Programador de Videojuegos

CLASE 16/07/25

Python

¿Qué es Python?

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general. Se destaca por su sintaxis clara y sencilla, lo que lo convierte en una excelente opción para quienes comienzan en el mundo de la programación.

¿Por qué usar Python en videojuegos?

- Sintaxis simple, clara y sencilla.
- Tiene bibliotecas especializadas como Pygame y Arcade para desarrollo de juegos 2D.
- Ideal para prototipos rápidos.
- Usado por estudios indie y en educación.
- Es multiplataforma, es decir, el intérprete de Python está disponible en multitud de plataformas por lo cual, si no utilizamos librerías específicas de cada plataforma nuestro programa podrá correr en todos los sistemas sin grandes cambios.

Conceptos previos

- **¿Qué es un algoritmo?**

Un algoritmo es un conjunto ordenado de pasos o instrucciones que se siguen para resolver un problema o realizar una tarea.
En programación, un algoritmo indica cómo resolver un problema de forma automática, paso a paso.

- **¿Qué es un dato?**

Un dato es una unidad mínima de información que puede ser manipulada por un programa. Es decir, es una representación de la realidad que se puede guardar y procesar en un sistema informático. Por ejemplo, puede ser un número, una palabra, una fecha, un conjunto de caracteres o cualquier otra cosa.

Tipo de dato

¿Qué es un tipo de dato?

En programación, se refiere al tipo de valor que puede ser almacenado en una variable. Los tipos de datos determinan la naturaleza de la información que manipulamos: números, texto, listas, verdadero/falso, etc. Cada dato en Python tiene un tipo, y este tipo define qué operaciones se pueden hacer con él.

Los tipos básicos se dividen en:

Tipo de dato	Nombre	Descripción	Ejemplo
`int`	Entero	Se usan para representar valores numéricos sin decimales.	`100` `-5`
`float`	Números con decimal	Se usan para representar valores numéricos de que sean enteros y/o con punto decimal (coma flotante).	`3.14` `0.5`
`str`	Texto	Se usan para representar cadenas de caracteres que contengan tanto valores numéricos, letras, palabras, decimales, símbolos, combinaciones alfanuméricas, etc.	`"Hola"`, `"Jugador"`
`bool`	Booleano/Lógico	Solo puede tomar dos valores: Verdadero (1) o Falso (0)	`True`, `False`

Variable

¿Qué es una variable?
Una variable es un contenedor que se utiliza para almacenar un valor o conjunto de valores. Es decir, son espacios reservados en memoria (RAM) para almacenar datos cuyo valor puede cambiar durante la ejecución del algoritmo, pero nunca varía su nombre y su tipo.El espacio reservado depende del tipo de la variable.

En Python, a diferencia de muchos otros lenguajes, no se declara el tipo de la variable al crearla. Para crearla simplemente:

nombreVariable = valor

Ejemplo:

```
puntos = 10
tiempo = 0.75
mensaje = "¡Bienvenido!" #El texto va siempre entre comillas simples o dobles.
vivo = true
enemigo_visible = false
```

¿Cómo saber el tipo de un dato?
Podés usar la función type() para averiguar/comprobar el tipo de dato de una variable.
Ejemplo:

```
vida = 100
print(type(vida))  # <class 'int'>
```

Variable

None es un tipo especial de valor en Python que representa la ausencia de valor o la nada. Es el único valor del tipo de datos NoneType.
Se usa cuando:

- Querés declarar una variable de momento sin valor.
- Una función no devuelve nada.
- Querés indicar que algo está vacío o desconocido.
- Querés resetear una variable.

<u>Sintaxis y ejemplos:</u>
Asignar None a una variable:

```
puntos = None
```

Verificar si algo es None:

```
if puntos is None:
    print("No tenés puntos.")
```

Importante: No se compara con ==, se usa is o is not.

Operador	¿Qué evalúa?	¿Cuándo usarlo?
==	Si dos valores son iguales	Comparar datos (números, textos, listas)
is	Si son el **mismo objeto**	Comparar identidad o con None
is not	Si **no son el mismo objeto**	Inverso de is, útil con None

Instrucciones

1) Función: print()

Se utiliza para mostrar un mensaje por pantalla.

Sintaxis: print(valor)

Donde valor puede ser un texto entre comillas, una variable, una operación, etc.

Ejemplo:

```
print("¡Bienvenido al juego!")
```

➔ Esta instrucción muestra el texto "¡Bienvenido al juego!" en pantalla.

2) Función: input()

Utilizada para pedirle un dato al usuario y almacenarlo. Es decir, se usa para leer lo que escribe el usuario desde el teclado.
Siempre devuelve un texto (tipo str).

Sintaxis:

```
variable = input("Mensaje que aparece en pantalla:")
```

➔ Esta función espera que el usuario escriba algo y presione Enter. Lo que el usuario escriba, se guarda en la variable.

Ejemplo:

```
nombre = input("Ingrese nombre de usuario: ")
```

Instrucciones

Podemos utilizar `input()` y `print()` para interactuar con el usuario.

Ejemplo:

```
nombre = input("Ingrese nombre de usuario: ")

print("¡Hola,", nombre + "!")
```

- ➔ `input()` pide el dato.
- ➔ Se almacena lo ingresado por el usuario en la variable `nombre`.
- ➔ `print()` muestra el saludo.
- ➔ El símbolo + se usa para unir textos (concatenar).

Instrucciones

3) Leer números del usuario

Todo lo que se ingresa con `input()` es texto (`str`), incluso si escribís un número. Por eso, si querés trabajar con números (por ejemplo, sumar o restar), tenés que **convertir el dato**.

Para convertir:

- A **entero**: `int(...)`
- A **decimal**: `float(...)`

a. Leer un número entero

Función: int(input(...))

Sintaxis:

```
nombreVariable = int(input("Ingrese un número entero: "))
```

Ejemplo:

```
edad = int(input("¿Cuántos años tenés?: "))

print("En 10 años tendrás:", edad + 10)
```

b. Leer un número decimal

Función: float(input(...))
Sintaxis:

```
nombreVariable = float(input("Ingrese un número decimal: "))
```

Ejemplo:

```
altura = float(input("¿Cuánto medís en metros?: "))
print("Tu altura es:", altura, "m")
```

Instrucciones

4) f-string

Un f-string (o formatted string literal) es una forma moderna y simple de insertar variables dentro de un texto en Python. Fue introducida en Python 3.6 y su nombre viene de que se usa una letra f antes de la cadena de texto.

```
nombre = "Ana"

print(f"Hola, {nombre}")
```

➔ Reemplaza {nombre} por el contenido de la variable nombre. Resultado: Hola, Ana

¿Para qué sirven?

- Para mostrar mensajes más claros con variables dentro.
- Para formatear números, decimales, expresiones, resultados, etc.
- Para que el código sea más legible y fácil de mantener.

Operadores

¿Qué es un operador?

Un operador es un símbolo que le dice a Python que realice una operación entre uno o más valores (llamados operandos).

Por ejemplo, en 5 + 3, el operador + suma dos operandos: 5 y 3.

Tipo de operador	Ejemplo	¿Para qué sirve?
Aritméticos	`+`, `-`, `*`, `/`	Realizar operaciones matemáticas
De asignación	`=`, `+=`, `-=`	Asignar y actualizar valores
Comparación (relacionales)	`==`, `!=`, `>`, `<`	Comparar valores
Lógicos	`and`, `or`, `not`	Combinar condiciones booleanas

Operadores

1) Operadores Aritméticos: Se usan para hacer operaciones matemáticas.

Operador	Significado	Ejemplo	Resultado
+	Suma	5 + 3	8
-	Resta	10 - 4	6
*	Multiplicación	2 * 5	10
/	División	8 / 2	4.0
//	División entera	9 // 2	4
%	Módulo (resto)	9 % 2	1
**	Potencia	2 ** 3	8

Operadores

2) Operadores de asignación: Asignan o actualizan el valor de una variable.

Operador	Ejemplo	Significado
=	x = 5	Asignación simple
+=	x += 1	Suma y asigna (x = x + 1)
-=	x -= 2	Resta y asigna
*=	x *= 3	Multiplica y asigna
/=	x /= 2	Divide y asigna

Operadores

3) Operadores de Comparación: devuelven True o False.

Operador	Significado	Ejemplo	Resultado
==	Igual a	5 == 5	True
!=	Distinto de	5 != 3	True
>	Mayor que	7 > 4	True
<	Menor que	2 < 1	False
>=	Mayor o igual que	6 >= 6	True
<=	Menor o igual que	4 <= 3	False

Operadores

4) Operadores Lógicos: Usados para combinar condiciones lógicas.

Operador	Significado	Explicación	Ejemplo	Resultado
and	Y	Devuelve True solo si ambas condiciones son verdaderas.	True and False	False
or	O	Devuelve True si al menos una condición es verdadera.	True or False	True
not	No (niega)	Invierte el valor lógico de la condición.	not True	False

Operadores

AND

➔ Devuelve True solo si ambas condiciones son verdaderas.

A	B	A and B
V	V	V
V	F	F
F	V	F
F	F	F

Sintaxis: condición1 **and** condición2

Ejemplo:

```
vida = 50

escudo = 0

if vida > 0 and escudo == 0:

    print("¡Cuidado! Estás sin defensa.")
```

Operadores

OR

➔ Devuelve True si al menos una condición es verdadera.

A	B	A or B
V	V	V
V	F	V
F	V	V
F	F	F

Sintaxis: condición1 **or** condición2

Ejemplo:

```
arma = "espada"

pocion = None

if arma == "espada" or pocion is not None:

    print("Tienes cómo defenderte o curarte.")
```

Operadores

NOT

➔ Invierte el valor lógico de la condición.

A	not A
V	F
F	V

Sintaxis: **not** condición2

Ejemplo:

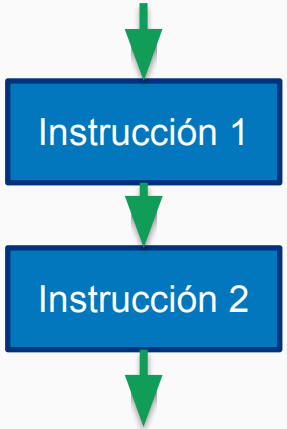
```
jugando = False

if not jugando:

    print("El juego está en pausa.")
```

Estructuras de selección

Estructuras

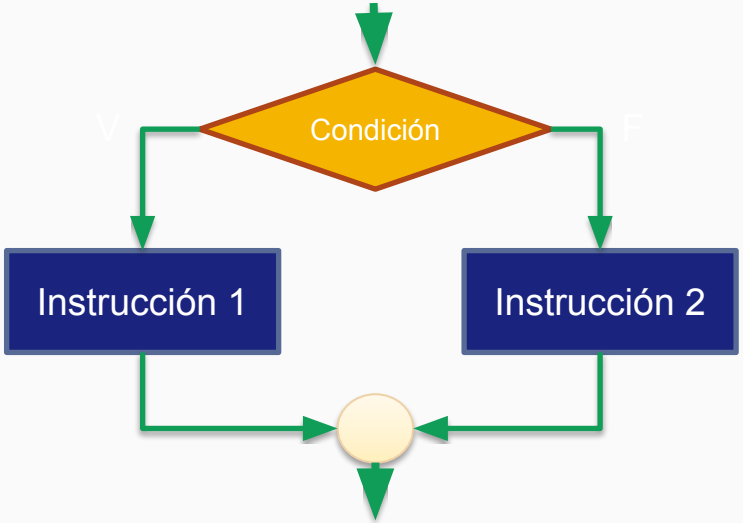
Secuencial

Las sentencias se ejecutan en el orden en el que aparecen en el programa, es decir, una detrás de la otra.



Selección / Condicional

Permite ejecutar una sentencia o conjunto de sentencias, es decir tomará un camino u otro, dependiendo del valor de la condición.



Una condición es una expresión lógica, solo puede devolver dos valores (VERDADERO o FALSO).
Las expresiones relacionales, también denominadas condiciones simples, se utilizan para comparar operandos.
Se pueden construir condiciones complejas utilizando expresiones relacionales mediante los operadores lógicos.

Estructuras de selección: IF

- **if**

Ejecuta una determinada acción o un conjunto de acciones sólo si la condición es verdadera. Si la condición es falsa no hace nada.

Sintaxis:

```
if condición:
    # bloque de código si la condición es True
```

Ejemplo:

```
vida = 100
if vida > 0:
    print("El jugador está vivo")
```

Estructuras de selección: IF-ELSE

- **if-else**

Permite elegir entre dos opciones o alternativas posibles a ejecutar en base al cumplimiento o no de una determinada condición.

Sintaxis:

```
if condición:
    # si la condición es verdadera
else:
    # si la condición es falsa
```

Ejemplo:

```
vida = 0
if vida > 0:
    print("Jugador vivo")
else:
    print("Game Over")
```

Estructuras de selección: IF-ELIF-ELSE

- **if - elif - else (condicional múltiple)**

Permite evaluar múltiples condiciones.

- Se ejecuta solo un bloque, el primero que sea verdadero.
- Si ninguna se cumple, se ejecuta el else.

Sintaxis:

```
if condición1:

    # se ejecuta si condición1 es verdadera

elif condición2:

    # se ejecuta si condición1 es falsa y condición2 es verdadera

elif condición3:

    # se ejecuta si las anteriores son falsas y esta es verdadera

else:

    # se ejecuta si ninguna condición se cumple
```

Ejemplo:

```
puntos = 65

if puntos >= 90:
    print("Nivel experto")
elif puntos >= 60:
    print("Nivel intermedio")
elif puntos >= 30:
    print("Nivel principiante")
else:
    print("Nivel inicial")
```

Estructuras de selección: IF ANIDADOS

- **if anidados (condiciones dentro de otras)**

Consta de una serie de estructuras if una dentro de otra. Es decir, una estructura if-else puede contener otra estructura if-else y está a su vez puede contener otra, y así sucesivamente.

Sintaxis:

```
if condición1:

    if condición2:

        # se ejecuta si ambas condiciones son verdaderas
```

Ejemplo:

```
vida = 50
armadura = true

if vida > 0:
    if armadura :
        print("El jugador está protegido")
    else:
        print("El jugador está vulnerable")
```

Ejercitación

Ejercicio 1: Estadísticas del jugador

Objetivo: Declarar variables y mostrar información.

Enunciado:

- Crear las siguientes variables:
  - nombre del jugador (tipo texto), se utilizará la función vista para que el usuario ingrese su nombre.
  - vida_inicial (entero), valor por defecto 100
  - poder_ataque (entero)
- Mostrar en pantalla un resumen como este:
  - Jugador: Felipe
  - Vida inicial: 100
  - Poder de ataque: 25

Extras:

- Personalizar el mensaje con f-strings.

Ejercitación

Ejercicio 2: Simulador de ataque

Objetivo: Uso de operadores y variables.

Enunciado:

- Simular un ataque. Definir las variables vida_enemigo y ataque.
- Calcular la nueva vida del enemigo luego del ataque y mostrar:
  - El enemigo recibió 15 de daño.
  - Vida restante del enemigo: 35

Extras:

- Hacer que el usuario ingrese el valor del daño.

Ejercicio 3: Calcular daño total

Enunciado: Un jugador tiene un ataque y un multiplicador de daño. Pedirle al usuario que ingrese:

- El ataque (entero)
- El multiplicador (decimal, por ejemplo 1.5)

Calcula el daño total multiplicando ambos y mostrar el resultado.

Ejercitación

Ejercicio 4: Comprobar nivel para desbloquear habilidad

Enunciado:

El jugador tiene más de 10 puntos para desbloquear una habilidad especial. Pedirle al usuario que ingrese un valor y mostrar el mensaje "Habilidad desbloqueada" si los puntos son mayores o iguales a 10.

Ejercicio 5: Entrada para activar modo difícil

Enunciado: Solicitarle al usuario que escriba "sí" o "no" para activar el modo difícil.

- Si el usuario responde "sí" (con tilde o sin tilde), muestra "Modo difícil activado"
- Si responde "no", muestra "Modo normal activado"
- Para cualquier otra respuesta, muestra "Respuesta inválida"

Ejercicio 6: Clasificación por niveles

Objetivo: Usar if/elif/else.

Enunciado: Solicitarle al usuario que ingrese los puntos obtenidos. Según la cantidad, mostrar:

- >= 100: Nivel "Maestro"
- >= 70: Nivel "Experto"
- >= 40: Nivel "Intermedio"
- < 40: Nivel "Principiante"

Ejercitación

Ejercicio 7: Inventario básico

Objetivo: Usar variables tipo texto, condiciones y entrada de usuario.

Enunciado:

El jugador puede tener una espada, una poción o nada. Pedir al usuario que indique qué opción desea y mostrar mensajes distintos según el objeto seleccionado:

- Si tiene una espada: "Estás listo para luchar"
- Si tiene una poción: "Podés curarte si estás herido"
- Si no tiene nada o escribe algo distinto: "Estás indefenso"

Ejercicio 8: ¿Puedes usar la llave?

Enunciado:

Para abrir una puerta, el jugador debe tener la llave ("sí" o "no") y tener puntos mayores a 50. Solicitar ambos datos y mostrar los siguientes mensajes:

- "Puerta abierta" si cumple ambas condiciones.
- "No puedes abrir la puerta" en caso contrario.

Ejercitación

Ejercicio 7: Elegí el camino

Objetivo: Estructura condicional anidada.

Enunciado:

Pedirle al usuario que elija un camino: "bosque", "cueva" o "montaña".

- Si elige "bosque" → Mostrar "Te enfrentas a un lobo".
- Si elige "cueva" → Mostrar "Hay un tesoro escondido".
- Si elige "montaña" → Mostrar "Un dragón bloquea el paso".
- Si elige otro → Mostrar "Camino no válido".

Estructuras de repetición

Bucle for

Itera (repite) sobre una secuencia (lista, cadena, rango, etc.) una cantidad fija de veces.

Sintaxis:

for variable in secuencia:

 # bloque de código a repetir

Ejemplo 1: Contar del 1 al 5

```
for i in range(1, 6):
    print("Número:", i)
```

En Python, la función range() es una función que devuelve una secuencia de números, que es especialmente útil para contar de una forma eficiente dentro de un bucle for.

range(inicio, fin, paso)

- inicio: El número desde el que empezar (por defecto es 0).
- fin: El número hasta el que contar (el valor final no se incluye).
- paso: El valor en el que se incrementa el contador (por defecto es 1).

Bucle for

Ejemplo 2: Recorrer una lista

```
enemigos = ["zombi", "esqueleto", "dragón"]

for enemigo in enemigos:

    print("¡Apareció un", enemigo + "!")
```

Ejemplo 3: Recorrer texto (caracteres de un string)

```
for letra in "Juego":

    print(letra)
```

Bucle while

Repite un bloque de código mientras una condición sea verdadera.

Sintaxis:

```
while condición:
    # bloque de código a repetir
```

Ejemplo 1: Contador hasta 5

```
contador = 1
while contador <= 5:
    print("Intento:", contador)
    contador += 1
```

¡Cuidado con los bucles infinitos!

Si la condición nunca se vuelve falsa, el bucle nunca termina.

Bucle while

```
productos = [101, 102, 103, 104, 105, 106]  # Lista de productos
producto_buscado = 104  # Producto que queremos encontrar
indice = 0  # Empezamos desde el primer índice
producto_encontrado = False  # Flag para controlar si encontramos el producto

# Iniciamos el bucle while
while indice < len(productos) and not producto_encontrado:  # Mientras no lo hayamos encontrado
    if productos[indice] == producto_buscado:  # Si encontramos el producto
        producto_encontrado = True  # Marcamos que lo encontramos
        print(f"Producto {producto_buscado} encontrado en la posición {indice}")
    else:
        indice += 1  # Si no lo encontramos, incrementamos el índice

# Si no encontramos el producto, mostramos un mensaje
if not producto_encontrado:
    print(f"Producto {producto_buscado} no encontrado.")
```

Instrucciones útiles en bucles

- **break**

Detiene el bucle inmediatamente.

Ejemplo: Muestra del 0 al 4 y luego se corta.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

- **continue**

Salta a la siguiente iteración del bucle.

Ejemplo: Muestra 0, 1, 3, 4 (salta el 2).

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Ejercitación bucles

Ejercicio 1: Mostrar por pantalla los números del 1 al 10. Cada número representa un paso que da el personaje.

```
# Salida esperada:
# Paso 1
# Paso 2
# ...
# Paso 10
```

Ejercicio 2: Mostrar los números del 5 al 1 en orden descendente.
Ejercicio 3: Utilizar un bucle while para avanzar desde la posición 0 hasta llegar a la posición 5. Mostrar en cada posición: "Posición actual: X".
Ejercicio 4: El jugador inicia una partida con 3 vidas. Utilizar un bucle while para restar 1 vida por turno, y mostrarle al jugador cuántas vidas le quedan. Al llegar a 0, mostrar "Game Over".
Ejercicio 5: Pedirle al usuario que escriba una contraseña. Mientras la contraseña no sea correcta (es decir, diferente a la almacenada en memoria) debe intentarlo nuevamente. Cabe mencionar, que solo dispone de tres intentos disponibles. Si ingresa la contraseña correcta, se pide mostrar un mensaje de bienvenida. En caso contrario, informar que es inválida o que ya no dispone de más intentos.
Ejercicio 6: Se pide generar un programa que selecciona un número aleatorio entre 1 y 20 y el jugador tiene 5 intentos para adivinarlo. Informar si el intento fue mayor o menor que el número.
Si acierta, mostrale "¡Ganaste!". Si no, "Perdiste. El número era X".

Estructuras de datos

Estructuras de datos

1. Listas
    a. Definición
    b. ¿Como insertar datos?
    c. ¿Cómo acceder a los datos?
    d. Métodos

Listas

Listas (List)

Una lista es una colección ordenada y modificable de elementos. Puede contener valores repetidos y datos de distintos tipos, incluyendo otros tipos de colecciones.

Características:

- Ordenadas: El orden de los elementos se mantiene.
- Mutables: Los elementos de una lista pueden ser modificados (agregar, eliminar, cambiar elementos).
- Indexación: Los elementos se acceden a través de su índice, que comienza desde 0.

Sintaxis:

```
nombre_variable_de_tipo_lista = [valor1, valor2, … , valorn]
```

Ejemplos:
Lista de números:

```
lista_numerica = [10, 20, 30, 40]
```

También puede contener datos de distintos tipos:

```
lista_mezclada = ["Hola", 3.14, True]
```

Lista de string:

```
frutas = ["manzana", "banana", "cereza"]
frutas.append("mango")  # Agrega mango al final
print(frutas)  # ['manzana', 'banana', 'cereza', 'mango']
```

Listas (List)

¿Cómo insertar valores en una lista?

Las listas son colecciones ordenadas y dinámicas, por lo que es muy común agregar elementos nuevos.

- **Método 1: append()**

La función append() permite agregar un elemento al final de la lista

```
frutas = ["manzana", "pera"]
frutas.append("banana")  # Agrega una fruta
print(frutas)  # ['manzana', 'pera', 'banana']
```

- **Método 2: insert(pos, valor)**

La función insert(pos, valor) permite agrega un elemento en una posición específica. Para lo cual, pasamos como primer parámetro la posición y como segundo parámetro el valor a insertar.

```
frutas.insert(1, "naranja")  # Inserta en la posición 1
print(frutas)  # ['manzana', 'naranja', 'pera', 'banana']
```

Precaución! el índice debe ser entero, sino dará error. Ejemplo incorrecto: frutas.insert("uno", "ciruela")

Para insertar varios valores:

- **Método 3: extend()**

La función extend() permite agregar al final una lista completa.

```
frutas.extend(["kiwi", "uva"])
print(frutas)  # ['manzana', 'naranja', 'pera', 'banana', 'kiwi', 'uva']
```

- **También se puede utilizar el operador +:**

```
nueva_lista = frutas + ["mango", "melón"]
print(nueva_lista)
```

Listas (List)

¿Cómo acceder a los elementos de la lista?
Una lista es una colección ordenada. Se accede a sus elementos por índice (posición).

```
# Teniendo la siguiente lista
nombres = ["A", "B", "C", "D"]
```

- Acceder por índice (empieza en 0):

```
print(nombres[0])  # "A"
print(nombres[2])  # "B"
```

- Índices negativos: Permiten acceder desde el final hacia el principio:

```
print(nombres[-1])  # Último elemento: "D"
print(nombres[-2])  # Penúltimo: "C"
```

- Recorrer una lista con for:

```
for nombre in nombres:
    print(nombre)
```

- Acceder con range e índices:

```
for i in range(len(nombres)):
    print(f"Posición {i}: {nombres[i]}")
```

Error común: IndexError si el índice no existe

Ejemplo: print(nombres[10])

Listas (List)

Métodos más comunes:

Método	Descripción
append(x)	Agrega un elemento al final. Siendo x el valor que se agregará.
insert(i, x)	Inserta en la posición i el valor x
remove(x)	Elimina la primera aparición de x (es decir, el primer valor que coincida con el valor pasado por paréntesis)
pop(i)	Elimina el elemento en la posicion i y lo retorna
sort()	Ordena los elementos de la lista (por defecto de menor a mayor).
reverse()	Invierte el orden de la lista
len(lista)	Devuelve la cantidad de elementos de la lista

Estructuras de datos

1. Listas
2. Arreglos
    a. Definición
    b. ¿Como insertar datos?
    c. ¿Cómo acceder a los datos?
    d. Métodos

Arreglos

Arreglos

En Python, los arreglos reales no son tan comunes como en otros lenguajes. Pero pueden usarse con el módulo array. Los arrays son más rápidos y eficientes, pero sólo pueden contener un mismo tipo de dato (por ejemplo, solo enteros o solo flotantes).
¿Cuándo usar un array?

- Cuando necesitás eficiencia y rendimiento (por ejemplo, miles de datos numéricos).
- Cuando los datos son todos del mismo tipo.

Sintaxis con el módulo array:

```
#Importar el módulo array
import array
nombre_array = array.array("i", [valor1, valor2, ... , valorN])
# "i" es para enteros (int)
```

Tipos de arreglos:

- 'i': entero (int)
- 'f': punto flotante
- 'u': carácter unicode

Ejemplo:

```
import array
vidas = array.array("i", [3, 2, 1])
vidas.append(0)
print(vidas)  # array('i', [3, 2, 1, 0])
```

Método	Descripción
append(x)	Agrega un elemento al final
insert(i, x)	Inserta en la posición i
remove(x)	Elimina la primera aparición de x
pop()	Elimina el último elemento
index(x)	Devuelve el índice de x
reverse()	Invierte el orden

Arreglos

¿Cómo insertar valores en un arreglo?
Los arreglos en Python (usando el módulo array) son estructuras optimizadas para números del mismo tipo.

```
#Teniendo el siguiente array
import array
numeros = array.array("i", [10, 20, 30])  # "i" = enteros
```

- **Método 1: append()** para agregar un elemento al final

```
numeros.append(40)
print(numeros)  # array('i', [10, 20, 30, 40])
```

Precaución! No se puede insertar string en array de enteros. Ejemplo incorrecto: numeros.append("hola")

- **Método 2: insert(pos, valor)** para agregar en una posición específica

```
numeros.insert(1, 15)  # Inserta 15 en la posición 1
print(numeros)  # array('i', [10, 15, 20, 30, 40])
```

- **Insertar varios valores en un array:** No existe un extend() directo en array, pero se puede usar un bucle.

```
valores = [50, 60]
for v in valores:
    numeros.append(v)
print(numeros)  # array('i', [10, 15, 20, 30, 40, 50, 60])
```

Arreglos

¿Cómo acceder a los elementos de un array?
Los arreglos (arrays) en Python se crean con el módulo array. Son similares a las listas pero solo aceptan un tipo de dato (ej. enteros, flotantes).

```
# Teniendo el siguiente array
vidas = array.array("i", [3, 2, 1, 0])
```

- Acceder por índice (empieza en 0):

```
print(vidas[0])  # 3
print(vidas[2])  # 1
```

- Recorrer un arreglo con for:

```
for vida in vidas:
    print(vida)
```

- Acceder con range e índices:

```
for i in range(len(vidas)):
    print(f"En índice {i}: {vidas[i]}")
```

Error común: IndexError si el índice no existe

Ejemplo: print(vidas[5])

Estructuras de datos

1. Listas
2. Arreglos
3. Diccionarios
    a. Definición
    b. ¿Como insertar datos?
    c. ¿Cómo acceder a los datos?
    d. Métodos

Diccionarios

Diccionarios

Es una colección no ordenada (hasta Python 3.6), donde cada elemento está compuesto por una clave (key) y un valor (value).

¿Cuándo usar un diccionario?
Cuando necesitas acceder a datos por nombre/clave, como configuraciones, inventarios, atributos de personajes, etc.

Sintaxis básica:

```
nombre_diccionario = {
        “clave1” :  “valor_clave_1”,
        “clave2” :  “valor_clave_2”,
        …
        “claveN” :  “valor_clave_N”
}
```

Si bien se pueden definir claves como enteros, las claves deberían ser strings o nombres lógicos.

Ejemplo:

```
jugador = {
    "nombre": "Mario",
    "vidas": 3,
    "nivel": 5
}
```

Diccionario

¿Cómo insertar valores en un diccionario?
Un diccionario guarda pares clave: valor, y podés insertar nuevos fácilmente.

- **Método 1: Asignación directa**

```
jugador = {"nombre": "Mario", "vidas": 3}
jugador["puntos"] = 100  # Agrega nueva clave
print(jugador)  # {'nombre': 'Mario', 'vidas': 3, 'puntos': 100}
```

- **Método 2: update()** función que permite agregar o actualizar varias claves. Si la clave existe, actualiza el valor. Si la clave no existe, inserta clave:valor.

```
# Actualiza vidas y agrega puntos
jugador.update({"vidas": 5, "puntos": 100})
print(jugador)
```

Diccionarios

¿Cómo acceder a los elementos de un diccionario?

Un diccionario no se accede por índice, sino por clave (key). Es una colección no ordenada de pares clave: valor.

```
# Teniendo el siguiente diccionario
jugador = {
    "nombre": "Mario",
    "vidas": 3,
    "nivel": 2
}
```

- Acceder a los valores por su clave:

```
print(jugador["nombre"])  # "Mario"
print(jugador["vidas"])   # 3
```

- Método get(), no da error si la clave no existe:

```
print(jugador.get("poder"))        # None (no da error)
print(jugador.get("poder", "Ninguno"))  # "Ninguno"
```

- Recorrer un diccionario:
  - Solo claves:

```
for clave in jugador:
    print(clave, jugador[clave])
```

  - Claves y valores con items():

```
for clave, valor in jugador.items():
    print(f"{clave}: {valor}")
```

Error común: KeyError si la clave no existe
Ejemplo: print(jugador["poder"])

Diccionarios

Método	Descripción	Ejemplo
get(clave) o get(clave, valor_por_defecto=None)	Obtiene el valor sin lanzar error	jugador = {"nombre": "Mario"} print(jugador.get("vidas")) # None print(jugador.get("vidas", 0)) # 0
keys()	Devuelve todas las claves	jugador = {"nombre": "Mario", "vidas": 3} print(jugador.keys()) # dict_keys(['nombre', 'vidas'])
values()	Devuelve todos los valores	print(jugador.values()) # dict_values(['Mario', 3])
items()	Devuelve pares (clave, valor)	for clave, valor in jugador.items(): print(f"{clave} → {valor}")
update({...})	Actualiza el diccionario	
pop(clave)	Elimina una clave y devuelve su valor	jugador = {"nombre": "Mario", "vidas": 3} vidas = jugador.pop("vidas") print(vidas) # 3 print(jugador) # {'nombre': 'Mario'}
copy()	Crea una copia del diccionario	copia = jugador.copy()
clear()	Vacía el diccionario	jugador.clear()

Diccionarios

¿Y si quiero manejar varios elementos, como por ejemplo, varios jugadores usando diccionarios?

- **Lista de diccionarios**

Cada jugador es un diccionario, y los guardás todos dentro de una lista:

```
jugadores = [
    {"nombre": "Mario", "vidas": 3},
    {"nombre": "Luigi", "vidas": 2},
    {"nombre": "Peach", "vidas": 4}
]
```

Para acceder a cada jugador:

```
for jugador in jugadores:
    print(f"{jugador['nombre']} tiene {jugador['vidas']} vidas")
```

- **Añadir jugadores dinámicamente**

Podés construir tu colección de jugadores en tiempo de ejecución.

```
jugadores = []
# Agregar jugador 1
jugador1 = {"nombre": "Toad", "vidas": 3}
jugadores.append(jugador1)
# Agregar otro
jugadores.append({"nombre": "Yoshi", "vidas": 5})
```

Diccionarios

- **Diccionario de jugadores por nombre**

En vez de usar una lista, usás un diccionario principal, donde la clave es el nombre del jugador, y el valor es otro diccionario con sus datos:

```
jugadores = {
    "Mario": {"vidas": 3, "nivel": 1},
    "Luigi": {"vidas": 2, "nivel": 2},
    "Peach": {"vidas": 4, "nivel": 3}
}
```

Acceder a un jugador:

```
print(jugadores["Luigi"]["vidas"])  # 2
```

Recorrer todos:

```
for nombre, datos in jugadores.items():
    print(f"{nombre} está en el nivel {datos['nivel']} con {datos['vidas']} vidas")
```

Resumen

Característica	Lista (list)	Arreglo (array)	Diccionario (dict)
Ordenado	Sí	Sí	(hasta Python 3.6)
Indexado	Por posición	Por posición	Por clave
Tipos mixtos	Sí	Solo un tipo	Clave-valor mixtos
Duplicados	Sí	Sí	Las claves no se repiten
Acceso	Por índice (lista[indice])	Por índice (array[indice])	• Por clave (dict["clave"]) • Usando get("clave", valor_default)
Ideal para…	Listas variadas	Datos numéricos rápidos	Atributos, configuraciones

Ejercitación

EJERCICIO 1 – LISTAS

Consigna 1:

Crear una lista llamada enemigos con los siguientes nombres: "zombi", "orco", "dragón".
Después se pide:

1. Agregar “fantasma" al final de la lista.
2. Insertar "gólem" en la posición 1.
3. Mostrar todos los enemigos uno por uno usando un for.

Consigna 2:

Tenés una lista con enemigos que aparecen en pantalla:

enemigos = ["zombi", "orco", "zombi", "dragón", "orco", "zombi"]

1. Contar cuántas veces aparece cada tipo de enemigo.
2. Mostrar una lista nueva (enemigos_unicos) sin repeticiones.
3. Mostrar cuántos enemigos hay en total.

Ejercitación

EJERCICIO 2 – ARREGLOS (array)

Consigna:

Usando el módulo array, creá un arreglo de enteros llamado puntos con los valores [10, 20, 30].
Se pide:

1. Agregar el número 40 al final.
2. Insertar el número 25 en la posición 2.
3. Mostrar todos los valores con un bucle for

EJERCICIO 2 – ARREGLOS (array) (Nivel intermedio)

Consigna:

Usá el módulo array para simular los puntos de vida recibidos por un personaje en 5 turnos de ataque enemigo.

1. Cargar un arreglo llamado daños con: [10, 5, 15, 0, 20]
2. Restar cada daño al valor de vida del jugador (empieza con 50)
3. Mostrar el valor de vida restante después de cada turno.

Ejercitación

EJERCICIO 3 – DICCIONARIO

Consigna 1:

Crear un diccionario llamado jugador con las siguientes claves y valores:
"nombre": "Mario"
"vidas": 3

Se pide:

1. Agregá una nueva clave "nivel" con valor 1.
2. Usá update() para agregar las claves "puntos": 100 y "arma": "espada".
3. Mostrá todas las claves y valores uno por uno con un for

Consigna 2:

Simular los datos de tres jugadores, con un diccionario de jugadores donde cada clave es el nombre, y el valor es otro diccionario con sus estadísticas.

```
jugadores = {
    "Mario": {"vidas": 3, "puntos": 120},
    "Luigi": {"vidas": 2, "puntos": 150},
    "Peach": {"vidas": 4, "puntos": 90}
}
```

1. Sumar 50 puntos a cada jugador.
2. Si un jugador tiene menos de 3 vidas, incrementarle 1 vida.
3. Mostrar todos los datos actualizados.

Ejercitación - Hogwarts

Contexto:
Estás programando un sistema para administrar estudiantes, hechizos y exámenes en una escuela mágica (como Hogwarts). Los estudiantes aprenden hechizos, rinden pruebas y acumulan puntos.

1ERA PARTE:

1. Crear una lista llamada hechizos_basicos con: "Lumos", "Alohomora", "Wingardium Leviosa"
2. Simular que un estudiante intenta usar un hechizo (ejemplo: hechizo = "Expelliarmus")
3. Si el hechizo está en la lista, mostrar "Hechizo lanzado con éxito"
4. Si no, mostrar "Ese hechizo no está aprendido aún”

2DA PARTE:
Crear un diccionario alumnos con los nombres como claves y su puntaje en hechizos como valores:

```
alumnos = {
    "Harry": 80,
    "Hermione": 95,
    "Ron": 60
}
```

Recorrer el diccionario y:

1. Si el puntaje es mayor a 90, mostrar "Excelente".
2. Si está entre 70 y 90, mostrar "Muy bien".
3. Si es menor a 70, mostrar "Debe practicar más".

Ejercitación - Hogwarts

3ERA PARTE

Consigna:
Crear un array con los puntajes obtenidos por un alumno en sus 5 exámenes mágicos: [70, 85, 60, 90, 75]

Calcular:
1. El puntaje total
2. El promedio
3. Cuántas veces obtuvo más de 80 puntos

4TA PARTE
Simular un sistema de calificaciones mágicas:

Crear un diccionario estudiantes donde cada valor sea otro diccionario con:
"hechizos_aprendidos": lista
"puntos": entero
"examenes": array de notas

Calcular y mostrar:
1. El promedio de exámenes
2. Si aprendió más de 3 hechizos
3. Si tiene más de 250 puntos → mostrar "¡Graduado!”