1. ¿Qué es un condicional?

Una condicional es una estructura de control que determina qué acción debe ejecutar el código dependiendo de si las condiciones establecidas son verdaderas (True) o falsas (False).

Un ejemplo:

```
edad = 13
if edad < 16: #condición
    print("Eres menor y no puedes trabajar") # acción
elif edad >= 18 and edad < 67:  #segunda condición
    print("Eres apto para trabajar") # acción
else:
    print("Ya puedes jubilarte") #acción si las condiciones anteriores son falsas.
```

En este ejemplo, se evalúa la variable edad para determinar su veracidad. Comienza con una estructura condicional utilizando if para evaluar una condición. Si la condición es verdadera (si la edad es menor que 18), se ejecuta el bloque de código indentado. Si la condición es falsa, se procede a evaluar la siguiente condición.

elif (opcional) permite verificar otra condición solo si las condiciones anteriores (*if* o *elif* previos) son falsas. Si una condición *elif* es verdadera (si la edad es igual o mayor que 18 pero menor que 67), se ejecuta su bloque de código asociado.

else (también es opcional) captura cualquier caso que no haya sido cubierto por las condiciones *if* o *elif* anteriores. El bloque de código asociado a *else* se ejecutará si todas las condiciones anteriores son falsas, es decir, si la edad es igual o mayor que 67.

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

Existen dos tipos principales de bucles: el bucle *for* y el bucle *while*.

El bucle *for* se utiliza para iterar sobre una secuencia, como una lista, tupla, cadena de caracteres u otro objeto iterable, y ejecutar un bloque de código una cantidad específica de veces. Por ejemplo:

```
numeros = [1, 2, 3, 4, 5] #lista
for num in numeros: # para (for) cada elemento (num) en el objeto iterable (la lista numeros)
        print(num) #acción a realizar. Se imprimirá cada número en la lista.
```

El bucle *while* se utiliza para ejecutar un bloque de código tantas veces como la condición específica sea verdadera.

Ejemplo:

```
contador = 0
```

```
while contador < 5: # la condición que se evalúa en cada iteración del bucle. Mientras la condición sea True, la acción o acciones se realizarán.
    print(contador) #acción
    contador += 1  # acción
```

Este bucle *while* imprimirá los números del 0 al 4, ya que incrementamos el contador en cada iteración. El bucle continuará ejecutándose mientras la condición (contador < 5) sea verdadera. Una vez que la condición se vuelva falsa el bucle se detendrá.

Los bucles permiten crear código más eficiente, compacto y flexible al automatizar tareas repetitivas.

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

La lista por comprensión ayuda a crear una nueva lista a partir de iterables existentes (listas, cadenas y tuplas) aplicando una expresión/condición a cada elemento para determinar qué elementos se incluirán en la lista resultante. Su sintaxis es:

<new_list> = [<expression> for <item> in <iterable>]

Esta sintaxis permite un código más legible y eficiente y conciso que los bucles *for* en modelos sencillos (donde la expresión no es compleja)

Por ejemplo, tenemos una lista y queremos crear una nueva lista donde los elementos sean el doble de los elementos iniciales.

Usando el bucle *for …. in* lo haríamos así:

```
nums = [1, 2, 3, 4, 5]
numeros_dobles = []
for num in numeros:
    numeros_dobles.append(num*2)
print(numeros_dobles) #Output: [1, 4, 6, 8,10]
```

el código utilizando el método de lista por comprensión es:

```
nums = [1, 2, 3, 4, 5]
numeros_dobles = [num*2 for num in números] # indicamos que cada elemento de la lista se multiplique por dos.
print(numeros_dobles) #Output: [1, 4, 6, 8,10]
```

Si quisiéramos extraer sólo los números impares, por ejemplo, el código con el bucle sería:

```
numeros_impares = []
for num in range(10):
    if num % 2 != 0: # si el resto de la división del número entre 2 es distinto a cero entonces el número es impar
```

```
        numeros_impares.append(num)

print(numeros_impares)# Output: [1, 3, 5, 7, 9]
```

Con la lista por comprensión:

```
numeros_impares = [num for num in range(10) if num % 2 != 0]
print(numeros_impares) # Output: [1, 3, 5, 7, 9]
```

4. ¿Qué es un argumento en Python?

Un argumento es un valor que se pasa a una función o método cuando este se llama, es decir, para llevar a cabo la acción específica. Los argumentos permiten a las funciones ser flexibles y poder ser utilizadas en varias veces, ya que los valores de los argumentos pueden ser diferentes cada vez que se llama a la función.

Hay diferentes tipos de argumentos:

1. **Argumentos posicionales:** se pasan a una función en un orden específico, basado en la posición de los parámetros definidos en la función. El número de argumentos y su orden deben coincidir con los parámetros definidos en la función.

Ejemplo:

```
def suma(a, b):
    return a + b

resultado = suma(3, 5) # 3 se asigna a a y 5 se asigna a b
print(resultado) # Output: 8
```

2. **Argumentos de palabra clave (keyword arguments):** utilizan el nombre del parámetro al que se quiere asignar el valor, lo que permite cambiar el orden de los argumentos sin afectar el resultado

Ejemplo:

```
def saludar(nombre, saludo):
    return f"{saludo}, {nombre}:"

mensaje = saludar(saludo="Hola", nombre="Juan")
print(mensaje) # Output: "Hola, Juan:"
```

3. **Argumentos por defecto:** tienen un valor predeterminado. Si no se proporciona un valor para ese argumento al llamar a la función, se utilizará el valor predeterminado.

Ejemplo:

```
def saludar(nombre, saludo="Hola"):
    return f"{saludo}, {nombre}."

mensaje = saludar("Sara")
print(mensaje) # Output: "Hola, Sara."
```

4. **Argumentos arbitrarios** (*args y **kwargs**):**

a. ***args**: Permite pasar un número variable de argumentos posicionales a una función. Los argumentos se recogen como una tupla dentro de la función.

Ejemplo:

```
def promedio(*numeros):
    return sum(numeros) / len(numeros) #sumamos los valores de los argumentos y los dividimos entre el número de argumentos

resultado = promedio(5, 10, 15) # decimos cuáles son los valores de los argumentos. Podemos especificar tantos como queramos.
print(resultado) # Output: 10.0
```

b. ****kwargs**: Permite pasar un número variable de argumentos de palabra clave a una función. Los argumentos se recogen como un diccionario dentro de la función.

Ejemplo:

```
def detalles(**info):
    for key, value in info.items(): # formato elementos de un diccionario
    print(f"{key}: {value}")

detalles(nombre="Ana", edad=30, ciudad="Madrid") # los valores para cada elemento del diccionario. Output:
nombre: Ana
edad: 30
ciudad: Madrid.
```

5. ¿Qué es una función Lambda en Python?

Una función lambda es una función anónima y de una sola línea que puede definirse de manera concisa utilizando la palabra clave *lambda*. A diferencia de las funciones definidas con *def,* las funciones lambda no requieren un nombre específico y se utilizan principalmente en situaciones donde se necesita una función rápida y temporal.

Ejemplo: queremos hacer una función de suma de dos parámetros x e y.

El código para la función lambda es:

```
suma = lambda x, y: x + y # la función lambda se asigna a la variable suma y luego se puede llamar como cualquier otra función.
resultado = suma(3, 5)
print(resultado) # Output: 8
```

El código de una función normal sería:

```
def suma(x, y):
        return x+y
resultado = suma(3, 5)
print(resultado) # Output: 8
```

Las funciones lambda son útiles cuando se necesitan funciones pequeñas y simples.

6. ¿Qué es un paquete pip?

pip es el sistema estándar de gestión de paquetes de software desarrollados para Python, es decir, permite instalar, actualizar y administrar esos paquetes. Un paquete es un conjunto de módulos o subpaquetes de bibliotecas y herramientas adicionales de código, datos o documentación que están agrupados y que ofrecen funcionalidades específicas. Una vez instalados pueden ser utilizados varias veces.

Para instalar un paquete utilizando *pip*, se utiliza el siguiente comando en la línea de comandos:

```
pip install nombre_del_paquete
```

pip instala el paquete solicitado y cualquier otro paquete o paquetes necesarios para que el paquete funcione correctamente. La mayoría de los paquetes se encuentran en el repositorio PyPI.