

CHECKPOINT 5

1.- ¿Qué es un condicional?

Un condicional es un requisito que podemos establecer para dinamizar un programa. Es decir, podemos programar, a través de la comprobación de requisitos previos establecidos, el comportamiento que va a tener el programa.

Antes de entrar en ejemplos, vamos a ver los elementos que forman parte de este proceso.

-Una condición que se debe cumplir:

Tomamos 1 o más elementos y decidimos a qué equivale. Por ejemplo, solicitamos el dato de la edad y establecemos nuestra condición en que debe ser 18.

-Un operador de comparación:

Los operadores de comparación son los que verifican la igualdad o desigualdad entre nuestra condición y el objeto introducido. Hay varios operadores:

- Igual ==
- Distinto !=
- Mayor que >
- Mayor o igual que >=
- Menor que <
- Menor o igual que <=

Los dos primeros operadores, igual y distinto, se pueden utilizar para condiciones tanto numéricas como cadenas de texto. Sin embargo, por motivos obvios, el resto sólo serán válidos en condiciones numéricas.

-La tarea que ejecutará el programa dependiendo de la condición.

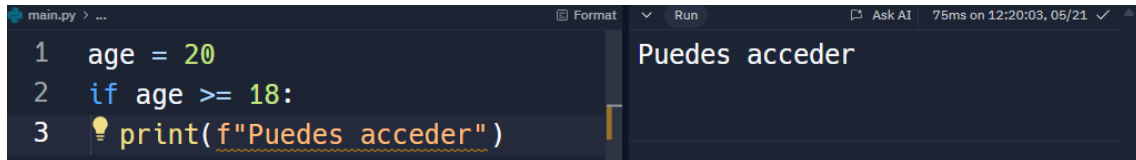
Ahora, para ver clara la sintaxis y entender mejor cada uno de los elementos, os pongo un ejemplo. Quiero que sólo los mayores de 18 años puedan acceder a mi página, así que establezco esa condición:

```
if age >= 18:
```

Y ahora le digo lo que quiero que pase si la condición se cumple:

```
    print(f"Puedes acceder")
```

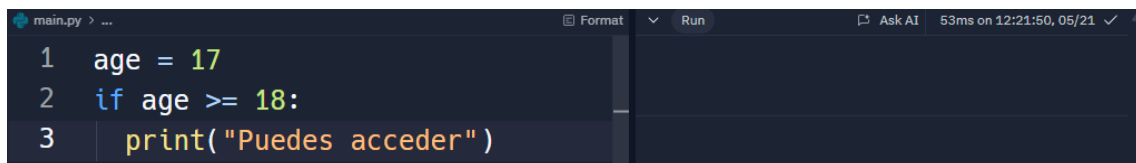
Esto basta para tener un programa dinámico que dependa del cumplimiento de una condición establecida para ejecutar una orden. Pero vamos a ver cómo quedaría en realidad. Si lo ejecuto así y establezco que va a entrar alguien de 20 años, le saldría el mensaje:



```
1 age = 20
2 if age >= 18:
3     print(f"Puedes acceder")
```

The screenshot shows a code editor with a Python file named 'main.py'. The code consists of three lines: line 1 sets 'age' to 20, line 2 starts an 'if' statement with the condition 'age >= 18', and line 3 indented under the 'if' statement is 'print(f"Puedes acceder")'. To the right of the code editor, the output of the program is displayed as 'Puedes acceder'.

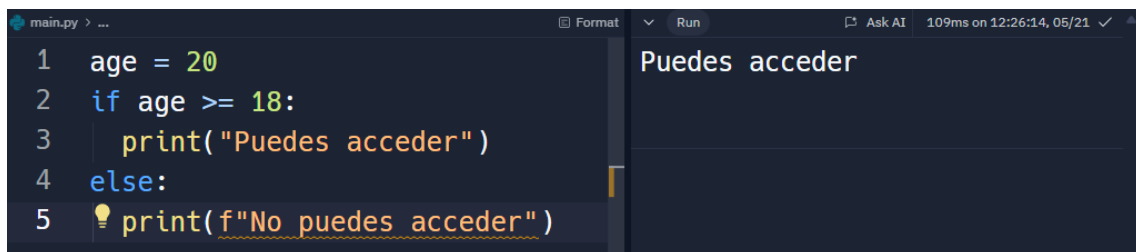
Sin embargo, si intenta entrar alguien menor, no saldrá nada.



```
1 age = 17
2 if age >= 18:
3     print("Puedes acceder")
```

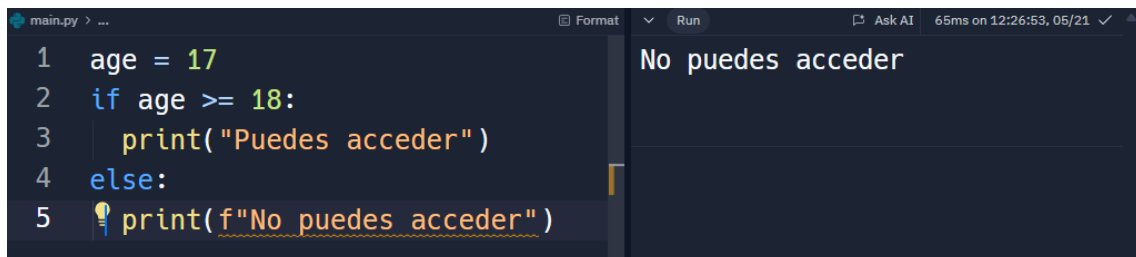
The screenshot shows a code editor with a Python file named 'main.py'. The code consists of three lines: line 1 sets 'age' to 17, line 2 starts an 'if' statement with the condition 'age >= 18', and line 3 indented under the 'if' statement is 'print("Puedes acceder")'. To the right of the code editor, the output is empty, indicating that nothing was printed.

Por eso, además de decidir qué debe ocurrir si la condición se cumple, es buena práctica ir un poco más allá y decidir qué queremos que ocurra si no se cumple.



```
1 age = 20
2 if age >= 18:
3     print("Puedes acceder")
4 else:
5     print(f"No puedes acceder")
```

The screenshot shows a code editor with a Python file named 'main.py'. The code consists of five lines: line 1 sets 'age' to 20, line 2 starts an 'if' statement with the condition 'age >= 18', line 3 indented under the 'if' statement is 'print("Puedes acceder")', line 4 is 'else:', and line 5 indented under the 'else' statement is 'print(f"No puedes acceder")'. To the right of the code editor, the output of the program is displayed as 'Puedes acceder'.



```
1 age = 17
2 if age >= 18:
3     print("Puedes acceder")
4 else:
5     print(f"No puedes acceder")
```

The screenshot shows a code editor with a Python file named 'main.py'. The code consists of five lines: line 1 sets 'age' to 17, line 2 starts an 'if' statement with the condition 'age >= 18', line 3 indented under the 'if' statement is 'print("Puedes acceder")', line 4 is 'else:', and line 5 indented under the 'else' statement is 'print(f"No puedes acceder")'. To the right of the code editor, the output of the program is displayed as 'No puedes acceder'.

En ambos casos, tanto si se cumple la condición como si no, tenemos una orden para el programa.

Algo importante a tener en cuenta es la sangría. Si no la respetamos, el programa nos dará error.

Una vez tenemos esta sintaxis clara, es fácil crear programas dinámicos un poco más complejos. Por un lado, podemos crear programas en los que, dependiendo de una variable, pueda haber varias condiciones. Un ejemplo fácil de ver es, por ejemplo, las calificaciones en un examen. En este caso, al haber más de una tarea, se usa el

equivalente en Python a if/else, que es elif. Podemos tomar los posibles resultados de un examen, desde 0 hasta 10, y decidir qué es suspenso y qué aprobado.

```
main.py > ...  
1 resultado = 7  
2 if resultado < 5:  
3     print(f"Insuficiente")  
4 elif resultado == 5 & resultado < 6:  
5     print(f"Suficiente")  
6 elif resultado == 6 & resultado < 7:  
7     print(f"Bien")  
8 elif resultado == 7 & resultado < 9:  
9     print(f"Notable")  
10 else:  
11     print(f"Sobresaliente")
```

Notable

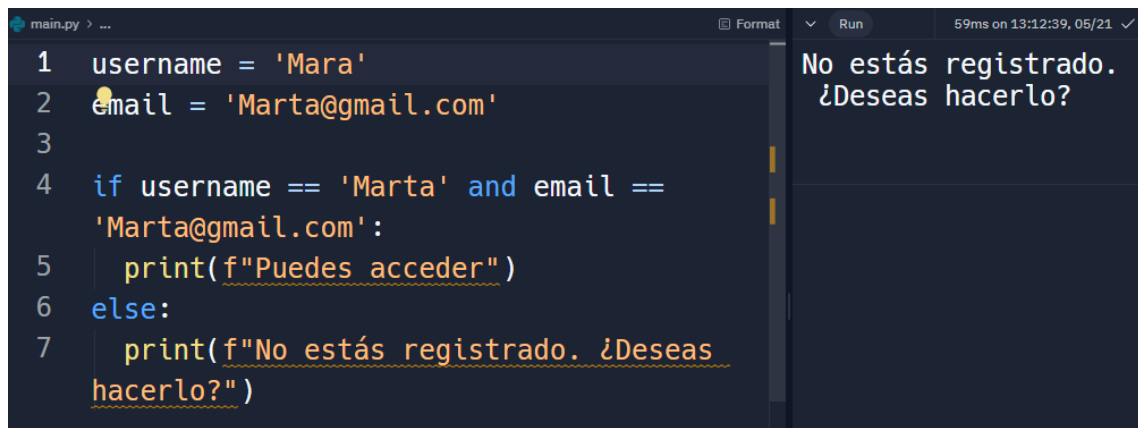
En este ejemplo, no he limitado los números por debajo del 0 ni por encima del 10, porque el resultado de los exámenes no va estar fuera de esos límites. Como podéis ver, he añadido un nuevo elemento, &. Esto se hace para poder trabajar con rangos de condiciones. Es decir, para que un resultado sea “Suficiente”, la condición es que el valor tiene que ser igual a 5 y menor que 6.

Otra opción sería tener más de una variable. Un ejemplo en el que puedes necesitar más de una variable es el permiso de acceso a una página en la que estamos registrados. Es un ejemplo que has visto un millón de veces.

```
main.py > ...  
1 username = 'Marta'  
2 email = 'Marta@gmail.com'  
3  
4 if username == 'Marta' and email ==  
   'Marta@gmail.com':  
5     print(f"Puedes acceder")  
6 else:  
7     print(f"No estás registrado. ¿Deseas  
   hacerlo?")
```

Puedes acceder

Si no estás registrado o te equivocas al meter tus datos:



The image shows a screenshot of a Python IDE. The editor on the left contains a script with the following code:

```
1 username = 'Mara'
2 email = 'Marta@gmail.com'
3
4 if username == 'Marta' and email ==
5     'Marta@gmail.com':
6     print(f"Puedes acceder")
7 else:
8     print(f"No estás registrado. ¿Deseas
9         hacerlo?")
```

The output console on the right displays the result of the script execution:

```
No estás registrado.
¿Deseas hacerlo?
```

The IDE interface includes a top bar with 'main.py > ...', 'Format', 'Run', and a status bar indicating '59ms on 13:12:39, 05/21'.

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

En Python tenemos dos tipos diferentes de bucles, for-in y while. Ambos sirven para recorrer listas, tuplas y diccionarios para que podamos interactuar con los elementos que hay en ellas. Empezaremos con for-in, que es el que usaremos en el 95% de las ocasiones.

Un bucle for-in va a recorrer un conjunto de datos tantas veces como elementos haya en él. Nosotros no necesitamos saber cuántos elementos hay, en el momento en que acabe la lista, el bucle se detiene. Esto es altamente efectivo en conjuntos en los que tenemos una cantidad grande de elementos y no sabemos cuántos son. No necesitas decirle cuándo debe detenerse. Ahora bien, cada tipo de conjunto de datos tiene su sintaxis.

-Listas: pongamos como ejemplo una lista de alumnos y digamos que quiero que el programa la imprima. La sintaxis sería:

```
alumnos = ['Mara', 'Sergio', 'Pedro', 'Sara']
```

Aquí tengo el nombre de la lista en la que quiero usar el bucle for-in. Para ello, le digo lo que necesito (for + variable) de la lista (in + lista):

```
for alumno in alumnos:
```

Es una convención en Python poner a la variable el nombre de la lista en singular cuando el nombre de la lista sea plural. Y después le digo, teniendo en cuenta la sangría, lo que quiero que haga en esa lista. En este caso, quiero que la imprima.

```
    print(alumno)
```



```
main.py > ...  
1 alumnos = ['Mara', 'Sergio', 'Pedro',  
2 'Sara']  
3 for alumno in alumnos:  
4     print(alumno)
```

Mara
Sergio
Pedro
Sara

75ms on 13:38:26, 05/21 ✓

-Tuplas: funciona igual que las listas en cuestión de bucles:

```
main.py > ...
1 alumnos = ('Mara', 'Sergio', 'Pedro', 'Sara')
2
3 for alumno in alumnos:
4     print(alumno)
```

Mara
Sergio
Pedro
Sara

-Diccionarios: al tener pares de clave-valor, va a ser un poco distinto a los dos tipos de conjuntos de datos anteriores. En este caso, no tenemos sólo una variable en bloque, tenemos que proporcionarle dos palabras. Esta vez, en el ejemplo tendremos un diccionario que se llama profesores. Y cada uno de ellos dará una asignatura. Para que la terminología no nos confunda, seguiremos la convención de utilizar el nombre del diccionario en singular y, para la clave, usaremos la palabra 'asignatura'. Nuestro diccionario será:

```
profesores = {
    'Mates': 'Sergio',
    'Lengua': 'Andrea',
    'Historia': 'Alberto',
    'Literatura': 'Adolfo'
}
```

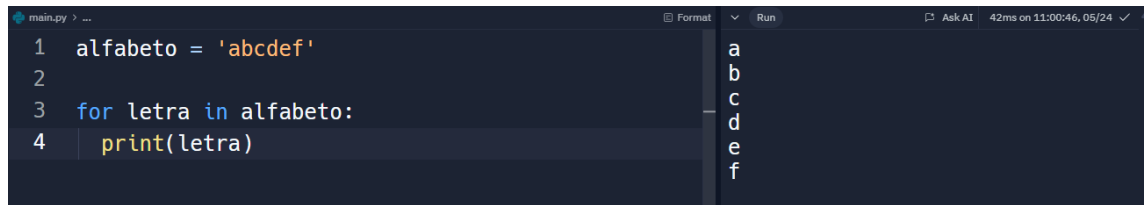
Como ya hemos dicho, al tener ese par de clave-valor, la sintaxis va a ser un poco distinta, ya que tienen que aparecer ambos.

```
for asignatura, profesor in profesores.items():
    print('Asignatura', asignatura)
    print('Profesor', profesor)
```

```
main.py > ...
1 profesores = {
2     'mates': 'Sergio',
3     'lengua': 'Andrea',
4     'historia': 'Alberto',
5     'literatura': 'Adolfo'
6 }
7
8 for asignatura, profesor in profesores.items():
9     print('Asignatura', asignatura)
10    print('Profesor', profesor)
```

Asignatura mates
Profesor Sergio
Asignatura lengua
Profesor Andrea
Asignatura historia
Profesor Alberto
Asignatura literatura
Profesor Adolfo

-Cadenas: el bucle for-in también puede recorrer cadenas y la sintaxis será igual que en el caso de las listas y las tuplas.



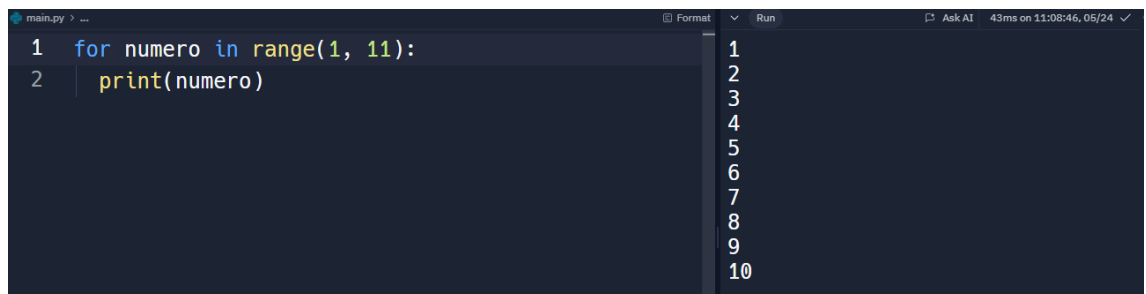
```
1 alfabeto = 'abcdef'
2
3 for letra in alfabeto:
4     print(letra)
```

a
b
c
d
e
f

-Rangos: aquí el bucle for-in cobra especial importancia, ya que nos permite recorrer sólo el rango de ese conjunto de datos que nosotros queramos. Es fácil de ver con un ejemplo. Quiero un rango determinado de números, por ejemplo, del 1 al 10. En ese caso, la sintaxis del bucle tiene que decirle dónde quiero que empiece a contar y dónde quiero que termine. Y, en vez de decirle en qué conjunto de datos, le pongo la función range, que representa una secuencia inmutable de números.

```
for numero in range(1, 11):
    print(numero)
```

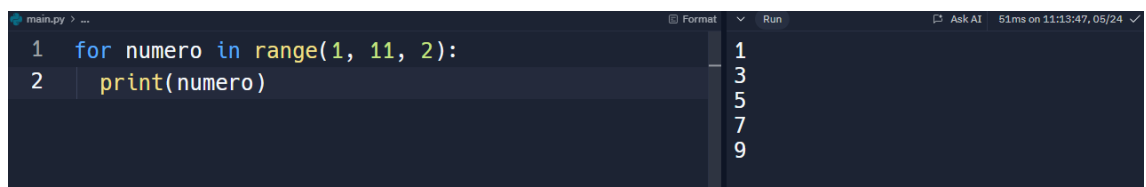
Y eso imprimirá el rango de números que quiero.



```
1 for numero in range(1, 11):
2     print(numero)
```

1
2
3
4
5
6
7
8
9
10

También puedo añadir a la función range un tercer argumento, que es el que dice qué intervalo quiero en ese listado.



```
1 for numero in range(1, 11, 2):
2     print(numero)
```

1
3
5
7
9

Un bucle for-in también se puede romper o alterar cuando nosotros queramos. Para eso tenemos los operadores de flujo 'continue' y 'break'. La diferencia entre ellos, es que 'continue' nos permite continuar recorriendo el conjunto de datos a pesar de alterar su comportamiento cuando encuentre la condición que le hemos dado. Sin embargo, el operador 'break' termina en esa condición especial el recorrido. A pesar de que parece complicado de entender, es bastante sencillo. Si tenemos un listado de amigos, por ejemplo, y ahí uno que me cae mal, le digo que recorra la lista y, cuando lo

encuentre, ponga una nota recordando que me cae mal. Si quiero seguir con el listado, usaría el operador 'continue' y así tendría el listado entero de amigos.

```
1 amigos = ['Leandro', 'Alberto', 'Juan', 'María', 'Sandra']
2
3 for amigo in amigos:
4     if amigo == 'Juan':
5         print('Juan me cae mal')
6         continue
7     else:
8         print(amigo)
```

Leandro
Alberto
Juan me cae mal
María
Sandra

Si usamos el operador 'break', el listado acaba cuando encuentre a Juan.

```
1 amigos = ['Leandro', 'Alberto', 'Juan', 'María', 'Sandra']
2
3 for amigo in amigos:
4     if amigo == 'Juan':
5         print('Juan me cae mal')
6         break
7     else:
8         print(amigo)
```

Leandro
Alberto
Juan me cae mal

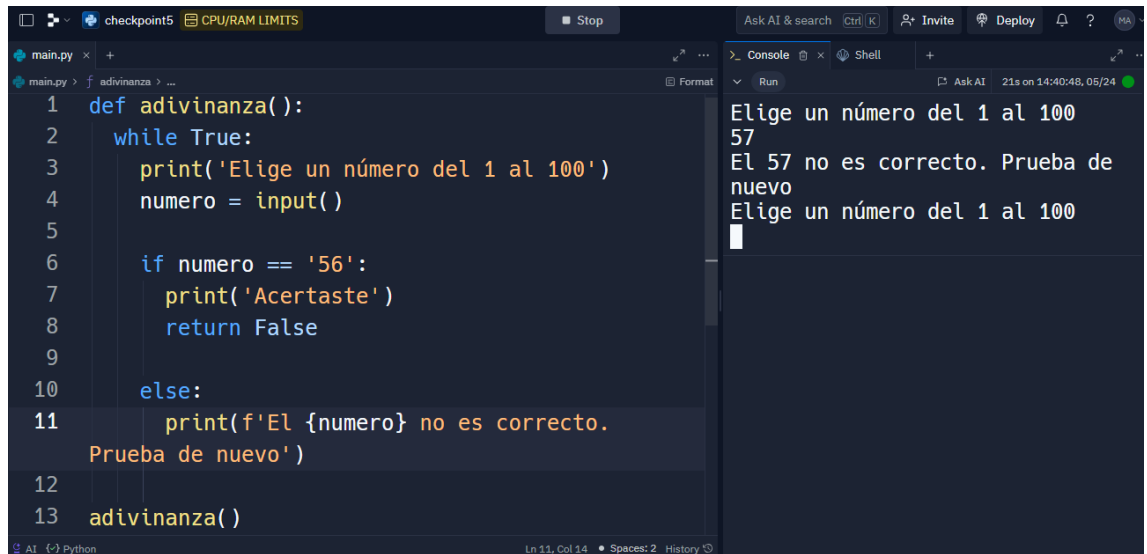
El bucle while apenas se usa. Es un bucle al que hay que decirle cuándo queremos que se detenga porque, de lo contrario, seguirá recorriendo el conjunto de datos hasta el infinito. Uno de los usos más comunes de un bucle while es cuando no sabes cuántas veces quieres que tu programa recorra el conjunto de datos. Un ejemplo de esto sería un juego en el que una persona tiene que adivinar un número. No sabemos cuándo lo va a acertar, así que usamos un bucle while. Esto hará que el bucle recorra el programa una y otra vez hasta que se acierte el número. Vamos a crear una función que se llame 'adivinanza'.

```
def adivinanza():
    while True:
        print('Elige un número del 1 al 100')
        numero = input()

        if numero == '56':
            print('Acertaste')
            return False

        else:
            print('El {numero} no es correcto. Prueba de nuevo.')

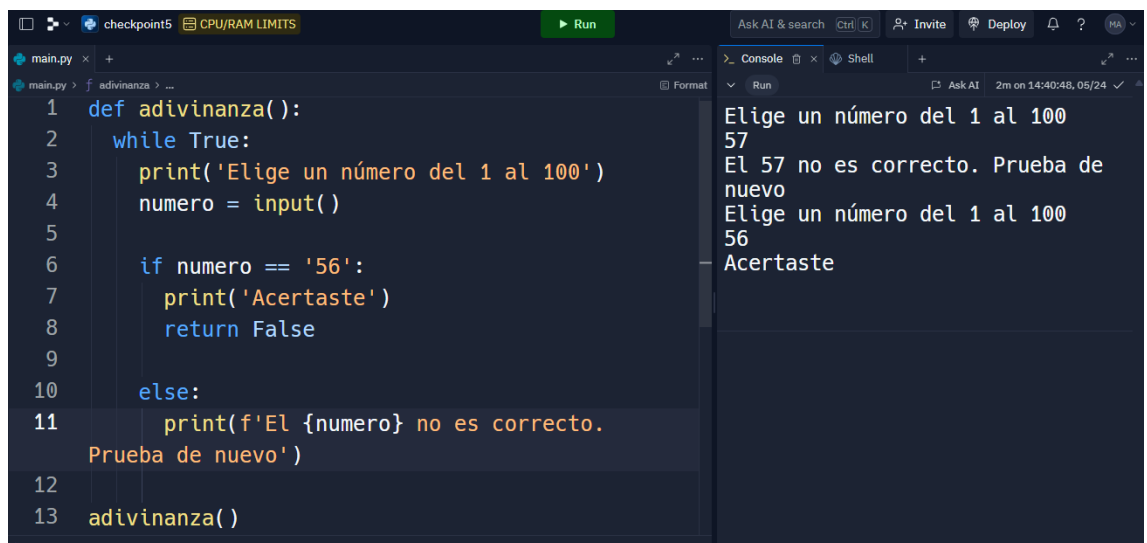
adivinanza()
```

```
1 def adivinanza():
2     while True:
3         print('Elige un número del 1 al 100')
4         numero = input()
5
6         if numero == '56':
7             print('Acertaste')
8             return False
9
10        else:
11            print(f'El {numero} no es correcto.
12              Prueba de nuevo')
13        adivinanza()
```

Elige un número del 1 al 100
57
El 57 no es correcto. Prueba de nuevo
Elige un número del 1 al 100
[]

Como veis, el número introducido no es correcto, por lo que me pide que vuelva a intentarlo. Si os fijáis, arriba del todo, pone 'stop' en vez de 'run', porque la función sigue activa hasta que acertemos el número.



```
1 def adivinanza():
2     while True:
3         print('Elige un número del 1 al 100')
4         numero = input()
5
6         if numero == '56':
7             print('Acertaste')
8             return False
9
10        else:
11            print(f'El {numero} no es correcto.
12              Prueba de nuevo')
13        adivinanza()
```

Elige un número del 1 al 100
57
El 57 no es correcto. Prueba de nuevo
Elige un número del 1 al 100
56
Acertaste

Ahora vemos que pone 'run', porque el bucle ya ha terminado de recorrer la lista de números, ya que hemos acertado el número que necesitábamos.

Hasta ahora, hemos visto que los bucles sirven para recorrer un conjunto de datos y decirnos qué hay en él. También hemos podido encontrar un elemento concreto y cambiar su comportamiento respecto a los demás elementos del conjunto. Ahora, vamos a ver otro uso muy útil de los bucles: combinar dos listas distintas. Esto es importante porque, por ejemplo, nos permite añadir a una lista de clientes ya creada, los elementos de otra lista de nuevos clientes. La sintaxis es un poco distinta, pero también tiene sentido.

Tenemos dos listas, clientes y nuevos_clientes:

```
clientes = ['Sandra', 'Roberto', 'Karen']  
nuevos_clientes = ['Manuel', 'Carolina']
```

Ahora le decimos, usando la sintaxis de bucle for-in de listas que queremos que recorra la lista clientes y se quede con los elementos, a los que llamamos cliente, en singular.

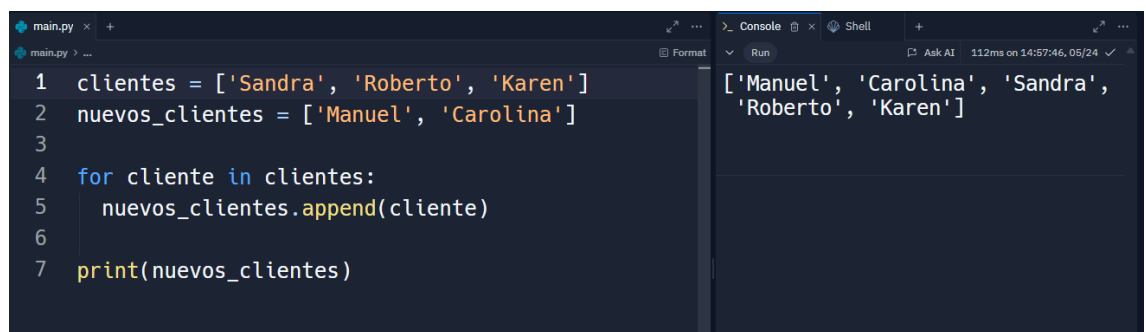
```
for cliente in clientes:
```

Y acto seguido, le decimos que en la lista nuevos_clientes añada los elementos cliente de la primera lista:

```
nuevos_clientes.append(cliente)
```

Y, para ver si ha funcionado, le decimos que imprima la lista nuevos_clientes.

```
print(nuevos_clientes)
```



The screenshot shows a code editor with a file named 'main.py' containing the following Python code:

```
1 clientes = ['Sandra', 'Roberto', 'Karen']  
2 nuevos_clientes = ['Manuel', 'Carolina']  
3  
4 for cliente in clientes:  
5     nuevos_clientes.append(cliente)  
6  
7 print(nuevos_clientes)
```

To the right of the code editor is a console window showing the output of the code:

```
['Manuel', 'Carolina', 'Sandra',  
'Roberto', 'Karen']
```

The console window also shows a 'Run' button and a status bar indicating the execution time as 112ms on 14:57:46, 05/24.

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

Una lista por comprensión es, básicamente, un conjunto de bucles for-in y condicionales que se colocan en una sola línea de código.

Imagina que tienes una lista de números en rango del 1 al 10 y quieres sacar los números pares. Con lo que hemos visto en los bucles for-in, la sintaxis sería:

Tenemos dos listas, una de números en la que el rango es de 1 a 10 y otra de números pares que no tiene ningún valor asignado.

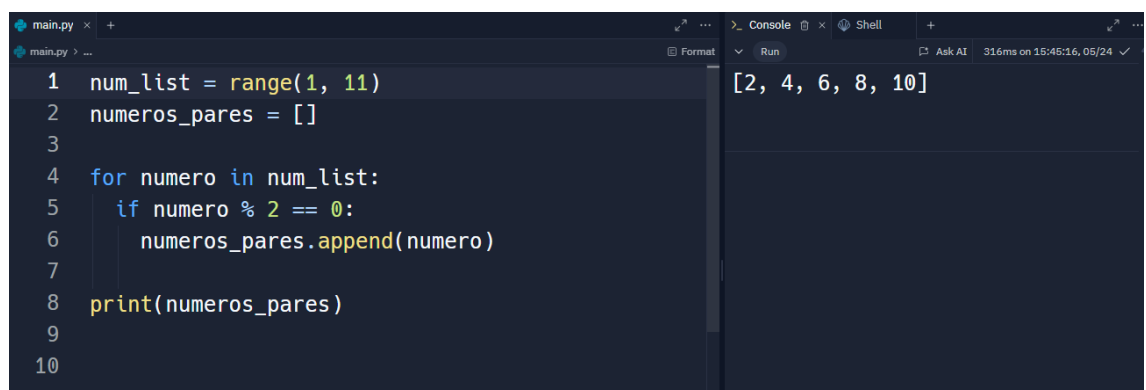
```
num_list = range(1, 11)
numeros_pares = []
```

Usamos el bucle for-in para decir que queremos el número de la lista de números y ponemos una condicional, que el número sea par. En ese caso, el número par se añade a la lista de números pares que estaba sin valor definido.

```
for numero in num_list:
    if numero % 2 == 0:
        numeros_pares.append(numero)
```

Decimos que imprima la lista de números pares y, siguiendo las instrucciones que le hemos dado, la imprimirá con los números de la lista de números que cumplan con la condición que le hemos dado, es decir, que sean pares.

```
print(numeros_pares)
```

A screenshot of a Python IDE with a dark theme. The editor shows a file named 'main.py' with the following code:

```
1 num_list = range(1, 11)
2 numeros_pares = []
3
4 for numero in num_list:
5     if numero % 2 == 0:
6         numeros_pares.append(numero)
7
8 print(numeros_pares)
9
10
```

The right-hand side of the IDE shows a 'Console' panel with the output '[2, 4, 6, 8, 10]'. Above the console, there are buttons for 'Run', 'Format', and 'Ask AI'. A status bar at the bottom right indicates '316ms on 15:48:16, 05/24'.

Con la lista por comprensión, podríamos poner todo en una línea de código de esta forma:

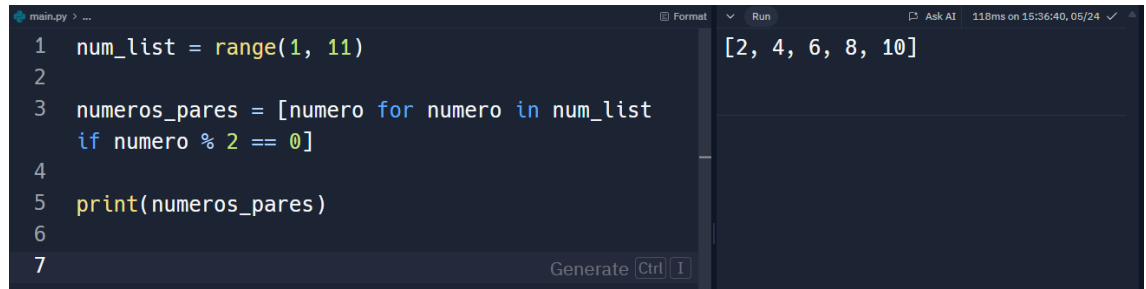
Sólo tenemos que tener definida la lista de números, con su rango del 1 al 10.
`num_list = range(1, 11)`

Ahora es cuando creamos la lista de números pares y le decimos que va a ser igual al número de la lista de números si el número es par.

```
numeros_pares = [numero for numero in num_list if numero % 2 == 0]
```

Y luego le decimos que imprima la lista de números pares.

```
print(numeros_pares)
```



The screenshot shows a Python IDE with a dark theme. The editor window displays the following code:

```
1 num_list = range(1, 11)
2
3 numeros_pares = [numero for numero in num_list
4 if numero % 2 == 0]
5 print(numeros_pares)
6
7
```

Below the code editor, there is a 'Generate' button and a keyboard shortcut 'Ctrl | I'. To the right of the editor, a console or output window shows the result of the execution: `[2, 4, 6, 8, 10]`. The top of the IDE window shows the file name 'main.py' and various icons for formatting, running, and asking AI.

4.- ¿Qué es un argumento en Python?

Los argumentos son valores que se pasan a una función cuando se la llama. Es decir, al crear la función, tienes que decir qué parámetros va a tener para saber qué argumentos se necesitará. Estos argumentos permiten que las funciones sean más flexibles y reutilizables al permitirles trabajar con diferentes datos en cada llamada. Existen dos tipos principales de argumentos en Python:

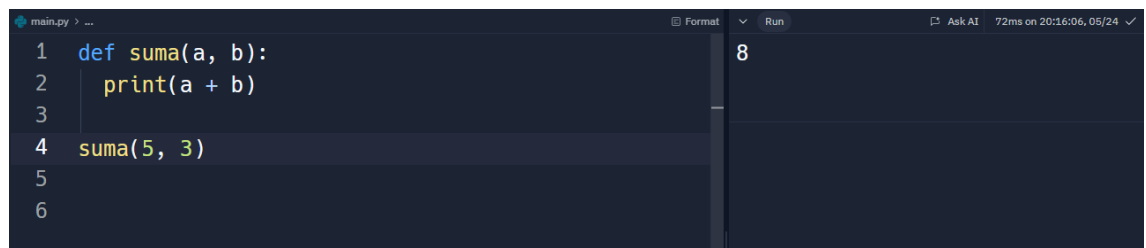
-Argumentos convencionales: son los argumentos que se pasan de manera posicional. Es decir, si tienes una función para sumar, los argumentos que proporcionas al llamar a la función se asignan a los parámetros en el mismo orden en que los pasaste.

Creemos la función 'suma' y le decimos que tiene dos parámetros, así que ya sabemos que va a tener dos argumentos. Y le decimos que, lo que debe hacer la función, es imprimir la suma de los dos parámetros.

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

Llamamos a la función y le decimos que los argumentos son 5 y 3. El sistema los pone en ese orden y ejecuta la función.

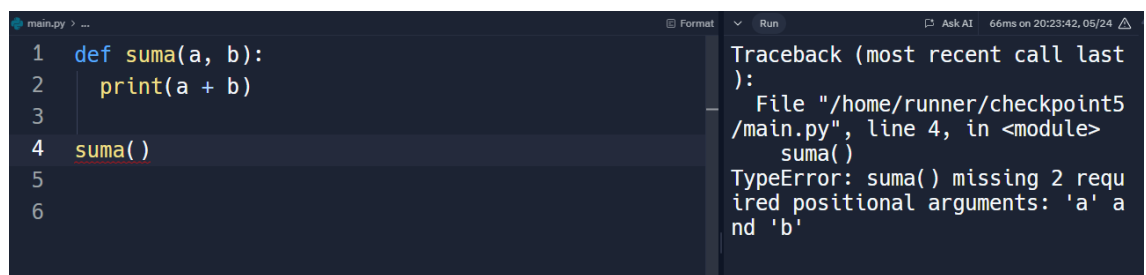
Suma(5, 3)



```
1 def suma(a, b):  
2     print(a + b)  
3  
4 suma(5, 3)  
5  
6
```

8

Si llamamos a la función sin argumentos, nos dará error y nos dirá que necesitamos los argumentos que requiere esa función:



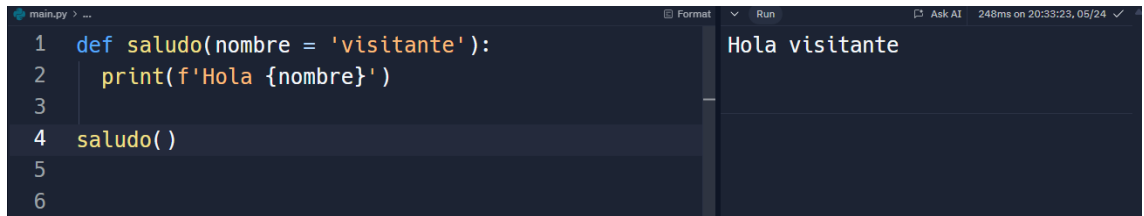
```
1 def suma(a, b):  
2     print(a + b)  
3  
4 suma()  
5  
6
```

```
Traceback (most recent call last):  
  File "/home/runner/checkpoint5/main.py", line 4, in <module>  
    suma()  
TypeError: suma() missing 2 required positional arguments: 'a' and 'b'
```

Otra opción que tenemos, es poner un argumento predeterminado, por si no tenemos el argumento al llamar a la función. De esa forma, no nos dará error. Por ejemplo, en una función de saludo, podemos predefinir el equivalente del parámetro nombre.

```
def saludo(nombre = 'visitante'):
    print(f'Hola, {nombre}')
```

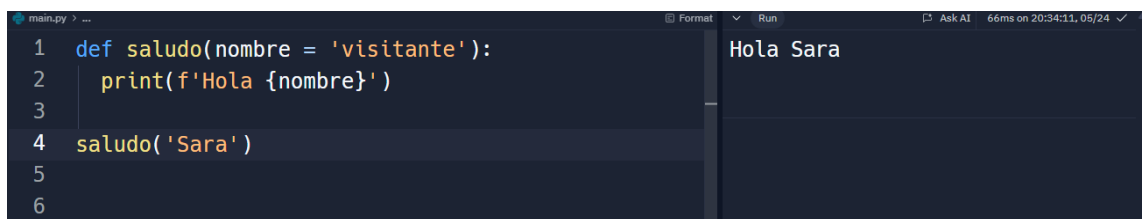
De esta forma, al llamar a la función sin especificar el argumento, no nos dará error, nos saldrá el mensaje con el argumento predefinido.



```
1 def saludo(nombre = 'visitante'):
2     print(f'Hola {nombre}')
3
4 saludo()
5
6
```

Hola visitante

Y si metemos el argumento:



```
1 def saludo(nombre = 'visitante'):
2     print(f'Hola {nombre}')
3
4 saludo('Sara')
5
6
```

Hola Sara

-Argumentos con nombre: en programas grandes, con más de dos argumentos, a veces los posicionales nos pueden dar problemas, porque se nos puede olvidar alguno y, en ese caso, los demás pueden no estar en su sitio. Por ese motivo, utilizamos argumentos con nombre. De esa forma, podemos usarlos en el orden que queramos. Por ejemplo, una función de saludo. Pido los parámetros de nombre, apellido y ciudad.

```
def saludo(nombre, apellido, ciudad
```

Y le digo que quiero imprimir un saludo:

```
print(f'Hola, {nombre} {apellido} de {ciudad}')
```

Al llamar a la función, tengo que nombrar los parámetros y darles el argumento:


```
saludo(nombre = 'Marta', apellido = 'Bezares', ciudad = 'Vitoria')
```



```
1 def saludo(nombre, apellido, ciudad):
2     print(f'Hola {nombre} {apellido}, de {ciudad}.')
3
4 saludo(nombre = 'Marta', apellido = 'Bezares',
5        ciudad = 'Vitoria')
6
```

Hola Marta Bezares, de Vitoria.

Otra de las ventajas, es que puedes cambiar el orden como te convenga.



The image shows a code editor window with a dark theme. The left pane contains Python code for a function named `saludo`. The right pane shows the output of the function. The code defines `saludo` with three parameters: `nombre`, `apellido`, and `ciudad`. It uses an f-string to print a greeting. The function is then called with keyword arguments: `nombre = 'Marta'`, `apellido = 'Bezares'`, and `ciudad = 'Vitoria'`. The output on the right is "De la ciudad de Vitoria, hola Marta Bezares."

```
1 def saludo(nombre, apellido, ciudad):
2     print(f'De la ciudad de {ciudad}, hola {nombre}
3     {apellido}.')
4
5 saludo(nombre = 'Marta', apellido = 'Bezares',
6        ciudad = 'Vitoria')
```

De la ciudad de Vitoria, hola Marta Bezares.

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

Es una herramienta que te permite envolver una función, normalmente más pequeña, y luego pasarla a otras funciones. Una lambda es similar a una variable, donde puedes almacenar algún tipo básico de valores como una cadena o un diccionario o algo así y usarla luego en una función. No puedes darle nombre a una función lambda, ya que son anónimas.

En la sintaxis, en vez de usar la palabra 'def', como hemos hecho hasta ahora en las funciones, usaremos 'lambda'.

Vamos a ver un ejemplo sencillo con una función 'saludo'. Primero creamos la función lambda y vamos a llamarla 'nombre_completo'. Dentro de esa variable, tendremos el nombre y apellido.

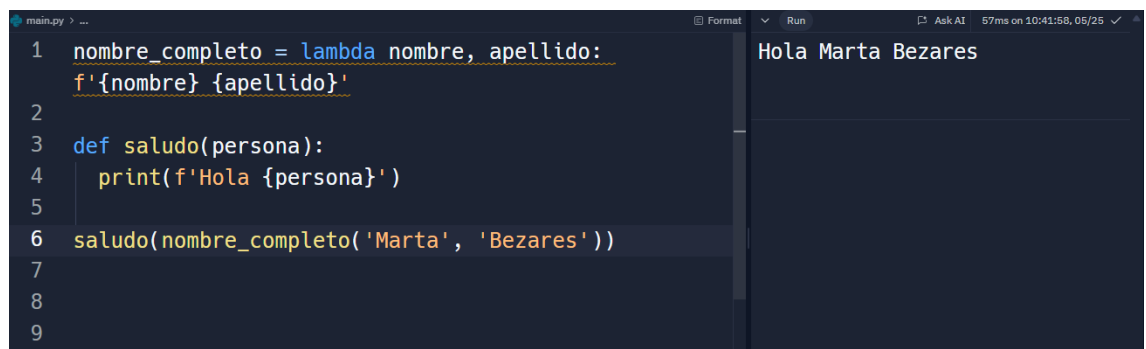
```
nombre_completo = lambda nombre, apellido: f'{nombre} {apellido}'
```

Ahora es cuando creamos una función que tiene un parámetro que hemos denominado 'persona'.

```
def saludo(persona):  
    print(Hola {persona})
```

A la hora de llamar a la función 'saludo', el argumento que le damos es la variable donde está almacenada la función lambda.

```
saludo(nombre_completo(Marta, Bezares))
```



```
1 nombre_completo = lambda nombre, apellido:  
  f'{nombre} {apellido}'  
2  
3 def saludo(persona):  
4     print(f'Hola {persona}')  
5  
6 saludo(nombre_completo('Marta', 'Bezares'))  
7  
8  
9
```

Hola Marta Bezares

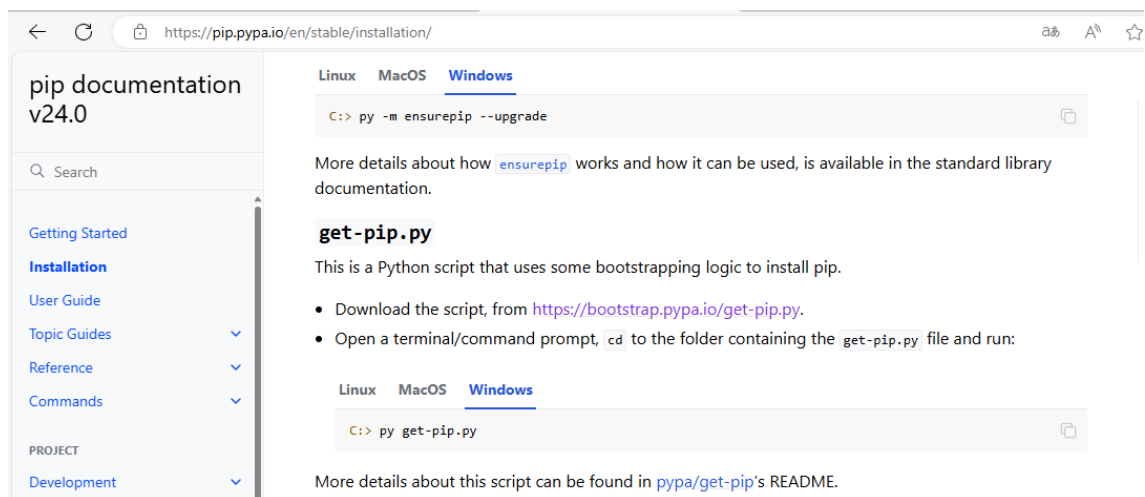
6.- ¿Qué es un paquete pip?

Pip es una herramienta que significa pip installs packages, es decir, nos permite traer paquetes desarrollados ya por otros programadores y usarlos en nuestros programas para facilitarnos el trabajo. Estos paquetes externos están almacenados en una especie de tienda llamada Pypi o cheeseshop

Para poder usar estos paquetes, lo primero que tienes que hacer es instalarlo en tu sistema. Para eso, debes ir a esta página:

<https://bootstrap.pypa.io/get-pip.py>

Haces clic en get-pip.py y eso te abre un archivo .py. Es el que debes guardar en tu ordenador.



Como te dice ahí, abre el terminal y copia ese comando. Una vez instalado, lo único que tienes que hacer es abrir el terminal y ejecutarlo como un archivo normal. Si quieres saber si ya lo tienes instalado, escribe en el terminal `pip --version` y así sabrás si ya está.