

## 1. ¿Qué es un condicional?

En Python, un condicional te permite tomar decisiones en tu código en función de ciertas condiciones. Te ayuda a controlar el flujo de tu programa, ejecutando diferentes partes del código según si se cumplen o no ciertas condiciones.

Es una forma de hacer que tu programa sea más inteligente y adaptable a diferentes situaciones.

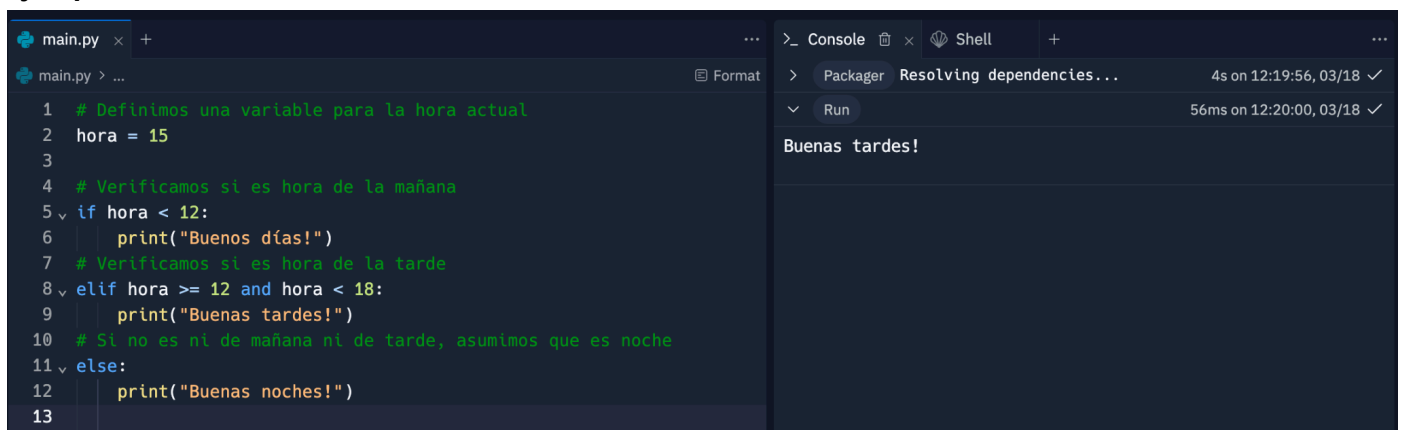
**if:** Permite ejecutar un bloque de código si una condición es verdadera. True o False

**elif:** Abreviatura de "else if".

Se utiliza para verificar múltiples condiciones después de la primera instrucción if. Solo se evalúa si la condición previa a elif es falsa.

**else:** Se ejecuta si ninguna de las condiciones anteriores (if o elif) es verdadera.

**Ejemplo:**



```
1 # Definimos una variable para la hora actual
2 hora = 15
3
4 # Verificamos si es hora de la mañana
5 if hora < 12:
6     print("Buenos días!")
7 # Verificamos si es hora de la tarde
8 elif hora >= 12 and hora < 18:
9     print("Buenas tardes!")
10 # Si no es ni de mañana ni de tarde, asumimos que es noche
11 else:
12     print("Buenas noches!")
13
```

The screenshot shows a code editor with a Python file named 'main.py'. The code defines a variable 'hora' with the value 15. It then uses an if-elif-else conditional to print a greeting based on the time. The console output shows 'Buenas tardes!'. The console also shows the execution time for the 'Run' command as 56ms on 12:20:00, 03/18.

Vemos en nuestro ejemplo que nos ha imprimido, Buenas tardes! ya que definimos nuestra hora a las 15:00, Python detectó nuestro condicional elif si es mayor o igual que las 12 o menor a las 18 imprimirá, Buenas tardes!

## 2. ¿Cuáles son los diferentes tipos de bucles en Python? ¿Por qué son útiles?

Primeramente explicaré que en Python, los bucles son estructuras de control que permiten ejecutar un bloque de código repetidamente hasta que se cumpla una condición específica.

Los principales tipos de bucles en Python son:

## Bucle “for”:

Este bucle se utiliza para iterar sobre una secuencia (como una lista, una tupla, un diccionario, etc.) o cualquier objeto iterable. Por cada iteración, el bucle for asigna un valor de la secuencia a una variable y ejecuta el bloque de código asociado.

### Ejemplo:

```
1 # Ejemplo de bucle for, imprime todos los elementos de la lista
2
3 frutas = ["manzana", "banana", "cereza"]
4 for fruta in frutas:
5     print(fruta)
```

Run 44ms on 12:30:17, 03

```
manzana
banana
cereza
```

Vemos en nuestro ejemplo que el bucle for recorre la lista frutas e imprime cada elemento de la lista.

## Bucle “while”:

Este bucle se utiliza para repetir un bloque de código mientras se cumpla una condición determinada. El bucle while ejecutará el bloque de código repetidamente hasta que la condición se vuelva falsa.

### Ejemplo:

```
main.py x +
main.py > ... Format
1 contador = 0
2 while contador < 5:
3     print("El contador es:", contador)
4     contador += 1
5
```

Console x Shell +

Packager Resolving dependencies... 2s on 12:36:01, 03/18 ✓

Run 64ms on 12:36:04, 03/18 ✓

```
El contador es: 0
El contador es: 1
El contador es: 2
El contador es: 3
El contador es: 4
```

En este ejemplo, el bucle while se ejecutará mientras el valor de contador sea menor que 5. En cada iteración, se imprime el valor actual del contador y luego se incrementa en 1. Cuando el contador alcanza el valor de 5, la condición `contador < 5` ya no se cumple y el bucle se detiene.

## ⚠ IMPORTANTE ⚠

**\*Este enfoque evita que el bucle se convierta en infinito, ya que hay una condición explícita que determina cuándo debe detenerse.**

Para evitar que nuestro bucle sea infinito también podemos utilizar la declaración **break**.

Veamos un ejemplo de cómo usarlo:

```
1 contador = 0
2 while True:
3     print("El contador es:", contador)
4     contador += 1
5
6     if contador >= 5:
7         break
```

Run 48ms on 13:26:16, 03/18 ✓

```
El contador es: 0
El contador es: 1
El contador es: 2
El contador es: 3
El contador es: 4
```

En este ejemplo, el bucle while se ejecutará indefinidamente hasta que la condición `contador >= 5` se cumpla. En ese momento, la declaración `break` se ejecutará y el bucle se detendrá, permitiendo que el programa continúe con la ejecución del código que sigue después del bucle. Esto evita que el bucle sea infinito y proporciona una forma de salir del bucle cuando sea necesario.

### 3. ¿Qué es una lista por comprensión en Python?

Una lista por comprensión en Python es una forma concisa y elegante de crear listas. Permite crear listas de una manera más compacta y legible mediante la aplicación de una expresión a cada elemento de otra secuencia o iterable, como otra lista, una tupla, un rango, etc.

La sintaxis general de una lista por comprensión es la siguiente:

```
nueva_lista = [expresion for elemento in iterable if condicion]
```

**expresion:** es la expresión que se aplicará a cada elemento del iterable.

**elemento:** es una variable que representa cada elemento del iterable.

**iterable:** es la secuencia o iterable de la cual se tomarán los elementos.

**condicion:** es una condición opcional que filtra los elementos del iterable.

#### Ejemplo:

Vamos a crear una lista de los cuadrados de los números del 1 al 5 usando una lista por comprensión de esta manera:

```
1 cuadrados = [x ** 2 for x in range(1, 6)]
2 print(cuadrados)
```

Console

```
> Packager Resolving dependencies... 2s on 12:45:02, 03/18 ✓
Run 64ms on 12:45:05, 03/18 ✓
[1, 4, 9, 16, 25]
```

Al imprimir vemos nuestro resultado tenemos una lista `[1, 4, 9, 16, 25]`, que son los cuadrados de los números del 1 al 5.

\*Las listas por comprensión son una herramienta poderosa y ampliamente utilizada en Python para crear listas de forma eficiente y legible.

## 4. ¿Qué es un argumento en Python?

En Python, un argumento se refiere a cualquier valor que se pasa a una función cuando se la llama. Los argumentos son utilizados para proporcionar datos de entrada a una función para que pueda realizar operaciones en ellos.

Hay varios tipos de argumentos en Python, incluyendo argumentos posicionales, argumentos de palabra clave, argumentos predeterminados y argumentos de longitud variable.

**Veamos unos ejemplos:**

### Argumentos posicionales:

Son argumentos que se pasan a una función en el mismo orden en el que están definidos en la declaración de la función. Se corresponden uno a uno con los parámetros de la función.

**Ejemplo:**

```
1 def suma(a, b):
2     return a + b
3
4 resultado = suma(3, 5)
5 print(resultado) # Output: 8
6
```

Run

63ms on 12:53:35, 03/18

8

### Argumentos de palabra clave:

Son argumentos que se pasan a una función utilizando el nombre del parámetro al que se desea asignar el valor.

Esto permite pasar los argumentos en un orden diferente al de la declaración de la función.

**Ejemplo:**

```
1 def saludar(nombre, saludo):
2     return f"{saludo}, {nombre}!"
3
4 mensaje = saludar(nombre="Juan", saludo="Hola")
5 print(mensaje) # Output: "Hola, Juan!"
```

Run

61ms on 12:55:10, 03/18

Hola, Juan!

### Argumentos predeterminados:

Son argumentos que tienen un valor asignado por defecto en la declaración de la función.

Si no se proporciona un valor para estos argumentos al llamar a la función, se utilizará el valor predeterminado.

**Ejemplo:**

```
1 def saludar(nombre, saludo="Hola"):
2     return f"{saludo}, {nombre}!"
3
4 mensaje = saludar("Juan")
5 print(mensaje) # Output: "Hola, Juan!"
6
```

Run

58ms on 12:58:00, 03/18

Hola, Juan!

### Argumentos de longitud variable:

Son argumentos que permiten pasar un número variable de argumentos a una función.

Esto se logra utilizando el operador `*` para los argumentos posicionales y `**` para los argumentos de palabra clave.

### Ejemplo:

```
1 def suma(*numeros):
2     total = 0
3     for num in numeros:
4         total += num
5     return total
6
7 resultado = suma(1, 2, 3, 4, 5)
8 print(resultado) # Output: 15
```

Run 53ms on 12:59:53, 03/18 ✓

15

\*Estos son algunos ejemplos de diferentes tipos de argumentos en Python y cómo se utilizan en las funciones. Los argumentos proporcionan flexibilidad y permiten que las funciones manejen una variedad de situaciones y datos de entrada.

## 5. ¿Qué es una función Lambda en Python?

Una función lambda en Python es una función anónima y de una sola línea que se define utilizando la palabra clave lambda.

Estas funciones pueden tener cualquier número de argumentos, pero solo pueden tener una expresión como cuerpo de la función.

Son útiles cuando necesitas una función temporal para una operación específica y no quieres definir una función regular utilizando la declaración `def`.

La sintaxis general de una función lambda es la siguiente:

```
lambda argumentos: expresión
```

Veamos un ejemplo:

```
1 numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
3 numeros_pares = list(filter(lambda x: x % 2 == 0, numeros))
4
5 print(numeros_pares) # Output: [2, 4, 6, 8, 10]
```

Run 56ms on 13:05:47, 03/18 ✓

[2, 4, 6, 8, 10]

En este ejemplo, la función lambda `lambda x: x % 2 == 0`

Se utiliza como argumento de la función `filter()`.

Esta función lambda toma un número `x` como entrada y devuelve `True` si `x` es par y `False` si no lo es.

La función `filter()` luego filtra los elementos de la lista `numeros` que cumplen con esta condición, devolviendo solo los números pares.

## 6. ¿Qué es un paquete pip?

Un paquete Pip es una distribución de software en Python que puede ser instalada utilizando la herramienta de gestión de paquetes de Python llamada Pip. Pip es el sistema estándar para instalar y administrar paquetes de software en Python. Los paquetes Pip pueden contener código Python, módulos, scripts, y otros recursos necesarios para una funcionalidad específica.

Los paquetes Pip son esenciales para la reutilización de código, ya que permiten a los desarrolladores compartir y distribuir fácilmente sus bibliotecas y herramientas con otros usuarios de Python. Esto promueve la colaboración y la creación de un ecosistema robusto de software en Python.

Vamos a ver unos comandos básicos para trabajar con pip:

### Instalar un paquete:

Utiliza el comando `pip install` seguido del nombre del paquete que deseas instalar.

Por ejemplo:

```
pip install nombre_paquete
```

```
pip install requests
```

Este comando instalará el paquete `requests`, que es una popular biblioteca para realizar solicitudes HTTP en Python.

### Desinstalar un paquete:

Utiliza el comando `pip uninstall` seguido del nombre del paquete que deseas desinstalar.

Por ejemplo:

```
pip uninstall nombre_paquete
```

### Actualizar un paquete:

Utiliza el comando `pip install --upgrade` seguido del nombre del paquete que deseas actualizar. Por ejemplo:

```
pip install --upgrade nombre_paquete
```

### Mostrar información sobre un paquete específico:

Utiliza el comando `pip show` seguido del nombre del paquete para mostrar información detallada sobre un paquete específico.

Por ejemplo:

```
pip show nombre_paquete
```

### Mostrar todos los paquetes instalados:

Utiliza el comando **pip list** para mostrar una lista de todos los paquetes instalados en tu sistema.

En este ejemplo, **pip list** muestra una lista de todos los paquetes instalados en el entorno de Python actual, junto con sus versiones correspondientes. Cada línea muestra el nombre del paquete seguido de su versión. Esto te permite verificar qué paquetes están instalados y qué versiones tienes en tu entorno de Python.

Package	Version
-----	-----
beautifulsoup4	4.9.3
certifi	2020.12.5
chardet	4.0.0
idna	2.10
numpy	1.20.0
pandas	1.2.1
pip	21.0.1
python-dateutil	2.8.1
pytz	2021.1
requests	2.25.1
setuptools	49.6.0
six	1.15.0
urllib3	1.26.3

\*Estos son solo algunos ejemplos de cómo puedes utilizar pip para administrar paquetes de Python en tu sistema. Puedes consultar la documentación oficial de pip para obtener más información y opciones avanzadas:

[Documentación de pip.](#)

### Ejemplo de cómo utilizar el paquete NumPy:

En este ejemplo, importamos NumPy con el alias np y luego creamos un arreglo unidimensional utilizando la función np.array(). Luego, realizamos varias operaciones con

este arreglo, como calcular la suma total, la media y el cuadrado de cada elemento, utilizando las funciones proporcionadas por NumPy. Finalmente, imprimimos los resultados de estas operaciones.

```
import numpy as np
```

```
# Crear un arreglo unidimensional
```

```
arr = np.array([1, 2, 3, 4, 5])
```

```
# Imprimir el arreglo
```

```
print("Arreglo original:")
```

```
print(arr)
```

```
# Calcular la suma de todos los elementos del arreglo
```

```
suma_total = np.sum(arr)
```

```
print("Suma total de los elementos:", suma_total)
```

```
# Calcular la media de los elementos del arreglo
```

```
media = np.mean(arr)
```

```
print("Media de los elementos:", media)
```

```
# Calcular el cuadrado de cada elemento del arreglo
```

```
arr_cuadrado = np.square(arr)
```

```
print("Arreglo con cada elemento al cuadrado:")
```

```
print(arr_cuadrado)
```

Run

Arreglo original:

[1 2 3 4 5]

Suma total de los elementos: 15

Media de los elementos: 3.0

Arreglo con cada elemento al cuadrado:

[ 1 4 9 16 25]