Universidad de Guadalajara
Centro Universitario de Ciencias Económico Administrativas

Materia
Analitica de datos

Nombre
Jose Roberto Vasquez Luna

Actividad
Actividad 3

Calendario
2024-B

Fecha
19/08/2024

Código
217882996

Carrera

Licenciatura en Tecnologías de la Información

1. Declaración de Variables

Descripción:
En Python, las variables son contenedores que almacenan datos. A diferencia de otros lenguajes, en Python no es necesario declarar explícitamente el tipo de la variable; el tipo se infiere automáticamente en el momento de la asignación.

Código:

```
# Declaración de variables
nombre = "Juan"  # Variable de tipo cadena (string)
edad = 25        # Variable de tipo entero (int)
altura = 1.75    # Variable de tipo flotante (float)
es_estudiante = True  # Variable de tipo booleano (bool)
```

Las variables se nombran de manera que sean descriptivas del valor que almacenan. Esto mejora la legibilidad del código.

2. Tipos de Variables

Descripción:
Los tipos de variables en Python pueden ser de diferentes clases como enteros, flotantes, cadenas de texto, y booleanos. El tipo de variable define qué tipo de datos se pueden almacenar en la variable y qué operaciones se pueden realizar con ella.

Código:

```
# Tipos de variables en Python
entero = 10            # Tipo entero (int)
flotante = 10.5        # Tipo flotante (float)
cadena = "Hola Mundo"  # Tipo cadena de texto (string)
booleano = False       # Tipo booleano (bool)
```

Diferenciar los tipos de variables es esencial para realizar operaciones adecuadas y para evitar errores de tipo en el código.

3. Operadores

Descripción:
Los operadores en Python se utilizan para realizar operaciones sobre variables y valores. Se dividen en varias categorías como aritméticos, de comparación, lógicos, de asignación, y más.

3.1. Operadores Aritméticos

Código:

```
# Operadores aritméticos
a = 10
b = 5

suma = a + b            # Suma: 15
resta = a - b           # Resta: 5
multiplicacion = a * b  # Multiplicación: 50
division = a / b        # División: 2.0
modulo = a % b          # Módulo: 0 (resto de la división)
exponente = a ** b      # Exponente: 100000
```

Estos operadores permiten realizar cálculos básicos, necesarios para la mayoría de las aplicaciones.

3.2. Operadores de Comparación

Código:

```
# Operadores de comparación
a = 10
b = 5

es_igual = (a == b)        # Igualdad: False
es_diferente = (a != b)    # Diferencia: True
mayor_que = (a > b)        # Mayor que: True
menor_que = (a < b)        # Menor que: False
mayor_o_igual = (a >= b)   # Mayor o igual: True
menor_o_igual = (a <= b)   # Menor o igual: False
```

Los operadores de comparación son esenciales para la toma de decisiones en los programas (por ejemplo, en estructuras condicionales).

3.3. Operadores Lógicos

Código:

```
# Operadores lógicos
x = True
y = False

resultado_and = x and y  # AND lógico: False
resultado_or = x or y    # OR lógico: True
resultado_not = not x    # NOT lógico: False
```

Los operadores lógicos se utilizan para combinar condiciones, lo que permite la evaluación de múltiples expresiones booleanas.

4. Comentarios

Descripción:
Los comentarios en Python se indican con el símbolo #. Son esenciales para explicar el código y hacer que sea más fácil de entender para otros desarrolladores (o para ti mismo en el futuro).

Código:

```
# Este es un comentario de una sola línea

"""
Este es un comentario
de múltiples líneas
"""
```

Los comentarios no afectan la ejecución del código y son vitales para la documentación y mantenimiento del código.

5. Funciones en Python

Las funciones en Python se definen utilizando la palabra clave `def`, seguida del nombre de la función, paréntesis y dos puntos.

El cuerpo de la función debe estar indentado.

```
# Ejemplo básico de función en Python
def saludar(nombre):
    '''Esta función imprime un saludo personalizado.'''
    print(f'Hola, {nombre}!')

# Llamar a la función
saludar('Juan')
```

Las funciones permiten organizar el código en bloques más pequeños y manejables. Esto mejora la legibilidad, facilita la depuración y permite la reutilización del código.

5.1. Parámetros y Argumentos

Las funciones pueden aceptar parámetros, que son valores que se pasan a la función para ser utilizados en su interior. Cuando se llama a la función, se deben proporcionar argumentos que correspondan a estos parámetros.

```
# Función con múltiples parámetros
def sumar(a, b):
    '''Esta función retorna la suma de dos números.'''
    return a + b

# Llamar a la función con argumentos
resultado = sumar(3, 5)
print(f'La suma es: {resultado}')
```

El uso de parámetros y argumentos hace que las funciones sean más flexibles y adaptables a diferentes contextos.

5.2. Funciones con Valores por Defecto

Es posible definir valores por defecto para los parámetros de una función. Esto permite que la función sea llamada sin necesidad de proporcionar un argumento para esos parámetros.

```
# Función con valor por defecto en un parámetro
def saludar(nombre, mensaje='Hola'):
    '''Esta función imprime un saludo con un mensaje opcional.'''
```

```
    print(f'{mensaje}, {nombre}!')

# Llamar a la función con y sin el argumento opcional
saludar('Juan')  # Usa el valor por defecto
saludar('Juan', 'Buenos días')  # Usa un valor personalizado
```

Los valores por defecto simplifican las llamadas a funciones, haciendo el código más limpio y menos propenso a errores.

6. Módulos en Python

6.1. Creación de Módulos

Un módulo en Python es simplemente un archivo con extensión `.py` que contiene funciones, variables, y otros elementos. Estos módulos pueden ser importados en otros scripts o módulos.

```
# Ejemplo de un módulo simple (archivo `mi_modulo.py`):
# def saludar(nombre):
#     print(f'Hola, {nombre}!')
#
# def despedir(nombre):
#     print(f'Adiós, {nombre}!')
```

6.2. Importación de Módulos

Para utilizar un módulo en otro script, se debe importar utilizando la palabra clave `import`. También es posible importar funciones o variables específicas desde un módulo utilizando `from ... import ...`.

```
# Importar un módulo completo
import mi_modulo

# Usar funciones del módulo
mi_modulo.saludar('Juan')
mi_modulo.despedir('Juan')

# Importar funciones específicas de un módulo
from mi_modulo import saludar

# Usar la función importada
saludar('Juan')
```

La organización del código en módulos hace que sea más fácil de mantener y reutilizar, permitiendo una estructura más modular y escalable.

Las estructuras condicionales permiten que el programa ejecute diferentes bloques de código en función de ciertas condiciones.

7. If, Elif, Else

La estructura `if` evalúa una condición y, si es verdadera, ejecuta el bloque de código asociado. La estructura `elif` permite evaluar condiciones adicionales si la primera es falsa, y `else` se ejecuta si todas las condiciones anteriores son falsas.

```
# Ejemplo de estructura if, elif, else
edad = 20

if edad < 18:
    print("Eres menor de edad.")
elif 18 <= edad < 65:
    print("Eres adulto.")
else:
    print("Eres adulto mayor.")
```

Las estructuras condicionales permiten tomar decisiones en el código y ejecutar diferentes bloques de código según las necesidades del programa.

8. Bucles en Python

Los bucles permiten ejecutar repetidamente un bloque de código mientras se cumpla una condición o sobre un rango de elementos.

8.1. Bucle For

El bucle `for` itera sobre una secuencia (como una lista o rango) y ejecuta un bloque de código para cada elemento de la secuencia.

```
# Ejemplo de bucle for
for i in range(5):
    print(f'Iteración {i}')
```

El bucle `for` es ideal para iterar sobre secuencias con un número conocido de elementos, permitiendo realizar operaciones en cada uno de ellos.

8.2. Bucle While

El bucle `while` continúa ejecutando un bloque de código mientras la condición especificada sea verdadera.

python
Copy code

```
# Ejemplo de bucle while
contador = 0

while contador < 5:
    print(f'Contador: {contador}')
    contador += 1
```

El bucle `while` es útil cuando no se conoce de antemano cuántas veces se debe ejecutar el bloque de código, sino que depende de una condición.

8.3. Bucle anidado

Los bucles pueden ser anidados dentro de otros bucles, lo que permite realizar iteraciones múltiples.

```
# Ejemplo de bucle anidado
for i in range(3):
    for j in range(2):
        print(f'i = {i}, j = {j}')
```

Los bucles anidados permiten realizar operaciones más complejas, como recorrer matrices o estructuras más complejas de datos.

Además de las estructuras condicionales y bucles, existen otras declaraciones de control de flujo que permiten alterar el comportamiento normal del código.

8.4. Break

La declaración `break` se utiliza para salir de un bucle antes de que termine su ejecución completa.

```
# Ejemplo de uso de break
for i in range(5):
    if i == 3:
        break
    print(f'Iteración {i}')
```

`break` permite interrumpir un bucle cuando ya no es necesario continuar, mejorando la eficiencia del código.

8.5. Continue

La declaración `continue` se utiliza para saltar a la siguiente iteración de un bucle, omitiendo el resto del código en la iteración actual.

```
# Ejemplo de uso de continue
for i in range(5):
    if i == 2:
        continue
    print(f'Iteración {i}')
```

`continue` permite omitir ciertas iteraciones en un bucle cuando no es necesario ejecutarlas, lo que puede simplificar la lógica del código.

8.6. Pass

La declaración `pass` es una operación nula; no hace nada y se utiliza como un marcador de posición donde se necesita un bloque de código sintácticamente pero no se requiere ninguna operación.

```
# Ejemplo de uso de pass
for i in range(5):
    if i == 3:
        pass  # Aquí no se realiza ninguna acción
    print(f'Iteración {i}')
```

`pass` se utiliza para definir estructuras de código incompletas o como un marcador de posición para futuras implementaciones.