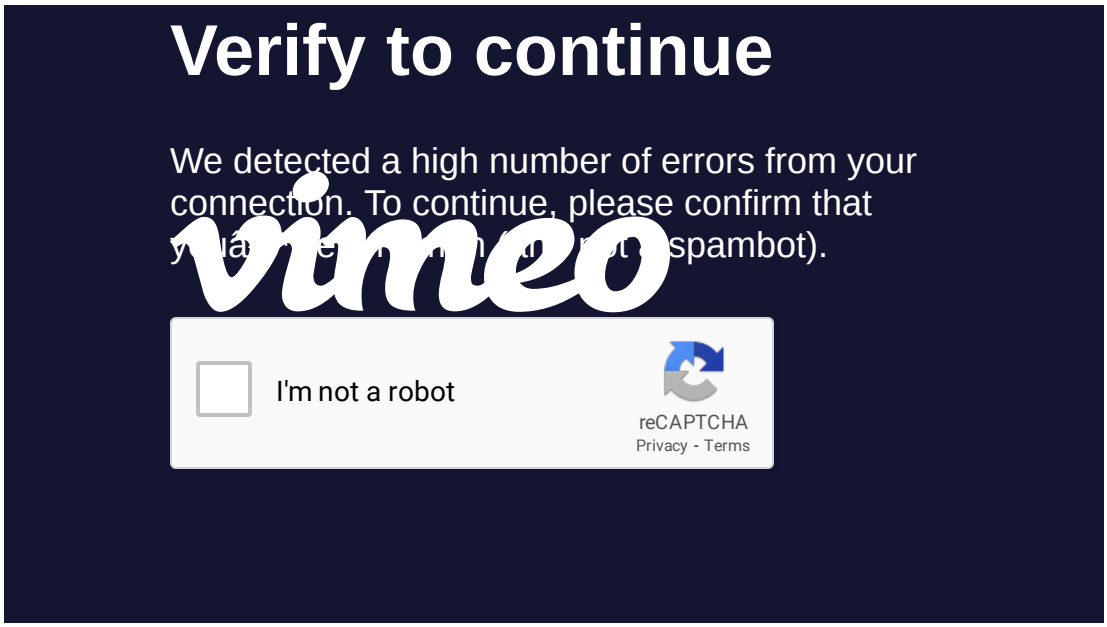
Para esto, revisaremos el pseudocódigo y los diagramas de flujo, dos maneras diferentes de representar algoritmos, para que sean entendidos por cualquier persona, pero que nos brindarán una estructura base al momento de diseñarlos.

La forma en que definiremos nuestra solución y las estructuras de control que podemos utilizar para llevar a cabo nuestro objetivo serán analizadas en esta unidad.

Por lo tanto, al terminar de revisar este módulo, tendrás un conocimiento más acabado de las herramientas que están a tu disposición para el diseño y construcción de un algoritmo.

Lección 1 de 6

Video de inmersión



CONTINUAR

Unidad 1. Representación gráfica de los procesos

Representar algoritmos o procesos puede ser algo difícil. No siempre se pueden encontrar las palabras adecuadas y, en muchos casos, sería muy difícil de entender o muy largo de leer si solamente se utilizan palabras. Es por ello que se creó un lenguaje de símbolos, para poder representar gráficamente los procesos mediante diagramas.

Otra forma de comunicar los procesos o algoritmos es mediante un pseudocódigo, que puede definirse como un conjunto de sentencias que no corresponden a ningún lenguaje de programación específico, pero que denotan la lógica que se debe seguir por dicho algoritmo.

Tema 1. Origen y características

Inicialmente, el proceso de programación se realizaba en lenguajes de programación más cercanos a la computadora, lenguajes de más bajo nivel (lenguaje máquina). Los datos están codificados en sistema binario, y no es necesaria la traducción. Entre las características del lenguaje máquina, cabe destacar las siguientes:

- Es dependiente de los recursos de la computadora; por lo tanto, para programar, el programador debe conocer la arquitectura sobre la que se programa.
- El programador se encarga de verificar que no existan errores sintácticos, pues no existe un compilador que los detecte.
- El programador trabaja directamente con direcciones de memoria.

El lenguaje ensamblador surgió como evolución natural, donde a cada secuencia de ceros y unos se le asocia un nombre nemotécnico. Estos nombres necesitan traducción, que se realiza mediante un programa que se llama como el lenguaje: ensamblador. Aunque fue un gran avance, todavía es necesario conocer cómo está constituida y qué recursos tiene la computadora.

Más tarde, se fueron asociando nombres a conjuntos de instrucciones que realizaban una tarea compleja determinada y programaban de manera independiente la computadora donde se iba a ejecutar el código. Nacen los lenguajes de alto nivel y se encuentran más cercanos a la forma de pensar de los humanos que al lenguaje que entiende la máquina.

La programación, de la mano de los lenguajes, fue evolucionando gracias a cuatro causas o motores que la impulsan. Estas son las siguientes:

- **Abstracción:** es el proceso mental por el que el ser humano extrae las características esenciales de algo e ignora los detalles superfluos. Es esencial para modelar el mundo real. En un

principio, se hacían programas pensando como una computadora. En la actualidad, se solucionan los problemas sin conocer la máquina donde va a ser ejecutado el programa.

- **Encapsulación:** es el proceso por el que se ocultan los detalles de las características de una abstracción. En programación, es esencial para reutilizar un código. Si se ocultan los detalles de cómo está hecho un programa, pero se conoce el modo de funcionamiento, se puede utilizar en cualquier otro programa.

- **Modularidad:** es el proceso de descomposición de un sistema en un conjunto de elementos poco acoplados (independientes) y cohesivos (con significado propio). Es esencial para abordar la resolución de problemas extensos o complicados de un modo más simple y organizado.

- **Jerarquía:** es el proceso de estructuración por el que se organiza un conjunto de elementos en distintos niveles, atendiendo a determinados criterios (responsabilidad, composición, etc.).

> ⓘ **A medida que se fueron añadiendo nuevas características al proceso y a las herramientas, fueron surgiendo múltiples estilos de programación.**

Características de la programación

Es importante tener algunas consideraciones para que la programación sea de buena calidad y los resultados sean o se acerquen lo más posible a los esperados. Para esto, se pueden enunciar las siguientes características que se deben tener en cuenta durante la programación de aplicaciones:

Eficacia

El programa ejecuta correctamente las tareas definidas por su especificación y satisface los objetivos de los usuarios. Un programa exacto regresa el resultado correcto del cálculo que hace o lleva a cabo la tarea requerida de la forma esperada.

Eficiencia

El programa hace un uso adecuado y no malgasta los recursos de la computadora, como la memoria y el tiempo de procesamiento. Un programa eficiente completará la tarea con mayor rapidez con respecto a otro programa que no lo es.

Integridad o completitud

Un programa es completo solo si ejecuta todas las operaciones que se codifican en los algoritmos al procesar un conjunto de datos.

Documentación

Consiste en el uso de documentos o herramientas auxiliares que expliquen cómo ocurre el procesamiento de los datos en un programa. Facilita el diseño y el entendimiento del programa.

Usabilidad

El programa es fácil de usar si las personas a las que está destinado pueden usarlo para realizar sus tareas de forma cómoda y sin esfuerzos innecesarios.

Mantenibilidad

El código fuente del programa permite localizar y corregir defectos rápidamente, así como también permite hacer cambios que resultan más fáciles para adaptarlo a las necesidades cambiantes de los usuarios.

Fiabilidad

Un programa es fiable si realiza sus tareas cuando es necesario y con la precisión requerida.

Tema 2. Conceptos fundamentales

La generación de un programa o un ***software*** necesita una metodología como modelo para lograr realizar los algoritmos y resolver el problema. Esta metodología llamada “ciclo de desarrollo del ***software***” consta de una serie de pasos lógicos secuenciales denominados “fases” y que son los siguientes:

1. Definición del problema
2. Análisis del problema
3. Diseño de la solución
4. Codificación
5. Prueba y depuración
6. Documentación
7. Implementación
8. Mantenimiento

Es importante conocer cada una de estas fases, ya que están íntegramente relacionadas. El resultado que producen sirve de entrada para el comienzo de la siguiente. Si bien la codificación es la fase que se encuentra naturalmente ligada con el programador, seguramente deba, además, realizar actividades de las demás. Cada rol involucrado en este proceso debe conocer e interactuar con cada una de las etapas.

Definición del problema

Conocer el problema es la primera consideración. Saber quién será el usuario final también es importante. La determinación de las entradas y salidas consiste en establecer la siguiente: cómo funcionará el programa y qué datos se necesitan para que esto suceda. Después de que esto se haya decidido, la factibilidad será la siguiente consideración. Finalmente, si el proyecto está listo, se deben tomar medidas para garantizar que el proyecto esté debidamente documentado y analizado.

Análisis del problema

Es la comprensión completa del problema con sus detalles. Es un requisito para lograr una solución eficaz.

Diseño de la solución

Es momento de comenzar a diseñar y modelar los algoritmos. Una computadora no tiene la capacidad para solucionar más de lo que se le indica en los algoritmos. Estos algoritmos indican las instrucciones para que la máquina ejecute. La información proporcionada al algoritmo constituye su entrada, y la información producida por el algoritmo constituye su salida.

Los problemas complejos se pueden resolver más eficazmente cuando se dividen en subproblemas más fáciles de solucionar que el original. A la descomposición del problema original en subproblemas más simples le sigue la división de estos subproblemas en otros más simples aún. Estas divisiones se realizan hasta que los subproblemas resultantes sean lo más pequeños posibles y permitan realizar sus respectivos diseños.

Dos formas comunes de diseñar la solución a un problema son dibujar un diagrama de flujo y escribir un pseudocódigo, o posiblemente ambas.

Codificación

Este paso consiste en empezar a escribir el código del programa, es decir, expresar la solución en un lenguaje de programación, traducir la lógica del resultado de la fase anterior a un lenguaje de programación. Como ya hemos señalado, un lenguaje de programación es un conjunto de reglas que proporciona una forma de instruir a la computadora qué operaciones realizar.

Prueba y depuración

Algunos expertos insisten en que un programa bien diseñado se puede escribir correctamente la primera vez. De hecho, afirman que hay formas matemáticas de demostrar que un programa es correcto. Sin embargo, las imperfecciones del mundo existen, por lo que la mayoría de los

programadores se acostumbran a la idea de que sus programas recién escritos probablemente tengan algunos errores. Esto es un poco desalentador al principio, ya que los programadores tienden a ser personas precisas, cuidadosas y orientadas a los detalles, personas que se enorgullecen de su trabajo. Aun así, hay muchas oportunidades para introducir errores en los programas. Por esta razón los programas deben probarse en un ambiente controlado, dedicado a tal fin.

La depuración es un término usado ampliamente en programación: significa detectar, localizar y corregir errores, generalmente, ejecutando el programa. En esta fase, se ejecuta el programa, manual o automáticamente, utilizando los datos de prueba diseñados previamente (casos de prueba). Se deben planificar los datos y casos de la prueba cuidadosamente para asegurarse de que se chequea cada parte del programa.

Documentación

Documentar es un proceso continuo y necesario. Sin embargo, como les pasa a muchos programadores, podés sentirte ansioso por realizar actividades más emocionantes centradas en la computadora. La documentación es una descripción detallada por escrito del ciclo de programación y hechos específicos sobre el programa. Los materiales típicos de documentación del programa incluyen el origen y la naturaleza del problema, una breve descripción narrativa del programa, herramientas lógicas como diagramas de flujo y pseudocódigos, descripciones de registros de datos, listas de programas y resultados de pruebas. Los

comentarios en el programa en sí también se consideran una parte esencial de la documentación. Muchos programadores documentan mientras codifican. En un sentido más amplio, la documentación del programa puede ser parte de la documentación de un sistema completo.

El programador inteligente continúa documentando el programa a lo largo de su diseño, desarrollo y prueba. Se necesita documentación para complementar la memoria humana y para ayudar a organizar la planificación del programa. Además, la documentación es importante para comunicarse con otras personas interesadas en el programa, especialmente, otros programadores que pueden ser parte de un equipo de programación.

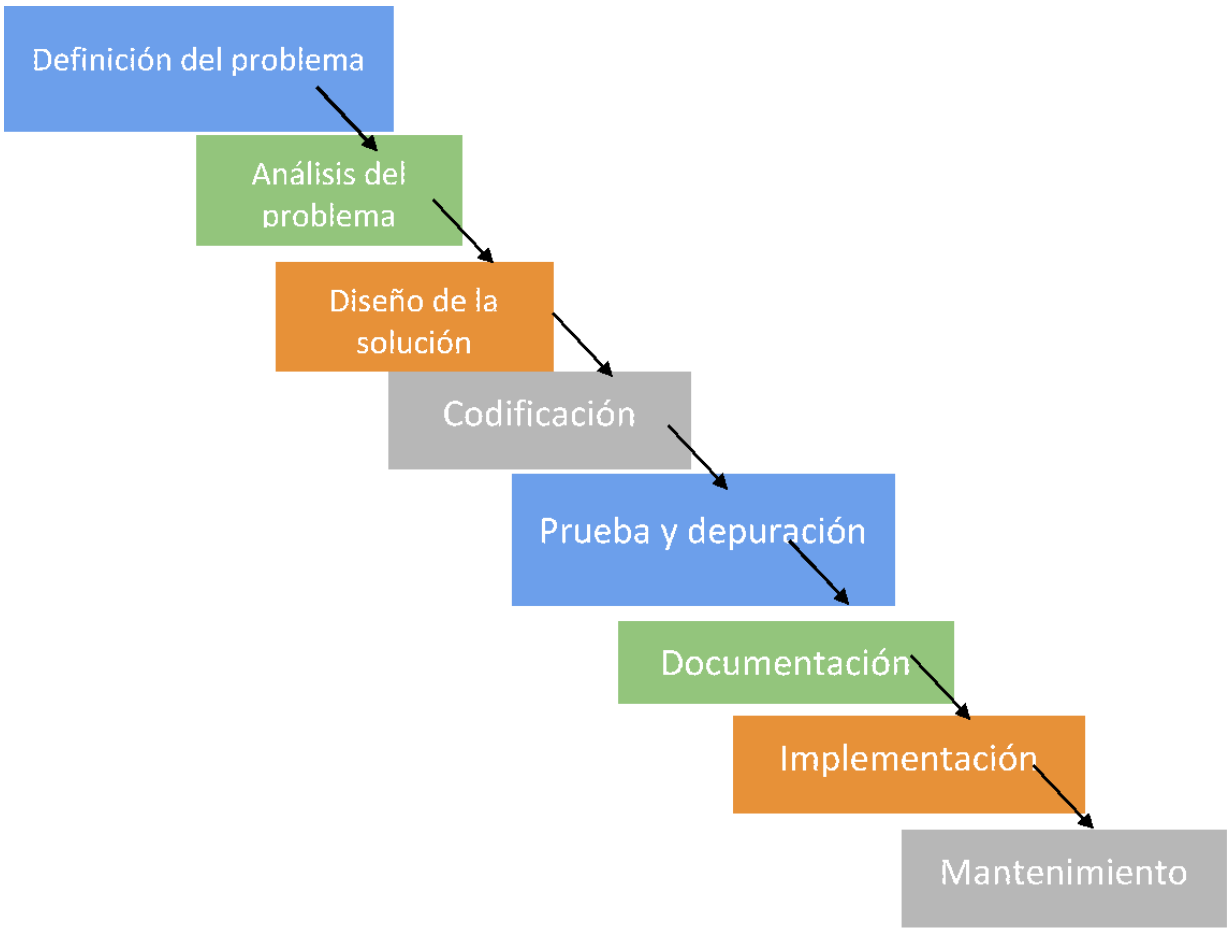
Implementación

El programa ya probado, revisado y mejorado se considera terminado y puede utilizarse con un alto grado de confianza para resolver los problemas que dieron origen a su creación. Si se está automatizando alguna tarea manual, esta última se desecha para emplear solamente el programa.

Mantenimiento

Es la fase de mayor duración, siempre y cuando el programa funcione bien. Este debe ser revisado cada cierto tiempo para realizar ajustes si es necesario.

Figura 1: Fases de un proyecto de *software*



Fuente: elaboración propia.

Tema 3. Diagrama de flujos y pseudocódigo

El pseudocódigo es considerado un lenguaje de alto nivel, es mucho más entendible para para los humanos que un lenguaje de programación. Sin embargo, cuenta con una estructura similar a un código.

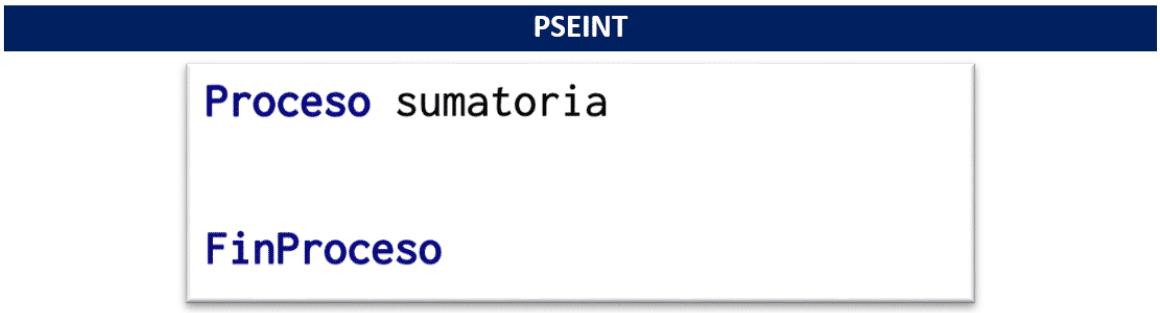
Los pseudocódigos no se compilan ni interpretan por ninguna computadora. Su propósito es, simplemente, representar un algoritmo o un proceso mediante una sintaxis similar a la presente en los lenguajes de programación. Sin embargo, se han creado aplicaciones que permiten simular la ejecución de nuestro algoritmo. Nosotros utilizaremos la aplicación PSeInt para poder ejemplificar cada una de las secciones del algoritmo en pseudocódigo. Puedes encontrarla en el siguiente link: http://PseInt.sourceforge.net/

Inicio / Fin

Como revisamos en el módulo anterior, un algoritmo es finito, es por eso que siempre debe tener un comienzo y un fin.

En PSeInt, este se ve reflejado por el nombre del algoritmo en el inicio y por un ***FinProceso*** al terminar el código. En algunos lenguajes, se utilizan las llaves para agrupar el código. Y otros lenguajes, como Python, utilizan la indentación.

Figura 2: Estructura de un algoritmo PSeInt



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Variables y tipos de datos

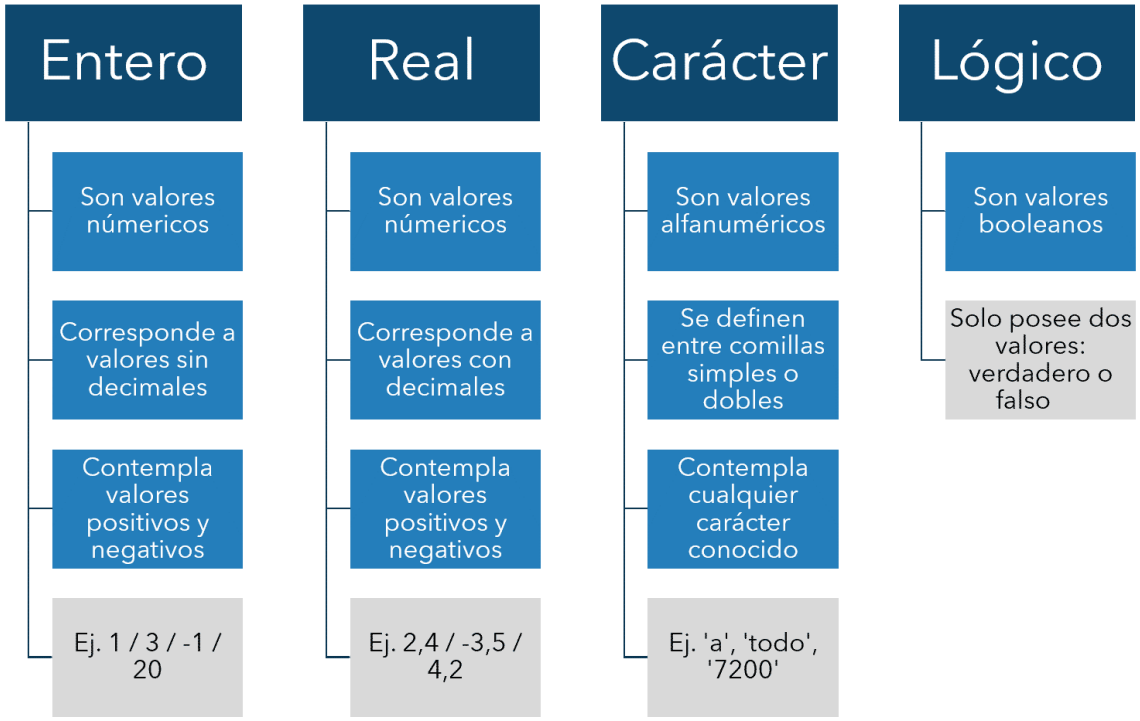
Es necesario declarar las variables que utilizaremos. Una variable será un medio para almacenar valores, los que podrán cambiar a lo largo del algoritmo.

Por ejemplo, si queremos rescatar valores entregados por un usuario, estos deben quedar almacenados en variables. Estas se definen con un nombre, que nos permitirá identificarlas a lo largo del algoritmo.

Al declarar la variable, debemos indicar el tipo de dato que almacenará. Aunque existen algunos lenguajes que no lo necesitan, la mayoría sí lo requiere y es bueno cuando se comienza a aprender.

Los tipos de datos básicos que puede tomar una variable son los siguientes:

Figura 3: Tipos de datos



Fuente: elaboración propia.

Dentro del PSeInt, las variables y sus tipos de datos se declaran al comienzo del algoritmo.

Figura 4: Declaración PSeInt

PSEINT

```
Proceso sumatoria
    //Definir Variables
    Definir num, num2 como Entero;
    Definir media como Real;
    Definir nombre como Caracter;
    Definir respuesta Como Logico;

FinProceso
```

Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Comentarios

Los comentarios son textos que nos permiten incluir notas en nuestro algoritmo o código. Estos son ignorados al ejecutar las instrucciones. En este caso, los utilizaremos para indicar las secciones.

En PSeInt, los comentarios en el código se expresan anteponiendo los caracteres *//* al texto que se va a comentar. En el caso de Python, se debe anteponer el carácter # al texto que se va a comentar. En el ejemplo anterior, el comentario nos entrega una referencia de la sección del código ***//Definir Variables***

Leer / *input*()

La instrucción ***Leer***, en PSeInt, permite recuperar datos desde el usuario. Esta instrucción se transcribe como ***input()*** en el lenguaje Python.

Figura 5: Lectura PSeInt

PSEINT

```
Proceso sumatoria
    //Definir Variables
    Definir num, num2 como Entero;
    Definir media como Real;
    Definir nombre como Caracter;
    Definir respuesta Como Logico;

    //Lectura de Variables
    Leer num1;
    Leer num2;

FinProceso
```

Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Escribir print()

La instrucción ***Escribir*** en PSeInt permite mostrar datos al usuario. Esta instrucción se transcribe como ***print()*** en el lenguaje Python.

Figura 6: Escritura PSeInt

PSEINT

```
Proceso sumatoria
    //Definir Variables
    Definir num, num2 como Entero;
    Definir media como Real;
    Definir nombre como Caracter;
    Definir respuesta Como Logico;

    //Lectura de Variables
    Leer num1;
    Leer num2;

    //Escribir Resultado
    Escribir num1 + num2;

FinProceso
```

Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Ejercicio

Se necesita un algoritmo que guarde la información de una persona. Identifica el tipo de datos de la siguiente lista de atributos y decláralo usando pseudocódigo.

1. Nombre
2. Apellido
3. Edad
4. EsCasado (verdadero o falso)

5. EsEstudiante (verdadero o falso)
6. Dirección
7. Promedio calificación

Solución

1. Definir nombre como Carácter.
2. Definir apellido como Carácter.
3. Definir edad como Entero.
4. Definir esCasado como Lógico.
5. Definir esEstudiante como Lógico.
6. Definir dirección como Carácter.
7. Definir promedio como Real.

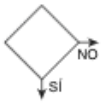
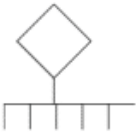
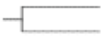
Diagrama de flujo

Un diagrama de flujo expresa, de manera gráfica, los pasos que se deben seguir y las decisiones que se van a tomar en un algoritmo o proceso específico.

Existen estándares que definen los símbolos que se deben utilizar en los diagramas de flujo. Dichos diagramas no siempre son algoritmos, pueden ser procesos de negocio, procesamiento electrónico de datos, entre otros.

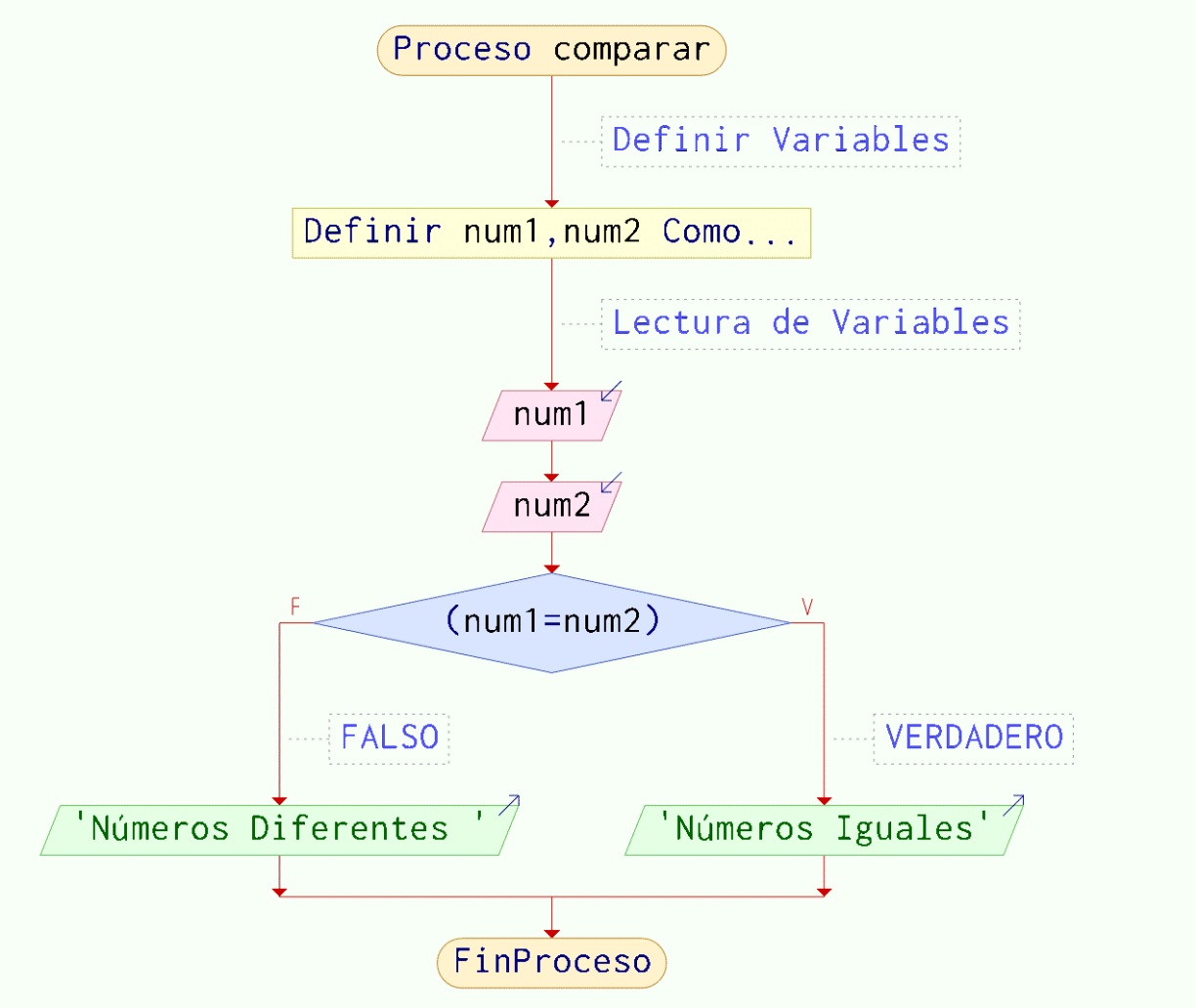
En este caso, utilizaremos los estándares ANSI (American National Standards Institute) para la diagramación administrativa, que son los utilizados para la diagramación de algoritmos.

Tabla 1: Simbología ANSI

Símbolos principales	Función
	Terminal (representa el comienzo, "inicio", y el final, "fin" de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa.
	Entrada/Salida (cualquier tipo de introducción de datos en la memoria desde los periféricos, "entrada", o registro de la información procesada en un periférico, "salida".
	Proceso (cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etc.).
	Decisión (indica operaciones lógicas o de comparación entre datos —normalmente dos— y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas —respuestas SÍ o NO— pero puede tener tres o más, según los casos).
	Decisión múltiple (en función del resultado de la comparación se seguirá uno de los diferentes caminos de acuerdo con dicho resultado).
	Conector (sirve para enlazar dos partes cualesquiera de un ordinograma a través de un conector en la salida y otro conector en la entrada. Se refiere a la conexión en la misma página del diagrama.
	Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).
	Línea conectora (sirve de unión entre dos símbolos).
	Conector (conexión entre dos puntos del organigrama situado en páginas diferentes).
	Llamada a subrutina o a un proceso predeterminado (una subrutina es un módulo independientemente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).
	Pantalla (se utiliza en ocasiones en lugar del símbolo de E/S).
	Impresora (se utiliza en ocasiones en lugar del símbolo de E/S).
	Teclado (se utiliza en ocasiones en lugar del símbolo de E/S).
	Comentarios (se utiliza para añadir comentarios clasificadores a otros símbolos del diagrama de flujo. Se pueden dibujar a cualquier lado del símbolo).

Fuente: Joyanes Aguilar, 2007, p. 72.

Figura 7: Diagrama de flujo PSeInt



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

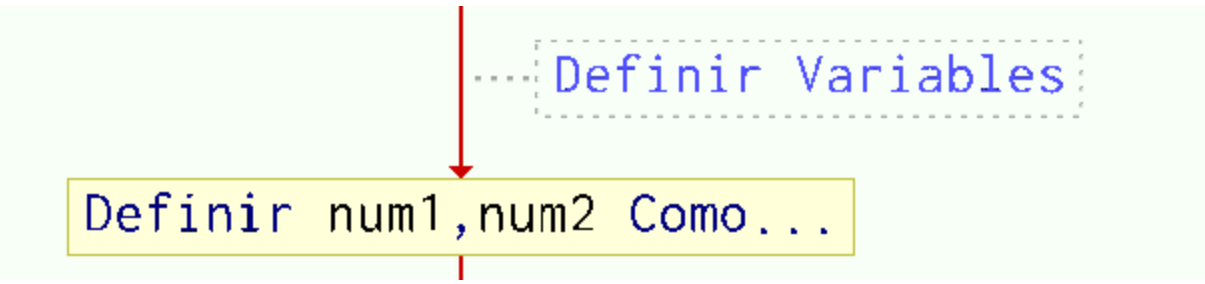
En el símbolo de ***Inicio***, PSeInt indica el nombre del algoritmo.

Figura 8: Inicio

Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Las definiciones de las variables son identificadas como parte del proceso. En la imagen, se incluye el comentario que indica la sección del algoritmo.

Figura 9: Proceso



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

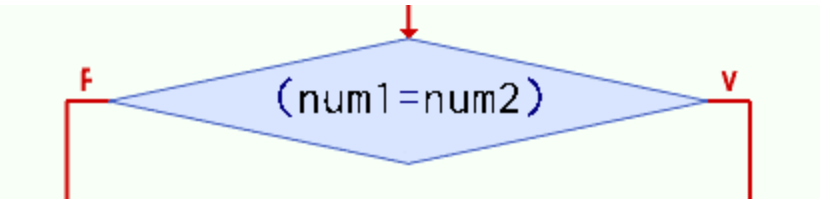
Las entradas de datos se representan, en PSeInt, con una flecha hacia el interior del símbolo, para indicar que son entradas, ya que los símbolos de entrada y salida son iguales. En la imagen, se incluye el comentario que indica la sección del algoritmo.

Figura 10: Entrada

Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Las decisiones se representan a través de un rombo, donde se definen ambos caminos, ***Verdadero*** y ***Falso***. Para los casos de decisiones iterativas, la diferencia es que, al terminar la serie de instrucciones, se retorna a la decisión en lugar de continuar secuencial.

Figura 11: Decisión

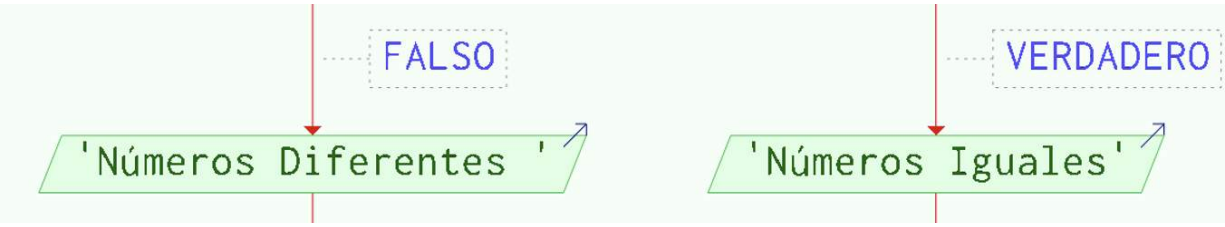


Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Las salidas de datos se representan, en PSeInt, con una flecha hacia el exterior del símbolo, para indicar que son salidas. En la imagen, se incluye

el comentario que indica la sección del algoritmo.

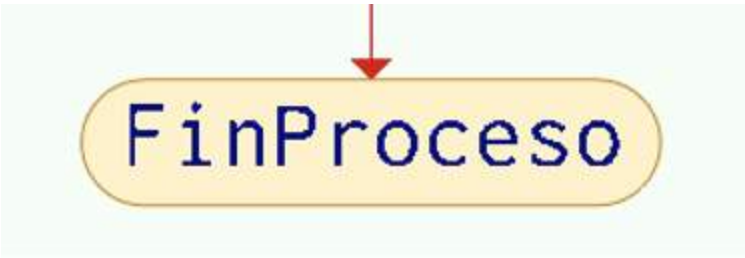
Figura 12: Salidas



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

El símbolo utilizado para el término del algoritmo es el mismo que se utiliza para el inicio, sin embargo, este indica específicamente que es el fin y desde él no salen nuevos caminos.

Figura 13: Fin



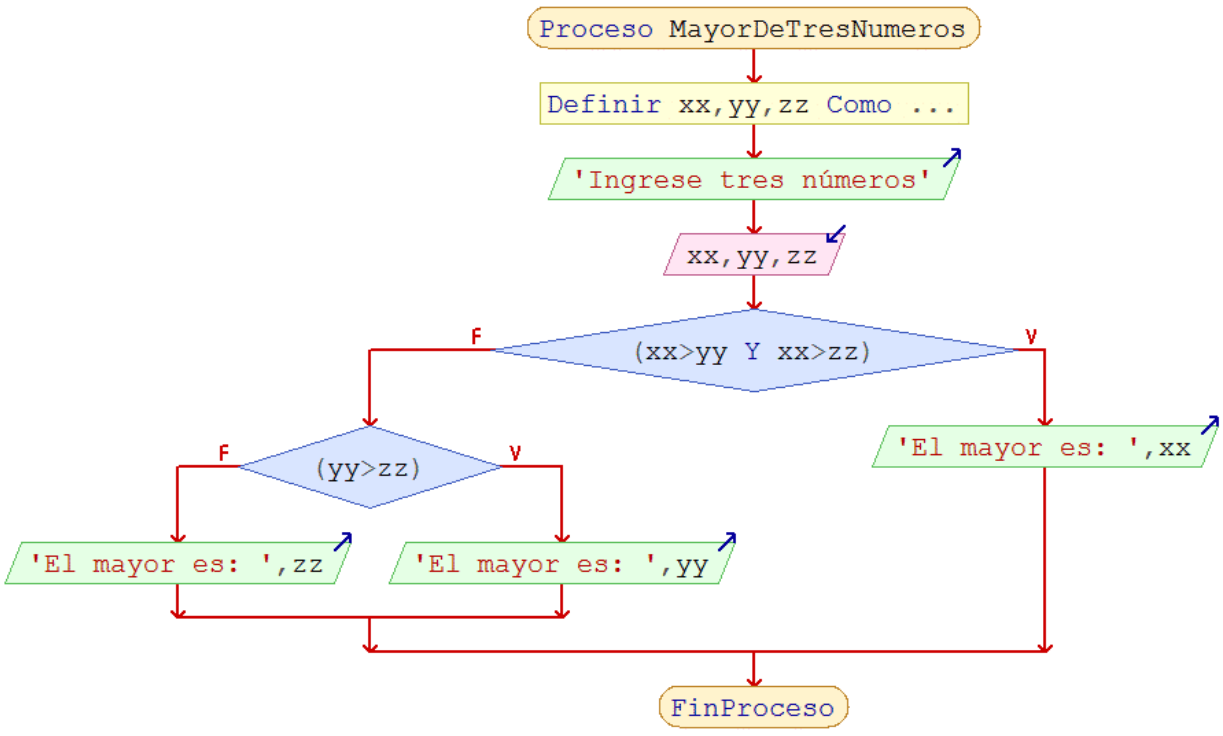
Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Los diagramas de flujo pueden ser muy simples —como los vistos ahora— o muy complejos y largos. Todo depende del problema que se esté intentando resolver.

Ejercicios

En el siguiente algoritmo, indica a qué elemento corresponde cada uno de los símbolos.

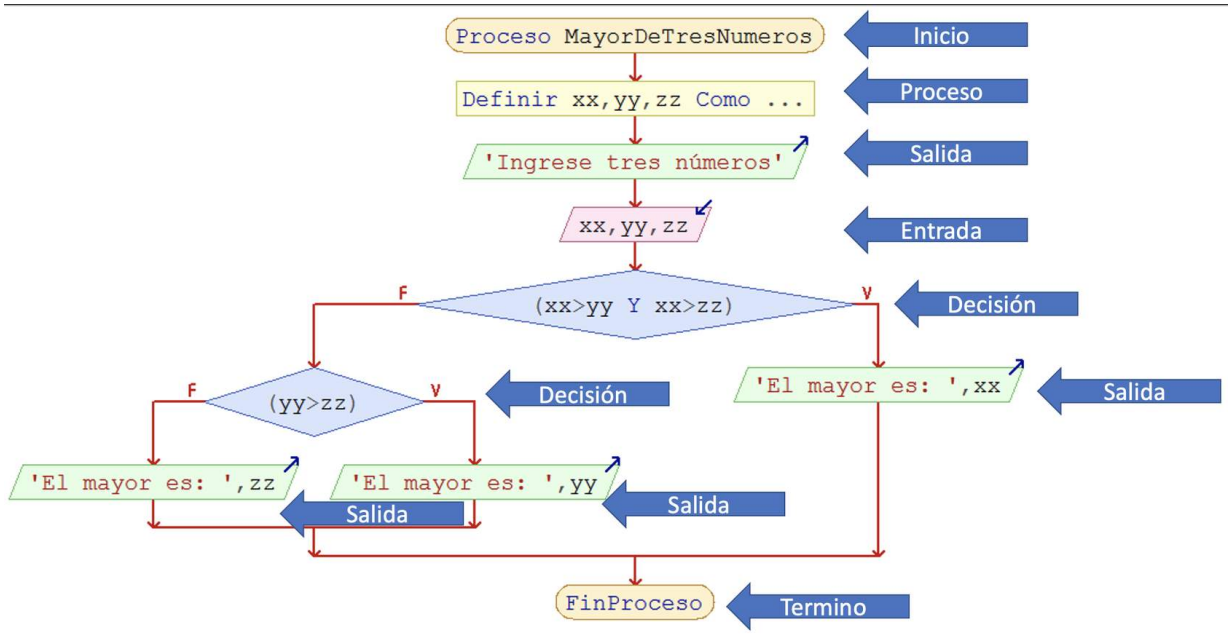
Figura 14: Ejercicio



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Resultado

Figura 15: Resolución



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Tema 4: Uso de variables y tipos de datos

Las variables son un elemento fundamental de la programación, dado que, más allá de lo que haga el sistema que estamos construyendo, siempre vamos a necesitar almacenar información en la memoria del computador. Por ese motivo a esas porciones de memoria que contienen los datos se les

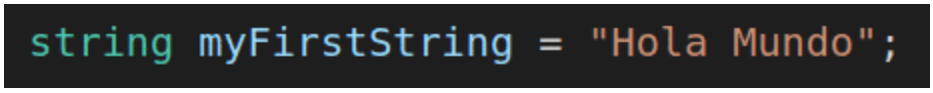
asignan nombres simbólicos (identificadores), y así conforman lo que un programador maneja como variables dentro de su código.

Casi todos los datos que se manejan en nuestro programa se almacenan en variables.

Se debe concebir "variable" como un contenedor de información.

Tomemos como ejemplo una definición de una variable de tipo cadena de caracteres (string) en C++:

Figura 16: Variable de tipo texto en C++



Fuente: elaboración propia.

Veamos la misma variable declarada en Python:

Figura 17: Variable en Python

```
myFirstVariable = "Hola Mundo"
```

Fuente: elaboración propia.

Las variables siempre tienen un tipo de dato asociado, que establece cómo el lenguaje tiene que tratar el dato contenido dentro de esa variable; por ejemplo, no es lo mismo manipular un número que una palabra.

Tipos de datos

Existen diferentes tipos de datos, los principales son los siguientes:

- ***Integers*:** este tipo de datos se utilizan cuando queremos almacenar un número sin decimales (un número entero). Por ejemplo, es lo que usamos si queremos calcular la suma de 100 + 300.
- ***Float*:** el tipo ***float*** permite la manipulación de números con decimales. El número 12,25 sería de tipo ***float***.
- ***Double*:** las variables de este tipo, al igual que las del tipo ***float***, permiten manipular números con decimales. La principal diferencia es la precisión. Si necesitamos manipular números con muchos decimales, entonces debemos utilizar este tipo de datos.
- ***Character*:** representa un único carácter, que puede ser un número, una letra o un símbolo.
- ***String*:** representa cadenas de caracteres. Es utilizado cada vez que necesitamos manipular o almacenar cadenas con letras, números y

símbolos. Un texto, por ejemplo, se debe almacenar con este tipo de dato.

- ***Boolean*:** puede tomar solamente los valores ***"true"*** (verdadero) o ***"false"*** (falso).

Figura 18: Tipos de datos en C++

```
string myFirstString = "Hola Mundo";
int myFirstInteger = 1;
float myFirstFloat = 10.25f;
double myFirstDouble = 5.5;
bool myFirstBoolean = true;
char myFirstCharacter = '@';
```

Fuente: elaboración propia.

Según el tipo de dato, las variables serán manejadas y procesadas dentro del programa de diferentes modos. Con variables numéricas, se podrán realizar, por ejemplo, conteos y operaciones matemáticas; sobre ***strings***, se podrán realizar concatenaciones o divisiones en cadenas más pequeñas, etcétera.

Constantes

Algunos datos necesarios tendrán información almacenada que no cambiará a lo largo del programa. En estos casos, es conveniente declarar ese dato como una constante en lugar de una variable.

Una constante es un valor que no puede ser alterado o modificado durante la ejecución de un programa, únicamente puede ser leído.

Figura 19: Constantes en C++

```
const double PI = 3.14159265359;
const float FUERZA_GRAVEDAD = 9.8f;
const int DIAS_DE_SEMANA = 7;
const char ARROBA = '@';
const string PRIMER_MES = "Enero";
```

Fuente: elaboración propia.

CONTINUAR

Unidad 2. Estructuras de control

A medida que los algoritmos se vuelven más complejos, es necesario incorporar otro tipo de estructuras que nos permitan generar bifurcaciones en los pasos que se van a describir, por ejemplo, en caso de tener un algoritmo para un supervisor y otro para un administrativo —y estos son iguales, a excepción de los valores que se van a utilizar— podemos tener el mismo algoritmo, utilizando alguna estructura de control para determinar el valor que se va a utilizar.

Estas estructuras nos permiten controlar el flujo de ejecución. En otras palabras, necesitamos ser capaces de controlar las acciones del algoritmo que estamos creando.

Supongamos que tenemos un algoritmo que permite cambiar el color de la luz de una vitrina mediante control remoto. El evento de entrada sería presionar el botón del control remoto, el proceso sería el cambio de color de la luz y el resultado sería el nuevo color mostrado en la vitrina.

Ahora bien, ¿cómo sabe el algoritmo que tiene que cambiar el color cuando se presiona el botón y no en otro lado? ¿Cómo sabe que tiene que cambiar el color de la vitrina y no de otra cosa? Todo este tipo de decisiones tienen

que estar plasmadas en el algoritmo y no dejar nada librado al azar. Esto se logra mediante las estructuras de control que veremos a continuación.

Los ejemplos, en este módulo, son codificados en PSeInt y Python, para que puedas tener dos puntos de vista diferentes, pero las estructuras de control son comunes en todos los lenguajes de programación. A PSeInt lo puedes encontrar en http://PseInt.sourceforge.net/ y te permite trabajar en lenguaje natural, posee ciertas reglas, pero la aplicación te va guiando para que el algoritmo se pueda ejecutar. Python es un lenguaje de programación, para utilizarlo es necesario realizar un número mayor de instalaciones, pero puedes encontrar su sintaxis en la documentación oficial, en el siguiente ***link***: https://docs.python.org/es/3/

A continuación, revisaremos las diferentes estructuras de control que podemos utilizar en el diseño de un algoritmo.

Tema 1. Condicionales

Los condicionales son instrucciones que evalúan la veracidad de una sentencia, por lo tanto, utilizan valores booleanos (verdadero o falso) para decidir si ejecutan o no una determinada sección de código. Desde ahora, veremos ejemplos en PSeInt y en Python, cuando el lenguaje lo permita.

La sentencia *SI* / *IF*

SI* o *IF establece la evaluación de una variable o expresión cuyo valor o resultado es de tipo booleano, es decir, verdadero o falso. En el caso de que sea verdadero, ejecuta una sección de código. Si es falso, no la ejecuta y pasa a la siguiente sección.

Cabe destacar que en las condiciones que se van a evaluar en la instrucción ***SI* o *IF***, es donde utilizaremos la lógica proposicional.

Figura 20: Estructura de control Si PSeInt

```
Proceso Condicional
    //Definir variables
    Definir num1, num2 como Entero;

    //Lectura de variables
    Leer num1;
    Leer num2;

    //Estructura de Control SI
    Si (num1 = num2) Entonces
        //Verdadero
        Escribir "Números Iguales";

    FinSi

FinProceso
```

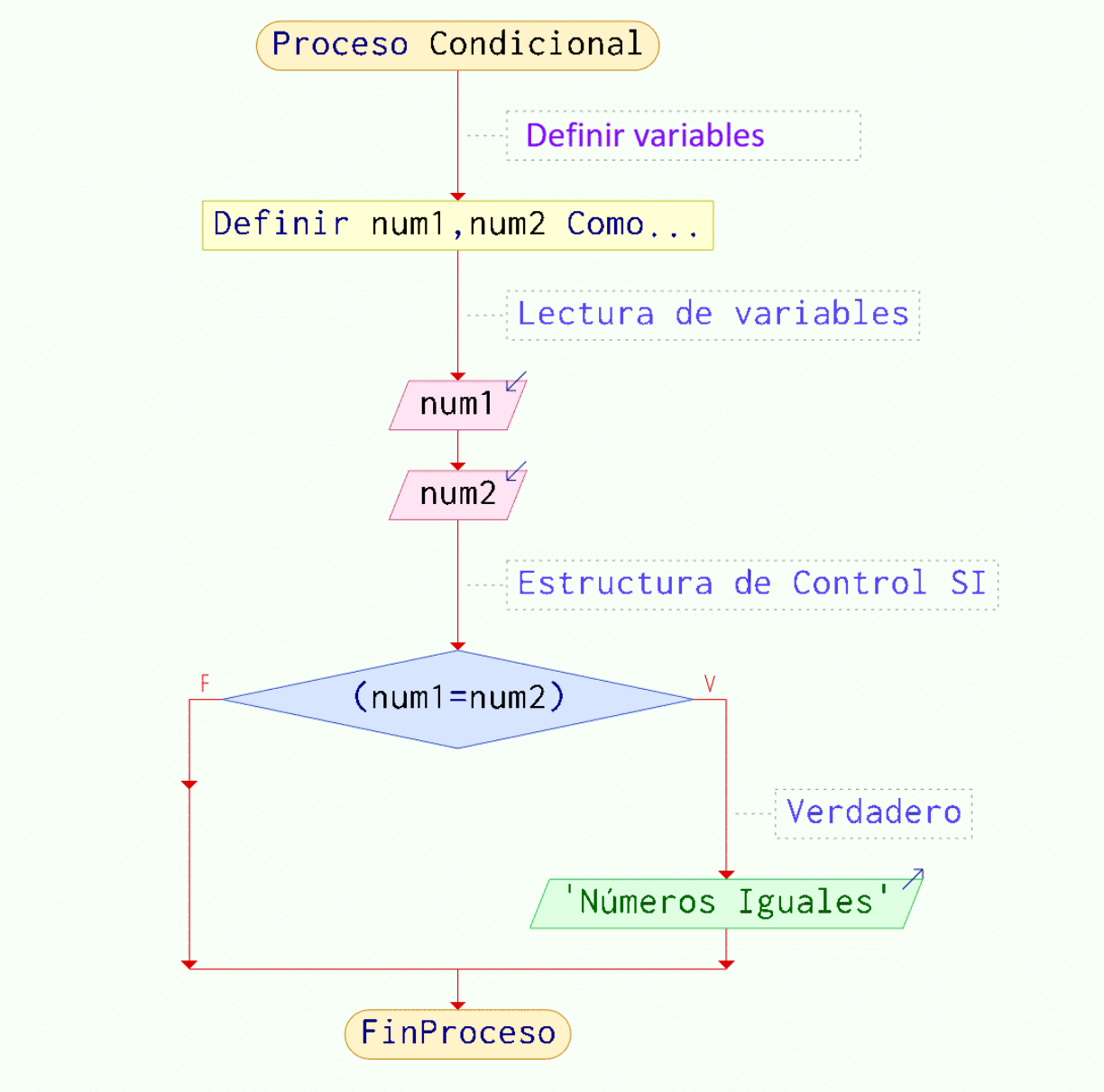
Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Figura 21: Estructura de control *If Python*

```
#Definir Variables
num1 = 0
num2 = 0

#Lectura de Variables
num1 = input()
num2 = input()

#Estructura de control IF
if num1 == num2:
    print("Números Iguales")
```

Fuente: elaboración propia a base del ***software Python***.

Figura 22: Estructura de control Si - Diagrama de flujo



Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

Esta sentencia corresponde a una estructura de control que permite condicionar las instrucciones contenidas al resultado de la expresión evaluada. En el ejemplo, se evalúa que ambas variables leídas, ***num1*** y

num2, sean iguales. En caso de que esta validación sea verdadera, entonces, se escribe el mensaje.

En la comparación de Python, se utiliza un doble signo igual, esto se produce porque, en este lenguaje y en muchos otros, un signo igual significa asignación. Por lo tanto, para diferenciarlo, la sintaxis indica que se deben usar dos cuando se desea comparar.

Las instrucciones contenidas dentro de la estructura **Si / *If*** en PSeInt se delimitan con la palabra ***FinSi***. En el caso de Python, se considera la indentación.

La sentencia *Sino* / *Else*

Existe la posibilidad de indicar también una sección de código, en el caso de que la condición validada sea falsa, esto se incorpora después de la sección de código para la opción verdadera. Queda de la siguiente manera:

Figura 23: Estructura de control SINO PSeInt

```
Proceso Condicional
    //Definir variables
    Definir num1, num2 como Entero;

    //Lectura de variables
    Leer num1;
    Leer num2;

    //Estructura de Control SI-SINO
    Si (num1 = num2) Entonces
        //Verdadero
        Escribir "Números Iguales";

    SiNo
        //Falso
        Escribir "Números Diferentes";
    FinSi

FinProceso
```

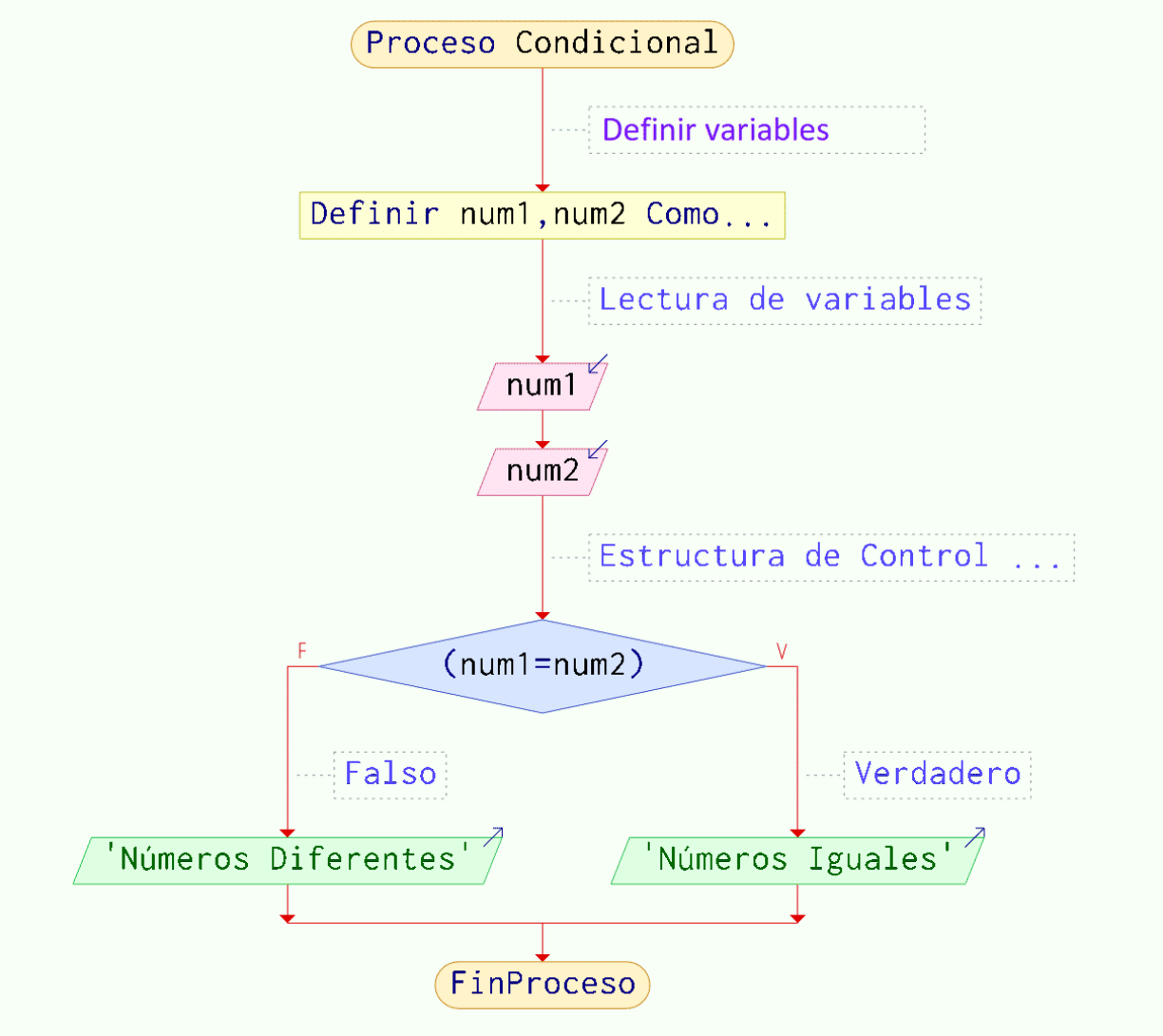
Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

Figura 24: Estructura de control *ELSE Python*

```
#Definir Variables
num1 = 0
num2 = 0

#Lectura de Variables
num1 = input()
num2 = input()

#Estructura de control IF
if num1 == num2:
    print("Números Iguales")
#Estructura de control ELSE
else:
    print("Números Diferentes")
```

Fuente: elaboración propia a base del ***software Python***.

Figura 25: Estructura de control Sino - Diagrama de flujo



Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

En ambos casos, se han incluido líneas de código después de la sección destinada a una validación verdadera. ***Sino / Else*** indica que se ejecutará esta sección en caso de que la condición anterior no se cumpla.

Con esta estructura, tendremos un camino para desarrollar en caso de que la condición sea verdadera o en caso de que sea falsa.

Para nuestro ejemplo, la instrucción que se va a ejecutar es el mensaje ***"Números Diferentes"***, en caso de que la comparación ***num1*** = ***num2*** sea falsa.

Es importante destacar que la sentencia ***Sino / Else*** no existe por sí sola, siempre será un anexo para incluir a la instrucción ***Si / IF***. Además, solo una de las secciones de código se ejecutará.

La sentencia Sino-Si/ *Else-If*

Hasta ahora, las sentencias ***SI / IF*** y ***Sino / Else*** pueden ser de gran utilidad, pero ¿qué sucede si necesitamos evaluar más de dos condiciones? Por ejemplo, debemos realizar un algoritmo que identifique qué número es mayor que otro, además de indicar si son iguales.

Esto es posible mediante la instrucción ***Sino-Si / Else-If***.

A continuación, modificaremos el código anterior incluyendo esta nueva validación ***num1 > num2***. Se enviará el mensaje que el número 1 es mayor; si no, el número 2 es mayor.

La sintaxis en PSeInt, en Python o en otros lenguajes puede variar, pero la esencia de la estructura es la misma.

Figura 26: Estructura de control *Sino-Si PSeInt*

```
Proceso Condicional
    //Definir variables
    Definir num1, num2 como Entero;

    //Lectura de variables
    Leer num1;
    Leer num2;

    //Estructura de Control SI-SINO SI-SINO
    Si (num1 = num2) Entonces
        //Primera Condición Verdadera
        Escribir "Números Iguales";
    Sino Si (num1 >num2) Entonces
        //Segunda Condición Verdadera
        Escribir "Número 1 Mayor";
    SiNo
        //Falso
        Escribir "Número 2 Mayor";
    FinSi
    FinSi

FinProceso
```

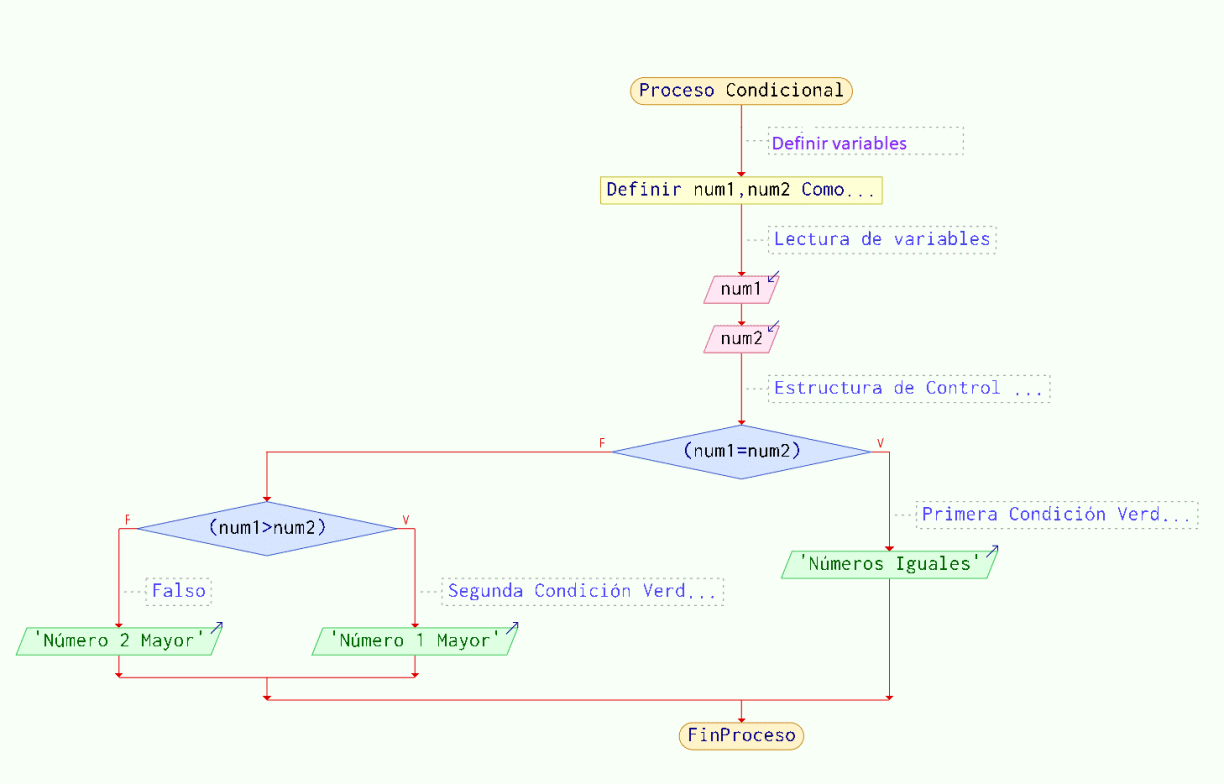
Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

Figura 27: Estructura de control *Else-if Python*

```
#Definir Variables
num1 = 0
num2 = 0

#Lectura de Variables
num1 = input()
num2 = input()

#Estructura de control IF
if num1 == num2:
    print("Números Iguales")
#Estructura de control ELSE IF
else if num1 > num2:
    print("Número 1 es mayor")
#Estructura de control ELSE
else:
    print("Número 2 es mayor")
```

Fuente: elaboración propia a base del ***software Python***.

Figura 28: Estructura de control Sino-Si - Diagrama de flujo



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

En ambos casos, tenemos 2 comparaciones. La primera nos permite identificar los casos en los que los números son iguales; y la segunda, cuando comparamos si la variable ***num1*** es mayor a ***num2***. Si no, por defecto, indicamos que ***num2*** es mayor que ***num1***.

Como indicamos en un comienzo, las sintaxis son diferentes, pero las estructuras reflejan las mismas decisiones.

Solo uno de los caminos se ejecutará y, al igual que la sentencia ***Sino / Else***, esta nueva instrucción ***Sino-Si / Else-if*** existe solo si tenemos una sentencia ***Si / IF*** y siempre se incluye antes de la instrucción ***Sino / Else***.

La sentencia *Segun* / *switch*

Con ***Segun / switch*** podemos evaluar el valor de una variable y, de acuerdo con este, ejecutar cierta sección del código. Es similar a ***else if***, ya que es útil cuando queremos comparar el valor de una variable con más de una opción. ***Switch*** y ***else-if*** cumplen la misma función.

Siempre es recomendado utilizar ***switch***, en vez de ***else-if,*** debido a su simplicidad. Aunque Python no contiene una instrucción ***switch***, por lo que solo la veremos en PSeInt.

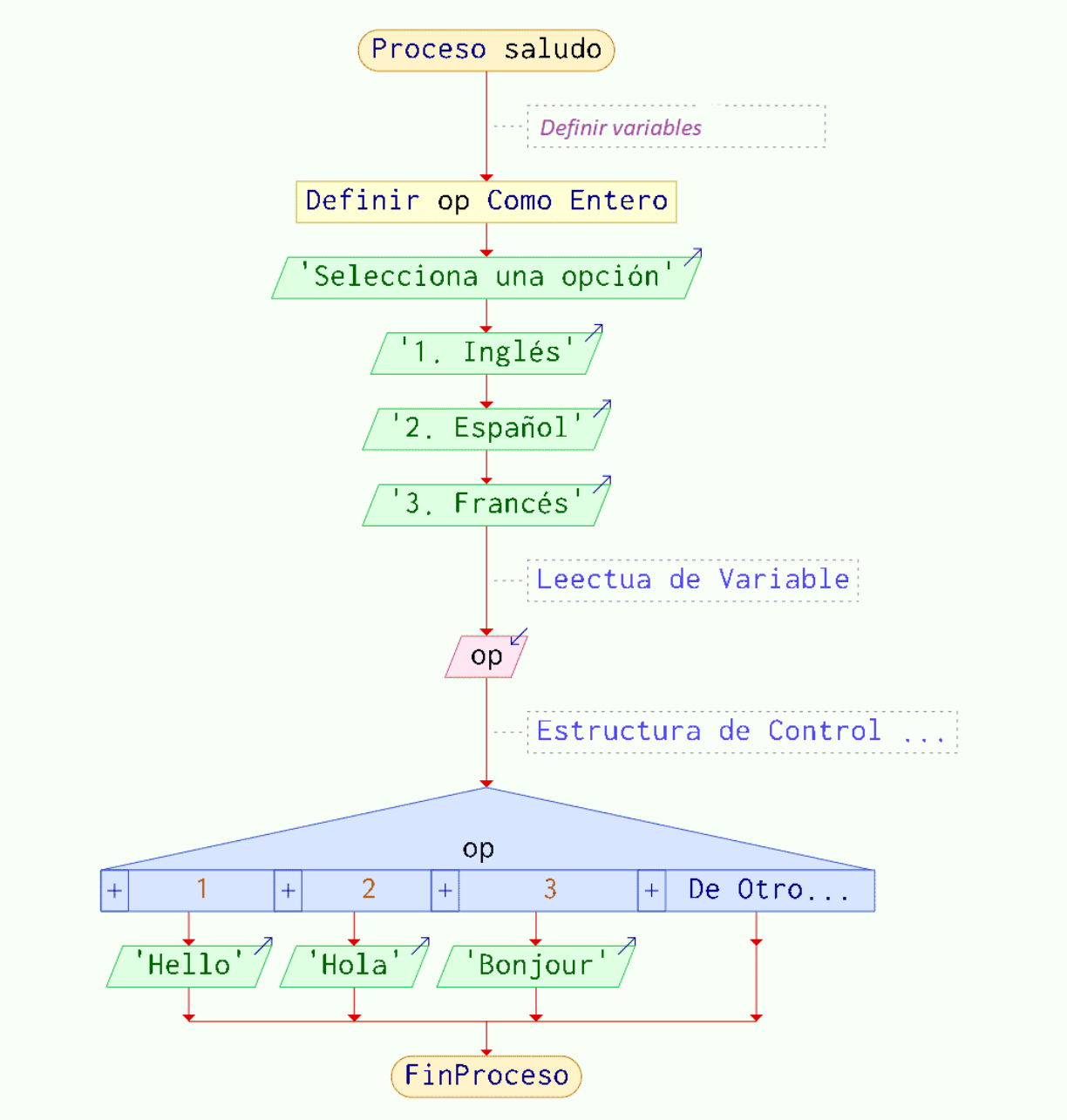
Por ejemplo, si queremos mostrar el autor de un libro específico, podemos utilizar la sentencia ***Segun / switch***.

Figura 29: Estructura de control Según PSeInt

```
Proceso saludo

    //Definir variables
    Definir op como Entero;

    Escribir "Selecciona una opción";
    Escribir "1. Inglés";
    Escribir "2. Español";
    Escribir "3. Francés";

    // Lectura de Variable
    Leer op;

    //Estructura de Control Segun
    Segun op Hacer
        1:
            Escribir "Hello";
        2:
            Escribir "Hola";
        3:
            Escribir "Bonjour";
    FinSegun
FinProceso
```

Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

Figura 30 Estructura de control Según - Diagrama de flujo



Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

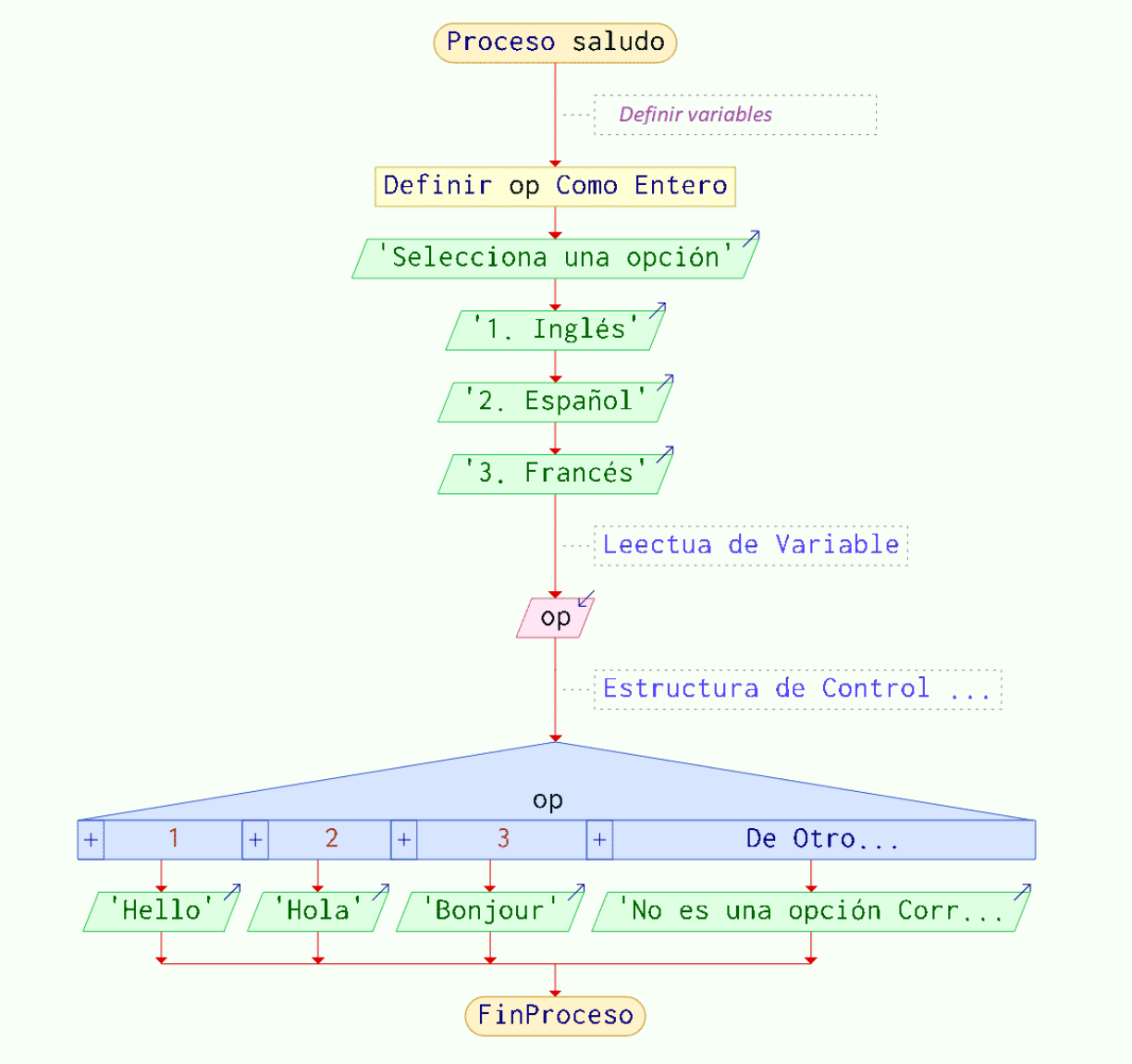
Segun /switch permite agregar una opción de **De otro *modo / default***, es decir, una opción que se ejecute cuando ninguna de las demás es verdadera (también puede ser nombrada opción por defecto o default). Esto permite responder, en caso de que el valor de la variable no corresponda con ninguna de las cuatro provincias. Por ejemplo, podríamos imprimir un mensaje que muestre ***No es una opción correcta.***

Figura 31: Estructura de control De otro modo PSeInt

```
Proceso saludo

    //Definir variables
    Definir op como Entero;

    Escribir "Selecciona una opción";
    Escribir "1. Inglés";
    Escribir "2. Español";
    Escribir "3. Francés";

    // Lectura de Variable
    Leer op;

    //Estructura de Control Segun
    Segun op Hacer
        1:
            Escribir "Hello";
        2:
            Escribir "Hola";
        3:
            Escribir "Bonjour";
        De Otro Modo:
            Escribir "No es una opción Correcta";
    FinSegun
FinProceso
```

Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

Figura 32: Estructura de control De otro modo - Diagrama de flujo



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

La cláusula ***De otro modo / default*** siempre va como opción final del ***Segun / switch***.

Tema 2. Ciclos o bucles

Un ciclo o bucle es una sección de código en la cual la ejecución se repite de acuerdo con una condición. Es similar al ***SI*** **o** ***IF***, pero con la capacidad de ejecutar varias veces la misma sección de código. Son muy útiles cuando queremos recorrer una lista de elementos o ejecutar una sección de código indefinidamente.

El bucle *Para* / *for*

Este bucle permite la repetición de una sección de código de acuerdo con el valor de una variable o expresión booleana. En el caso de que sea verdadera, el código se repite. En caso de que sea falsa, el código no se repite y pasa a la línea siguiente.

Se utiliza, comúnmente, para controlar la cantidad de veces que se necesita repetir el fragmento de instrucciones contenidas en el ciclo.

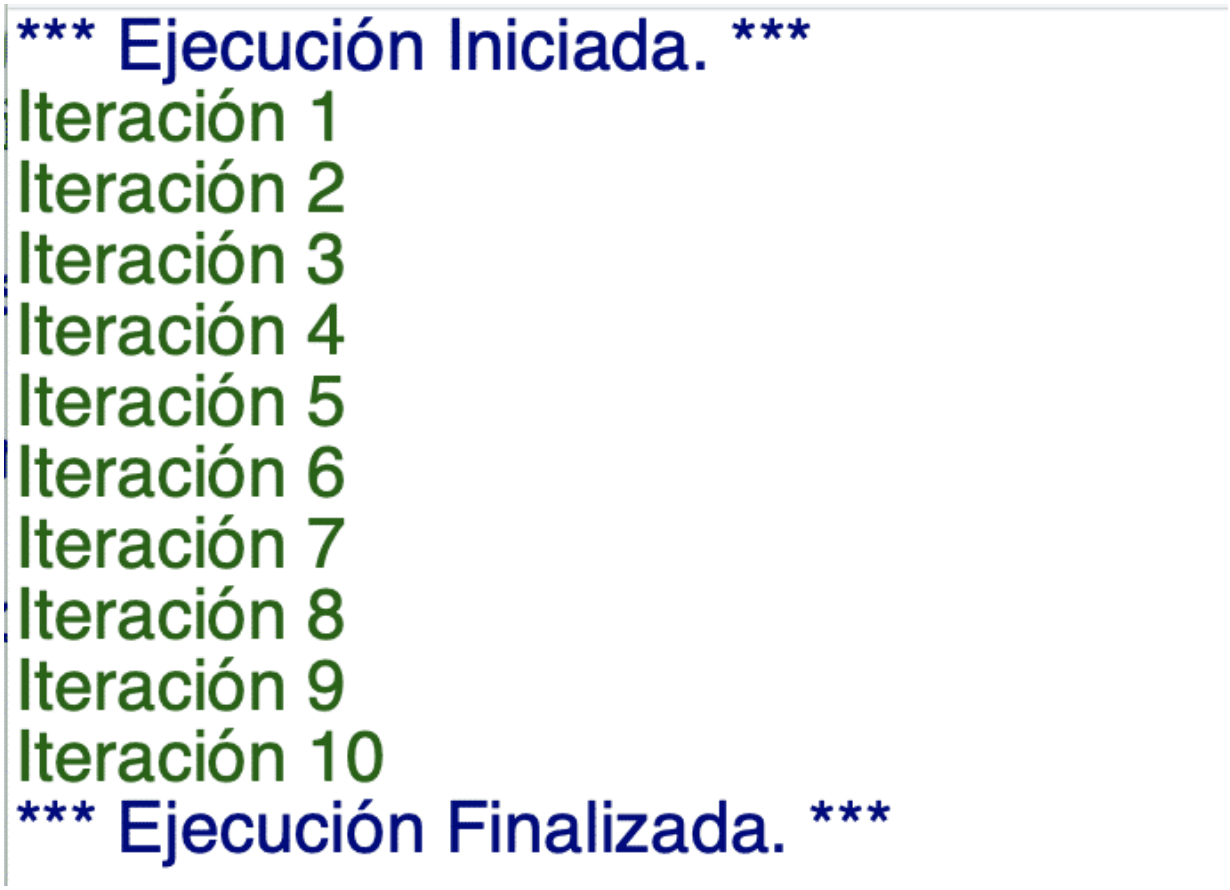
Por ejemplo, queremos mostrar los números del 1 al 10.

Figura 33: Estructura de control Para PSeInt

```
Proceso ciclo_para

    //Definir Variables
    Definir i como Entero;

    //Estructura de control Para
    Para i←i Hasta 10 Hacer

        Escribir "Iteración ", i;

    FinPara

FinProceso
```

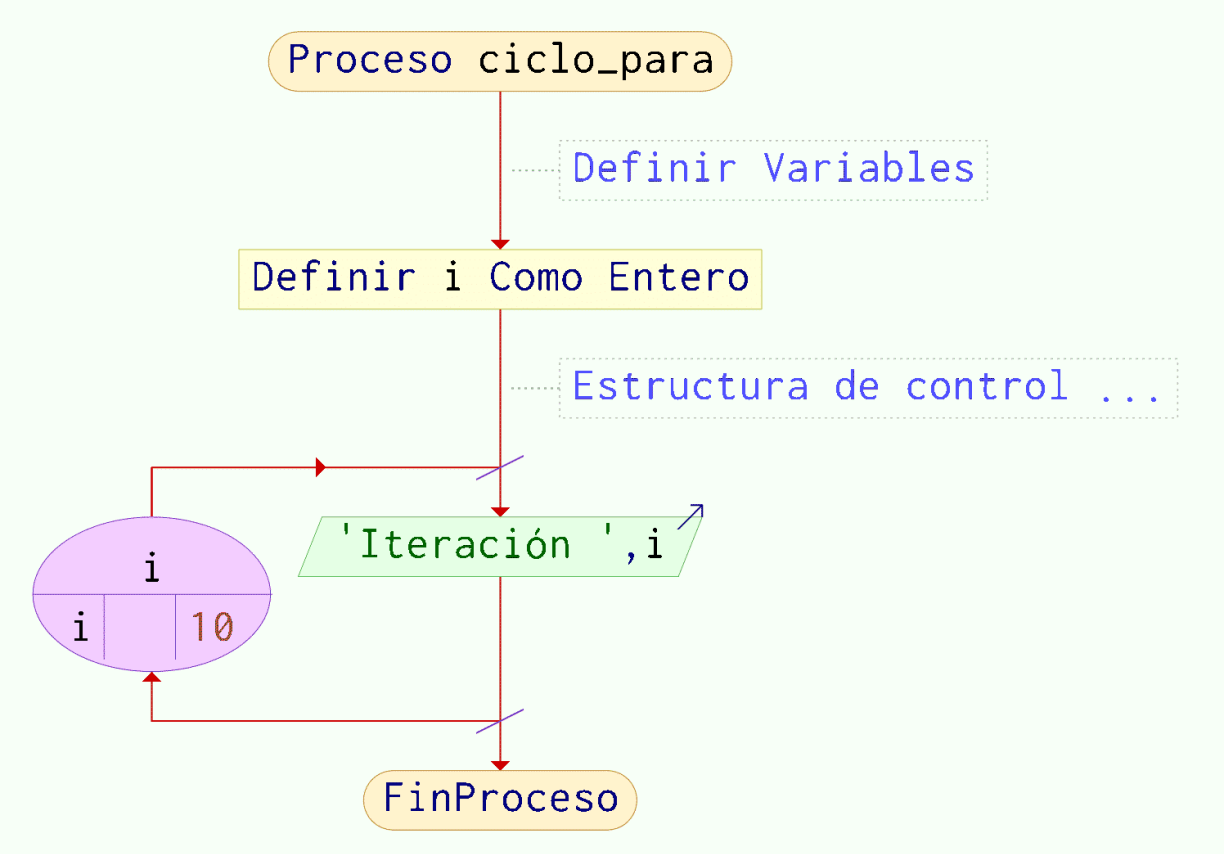
Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Figura 34: Resultado estructura de control Para PSeInt



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Figura 35: Estructura de control Para - Diagrama de flujo



Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

En el caso de PSeInt, la variable i parte en el número 1, porque le especificamos que así sea.

Sin embargo, en el caso de Python, la variable ***i*** parte en 0, por defecto.

Se ejecutan secuencialmente la cantidad de iteraciones indicadas en la sentencia, en este caso, corresponde a 10 iteraciones.

Figura 36: Estructura de control Para PSeInt

Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Figura 37: Estructura de control Para Python

Fuente: elaboración propia a base del software Python.

El bucle *Mientras / while*

Mientras / While es otro tipo de bucle que, al igual que ***Para / For***, permite la repetición de ejecución en un bloque de código determinado. Una de las diferencias de ***Mientras / While*** es que no se puede definir una variable ni cambiar su valor en la misma línea, como en el caso de ***Para / For.*** Sino que se ejecuta mientras la condición sea verdadera.

Este bucle o ciclo se utiliza cuando existe un nivel de incertidumbre en la condición, pues no siempre sabemos la cantidad de iteraciones que se ejecutarán.

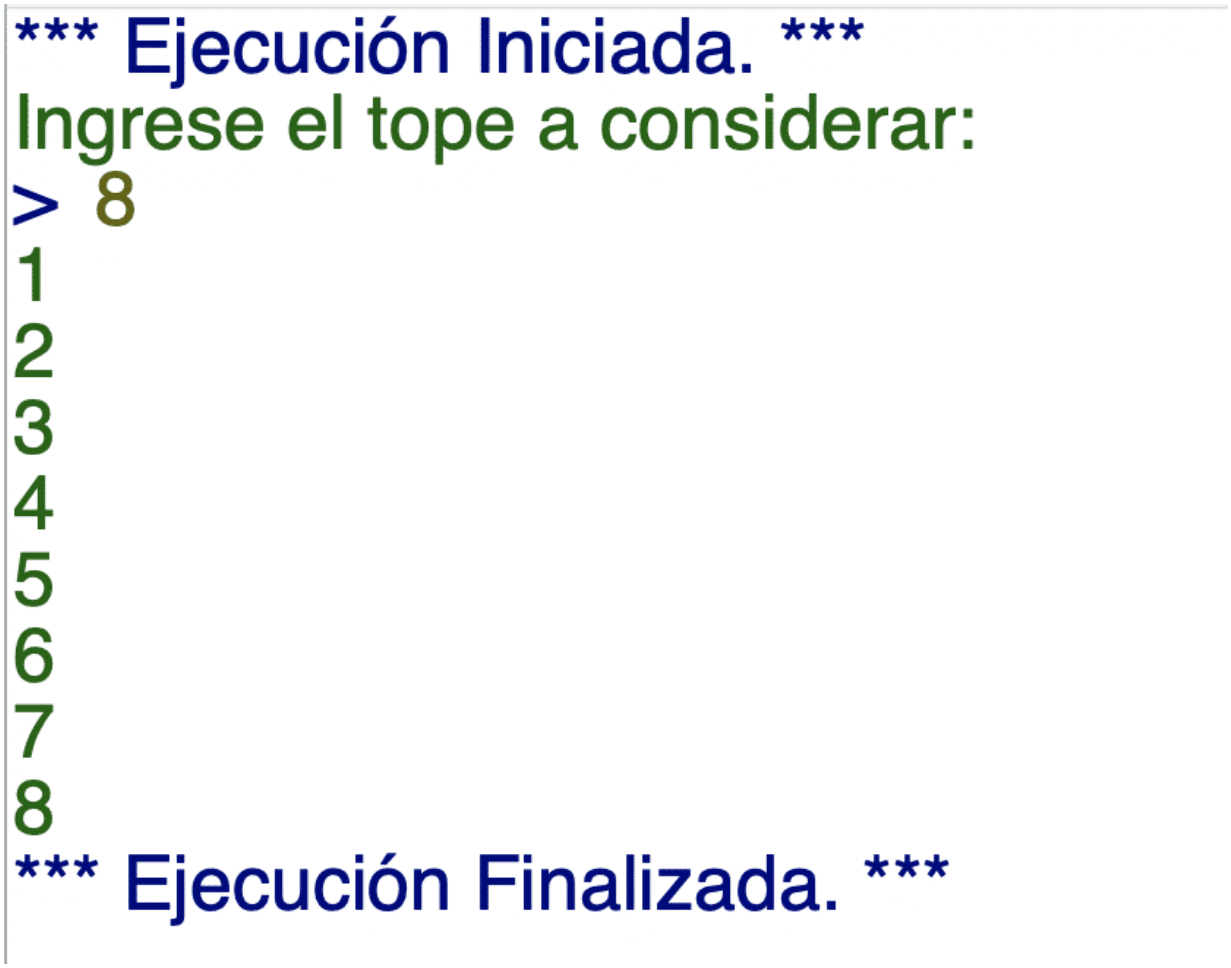
Por ejemplo, realizaremos un algoritmo que nos permita sumar 1 unidades a un número que comienza en 1, mientras el resultado no supere el tope solicitado por el usuario.

Figura 38: Estructura de control *Mientras PSeInt*

```
Proceso ciclo_mientras

    //Definir Variables
    Definir num, tope como Entero;

    //Asignar num
    num←1;

    //Solicitar el tope
    Escribir "Ingrese el tope a considerar: ";
    Leer tope;

    //Estructura de control Mientras
    Mientras num ≤ tope Hacer
        Escribir num;
        num ←num+1;

    FinMientras

FinProceso
```

Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

Figura 39: Resultado estructura de control Mientras PSeInt



Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

Figura 40: Estructura de control Mientras Python

```python
#Definir Variables
num = 1
tope = input("Ingrese el tope a considerar: ")

#Estructura de Control Mientras
while num <= int(tope):
    print(num)
    num = num + 1
```

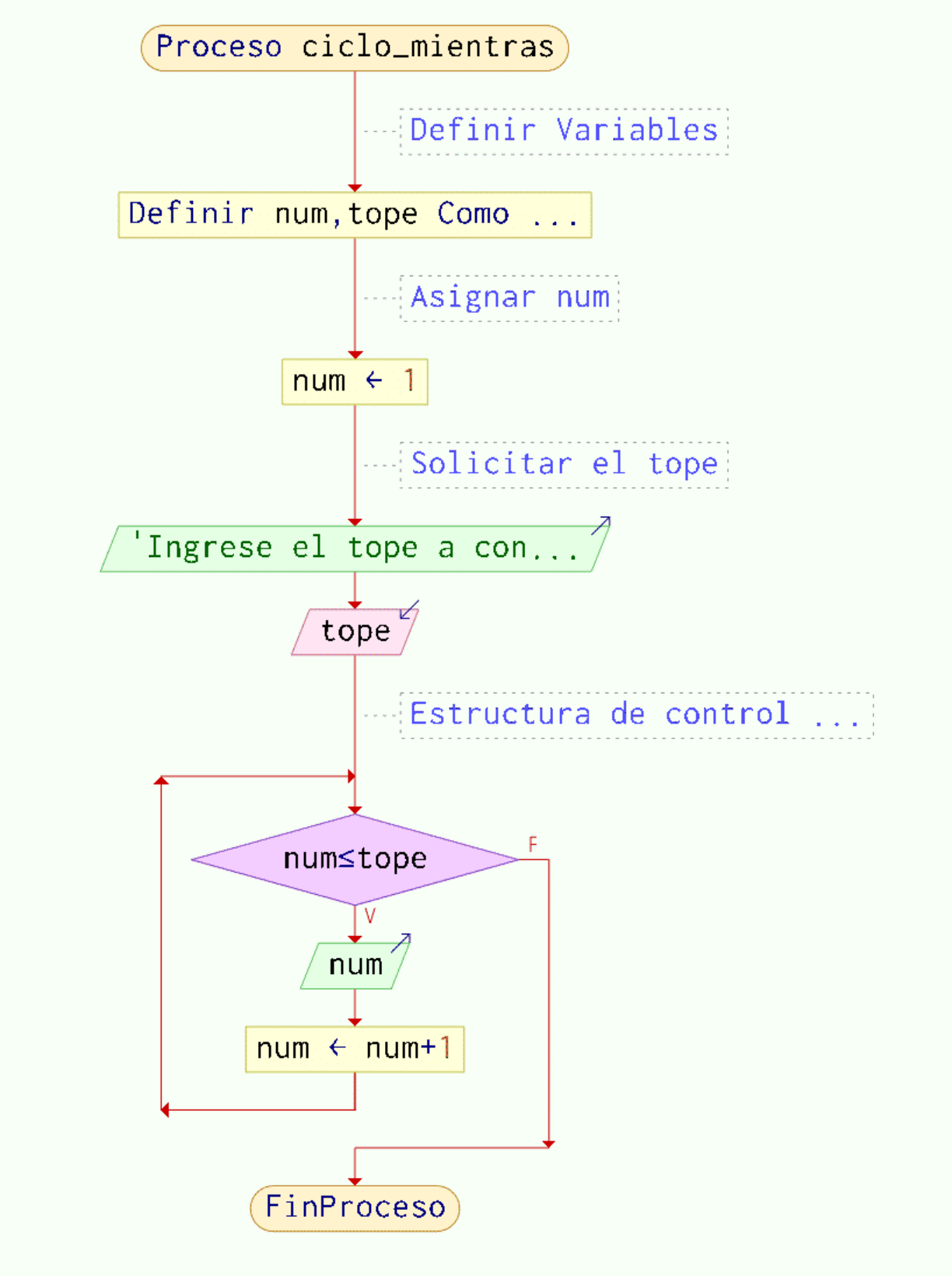
Fuente: elaboración propia a base del software Python.

Figura 41: Resultado estructura de control *Mientras Python*

```
Ingrese el tope a considerar: 8
1
2
3
4
5
6
7
8
>>>
```

Fuente: elaboración propia a base del ***software Python***.

Figura 42: Estructura de control Mientras - Diagrama de flujo



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

El bucle Repetir / *Do-while*

Este bucle cumple las mismas funciones que los ciclos anteriores (permite la repetición de la ejecución en una sección de código), pero con la diferencia de que asegura que el código se repita, al menos, una vez. Lo que hace es ejecutar el código que se va a repetir y luego evaluar si lo tiene que volver a ejecutar. Los ciclos anteriores siempre evalúan. Primero, tienen que ejecutar el código (puede no ejecutarse nada, si la condición de repetición es falsa), mientras que **Repetir / *Do-while*** lo ejecuta primero una vez y luego evalúa la condición de repetición.

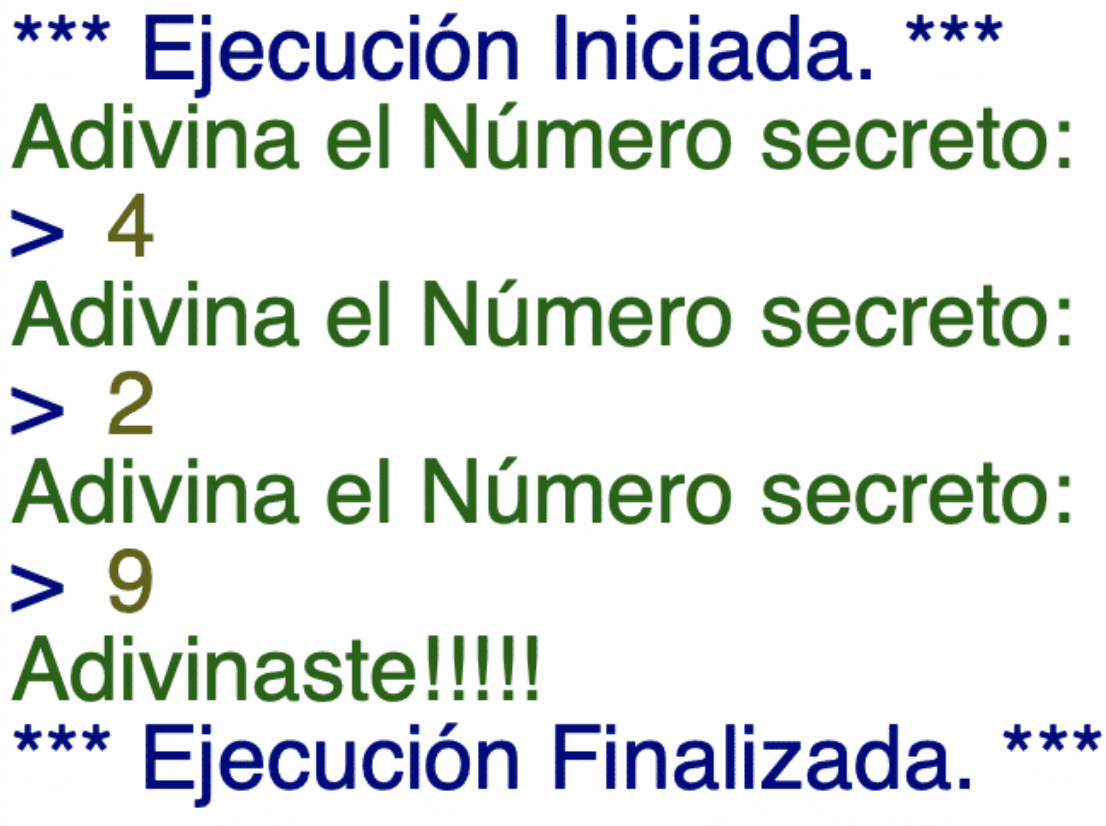
En el ejemplo, jugaremos al número secreto, es decir, tendremos un número secreto que el usuario debe adivinar, le preguntaremos al usuario un número hasta que este sea igual al número secreto.

Figura 43: Estructura de control Repetir PSeInt

```
Proceso ciclo_repetir

    //Definir Variables
    Definir num_secreto, num_usuario como Entero;

    //Definir valor número secreto
    num_secreto←9;

    //Estructura de control Repetir
    Repetir
        Escribir "Adivina el Número secreto";
        Leer num_usuario;
    Hasta Que num_secreto = num_usuario;

    Escribir "Adivinaste!!!";

FinProceso
```

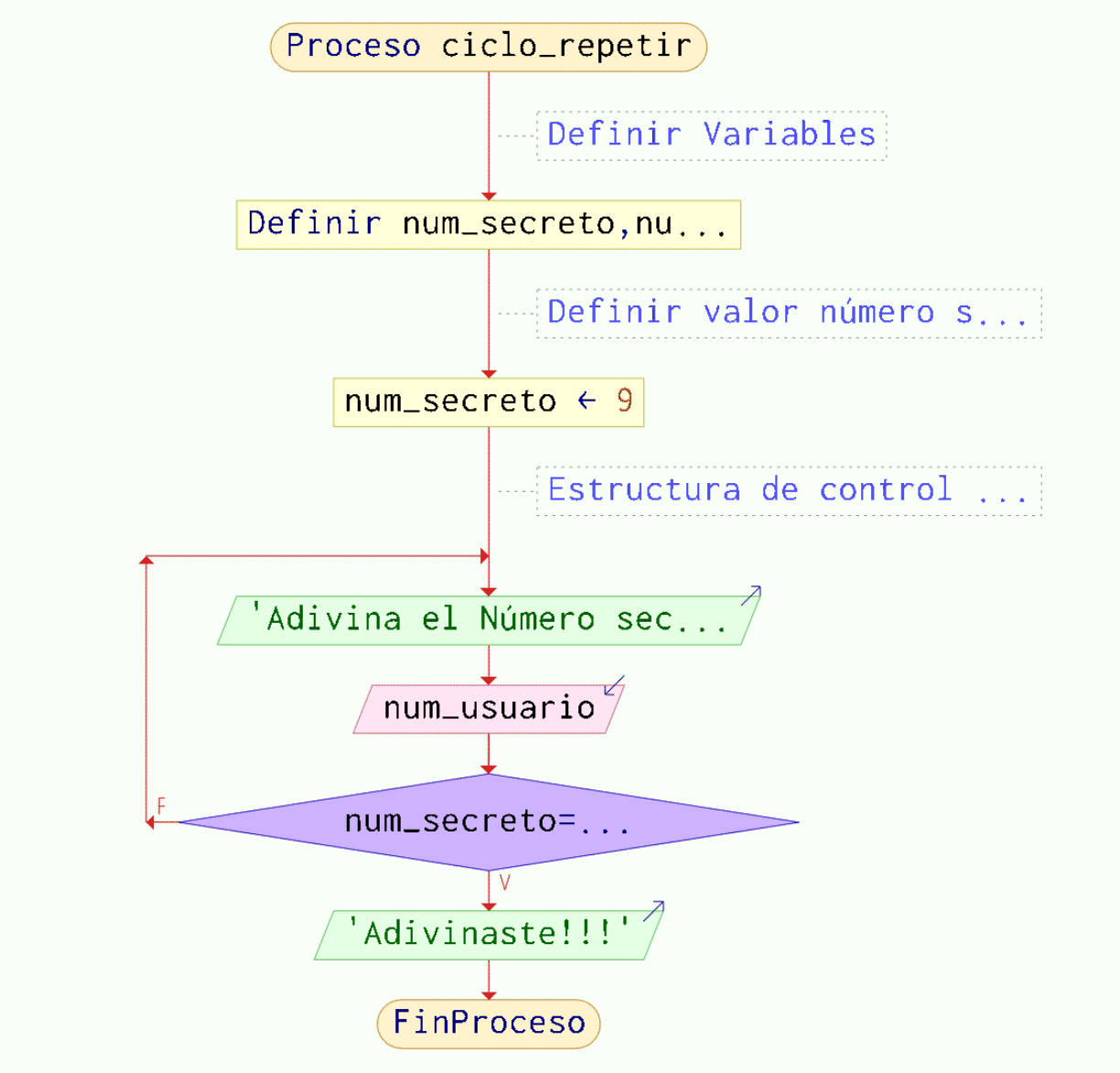
Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Figura 44: Resultado estructura de control Repetir PSeInt



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Figura 45: Estructura de control Repetir - Diagrama de flujo



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

En el caso de este ciclo, Python no tiene una sintaxis para él. Por lo que, en este lenguaje, se utilizan otras técnicas. Lo que no sucede en otros lenguajes como Java, que sí tiene la estructura de control ***Repetir / Do-while***.

El bucle *For each*

Este ciclo es específico para recorrer estructuras de manejo de datos, que revisaremos en el siguiente módulo.

Tema 3. Funciones y funciones anónimas

Hay muchas formas de codificar un sistema y algo particular sobre el desarrollo de software es que no existe una única forma de construir ese sistema. Ahora bien, sí hay lineamientos y guías para que ese desarrollo pueda ser reutilizado (ahorrando costos, ya que no tenemos que gastar tiempo haciendo las cosas de nuevo) y extensible (la capacidad de agregarle funcionalidad de manera fácil).

Uno de esos lineamientos es crear funciones. Las funciones son secciones de código que podemos reutilizar, de forma tal que no tengamos que codificar varias veces el mismo código y lo podamos reutilizar.

Por ejemplo, podemos tener una función que nos permita mostrar un menú por pantalla.

Figura 46: Función PSeInt

Fuente: elaboración propia a base del ***software*** PSeInt (Novara, 2021).

Para poder utilizar este fragmento de código o de algoritmo, debemos hacer una llamada a dicha función. Esto se realiza solo indicando el nombre de la función seguido de paréntesis.

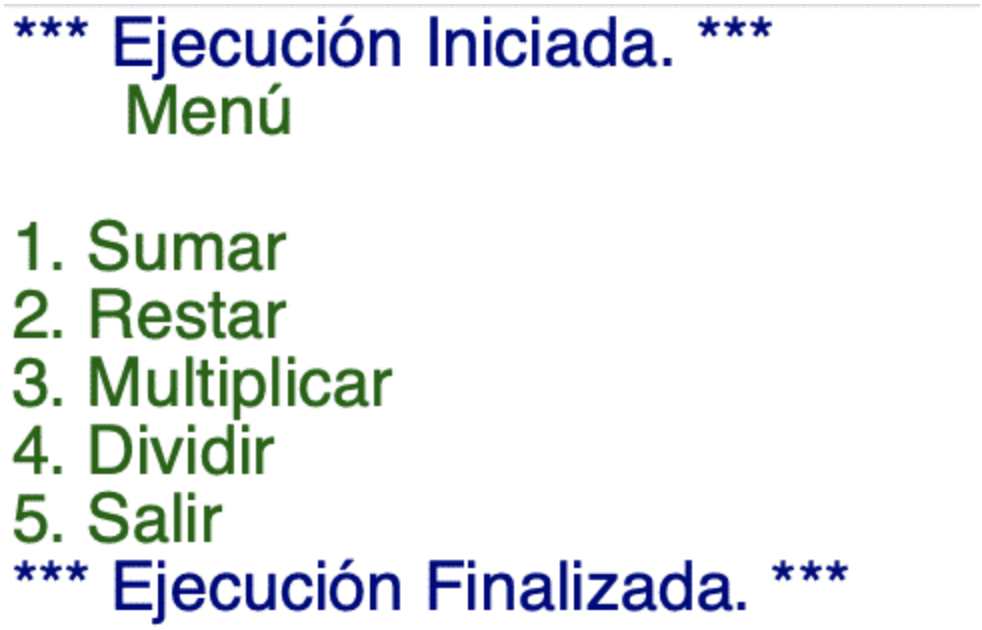
Figura 47: Llamada función PSeInt

```
Proceso calculadora

    menu();

FinProceso
```

Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

El resultado de esta llamada queda de la siguiente manera.

Figura 48: Resultado función PSeInt



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Una de las ventajas de crear funciones es que podemos llamar a este fragmento de código desde diversas partes de nuestro algoritmo, pero si se quiere hacer un cambio o incluir un código, esto se realiza solo una vez dentro de la función.

De esta manera, cuando necesitemos agregar o cambiar el formato de cómo se muestran los mensajes, solo tenemos que cambiar la función Imprimir y no todas las líneas de código donde mostramos el menú.

Funciones anónimas

Las funciones de tipo anónimas se caracterizan por no poseer un nombre. En nuestro ejemplo anterior, el nombre de nuestra función es Menú, el cual usamos para ejecutar dicha función.

Ahora bien, si una función no tiene nombre, ¿cómo la utilizamos? En realidad, estas funciones no se utilizan más que en el lugar donde son definidas, ya que no pueden ser llamadas. El propósito de estas funciones es darle flexibilidad al programador, pero como regla general no deberíamos usarlas, salvo que sea estrictamente necesario, porque agregan más complejidad al código. En su lugar, deberíamos utilizar funciones nombradas o no anónimas, como vimos en el ejemplo anterior.

Tema 4. Recursividad

Llamar una función

En programación, llamar una función significa utilizarla, es decir, agregarla a la ejecución de la aplicación.

Existen algunas situaciones en donde necesitamos otro tipo de iteración, se puede convertir en un código muy complejo al utilizar bucles. Una de las opciones ante esos casos es la recursividad.

Podemos definir un algoritmo recursivo como aquel que se utiliza dentro de sí mismo. Una función recursiva es la codificación de dicho algoritmo, se caracteriza por llamarse a sí misma dentro de la función. Son comunes para expresar series matemáticas cuando se necesitan en una aplicación específica.

Los algoritmos recursivos deberían utilizarse cuando fuera necesario, ya que consumen más recursos computacionales que los algoritmos que no lo son.

Estas funciones tienen 2 características importantes:

- La función debe llamarse a sí misma dentro del código.
- Debe existir un caso base que nos permita detener la ejecución.

Figura 49: Recursividad PSeInt

```
SubProceso val←factorial(num)
    Definir val Como Entero;
    Si num =1 Entonces
        //Caso Base
        val ← 1;
    SiNo
        //Llamada Recursiva
        val ← num * factorial(num-1);
    FinSi

FinSubProceso
```

Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Ejercicio

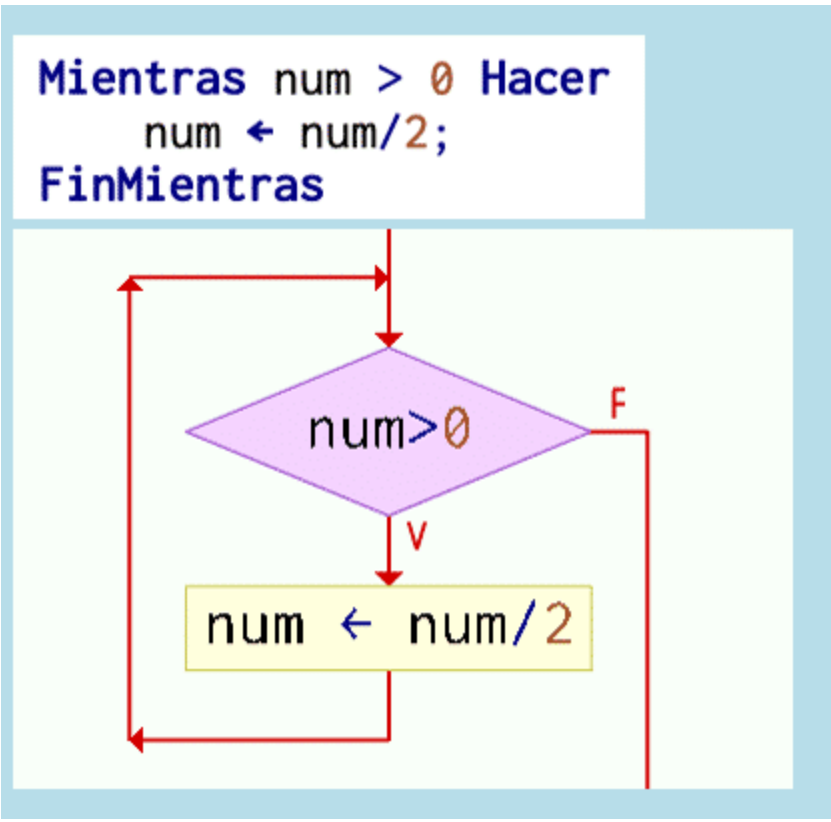
Genera el diagrama de flujo y pseudocódigo para cada una de las siguientes solicitudes.

1. Ciclo iterativo que divide un número en 2, se ejecutará mientras el resultado sea mayor a 0.
2. Ciclo iterativo que multiplica un número por 3, 11 veces.
3. Ciclo iterativo que suma a un número 3 hasta que sea mayor a 100, y que se ejecuta, al menos, una vez.

4. Ciclo condicional que permita identificar mayores y menores de edad.
5. Ciclo condicional que entregue la capital de los siguientes países: (1) España, (2) Francia, (3) Canadá.

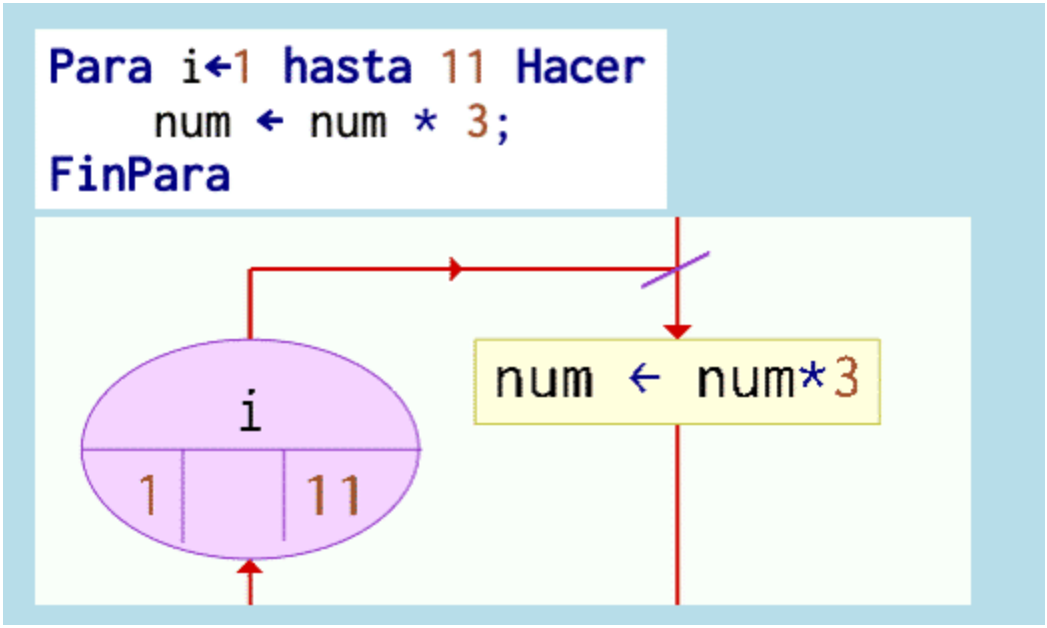
Resultados

Figura 50. Resultados



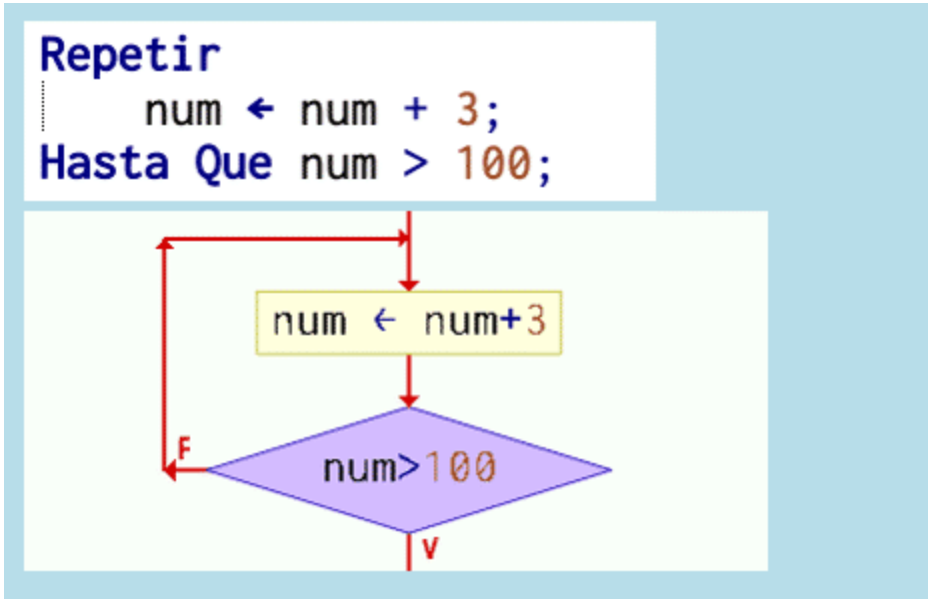
Fuente: elaboración propia a base del ***software PSeInt*** (Novara, 2021).

Figura 51. Resultados



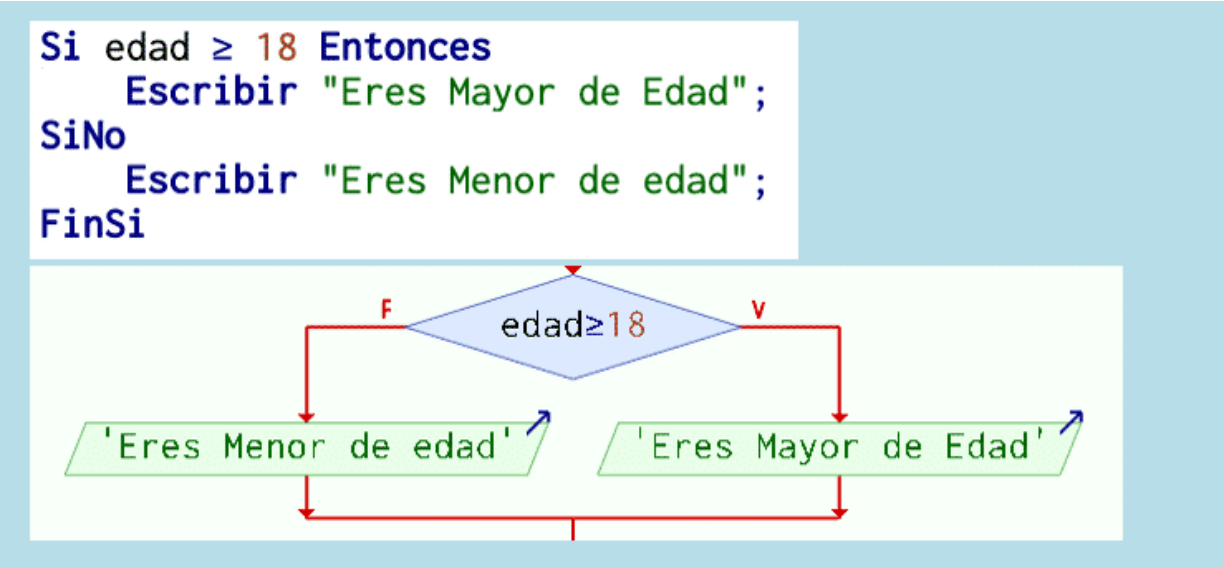
Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Figura 52. Resultados



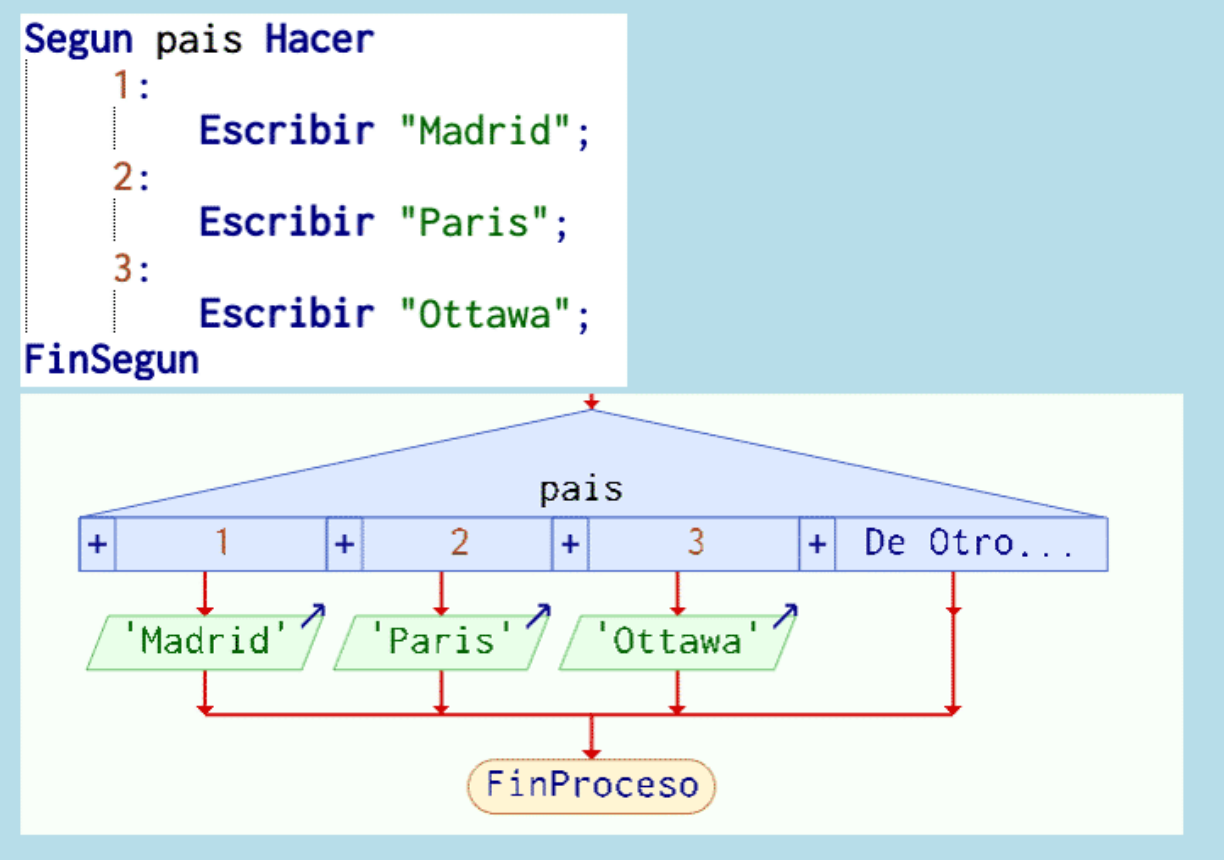
Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

Figura 53. Resultados



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

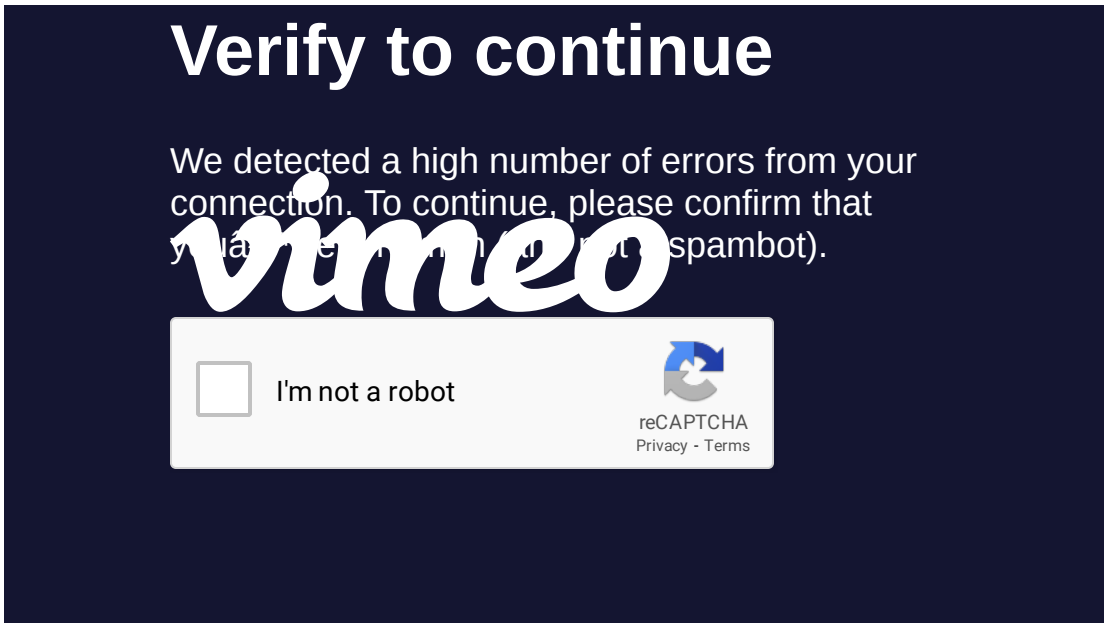
Figura 54. Resultado



Fuente: elaboración propia a base del software PSeInt (Novara, 2021).

CONTINUAR

Video de habilidades



En un diagrama de flujo, si un conjunto de sentencias del algoritmo no está relacionado con el resto se puede representar sin flechas.

- Verdadero.
- Falso.

SUBMIT

Un diagrama de flujo siempre debe comenzar y finalizar con un óvalo (o círculo).

- Verdadero.
- Falso.

SUBMIT

El símbolo del rectángulo puede representar la finalización de un algoritmo.

- Verdadero.
- Falso.

SUBMIT

En el ejemplo del pseudocódigo del video, ¿cuál es una expresión lógica?

- “informarRechazo”.
- “velocidadActual = velocidadMáxima”.
- “velocidadActual = velocidadActual + 10”.
- “leerVelocidadActual”.

SUBMIT

Ante la necesidad de diseñar el diagrama de flujo y el pseudocódigo de un algoritmo:

- Primero se tiene que diseñar el diagrama de flujo para luego poder escribir el pseudocódigo.
- Primero se tiene que escribir el pseudocódigo para luego poder diseñar el diagrama de flujo.
- No es necesario realizar primero uno para luego hacer el otro.
- Sí o sí se deben generar al mismo tiempo.

SUBMIT

CONTINUAR

Cierre

Pseudocódigo

Este se define como una representación en un código más explicativo y fácil de leer que los lenguajes de programación.
Los pseudocódigos no se compilan ni interpretan por ninguna computadora. Su propósito es, simplemente, representar un algoritmo o un proceso mediante una sintaxis similar a la presente en los lenguajes de programación.

Diagrama de flujo

Un diagrama de flujo expresa, de manera gráfica, los pasos que se deben seguir y las decisiones que se deben tomar de un algoritmo o proceso específico.
Utilizamos la notación ANSI para identificar los diferentes símbolos que se van a utilizar en el diseño de algoritmos.

Estructuras de control

Para poder crear un programa, necesitamos ser capaces de controlar las acciones del programa que estamos creando.
Al desarrollar ***software***, tendremos tres componentes:

- Evento de entrada
- Proceso
- Salida

Condicionales

Los condicionales son instrucciones que evalúan la veracidad de una sentencia, por lo tanto, utilizan valores booleanos (verdadero o falso) para decidir si ejecutan o no una determinada sección de código.

Ciclos o bucles

Un ciclo o bucle es una sección de código en la cual la ejecución se repite de acuerdo con una condición. Son muy útiles cuando queremos recorrer una lista de elementos o ejecutar una sección de código indefinidamente. Las variantes son:

- El bucle ***Para***
- El bucle ***Mientras***
- El bucle ***Repetir***
- El bucle ***For-each***

Funciones/Funciones anónimas

Hay muchas formas de codificar un sistema y algo particular sobre el desarrollo de software es que no existe una única forma de construir ese sistema.
Las funciones son secciones de código que podemos reutilizar de forma tal que no tengamos que codificar varias veces el mismo código y lo podamos reutilizar.

CONTINUAR

Referencias

Joyanes Aguilar, L. (2007). ***Fundamentos de programación.*** Madrid, ES: McGraw-Hill.

Novara, P. (2021). ***PSeInt*** [***Software*** intérprete de pseudocódigo]. AR.