Elementos de un programa

septiembre 2023

CONTENIDO:

1. Estructura general de un programa.
2. Tipos de dato.
3. Identificadores.
4. Operadores.
5. Expresiones.
6. Variables y constantes.
7. Entrada/salida.

1. ESTRUCTURA GENERAL DE UN PROGRAMA

Un programa es un conjunto secuencial de instrucciones escritas en un lenguaje de programación, que le indican al computador los pasos que debe ejecutar para solucionar un problema.

Escribir un programa (o codificarlo en lenguaje de programación) es lo que llamamos **programación**, que no es más que un proceso para dar solución a problemas, para esto se requieren las siguientes fases de desarrollo:

i. Análisis del problema.
ii. Diseño del algoritmo.
iii. Pruebas del algoritmo.
iv. Codificación del programa.
v. Depuración y pruebas del programa.
vi. Documentación.
vii. Mantenimiento.

Todo el proceso, estructura y documentación correspondientes al análisis del problema (i), se tratará en la guía: Ciclo de resolución de problemas.

En esta guía se desarrolla y describe el proceso a aplicar en los pasos i al vi.

1.1. Instrucciones de un programa

Los pasos secuenciales de un programa son las acciones o **instrucciones** que el computador debe seguir para solucionar un problema. **Una instrucción** puede ser:

a. Una asignación.
b. Una expresión aritmética (ver sección 4, de esta guía.).
c. La entrada de un dato (lectura desde el teclado).
d. La salida de un dato o información (Mostrar en pantalla).
e. Un ciclo de control (condicionales o iterativos).
f. La llamada a una función.

En **Python una instrucción** es una orden para que el intérprete realice una operación determinada.

1.2. Programas lineales y no lineales

De acuerdo al orden de ejecución de las instrucciones un programa puede ser:

1.2.1. Programa lineal

Es un programa que se ejecuta paso a paso en el orden o secuencia en que fueron escritas las instrucciones (También llamado programa secuencial.).

Una vez finalizada la instrucción actual, se podrá ir a la siguiente y no hay forma de regresar a las instrucciones que ya se ejecutaron, el programa continuará ejecutándose hasta finalizar todas las instrucciones.

1.2.2. Programa no lineal

Un programa no lineal permite la alteración de la secuencia de acuerdo a situaciones manejadas por herramientas de los lenguajes de programación. Ese cambio o alteración de la secuencia ocurre de tres formas:

Usando estructuras de control

i. La alteración de la secuencia puede ejecutar una instrucción, ejecutar otra distinta o no ejecutar ninguna, esto, de acuerdo a un proceso de decisiones llamado **ciclo condicional**. Después del condicional se continua con la secuencia del programa.

ii. **Utilizando ciclos iterativos (o de repetición)**, el programa puede repetir muchas veces una instrucción o un conjunto de instrucciones, y cuando finaliza la repetición, continua con la secuencia del programa.

Usando subprogramas

iii. La secuencia se puede alterar con **la llamada a un subprograma**, en este caso, queda en pausa la secuencia del programa para que el computador inicie el conjunto de instrucciones que correspondan a un subprograma, cuando el mencionado subprograma finalice se retomará la secuencia del programa que fue pausada.

1.3. Elementos básicos de un programa

Los elementos básicos constitutivos de un programa o algoritmo son:

- Palabras reservadas.
- Identificadores. (nombres de variables y funciones,).
- Expresiones
- Instrucciones.
- Caracteres especiales (coma, punto y coma, paréntesis).
- Constantes.
- Variables.

La forma correcta de usar y escribir estos elementos forma parte de **las reglas de sintaxis** de un lenguaje de programación, solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora, los programas que contengan errores de sintaxis serán rechazados.

Existen otros elementos además de estos elementos básicos, que también forman parte de los programas, y son vitales para el buen diseño de un algoritmo o programa. Estos elementos son:

- Estructuras de control
  - Secuenciales
  - Condicionales
  - Iterativas
- Contadores o acumuladores
- Banderas, centinelas o interruptores.

Un buen programador necesita conocer todos estos elementos de programación, que son herramientas, que al integrarlas en los programas permiten solucionar cualquier problema.

2. TIPOS DE DATO

La solución a un problema requiere de un conjunto de datos, estos datos o valores que serán utilizados por un programa deben estar asociados a un tipo de dato, y representados por identificadores de variables se almacenaran en la memoria del computador.

Los tipos de datos aportan estructura y un rango de valores donde estarán enmarcados los datos de un problema, estos datos pueden ser de tipo numérico, carácter, booleano, cadena de caracteres, o estructuras de dato.

Cada dato contenido en la RAM es considerado un objeto, en lenguaje de programación Python todo es un objeto.

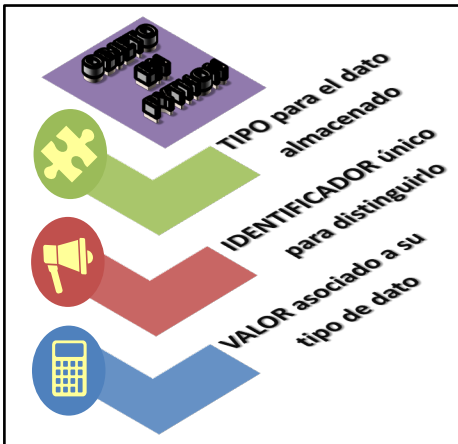

Figura 1: Campos que contiene un objeto

2.1. Tipos de dato primitivos

Los tipos de dato primitivos son lo más elemental de un lenguaje de programación. Son los componentes básicos para tratar todo tipo de valores de datos puros y simples.

Python tiene cuatro tipos primitivos:

i. Enteros (int).
ii. Reales (float).
iii. Booleanos (bool).
iv. Cadenas de caracteres o string (str).

En Python Los tipos de datos compuestos estándar se pueden clasificar como:

Mutable: El valor asociado puede cambiar en tiempo de ejecución.

Inmutable: El valor asociado no puede cambiar de forma directa en tiempo de ejecución, a menos que se le especifique un cambio de tipo.

Tipos de dato primitivos (básicos)

	Tipo	Rango de valores	Descripción
Numéricos inmutables	`int`	-2.147.483.648 a 2.147.483.647	En plataformas de 32bits
		-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	En plataformas de 64bits
	`float`	$2.2250738585072020 \times 10^{-308}$ a $1.7976931348623157 \times 10^{308}$	Basado en el tipo real double para mayor precisión en los cálculos.
	`complex`	se representan de la siguiente forma: 1.3 + 7j	Números complejos: Poseen una parte real y una imaginaria, se almacena usando coma flotante, debido a que estos números son una extensión de los reales.
Booleanos inmutables	`bool`	True o False	Contiene los valores True o False. También son numéricos ya que True=1 y False=0
Secuencias inmutables	`str`	Compuestas por uno o más de los caracteres representados en la tabla ASCII	Las cadenas de caracteres, contienen caracteres encerrados entre comillas simples (') o dobles ("). También usa *basestring*. **Son tipos de dato de secuencia.**

Tipos de dato de secuencia

	Tipo	Valores	Descripción
Secuencias inmutables	`unicode`	Compuestos por cadenas de caracteres Unicode	Las clases están integradas para los tipos de cadenas de caracteres en: *basestring*
	`tuple`	Ejemplo: valores = ("Py", True, 5, 'a')	Son objetos de tipo secuencia, un tipo de dato lista inmutable. Esta no puede modificarse de ningún modo después de su creación.

Estructuras de datos

	Tipo	Descripción
Secuencias mutables	`List`	es una estructura de datos formada por una secuencia ordenada de objetos. Ejemplo: [-2, 5, 80, 9] Son heterogéneas (pueden estar conformadas por elementos de distintos tipos, incluidos otras listas) y son mutables (sus elementos pueden modificarse).
Conjuntos mutables	`set`	Es una colección no ordenada y sin elementos repetidos. Usos básicos: (a) verificación de pertenencia y (b) eliminación de entradas duplicadas
Conjuntos inmutables	`frozenset`	Sin orden, no contiene duplicados y compuestos por objetos inmutables

Tipos de datos integrados

	Tipo	Descripción
Objetos integrados	`None`	Posee solo un valor. Se accede a este objeto a través del nombre incorporado "*None*". Se utiliza para indicar la ausencia de un valor, por ejemplo, se devuelve desde funciones que no retornan nada explícitamente. Su valor de verdad es *False*.
	`NotImplemented`	Posee un solo valor. Se accede a este objeto usando el nombre incorporado "*NotImplemented*".
	`file`	El objeto *file()* se implementa con métodos integrados. Las clases de tipo archivos es utilizada para crear, abrir y manipular archivos.

3. IDENTIFICADORES

Existe un orden en la memoria RAM, donde es guardado un objeto que corresponde a un **tipo de dato**, estos objetos deben estar contenidos en espacios de almacenamiento que poseen una única dirección de memoria, cada acceso a la memoria se acompaña de esa dirección especifica del espacio al que se hace referencia. Recordar estas direcciones de memoria es complicado, es aquí donde surge la utilidad de los **identificadores**.

Un **identificador** es un nombre escrito bajo un conjunto de reglas establecidas en el lenguaje de programación, son utilizados para hacer referencia a un espacio de memoria donde está almacenado un dato (Wirth, 1986).

Son nombres únicos que se le da a variables, constantes y subprogramas.

3.1. Reglas para escribir un identificador

i. Solamente se pueden escribir con (a) letras mayúsculas y minúsculas del alfabeto inglés, (b) dígitos del 0 al 9, y (c) guion bajo.

ii. Solo pueden iniciar con una letra o un guion bajo.

iii. Pueden tener cualquier longitud.

iv. En *Pyton* son *case-sensitive*, es decir, son sensibles al uso de mayúsculas y minúsculas.

v. No se puede usar palabras reservadas para nombrar identificadores (ver la lista de palabras reservadas en la siguiente página).

vi. No puede haber identificadores duplicados.

Ejemplos de identificadores:

Correctos	
Identificador	Descripción
`edad`	Compuesto solo por letras
`_cedula`	Inicia con un guion bajo
`valor_1`	Inicia con una letra

Incorrectos	
Identificador	Descripción
`*`	No es un carácter válido
`mes actual`	Contiene un espacio
`1erValor`	Inicia con un número
`str`	Es una palabra reservada
`$hoy`	Inicia con el símbolo "$"

Lista de palabras reservadas (“*keywords*”) de Python.			
False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

Las palabras claves que están subrayadas su primera letra debe estar siempre en mayúsculas. Fuente: Ayuda del terminal Python 3.11.4

4. OPERADORES

La solución a un problema requiere de algún tipo de cálculo aritmético, comparativo o lógico; para realizar estos cálculos los lenguajes de programación incorporan **operadores** que en combinación con los valores aportados por los datos generaran resultados de solución.

La aridad de un operador: Es el número de operandos o argumentos necesarios para que un operador específico funcione, se pueda evaluar o calcular. En programación de acuerdo con su aridad los operadores se clasifican en:

a. Operadores binarios:
Un operador binario requiere de dos operandos unidos por un operador central con el que se obtendrá un resultado, ejemplos:

i. `5 + 9` ii. `a >= b` iii. `p and q`

En los ejemplos i, se suma cinco más nueve. En el ii, se esta verificando en la operación relacional si “a” en mayor o igual que “b”. Y el ejemplo iii, realiza la conjunción lógica “p” y “q”. Cada operador requiere de un valor (operando o argumento) a la izquierda y otro a la derecha para poder realizar la operación.

b. Operadores unarios:
Un operador unario solo requiere un valor u operando que siempre estará ubicado a la derecha del operador, ejemplo:

i. `-1` ii. `+10` iii. `not p`

En el ejemplo i, se muestra el operador “-” unario, convirtiendo el número uno en negativo. El ejemplo ii, utiliza el “+” unario para convertir el diez en positivo, y el ejemplo iii, se utiliza el operador unario “not” para invertir el valor booleano contenido en “P”.

La asociatividad de operadores: indica el sentido en que será resuelta la operación individual cuando ocurra que en una misma expresión aparezca **más de un** operador con la misma prioridad

(ver el apartado 3.6 donde se especifica la prioridad de operadores). Tendremos solo dos tipos de asociatividad: (i) por la izquierda y (ii) por la derecha. (La asociatividad se indica en las tablas de operadores de las secciones 4.1. a la 4.4.). Ejemplos:

i. Asociatividad por la izquierda:

```
5 * 3 + 2 * 6
```

Tenemos dos multiplicaciones, (1ro.) resuelve la que está más a la izquierda 5 * 3, (2do.) luego 2 * 6 y final mente se suman los dos resultados parciales (resultado: 27).

ii. Asociatividad por la derecha:

```
2 ** 3 / 2 ** 1
```

Se nos presentan dos exponenciaciones en la misma expresión, (1ro.) se resuelve 2 ** 1, (2do.) luego 2 ** 3 y por último se dividen los dos resultados parciales (resultado: 4)

4.1. Operadores aritméticos:

El resultado obtenido de estas operaciones siempre será un valor numérico que puede ser de tipo real (float) o de tipo entero (int).

Operador	Descripción	Aridad	Asociatividad
+	Suma	Binario	A la izquierda
-	Resta		
*	Multiplicación		
/	División real		
//	División entera		
%	Residuo de la división entera		
**	Exponenciación		A la derecha

4.2. Operadores relacionales

Las operaciones relacionales siempre arrojaran un resultado de tipo booleano (bool)

Operador	Descripción	Aridad	Asociatividad
==	Igual que (igualdad)	Binario	A la izquierda
!=	Diferente de		
<	Menor que		
<=	Menor o igual que		
>	Mayor que		
>=	Mayor o igual que		

4.3. Operadores lógicos

Los operadores lógicos solo operan con valores (operandos) de tipo booleano y su resultado siempre será un valor booleano (bool).

Operador	Descripción	Aridad	Asociatividad
`and`	Conjunción ("y" lógica)	Binario	A la Izquierda
`or`	Disyunción ("o" lógica)		
`not`	Negación o inversor	Unario	

4.4. Operadores delimitadores

Se utilizan para establecer un orden de prioridad superior a otros operadores dentro de la expresión, también pueden ser usados para ayudar a dar orden al escribir una expresión muy larga y facilitar la lectura de la expresión usándolos como ayuda visual (al usarlos como ayuda visual no debe cambiar la prioridad y el funcionamiento de la expresión).

Operador	Descripción	Asociatividad
`(`	Paréntesis que abre	A la Izquierda
`)`	Paréntesis que cierra	

4.5. Operadores de asignación

4.5.1. Asignación

La asignación simple toma el valor que se encuentra a su derecha y lo almacena en la variable que esta a su izquierda, este valor puede ser (a) un valor directo perteneciente a alguno de los tipos de dato, (b) un valor almacenado en otra variable, (c) un valor resultado de una expresión a la derecha de la asignación o (d) un valor retornado desde la llamada a una función también a la derecha del operador.

Operador	Descripción	Asociatividad
`=`	El signo igual se usa como asignación	El valor de la derecha se asigna a la variable de la izquierda, ejemplo: edad = 18

4.5.2. Asignación aumentada

La asignación aumentada es una asignación que cumple funciones del operador que la compone y luego asigna el valor resultante, simplificando la forma de escribir la expresión.

Las asignaciones aumentadas son operadores binarios, ideales para usarlas como contadores o acumuladores, en la siguiente tabla se muestra la descripción de cada asignación:

Operador	Operación	Descripción	Funcionamiento
`+=`	Suma	`y+=9` equivale a `y= y + 9`	De acuerdo a los ejemplos presentados en la descripción: se evalúa la operación de "y" con el valor nueve (9) y luego el resultado se asigna a la variable "y" (el valor de "y" es sustituido por el resultado de la operación).
`-=`	Resta	`y-=9` equivale a `y= y - 9`	
`*=`	Multiplicación	`y*=9` equivale a `y= y * 9`	
`/=`	División real	`y/=9` equivale a `y= y / 9`	
`//=`	División entera	`y//=9` equivale a `y= y // 9`	
`%=`	Residuo de la división entera	`y%=9` equivale a `y= y % 9`	
`**=`	Potencia	`y**=9` equivale a `y= y ** 9`	

4.6. Prioridad de operadores

En una expresión podemos encontrar distintos tipos de operadores, en la tabla siguiente se indica cual es la prioridad o precedencia entre unos y otros, el operador de prioridad más alta se evalúa antes que el operador con prioridad más baja.

Para otros casos referentes a la prioridad de operadores que no puedan ser resueltos con esta tabla, tendremos que aplicar las reglas de evaluación de expresiones (ver en el apartado 4.3.).

Operación	Operador	Prioridad
Delimitadores	`(  )`	1 - Alta -
Potencia	`**`	2
Cambio de signo	`+  -`	2
Multiplicación y divisiones	`*  /  //  %`	3
Suma y resta	`+  -`	4
Operadores relacionales	`==  !=  <  <=  >  >=`	5
Negación o inversor	`not`	6
Conjunción	`and`	7
Disyunción	`or`	8
Asignación	`=`	9 - Baja -

5. EXPRESIONES

> "Son combinaciones de variables, constantes, llamadas a subprogramas y/o valores separados por operadores" (Cuba, 2014).

Una expresión es una porción de código que al ser evaluada produce o un valor (resultado). Una expresión puede ser:

a. Una variable constituye por si sola una expresión evaluable.
b. La asignación de un valor a una variable.
c. Una constante o variable, cuando esta precedida por un operador unario.
d. Un conjunto de constantes, variables, funciones o valores; separados por operadores.

5.1. Tipos de expresiones

Los operadores o conjunto de operadores que componen una expresión, determinan a que tipo pertenece, y pueden ser:

a. **Expresiones aritméticas**: compuestas de operadores aritméticos, operan con de tipo de dato int o float, y su resultado es de tipo int o float.

b. **Expresiones relacionales**: compuestas de operadores relacionales, operan con valores (int, float, carácter, str y otros) su resultado siempre será de tipo de dato booleano (bool).

c. **Expresiones lógicas**: compuestas de operadores lógicos, solo operan con valores bool y arrojan resultados de tipo de dato bool.

d. **Expresiones mixtas**: compuestas de una combinación de varios tipos de operadores (aritméticos, relacionales y lógicos), siempre arrojarán resultados de tipo de dato bool.

Los operadores delimitadores (paréntesis) y la asignación simple (=), pueden ser parte de cualquier expresión, pero no influyen de forma alguna con su clasificación.

Estos operadores de asignación aumentada: `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`; pueden constituir por si solos expresiones aritméticas o dependiendo de las combinaciones con otros operadores podrán ser expresiones mixtas.

5.2. Evaluación de expresiones

La evaluación de una expresión muestra la necesidad de una íntima interconexión entre el procesador y la memoria, dado que el flujo de información entre ambos es bastante elevado la memoria contiene objetos que deben ser accesibles mediante distintos nombres (identificadores), entonces:

a. La memoria (RAM) contiene los objetos a manipular, a estos objetos los llamamos datos.
b. El procesador (unidad aritmético lógica) esta unidad realiza las operaciones y los datos son tomados o leídos de la memoria para ser procesados.
c. Luego, los resultados son escritos en la memoria.

El procesador solo contiene dos datos (para el caso de operadores binarios) a la vez para ser procesados inmediatamente, para el caso de operadores unarios solo contendrá un valor.

Para evaluar una expresión con varios operadores y resultados intermedios aplicamos la técnica de descomponer una tarea complicada en una secuencia de tareas más simples, aplicando un conjunto de **reglas establecidas para la evaluación de presiones**, entonces:

a. Las instrucciones del programa obtienen los valores guardados en memoria RAM para realizar operaciones.
b. Las operaciones emplean al procesador específicamente los registros dentro del procesador para guardar los datos intermedios.
c. Y las instrucciones finalmente depositan los resultados de nuevo en espacios de memoria RAM.

Este método para descomponer enunciados en pasos más elementales y salvar resultados intermedios en la memoria es la razón por la que los mismos procesos de cálculo pueden ejecutarse en cualquier computadora, el método de descomposición es la esencia misma de la programación en las computadoras, la base para la aplicación de mecanismos relativamente simples para la resolución de problemas de enorme complejidad.

La evaluación de expresiones es fundamental para entender el funcionamiento de un programa, es parte esencial de **la prueba experimental de programa** puede ser empleada para encontrar y corregir errores. (Wirth, 1986)

5.3. Reglas para la evaluación de expresiones.

Evaluar una expresión consiste en ejecutar todos los cálculos de sus operadores con los valores involucrados para saber cuál es su valor resultado. **Se debe aplicar estas reglas de evaluación de expresiones en todo momento hasta llegar al resultado**. A continuación, se listan las mencionadas reglas de evaluación de expresiones:

i. Se aplica la prioridad de operadores.
ii. Cuando en una expresión tenemos más de un operador con la misma prioridad se evalúa primero el que esta más a la izquierda (asociatividad por la izquierda).
iii. En el caso de la potencia (cuando hay más de una potencia en la expresión), se evalúa primero el de la derecha (asociatividad por la derecha).
iv. Las operaciones que están entre paréntesis se evalúan primero y cuando tenemos más de un paréntesis se evalúa primero el que está más a la izquierda.
Los paréntesis más externos son los que determinan la prioridad, y luego si dentro de un paréntesis hay otros paréntesis, estos que están dentro tendrán la mayor prioridad en la expresión interna.
v. Los cálculos individuales de operadores binarios se ejecutan de izquierda a derecha.

6. VARIABLES Y CONSTANTES

6.1. Variables

Una variable es un espacio de memoria RAM que esta asociado a un identificador, en ella se almacena un dato que pertenece a alguno de los tipos de datos existentes.

En Python las variables son mutables. Es decir, el valor que almacena la variable puede cambiar, y su tipo de dato también puede cambiar, todo esto durante la ejecución del programa.

6.1.1. Declaración de variables

La sintaxis para declarar variables consiste en escribir un identificador seguido de una asignación, y luego un valor perteneciente a los tipos de dato conocidos.

En Python una variable queda declarada la primera vez que se le asigna un valor.
Ejemplos:

```
saludo = "Buenos días"
edad = 18
sueldo = 130.0
salir = False
```

Importante: si el dato a almacenar es texto o un carácter debe escribirlo entre comillas.

La sintaxis establece que siempre:

i. Primero, se escribe a la izquierda el nombre de la variable.
ii. Luego el operador de asignación.
iii. Por último, a la derecha se escribe el valor que va a guardar.

6.2. Constantes

Una constante es un espacio de memoria RAM que esta asociado a un identificador, en ella se almacena un dato que pertenece a alguno de los tipos de datos existentes, **pero este dato asignado no puede cambiar de valor**.

En Python (Python 3), las constantes no existen, pero se pueden simular.

6.2.1. Declaración de constantes

Es posible declarar una constante tal como se declara una variable, pero su identificador debe estar escrito únicamente en mayúsculas. Ejemplo:

```
CIEN_PORCIENTO = 100
```

Pero esta declaración corresponde a una variable, por lo que es responsabilidad del programador evitar que cambie de valor.

6.2.2. Declaración de constantes inamovibles usando tuplas

En Python las tuplas son inamovibles lo que nos permitiría crear constantes que no cambien de valor.

Crearemos una tupla de un solo valor y la asociaremos a un identificador que usaremos como constante cada vez que se necesite:

i. Se importa "*namedtuple*" desde "*collections*", esta acción se ejecuta solo una vez al principio del programa, junto a las demás importaciones que sean necesarias.
ii. Primero se invoca la función "*namedtuple*" que necesitará:
    1) El nombre del tipo de dato que representará esa tupla (será Constantes),
    2) Y el nombre de la constante (CONST).
    3) Indicaremos los valores de las constantes (3.141592)
    4) Se asignará al nombre de la tupla "PI".
iii. Ya podemos usar la tupla con nombres con sus constantes. Ejemplo:

```
# i. Se importa "namedtuple" desde "collections"
from collections import namedtuple

# ii. Se invoca la función "namedtuple" y se aplican
      los pasos: 1, 2, 3 y 4
PI = namedtuple('constantes', 'CONST')(3.141592)

# iii. accedemos al valor de nuestra constante
print(PI.CONST)
```

La última instrucción print(PI.CONST), muestra en pantalla el valor de la constante PI.CONST:

```
3.141592
```

En Python no existen las constantes estrictas y tampoco se pueden simular a la perfección. Cada alternativa para hacerlo tiene sus problemas y detalles, aunque **se recomienda la simulación con "*namedtuples*"**.

Conceptualmente sí existen las constantes en Python. Es deber del programador mantener la filosofía de Python en la que aquellas variables declaradas con identificadores en mayúsculas pasan a ser constantes y su valor no puede (debe) ser modificado.

7. ENTRADA/SALIDA

7.1. Entrada (Función input)

Cuando hablamos de entrada, nos referimos a la forma como un programa recibe los datos desde el exterior y esos datos serán almacenados en variables.

Solo hay dos formas de que una variable reciba datos para almacenarlos:

i. **Cuando se le asignan**: Utilizando el operador de asignación o algún operador de asignación aumentada (ver sección 4.5 de esta guía).

ii. **Cuando se leen**: Durante la ejecución de un programa puede ser necesario recibir datos desde el exterior, generalmente solicitándolos a la persona que está usando el programa.

Los datos a leer provienen del teclado (que es la entrada estándar). En el programa esta acción de leer se logra utilizando la función input.

7.1.1. Función Input: Cuando se lee un dato se necesita indicarle al usuario del programa cual es el dato que se requiere y esperar que ese dato sea escrito desde el teclado.

Pasos que ejecuta la función input:

i. Especificar cual es el dato que se necesita y que será ingresado desde el teclado.

ii. Mostrar la solicitud y el cursor en pantalla, indicando que esta esperando que ingresen el dato desde el teclado y luego presionen “enter”.
iii. Almacenar el dato ingresado en una variable.
iv. Finaliza el proceso de lectura de este dato.

Sintaxis para implementar input:

La siguiente línea de código muestra la sintaxis para leer un dato desde el teclado usando la función input:

```
edad = input('Ingrese su edad: ')
```

Este ejemplo muestra en pantalla el mensaje “Ingrese su edad: ”, una vez ingresada de acuerdo a los pasos de ejecución de input, el dato es guardado en la variable edad utilizando el operador de asignación. Al finalizar la instrucción la edad ingresada queda almacenada en la variable edad.

Importante: esta forma de capturar datos de pantalla genera problemas cuando los datos requeridos son datos numéricos, el ejemplo anterior captura una edad pero la guarda como tipo string (cadena de caracteres), este inconveniente se soluciona con un conjunto de funciones que a continuación se describen.

Es posible leer desde el teclado datos de tipo entero, real, carácter y cadena de caracteres, pero el valor ingresado es guardado como un string para prevenir esto y realmente capturar el tipo de dato esperado debemos aplicar un proceso de “casting” usando las siguientes funciones:

Funciones de casting para conversión a tipos de datos

Tipo	Función	Descripción
Entero	`int()`	Convierte a entero
Real	`float()`	Convierte a flotante
Booleano	`bool()`	Entrega False si es cero y True en el resto
Cadena de caracteres	`str()`	Convierte a string

El anterior ejemplo de sintaxis con input, donde se solicitaba la edad de la persona se puede mejorar para que la edad ingresada sea guardada como un valor de tipo de dato entero, para haremos casting con la función del tipo primitivo int() como se muestra a continuación:

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

Con esta instrucción estamos leyendo la edad ingresada y guardándola en la variable como un valor de tipo de dato entero (int).

Del mismo modo también podemos hacer casting con las funciones float(), bool() y str().

7.2. Salida (función print)

La salida se genera utilizando la función print, con ella podemos mostrar en pantalla (salida stándar):

- Mensajes o cadenas de texto.

```
print("Verificar la edad de una persona")
```

- Mensajes de texto usando comillas simples.

```
print('Solo si es "mayor de edad" ')
```

- Valores contenidos en variables.

```
print("Su edad es", edad, "años")
```

- Mostrar párrafos de texto "*docstrings*" usando triple comilla doble, Python respetara los saltos de línea tal cual está escrito en el código,

```
print("""
 Se solicitará su edad,
 ud. debe ingresar un valor entero
""")
```

- Resultados de operaciones.

```
print("El próximo año tendrá: ", edad + 1, " años")
```

- Valores retornados por funciones.

```
print("Ud. tiene: ", int(edad), " años")
```

7.3. Saltos de línea usando print

La función print siempre realiza un salto de línea después de escribir su contenido, pero a veces no queremos saltar a la línea siguiente después de mostrar algún mensaje en pantalla. Para evitar el salto de línea, puede agregar la palabra reservada "end", que le indica al interprete de Python como terminar la línea que esta mostrando.

La sintaxis para evitar el salto automático de línea es la siguiente:

```
print('Bienvenido, por favor espere... ', end='')
print('Hora ingrese su edad:')
```

En el ejemplo anterior, a pesar de escribir dos prints en dos líneas separadas, al colocar end=’’ se muestra en pantalla como si ambos mensajes estuvieran concatenados:

```
Bienvenido, por favor espere… Hora ingrese su edad:
```

Entre las comillas de end podemos colocar el caracter que deseemos:

```
print('Bienvenido, por favor espere', end='.')
print(' Hora ingrese su edad:')
```

Y mostrará en pantalla un punto al final de la primera cadena de texto:

```
Bienvenido, por favor espere. Hora ingrese su edad:
```

REFERENCIAS

Castaño, S. (s.f.). *Python desde cero*. Recuperado el 18 de 08 de 2023, de https://controlautomaticoeducacion.com/category/python-desde-cero/

Cuba, N. (9 de Mayo de 2005). Programación estructutrada: Introducción a los diagramas de flujo. Puerto Ordaz, Venezuela.

Joyanes Aguilar, L. (1987). *Metodología de la programación: Diagramas de flujo, algoritmos y programación estructurada* (Primera ed.). Juárez, México: McGraw-Hill de México, S.A. de C.V

Joyanes Aguilar, L. (2008). *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (4° ed.). Madrid, España: McGraw-Hill/Interamericana de España, S.A.U.

Laca, M. (s.f.). *Aprende a programar en Python*. Recuperado el 21 de 08 de 2023, de https://pythones.net/

Marzal Varó, A., Gracia Luengo, I., & García Sevilla, P. (2014). *Introducción a la programación en Python 3* (1° ed.). (P. d. I., Ed.) Castelló de la Plana, España. Obtenido de http://www.tenda.uji.es

Prieto Espinoza, A., Lloris Ruíz, A., & Torres Cantero, J. C. (2002). *Introducción a la informática* (Tercera ed.). Madrid, España: McGraw-Hill/Interamericana de España, S.A.U.

Wirth, N. (1986). *Introducción a la programación sistemática* (Segunda ed.). Buenos Aires, Argentina: El Ateneo.