Parte I: Bases - Sección I: Variables y tipos de datos simples

Variables

Toda **variable** está conectada a un valor, que es la información asociada con ese valor. En este caso se agrega la variable message la cual contiene el valor de texto “Hello world!”.

```
message = “Hello world!”
print(message)
```

```
>>> Hello world!
```

Añadir variables hace un poco más de trabajo para el interpretador de python. Cuando procesa la última línea asocia la variable message con el primer valor otorgado. Al continuar con la ejecución del texto este utiliza el último valor que se le dio a la variable.

```
message = “Hello world!”
print(message)

message = “Hello Python!”
print(message)
```

```
>>> Hello world!
>>> Hello Python!
```

Nombrar y utilizar variables

Cuando se utilizan variables se suele seguir algunas reglas que, al romperlas, puede causar errores o dificultades a la hora de leer y entender el código escrito:

a) El nombre de la variable solo debe contener letras, números y guiones bajos. Puede comenzar con un guión bajo pero no con un número (puede ser message_1, pero no 1_message);
b) No se permiten espacios;
c) No se utilizan palabras clave de Python o nombre de funciones;
d) Deben ser nombres cortos y descriptivos del valor que almacenan.
e) De ser posible, las variables se escriben en minúsculas.

Strings

Un **string** es una serie de caracteres, Todo aquello que se encuentre dentro de unas comillas, dobles o simples, es considerado un string en Python.

```
texto_1 = “Esto es un string”
texto_2 = ‘Esto también’
```

Esta flexibilidad permite que se utilizan comillas y apóstrofos dentro de un string:

```
texto_1 = ‘Y le dije “Messi es mejor que Cristiano” ’
texto_2 = ‘...Cristiano es “Mejor” que Messi …’

print(texto_1)
```

```
>>> Y le dije “Messi es mejor que Cristiano”
```

Métodos de strings

Una de las tareas más simples es cambiar la mayúscula al principio del texto en la variable

---

```
nombre_1 = ‘joaquin lopez” ’

print(nombre_1.title() )
```

```
>>> Joaquin lopez
```

El **método** title() aparece luego de la variable en el print. Un método es una acción que Python puede ejecutar en una pieza de información. El punto (.) luego del nombre_1 le dice a Python que aplique el método title() en nombre_1. Los métodos generalmente llevan paréntesis luego de ser utilizados porque necesitan información adicional la cual va dentro del paréntesis. En el caso de title() la función no necesita información adicional.

También se puede cambiar un sting para que sea todo mayuscula y todo minuscula:

```
print(nombre_1.upper())
print(nombre_1.lower())
```

```
>>> JOAQUIN LOPEZ
>>> joaquin lopez
```

El método lower() es particularmente útil para almacenar data ya que quita las mayúsculas del texto

Utilizar variables en strings

En muchas situaciones, se utilizaran valores de variables dentro de un string, para lograr esto se coloca una f antes de abrir comillas y se colocan llaves {} donde se incluirá el nombre de la variable. Estos strings se llaman f-strings donde la f significa formato

---

```
nombre = “joaquín”
apellido = “lopez”
nombre_completo = f”{nombre} {apellido}”
print(f“Buen dia {nombre_completo .title()}”)
```

```
>>> Joaquin Lopez
```

También se pueden almacenar f-strings en variables

```
saludos = f“Buen dia {nombre_completo .title()}”
```

Añadir espacios en blanco con tabulaciones

En programación, espacios en blanco se refiere a caracteres no imprimibles, como espacios, tabulaciones, etc. Permite generar salidas que sean fáciles de leer, para añadir una tabulación a tu texto, utiliza \t:

```
print(“\tPython”)
```

```
>>>     Python
```

Para añadir una nueva línea utiliza \n:

```
print(“Lenguajes:\n1 -Python\n2 - Francia\n3- C++”)
```

```
>>> Lenguajes
>>> 1 - Python
>>> 2 - Francia
>>> 3 - C++
```

Esto también se puede combinar:

```
print(“Lenguajes:\n1 -Python\n\t2 - Francia\n3- C++”)
```

```
>>> Lenguajes
>>> 1 - Python
>>>     2 - Francia
>>> 3 - C++
```

Eliminar espaciado

Los espaciados pueden generar problemas a la hora de leer strings en Python. “python” y “python “ son considerados string diferentes. Python puede buscar por espacios extra tanto a la izquierda como a la derecha de un string utilizando el método rstrip():

```
espacio_derecho = “python “
print(espacio_derecho.rstrip())
```

```
>>> ‘python’
```

Tambien se pued reutilizar lstrip para eliminar espaciado del lado izquierdo o strip si se quiere eliminar espaciado de ambos lados:

```
espacios = “ python “
print(espacios.lstrip())
print(espacios.lstrip())
```

```
>>>’python ‘
>>>’python‘
```

Quitar prefijos

Otra tarea común es remover prefijos. Cuando se trabaja con URL, un prefijo común es https://. Si se quiere remover esto para que se pueda utilizar la URL:

```
google_url= “https://google.com“
print(google_url.removeprefix("https://"))
```

```
>>> google.com
```

Se ingresa el nombre de la variable seguido de un punto seguido por el método removeprefix(). Dentro del método se ingresa el prefijo que se quiere remover del string original. Al igual que cuando se quitan los espacios extras, este método deja a la variable intacta por si se quiere utilizar su forma inicial

Números

Python trata los números de maneras distintas, dependiendo de cómo estén siendo utilizados. En primer lugar se miran a los integers ya que son los más simples para trabajar.

Integers

Los **integers** son valores donde se puede sumar (+), restar (-), multiplicar (*) y dividir (/). Cuando se escribe en la terminal python simplemente devuelve el resultado. A su vez python utiliza dos símbolos de multiplicación para representar exponentes y acompaña la orden de la operacion asi que se puede modificar el orden de la operación con paréntesis:

```
2 + 2
2 - 2
2 * 2
2 / 2
```

```
>>> 4
>>> 0
>>> 4
>>> 1
```

```
3 ** 2
2 + 3 * 4
(2 + 3) * 4
```

```
>>> 9
>>> 14
>>> 20
```

El espaciado no afecta en como Python evalúa las expresiones, solamente ayudan para la lectura de la misma.

Floats

Python llama a cualquier decimal **float**. Se llama float en casi todos los lenguajes para referirse a los valores en donde puede aparecer un punto decimal en cualquier posición del número. Hay que tener en cuenta que a veces se puede obtener un número arbitrario de decimales:

```
0.2 + 0.2
0.2 - 0.2
2 * 0.1
0.2 + 0.1
```

```
>>> 0.4
>>> 0.0
>>> 0.2
>>> 0.3000000000004
```

Python intenta representar los valores con la mayor exactitud posible lo que a veces puede ser difícil dado como los computadores tienen que representar los valores de forma interna.

Integers a Floats

Cuando se dividen 2 números, aunque estos sean integers, el resultado ser aún float:

```
4 / 2
```

```
>>> 2.0
```

Si estos se mezclan, el resultado también será un float:

```
1 + 2.0
```

```
>>> 3.0
```

Guiones bajos en números

Cuando se escriben números largos, se puede utilizar guinness bajos en lugar de puntos para hacer esto más legible. Cuando este se imprime, Python imprime solamente los dígitos:

```
numero_largo = 10_000_000_000_000_000
print(numero_largo)
```

```
>>> 10000000000000000
```

Python ignora los guiones bajos cuando se almacena este tipo de información, aunque este no se guarde en grupos de 3, el valor no se verá afectado.

Asignaciones múltiples

Se puede asignar valores a variables múltiples en una sola línea de código, ayudando a acortar la cantidad de código escrito y haciéndolo fácil de leer:

```
x, y, z = -1, 0, 1
```

Estos necesitan estar separados por comas, tanto las variables como los valores que se le otorgan a estas

Comentarios

Los comentarios son una parte fundamental de cualquier lenguaje de programación. A medida que los códigos se van tornando más largos y complicados, los comentarios facilitan la comprensión del mismo. Los comentarios son notas que describen lo que está sucediendo.

¿Cómo se escriben comentarios?

En Python,el numeral (#indica que es un comentario. Todo lo que se encuentre luego de un numeral es ignorado por el interpretado de Python:

```
# Tu nombre almacenado en una variable mas arriba
print(nombre_1.title() )
```

```
>>> Joaquin lopez
```

¿Qué tipo de comentarios deberías escribir?

La principal razón para escribir comentarios es explicar qué hace tu código y cómo funciona. Cuando estás trabajando en un proyecto, entiendes cada parte, pero con el tiempo puedes olvidar detalles. Los comentarios te ayudan a recordar tu enfoque sin tener que volver a analizar todo.

Si quieres ser un programador profesional o colaborar con otros, es fundamental escribir comentarios claros y útiles, ya que la mayoría del software se desarrolla en equipo. Los buenos programadores esperan ver comentarios que expliquen el código.

Cuando dudes si deberías comentar algo, pregúntate si te llevó pensar varias soluciones antes de llegar a la correcta. Si fue así, deja un comentario explicándolo. Siempre es más fácil eliminar comentarios innecesarios después que tener que agregarlos más tarde.

A partir de ahora, el autor usará comentarios en sus ejemplos para explicar mejor el código.

"The Zen of Python"

Es un conjunto de principios y filosofías que guían el diseño y la escritura de código en Python.

Fue escrito por Tim Peters y es una especie de “manifiesto” que describe cómo debería ser el código Python: claro, simple y elegante.

Puedes verlo directamente escribiendo en la consola de Python:

```
import this
```

Algunos de los principios más destacados son:

- Lo bello es mejor que lo feo;
- Lo explícito es mejor que lo implícito;
- La simplicidad es mejor que la complejidad;
- La legibilidad cuenta;
- En caso de ambigüedad, rechaza la tentación de adivinar.

Anexo Parte I: Bases - Sección I: Variables y tipos de datos simples

Variables

En Python, **una variable** es un nombre que se usa para almacenar un valor que puede cambiar o reutilizarse a lo largo del programa. Se define escribiendo el nombre de la variable, un signo igual (=) y el valor que quieres asignarle.

Reglas para nombrar variables:

- No puede empezar con un número.
- No puede contener espacios (se usa *guion bajo*).
- Solo puede contener letras, números y guiones bajos (_).
- No debe ser una palabra reservada de Python (como if, class, etc.).

Strings

Python puede manipular texto (representado por el tipo str, conocido como «cadenas de caracteres») al igual que números. Esto incluye caracteres «!», palabras «conejo», nombres «París», oraciones «¡Te tengo a la vista!», etc. «Yay! :)». Se pueden encerrar en comillas simples ('...') o comillas dobles ("...") con el mismo resultado

Para citar una cita, debemos «escapar» la cita procediéndose con \. Alternativamente, podemos usar el otro tipo de comillas:

‘dosen\’t’
“doesn’t”

>>> doesn’t
>>> doesn’t

Si no quieres que los caracteres precedidos por \ se interpreten como caracteres especiales, puedes usar cadenas sin formato agregando una **r** antes de la primera comilla:

---

print(r'C:\some\name')

>>> C:\some\name

Las cadenas se pueden concatenar (pegar juntas) con el operador + y se pueden repetir con *:

```
3 * 'a' + 'las'
```

```
>>> aaalas
```

Dos o más cadenas literales (es decir, las encerradas entre comillas) una al lado de la otra se concatenan automáticamente.

```
'a' 'las'
```

```
>>> alas
```

Si quieres concatenar variables o una variable y un literal, usa +:

```
variable_de_texto = “a”
variable_de_texto + 'las'
```

```
>>> alas
```

Métodos de strings

Método	Descripción	Ejemplo
lower()	Convierte el texto a minúsculas	"Hola".lower() ➡️ "hola"
upper()	Convierte el texto a mayúsculas	"hola".upper() ➡️ "HOLA"
capitalize()	Convierte la primera letra a mayúscula	"python".capitalize() ➡️ "Python"
title()	Convierte la primera letra de cada palabra a mayúscula	"hola mundo".title() ➡️ "Hola Mundo"
strip()	Elimina espacios en blanco al inicio y final	" hola ".strip() ➡️ "hola"
replace(a, b)	Reemplaza todas las apariciones de a por b	"hola mundo".replace("mundo", "Python") ➡️ "hola Python"
split(sep)	Divide el string en una lista usando el separador sep	"a,b,c".split(",") ➡️ ['a', 'b', 'c']
join(lista)	Une una lista de strings con un separador	",".join(['a', 'b', 'c']) ➡️ "a,b,c"
find(sub)	Devuelve la posición de la primera aparición de sub	"hola mundo".find("m") ➡️ 5
count(sub)	Cuenta cuántas veces aparece sub	"banana".count("a") ➡️ 3
startswith(sub)	Devuelve True si empieza con sub	"hola mundo".startswith("hola") ➡️ True
endswith(sub)	Devuelve True si termina con sub	"archivo.txt".endswith(".txt") ➡️ True
isalpha()	Devuelve True si solo contiene letras	"hola".isalpha() ➡️ True
isdigit()	Devuelve True si solo contiene dígitos	"123".isdigit() ➡️ True
islower()	True si todas las letras están en minúsculas	"hola".islower() ➡️ True
isupper()	True si todas las letras están en mayúsculas	"HOLA".isupper() ➡️ True
len()	(No es método, es función) Devuelve la longitud del string	len("hola") ➡️ 4

Números

El intérprete funciona como una simple calculadora: puedes introducir una expresión en él y éste escribirá los valores. La sintaxis es sencilla: los operadores +, -, * y / se pueden usar para realizar operaciones aritméticas; los paréntesis (()) pueden ser usados para agrupar.

Los números enteros (ej. 2, 4, 20) tienen tipo int, los que tienen una parte fraccionaria (por ejemplo 5.0, 1.6) tiene el tipo float. Vamos a ver más acerca de los tipos numéricos más adelante en el tutorial.

En el modo interactivo, la última expresión impresa se asigna a la variable _. Esto significa que cuando se está utilizando Python como calculadora, es más fácil seguir calculando.

```
tax = 12.5 / 100
price = 100.50
price * tax
price + _
round(_, 2)
```

```
>>> 12.5625
>>> 113.0625
>>> 113.06
```

Esta variable debe ser tratada como de sólo lectura por el usuario. No le asignes explícitamente un valor; crearás una variable local independiente con el mismo nombre enmascarando la variable con el comportamiento mágico.

Además de int y float, Python admite otros tipos de números, como Decimal y Fracción. Python también tiene soporte incorporado para complejos números, y usa el sufijo j o J para indicar la parte imaginaria (por ejemplo, 3+5j).

Métodos de integers y floats

Función / Método	Descripción	Ejemplo
int(x)	Convierte un valor a entero (si es posible)	int(5.9) ➡️ 5
float(x)	Convierte un valor a decimal	float(5) ➡️ 5.0
abs(x)	Valor absoluto de un número	abs(-4) ➡️ 4
pow(x, y) o x**y	Eleva un número a la potencia de otro	pow(2, 3) ➡️ 8 / 2**3 ➡️ 8
round(x, n)	Redondea el número x a n decimales	round(3.14159, 2) ➡️ 3.14
divmod(a, b)	Devuelve una tupla (cociente, resto) de la división entera	divmod(10, 3) ➡️ (3, 1)
max(a, b, ...)	Devuelve el mayor valor	max(1, 4, 2) ➡️ 4
min(a, b, ...)	Devuelve el menor valor	min(1, 4, 2) ➡️ 1
sum(lista)	Suma todos los elementos de una lista	sum([1, 2, 3]) ➡️ 6
math.floor(x)	Redondea hacia abajo (requiere import math)	math.floor(3.7) ➡️ 3
math.ceil(x)	Redondea hacia arriba (requiere import math)	math.ceil(3.1) ➡️ 4
math.sqrt(x)	Raíz cuadrada (requiere import math)	math.sqrt(16) ➡️ 4.0
is_integer() (método)	Para float, devuelve True si el valor es un número entero	(3.0).is_integer() ➡️ True

Bibliografía:

Matthes, E. (2015). *Python crash course: A hands-on, project-based introduction to programming*. No Starch Press. ISBN 978-1-59327-603-4

Python Software Foundation. (2024). *The Python Tutorial*. Retrieved from https://docs.python.org/3/tutorial/

Introducción a las listas

¿Que es una lista?

Una lista es una colección de items en un orden particular. Se puede hacer una lista que incluya las letras del alfabeto o de números del 0 al 9. Una lista está indicada por corchetes [] y los elementos individuales están separados por comas (es una buena idea nombrar a la lista segun la informacion que contiene):

```
autos = [“nissan”,”chevrolet”,”ford”]
print(autos)
```

```
>>> [“nissan”,”chevrolet”,”ford”]
```

Al imprimir, Python devuelve la representación de la lista incluyendo los corchetes.

Acceder a los elementos de la lista

Las listas tienen un orden, así se puede acceder a los elementos que están en ellas diciéndole a Python la posición, o índice, del dato que necesitamos. Para acceder a los elementos de las listas hay que incluir la posición del dato entre corchetes comenzando desde el 0:

```
autos = [“nissan”,”chevrolet”,”ford”]
print(autos[0])
```

```
>>> [“nissan”]
```

Al preguntar por un solo dato, Python devolverá ese dato sin los corchetes. A esto se le puede aplicar los métodos de la sección I:

```
print(autos[0].title())
```

```
>>> Nissan
```

Posiciones de índice

Las posiciones comienzan desde el 0 en adelante. Esta es una característica de la mayoría de los lenguajes de programación y se debe a como las listas están indexadas basadas en cero (zero-based indexing)

Python tiene una sintaxis especial para acceder al último elementos de una lista, que es utilizando el -1

```
print(autos[-1].upper())
```

```
>>> FORD
```

Usar valores individuales de una lista

Se pueden utilizar valores individuales de una lista justo como cualquier otra variable. Por ejemplo en un f-string para crear un mensaje basado en una lista.

```
message = f”Mi primer auto fue un {autos[1].title()}.”
print(message)
```

```
>>> Mi primer auto fue un Chevrolet
```

Modificar, añadir y quitar datos

La mayoría de las listas que se crean son dinámicas, esto significa que luego de crearlas podrás añadir y quitar datos de la lista a medida que tu código se vaya ejecutando.

Modificar elementos de una lista

La sintaxis para modificar datos es similar a la sintaxis para acceder a ellos. Para esto se utiliza el nombre de la lista seguido por el índice del dato que se quiere modificar, y luego proveer un nuevo valor para ese dato:

```
autos_modificados = autos
autos_modificados [0] = “renault”
print(autos)
```

```
>>> [“renault”,”chevrolet”,”ford”]
```

¿Por qué los índices empiezan en 0?
Esto se debe a cómo los lenguajes como C (en el que Python está escrito) manejan la memoria. En términos de punteros, la dirección base de un array es la del primer elemento. Si el índice empezará en 1, habría que hacer un desplazamiento adicional en cada acceso, lo cual sería menos eficiente.

Por eso, la indexación basada en cero simplifica la aritmética de punteros y hace que los cálculos de posición sean más rápidos y consistentes.

Anteriormente este dato era “nissan” pero ahora es “renault” demostrando que la posición 0 de la lista fue modificada.

Añadir elementos a la lista

Existen muchos motivos por los cuales se querrán añadir métodos a una lista. Para esto Python posee varias maneras para lograr esto.

Añadir un elemento al final de la lista

La manera más simple de añadir un elemento es con el método append() el cual agrega un nuevo elemento al final de la lista.

```
agregar_auto = autos
agregar_auto.append(“ferrari”)
print(autos)
```

```
>>> [“renault”,”chevrolet”,”ford”,“ferrari”]
```

El método append hace fácil añadir elementos a las listas haciendo de estas listas más dinámicas. Por ejemplo, se puede iniciar con una lista vacía e ir añadiendo elementos a medida que el código se va ejecutando:

```
lista_de_autos_vacia = []

lista_de_autos_vacia.append(“ford”)
lista_de_autos_vacia .append(“ferrari”)

print(lista_de_autos_vacia )
```

```
>>> [”ford”,“ferrari”]
```

Esta manera de escribir listas es muy común ya que generalmente no se sabe la información que se cargará en el programa hasta después de que el programa esté corriendo, para eso se crea una lista vacía que irá reteniendo los datos.

Insertar datos en una lista

Se pueden añadir elementos en cualquier posición con el método insert(). Con este método se puede especificar en donde se busca incluir un dato en la lista:

```
autos.insert(0, “toyota”)
print(autos)
```

```
>>> [“toyota”,“renault”,”chevrolet”,”ford”]
```

Este método desplaza a los demás elementos una posición a la derecha desde el lugar en el cual se agregó.

Quitar datos de una lista

De igual manera que se buscan agregar datos, se busca quitar datos ya sea porque no se necesitan o quedan obsoletos. Python provee varias maneras de hacerlo:

Quitar un elemento con del

Se se sabe la posición del dato que se quiere quitar, se puede utilizar del statement:

```
del autos[0]
print(autos)
```

```
>>> [“renault”,”chevrolet”,”ford”,”ferrari”]
```

Ya que anteriormente añadimos “toyota” a la lista, ahora lo podemos quitar utilizando este método. Cambiando la posición que se le dio a del se quitara otro elemento de la lista, por ejemplo:

```
del autos[-1]
print(autos)
```

```
>>> [“renault”,”chevrolet”,”ford”]
```

Quitar un elemento utilizando el método pop()

En algunas ocasiones se buscará utilizar algún valor de la lista luego de quitarlo. El método pop() permite hacer esto. El término pop proviene de pesar en la lista como una pila de datos en donde se está quitando el dato de arriba.

```
popped_autos = autos.pop()
print(popped_autos)
```

```
>>> ford
```

Quitar un elemento utilizando el método pop() de cualquier lugar en la lista

Se puede utilizar el método pop() para quitar elementos en cualquier posición en la lista solo con incluir el índice del dato entre los paréntesis del metoodo:

```
mi_primer_auto = autos.pop(1)
print(f"Mi primer auto fue un {mi_primer_auto.title()}.")
```

```
>>> Mi primer auto fue un Chevrolet
```

Cabe recordar que cada vez que se utilice pop(), el elemento con el que se trabaja no se encontrará más en la lista.

Quitar un dato según el valor

Algunas veces no sabrás la posición del valor que se quiere quitar. Si solamente sabes el valor del elemento, puedes utilizar el método remove().

```
removed_autos = autos.remove("renault")
print(removed_autos)
```

```
>>> None
```

Organizar una lista

El hecho de ordenar o no una lista dependerá de para que será utilizada esta. Python provee de varias maneras distintas de organizar una lista, según cual sea el fin.

Organizar utilizando el método sort()

Es una manera sencilla de organizar una lista. Este método cambia el orden de la lista de manera permanente en orden alfabético, así que no podremos obtener en la lista original en caso de que la necesitemos.

El método remove() elimina solo la primera aparición del valor que especifiques. Si existe la posibilidad de que el valor aparezca más de una vez en la lista, necesitarás usar un bucle para asegurarte de que se eliminen todas las ocurrencias del valor.

```
autos.sort()
print(autos)
```

```
>>> [”chevrolet”,”ford”,“renault”]
```

Si se quiere ordenar alfabéticamente pero al revés se puede utilizar el argumento reverse=True:

```
autos.sort(reverse=True)
print(autos)
```

```
>>> [“renault”,”ford”,”chevrolet”]
```

Organizar una lista de forma temporal utilizando sorted()

Para mantener el orden de la lista original se utiliza el método sorted(), la cual te deja orden la lista en un orden particular soon cambiar el orden original:

```
print(sorted(autos))
print(autos)
```

```
>>> [”chevrolet”,”ford”,“renault”]
>>> [“renault”,”ford”,”chevrolet”]
```

Invertir una lista

Si se desea invertir los valores dentro de una lista se puede utilizar el método reverse():

```
autos.reverse()
print(autos)
```

```
>>> [”chevrolet”,”ford”,“renault”]
```

Este método no ordena de manera alfabética, cambiando el orden de manera permanente, aunque se puede utilizar el mismo método de nuevo y obtener el valor original.

Ordenar una lista alfabéticamente es un poco más complicado cuando no todos los valores están en minúsculas. Existen varias formas de interpretar las letras mayúsculas al determinar un orden de clasificación, y especificar el orden exacto puede ser más complejo de lo que queremos abordar en este momento. Sin embargo, la mayoría de los métodos de ordenación se basarán directamente en lo que aprendiste en esta sección.

Largo de listas

Se puede saber la longitud (por longitud hablamos de la cantidad de datos) de las listas utilizando la función len().

```
print(len(autos))
```

```
>>> 3
```

Python cuenta los datos comenzando por 1 ya que así se evitan errores de faltantes cuando se revisa los largos de la lista.

Evitar errores de indexado en las listas

Hay un tipo de error que es común ver cuando trabajas con listas por primera vez.

```
print(autos[3])
```

```
>>> Traceback (most recent call last):
      File "xxxxxxxxxx.py", line x, in
            print(motorcycles[3])
                  ~~~~~~~~~^^^
    IndexError: list index out of range
```

Python intenta darte el elemento en el índice 3. Pero cuando busca en la lista, no encuentra ningún elemento en motorcycles con un índice de 3. Debido a la naturaleza de la indexación basada en cero, este error es común. Las personas suelen pensar que el tercer elemento tiene el número 3, porque comienzan a contar desde 1. Sin embargo, en Python, el tercer elemento tiene el índice 2, ya que la indexación comienza en 0.

Un IndexError significa que Python no puede encontrar un elemento en el índice que solicitaste. Si este error ocurre en tu programa, intenta ajustar el índice restando uno y vuelve a ejecutar el programa para ver si los resultados son correctos.

Anexo Parte I: Bases - Sección II: Introducción a las listas

Python tiene varios tipos de datos compuestos, utilizados para agrupar otros valores. El más versátil es la lista, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo.

```
squares = [1, 4, 9, 16, 25]
print(squares)
```

```
>>> [1, 4, 9, 16, 25]
```

Al igual que las cadenas (y todas las demás tipos integrados sequence), las listas se pueden indexar y segmentar:

```
print(squares[-3:]) # La segmentación devuelve una nueva lista
```

```
>>> [9, 16, 25]
```

Las listas también admiten operaciones como concatenación:

```
concatenado = squares + [36, 49, 64, 81, 100]
print(concatenado )
```

```
>>> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Todas las operaciones de rebanado retornan una nueva lista que contiene los elementos pedidos. Esto significa que la siguiente rebanada retorna una shallow copy de la lista:

```
squares + [36, 49, 64, 81, 100]
squares = squares [:]
```

```
>>> [9, 16, 25]
```

- Una copia superficial (shallow copy) construye un nuevo objeto compuesto y luego (en la medida de lo posible) inserta una referencias en él a los objetos encontrados en el original.
  Una copia profunda (deep copy) construye un nuevo objeto compuesto y luego, recursivamente, inserta copias en él de los objetos encontrados en el original.

Es posible anidar listas (crear listas que contengan otras listas), por ejemplo:

```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]
print(x)
print(x[0])
print(x[0][1])
```

```
>>> [['a', 'b', 'c'], [1, 2, 3]]
>>> ['a', 'b', 'c']
>>> 'b'
```

Métodos de listas

Método	Descripción	Ejemplo
append(x)	Agrega un elemento al final de la lista.	lista.append(5)
extend(iterable)	Agrega múltiples elementos al final de la lista.	lista.extend([6, 7, 8])
insert(i, x)	Inserta un elemento en una posición específica.	lista.insert(1, 10)
remove(x)	Elimina la primera aparición de un elemento.	lista.remove(5)
pop([i])	Elimina y devuelve el elemento en la posición i. Si no se especifica, elimina el último.	lista.pop(2)
index(x, [start, end])	Devuelve la posición del primer elemento con el valor x.	lista.index(5)
count(x)	Devuelve la cantidad de veces que x aparece en la lista.	lista.count(5)
sort([key, reverse])	Ordena la lista en orden ascendente (o descendente si reverse=True).	lista.sort(reverse=True)
reverse()	Invierte el orden de los elementos de la lista.	lista.reverse()
copy()	Devuelve una copia superficial de la lista.	nueva_lista = lista.copy()
clear()	Elimina todos los elementos de la lista.	lista.clear()

Bibliografía:

Matthes, E. (2015). *Python crash course: A hands-on, project-based introduction to programming*. No Starch Press. ISBN 978-1-59327-603-4

Python Software Foundation. (2024). *The Python Tutorial*. Retrieved from https://docs.python.org/3/tutorial/

Python Cheat Sheet. (s.f.). *Lists and Tuples*. Recuperado el [fecha de acceso], de https://www.pythoncheatsheet.org/cheatsheet/lists-and-tuples