CTOP - PYTHON

Unidad 02

Desarrollo de algoritmos

Índice

Unidad 01 Desarrollo de algoritmos

1. Objetivos

Después de completar esta unidad, será capaz de:

- Crear diagramas de flujo que representen algoritmos de manera efectiva.
- Escribir código en Python que implemente algoritmos diseñados.
- Realizar pruebas para validar la funcionalidad de los algoritmos.
- Identificar y solucionar errores en algoritmos.
- Explicar el proceso de desarrollo de un algoritmo de forma clara.

2. Diagramas de flujo y pseudocódigo

Un **algoritmo** es un conjunto ordenado y finito de pasos que permiten resolver un problema o realizar una tarea específica. En programación, los algoritmos son la base del desarrollo del software, ya que describen la lógica antes de traducirla a código.

Para representar algoritmos, se utilizan herramientas como los **diagramas de flujo** y el **pseudocódigo**.

2.1 Diagramas de flujo

Un **diagrama de flujo** es una representación gráfica de los pasos o fases de un algoritmo. Utiliza símbolos normalizados para representar operaciones, decisiones o inicios y finales.

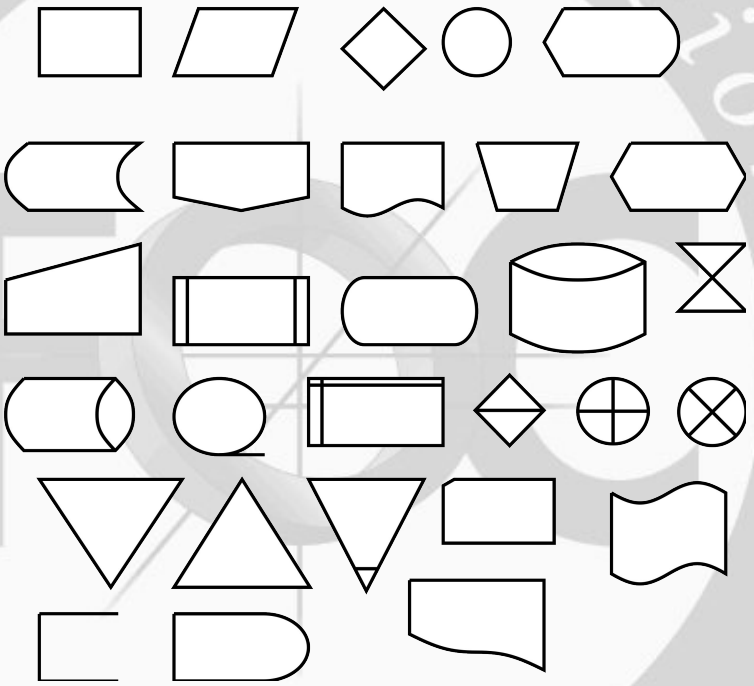

Imagen: Símbolos normalizados de un diagrama de flujo

Entre las ventajas de los diagramas de flujo, podemos destacar que:

- Facilitan la **comprensión visual de un proceso**.
- Permiten **detectar errores lógicos** antes de programar.
- Son útiles para **documentar el funcionamiento** de un programa.

El siguiente diagrama de flujo representa el flujo de ejecución de un algoritmo muy simple, que consiste en leer un número entero y comprobar si es mayor, igual o menor que cero, y en función de la condición que se cumpla, mostrar un mensaje al usuario y finalizar el programa.

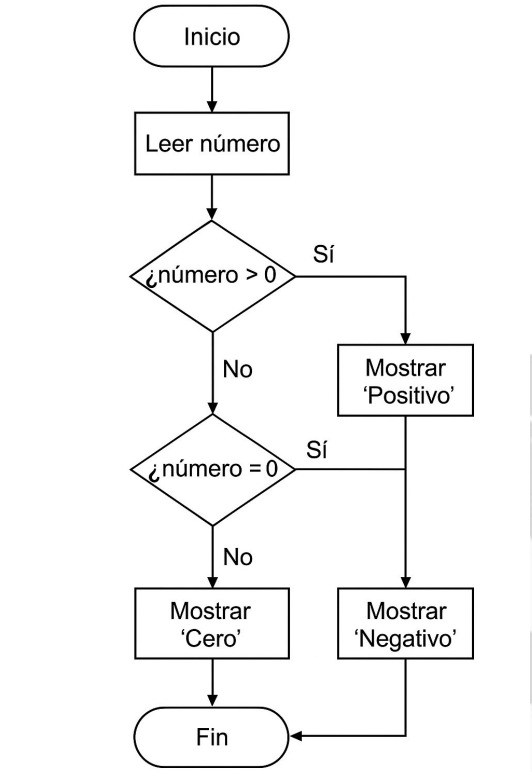

Imagen: Diagrama de flujo

2.2 Pseudocódigo

El **pseudocódigo** es una descripción textual de los pasos de un algoritmo, escrita usando una sintaxis muy próxima al lenguaje natural. De esta forma, resulta más fácil entender qué se está haciendo, paso a paso, para resolver un problema dado.

Un ejemplo sencillo de pseudocódigo, aplicado al algoritmo del diagrama de flujo anterior, sería:

```
Inicio
LEER número
SI número > 0 ENTONCES:
      Escribir "Positivo"
      SINO
      SI número = 0 ENTONCES:
            Escribir "Cero"
            SINO: Escribir "Negativo"
Fin
```

Algunas pautas genéricas y útiles para diseñar algoritmos eficientes son:

- Definir claramente el problema y los datos de entrada/salida.
- Dividir el problema en pasos lógicos y simples.
- Evitar repeticiones innecesarias.
- Verificar la correcta terminación del algoritmo (¿cuál es la condición de parada?).

3. Condicionales y operadores lógicos

3.1 Operador de asignación (=)

El **operador de asignación** (**=**) se utiliza para almacenar un valor en una variable. Por ejemplo, la sentencia: x = 10 asignaría el valor 10 a la variable x.

Como Python es un lenguaje con **tipado dinámico**, basta asignar un valor (sea un entero, un número real, un literal de tipo cadena, etc) para que la variable adopte el tipo del valor asignado, sin necesidad de declaración previa.

3.2 Operador de igualdad (==)

El **operador de igualdad** (**==**) compara dos valores, y devuelve **True** si son iguales o **False** en caso contrario. Ejemplo: la expresión: 5 == 5 devuelve **True**.
Su negación es el **operador de desigualdad** (**!=**) que devuelve **True** cuando los valores son diferentes. Ejemplo: la expresión:5 != 3 devuelve **True**.

3.3 Sentencia condicional 'if-elif-else'

La **sentencia condicional** permite controlar y ejecutar diferentes bloques de código en función de si se cumplen (o no) ciertas condiciones.

La sintaxis básica es:

```
if condición:
     # código si la condición es verdadera
elif otra_condición_1:
     # código si la primera fue falsa y esta es verdadera
...
elif otra_condición_N:
     # código si la (N-1) fue falsa y esta es verdadera
else:
     # código si ninguna condición se cumple
```

El algoritmo del pseudocódigo y del diagrama de flujo anterior podría codificarse en Python mediante el siguiente bloque condicional 'if-elif-else':

```
x = int(input('Introduce un número entero:'))
if x > 0:
      print("Positivo")
elif x == 0:
      print("Cero")
else:
      print("Negativo")
```

4. Errores y excepciones

Los **errores**, también conocidos como '**bugs**', son fallos en el código que impiden su correcto funcionamiento. Pueden deberse a un fallo de escritura, a una lógica incorrecta o a situaciones imprevistas durante la ejecución. Básicamente, los tres tipos de errores que pueden darse: **errores de sintaxis**, **errores lógicos** y **errores de ejecución**. Detectar y corregir los errores es una parte esencial del proceso de programación.

La detección y corrección de errores forma parte del ciclo de desarrollo del software. Se pueden identificar usando **mensajes de error**, **impresión de variables** o **herramientas de depuración**.

4.1. Tipos principales de errores en Python

- **Errores de sintaxis:** ocurren cuando el código no cumple las reglas del lenguaje. Son advertidos de forma inmediata por el intérprete al procesar el código. Ejemplo: olvidar los dos puntos (**:**) al final de una instrucción **if**.
- **Errores de ejecución:** se producen mientras el programa se está ejecutando. Por ejemplo, al intentar dividir entre cero o acceder a un índice inexistente.
- **Errores lógicos:** el programa se ejecuta sin mostrar errores, pero el resultado obtenido no es el esperado debido a un fallo en la lógica del algoritmo.

4.2. Manejo de excepciones con 'try-except'

Python utiliza las **excepciones** para manejar errores de ejecución. Incluso aunque el programa esté impecablemente escrito (sintácticamente perfecto), puede dar lugar a excepciones durante su ejecución. Las excepciones son errores detectados durante la ejecución, y no son incondicionalmente fatales (el programa no tiene porqué terminar siempre si existe una excepción).

El **manejo de excepciones** permite que un programa continúe ejecutándose incluso cuando ocurre un error. Para ello se utiliza la estructura 'try-except', que prueba un bloque de código y captura las excepciones que puedan surgir. El bloque **'try-except'** permite controlar los errores y que el programa continúe así de forma segura.

A continuación, un ejemplo simple de bloque **'try-except'** que comprueba que el usuario haya proporcionado un valor de tipo entero:

```
try:
      numero = int(input("Introduce un número: "))
      print("El número es", numero)
except ValueError:
      print("Error: Debes introducir un número válido.")
```

Es posible manejar varios tipos de errores con diferentes bloques **'except'**, o capturar cualquier error con **'Exception'**:

```
try:
      resultado = 10
      int(input("Introduce un divisor: "))
except ValueError:
      print("Debes introducir un número.")
except ZeroDivisionError:
      print("No se puede dividir entre cero.")
except Exception as e:
      print("Ha ocurrido un error inesperado:", e)
```

También puede usarse **'else'** para ejecutar código si no se produce ningún error, y **'finally'** para ejecutar siempre un bloque, ocurra o no una excepción. Veamos el siguiente ejemplo de código que comprueba que un fichero exista, y en tal caso, lea su contenido y finalmente, siempre cierre el descriptor de fichero:

```
try:
      archivo = open("datos.txt", "r")
      contenido = archivo.read()
except FileNotFoundError:
      print("Archivo no encontrado.")
else:
      print("Archivo leído correctamente.")
finally:
      archivo.close()
```

El manejo de excepciones mejora la **robustez del código**, evitando que errores simples detengan el programa por completo.

5. Pruebas y depuración en Python

Las **pruebas** y la **depuración (*debugging*)** son fases esenciales en el desarrollo de programas informáticos. Permiten garantizar la calidad, fiabilidad y corrección del software antes de su entrega o implementación.

Pruebas de software

Las **pruebas** consisten en ejecutar el programa en diferentes condiciones para comprobar si se comporta como se espera. Son procedimientos que permiten verificar que el programa cumple los requisitos y funciona correctamente.

Principales tipos de pruebas

- **Pruebas unitarias:** verifican que las funciones o módulos individuales funcionan correctamente.
- **Pruebas de integración:** comprueban que las distintas partes del sistema funcionan correctamente juntas.
- **Pruebas de sistema:** evalúan el sistema completo en condiciones reales de uso.
- **Pruebas de aceptación:** confirman que el sistema cumple los requisitos del usuario final.

Python incluye el módulo **'unittest'**, que permite automatizar las pruebas.

Por ejemplo, en el siguiente código, se comprueba que la función multiplicar devuelve el resultado correcto de multiplicar los enteros 3 y 4 (12):

```
import unittest

def multiplicar(a, b):
     return a * b

class TestMultiplicar(unittest.TestCase):
     def test_resultado(self):
          self.assertEqual(multiplicar(3, 4), 12)

if __name__ == '__main__':
     unittest.main()
```

Depuración del código

La **depuración** (***debugging***) es el proceso de identificar, analizar y corregir errores en el código. Durante la depuración, el programador puede examinar el flujo de ejecución, los valores de las variables y comprobar si las condiciones se cumplen correctamente. Si un programa no muestra el resultado esperado, usando el depurador puedes localizar en qué momento cambian los valores incorrectamente. Esto permite localizar errores de forma precisa y eficiente.

La combinación de buenas **prácticas de prueba** y **depuración** es esencial para desarrollar software fiable, fácil de mantener y con un comportamiento predecible.

Existen distintas técnicas de depuración:

- **Uso de mensajes 'print()'** para verificar valores intermedios.
- Revisión manual del código para detectar errores lógicos.
- Uso de depuradores integrados en los entornos de desarrollo.

En **IDLE**, el entorno de Python, se puede utilizar el **depurador** (**Debugger**) de la siguiente forma:

- Abrir el menú ***Debug*** → ***Debugger***.
- Marcar la opción ***Source*** para ejecutar el programa paso a paso.
- Observar los valores de las variables en cada paso del programa.
- Establecer **'breakpoints'** (**puntos de parada**) para analizar partes o fragmentos específicos del código. Es decir, el programa avance y se ejecute hasta una determinada instrucción, y verificamos los valores de las variables involucradas para ver si el error está ahí.

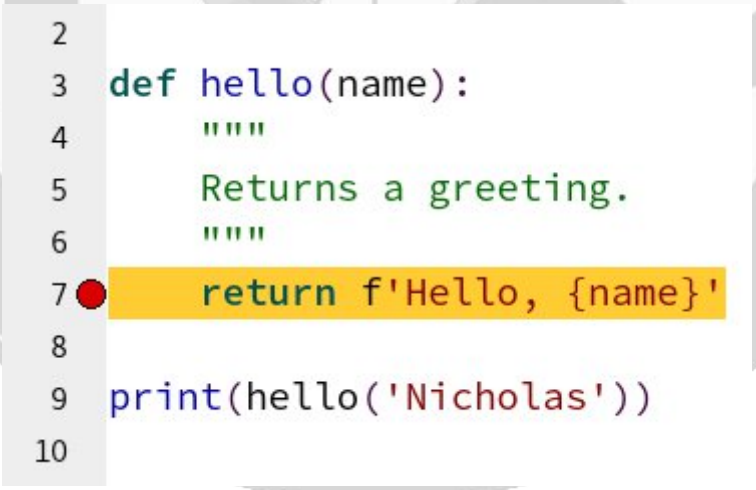

Imagen: Punto de parada (breakpoint) usando el depurador de Python