Programación Imperativa II

Semántica de la asignación

- Sea e una expresión cuya evaluación no modifica el estado

  #estado a

  ```
  v = e;
  ```

  #vale $v == e@a \land z_1 = z_1@a \land \cdots \land z_k = z_k@a$

  donde $z_1, \ldots, z_k$ son todas las variables del programa en cuestión distintas a v que están definidas hasta ese momento.

- Las otras variables se supone que no cambian así que, por convención, no hace falta decir nada.

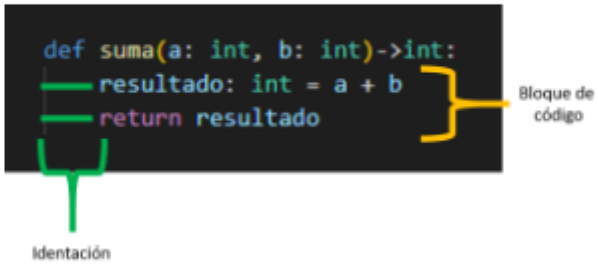
- Si la expresión e es la invocación a una función que recibe parámetros por referencia, puede haber más cambios, pero al menos sabemos que:

  #vale $v == e@a$

  está en la *poscondición* o *asegura* de la asignación.

Identación

- La identación es a un lenguaje de programación, lo que la sangría al lenguajo humano escrito.
- En ciertos lenguajes de programación, la identación determina la presencia de un bloque de instrucciones (Python es uno de ellos).
- En otros lenguajes, un bloque puede determirse de otra manera: por ejemplo encerrandolo entre llaves { }.



Alcance, ámbito o scope

El concepto de "alcance" o "ámbito" (scope en inglés) se refiere al contexto en el que una variable o un identificador es válido y puede ser utilizado dentro de un programa. Define la parte del programa donde una variable puede ser referenciada y su acceso es válido.

El alcance determina la visibilidad y disponibilidad de una variable en diferentes secciones del código, como funciones, bloques de código o archivos. Ayuda a evitar conflictos y colisiones entre variables con el mismo nombre y permite un uso adecuado de los recursos en un programa.

- **Alcance global**: Una variable con alcance global *se declara fuera* de cualquier función o bloque de código y *es accesible desde cualquier parte* del programa. Está disponible para todos los módulos, funciones y bloques de código que se encuentren dentro del programa. Una vez declarada, la variable global puede ser utilizada en cualquier sección.
- **Alcance local**: Una variable con alcance local *se declara dentro* de una función o bloque de código y *solo es visible y accesible dentro de ese ámbito específico*. Las variables locales solo existen mientras la función o el bloque de código están en ejecución y *se destruyen una vez que la ejecución sale de ese ámbito*. Además, si se define una variable con el mismo nombre en un ámbito local y global, la variable local tendrá precedencia dentro de su ámbito.

Python

Variables locales

- x sólo está definida dentro del bloque de instrucciones del procedimiento ejemploLocalScope.
- El intento de acceder a x fuera del procedimiento termina en un error en tiempo de ejecución.
- Aunque el IDE ya nos lo había advertido.

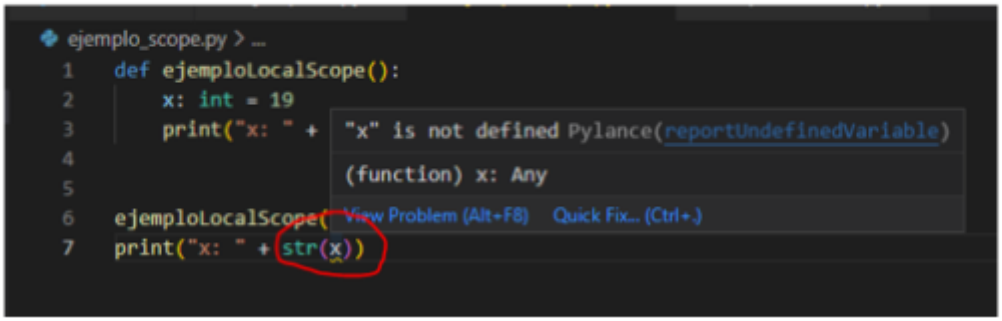
```
def ejemploLocalScope():
    x: int = 19
    print("x: " + str(x))


ejemploLocalScope()
print("x: " + str(x))
```

Variable Local

Imprimirá: x: 19

NameError: name 'x' is not defined



Variables globales

Un ejemplo con Python

- x sólo está definida de manera global.
- Cualquier bloque puede acceder a ella.

```
def ejemploGlobalScope():
    print("x: " + str(x))

def sumarEnElGlobal():
    global x
    x = x + 120
    print("x: " + str(x))

x: int = 20
ejemploGlobalScope()
sumarEnElGlobal()
ejemploGlobalScope()
print("x: " + str(x))
```

Imprimirá:
- Primera vez: x: 20
- Segunda vez: x; 140

En Python, explícitamente hay que referenciar a la variable global para modificarla.

Variable Global

Tipos de alcance en Python

- **Alcance global**: Las variables declaradas fuera de cualquier función o clase se consideran globales y están disponibles en todo el programa. Pueden ser accedidas y modificadas tanto dentro como fuera de las funciones.

```
x = 10 # Variable global

def my_function():

   print(x) # Accediendo a la variable global dentro de la función

my_function() # Imprime 10
```

- **Alcance local**: Las variables declaradas dentro de una función tienen alcance local y solo son accesibles dentro de esa función. Estas variables se crean cuando la función es llamada y se destruyen al finalizar la ejecución de la función.

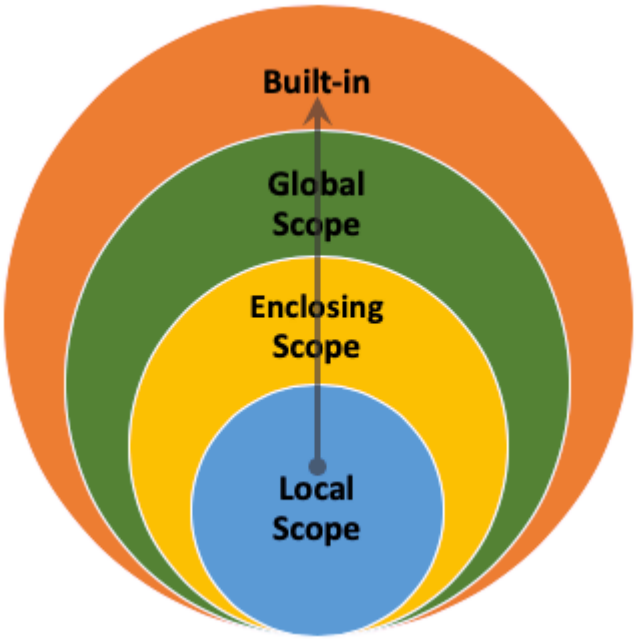
```
def my_function():
    y = 20  # Variable local
    print(y)

my_function()  # Imprime 20
print(y)  # Genera un error, ya que 'y' no está definida fuera de la función
```

- **Alcance de bloque (no local o enclosed)**: A partir de Python 3.x, los bloques de código delimitados por llaves, como los utilizados en estructuras de control (if, for, while), no crean un nuevo alcance. Las variables definidas dentro de estos bloques están disponibles fuera de ellos.

```
if True:
    z = 30 # Variable disponible fuera del bloque if
print(z) # Imprime 30
```

- **Alcance integrado o built-in:** Son todas las declaraciones propias de Python (ej: def, print, etc). Se refiere a un alcance predefinido que contiene un conjunto de funciones y objetos que son parte del lenguaje Python sin necesidad de importar módulos adicionales. Estas funciones y objetos son accesibles desde cualquier parte del programa sin la necesidad de declararlos o definirlos previamente.



La referencias también tienen su scope:

- Al pasar un parámetro por referencia, esta referencia vivirá dentro del scope de la función
- Analicemos este caso:
  - En la primer instrucción, *y* toma las referencias de *x* (en este scope, las referencias de *y* se 'pierden')
  - Al modificar *y*, se está modificando el valor de *x*
  - Al salir de la función, *y* nunca cambió

```
4  def duplicar(x: list, y: list):
5      y = x
6      y *= 2
7
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

ANTES:
x: ['a', 'b', 'c']
y: ['d', 'e']
DESPUES:
x: ['a', 'b', 'c', 'a', 'b', 'c']
y: ['d', 'e']
```

Condicionales

- Todo el condicional tiene su precondición y su postcondición: P_{if} y Q_{if}
- Cada bloque de instrucciones, también tiene sus precondiciones y postcondiciones.

```
  # estado P_if
if (B):
      # estado P_uno
    uno
      # estado Q_uno
else:
      # estado P_dos
    dos
      # estado Q_dos
  # estado Q_if
  # (B ∧ estado Q_uno) ∨ (¬B ∧ estado Q_dos)
  # Después del IF, se cumplió B y Q_uno o, no se cumplió B y Q_dos
```

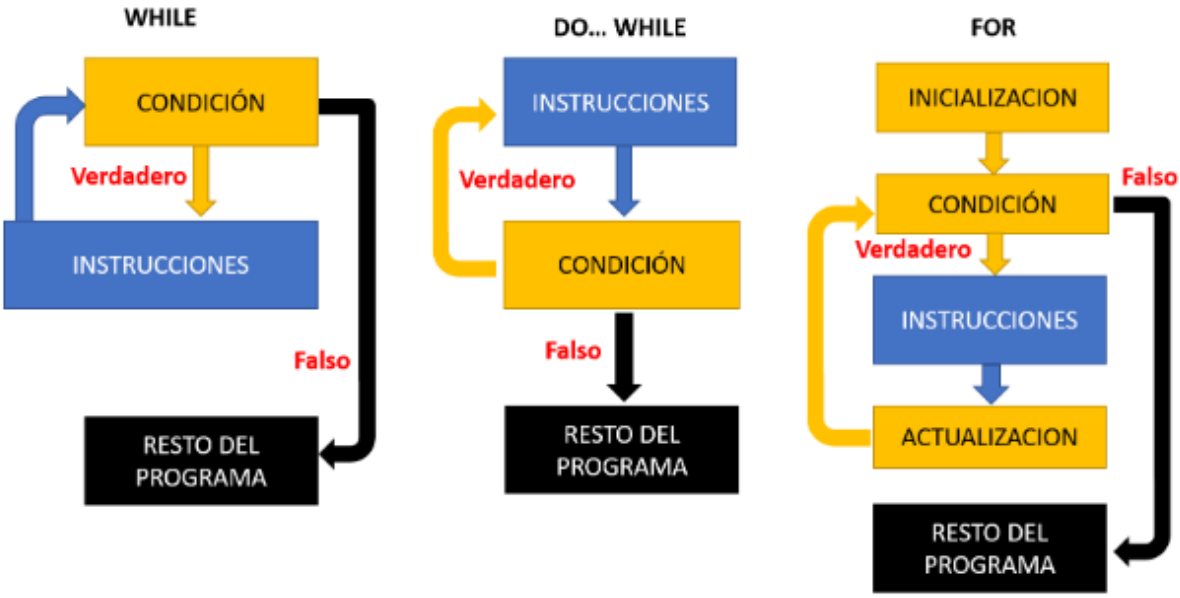
B tiene que ser una expresión booleana (se llama "guarda").

```python
def compare_numbers(a: int, b: int) -> str:
    if a > b:
        return "El número 'a' es mayor que 'b'"
    elif a < b:
        return "El número 'b' es mayor que 'a'"
    else:
        return "Los números 'a' y 'b' son iguales"

# Ejemplo de uso
num1 = 10
num2 = 5
resultado = compare_numbers(num1, num2)
print(resultado)
```

Ciclos

- En los lenguajes imperativos, existen estructuras de control encargadas de repetir un bloque de código mientras se cumpla una condición.
- Cada repetición suele llamarse iteración.
- Existen diferentes esquemas de iteración, los más conocidos son:
  - While, Do While, For



- While

Un programa que muestra por pantalla el número ingresado por el usuario, hasta que el usuario ingresa 0.

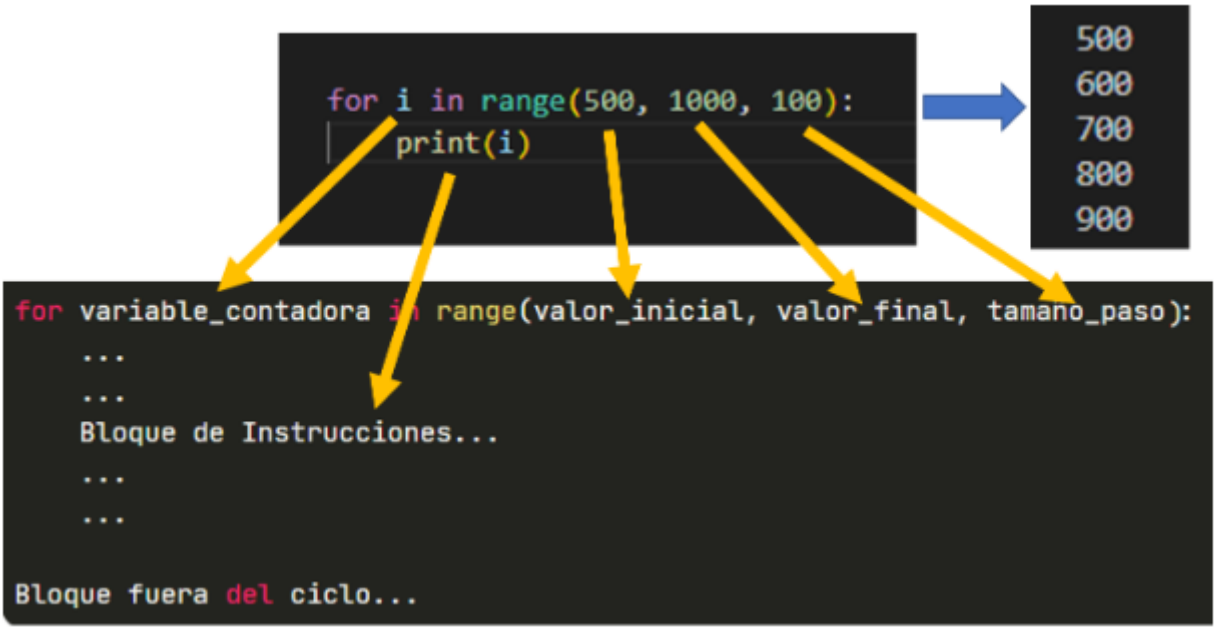
```
numero = int(input('Ingresa un número. 0 para terminar: '))

while(numero != 0):
    print('Usted ingresó: ', numero)
    numero = int(input('Ingresa un número. 0 para terminar: '))

print('Fin del programa.')
```

- input: espera que el usuario ingrese algo por teclado.
- int(input('...')): convierte en int lo que el usuario ingresó por teclado.

- For



La instrucción "**break**" se utiliza en ciclos (como "for" o "while") en muchos lenguajes de programación, incluyendo Python, para **interrumpir** la ejecución del ciclo de forma prematura. Cuando se encuentra la instrucción "break", el ciclo se detiene inmediatamente y el control del programa se transfiere a la siguiente línea de código después del ciclo.

Ciclos y transformación de estados

La relación entre los ciclos y la transformación de estados radica en que los ciclos suelen utilizarse para iterar sobre una serie de estados o elementos y aplicar transformaciones a medida que se avanza en cada iteración. En cada ciclo, se puede realizar una transformación en el estado actual, lo que puede implicar actualizar valores, acumular resultados, modificar estructuras de datos, entre otros.

```
while(condición de finalización):
    ...
    ...
    Bloque de Instrucciones...
    Dentro del while
    ...
    ...

Bloque de Instrucciones...
FUERA del while
```

- En un programa imperativo, cada instrucción transforma el estado.
- Mediante la transformación de estados, podemos hacer una ejecución simbólica del programa.
- ¿Cómo sería la transformación de estados de un ciclo?
  - Podemos pensar en el ciclo como una instrucción: con un estado previo y uno posterior
  - ¿Qué sucede dentro del ciclo? ¿Qué sucede en cada iteración?
- Más adelante en la carrera, verán cómo manejar estas situaciones.