Estructuras de Control: Condicionales e Iteraciones

.UBAfiuba
FACULTAD DE INGENIERÍA

Fundamentos de Programación / Algoritmos y Programación I
Semana 3 · Clase Teórica · Prof. Servetto

- Estructuras condicionales: if / elif / else
- Expresiones booleanas compuestas y cortocircuito lógico
- Estructuras iterativas: while, for, range()
- Iteración sobre secuencias y patrones de acumulación
- Diseño de casos de prueba: normales, límite y extremos
- Introducción a la complejidad algorítmica y notación O()
- [Holístico] Saltos en lenguaje de máquina · Memoria · SO como gestor

Estructuras Condicionales: if / elif / else

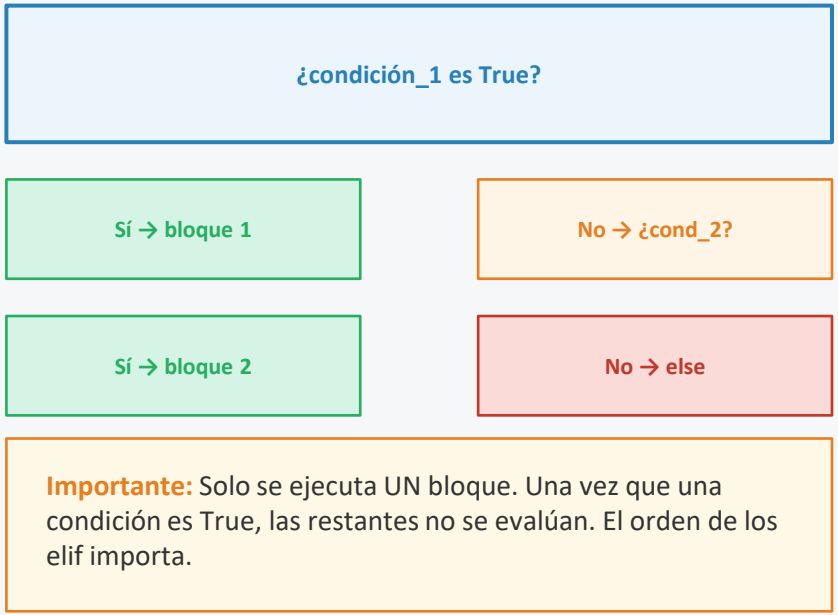
Permiten ejecutar bloques de código diferentes según se cumplan o no determinadas condiciones.

Estructura general

```
if condición:
    bloque_si_verdadero
elif otra_condición:
    bloque_alternativo
else:
    bloque_por_defecto
```

Elemento	Descripción
if	Obligatorio. Evalúa una condición booleana.
elif	Opcional. Puede haber 0 o más. Se evalúan en orden.
else	Opcional. Se ejecuta si ninguna condición fue verdadera.
Indentación	Define los bloques. Python usa 4 espacios por convención.

Flujo de ejecución



Importante: Solo se ejecuta UN bloque. Una vez que una condición es True, las restantes no se evalúan. El orden de los elif importa.

Expresiones Booleanas y Cortocircuito Lógico

Las condiciones compuestas combinan expresiones simples con operadores lógicos. Python evalúa solo lo necesario.

Operadores de comparación y lógicos

Operador	Significado
==, !=	Igualdad / desigualdad
<, >, <=, >=	Comparaciones de orden
and	True si ambos operandos son True
or	True si al menos uno es True
not	Niega el valor de verdad
in	Pertenencia a una secuencia

Cortocircuito lógico (short-circuit)

and
Si el primer operando es False, el segundo NO se evalúa.

or
Si el primer operando es True, el segundo NO se evalúa.

```
# Evitar división por cero
if divisor != 0 and n/divisor > 10:
# Si divisor==0, la división nunca ocurre
```

Valores truthy y falsy en Python

Categoría	Valores
Falsy (equivalen a False)	0, 0.0, '' (cadena vacía), None, [], {}, set(), False
Truthy (equivalen a True)	Cualquier otro valor: 1, -5, 'hola', [1,2], True

Estructura Iterativa: while

Repite un bloque mientras una condición sea verdadera. La condición se evalúa antes de cada iteración.

Sintaxis y semántica

```
while condición:
    # cuerpo del bucle
    # debe modificar la condición
# continúa aquí al salir
```

Patrón	Ejemplo esquemático

Contador	`i = 0`
	`while i < n:`
	`  ...  ;  i += 1`

Centinela	`dato = input()`
	`while dato != 'FIN':`
	`  ...  ;  dato = input()`

Convergencia	`while abs(aprox - val) > tol:` `  ...  # refinar aprox`

Errores frecuentes y buenas prácticas

Bucle infinito:
Si la condición nunca se vuelve False, el programa no termina.

Causas típicas:
· Olvidar actualizar la variable de control
· Condición que nunca se satisface por error lógico
· Comparar floats con == en lugar de tolerancia

Propiedad de finitud (Pólya):
Todo algoritmo debe garantizar terminación.

Verificar: ¿la variable de control se acerca al valor de terminación en cada iteración?

Estructura Iterativa: for y range()

El bucle for recorre los elementos de un iterable. Combinado con range(), permite iteraciones contadas.

Sintaxis del for

```
for variable in iterable:
    # cuerpo: usa 'variable'
# 'variable' conserva su último valor
```

La función range()

Forma	Valores generados	Cantidad
`range(n)`	0, 1, 2, …, n−1	n elementos
`range(a, b)`	a, a+1, …, b−1	b−a elementos
`range(a, b, paso)`	a, a+paso, …	hasta < b (o > b si paso<0)

while vs. for — ¿cuándo usar cada uno?

Criterio	while	for
Nro. de iteraciones	Desconocida a priori	Conocida o determinable
Condición de parada	Expresión booleana	Fin del iterable
Bucle infinito	Posible	No (rango finito)
Ejemplo típico	Validar entrada	Sumatoria, recorrido

```
# Suma de 1 a n con for
suma = 0
for i in range(1, n + 1):
    suma += i
```

Iteración sobre Secuencias

Python permite recorrer directamente cadenas, listas, tuplas y otros iterables con for … in, sin necesidad de índices.

Patrones de iteración sobre secuencias

Patrón	Ejemplo
Recorrido directo	`for c in 'hola':` `  print(c)`
Con índice	`for i in range(len(s)):` `  print(i, s[i])`
enumerate()	`for i, c in enumerate(s):` `  print(i, c)`
Acumulación	`total = 0` `for x in seq: total += x`
Filtrado	`for x in seq:` `  if condición(x): ...`

Secuencias e iterables en Python

Tipo	Descripción	Ejemplo
str	Secuencia de caracteres Unicode	'hola' → 'h','o','l','a'
range	Secuencia perezosa de enteros	range(3) → 0, 1, 2
list	Secuencia mutable ordenada	[10, 20, 30]
tuple	Secuencia inmutable ordenada	(1, 2, 3)

Conexión con complejidad:
Recorrer una secuencia de n elementos con un solo for tiene costo proporcional a n: cada elemento se visita una vez.

Esto se formaliza como O(n) — complejidad lineal.

Patrones Algorítmicos: Acumulación e Iteración

Los patrones de acumulación transforman una secuencia en un valor resumen. Conocerlos facilita diseñar soluciones.

Patrón	Esquema	Ejemplo de uso
Sumatoria	`resultado = 0` `for x in seq: resultado += x`	Suma de notas, total ventas
Productoria	`resultado = 1` `for x in seq: resultado *= x`	Factorial, producto acumulado

Conteo	`cont = 0`
	`for x in seq:`
	`  if cond(x): cont += 1`
		Contar aprobados, pares

Máximo/Mín.	`maximo = seq[0]`
	`for x in seq:`
	`  if x > maximo: maximo = x`
		Nota más alta, temp. extrema

Búsqueda	`encontrado = False`
	`for x in seq:`
	`  if x == buscado:`
	`  encontrado = True; break`
		Verificar existencia

Modelo Declarativo + Algorítmico (Literate Programming): En la sección declarativa, documentar el patrón. En el prólogo, inicializar el acumulador. En la resolución, iterar.

Diseño de Casos de Prueba (Casos de Análisis)

Fase de Análisis de Pólya: antes de codificar, identificar los escenarios que el algoritmo debe manejar.

Categoría	Descripción	Ejemplos
Normal	Valores típicos dentro del dominio esperado	n = 42 → resultado esperable Lista de 10 elementos
Límite (boundary)	Valores en los bordes del dominio válido	n = 0, n = 1, lista vacía [] Primer/último elemento
Extremo	Valores fuera del dominio o excepcionales	n negativo, cadena vacía Valor None, overflow

Ejemplo aplicado: buscar el máximo de una secuencia

Tipo	Entrada	Esperado	Observación
Normal	[3, 7, 2, 9, 1]	9	Caso base, funciona sin desafío
Límite	[5]	5	Un solo elemento
Límite	[4, 4, 4]	4	Todos iguales
Límite	[-3, -1, -7]	-1	Todos negativos
Extremo	[]	Error / indef.	Secuencia vacía: ¿qué devolver?

Introducción a la Complejidad Algorítmica

¿Cómo medir la eficiencia sin depender del hardware? Contando operaciones en función del tamaño de la entrada.

¿Por qué importa la eficiencia?

Dos algoritmos correctos pueden tardar tiempos muy distintos:

Ejemplo: buscar un valor en 1.000.000 de elementos.

Búsqueda lineal: hasta 1.000.000 de comparaciones.
Búsqueda binaria: solo ~20 comparaciones (si está ordenada).

La diferencia crece con el tamaño del problema.
Necesitamos una forma de expresar esta relación.

Conceptos fundamentales

Concepto	Definición
Tamaño de entrada (n)	Dimensión del problema: cantidad de datos, longitud de secuencia.
Operación elemental	Unidad de trabajo: comparación, suma, acceso a memoria.
Peor caso	Máxima cantidad de operaciones para entrada de tamaño n.
Caso promedio	Cantidad esperada bajo distribución típica de entradas.

Principio clave: No medimos tiempo en segundos (depende del hardware). Medimos la tasa de crecimiento del número de operaciones respecto al tamaño de la entrada.

Notación O() — Orden de Complejidad

La notación O (Big-O) describe una cota superior asintótica: cómo crece el costo en el peor caso cuando n → ∞.

Definición intuitiva: f(n) es O(g(n)) si, para entradas suficientemente grandes, f(n) no crece más rápido que g(n) (salvo constante).

Notación	Nombre	Ejemplo algorítmico	Crecimiento
O(1)	Constante	Acceso por índice: lista[3]	No crece con n
O(log n)	Logarítmica	Búsqueda binaria	Crece muy lento
O(n)	Lineal	Recorrer secuencia con for	Proporcional a n
O(n log n)	Linealítmica	Merge sort, sort() de Python	Levemente superlineal
$O(n^2)$	Cuadrática	Dos for anidados sobre n	Crece rápido
$O(2^n)$	Exponencial	Subconjuntos, fuerza bruta	Intratable para n > ~30

Reglas de simplificación

1. Descartar constantes: $3n + 5 \rightarrow O(n)$ **2. Término dominante:** $n^2 + 100n \rightarrow O(n^2)$ **3. Bloques en secuencia suman:** $O(n) + O(n^2) \rightarrow O(n^2)$

Impacto de la Complejidad: Valores Numéricos

Una tabla con operaciones para distintos tamaños de entrada muestra por qué la elección del algoritmo importa.

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(2^n)$
10	1	~3	10	~33	100	1.024
100	1	~7	100	~664	10.000	$\sim 10^{30}$
1.000	1	~10	1.000	~10.000	1.000.000	$\sim 10^{301}$
1.000.000	1	~20	1.000.000	~20.000.000	10^{12}	—

Algoritmos O(n) y O(n log n):
Escalan bien con millones de datos. Son los que usamos para la mayoría de problemas prácticos.

Algoritmos $O(n^2)$ y peores:
Para n=1.000.000, un $O(n^2)$ requiere 10^{12} operaciones (~min a horas). $O(2^n)$ es imposible de ejecutar.

En este curso: trabajaremos con O(n) y $O(n^2)$. En semanas posteriores: O(log n) con búsqueda binaria y O(n log n) con merge sort.

Análisis de Complejidad sobre Iteraciones

Cada estructura iterativa contribuye al costo. Bucles anidados multiplican; en secuencia, suman.

Caso	Esquema	Análisis
Un for de 1 a n	`for i in range(n): op()`	`n ops → O(n)`
Dos for en secuencia	`for i in range(n): ...` `for j in range(n): ...`	`n + n = 2n → O(n)`
Dos for anidados	`for i in range(n):` `  for j in range(n): op()`	`n × n = n² → O(n²)`
For + cuerpo O(n)	`for i in range(n):` `  buscar_lineal(lista)`	`n × n → O(n²)`
For con medio rango	`for i in range(n):` `  for j in range(i, n): ...`	`n(n−1)/2 → O(n²)`

Método para analizar complejidad:

1. Identificar bucles y su rango.
2. ¿Anidados o en secuencia?
3. Multiplicar rangos anidados; sumar secuenciales.
4. Descartar constantes → término dominante.

Ejemplo Integrador: Suma de Dígitos

Dado un entero positivo n, calcular la suma de sus dígitos. Ejemplo: 7345 → 7 + 3 + 4 + 5 = 19.

Solución con while

```
"""
Suma de dígitos de un entero positivo n.
Patrón: acumulación sobre dígitos.
"""
# Prólogo
suma = 0
# Resolución
while n > 0:
    suma += n % 10
    n //= 10
```

Análisis de complejidad

¿Cuántas iteraciones ejecuta el while?

En cada paso, n se divide por 10.

El bucle termina cuando n llega a 0.

Iteraciones = cantidad de dígitos de n

Un número n tiene $\lfloor \log_{10}(n) \rfloor + 1$ dígitos.

Complejidad: O(log n) (base 10)

Traza para n = 7345

n	n % 10	n //= 10	suma
7345	5	734	5
734	4	73	9
73	3	7	12
7	7	0	19

Casos de prueba

Tipo	n	Esperado
Normal	7345	19
Límite	0	0
Límite	5	5 (un dígito)
Normal	999	27

[Holístico] Control de Flujo en Lenguaje de Máquina

Las estructuras if y while de Python se traducen a saltos condicionales e incondicionales a nivel de CPU.

if → CMP + salto condicional

Instrucción	Efecto
`CMP reg1, reg2`	Compara: actualiza flags (Z, S, O, C)
`JE etiqueta`	Salta si iguales (Zero flag = 1)
`JNE etiqueta`	Salta si distintos
`JG / JL`	Salta si mayor / menor (con signo)

while → etiqueta + CMP + JMP

```
inicio_while:
    CMP  i, n          ; comparar
    JGE  fin_while     ; si i>=n, salir
    ...                ; cuerpo
    ADD  i, 1          ; i += 1
    JMP  inicio_while  ; volver
fin_while:
```

Memoria operativa vs. almacenamiento

Concepto	Descripción
RAM (operativa)	Volátil, rápida. El programa reside aquí durante la ejecución.
Disco (almacenam.)	Persistente, más lento. Archivos y programas guardados.
Ciclo de ejecución	SO carga programa de disco a RAM. CPU lee y ejecuta.

SO como gestor de recursos

Función del SO	Descripción
Procesos	Múltiples programas comparten CPU (time-sharing).
Memoria	Asigna RAM a cada proceso y protege su espacio.
E/S	Media acceso a dispositivos: teclado, pantalla, disco.

Buenas Prácticas y Conexiones Conceptuales

Buenas prácticas con estructuras de control

Práctica	Descripción
Elegir estructura adecuada	for cuando se conoce rango; while cuando depende de condición.
Evitar bucles infinitos	Verificar convergencia de la variable de control.
Documentar invariante	Qué se mantiene verdadero en cada iteración (Literate Prog.).
Casos de prueba primero	Antes de codificar: normal, límite, extremo (Pólya).
Evaluar complejidad	Estimar O() para evitar soluciones que no escalen.

[Holístico] Conexiones entre niveles

Python	Máquina / Sistema
if / elif / else	CMP + saltos condicionales (JE, JNE, JG, JL)
while / for	Etiqueta + CMP + JMP (salto incondicional)
Variables en RAM	Direcciones gestionadas por el SO
Ejecución del programa	Fetch → Decode → Execute
Complejidad O()	Cantidad de ciclos F-D-E de la CPU

Literate Programming aplicado

```
# suma_digitos: int → int, O(log n)
# Patrón: acumulación. Invariante: suma contiene la suma de dígitos ya procesados.
# Casos: n=0 → 0; n=5 → 5; n=7345 → 19; n=999 → 27
```

Síntesis de la Clase

Tema	Idea central
Condicionales	if / elif / else bifurcan el flujo. Solo se ejecuta un bloque. Orden de elif importa.
Expr. booleanas	and, or, not con cortocircuito. Valores truthy y falsy en Python.
while	Itera mientras condición sea True. Requiere convergencia → finitud.
for / range()	Recorrido contado y sobre secuencias. Rango finito → no hay bucle infinito.
Iteración secuencias	Patrones: acumulación, conteo, máximo/mínimo, búsqueda, filtrado.
Casos de prueba	Diseñar antes de codificar: normales, límite y extremos (Pólya).
Complejidad O()	Mide crecimiento del costo. $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$.
[Holístico]	if/while → saltos cond./incond. RAM vs. disco. SO: procesos, memoria, E/S.

Actividades de extracursado: lectura 'Procesamiento de Secuencias y Condiciones Lógicas' · 'Funciones del Sistema Operativo' · Curso virtual: problemas de procesamiento de secuencia y foro sobre casos de prueba