0️⃣

Iniciación Python

Hello Python + tipo de datos

```
#Hola mundo
print("Hola python")

#Consultar el tipo de dato: type()
print(type("Soy un dato str")) #Tipo 'str' (string)
print(type(10)) #Tipo 'int' (integer)
print(type(10.5)) #Tipo 'float' (float)
print(type(True)) #Tipo 'bool' (boolean)
print(type(3 + 1j)) #Tipo 'complex' (complejo)
```

Creación de variables

```
#Creación de variables
helloworld = "Hello" + " " + "World"
print(helloworld)

lotofhellos = "hello" * 10
print(lotofhellos)

even_numbers = [1,2,4,8]
odd_numbers = [3,5,7,9]
all_numbers = even_numbers + odd_numbers
print(all_numbers)

print([1,2,3] * 3)

name = "John"
print("Hello, %s!" % name)
```

El `%s` en el código es un marcador de posición para una cadena de texto (string) en Python. Cuando se usa `% name` después de la cadena de texto, Python reemplaza el `%s` en la cadena de texto con el valor de la variable `name`.

La razón por la que se podría preferir usar `%s` en lugar de concatenación de cadenas con `+` es que `%s` puede ser más legible y menos propenso a errores cuando se trabaja con múltiples variables y cadenas largas. Además, el uso de `%s` puede proporcionar un control más detallado sobre el formato de la salida.

La razón por la que se podría preferir usar `%s` en lugar de concatenación de cadenas con `+` es que `%s` puede ser más legible y menos propenso a errores cuando se trabaja con múltiples variables y cadenas largas. Además, el uso de `%s` puede proporcionar un control más detallado sobre el formato de la salida.

```
name = "John"
age = 30
print("Hello, %s! You are %s years old." % (name, age))
```

En este caso, usar `%s` hace que la cadena de formato sea más legible que la concatenación de cadenas.

Sin embargo, en Python moderno, a menudo se prefiere el método `format()` o las f-strings para el formateo de cadenas, ya que son más legibles y versátiles. Aquí está el mismo ejemplo usando f-strings:

```
name = "John"
age = 30
print(f"Hello, {name}! You are {age} years old.")
```

Cualquier objeto que no sea una cadena también se puede formatear utilizando el operador `%s`

```
#this prints out: A list: [1, 2, 3]
mylist = [1,2,3]
print("A list: %s" % mylist)
```

IMPORTANTE:

A continuación se muestran algunos especificadores de argumentos básicos que se deben conocer:

- **%s** = String (o cualquier objeto que se quiera representar como una cadena, como los números)
- **%d** = Integers
- **%f** = FLoat (números de punto flotante o decimales)
- **%.<numero de dígitos>f** = Números de coma flotante con una cantidad fija de dígitos a la derecha del punto
- **%x/%X** = Integers en representación hexadecimal (minúsculas, mayúsculas)

Función len()

La función `len()` en Python devuelve la longitud (el número de caracteres) de una cadena de texto (string). En la línea 31 de tu código, `len(astring)` devuelve la longitud de la cadena `astring`, que es "Hello world!". Por lo tanto, `len(astring)` devolverá 12, porque "Hello world!" tiene 12 caracteres, incluyendo el espacio en blanco.

```
astring = "Hello world!"
print("single quotes are ' '")

print(len(astring))
```

> **Output:**
>
> single quotes are ' '
> 12

Sim embargo:

Función index()

```
astring = "Hello world"
print(astring.index("o"))
```

> **Output:**
>
> 4

Imprime 4, porque la posición de la primera ocurrencia de la letra “o” es 4 caracteres (es decir de la poscisión 0-4) con respecto al primer carácter. Aunque haya más de una “o” en el string, solamente reconoce el primero

Función count()

Esta función cuenta el número de repeticiones de una letra en la cadena (en este caso el numero de veces que aparece la letra l (L minúscula):

```
astring = "Hello world!"
print(astring.count("l"))
```

> **Output:**
>
> 3

Sin embargo:

```
astring = "Hello world!"
print(astring[3:7]
```

> **Output:**
>
> lo w

Esto imprime una porción de la cadena, comenzando en el índice 3 (0,1,2,3) y terminando en el índice **6.** Pero ¿Por qué 6 y no 7??

Nuevamente, la mayoría de los lenguajes de programación hacen esto: facilita las operaciones matemáticas dentro de esos paréntesis.

Si solo tiene un número entre corchetes, le dará el carácter único en ese índice. Si omite el primer número pero mantiene los dos puntos, le dará una porción desde el principio hasta el número que dejó. Si omite el segundo número, le dará una porción desde el primer número hasta el final.

Incluso puedes poner números negativos entre paréntesis. Son una forma sencilla de empezar por el final de la cadena en lugar de por el principio. De esta manera, -3 significa "tercer carácter desde el final".

```
astring = "Hello world!"
print(astring[3:7:2])#pos 3-7, de 2 en 2
```

La línea de código `print(astring[3:7:2])` está utilizando el concepto de rebanado (slicing) en Python.

El rebanado en Python te permite obtener una subcadena (o subsecuencia) de una cadena (o secuencia) existente. La sintaxis general para el rebanado es `secuencia[inicio:fin:paso]`.

En tu línea de código, `astring[3:7:2]` hace lo siguiente:

- `3` es el índice de inicio. Python comenzará a rebanar desde el cuarto carácter (Python comienza a contar desde 0).
- `7` es el índice de fin. Python detendrá el rebanado antes del octavo carácter.
- `2` es el paso. Python tomará cada segundo carácter dentro del rango definido.

Por lo tanto, `print(astring[3:7:2])` imprimirá cada segundo carácter de la cadena `astring` desde el cuarto carácter hasta el séptimo carácter.

Por ejemplo, si `astring` es `"Hello, world!"`, entonces `print(astring[3:7:2])` imprimirá `"l, "` porque comienza en el cuarto carácter (`l`), termina antes del octavo carácter (`,`), y toma cada segundo carácter en ese rango.

Imprimir una cadena al revés

```
astring = "Hello world!"
print(astring[::-1])
```

> **Output:**
>
> !dlrow olleH

Le indicamos que [0:0:-1]: los 0s para que coja toda la cadena, ya que si le pones un 1 cogería la segunda posición: (0,**1**). y -1 para que empiece a imprimir de 1 en 1 pero el - hace que cuente desde el final de la cadena.

Para transformar una cadena a mayúsculas o minúsculas:

```
print(astring.upper())
print(astring.lower())
```

Para determinar si una cadena empieza o termina con algo, respectivamente:

```
print(astring.startswith("Hello"))
print(astring.endswith("asdfasdfasdf"))
```

> **Output:**
>
> True.
>
> False

Condiciones

El operador “and” y “or” te permiten construir expresiones booleanas complejas, por ejemplo:

```
name = "John"
age = 23
if name == "John" and age == 23:
    print("Your name is John, and you are also 23 years old."

if name == "John" or name == "Rick":
    print("Your name is either John or Rick.")
```

El operador “in”

El operador **“in”** puede usarse para verificar si un objeto específico existe dentro de un contenedor de objetos iterables, como una lista:

```
name = "John"
if name in ["John", "Rick"]:
    print("Your name is either John or Rick.")
```

Python usa sangría para definir bloques de código, en lugar de corchetes. La sangría estándar de Python es de 4 espacios, aunque las tabulaciones y cualquier

otro tamaño de espacio funcionarán, siempre que sea coherente. Tenga en cuenta que los bloques de código no necesitan ninguna terminación.

Estructura condicional “if”

```
statement = False
another_statement = True
if statement is True:
    # do something
    pass
elif another_statement is True: # else if
    # do something else
    pass
else:
    # do another thing
    pass
```

El operador “is”

En Python, `==` y `is` son dos operadores de comparación diferentes.

- `==` compara el valor de las variables. En tu código, `x == y` devuelve `True` porque las listas `x` e `y` tienen los mismos valores.
- `is` compara las identidades de las variables. En Python, cada objeto tiene una identidad única, que puedes pensar como la dirección de memoria del objeto. En tu código, `x is y` devuelve `False` porque `x` e `y` son dos objetos diferentes, aunque sus valores sean iguales.

Aquí está una analogía para ayudar a entender la diferencia: imagina que tienes dos libros con el mismo contenido. Si comparas el contenido de los libros (`==`), son iguales. Pero si comparas los libros mismos (`is`), son diferentes porque son dos objetos físicos diferentes.

Por lo tanto, en la mayoría de los casos, cuando quieras comparar si dos variables son "iguales" en el sentido común, debes usar `==` en lugar de `is`.

```
x = [1,2,3]
y = [1,2,3]
print(x == y) # Prints out True
print(x is y) # Prints out False
```

Si usamos el operador “not”:

invierte la expresión.

Bucles

Bucle “for”

Los bucles for iteran sobre una secuencia determinada:

```
primes = [2, 3, 5, 7]
for prime in primes:
        print(prime)
```

Los bucles for pueden iterar sobre una secuencia de números usando “range” y “xrange”. La diferencia entre estos es que la función range() devuelve una nueva lista con números del rango especificado, mientras que xrange() devuelve un iterador, que es más eficiente. (Python 3 usa la función range(), como si fuera xrange). La función empieza a contar desde 0. Ejemplo:

```
#0,1,2,3,4
for x in range(5)
        print(x)
#3,4,5,6
for x in range(3, 7)
        print(x)
#3,5,7
for x in range(3, 8, 2)#de 2 en 2
        print(x)
```

Bucle “while”

Los bucles while se repiten siempre que se cumpla una determinada condición booleana. Por ejemplo:

```
count = 0
while count < 5
        print(count)
        count +=1
```

Declaraciones “break” y “continue”

break se usa para salir de un bucle for o while, mientras que **continue** se usa para omitir el bloque actual y regresar a la instrucción for o while. Ejemlos:

```
#imprime 0,1,2,3,4
count = 0
while True:
        print(count)
        count += 1
        if count >= 5:
            break
```

```
count = 0
for x in range(10):
    if x % 2 == 0:
        continue
    print(x)
```

La palabra clave `continue` en Python se utiliza para saltar el resto del código dentro de la iteración actual de un bucle y pasar directamente a la siguiente iteración.

En tu código, `continue` se ejecuta cuando `x % 2 == 0`, es decir, cuando `x` es un número par. Cuando esto sucede, el código salta la línea `print(x)` y pasa a la siguiente iteración del bucle `for`. Como resultado, `print(x)` solo se ejecuta cuando `x` es un número impar.

Por lo tanto, este código imprimirá todos los números impares del 0 al 9.

Uso cláusula “else” para los bucles

A diferencia de lenguajes como C, CPP... podemos usar **else** for bucles. Cuando falla la condición de bucle de la instrucción "for" o " while ", se ejecuta la parte del código en "else". Si se ejecuta una instrucción **break dentro del bucle for, se omite la parte "else".** Tenga en cuenta que la parte "else" se ejecuta incluso si hay una instrucción **de continuación** .

```
# Prints out 0,1,2,3,4 and then it prints "count value reache

count=0
while(count<5):
```

```
    print(count)
    count +=1
else:
    print("count value reached %d" %(count))

# Prints out 1,2,3,4
for i in range(1, 10):
    if(i%5==0):
        break
    print(i)
else:
    print("esto no se imprime porque el bucle
                        for finaliza debido a una interrup
                        ción, pero no debido a falla en
                        condición")
```

Ejercicio

Recorre e imprime todos los números pares de la lista de números en el mismo orden en que se reciben. No imprimas ningún número que venga después de 237 en la secuencia:

```
numbers = [
    951, 402, 984, 651, 360, 69, 408, 319, 601, 485, 980, 507
    615, 83, 165, 141, 501, 263, 617, 865, 575, 219, 390, 984
    386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978
    399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 95
    815, 67, 104, 58, 512, 24, 892, 894, 767, 553, 81, 379, 8
    958, 609, 842, 451, 688, 753, 854, 685, 93, 857, 440, 380
    743, 527
]
# your code goes here
for number in numbers:
    if number == 237:
        break

    if (number % 2 != 0):
        continue
```

```
    print(number)
```

Funciones

```
def my_function():
    print("Hello from a function")

def my_function_with_args(username, greeting):
    print("Hello, %s, From My Funtion!, I wish you %s"%(usern

def sum_two_numbers(a,b):
    return a + b


my_function()
my_function_with_args("John Doe", "a great year!")
x = sum_two_numbers(1,2)
print(x)
```

Clases y objetos

Los objetos son una encapsulación de variables y funciones en una sola entidad. Los objetos obtienen sus variables y funciones de clases. Las clases son esencialmente una plantilla para crear tus objetos.

Una clase muy básica sería:

```
Class myClass:
    variable = "blah"
    def function(self):
        print("This is a message inside the class.")
```

En Python, `self` es una convención para referirse a la instancia actual de una clase. Se utiliza para acceder a las variables y métodos de la clase dentro de la misma.

Cuando defines un método dentro de una clase, el primer parámetro es siempre `self` (aunque el nombre `self` es solo una convención y no una palabra clave de Python, es ampliamente aceptado y se recomienda su uso). Python pasará

automáticamente la instancia actual de la clase a este parámetro cuando llames a un método de la instancia.

Por ejemplo, en tu código, `self` se utiliza en la definición del método `function`. Esto significa que puedes usar `self` dentro de `function` para acceder a `variable` o a cualquier otro atributo o método de la instancia.

Aquí hay un ejemplo de cómo usar `self` para acceder a `variable`:

```
class myClass:
    variable = "blah"

    def function(self):
        print("This is a message inside the class.")
        print("The variable is " + self.variable)
```

En este código, `self.variable` se refiere a la `variable` de la instancia actual de `myClass`.

```
class myClass:
    variable = "blah"
    def function(self): # self es el equivalente a this en ot
        print("This is a message inside the class.")

myobjectx = myClass() #Instanciación de la clase
```

Ahora la variable “myobjectx contiene un objeto de la clase “MyClass” que contiene la variable y la función definida dentro de la clase llamada “MyClass”.

Accediendo a variables de objeto

Para acceder a la variable dentro del objeto recién creado “myobjectx”:

```
class myClass:
    variable = "blah"
    def function(self):
        print("This is a message inside the class.")
myobjectx = myClass()
myobjectx.variable
```

Puedes crear varios objetos diferentes que sean de la misma clase (tengan definidas las mismas variables y funciones). Sim embargo, cada objeto contiene copias independientes de las variables definidas en la clase. Por ejemplo, si tuviéramos que definir otro objeto con la clase “MyClass” y luego cambiar la cadena en la variable anterior:

```
class myClass:
    variable = "blah"
    def function(self):
        print("This is a message inside the class.")
myobjectx = myClass()
myobjecty = MyClass()

myobjecty.variable = "yackity"

# Then print out both values
print(myobjectx.variable)
print(myobjecty.variable)
```

Acceder a funciones de objetos

```
class MyClass:
    variable = "blah"

    def function(self):
        print("This is a message inside the class.")

myobjectx = MyClass()

myobjectx.function()
```

Init()

La funcion init() es una función especial que se llama cuando se inicia la clase. Se utiliza para asignar valores en una clase.

```
class NumberHolder:
    def __init__(self, number):
```

```
        self.number = number
```

Diccionarios

Un diccionario es un tipo de datos similar a las matrices, pero funciona con claves y valores en lugar de índices. Se puede acceder a cada valor almacenado en un diccionario mediante una clave, que es cualquier tipo de objeto (una cadena, un número, una lista, etc.) en lugar de utilizar su índice para abordarlo.

Por ejemplo, una base de datos de números de teléfono podría almacenarse usando un diccionario como este:

```
phonebook = {}
phonebook["John"] = 938477566
phonebook["Jack"] = 938377264
phonebook["Jill"] = 947662781
print(phonebook)
```

Otra forma más sencilla de inicializar un diccionario:

```
phonebook = {
    "John" : 938477566,
    "Jack" : 938377264,
    "Jill" : 947662781
}
print(phonebook)
```

Iterando sobre diccionarios

Los diccionarios se pueden repetir, como una lista. Sin embargo, un diccionario, a diferencia de una lista, no mantiene el orden de los valores almacenados en él. Para iterar sobre pares clave-valor, utilice la siguiente sintaxis:

```
phonebook = {"John" : 938477566,"Jack" : 938377264,"Jill" : 9
for name, number in phonebook.items():
    print("Phone number of %s is %d" % (name, number))
```

Eliminar un valor

```
phonebook = {
   "John" : 938477566,
   "Jack" : 938377264,
   "Jill" : 947662781
}
del phonebook["John"]
print(phonebook)

#Otra forma de eliminación
phonebook = {
   "John" : 938477566,
   "Jack" : 938377264,
   "Jill" : 947662781
}
phonebook.pop("John")
print(phonebook)
```