

```
1
2 Programming 'Language' = {
3
4     Introducción = [Python,
5                     Micropython]
6
7
8
9
10
11 }
12
13
14
```



```
1
2
3 Clase 02 = {
4
5     Presentación = [Les
6                     damos la bienvenida al
7                     curso]
8
9
10
11
12 }
13
14
```



# Clases = {

## Clase 01

Breve historia de Python y su Filosofía. Principios de diseño de Python (PEP 20). Instalación y Configuración de Python y entornos de desarrollo (IDE).



## Clase 02

**Sintaxis Básica y Estructuras de Control. Variables, tipos de datos y operadores. Estructuras de control (if, for, while).**

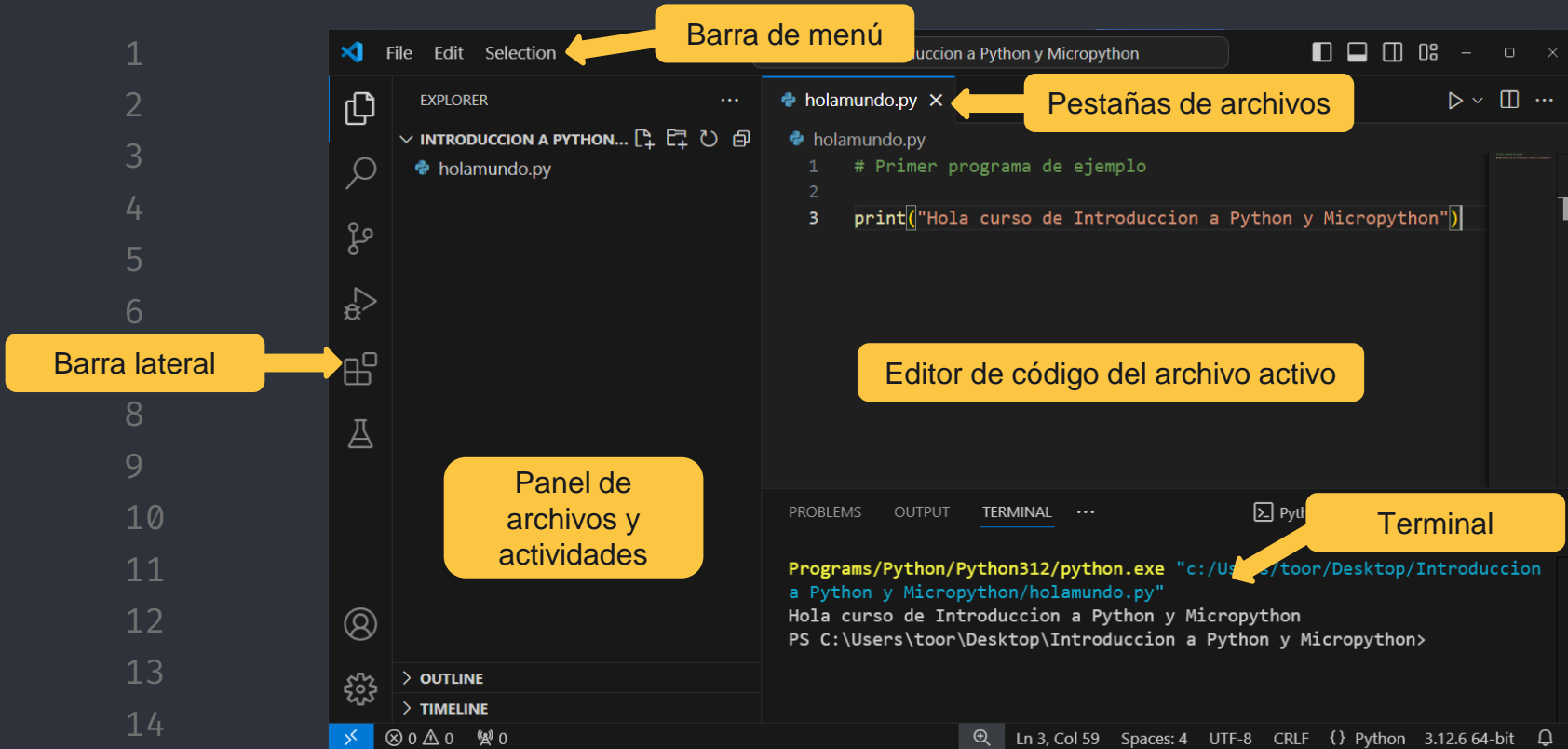
## Clase 03

Estructuras de Datos. Listas, tuplas, diccionarios y conjuntos. Manipulación y métodos asociados.

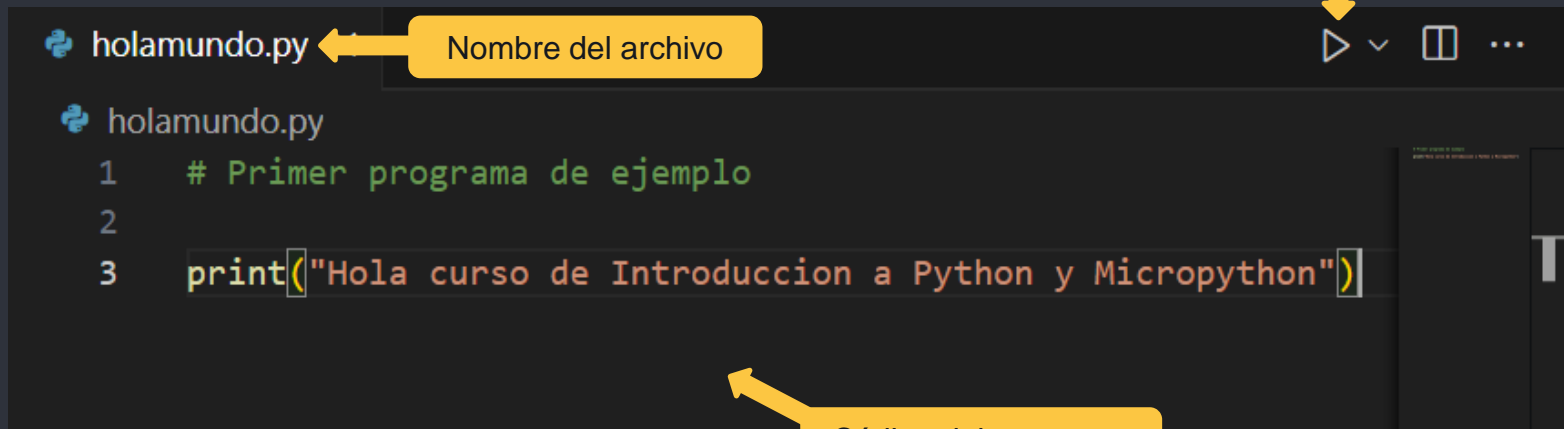
## Clase 04

Funciones y Módulos. Definición y uso de funciones. Importación y creación de módulos.





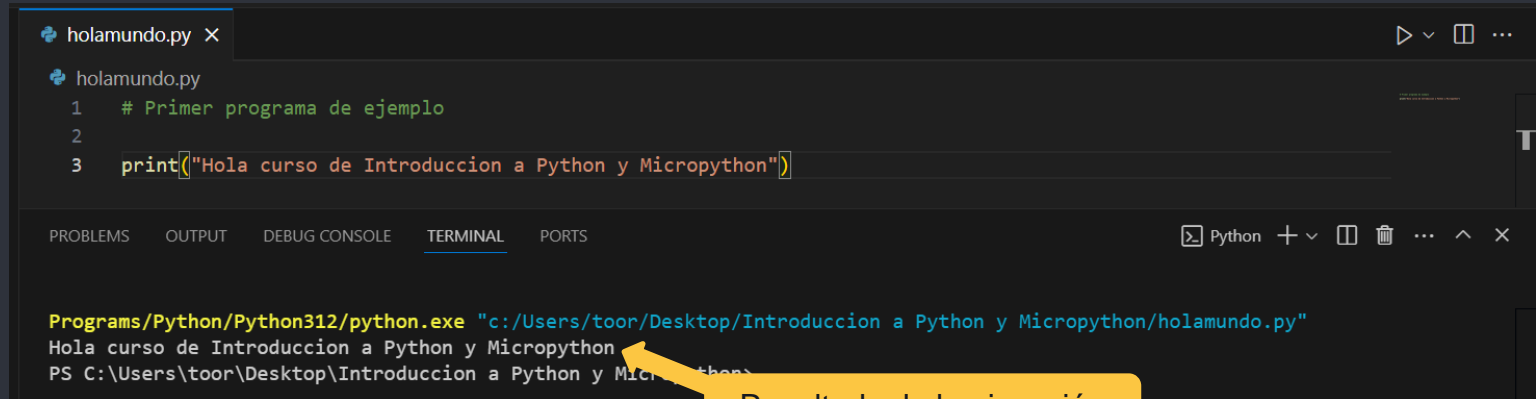
# Ejecutando {



}



# Terminal {



The screenshot shows a code editor with a file named `holamundo.py`. The code contains three lines: a comment, a blank line, and a `print` statement. Below the code, the `TERMINAL` tab is active, showing the command used to run the script and its output. A yellow callout box points to the output text.

```
holamundo.py x
holamundo.py
1 # Primer programa de ejemplo
2
3 print("Hola curso de Introduccion a Python y Micropython")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Python + -

Programs/Python/Python312/python.exe "c:/Users/toor/Desktop/Introduccion a Python y Micropython/holamundo.py"

Hola curso de Introduccion a Python y Micropython

PS C:\Users\toor\Desktop\Introduccion a Python y Micropython>

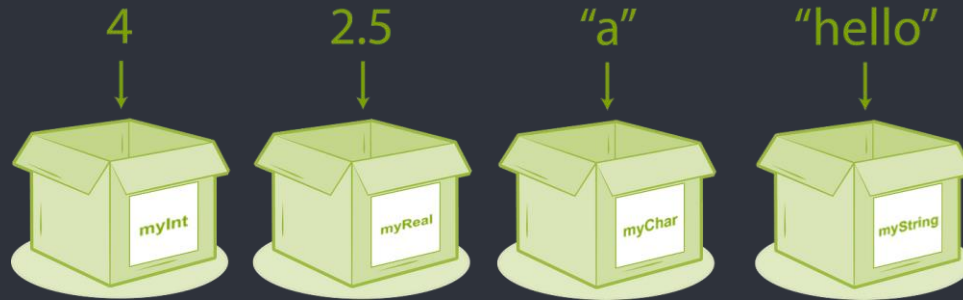
Resultado de la ejecución



# Variables {

# Una variable apunta a un espacio de memoria donde se encuentra un dato. En ellas podemos almacenar y recuperar un dato que utiliza nuestro programa.

# Las variables pueden cambiar su valor a lo largo del desarrollo del programa. En Python es posible que una variable al ser creada contenga un dato de un tipo, y más adelante tenga uno de un tipo diferente.



# Convención de nombres {

# Los nombres de variables, funciones, etc, deben respetar las siguientes convenciones. En Python se llaman identificadores.

# Pueden ser cualquier combinación de letras (mayúsculas y minúsculas), dígitos y el carácter guión bajo (\_), pero no puede comenzar por un dígito.

# No se pueden usar las palabras reservadas.

# Se recomienda usar nombres que sean expresivos. Por ejemplo, contador es mejor que simplemente c.

# Python es “case sensitive”, diferencia entre mayúsculas y minúsculas.

# En Python no existen las constantes. Se suelen declarar variables con la primera letra mayúscula para identificar las que tomaremos como tal.

}





# Palabras reservadas {

```
# Python tiene una serie de palabras reservadas, que se  
utilizan para definir la sintaxis y estructura del  
lenguaje. No pueden usarse como identificadores
```

```
and, as, assert, break, class, continue, def,  
del, elif, else, except, False, finally, for,  
from, global, if, import, in, is, lambda,  
None, nonlocal, not, or, pass, raise, return,  
True, try, yield, while, with
```

```
}
```



# Ejemplo de identificadores {

# Nombres validos y recomendados

suma

area\_total

Importe\_final

\_saldo

anio

area29

# Nombres validos pero no recomendados

Suma

areatriang

ImporteTotal

xd334xsdg4

año

\_\_\_nombre

# Nombres no validos (el interprete devuelve error)

Suma Total

23dias

for

21%imp

\$a\_pagar

while

}



# Tipos de datos {

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14
```

# En Python, un "tipo de dato" se refiere a la categoría o clasificación de los valores que una variable puede almacenar. Cada valor en Python pertenece a un tipo de dato específico, y estos tipos de datos determinan cómo se almacenan los datos en la memoria y qué operaciones se pueden realizar en ellos.

# Los tipos de datos en Python son fundamentales para el manejo de variables y la realización de operaciones en el código.

}

# Tipos de datos {

# Python no requiere especificar el tipo de datos de una variable al declararla.

# Python infiere automáticamente el tipo de datos en tiempo de ejecución según el valor que se le asigna a la variable. Esto proporciona flexibilidad y simplifica la escritura de código.

# Para asignar valores a una variable se debe usar el símbolo igual “=”. Este símbolo es conocido como operador de asignación.

}

# Ejemplo de cambio de tipo {

holamundo.py X

holamundo.py > nombre

```
1 nombre = "Maximiliano"
2 print(nombre)
3 nombre = 23
4 print(nombre)
5 nombre = True
6 print(nombre)
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

```
PS C:\Users\toor\Desktop\Introduccion a Python y Micropython> & C
c:/Users/toor/Desktop/Introduccion a Python y Micropython/holamun
Maximiliano
23
True
PS C:\Users\toor\Desktop\Introduccion a Python y Micropython> |
```

# Tipos de datos simples {

**[ Z ]** int (enteros)

3  
128  
-558

**[ R ]** float (reales)

0,54  
-12,558  
3,1415

**[ A ]** string (cadenas)

Hola  
Juan  
Bienvenidos al curso

**[ X ]** bool (lógicos)

True  
False

}

# Tipos de datos {

Familia	Tipo
Textos	str
Numéricos	int, float, complex
Colecciones	list, tuple, range
Mapeos	dict
Conjuntos	set, frozenset
Booleanos	bool
Binarios	bytes, bytearray, memoryview

# Conversión de tipo de datos {

# En ocasiones es necesario aplicar conversiones de valores entre tipos de datos para manipular los valores de forma diferente. Por ejemplo, para concatenar valores numéricos con cadenas o representar posiciones decimales en números que se iniciaron como valores enteros. Este proceso se llama casting (o casteo españolizado)

```
cadena = "1234"  
entero = 45  
flotante = 56.2
```

```
int(cadena)          # convierte a entero (1234)  
float(cadena)        # convierte a coma flotante (1234.0)  
str(entero)          # convierte a cadena ("45")  
str(flotante)        # convierte a cadena ("56.2")
```

```
}
```



# Obtener el tipo de datos {

```
# Se puede conocer el tipo de dato de cualquier objeto utilizando la
función type():
```

```
    nombre = "Juan"
```

```
    edad = 18
```

```
    notaFinal = 8.50
```

```
    esEstudiante = True
```

```
    domicilio = "Rivadavia 1050"
```

```
    print(type(nombre))           # <class 'str'>, cadena (string)
```

```
    print(type(edad))            # <class 'int'>, entero (integer)
```

```
    print(type(notaFinal))       # <class 'float'>, coma flotante (float)
```

```
    print(type(esEstudiante))    # <class 'bool'>, lógico o booleano (boolean)
```

```
    print(type(domicilio))       # <class 'str'>, cadena (string)
```

```
}
```

# Cadenas de caracteres {

# Python, provee un tipo de datos específico para tratar las cadenas de caracteres (strings).

# Se trata de un tipo de dato con longitud variable, ya que debe adecuarse a la cantidad de caracteres que albergue la cadena. Las cadenas también pueden contener longitud nula, es decir, no tener caracteres en su interior. Las cadenas pueden delimitarse con comillas simples o dobles.

```
dia1 = "Lunes"      # Definición de cadenas usando comillas dobles
x = ""              # x es un string de longitud cero
dia2 = 'Martes'     # Definición de cadenas usando comillas simples
z = '121'            # z contiene dígitos, pero es un string
```

```
print("Mi perro 'Toby'") # Mi perro 'Toby'
print('Mi perro "Toby"') # Mi perro "Toby"
```

}

# Cadenas de caracteres {

# Una cadena puede **replicarse** con el operador **\***.

```
risa = 'ja'
```

```
carcajada = risa*5           # jajajajaja
```

```
asteriscos = "*" * 10       # *****
```

# También pueden usarse triples comillas simples o dobles, que proveen un método sencillo para crear cadenas multilinea usando más de una línea de código.

```
cadena1 = """En Python es posible definir  
cadenas de caracteres utilizando más de una  
línea de código"""
```

```
cadena2 = '''Por supuesto, se puede hacer  
lo mismo utilizando comillas simples'''
```

}

# Cadenas de caracteres {

# Para concatenar dos o más cadenas se utiliza el **operador +** (más).

# Este operador "+" suma números o concatena cadenas. No funciona con tipos mixtos; para evitar errores, usa funciones de conversión:

```
var1 = 3 + 5           # 8 (entero)
var2 = "3" + "5"       # 35 (cadena)
var3 = 3 + "5"         # TypeError
var4 = str(3) + "5"    # 35 (cadena)
var5 = 3 + int("5")    # 8 (entero)
```

}

# Cadenas de caracteres {

# En Python disponemos de la función **len()**, que retorna la cantidad de caracteres que contiene un string:

```
nombre = 'Introduccion a Python y Micropython'
print(len(nombre)) # se imprime 35
```

# Se accede a los elementos de la cadena utilizando subíndices::

```
cadena = "Introduccion a Python y Micropython"
print(cadena[0])      # imprime I
print(cadena[5])      # imprime d
print(cadena[-1])     # imprime n
print(cadena[-2])     # imprime o
```

}

# Cadenas de caracteres {

```
# Las cadenas tienen una serie de métodos (funciones) que simplifican el
desarrollo de código.
```

```
cadena = "Aprendiendo CADENAS en python"
```

```
print(cadena.upper())      # APRENDIENDO CADENAS EN PYTHON
print(cadena.lower())      # aprendiendo cadenas en python
print(cadena.capitalize()) # Aprendiendo cadenas en python
print(cadena.title())      # Aprendiendo Cadenas En Python
print(cadena.center(40,"*")) # *****Aprendiendo CADENAS en python*****
print(cadena.rjust(40,"*")) # *****Aprendiendo CADENAS en python
print(cadena.ljust(40,"*")) # Aprendiendo CADENAS en python*****
```

```
}
```

# Cadenas de caracteres {

```
# f-Strings tiene una sintaxis simple y fluida que simplifica la tarea
de dar formato a cadenas de texto. Para mostrar variables se coloca el
nombre de las variables entre llaves {}, en una cadena que antepone la
letra f a su contenido.
```

```
# Al ejecutar el código, todos los nombres de las variables se
reemplazan por sus respectivos valores.
```

```
legajo = 49588
nombre = "Manuel"
nota = 10
print(f'Legajo: {legajo} Nombre: {nombre} Nota: {nota}')
```

```
# Legajo: 49588 Nombre: Manuel Nota: 10
```

```
}
```

# Cadenas de caracteres {

```
num1 = 10
num2 = 5
resultado = f"La suma de {num1} y {num2} es {num1 + num2}."
print(resultado)

# La suma de 10 y 5 es 15.

x = 3
y = 4
cuadrado_suma = f"La suma de {x} y {y} al cuadrado es { (x + y) ** 2 }."
print(cuadrado_suma)

# La suma de 3 y 4 al cuadrado es 49.
```

}



# Cadenas de caracteres {

# Además podemos controlar el formato. Para darle formato debemos dentro de las llaves luego de la variable colocar dos puntos (:) y acto seguido el formato:

- f para indicar números decimales.
- % para indicar porcentajes, lo que ya multiplicará los valores por 100.
- se puede proceder de una expresión n.m donde n es el número de dígitos y m el número de decimales.

```
numero = 0.123456789
```

```
print(f'Valor con cuatro dígitos: {numero:.4f}')
```

```
# Valor con cuatro dígitos: 0.1235
```

```
print(f'Valor como porcentaje: {numero:.2%}')
```

```
# Valor como porcentaje: 12.35%
```

```
}
```

# Entrada / Salida de datos {

# Ya sabemos usar la función `print()` que permite mostrar datos en la terminal. Recibe entre paréntesis lo que se conoce como parámetros, que pueden ser variables y/o literales. Las cadenas pueden delimitarse con comillas simples o dobles.

```
print("Linea 1")  
print()  
print("Linea 2")
```

```
PS C:\Users\toor\Desktop\Introduccion a Python y Micropython> & C:/Users  
c:/Users/toor/Desktop/Introduccion a Python y Micropython/holamundo.py"  
Linea 1  
  
Linea 2
```

# Entrada / Salida de datos {

# Podemos pasar los parámetros también separados por comas, en lugar de usar f-strings. Al pasar los parámetros con comas, se hace el proceso de concatenación de strings, pero siempre agregando un espacio entre los caracteres.

```
nombre = "Adrián"
```

```
a = 40
```

```
b = 30
```

```
promedio = (a + b) / 2
```

```
print("Mi nombre es", nombre)
```

```
print("La suma de",a,"y",b,"es",a+b)
```

```
print("Promedio:", promedio)
```

```
}
```

Mi nombre es Adrián

La suma de 40 y 30 es 70

Promedio: 35.0

# Entrada / Salida de datos {

# **end=' '** es un parámetro opcional de la función `print()` y determina qué se debe imprimir al final de cada llamada a `print()`. Por defecto es un carácter de salto de línea (`'\n'`), lo que significa que cada llamada a `print()` agrega automáticamente una nueva línea al final.

# Al cambiar `end` a una cadena nula (`' '`), especificamos que nada debe imprimirse al final de cada llamada a `print()`:

```
print("Hola Mundo!", end=' ')  
print("Estamos aprendiendo Python")
```

```
# Hola Mundo! Estamos aprendiendo Python
```

```
}
```

# Entrada / Salida de datos {

# `\n` (salto de línea) representa un carácter especial que se utiliza para insertar una nueva línea en una cadena de texto. Cuando encontramos `\n` dentro de una cadena y la imprimimos, el texto que le sigue se colocado en una nueva línea.

# Si deseamos que no se genere un espacio de más en la siguiente línea no dejamos espacio entre `\n` y el texto que le sigue:

```
print("Estamos \nimprimiendo en \n varias líneas")
```

```
# Estamos
# imprimiendo en
#  varias líneas
```

}

# Entrada / Salida de datos {

# `\t` (tabulación) se utiliza para insertar un tabulador (o tabulación) en una cadena de texto. Cuando Python encuentra `\t` dentro de una cadena a imprimir, inserta un espacio equivalente al de una tabulación.

# La efectividad de `\t` depende de la longitud de las cadenas que lo rodean. La tabulación agrega un número fijo de espacios, y si las cadenas son más largas o más cortas de lo esperado, la alineación puede verse afectada.

```
print("Nombre \tApellido \tEdad")  
print("Juan \tFernández \t32")
```

# Nombre	Apellido	Edad
# Juan	Fernández	32

}

# Entrada / Salida de datos {

# **input()** proporciona un mecanismo para que el usuario introduzca datos en nuestro programa. Muestra el cursor en la terminal, lee lo que se escribe, y cuando se presiona Enter, este contenido, en formato de cadena de caracteres, se puede asignar a una variable.

```
nombre = input("Por favor, ingrese su nombre ")
```

Variable que recibe  
el valor

Función input

Mensaje de ayuda al  
operador

# Entrada / Salida de datos {

```
# Dado que input() devuelve únicamente valores tipo string, es necesario
realizar una conversión a algún formato numérico si se requiere operar
matemáticamente con esos valores. Para ello, usamos las funciones int()
y float():
```

```
num1 = input("Ingrese un número: ")
numero = float(num1)
resultado = numero * 2
print(numero,"x 2 =", resultado)
```

```
Ingrese un número: 75
75.0 x 2 = 150.0
```

```
}
```



# Expresiones y sentencias {

# Una expresión es una unidad de código que devuelve un valor y está formada por una combinación de operandos (variables y literales) y operadores.

5 + 2                      # Suma del número 5 y el número 2

a < 10                    # Compara si el valor de la variable a es menor que 10

b is None                # Compara si la identidad de la variable b es None

3 \* (200 - c)            # Resta a 200 el valor de c y lo multiplica por 3

# Una sentencia o declaración define una acción. Puede contener alguna(s) expresiones. Son las instrucciones que componen el código de un programa y determinan su comportamiento. Finalizan con un Enter.

}

# Expresiones y sentencias {

# Aquellas sentencias que son muy largas pueden ocupar más de una línea (se recomienda una longitud máxima de 72 caracteres). Para dividir una sentencia explícitamente en varias líneas se utiliza el carácter \.

```
a = 2 + 3 + 5 + 7 + 9 + 4 + 6 + 3 + 5 + 1 + 9 + 3 + 2 + 4 + 5
```

```
a = 2 + 3 + 5 + 7 + 9 + 4 + 6 + \  
    3 + 5 + 1 + 9 + 3 + 2 + 4 + 5
```

# Además, en Python la continuación de línea es implícita siempre y cuando la expresión vaya dentro de los caracteres (), [] y {}.

```
a = [1, 2, 7, 3, 1, 4,  
     3, 2, 4, 3, 8 ]
```

}

# Operadores {

# Un operador es un carácter o conjunto de caracteres que actúa sobre una, dos o más variables y/o literales para llevar a cabo una operación con un resultado determinado.

# Ejemplos de operadores comunes son los operadores aritméticos.

# También en Python existen los operadores lógicos y operadores relacionales.

}

# Operador de asignación {

# El **operador de asignación** = (igual) es muy importante en Python. Su función es diferente a la que habitualmente le damos en otros contextos, como la matemática. Se lo denomina “**operador de asignación**” y permite asignar un valor a una variable.

Todo lo que está a la derecha del “=” se asigna a la variable de la izquierda.

```
cantidad = 43      # int (enteros)
precio = 12.45     # float (reales)
nombre = "Adrián"  # string (cadenas)
encendido = True   # bool (lógicos)
```

}

# Operadores aritméticos {

# Realizan operaciones aritméticas. Requieren uno o dos operandos (operadores unarios o binarios). Se aplican las reglas de precedencia.

Operador	Descripción
+	<b>Suma:</b> Suma dos operandos.
-	<b>Resta:</b> Resta al operando de la izquierda el valor del operando de la derecha. Utilizado sobre un único operando, le cambia el signo.
*	<b>Multiplicación:</b> Producto de dos operandos.
/	<b>División:</b> Divide el operando de la izquierda por el de la derecha (el resultado siempre es un float).
%	<b>Operador módulo:</b> Obtiene el resto de dividir el operando de la izquierda por el de la derecha. uno de sus usos es para saber si un número es par o impar
//	<b>División entera:</b> Obtiene el cociente entero de dividir el operando de la izquierda por el de la derecha.
**	<b>Potencia:</b> El resultado es el operando de la izquierda elevado a la potencia del operando de la derecha.

# Operadores aritméticos {

```
suma = 12 + 5
print(suma)           # 17
resta = 23 - 4
print(resta)          # 19
multiplicacion = 5 * 8
print(multiplicacion) # 40
division2 = 14 / 5
print(division2)      # 2.8
modulo = 9 % 2
print(modulo)         # 1
divEntera = 10 // 6
print(divEntera)      # 1
potencia = 2 ** 3
print(potencia)       # 8
```

}

# Operadores relacionales {

# Los operadores relacionales son símbolos que se utilizan para comparar dos valores y determinar la relación entre ellos. Estos operadores devuelven un valor booleano (True o False) que indica si la relación es verdadera o falsa. Los operandos pueden ser variables, constantes o expresiones aritméticas.

Operador	Descripción
<b>==</b>	<b>Igual a:</b> compara si dos valores son iguales.
<b>!=</b>	<b>Diferente de:</b> compara si dos valores no son iguales.
<b>&lt;</b>	<b>Menor que:</b> verifica si el valor de la izquierda es menor que el de la derecha.
<b>&gt;</b>	<b>Mayor que:</b> verifica si el valor de la izquierda es mayor que el de la derecha.
<b>&lt;=</b>	<b>Menor o igual que:</b> verifica si el valor de la izquierda es menor o igual al de la derecha.
<b>&gt;=</b>	<b>Mayor o igual que:</b> verifica si el valor de la izquierda es mayor o igual al de la derecha.

}

# Operadores relacionales {

```
print(7 == 7)      # True
print(9 == 10)     # False
print(7 != 7)      # False
print(9 != 10)     # True
print(7 < 7)       # False
print(9 < 10)      # True
print(8 > 6)       # True
print(5 > 5)       # False
print(8 <= 6)      # False
print(5 <= 5)      # True
print(8 >= 8)      # True
print(4 >= 5)      # False
```

```
}
```



# Operadores lógicos {

# Se utiliza un operador lógico para tomar una decisión basada en múltiples condiciones. Los operadores lógicos utilizados en Python son and, or y not.

Operador	Descripción	Uso
<b>and</b>	Devuelve True si <b>ambos</b> operandos son True	a and b
<b>or</b>	Devuelve True si <b>alguno</b> de los operandos es True	a or b
<b>not</b>	Devuelve True si el operandos False, y viceversa	not a

# a y b son expresiones lógicas. Cada una de ellas puede ser verdadera o falsa. Si a y/o b son valores numéricos, se tratan como True o False según su valor sea cero o no.

}

# Oper. de asignación compuestos {

# Además del operador de asignación, existen los operadores de asignación compuestos. Realizan la operación indicada sobre la misma variable y es una forma abreviada de usar un operador aritmético mas una asignación clásica.

Operador	Ejemplo	Equivalencia
<b>+=</b>	x += 2	x = x + 2
<b>-=</b>	x -= 2	x = x - 2
<b>*=</b>	x *= 2	x = x * 2
<b>/=</b>	x /= 2	x = x / 2
<b>%=</b>	x %= 2	x = x % 2
<b>//=</b>	x //= 2	x = x // 2
<b>**=</b>	x **= 2	x = x ** 2

# Oper. de asignación compuestos {

# Por ejemplo, `x += 1` equivale a `x = x + 1`. Los operadores compuestos realizan la operación indicada antes del signo igual, tomando como operandos la propia variable y el valor a la derecha del signo igual. Y el resultado se guarda en la variable.

```
x = 14      # valor de x es 14
```

```
x = x + 5   # x vale 14, se le suma 5 y se le asigna a x el nuevo valor 19
```

```
print(x)    # 19
```

```
}
```

# Operadores de pertenencia {

# Los operadores de pertenencia se utilizan para comprobar si un caracter o cadena se encuentran dentro de otra.

Operador	Descripción
<b>in</b>	Devuelve <b>True</b> si el valor se encuentra en una secuencia; <b>False</b> en caso contrario.
<b>not in</b>	Devuelve <b>True</b> si el valor no se encuentra en una secuencia; <b>False</b> en caso contrario.

```
cadena = "Introduccion a Python y Micropython"
print("C" in cadena) # False
print("n" in cadena) # True
print("maxi" in cadena) # False
print("Python" in cadena) # True
print("A" not in cadena) # True
print("o" not in cadena) # False
```

}

# Comentarios {

```
# Los comentarios de una línea, comienzan siempre con #. Todo lo que  
# siga después del símbolo # no se interpreta.
```

```
# Esto es un comentario de una línea
```

```
# Los comentarios multilínea, se rodean en el principio y en el final  
# con triple comilla simple o doble.
```

```
"""Esto es un comentario de muchas líneas.  
El cual debe estar encerrado entre triple  
comillas dobles o comillas simples"""
```

```
}
```

# Bloques de código e indentación {

```
1  
2  
3 # El código puede agruparse en bloques, que delimitan sentencias  
4 relacionadas. Python, utiliza la indentación o sangrado, que consiste en  
5 mover el bloque de código hacia la derecha insertando espacios o  
6 tabuladores al principio de la línea, dejando un margen a su izquierda.
```

```
7  
8 # Un bloque comienza con un nuevo sangrado y acaba con la primera línea  
9 cuyo sangrado sea menor. La guía de estilo de Python recomienda usar  
10 espacios en lugar de tabulaciones. Para realizar el sangrado, se suelen  
11 utilizar 4 espacios.
```

```
12  
13 # No todos los lenguajes de programación necesitan de una indentación,  
14 aunque sí se estila implementarla a fin de otorgar mayor legibilidad al  
código fuente. Pero en el caso de Python, la indentación es obligatoria,  
ya que de ella dependerá su estructura.
```

```
}
```

# Bloques de código e indentación {

```
def suma_numeros(numeros): # Bloque 1
    suma = 0                # Bloque 2
    for n in numeros:       # Bloque 2
        suma += n           # Bloque 3
        print(suma)         # Bloque 3
    return suma             # Bloque 2
print()
```

# Si bien aún no sabemos exactamente qué hace el código anterior, se pueden ver los bloques de instrucciones indicados mediante los tabuladores sobre el margen izquierdo. Es posible incluir un bloque dentro de otro, para crear estructuras complejas

}

# Estructuras de control {

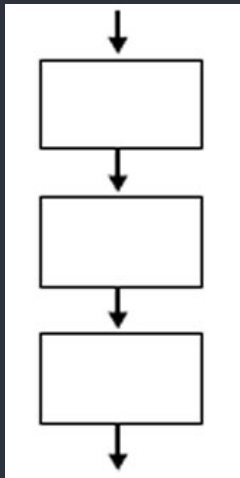
# En programación, las estructuras de control permiten modificar el flujo de ejecución de las instrucciones de un programa. Con ellas se puede ejecutar un grupo u otro de sentencias, según se cumpla o no una condición (if).

# También existen estructuras de control que permiten ejecutar un grupo de sentencias mientras se cumpla una condición (while) o repetir un grupo de sentencias, que veremos más adelante, un número determinado de veces (for).

}



# Estructuras de control {



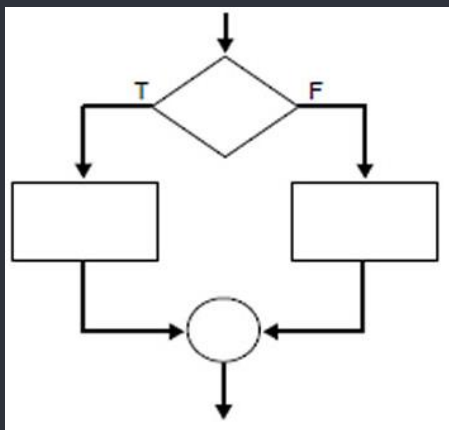
Secuencial

< Las acciones se llevan a cabo en sucesión, cada instrucción se ejecuta de manera consecutiva, y el flujo de control se desplaza de una acción a la siguiente en orden lineal, de arriba hacia abajo. Todas las instrucciones se procesan una única vez. >

```
#Programa Suma: suma dos números
nro1 = int(input("Ingrese el primer número: "))
nro2 = int(input("Ingrese el segundo número: "))
suma = nro1 + nro2
print("La suma es:", suma)
```

< El código del ejemplo pide un número, luego pide otro, realiza la suma de ambos valores guardando el resultado en suma y finalmente muestra un mensaje por pantalla >

# Estructuras de control {



Condicional

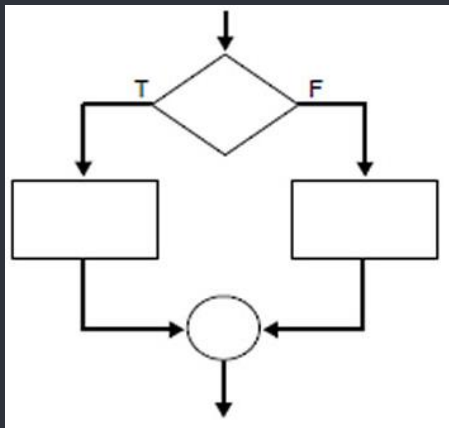
< Las estructuras condicionales tienen como objetivo ejecutar un bloque de instrucciones u otro en base a una condición que puede ser verdadera o falsa. La palabra clave asociada a esta estructura es **if** >

< Si la condición es **True** se ejecuta el bloque dentro del if. Luego, independientemente del valor de verdad de la condición, el programa continúa con la ejecución del resto del programa >

```
nota = float(input("Ingrese la calificación: "))
```

```
if nota >= 7:
    print("Aprobado.")
```

# Estructuras de control {



Condicional

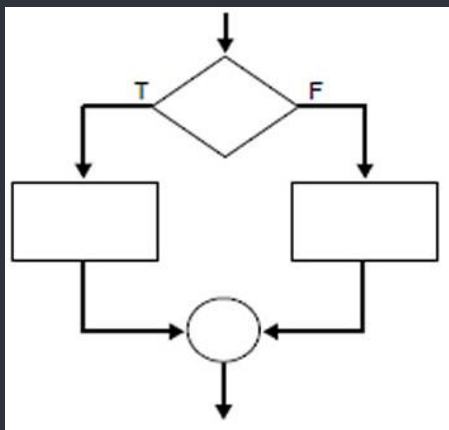
< En Python podemos utilizar la cláusula **else** para determinar un grupo de instrucciones que se ejecutará en caso de que la evaluación de la condición resulte ser falsa >

< Con este agregado, una estructura if tiene la posibilidad de ejecutar un bloque de instrucciones u otro, dependiendo de si la condición es verdadera o falsa>

```
edad = float(input("Ingrese la edad: "))
```

```
if edad >= 18:
    print("Puedes pasar.")
else:
    print("No admitido.")
```

# Estructuras de control {



Condicional

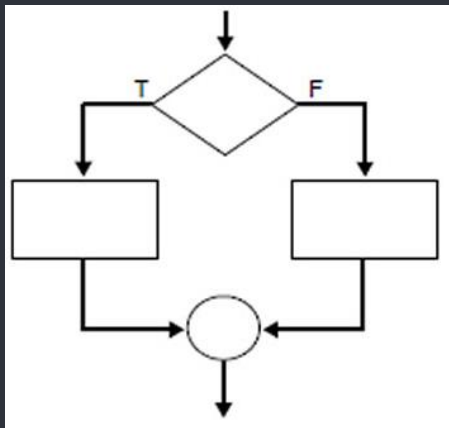
< Además de utilizar valores numéricos, en las condiciones podemos comparar cadenas de texto. En código se ingresa el importe y la forma de pago (A o B). Esta elección modifica el valor final del importe >

```
importe = int(input("Ingrese el monto de la compra: "))
formaPago = input("Forma de pago? A: Contado. B: Otra. ")
```

```
if formaPago == "A":
    importe = importe * 0.9
else:
    importe = importe * 1.15
```

```
print("El importe a pagar es:", importe)
```

# Estructuras de control {



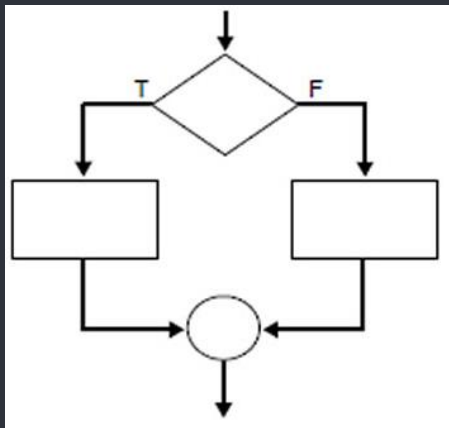
Condicional

< Podemos utilizar una estructura condicional anidada, en la que según si la condición se cumple o no, podemos volver a corroborar otra u otras condiciones >

```
nota = int(input("Ingrese la nota: "))
```

```
if nota >= 7:
    print("Ha aprobado la materia.")
else:
    if nota >= 4:
        print("Debe rendir examen.")
    else:
        print("Debe recursar.")
```

# Estructuras de control {



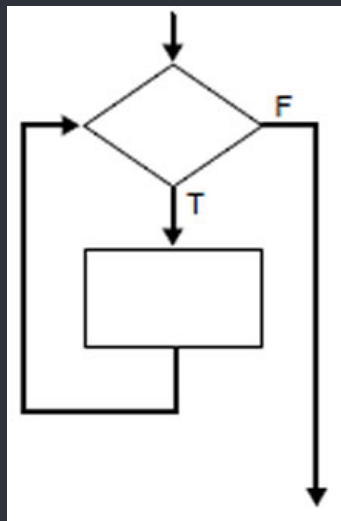
Condicional

< Una forma mas rápida y fácil de escribir los condicionales anidados es con la instrucción **elif** >

```
nota = int(input("Ingrese la nota: "))
```

```
if nota >= 7:
    print("Ha aprobado la materia.")
elif nota >= 4:
    print("Debe rendir examen.")
else:
    print("Debe recursar.")
```

# Estructuras de control {

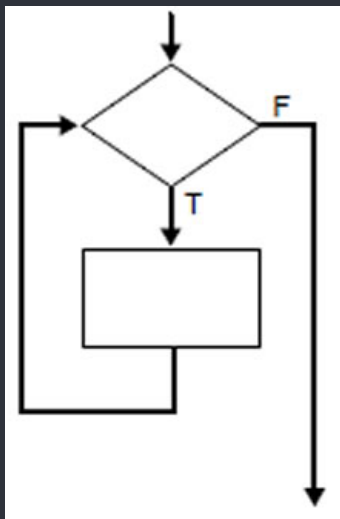


Repetición

< Repiten un código dependiendo de una condición o de un contador. Si se cumple la condición se ejecuta un bloque de código y se comprueba nuevamente la condición. Pueden ser de dos clases >

- **Ciclos Exactos:** Conocemos la cantidad exacta de repeticiones. Ese valor es aportado al iniciar el programa o por el usuario antes de que se inicie el ciclo.
- **Ciclos Condicionales:** No se conoce de antemano la cantidad de repeticiones. Dependen de una condición que puede variar. Finaliza cuando la condición es falsa. Se puede repetir una vez, varias veces o ninguna vez.

# Estructuras de ciclo condicional {



Repetición

< Un bucle o lazo **while** ejecuta un bloque de código siempre que su condición sea **True**. Una vez que la condición devuelve el valor **False**, el programa sale del lazo >

# Sumar todos los numeros hasta que se ingrese 9

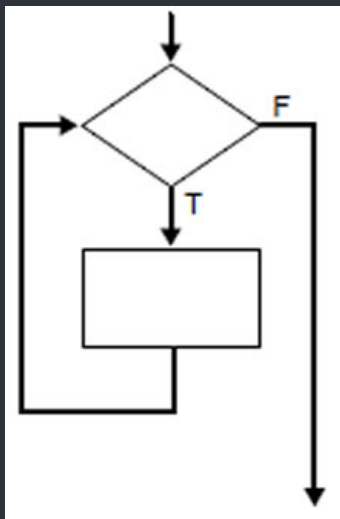
```
suma = 0
numero = 0
```

```
while numero != 9:
    numero = int(input("Ingrese un numero: "))
    suma += numero
```

```
print("La suma de los numeros es: ", suma)
```



# Estructuras de ciclo condicional {



Repetición

}

< Si necesitamos escapar del lazo sin que se cumpla la condición original, podemos usarlo con **break** >

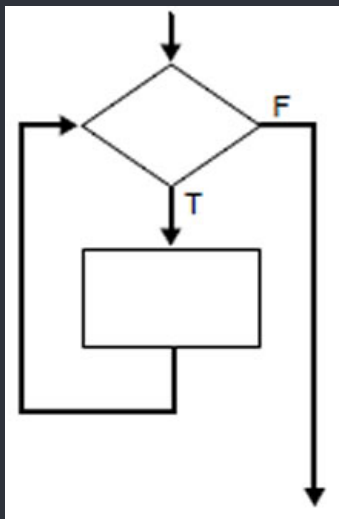
```
# Sumar todos los números hasta que se ingrese 9  
# o la suma supere 30
```

```
suma = 0  
numero = 0
```

```
while numero != 9:  
    numero = int(input("Ingrese un numero: "))  
    suma += numero  
    if suma > 30:  
        print("La suma supera 30")  
        break
```

```
print("La suma de los numeros es: ", suma)
```

# Estructuras de ciclo exacto {

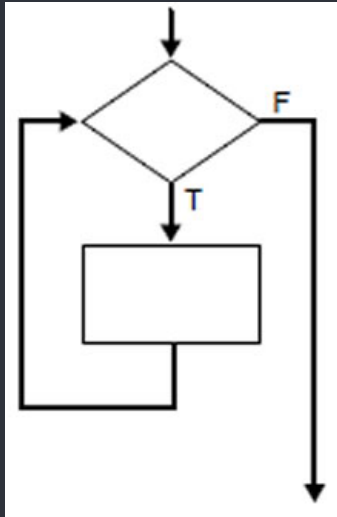


Repetición

< El bucle for es una estructura de control que se utiliza para iterar sobre una secuencia de elementos, como una lista, cadena, rango o cualquier otro objeto iterable. Su sintaxis es simple y compacta, y consta de tres partes: la palabra clave for, una variable de iteración que toma el valor de cada elemento en la secuencia en cada iteración, la palabra clave in, y la secuencia sobre la que se va a iterar. El bucle for recorre cada elemento de la secuencia y ejecuta un bloque de código para cada uno de ellos, lo que lo hace ideal para realizar tareas repetitivas o para procesar cada elemento de una colección >

```
for i in range(inicio, fin, paso):  
    # sentencia1  
    # sentencia2  
# primer sentencia fuera del for
```

# Estructuras de ciclo exacto {



Repetición

# Programa que imprime los números del 1 al 10.

```
for num in range(1, 11, 1):  
    print(num, end=" ")
```

# Programa que imprime los números del 5 al 15 de dos en dos.

```
for num in range(5, 16, 2):  
    print(num, end=" ")
```

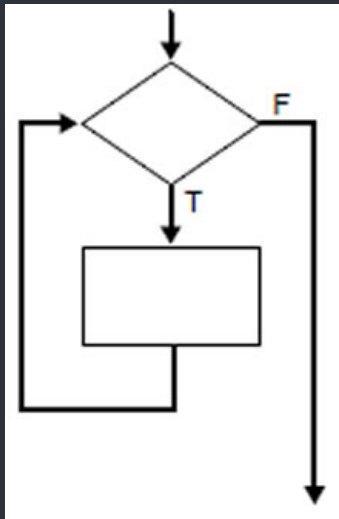
# Programa que imprime los números del 3 al 10.

```
for num in range(3, 11):  
    print(num, end=" ")
```

# Programa que imprime los números del 0 al 9.

```
for num in range(10):  
    print(num, end=" ")
```

# Estructuras de ciclo exacto {



Repetición

< El bucle for permite no solo iterar en un rango, sino también podemos iterar con cadenas, con listas, con tuplas, etc >

```
cadena = "Python"
```

```
for letra in cadena:
    print(letra, end="")
```

# Resultado: Python

```
numeros = [2, 4, 6, 8, 10]
```

```
for num in numeros:
    print(num*2, end=" ")
```

# Resultado: 4 8 12 16 20

```
1
2
3 Aprender a programar
4
5 es aprender a pensar.
6
7
8
9
10
```

```
11 { Steve Jobs; }
12
13
14
```



```
1
2
3
4
5 { Nos vemos en la
6 proxima clase }
7
8
9
10
11
12
13
14
```

