

Introducción a la Programación Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2025

Introducción a la Programación Imperativa

IP - AED I: Temario de la clase

- ▶ Introducción a la programación imperativa
 - ▶ Programación imperativa (Diferencias con funcional)
 - ▶ Características de python
 - ▶ Qué es un programa en python?
 - ▶ Variables en imperativo
 - ▶ La Asignación
 - ▶ Instrucción Return
 - ▶ Transformación de estados. Ejecución Simbólica
 - ▶ Tipos de pasaje de parámetros. Por valor y Por referencia.
 - ▶ Extensión del lenguaje de especificación
 - ▶ Pasaje de argumentos en python
 - ▶ Scope de variables
 - ▶ Condicionales (IF... ELSE..)
 - ▶ Ciclos (While, For)

Algoritmos y programas

Repasando, retomando, continuando...

- ▶ Primero aprendimos a especificar problemas.
- ▶ El objetivo luego fue escribir un **algoritmo** que cumpla esa especificación:
 - ▶ Secuencia de pasos que pueden llevarse a cabo mecánicamente.
- ▶ Puede haber varios algoritmos que cumplan una misma especificación.

Algoritmos y programas

Repasando, retomando, continuando...

- ▶ Primero aprendimos a especificar problemas.
- ▶ El objetivo luego fue escribir un **algoritmo** que cumpla esa especificación:
 - ▶ Secuencia de pasos que pueden llevarse a cabo mecánicamente.
- ▶ Puede haber varios algoritmos que cumplan una misma especificación.
- ▶ Una vez que se tiene el algoritmo, se escribe el **programa**:
 - ▶ Expresión formal de un algoritmo.
 - ▶ Lenguajes de programación:
 - ▶ Sintaxis definida.
 - ▶ Semántica definida.
 - ▶ Qué hace una computadora cuando recibe ese programa.
 - ▶ Qué especificaciones cumple.
 - ▶ Ejemplos: Haskell, C, C++, C#, Java, Smalltalk, Prolog, etc.

Paradigmas

Repasando, retomando, continuando...

- ▶ Paradigmas de programación:
 - ▶ Formas de pensar un algoritmo que cumpla una especificación.
 - ▶ Cada uno tiene asociado un conjunto de lenguajes.
 - ▶ Nos llevan a encarar la programación según ese paradigma.
- ▶ Próximo paso: Programación imperativa.

Programación imperativa (diferencias con funcional)

- ▶ Los programas no necesariamente son funciones
 - ▶ Ahora pueden *devolver* más de un valor
 - ▶ Hay nuevas formas de pasar argumentos

Programación imperativa (diferencias con funcional)

- ▶ Los programas no necesariamente son funciones
 - ▶ Ahora pueden *devolver* más de un valor
 - ▶ Hay nuevas formas de pasar argumentos
- ▶ Nuevo concepto de **variables**
 - ▶ Posiciones de memoria
 - ▶ Cambian explícitamente de valor a lo largo de la ejecución de un programa
 - ▶ Pérdida de la transparencia referencial

Programación imperativa (diferencias con funcional)

- ▶ Nueva operación: la asignación
 - ▶ Cambiar el valor de una variable

Programación imperativa (diferencias con funcional)

- ▶ Nueva operación: la asignación
 - ▶ Cambiar el valor de una variable
- ▶ Distinto mecanismo de repetición
 - ▶ En lugar de la recursión nosotros usaremos la **iteración**

Programación imperativa (diferencias con funcional)

- ▶ Nueva operación: la asignación
 - ▶ Cambiar el valor de una variable
- ▶ Distinto mecanismo de repetición
 - ▶ En lugar de la recursión nosotros usaremos la **iteración**
- ▶ Nuevo tipo de datos: el **arreglo**
 - ▶ Secuencia de valores de un tipo (como las listas)
 - ▶ Longitud prefijada
 - ▶ Acceso directo a una posición (en las listas, hay que acceder primero a las anteriores)
 - ▶ (y también habrá listas, y muchos otros tipos más... como en cualquier lenguaje)

Lenguaje Python

- ▶ Vamos a usarlo para la programación imperativa (también soporta parte del paradigma de objetos, y parte del paradigma funcional)
- ▶ Vamos a usar un subconjunto (como hicimos con Haskell)
 - ▶ No objetos, no memoria dinámica, etc.
 - ▶ Sí vamos a usar la notación de clases, para definir tipos de datos
- ▶ Es un lenguaje interpretado
- ▶ Tiene tipado dinámico:
 - ▶ Una variable puede tomar valores de distintos tipos
 - ▶ *Nosotros lo vamos a pensar con tipado estático (con fines didácticos)*
 - ▶ Declararemos el tipo de cada variable en tiempo de diseño
- ▶ Es fuertemente tipado:
 - ▶ Dado el valor de una variable de un tipo concreto, no se puede usar como si fuera de otro tipo distinto a menos que se haga una conversión explícita de tipos (casting). Es decir, no se permiten violaciones de los tipos de datos.

Programa Python

- ▶ Colección de tipos y funciones.
- ▶ Definición de función:

```
def nombreFunción (parámetros) -> tipoResultado:  
    bloqueInstrucciones
```
- ▶ Su evaluación consiste en ejecutar una por una las instrucciones del bloque.
- ▶ El orden entre las instrucciones es importante:
 - ▶ Siempre de arriba hacia abajo.

Programa Python

► Ejemplo

```
problema suma2( $x : \mathbb{Z}, y : \mathbb{Z}$ ) :  $\mathbb{Z}$ {  
  asegura:  $res = x + y$   
}
```

```
def suma2 (x: int, y: int) -> int:  
    res: int = x + y  
    return res
```

Programa Python

Aclaración por aquello de tipado dinámico versus tipado estático

- ▶ Tipado dinámico: Una variable puede tomar valores de distintos tipos
- ▶ Tipado estático: La comprobación de tipificación se realiza durante la compilación (y no durante la ejecución)

Python es un lenguaje de tipado dinámico, por lo tanto no es necesario declarar los tipos de sus variables.

```
def suma2 (x, y):  
    res = x + y  
    return res
```

- ▶ En la materia lo pediremos con fines didácticos.
- ▶ Existen implementaciones de Python con tipado estático
 - ▶ <https://mypy.readthedocs.io/en/stable/>

```
def suma2 (x: int, y: int) -> int:  
    res: int = x + y  
    return res
```

Variables en imperativo

- ▶ Nombre asociado a un espacio de memoria
- ▶ La variable puede tomar distintos valores a lo largo de la ejecución
- ▶ En Python se **declaran** dando su nombre (y opcionalmente su tipo)
`x: int` # `x` es una variable de tipo `int`
`y: float` # `y` es una variable de tipo `float`
- ▶ Programación imperativa
 - ▶ Conjunto de variables
 - ▶ Instrucciones que van cambiando sus valores
 - ▶ Los valores finales, deberían resolver el problema

Instrucciones

- ▶ Asignación
- ▶ Condicional (if ... else ...)
- ▶ Ciclos (while ...)
- ▶ Procedimientos
(funciones que no devuelven valores pero modifican sus argumentos)
- ▶ Retorno de control (con un valor, return)

La asignación

- ▶ Es la operación fundamental para modificar el valor de una variable.
 - ▶ Sintaxis: *variable = expresión;*
- ▶ Es una operación asimétrica
 - ▶ Lado izquierdo: debe ir una variable u otra expresión que represente una posición de memoria.
 - ▶ Lado derecho: una expresión del mismo tipo que la variable
 - ▶ constante,
 - ▶ variable o
 - ▶ función aplicada a argumentos.
- ▶ Efecto de la asignación:
 1. Se evalúa la expresión de la derecha y se obtiene un valor.
 2. Ese valor se copia en el espacio de memoria de la variable.
 3. El resto de la memoria no cambia.

La asignación

► Ejemplos:

`x = 0`

`y = x`

`x = x+x`

`x = suma2(z+1,3)`

`x = x*x + 2*y + z`

La asignación

► Ejemplos:

`x = 0`

`y = x`

`x = x+x`

`x = suma2(z+1,3)`

`x = x*x + 2*y + z`

► No son asignaciones:

`3 = x`

`doble (x) = y`

`8*x = 8`

La instrucción *return*

- ▶ Termina la ejecución de una función.
- ▶ Retorna el control a su invocador.
- ▶ Devuelve el valor de la expresión como resultado.

La instrucción *return*

- ▶ Termina la ejecución de una función.
- ▶ Retorna el control a su invocador.
- ▶ Devuelve el valor de la expresión como resultado.

```
problema suma2( $x : \mathbb{Z}, y : \mathbb{Z}$ ) :  $\mathbb{Z}$ {  
  asegura:  $res = x + y$   
}
```

```
def suma2 (x: int, y: int) -> int:  
  res: int = x + y  
  return res
```

```
def suma2 (x: int, y: int) -> int:  
  return x + y
```

Transformación de estados

- ▶ Llamamos **estado** de un programa a los valores de todas sus variables en un punto de su ejecución:
 - ▶ Antes de ejecutar la primera instrucción.
 - ▶ Entre dos instrucciones.
 - ▶ Después de ejecutar la última instrucción.
- ▶ Veremos la ejecución de un programa como una **sucesión de estados**.
- ▶ La asignación es la instrucción que transforma estados.
- ▶ El resto de las instrucciones son de control: modifican el flujo de ejecución es decir, el orden de ejecución de las instrucciones.

Ejemplo de *transformación de estados*

```
def ejemplo() -> int:  
    x: int = 0  
    x = x + 3  
    x = 2 * x  
    return x
```

Ejemplo de transformación de estados:

```
x = 0  
    //Estado 1 x == 0  
x = x + 3  
    //Estado 2 x == 3  
x = 2 * x  
    //Estado 3 x == 6
```

Ejemplo de *transformación de estados*

- Podemos pensar que cada instrucción define un nuevo estado.
- A cada estado se le puede dar un nombre, que representará el conjunto de valores de las variables entre dos instrucciones de un programa.

```
instrucción  
// estado nombre_estado  
otra instrucción
```

Luego de nombrar un estado, podemos referirnos al valor de una variable en dicho estado.

```
nombre_variable@nombre_estado
```


Ejecución simbólica

```
def suc(x: int) -> int:
    //estado a;
    x = x + 2
    //estado b
    //vale x == x@a+2;
    «En el estado b, x vale lo que valía en el estado a más 2»
    x = x - 1
    //estado c
    //vale x == x@b-1;
    «En el estado c, x vale lo que valía en el estado b menos 1»
    return x
```

- ▶ De esta manera, mediante la transformación de estados, podremos realizar una ejecución simbólica del programa, declarando cuánto vale cada variable, en cada estado del programa, en función de los valores anteriores.
- ▶ Algunas técnicas de verificación estática utilizan estos recursos.

Los argumentos de entrada de las funciones

Para indicar que una función recibe argumentos de entrada usamos variables.

Estas variables toman valor cuando el llamador invoca a la función.

En los lenguajes imperativos, en general, existen dos tipos de pasajes de parámetros:

- ▶ **Pasaje por valor:** se crea una copia local de la variable dentro de la función.
- ▶ **Pasaje por referencia:** se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera.

Hay lenguajes de programación imperativa que se toman en serio que los argumentos de entrada son exactamente eso: **de entrada**.

Sin embargo existen otros lenguajes donde es posible escribir programas que reciben un argumento de entrada en una variable, y luego pueden modificar la variable a gusto.

La mayoría manejan ambos conceptos.

Valor & Referencia

Nota: el manejo de memoria no es parte del temario de la materia

- ▶ No está dentro del alcance de la materia hablar sobre temas relacionados a manejo de memoria (temas que son abordados más adelante en la carrera).
- ▶ Veamos un modelo **muy simplificado** para entender la diferencia entre valor y referencia.

Pensemos la memoria como una grilla de 25 posiciones y tres variables x , y y z

| Memoria | | | | | |
|---------|----|---|----|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| A | | | | | |
| B | | | | 5 | |
| C | | | | | |
| D | 25 | | | | |
| E | | | 13 | | |

| Variables | | |
|-----------|-------|------------|
| Nombre | Valor | Referencia |
| x | 5 | B4 |
| y | 25 | D1 |
| z | 13 | E3 |

- ▶ La variable x tiene un valor de 5 y su referencia es B4
- ▶ La variable y tiene un valor de 25 y su referencia es D1
- ▶ La variable z tiene un valor de 13 y su referencia es E3

Pasaje de argumentos en lenguajes de programación

En los lenguajes imperativos, en general, existen dos tipos de pasajes de parámetros:

- ▶ **Pasaje por valor** se crea una copia local de la variable dentro de la función.
- ▶ **Pasaje por referencia** se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera.

Ver ejemplo

Pasaje de argumentos en lenguajes de programación

Pasaje por valor

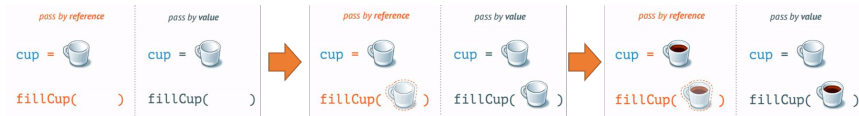
- ▶ Coloca en la posición de memoria del argumento de entrada el *valor* de la expresión usada en la invocación.
- ▶ Se llama también pasaje por **por copia**.
- ▶ La expresión original con la que se realizó la invocación queda protegida contra escritura.



Pasaje de argumentos en lenguajes de programación

Pasaje por referencia

- ▶ La función no recibe un valor sino que implícitamente recibe una dirección de memoria donde encontrar el argumento.
- ▶ La función puede leer esa posición de memoria pero también puede escribirla.
- ▶ Todas las asignaciones hechas dentro del cuerpo de la función cambian el contenido de la memoria. Así, los argumentos por referencia sirven para dar resultados de salida (o de entrada y salida).
- ▶ La expresión con la que se realiza la invocación debe ser necesariamente una *variable*, porque necesita tener asociada una posición de memoria.
- ▶ Es decir, la expresión con la que se realiza la invocación no puede ser una constante, ni una función aplicada.



Programación imperativa

Recapitulando

- ▶ Funciones y Procedimientos: ambos ejecutan un grupo de sentencias.
 - ▶ Funciones: devuelve un valor.
 - ▶ Procedimiento: no devuelve un valor.
- ▶ Conceptualmente, existen tres tipos de pasajes de parametros:
 - ▶ Entrada (**in**): al salir de la función o procedimiento, la variable pasada como parámetro **continuará teniendo su valor original**.
 - ▶ Salida (**out**): al salir de la función o procedimiento, la variable pasada como parámetro **tendrá un nuevo valor asignado dentro de dicha función o procedimiento**. Su valor inicial **no importa ni debería ser leído dentro de la función o procedimiento en cuestión**.
 - ▶ Entrada y salida (**inout**): al salir de la función o procedimiento, la variable pasada como parámetro **tendrá un nuevo valor asignado dentro de dicha función o procedimiento**. Su valor inicial **SI importa dentro de la función o procedimiento en cuestión**.

Veamos como podemos incorporar estas ideas a nuestro lenguaje de especificación.

Especificación de un problema: Extensión

Pasaje de parámetros: in, out e inout - Funciones y Procedimientos

problema *nombre*(*parámetros*) : tipo de dato del resultado (*opcional*) {
 requiere *etiqueta*: { condiciones sobre los parámetros de entrada }
 modifica: *parametros que se modificarán*
 asegura *etiqueta*: { condiciones sobre los parámetros de salida }
 Si *x* es un parametro inout, *x@pre* se refiere al valor que tenía *x* al entrar a la función }

- ▶ *nombre*: nombre que le damos al problema
 - ▶ será resuelto por una función con ese mismo nombre
- ▶ *parámetros*: lista de parámetros separada por comas, donde cada parámetro contiene:
 - ▶ Tipo de pasaje (entrada: **in**, salida: **out**, entrada y salida: **inout**)
 - ▶ Nombre del parámetro
 - ▶ Tipo de datos del parámetro o una **variable de tipo**
- ▶ *tipo de dato del resultado*: tipo de dato del resultado del problema (inicialmente especificaremos funciones) o una **variable de tipo**
 - ▶ En los asegura, podremos referenciar el valor devuelto con el nombre de **res**
- ▶ El tipo de dato del resultado se vuelve opcional, pues ahora podremos especificar programas que no devuelvan resultados, sino que sólo modifiquen sus parámetros.

Pasaje de argumentos en Python

En Python suceden las siguientes situaciones:

- ▶ Conceptualmente, el comportamiento va a depender del tipo de datos de las variables:
 - ▶ Tipo primitivos (int, float, str, tuple, bool, etc): se pasan por valor.
 - ▶ Tipos compuestos y estructuras (listas, etc): se pasan por referencia.
- ▶ Aunque técnicamente, sucede lo siguiente:
 - ▶ Todos los parámetros son por referencia siempre.
 - ▶ Las variables de tipos primitivos, tienen referencias a valores **inmutables** (Ej x: int = 1, la constante 1 nunca cambia... si hacemos x = 3, lo que estamos haciendo es que ahora x referencie al valor 3... pero en IP no profundizaremos).

Ejemplos de pasaje de argumentos en Python

```
def duplicar(valor: str, referencia: list):
    valor *= 2
    print("Dentro de la función duplicar: str: " + valor)
    referencia *= 2
    print("Dentro de la función duplicar: referencia: " + str(referencia))

x: str = "abc"
y: list = ['a', 'b', 'c']
print("ANTES: ")
print("x: " + x)
print("y: " + str(y))
duplicar(x, y)
print("DESPUES: ")
print("x: " + x)
print("y: " + str(y))
```

```
ANTES:
x: abc
y: ['a', 'b', 'c']
Dentro de la función duplicar: str: abcabc
Dentro de la función duplicar: referencia: ['a', 'b', 'c', 'a', 'b', 'c']
DESPUES:
x: abc
y: ['a', 'b', 'c', 'a', 'b', 'c']
```

Ejemplos de pasaje de argumentos en Python

```
problema duplicar(inout x : seq(T)){  
  modifica: x  
  asegura: {x tendrá todos los elementos de x@pre y además, se le  
    concatenará otra copia de x@pre a continuación.}  
  asegura: {x tendrá el doble de longitud que x@pre.}  
}
```

```
def duplicar(x: list):  
    x *= 2
```

Ejemplos de pasaje de argumentos en Python

problema *duplicar*(*in* $x : seq\langle T \rangle$) : $seq\langle T \rangle$ {
 asegura: {*resu* será una copia de x y además, se le concatenará
 otra copia de x a continuación.}
 asegura: {*resu* tendrá el doble de longitud que x .}
}

```
def duplicar(x: list) -> list:  
    y: list = x.copy()  
    y *= 2  
    return y
```

Nota: más adelante veremos el tipo lista, sus operaciones y como recorrer sus elementos.

Ejemplos de pasaje de argumentos en Python

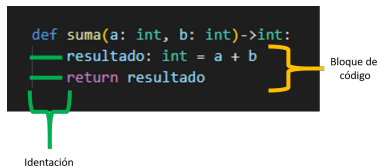
problema *duplicar*(*in* $x : seq\langle T \rangle$, *out* $y : seq\langle T \rangle$){
 asegura: { y será una copia de x y además, se le concatenará
 otra copia de x a continuación.}
 asegura: { y tendrá el doble de longitud que x .}
}

```
def duplicar(x: list, y: list):  
    y.clear()  
    y += x  
    y *= 2
```

Nota: más adelante veremos el tipo lista, sus operaciones y como recorrer sus elementos.

Identación

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



```
def suma(a: int, b: int)->int:
    resultado: int = a + b
    return resultado
```

Identación

Bloque de código

Alcance, ámbito o scope de las variables

- ▶ El alcance de una variable, se refiere al ámbito o espacio donde una variable es reconocida.
- ▶ Una variable sólo será válida dentro del bloque (función/procedimiento) donde fue declarada. Al terminar el bloque, la variable se destruye. Estas variables se denominan **variables locales**.
- ▶ Las variables declaradas fuera de todo bloque son conocidas como **variables globales** y cualquier bloque puede acceder a ella y modificarla.

Variables locales

Un ejemplo con Python

- ▶ `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

```
ejemplo_scope.py > ...  
1 def ejemploLocalScope():  
2     x: int = 19  
3     print("x: " +  
4  
5  
6 ejemploLocalScope()  
7 print("x: " + str(x))
```

"x" is not defined Pylance([reportUndefinedVariable](#))
(function) x: Any
[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)

Variables globales

Un ejemplo con Python

- ▶ `x` 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

Variables locales

Particularidades de Python: bloque del IF

- ▶ Los conceptos de variables locales y globales trascienden a los lenguajes.
- ▶ De hecho, además de otros tipos de scope, habrá scope a nivel de funciones, procedimientos, clases, packages, etc.
- ▶ Y no todos los lenguajes tendrán siempre el mismo comportamiento con respecto a estos temas.

```
#include <iostream>

using namespace std;

int main()
{
    int x = 10;
    if(x > 5){
        int y = 6;
    } else {
        int y = 12;
    }
    cout<<y;

    return 0;
}
```

Y tiene un scope local al bloque del IF (Ejemplo en C++)

```
x: int = 10
if(x > 5):
    y: int = 6
else:
    y: int = 12

print("y = " + str(y))
```

En Python el bloque mínimo es a nivel función. Los bloques dentro de un IF o un WHILE están dentro del anterior.

y = 6

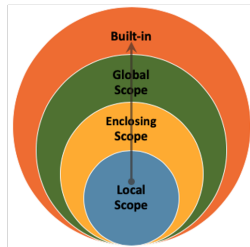
```
main.cpp: In function 'int main()':
main.cpp:21:12: error: 'y' was not declared in this scope
 21 |     cout<<y;
    |         ^
```

Alcance de Variables en Python

Python distingue 4 niveles de visibilidad o alcance

- ▶ Local: corresponde al ámbito de una función.
- ▶ No local o Enclosed: no está en el ámbito local, pero aparece en una función que reside dentro de otra función.
- ▶ Global: declarada en el cuerpo principal del programa, fuera de cualquier función.
- ▶ Integrado o Built-in: son todas las declaraciones propias de Python (por ejemplo: def, print, etc)

```
def outer():  
    enclosed: int = 1  
  
    def inner():  
        local: int = 2  
        print("INNER: variableGlobal declarada fuera de todo: ", variableGlobal)  
        print("INNER: enclosed declarada en outer: ", enclosed)  
        print("INNER: local declarada en inner: ", local)  
  
    inner()  
  
    print("OUTER: variableGlobal declarada fuera de todo: ", variableGlobal)  
    print("OUTER: enclosed declarada en outer: ", enclosed)  
    print("OUTER: local declarada en inner: ", local)  
  
variableGlobal: int = 3  
outer()  
print("GLOBAL: variableGlobal declarada fuera de todo: ", variableGlobal)  
print("GLOBAL: enclosed declarada en outer: ", enclosed)  
print("GLOBAL: local declarada en inner: ", local)
```



Alcance de Variables en Python

Las 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

```
if (B):  
    uno  
else:  
    dos
```

- ▶ B Tiene que ser una expresión booleana. Se llama **guarda**.
- ▶ *uno* y *dos* son bloques de instrucciones.
- ▶ Pensemos el condicional desde la transformación de estados...

Condicionales

Pensemos el condicional desde la transformación de estados

- ▶ Todo el condicional tiene su precondition y su postcondition: P_{if} y Q_{if}
- ▶ Cada bloque de instrucciones, también tiene sus condiciones y postcondiciones.

```
# estado  $P_{if}$   
if (B):  
    # estado  $P_{uno}$   
    uno  
    # estado  $Q_{uno}$   
else:  
    # estado  $P_{dos}$   
    dos  
    # estado  $Q_{dos}$   
# estado  $Q_{if}$   
#  $(B \wedge \text{estado } Q_{uno}) \vee (\neg B \wedge \text{estado } Q_{dos})$   
# Después del IF, se cumplió B y  $Q_{uno}$  o, no se cumplió B y  $Q_{dos}$ 
```

Condicionales

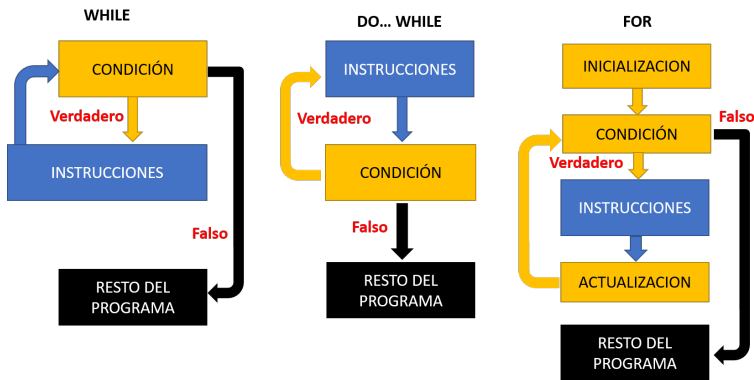
Un ejemplo con Python

```
def elegirMayor(x: int, y: int) -> int:
    z: int
    print("x = " + str(x) + " | y = " + str(y) )
    if x > y :
        print("x es mayor que y")
        z = x
        print("(Se cumple B) -> z toma el valor de x")
    else:
        print("y es mayor o igual que x")
        z = y
        print("(No se cumple B) -> z toma el valor de y")

    return z
```

Ciclos

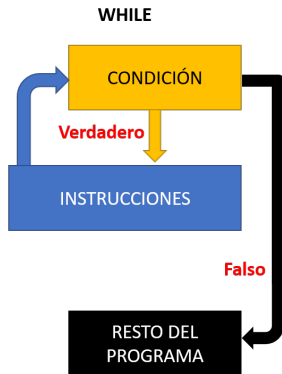
- ▶ En los lenguajes imperativos, existen estructuras de control encargadas de repetir un bloque de código mientras se cumpla una condición.
- ▶ Cada repetición suele llamarse iteración.
- ▶ Existen diferentes esquemas de iteración, los más conocidos son:
 - ▶ While, Do While, For



While en Python

Sintaxis del While

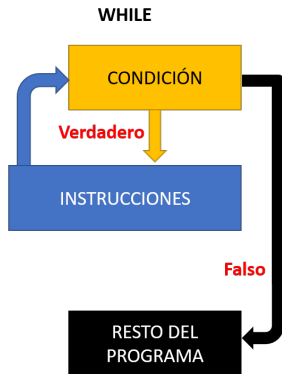
```
while(condición de finalización):  
    ...  
    ...  
    Bloque de Instrucciones...  
    Dentro del while  
    ...  
    ...  
  
Bloque de Instrucciones...  
FUERA del while
```



While en Python

Sintaxis del While

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



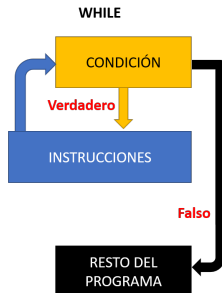
While en Python

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.')
```

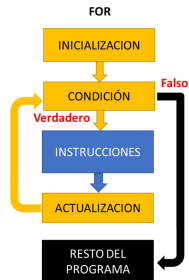


- ▶ `input`: espera que el usuario ingrese algo por teclado.
- ▶ `int(input('...'))`: convierte en `int` lo que el usuario ingresó por teclado.

For en Python

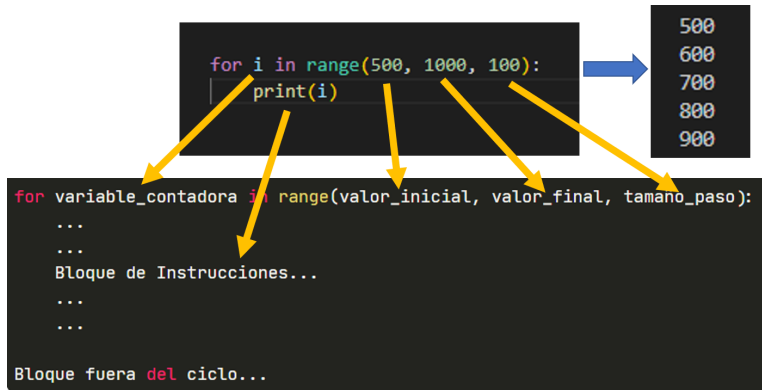
Sintaxis del For

```
for variable_contadora in range(valor_inicial, valor_final, tamaño_paso):  
    ...  
    ...  
    Bloque de Instrucciones...  
    ...  
    ...  
Bloque fuera del ciclo...
```



For en Python

¿Qué hace este programa?



Interrumpiendo ciclos: Break

- ▶ La instrucción Break permite romper la ejecución de un ciclo.
- ▶ Pero tenemos que tener cuidado en el uso de esta instrucción:
 - ▶ Su abuso (tener muchos Break en un mismo ciclo) le puede quitar declaratividad al código.
 - ▶ Desde el punto de vista de analizar la correctitud de un programa, habrá que demostrar que el programa es equivalente a otro que no use break y luego analizar la correctitud de este último programa (pero eso ya lo verán más adelante en la carrera).

```
while(True):  
    numero = int(input('Ingresa un número. 0 para terminar: '))  
    print('Usted ingresó: ', numero)  
    if(numero==0):  
        break  
  
print('Fin del programa.')
```

Ciclos y transformación de estados...

```
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.

Ciclos y transformación de estados...

```
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

Ciclos y transformación de estados...

```
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?

Ciclos y transformación de estados...

```
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.